NUMBER SYSTEMS

and

DATA REPRESENTATION

for

COMPUTERS

05 March 2008

Table of Contents

To the Midshipmen and Cadets
of the United States of America,
and their instructors.

Prologue

This tutorial is an introduction to number systems and representation of data for computers. The original work by Mr. Bob Ralph of Fayetteville Technical Community College served soldiers at Ft. Bragg and airmen at Pope AFB. Drew McCall III motivated the first expansion of this work (originally 13 pages) through his 2002 High School Science Fair project on BCD arithmetic and prime numbers.

It is important how we represent our data, whether it is a message in words, sound, an image, color, smell, numbers, or something else. The representation we choose affects our ability to store and manipulate ideas efficiently with an acceptably accurate approximation. The representation we choose can be a catalyst for new interpretation and discovery.

The primary context for this tutorial is the binary digital computer. Some discussion is included to spark the imagination for those who may want to go beyond this immediate context.

A binary digital computer must represent all data internally as patterns of zeros and ones. Any data that has only two values of interest can be represented by a single binary digit. Examples:

Table 1. Representing Two States as a Binary Number

Numbers: {0, 1}
Sign of a Number: {Plus, Minus}
Sex: {Male, Female}
Directions: {Right, Left}, {Up, Down}, {In, Out}
Decision, Logic: {Yes, No}, {True, False}
Switch Position: {On, Off}
Voltages in a Computer: {+5 V DC, 0 V DC}, {High, Low}
CD-ROM {Pit, Land} Pit = low intensity reflection = 0, Land = high intensity reflection = 1
Spin Direction: {Clockwise, Counterclockwise}
Magnetic Domain Orientation {North-South, South-North}

The choice of representation on a particular computer is a shared responsibility between the hardware engineer and the programmer. Standards have been established to make data files and programs portable, and results of computations predictable. This tutorial presents basic concepts at an applied level.

People in Information Science and Computer Engineering need to know about data representation to understand compatibility issues of data from different sources. This is necessary both to know what to expect on a particular machine for performance, and also to know what to expect when importing or exporting data.

Most serious data processing is done by business on IBM computers which use EBCDIC format for character data. Other computers usually use ASCII, which is not directly compatible with EBCDIC. ASCII has been extended to accommodate other European languages. UNICODE is being defined to accommodate many additional character sets. The most important goal was to represent languages such as Japanese, Korean, Chinese, Hebrew, Greek, Arabic, and symbols used in mathematics, science, and engineering. Ancient languages and localized languages are now also being added to UNICODE.

Differences exist in number formats between processors. This is most dramatically seen in the variety of floating point formats. IBM pioneered the Hexadecimal Floating Point (HFP) format with Excess notation. Non-IBM processors adopted a simpler Binary Floating Point (BFP) format. As memory costs decreased and processor speeds increased, the innovation of Hexadecimal Floating Point format became less important. The lower cost of non-IBM workstations for scientific and engineering work led to standardization efforts by the Institute of Electrical and Electronics Engineers (IEEE). To remain competitive, IBM has implemented both the Hexadecimal and the IEEE Binary Floating Point formats on its IBM System/390.

Not only are data formats different, the order of storage of multi-byte data is different. This is a function of how a processor is designed to fetch a block of data from memory most efficiently. The two primary methods are called *Big Endian* and *Little Endian*. Machines with Big Endian (Big End In) architecture store the Most Significant Byte (MSB) in the lowest address in memory. Little Endian (Little End In) machines store the Least Significant Byte (LSB) in the lowest address in memory. There are other methods. One is called *Middle Endian*. This affects importing and exporting data between computers, protocol numbers and port numbers on the Internet, design of display device interfaces and drivers, character code representation standards, and storage formats for integers and floating point numbers. For data, this affects operations of comparisons, selection, searching, sorting, and other operations. When you grow from a small to a medium size company and switch to an IBM mainframe, these are issues that even the non-technical people need to know exist so that conversion efforts can be planned, scheduled, funded, and staffed to make the conversion go smoothly.

We have long had to live with considering the impact of working with different units of measure and systems of units. Anyone who has spent time in a kitchen knows this. Working with a variety of units of measure has also been the bane of scientists, engineers, and technicians. The surveyor and the sailor have historically had to cope with a variety of units of measure in practical life just to do a basic job to survive. (It is actually a conspiracy just to make life hard for students, but do not let the secret out.)

The different ways of measuring an angle illustrate that we already are used to representing the same data in different ways. Measurement of angles is important to sailors, explorers, space flight, surveyors, artillerymen, engineers, and students.

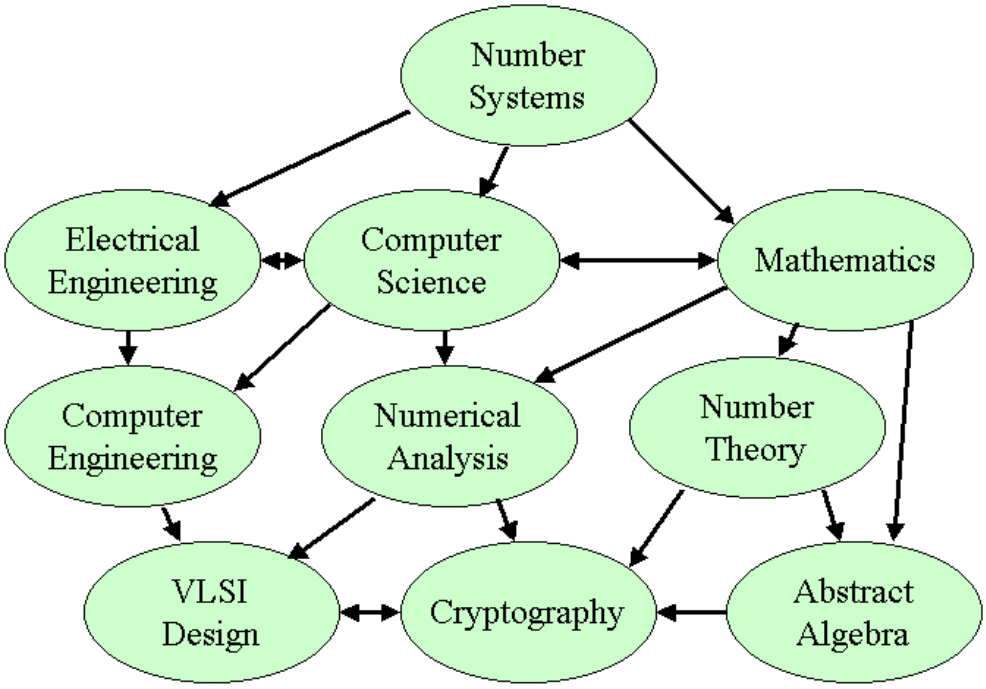
Table 2. Angle Measurements in Binary and Other Systems

Points of a Compass	Direction	Degrees = 360 * Binary Fraction	Binary Fraction	Grads	mil U.S. Army in WW II	mil NATO	Target Shooting (1000 * radians)
0	N	0, 360	0.00000	0	0	0	0
1		11.25	0.00001	12.5	125	200	62.5π
2	NNE	22.5	0.00010	25	250	400	125π
3		33.75	0.00011	37.5	375	600	187.5π
4	NE	45	0.00100	50	500	800	250π
5		56.25	0.00101	62.5	625	1000	312.5π
6	ENE	67.5	0.00110	75	750	1200	375π
7		78.75	0.00111	87.5	875	1400	437.5π
8	E	90	0.01000	100	1000	1600	500π
9		101.25	0.01001	112.5	1125	1800	562.5π
10	ESE	112.5	0.01010	125	1250	2000	625π
11		123.75	0.01011	137.5	1375	2255	687.5π
12	SE	135	0.01100	150	1500	2400	750π
13		146.25	0.01101	162.5	1625	2600	812.5π
14	SSE	157.5	0.01110	175	1750	2800	875π
15		168.75	0.01111	187.5	1875	3000	937.5π
16	S	180	0.10000	200	2000	3200	1000π
17		191.25	0.10001	212.5	2125	3400	1062.5π
18	SSW	202.5	0.10010	225	2250	3600	1125π
19		213.75	0.10011	237.5	2375	3800	1187.5π
20	SW	225	0.10100	250	2500	4000	1250π
21		236.25	0.10101	262.5	2625	4200	1312.5π
22	WSW	247.5	0.10110	275	2750	4400	1375π
23		256.75	0.10111	287.5	2875	4600	1437.5
24	W	270	0.11000	300	3000	4800	1500π
25		281.25	0.11001	312.5	3125	5000	1562.5π
26	WNW	292.5	0.11010	325	3250	5200	1625π
27		303.75	0.11011	337.5	3375	5400	1687.5π
28	NW	215	0.11100	350	3500	5600	1750
29		226.25	0.11101	362.5	6325	5800	1812.5π
30	NNW	237.5	0.11110	375	3750	6000	1875π
31		248.75	0.11111	387.5	3875	6200	1937.5π
32	N	360	1.00000	400	4000	6400	2000π

The method of representing numerical data affects the speed, accuracy, and cost of computation and storage. Communication software (including networking), system software, business applications, embedded computers, numerically controlled machinery, interactive video simulations, and scientific and engineering work have very different requirements. Numerical computations in science and engineering are much more

demanding than in other fields. Consequently, special hardware features have been designed to make these computations faster, at additional cost of the hardware. By reducing the time required for doing computations, special hardware reduces the overall cost of running these programs and makes software responsive enough to accomplish tasks that could not be done in years past. The Institute of Electrical and Electronics Engineers (IEEE) has defined standards for representing numerical data and computations. These standards promote portability and predictability of computations between different computers and computer languages.

If you find the concepts in this tutorial fascinating, your options for future study could lead you to computer science, computer engineering, electrical engineering, numerical analysis, number theory, abstract algebra, and cryptology. An article by Dunne that includes history of numbers and aids for calculation can be found on the Internet.[1]



For the serious computer science major, an extraordinary reference is Donald E. Knuth's Volume 2, "Seminumerical Algorithms" of *The Art of Computer Programming.*[2]

[1] Paul E. Dunne, "Mechanical Aids to Computation and the Development of Algorithms", http://www.csc.liv.ac.uk/~ped/teachadmin/histsci/htmlform/lect2.html visited 31 December 2007

[2] Donald E. Knuth, "Seminumerical Algorithms", Vol. 2 of *The Art of Computer Programming*, Addison Wesley (1971).

Number System Bases Introduction

There are several definitions for the base of a number system. The one most important for students starting the journey of learning about computers is the first one.

1. **The number of symbols available to use for representing a quantity is called the base or radix of a number system.**

Table 3. Common Number System Examples

Name	Base	Symbols
Decimal	**10**	**{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}**
Binary	**2**	**{0, 1}**
Octal	**8**	**{0, 1, 2, 3, 4, 5, 6, 7}**
Hexadecimal	**16**	**{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}**

In the world of computers, it is common to use bases 2, 8, 10, and 16.

$\ldots + (c_4 \times b^4) + (c_3 \times b^3) + (c_2 \times b^2) + (c_1 \times b^1) + (c_0 \times b^0) + (c_{-1} \times b^{-1}) + (c_{-2} \times b^{-2}) + \ldots$

Because the value of b is the same in each term above, we often omit it in the notation and merely write

$$\ldots c_4 c_3 c_2 c_1 c_0 . c_{-1} c_{-2} \ldots$$

More will be said about this later under the section of positional number systems.

2. In a **function series number system**, numbers are represented by the sum of coefficients c_k times the k^{th} function belonging to a set of functions. The "base" of such a number system identifies the function. Numbers are computed in the form of

$\ldots + (c_4 \times f_4) + (c_3 \times f_3) + (c_2 \times f_2) + (c_1 \times f_1) + (c_0 \times f_0) + (c_{-1} \times f_1) + (c_{-2} \times f_2) + \ldots$

Examples of such number systems are the Fibonacci base number system, and the Phi base number system.

3. A **power series number system** is a special case of a function series number system, where the function is some constant b raised to the power k. Numbers are represented by the sum of coefficients c_k times a constant b raised to the power k. The constant b is the "base" of the number system. The constant b is not necessarily an integer.

$\ldots + (c_4 \times b^4) + (c_3 \times b^3) + (c_2 \times b^2) + (c_1 \times b^1) + (c_0 \times b^0) + (c_{-1} \times b^{-1}) + (c_{-2} \times b^{-2}) + \ldots$

The first number system above is a special case of this number system, where b is an integer.

Base 60 Number System

You have used several number systems. You most often use base 10. Our systems for time and navigation are derived from the base 60 number system. The number 60 is divisible by 2, 3, 4, 5, 6, 10, 12, 15, 30, and 60.We have 60 minutes per hour, 60 seconds per minute of time, approximately 360 days per year (and exactly 360 days per banker year), 60 minutes of an arc per degree, 60 seconds of an arc per minute of an arc.

When we write base 60 numbers, we write the coefficient using our decimal numbers, followed by a symbol for the place value. For example, we write 127° 18' 23" to record 127 degrees, 18 minutes, and 23 seconds. The small circle for degrees, the apostrophe for minutes, and the quote symbol for seconds, perform the function of identifying place value in base 60 arithmetic. We are limited by not having 60 different number symbols.

In the Babylonian system of weights and coinage, 1 talent = 60 minae, 1 minae = 60 shekels.[3] The Oxford English Dictionary gives a different account, stating that it was the Greek system that used 1 *talent* = 60 minae.[4] There 60 drops to a teaspoon in liquid measure used in cooking.[5] More history on the introduction of the sexagesimal (base 60) system by Babylon, 1900 BC – 1800 BC, can be found on the Internet.[6]

The Chinese calendar names years in a manner that produces a 60 year cycle, called a "Sexagenary cycle". The following is Table 6.1.1 from *Explanatory Supplement to the Astronomical Almanac*.[7] For each new year, advance from the old year by one Celestial Stem and by one Earthly Branch.

Table 4. Chinese Sexagenary Cycle of Days and Years

Celestial Stem	Earthly Branch
1. jia	1. zi (rat)
2. yi	2. chou (ox)
3. bing	3. yin (tiger)
4. ding	4. mao (hare)
5. wu	5. chen (dragon)
6. ji	6. si (snake)
7. geng	7. wu (horse)
8. xin	8. wei (sheep)
9. ren	9. shen (monkey)
10. gui	10. you (fowl)
	11. xu (dog)
	12. hai (pig)

[3] Karl Menninger, *Number Words and Number Symbols: A Cultural History of Numbers*, page 162, Dover Publications (1992).
[4] *Oxford English Dictionary*, Second Edition (1989), CD-ROM Version 3.00, Oxford University Press (2002).
[5] Irma S. Rombauer and Marion Rombauer Becker, *Joy of Cooking*, page 546, Bobbs-Merrill Company (1964).
[6] J J O'Connor and E F Robertson, "Babylonian numerals", School of Mathematics and Statistics, University of St. Andrews, Scotland (December 2000). http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/Babylonian_numerals.html, visited 17 December 2007.
[7] *Explanatory Supplement to the Astronomical Almanac*, P. Kenneth Seidelmann, editor, University Science Books, Sausalito, CA 94965, as quoted by L. E. Doggett, "Calendars", http://astro.nmsu.edu/~lhuber/leaphist.html, 12 April 2002.

Table 5. Chinese Year Names

Year Names					
1. jia-zi	2. yi-chou	3. bing-yin	4. ding-mao	5. wu-chen	6. ji-si
7. geng-wu	8. xin-wei	9. ren-shen	10. gui-you	11. jia-xu	12. yi-hai
13. bing-zi	14. ding-chou	15. wu-yin	16. ji-mao	17. geng-chen	18. xin-si
19. ren-wu	20. gui-wei	21. jia-shen	22. yi-you	23. bing-xu	24. ding-hai
25. wu-zi	26. ji-chou	27. geng-yin	28. xin-mao	29. ren-chen	30. gui-si
31. jia-wu	32. yi-wei	33. bing-shen	34. ding-you	35. wu-xu	36. ji-hai
37. geng-zi	38. xin-chou	39. ren-yin	40. gui-mao	41. jia-chen	42. yi-si
43. bing-wu	44. ding-wei	45. wu-shen	46. ji-you	47. geng-xu	48. xin-hai
49. ren-zi	50. gui-chou	51. jia-yin	52. yi-mao	53. bing-chen	54. ding-si
55. wu-wu	56. ji-wei	57. geng-shen	58. xin-you	59. ren-xu	60. gui-hai

Here are two pages from a contemporary Japanese calendar.

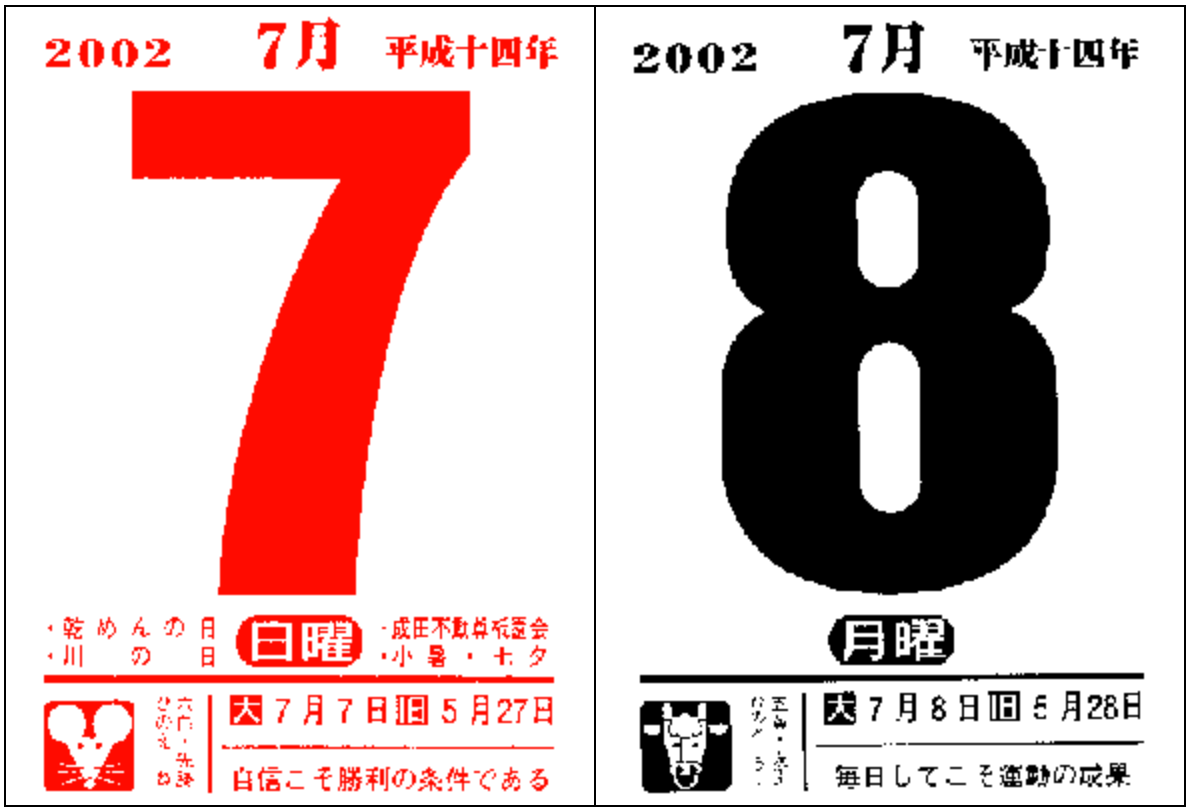


Figure 1. Japanese Calendar Example Pages

Base 12 Number System

You have also used the base 12 (duodecimal) system. There are 12 hours of day time and 12 hours of night time. There are 12 months in a year. There are 12 inches per foot. There are 72 printer's big points per inch, which gives us 6 lines per inch with type that is 12 big points tall. There are 12 items in a dozen, and 144 (12^2) in a gross. There are 12 tribes of Israel, and 144 thousand servants of God from the tribes of Israel (Revelation 7:3 - 4) sealed on their foreheads.

Most people have 4 fingers and 1 thumb on each hand. Each finger has 3 knuckles and three bones. If you use your thumb as a pointer, you can count to 12 on each hand by pointing either to knuckles or bones. With two hands, you can count to 24, the number of hours per day.

The number 12 is evenly divisible by 2, 3, 4, 6, and 12. Common fractions are exactly represented in base 12.

In Western music of 2007 A.D., the "equal temperament" scale has 12 divisions (called "half-steps") per octave (doubling of frequency). These half-steps are equally spaced logarithmically. In this system, the frequency multipliers within an octave of some reference frequency *A* takes the form of $(A \text{ x } 2^{\frac{n}{12}})$ where $0 \le n \le 11$. The symbols assigned to the different notes within an octave are {*A, A#, B, C, C#, D, D#, E, F, F#, G, G#* }. The reference frequency for *A* is 440 Hertz (abbreviated by "Hz"). The vertical position of a note on a music score establishes the frequency value of the note.

Brilliant musicians understood the goal of assigning notes before the mathematical concepts caught up. Consequently, the musicians approximated the exact intervals by simpler rational numbers, from which terminology for notes was established.

Table 6. Common Fractions in Base 12

Fraction	Base 12 Representation
1/2	0.6
1/3	0.4
1/4	0.3
1/6	0.2
1/8	0.12
1/12	0.1

Base 6 (Senary) Number System

Dice are cubes labeled with dots on each face. Dice are usually considered to be random number generators, assuming that each face has an equal probability of landing with any face pointing up. Of course, in space, you have to carefully define "up".

The dots, usually considered to represent 1 through 6, can be used to represent 0 through 5. If dice are thrown one at a time, and results recorded in the order the dice are thrown, the result is a random number recorded in base 6.

Base 3 (Trinary) Number System

Kepler used a base 3 number system when doing his investigations.[8]

Base 3 has the property in a finite length word to have a unique representation for the number zero. For fixed finite length arithmetic representing negative numbers in complement arithmetic, the number zero has a unique representation in base 3, but not in base 2. We also have the curiosity that $2 + 2 = 10_3$.

International Morse Code is base 3. One symbol is for dot, one for dash, and one for space. The space is a required symbol since the code is a variable length code. There has to be a way to determine when one symbol ends and another one begins. This is done by the time between characters.

Table 7. Alphabet in International Morse Code

A	B	C	D	E	F	G	H	I	J	K	L	M
01	1000	1010	100	0	0010	110	0000	00	0111	101	0100	11

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
10	111	0110	1101	010	000	1	001	0001	011	1001	1011	1100

Let the space between symbols in Morse Code be represented by the number 2. Then the message, "International Morse Code" is given by "00210212020102102012120021112102012010022112111201020002022101021112100 20". Morse Code was designed for fast transmission using American English. The most frequently used letters take the shortest time to send. The 0 represents a very short transmission, called a "dit". The 1 represents a longer transmission, called a "dah".

Base 3 is used in the construction of the "Cantor set".[9] Begin with the closed unit interval [0,1]. By "closed", we mean that the end points of the interval are included in the interval. The Cantor set is constructed by iteratively partitioning each closed interval into three equal pieces and removing the middle piece, leaving the two end pieces as closed sets. The end points of each interval can be written as fractions using base 3. The first 4 intervals are shown below.

[8] J.R.Newman, *The World of Mathematics*, volume 1., cited by Alexander Bogomolny, "Addition and Multiplication Tables in Various Bases" from Interactive Mathematics Miscellany and Puzzles (2002). http://www.cut-the-knot.org/blue/SysTable.shtml visited 04 June 2002.

[9] Adrian Ocneanu, Construction of the Cantor Set, Math 401 Class Notes, The Pennsylvania State University (06 October 1986).

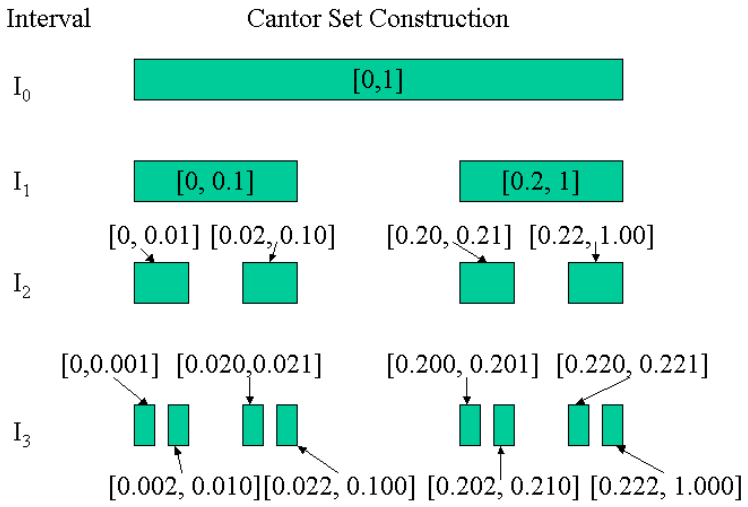


Figure 2. Construction of the Cantor Set

There is a countably infinite number of these interval sets.

The Cantor Set is the intersection of all of these intervals: $C = \bigcap_{n=1}^{\infty} I_n$.

Aside: Observing that when the digit "1" appears, it only occurs as the right-most non-zero digit. This allows substitution of "1" with "02" where "2" repeats infinitely to the right. This transforms the base 3 representation to a base 2 representation. You will see this trick used in Math class when studying the properties of Real numbers.

Base 1 Number System

Yes! There is a base 1 number system. A base 1 system uses only one symbol. You can choose any symbol.

Suppose you choose a pebble as your symbol. A shepherd can count the number of sheep entering the coral at night by putting a pebble in a leather bag for each sheep as it enters the coral. The shepherd can determine if any sheep are missing in the morning as they depart the coral by removing a pebble from the leather bag for each sheep that exits. The number of missing sheep is the number of pebbles remaining in the leather bag after all the sheep have departed.

For a more enduring record, you can cut a line into a piece of wood or even a flattened piece of bark rather than put a pebble into a bag. For example,

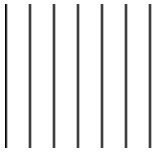


Figure 3. Base One Number System Example

in base 1 is equal to 7 in base 10. If you want a copy of the record, slice the bark in half.

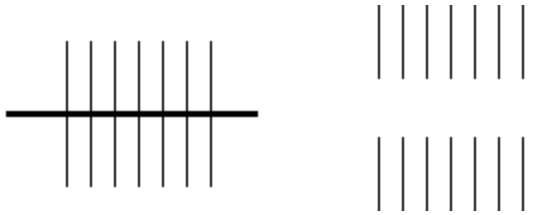


Figure 4. Duplicating a Number in Base One

The symbols recording a number written in base 1 are commutative. You can exchange the position of any two symbols without changing the value of the represented number. This is not a positional number system.

The numbers can be recorded in groups to make them easier to count accurately, such as: 11111 11. A variation of this grouping is to write four vertical marks, and then draw a diagonal line to indicate a complete group of five.

Figure 5. Grouping in Base One

A tally system may have been the first number system you ever used.

Other Bases

Without debasing the discussion, even irrational bases have been studied.[10] People have considered $\sqrt{2}$, π, and e as number system bases. Observe that just because the base of a number system is irrational, it does not mean that only irrational numbers can be represented in that base. Consider a number system based on $\sqrt{2}$. This effectively becomes a base 2 (binary) number system if all odd numbered coefficients (positive and negative) are zero, and even-numbered coefficients are either zero or one in the expression

$$\cdots a_2 b^2 + a_1 b^1 + a_0 b^0 + a_{-1} b^{-1} + a_{-2} b^{-2} + \cdots$$

A fascinating example of an irrational base number system in the base Phi (Φ) representation. The defining equation for Phi includes the square root of 5: This is discussed later as a special topic.

Know Nothing

Some numbers that are important to Western culture have been recorded using notation systems that have no symbol for zero. For example: Roman numerals, Hebrew, Greek, and Aramaic. The idea of "zero" came to Western culture through the Hindu-Arabic number system.

[10] "Irrational-Base Number System", everything2 (a discussion forum on the Internet), (15 June 2000) http://everything2.com/index.pl?node_id=604415 visited 17 December 2007

Indexes and Subscripts

We use subscripts or other special symbols to identify which number system a number belongs to. Let us review the concept of subscripts and related ideas.

Figure 6. Index and Subscript Positions

The letter A shown above has 4 index positions illustrated here as boxes which are labeled w, x, y, and z. Notice that the boxes are bisected by horizontal lines that touch the upper and lower edges of the letter A, and that the boxes are just a little less than half as tall as the height of the A. Normally, the lines are not drawn; nor are the boxes.

Labels in upper positions, w and z, are called ***superscripts***. Labels in lower positions, x and y, are called ***subscripts***. Positions w and x are the most common positions used. Positions y and z are used in chemistry, physics, and in some special areas of mathematics and engineering. The purpose of a label depends on the context of its use. It can be merely an index to identify one item from a collection of similar items. For example, A_{boy} identifies the subset of variable A such that only those which are boys are selected. This distinguishes that subset from A_{girl}. A subscript can also be used as an index to select one member from a group. For example, the subscript "13" in A_{13} distinguishes that member from A_{12} and A_{14}. Sometimes the contents of a box is merely a label. Sometimes it is a number or a simple variable, like **k**, that takes on values of counting numbers or integers. Sometimes the contents of a box can be a complicated formula.

We will use a subscript after a number to identify the base of the number system that number belongs to. For example, 101_2 is a number that belongs to the base 2 number system. The number 792_{10} belongs to the base 10 number system. The number 704_8 belongs to the base 8 number system. The number $F2A_{16}$ belongs to the base 16 number system.

We will use subscripts to identify the position of a particular digit within a number, starting our count with zero from the right hand end. For example, the number 496 has a 6 in position zero, a 9 in position 1, and a 4 in position 2. We can refer to these more abstractly by saying

Table 8. Digits in a Positional Number.

$A_0 = 6$
$A_1 = 9$
$A_2 = 4$

Then $A_2A_1A_0 = 496$.

Sometimes, these labels take on special meanings beyond just being an index or locator. In mathematics, it is common for a superscript in position w to represent a power. For **example**, A^3 means 3 copies of A are multiplied together. A^3 = A A A. When A = 2, then 2^3 = 2 x 2 x 2 = 8. We will use this meaning.

Common Number Systems for Computers

Decimal Numbers

The decimal number system used in our every day life is a number system involving ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Most people have 10 digits, 5 on each hand… the original digital calculator. Because this is the number system we use most of the time, no special markings are normally used to indicate a number as **belonging** to the decimal system. When it is necessary to distinguish a number as being a decimal number, we use the subscript **10**, the capital letter **D**, or a lower case **d**. For example, the subscript **10** in 101_{10} or the capital letter **D** in 101**D** identifies the number 101 as being a base 10 number.

Binary Numbers

The computer uses a number system, called the **binary** number system, which uses only the digits **0** and **1**. A **bi**nary digi**t** is called a "**bit**". A "1" bit is "on", and a "0" bit is "off". Computers use binary because it is easy and inexpensive to build hardware that can normally be only in one of two states. We represent one state by "0", and the other state by "1".

A binary number is represented with a subscript **2** after the binary number, or the letters **B** or **b** after the binary number, or the symbol **%** before the binary number. The non-subscript forms are useful for high-speed display. It is useful to leave a space or insert a colon (**:**) between every 4th binary digit, starting from the right end. This is like using a comma for grouping digits when writing decimal numbers. It is useful, but not required, to write binary numbers in groups of 4 bits, appending zeros to the left end if necessary. Examples:

11 0101_2	11 0101**B**	11 0101**b**	**%**11 0101	blanks
11:0101_2	11:0101**B**	11:0101**b**	**%**11:0101	colons
0011:0101_2	0011:0101**B**	0011:0101**b**	**%**0011:0101	padded on the left

Table 9. Grouping and Padding Binary Numbers

Notice that the symbol identifying the base of the number appears only once in a number, **not** with each digit in the number.

Hexadecimal Numbers

Programmers and computer technicians sometimes use a coding system called ***hexadecimal*** (with a base of **16**). We often abbreviate the name as ***hex***. The digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. (The hexadecimal *number* **A** is equivalent to the decimal number 10, the hexadecimal *number* **B** is equivalent to the decimal number 11, etc.) A hexadecimal number requires exactly 4 binary digits to represent it. Instead

of copying a long binary number, it is much easier to write the number using hexadecimal. It uses less space, it is easier to remember, and you make fewer mistakes. You may sometimes see hexadecimal numbers flash before your eyes shortly after you turn your computer on and the computer performs its power-on self test.

The electronic computer is not the first beneficiary of the hexadecimal system. The English system of weights used the Roman ounce: 1 pound = 16 ounces (weight), 1 ounce (weight) = 16 drams, 1 dram = 3 scruples. 1 gallon = 16 cups, 1 cup = 16 tablespoons. 1 pint (liquid measure) = 16 fluid ounces. 1 quart = quarter of a gallon = 4 cups. 1 peck = 16 pints (dry measure). 1 ounce (liquid measure) of water at standard temperature and pressure weighs 1 ounce (weight).[11] (Rombauer, pg. 546) A pint, a pound, the world 'round.

An hexadecimal number is represented with a subscript of **16** after the hexadecimal number, the letters **H** or **h** after the hexadecimal number, or the symbols **Ox** before the hexadecimal number. The "O" in "Ox" is the capital letter O (Oh, not zero). The non-subscript forms are useful for high-speed display. It is useful to leave a space between every 4th hexadecimal digit, starting from the right end.

Examples: $B4AD_{16}$ B4AD**H** B4AD**h** **Ox**B4AD

Octal Numbers

Programmers and technicians sometimes use a coding system called **octal** (with a base of **8**). To remember that octal means 8, think of an octopus, which has 8 legs, or an octogon which has 8 sides (3 more than the Pentagon). The digits are 0, 1, 2, 3, 4, 5, 6, and 7. An octal number takes exactly 3 bits to represent. Octal is found on special purpose computing systems, including some numerically controlled machines and embedded processors.

An octal number is represented with a subscript of **8** or the capital letter **Q** (for Quaternion) after the octal number, or the capital letter **O** (Oh, not zero) before the octal number. The non-subscript form is useful for high-speed display. It is useful to leave a space between every 4th digit, starting from the right end.

Examples: 4273_{8} 4273**Q** **O**4273

A Quaternion is a generalization of a complex number, introduced by Hamilton. We do not use H as a label for these because H is used for Hexadecimal. It is useful in physics and engineering. Mathematicians, physicists, astronomers, and nuclear engineers know of these, and electrical engineers would profit from using them. One simple theory now connects thermal, electric, and magnetic phenomena.[12]

Numbers Names

We have a special vocabulary for saying numbers in the decimal system. Some number names that are peculiar to the decimal system are in the table below. When we use any of these names, we automatically imply that the number we are saying belongs to the base 10 (decimal) number system. These number names are absolutely meaningless

[11] Irma S. Rombauer and Marion Rombauer Becker, *Joy of Cooking*, Bobbs-Merrill Company (1964).
[12] Peter Michael Jack, Hypercomplex Systems, "The Quaternion Electromagnetic Equations", http://www.hypercomplex.com/ (2006), visited 03 February 2008.

in any number system except base 10 and are not to be used except when referring to decimal numbers.

For numbers from any other base number system, the digits are read individually from left to right. For example, in binary, the number 1011_2 is read as "one-zero-one-one, base 2". In hexadecimal, the number $F2A_{16}$ is read as "F-two-A, base 16". When the context is not clear, the base of the number system must be explicitly stated.

Table 10. Decimal System Number Names

Decimal Number	***Decimal* Number Name**
10	*ten*
11	*eleven*
12	*twelve*
13	*thirteen*
14	four*teen*
…	…
19	nine*teen*
20	*twenty*
21	*twenty*-one
22	*twenty*-two
…	…
30	*thirty*
31	*thirty*-one
…	…

Decimal Number	***Decimal* Number Name**
40	*forty*
50	*fifty*
…	…
90	*ninety*
100	one *hundred*
200	two *hundred*
…	…
1,000	one *thousand*
…	…
1,000,000	one *million*
1,000,000,000	one *billion* (USA)
1,000,000,000,000	one *trillion* (USA) one billion (UK, France)
1,000,000,000,000,000,000	one *trillion* (UK)
Many	one gezillion ☺

Decimal Numbers 0 through 15 and Equivalent Hexadecimal and Binary Numbers

Before you go to bed tonight, write the following table of decimal, hexadecimal, and binary numbers from memory.

In a moment, you will see that it is not that hard. You only are to memorize the table for zero through fifteen, which is the first sixteen numbers. It is important to keep table entries in the correct row. For example, you should associate decimal 13 with hexadecimal D_{16} and binary 1101_2.

Table 11. The Number System Table

Decimal	Hexadecimal	Binary
Base 10	Base 16	Base 2
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

There are four ways to learn this table. You need to use only one of these ways: memorization, pattern recognition, position value addition, and counting. Use the way that fits your learning style.

Memorization

Writing the list of hexadecimal numbers is easy. It is just like decimal numbers until you get to 10. The hexadecimal number system gives us 16 single-digit numbers. The last 6 numbers are: the number A, the number B, the number C, the number D, the number E, and the number F. When working in hexadecimal, think of A through F as being numbers (not letters). The part that takes work is learning the binary. If you memorize lists easily, write the table until you get it right several times.

Pattern Recognition Method.

Number the columns from the right end, beginning with position #0. Number the rows from the top, beginning with row #0. Beginning with the number zero, the first 4-bit binary number is a string of four zeros.

Table 12. Pattern Recognition Method of Writing the Number System Table

	Binary	Binary	Binary	Binary
	Step 1	Step 2	Step 3	Step 4
Row Number	Position Number 3210	Position Number 3210	Position Number 3210	Position Number 3210
	Place Value 8421	Place Value 8421	Place Value 8421	Place Value 8421
0	0___	00__	000_	0000
1	0___	00__	000_	0001
2	0___	00__	001_	0010
3	0___	00__	001_	0011
4	0___	01__	010_	0100
5	0___	01__	010_	0101
6	0___	01__	011_	0110
7	0___	01__	011_	0111
8	1___	10__	100_	1000
9	1___	10__	100_	1001
10	1___	10__	101_	1010
11	1___	10__	101_	1011
12	1___	11__	110_	1100
13	1___	11__	110_	1101
14	1___	11__	111_	1110
15	1___	11__	111_	1111

Position Value Addition

Each decimal number from 0 through 15 can be represented by a 4-bit binary number. Write the position values in four columns, 8 4 2 1. Find that combination of position values that adds up to the decimal value for that row of the table.

Example: Find the binary value corresponding to decimal 13.

Table 13. Position Values for 4-bit Binary Number

Position Value	8	4	2	1
13 =	1	1	0	1

8 + 4 +0+ 1 = 13, so the corresponding binary number is 1101_2 .

Counting Method

Use the table for binary addition to add “1” to each row to get to the next row, with row #0 containing the number “0000”. Before looking at the addition table for

binary, briefly review addition for decimal numbers. Suppose we want to add the decimal numbers 879 and 964.

Table 14. Adding Decimal Numbers With Carry

Carry	1	1	1	
		8	7	9
Add		9	6	4
Answer	1	8	4	3

When we add 9 + 4, the answer is 3, with a carry of 1 to the next column to the left. When we add 7 + 6 + 1, the answer is 4, with a carry of 1 to the next column to the left. When we add 8 + 9 + 1, the answer is 8 with a carry of 1 to the next column to the left. When we run out of single-digit numbers, we must generate a carry to the next column to the left.

Binary Addition

We do the same procedure in binary, except that only two numbers can be used (0 and 1). The addition table for binary is as follows.

Table 15. Binary Addition Table

Binary ADDITION	x=0	x=1
y=0	0 **0**	0 **1**
y=1	0 **1**	1 **0**

The addition table is just like the decimal table, except we can only go up to 1 rather than 9. Here is what the table means.

- x + y = 0 + 0 = **0** with a carry of 0.
- x + y = 1 + 0 = **1** with a carry of 0.
- x + y = 0 + 1 = **1** with a carry of 0.
- x + y = 1 + 1 = **0** with a carry of 1.

Sample Binary Addition Problems.

Add $1101_2 + 0011_2$.

Carry	1	1	1	1	
		1	1	0	1
Add		0	0	1	1
Answer	1	0	0	0	0

The answer is 10000_2.

Add $0110_2 + 0101_2$.

Carry	0	1	0	0	
		0	1	1	0
Add		0	1	0	1
Answer		1	0	1	1

The answer is 1011_2.

To create the binary table from 0 to 15 decimal, begin with 0000 in row #0. To get to each additional row, add 1 to the current row using the binary addition table.

Carry	**0**	**0**	**0**	**0**	
Row #0		0	0	0	0
Add		0	0	0	1
Row #1		**0**	**0**	**0**	**1**

Carry	**0**	**0**	**0**	**1**	
Row #1		0	0	0	1
Add		0	0	0	1
Row #2		**0**	**0**	**1**	**0**

Carry	**0**	**0**	**0**	**0**	
Row #2		0	0	1	0
Add		0	0	0	1
Row #3		**0**	**0**	**1**	**1**

Carry	**0**	**0**	**1**	**1**	
Row #3		0	0	1	1
Add		0	0	0	1
Row #4		**0**	**1**	**0**	**0**

Carry	**0**	**0**	**0**	**0**	
Row #4		0	1	0	0
Add		0	0	0	1
Row #5		**0**	**1**	**0**	**1**

Carry	**0**	**0**	**0**	**1**	
Row #5		0	1	0	1
Add		0	0	0	1
Row #6		**0**	**1**	**1**	**0**

Carry	**0**	**0**	**0**	**0**	
Row #6		0	1	1	0
Add		0	0	0	1
Row #7		**0**	**1**	**1**	**1**

Carry	**0**	**1**	**1**	**1**	
Row #7		0	1	1	1
Add		0	0	0	1
Row #8		**1**	**0**	**0**	**0**

Carry	**0**	**0**	**0**	**0**	
Row #8		1	0	0	0
Add		0	0	0	1
Row #9		**1**	**0**	**0**	**1**

Carry	**0**	**0**	**0**	**1**	
Row #9		1	0	0	1
Add		0	0	0	1
Row #10		**1**	**0**	**1**	**0**

Abstract Counting in Base 4

Good. You have counted in binary. Generalize your knowledge. Try counting in base 4 using {0,1,2,3}. You can also try it with {circle, stick, square, star} or {w,x,y,z}. Here is a partial table using four base-4 digits, corresponding to 8 binary digits. Each base 4 digit corresponds to 2 bits.

Table 16. Counting in Base 4

0000	0100	0200	0300	1000	1100	1200	1300	2000	2100	2200
0001	0101	0201	0301	1001	1101	1201	1301	2001	2101	2201
0002	0102	0202	0302	1002	1102	1202	1302	2002	2102	2202
0003	0103	0203	0303	1003	1103	1203	1303	2003	2103	2203
0010	0110	0210	0310	1010	1110	1210	1310	2010	2110	2210
0011	0111	0211	0311	1011	1111	1211	1311	2011	2111	2211
0012	0112	0212	0312	1012	1112	1212	1312	2012	2112	2212
0013	0113	0213	0313	1013	1113	1213	1313	2013	2113	2213
0020	0120	0220	0320	1020	1120	1220	1320	2020	2120	2220
0021	0121	0221	0321	1021	1121	1221	1321	2021	2121	2221
0022	0122	0222	0322	1022	1122	1222	1322	2022	2122	2222
0023	0123	0223	0323	1023	1123	1223	1323	2023	2123	2223
0030	0130	0230	0330	1030	1130	1230	1330	2030	2130	2230
0031	0131	0231	0331	1031	1131	1231	1331	2031	2131	2231
0032	0132	0232	0332	1032	1132	1232	1332	2032	2132	2232
0033	0133	0233	0333	1033	1133	1233	1333	2033	2133	2233

If you do not understand this, but can write the decimal, hexadecimal, and binary table, don't waste your time. Move on.

Problem: Fill out the Number Systems Table four times.

Number Systems Table

0→15	0→F	0000→1111
Dec	**Hex**	**Binary**
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

0→15	0→F	0000→1111
Dec	**Hex**	**Binary**

0→15	0→F	0000→1111
Dec	**Hex**	**Binary**
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

0→15	0→F	0000→1111
Dec	**Hex**	**Binary**

Conversion Between Binary and Hexadecimal

There is a special relationship between binary and hexadecimal (hex) numbers. Each hex digit can be represented by a 4-bit code, and each 4-bit code can be represented by a single hex digit. Characters of the alphabet are represented by 8 bits. Using this conversion, we can represent a character as 2 hex digits.

We do conversions between binary and hexadecimal by table lookup.

Examples:

E_{16}	= 1110_2	= 1110**B**	= 1110**b**	= **%**1110
0_{16}	= 0000_2	= 0000**B**	= 0000**b**	= **%**0000
7_{16}	= 0111_2	= 0111**B**	= 0111**b**	= **%**0111
F_{16}	= 1111_2	= 1111**B**	= 1111**b**	= **%**1111

Example: Convert the hexadecimal number C_{16} to binary.

C_{16}	= 1100_2	= 1100**B**	= 1100**b**	= **%**1100

Remember that

Table 17. ACE Your Knowledge of Hexadecimal

10		**A**
12		**C**
14		**E**ven

A_{16} is 10_{10}, C_{16} is 12_{10}, and E_{16} is 14_{10} in decimal.

Problems: Fill in all the missing values. Convert the single digit hexadecimal numbers to 4-bit binary numbers.

Table 18. Simple Conversions by Inspection

	Hexadecimal	Decimal	Binary
Problem 1.	2_{16}	= 2_{10}	= 0010_2
Problem 2.	7_{16}	= 7_{10}	=
Problem 3.	8_{16}	=	=
Problem 4.	B_{16}	= 11_{10}	=
Problem 5.	D_{16}	=	=
Problem 6.	F_{16}	=	=

Use table lookup in the opposite direction to convert a 4-bit binary number to hex. Example: Convert 1010_2 and 0101_2 to hexadecimal.

1010_2 = A_{16} = A**H** = A**h** = **Ox**A
0101_2 = 5_{16} = 5**H** = A**h** = **Ox**5

To convert a long string of binary digits to hexadecimal, the first step is to group bits into groups of four, starting from the right hand side. What is important is to start from the right hand side when doing the grouping. What is important? It is important is to start the grouping from the right hand side. One more time... What is important? (Your turn.) Look at an example.

Convert $11101001001001111000000001_2$ to hexadecimal.

We being grouping by starting at the position zero number.

Table 19. Grouping Bits for Conversion from Binary to Hexadecimal

←	←	←	←	←	←	Start Here ?
11	:1010	:0100	:1001	:1110	:0000	:0001

The left-most group has only two bits, so we **pad on the left end with zeros** to get

Table 20. Zero Padding on the Left

0011	:1010	:0100	:1001	:1110	:0000	:0001

The next step is to do table lookup. Find each group of 4 binary digits and locate the corresponding single-digit hexadecimal number.

Table 21. Convert from Binary to Hexadecimal by Inspection

0011	:1010	:0100	:1001	:1110	:0000	:0001
3	**A**	**4**	**9**	**E**	**0**	**1**

The answer is $3A49E01_{16}$ = 3A49E01**H** = 3A49E01**h** = **Ox**3A49E01

Binary to Hexadecimal Problems

Convert the following binary numbers to hexadecimal.

Problem 7. 1101_2
Problem 8. 1001**B**
Problem 9. **%**0111
Problem 10. 0101_2
Problem 11. 1 0000 0001**b**
Problem 12. **%**110 1011 1000 0001 0101

To convert several hex digits to binary, convert each hex digit to a 4-bit code using table lookup. The result is the binary code for the hex number.

Examples: Convert hexadecimal AF_{16} to binary.

A	F		
1010	1111	=	$1010:1111_2$

Convert hexadecimal 36E**H** to binary.

3	6	E_{16}		
0011	0110	1110	=	$0011:0110:1110_2$

Convert hexadecimal **Ox**FACE to binary

F	A	C	E		
1111	1010	1100	1110	=	$1111:1010:1100:1110_2$

Hexadecimal to Binary Problems

Problem 13. Convert $5E_{16}$ to binary.

Problem 14. Convert **Ox**DAB to binary.

Problem 15. Convert 83C2FA**H** to binary.

Problem 16. Convert $10ED7_{16}$ to binary.

Data Representation

A Little Byte of History

Computers were first used for computational tasks. The application of using computers to process non-numeric data came at a slightly later time.

The approach to representation of character data was to represent each character by a unique code. This is a logical approach for alphabet-based languages. It is not the only logical approach. Another approach is to represent line strokes by a code, and require characters to be formed by a combination of line strokes. This approach is a reasonable way of representing ideographs. It requires the ability of the display device to overlay images.

An early character code by IBM was Binary Coded Decimal (BCD). Each character was represented by 6 bits (a pair of octal numbers). This was enough to represent base-10 digits, the Latin alphabet with capital letters, and a few punctuation characters. The implementation of character handling internally depended upon the machine. In the early 1960s, the IBM 7040, a scientific computer, had a 36-bit word. The IBM 7040 fetched words from memory 36 bits at a time as a unit. Special instructions were implemented to permit extraction of 6-bit bytes from this 36-bit word after the whole word was in the accumulator. The IBM 7040 was the descendent of the IBM 704.

Another scientific computer was the much smaller IBM 1620. It fetched and operated on data one character at a time. It did addition and multiplication by decimal table look-up, and was used at University of South Carolina to control scientific experiments. The corresponding business machine was the IBM 1401. It was the ubiquitous machine of its day. It also was a character oriented machine with 6-bits for a BCD alphanumeric character, plus two more bits (parity and field definition). Its word length was variable. It used a simple language for business data processing, RPG.

Digital Equipment Corporation produced a minicomputer in 1965, the PDP-8. This was the first computer sold retail, at a price that small organizations could purchase. It sold for approximately $10,000 of 1965 U.S. dollars (about 15 month's salary of a new college graduate in a technical field). It filled a big gap in the market at a time when other machines were only leased, and required large dedicated staffs.

The basic memory configuration of the PDP-8 was 4 kB of 12-bit words. The PDP-8 assumed a character set of 7-bit ASCII, with the 8th bit set to 1. Description of the 7-bit ASCII code is available on the Internet.[13] This gave the ability to only represent upper case letters. The programmer could use other character sets. Commonly, a stripped 7-bit ASCII code was used by ANDing an ASCII character with octal 77. This permitted storing two characters per 12-bit word.[14]

---

[13] Roman Czyborra, "Good ole' ASCII" (1998). http://czyborra.com/charsets/iso646.html visited 01 January 2008.

[14] Douglas Jones, "Frequently Asked Questions About the DEC PDP-8 Computer" ftp://rtfm.mit.edu/pub/usenet/alt.sys.pdp8/PDP-8_Frequently_Asked_Questions

IBM introduced the 8-bit EBCDIC (Extended Binary Coded Decimal Interchange Code) character set which includes codes for lower case letters and additional special characters. The IBM 360 accessed memory at an 8-bit byte level. This was a key development in the design of machines for business data processing. It also implemented the concept of microinstructions. Prior to the IBM 360, machine language instructions were loaded into instruction registers, and the bit pattern was used directly to select the hardware logic to perform the desired operation. The logic circuits required multiple clock cycles to complete one operation. The microcode approach was to execute a sequence of microinstructions, each requiring one clock cycle, for each macroinstruction loaded into the instruction register. This gave IBM the ability to add to the power of its machines by modifying its microcode. It also made hardware diagnostics easier. IBM established dominance in the business world with the IBM 360 series. A nice introduction to microprogramming by Mark Smotherman is available on the Internet.[15]

UNICODE is an attempt to create a single international code for information interchange. It includes a version of ASCII as a subset.

Character Codes

In today's digital computers, data is represented in computer memory as binary digits. Each character is represented as a unique binary code in memory. The number of binary digits required to represent one character is called a **byte**. There are three primary character codes, ASCII, EBCDIC, and UNICODE.

US-ASCII (American Standard Code for Information Interchange) is an 8-bit code used on almost all computers worldwide except for IBM mainframes. ASCII was originally introduced as a 7-bit code in 1968 as ANSI X3.4. The 8-bit variations are extensions of ASCII. Jim Price has a nice description and history for ASCII on the Internet.[16] ASCII has been extended to several 8-bit codes. Another nice, brief history of ASCII is by Roman Czyborra.[17] This includes images of several ASCII character sets.

EBCDIC (Extended Binary Coded Decimal Interchange Code) is an 8-bit code used on IBM mainframes. EBCDIC is proprietary to IBM. Prior to the Internet, EBCDIC was the predominant character code in use. The Internet and the ubiquitous presence of the personal computer have made ASCII the most commonly used code.

EBCDIC remains important in the mainframe world of commercial data processing. See Multilingual Support in Internet/IT Applications, Trans-European Research and Education Networking Association.[18] Kosta Kostis has provided EBCDIC code tables.[19] [20]

---

[15] Mark Smotherman, "A Brief History of Microprogramming" (March 1999)
[16] Jim Price, "ASCII Chart" (28 October 2001). Visited 02 November 2001.
[17] Roman Czyborra, "The ISO 8859 Alphabet Soup" (12 January 1998). Visited 02 November 2001.
[18] Trans-European Research and Education Networking Association, http://www.terena.org/. Visited 01 January 2008.
[19] Kosta Kostis, "EBCDIC Codepage 037", http://www.kostis.net/charsets/ebc037.htm (17 February 2001). Visited 02 November 2001.
[20] Kosta Kostis, "EBCDIC Codepage 1047", http://www.kostis.net/charsets/ebc1047.htm (17 February 2001). Visited 02 November 2001.

UNICODE (Universal Code) is a 16-bit extension code of ASCII used to represent symbols of other languages, mathematics, and other disciplines. This includes languages that use ideograms (like Chinese, Japanese, Korean) as well as alphabets (like Latin, Hebrew, Greek). Microsoft Office 2000 (and later versions) use UNICODE. Corel PerfectOffice permits import and export of UNICODE text files.

The chart on the following page shows the alphabetic, numeric, and blank character (␣) with their codes in hex. The symbol for the blank is often made with the slash through the **b**. The code for the letter "Oh" is different than the code for the number zero. The number zero is often written with a slash. Upper case characters have a code different than lower case characters.

For example:

The *character* **A** represented in EBCDIC is $C1_{16}$
The *character* **A** represented in ASCII is 41_{16}
The *character* **A** represented in UNICODE is 0041_{16}

UNICODE representation of ASCII consists of the ASCII code prefixed with 00_{16}. This selection was made to keep UNICODE compatible with most existing software. There are many symbols in UNICODE that have hex numbers different than zero in the first two positions. A complete definition of UNICODE can be obtained from Unicode, Inc.[21]

Codes used for data processing, such as those above, are suitable for sorting. The order in which characters appear in a code is called the collating sequence or lexicographic order. In applications, the treatment of the blank and special rules of language impose additional considerations. This is important in constructing dictionaries and telephone books. For example, European names sometimes include "von", "van" and other prefixes. European rules for name collating are not the same as in the United States. Internal character codes are just the beginning of the process.

There are companies that sell the service of converting data from one format to another. VEDIT by Greenview Data, Inc. is one example.[22]

[21] Unicode Home Page, http://www.unicode.org/. Visited 01 January 2008.
[22] Greenview Data, Inc., "VEdit", http://www.vedit.com/index.html (2007). Visited 01 January 2008.

Example: Use Table 22 on page 36 to code the following characters first to hexadecimal, and then to binary.

Character	**2**	**␣**	**C**	**A**	**T**	**S**
EBCDIC	F2	40	C3	C1	E3	E2
Binary	1111:0010	0100:0000	1100:0011	1100:0001	1110:0011	1110:0010

Character	**2**	**␣**	**C**	**A**	**T**	**S**
ASCII	32	20	43	41	54	53
Binary	0011:0010	0010:0000	0100:0011	0100:0001	0101:0100	0101:0011

Character	**2**	**␣**	**C**
UNICODE	0032	0020	0043
Binary	0000:0000:0011:0010	0000:0000:0010:0000	0000:0000:0100:0011

Character	**A**	**T**	**S**
UNICODE	0041	0054	0053
Binary	0000:0000:0100:0001	0000:0000:0101:0100	0000:0000:0101:0011

Character Code Problems

Code the following in hexadecimal using EBCDIC and ASCII, and then code each into binary. Notice upper and lower case, letters and numbers.

Problem A1. 2 Zoos

Character	**2**	**␣**	**Z**	**o**	**o**	**s**
EBCDIC						
Binary	____:____	____:____	____:____	____:____	____:____	____:____

Character	**2**	**␣**	**Z**	**o**	**o**	**s**
ASCII						
Binary	____:____	____:____	____:____	____:____	____:____	____:____

Problem A2. 1 Bill

Character	**1**	**␣**	**B**	**i**	**l**	**l**
EBCDIC						
Binary	____:____	____:____	____:____	____:____	____:____	____:____

Character	**1**	**␣**	**B**	**i**	**l**	**l**
ASCII						
Binary	____:____	____:____	____:____	____:____	____:____	____:____

EBCDIC, US-ASCII, and UNICODE Character Codes

Table 22. EBCDIC, US-ASCII, UNICODE for Latin Alphabet

TEXT	EBCDIC	ASCII	UNICODE	TEXT	EBCDIC	ASCII	UNICODE
␣	40	20	0020				
,	6B	2C	002C				
.	75	2E	002E				
0	F0	30	0030				
1	F1	31	0031				
2	F2	32	0032				
3	F3	33	0033				
4	F4	34	0034				
5	F5	35	0035				
6	F6	36	0036				
7	F7	37	0037				
8	F8	38	0038				
9	F9	39	0039				
?	6F	3F	003F				
A	**C1**	**41**	**0041**	**a**	**81**	**61**	**0061**
B	C2	42	0042	b	82	62	0062
C	C3	43	0043	c	83	63	0063
D	C4	44	0044	d	84	64	0064
E	C5	45	0045	e	85	65	0065
F	C6	46	0046	f	86	66	0066
G	C7	47	0047	g	87	67	0067
H	C8	48	0048	h	88	68	0068
I	C9	49	0049	i	89	69	0069
J	D1	4A	004A	j	91	6A	006A
K	D2	4B	004B	k	92	6B	006B
L	D3	4C	004C	l	93	6C	006C
M	D4	4D	004D	m	94	6D	006D
N	D5	4E	004E	n	95	6E	006E
O	D6	4F	004F	o	96	6F	006F
P	D7	50	0050	p	97	70	0070
Q	D8	51	0051	q	98	71	0071
R	D9	52	0052	r	99	72	0072
S	E2	53	0053	s	A2	73	0073
T	E3	54	0054	t	A3	74	0074
U	E4	55	0055	u	A4	75	0075
V	E5	56	0056	v	A5	76	0076
W	E6	57	0057	w	A6	77	0077
X	E7	58	0058	x	A7	78	0078
Y	E8	59	0059	y	A8	79	0079
Z	E9	5A	005A	z	A9	7A	007A

Non-printing codes used for control purposes are not shown in this table.

The purpose of problems 3 and 4 is to illustrate that different characters that appear similar have different codes. Be sure to read the problem notes before doing the problem. Distinguish between numbers, upper case letters, and lower case letters.

Problem A3. Hi ␣Ho ␣10 ␣lO?

Notes:

- "Hi" are letters.
- "Ho" are letters.
- "10" are numbers.
- "lO" are letters, followed by punctuation.

Character	**H**	**i**	␣	**H**	**o**	␣
EBCDIC						
Binary	:	:	:	:	:	:
ASCII						
Binary	____:____	____:____	____:____	____:____	____:____	____:____

Character	**1**	**0**	␣	**l**	**O**	**?**
EBCDIC						
Binary	:	:	:	:	:	:
ASCII						
Binary	____:____	____:____	____:____	____:____	____:____	____:____

Problem A4. 2ZS50Oo ␣1li.

- 2ZS5 Number, Letter, Letter, Number
- 0Oo Number, Letter, Letter
- 1li. Number, Letter, Letter, Punctuation

Character	**2**	**Z**	**S**	**5**	**0**	**O**
EBCDIC						
Binary	:	:	:	:	:	:
ASCII						
Binary	____:____	____:____	____:____	____:____	____:____	____:____

Character	**o**	␣	**1**	**l**	**i**	**.**
EBCDIC						
Binary	:	:	:	:	:	:
ASCII						
Binary	____:____	____:____	____:____	____:____	____:____	____:____

Problem A5. Code the following in hexadecimal using ASCII, and then convert to binary. Oven Hot

Character	O	v	e	n
ASCII hex				
Binary				

Character		H	o	t
ASCII hex				
Binary				

Problem A6. Code the following in hexadecimal using **EBCDIC**, and then convert to binary. 10 is lo

Character	1	0		i
EBCDIC hex				
Binary				

Character	s		l	o
EBCDIC				
Binary				

Problem A7. Convert the following to hexadecimal, and then to characters using **EBCDIC** code.

Binary	1111:0001	0100:0000	1101:0001	1010:0100	1001:0100	1001:0111
EBCDIC Hex						
Character						

Jubilation Code and Gran Zeff Code

The purpose of this section is to demonstrate the benefit of Unicode by using a symbol set not represented in US-ASCII. Unicode for Latin letters have 00 as the first two hexadecimal characters. The following symbol set shows codes that are not 00 in the first two Unicode positions. The first two hexadecimal characters index which code table to use. The “Secret Symbols” are all Greek to me.

Table 23. Gran Zeff Code, Jubilation Code, and UNICODE

SECRET SYMBOL	JUBILATION CODE (Hexadecimal)	GRAN ZEFF CODE (Hexadecimal)	UNICODE (Hex)	SECRET SYMBOL	JUBILATION CODE (Hexadecimal)	GRAN ZEFF CODE (Hexadecimal)	UNICODE (Hex)
blank	40	20	0020				
a	A1	45	03B1	x	D2	76	03BE
b	A2	46	03B2	o	D3	77	03BF
g	A3	47	03B3	p	D4	78	03C0
d	A4	48	03B4	r	E1	85	03C1
e	B1	55	03B5	**s**	E2	86	03C2
z	B2	56	03B6	s	E3	87	03C3
h	B3	57	03B7	t	E4	88	03C4
q	B4	58	03B8	u	F1	95	03C5
i	C1	65	03B9	j	F2	96	03C6
k	C2	66	03BA	c	F3	97	03C7
l	C3	67	03BB	y	F4	98	03C8
m	C4	68	03BC	w	F5	99	03C9
n	D1	75	03BD				

Example 1: Code the word δεκα (ten) from SECRET SYMBOL to JUBILATION CODE hexadecimal, GRAN ZEFF CODE hexadecimal, and UNICODE hexadecimal representations.

Secret Symbol	d	e	k	a
Jubilation Code	A4	B1	C2	A1
Gran Zeff Code	48	55	66	45
Unicode	03B4	03B5	03BA	03B1

Example 2: Code the GRAN ZEFF CODE binary into GRAN ZEFF CODE hexadecimal, and then to SECRET SYMBOL. (man)

0100:0101	0111:0101	0101:1000	1000:0101	1001:1001	0111:1000	0111:0111	1000:0110
45	75	58	85	99	78	77	86
a	n	q	r	w	p	o	**s**

Example 3: Code the UNICODE binary into UNICODE hexadecimal, and then into SECRET SYMBOL. (word).

0000:0011:1011:1011	03BB	l
0000:0011:1011:1111	03BF	o
0000:0011:1011:0011	03B3	g
0000:0011:1011:1111	03BF	o
0000:0011:1100:0010	03C2	**s**

Problem B1: Code the word a d e l f o **s** (brother) from SECRET SYMBOL to JUBILATION CODE hexadecimal, and then to JUBILATION CODE binary.

a	d	e	l	j	o	**s**

Problem B2: Code the word d w r o n (gift) from SECRET SYMBOL to GRAN ZEFF CODE hexadecimal, and then to GRAN ZEFF CODE binary.

d	w	r	o	n

Problem B3: Code the word s o f o s (wise) from SECRET SYMBOL to UNICODE hexadecimal, and then to UNICODE binary.

s	o	j	o	**s**

Problem B4: Code the word k a r d i a (heart) from SECRET SYMBOL to UNICODE hexadecimal, and then to UNICODE binary.

k	a	r	d	i	a

Problem B5: Code the JUBILATION CODE binary into JUBILATION CODE hexadecimal, and then to SECRET SYMBOL. (death)

1011:0100	1010:0001	1101:0001	1010:0001	1110:0100	1101:0011	1110:0010

Problem B6: Code the GRAN ZEFF CODE binary into GRAN ZEFF CODE hexadecimal, and then to SECRET SYMBOL. (wicked)

0111:1000	0111:0111	0111:0101	0101:0111	1000:0101	0111:0111	1000:0110

Problem B7: Code the UNICODE binary into UNICODE hexadecimal, and then into SECRET SYMBOL. (nation)

0000:0011: 1011:0101	0000:0011: 1011:1000	0000:0011: 1011:1101	0000:0011: 1011:1111	0000:0011: 1100:0010

Additional Information About Codes

A nice history of character codes is given by Steven J. Searle.[23] For the detail-oriented person, see Jukka Korpela, "A tutorial on character code issues".[24]

A common exercise for first-semester programming in computer science is to generate a printout of a character set used by a printer. The exercise includes printing the corresponding hexadecimal, decimal, octal, and binary values of the internal character code. OSdata.com has made available code tables indexed by symbol or control code.[25]

Communication imposes different requirements than data processing. In communication, the goal is to transmit codes in a way that helps reconstruction of the original message. If a code is intended to support transmission of sequences, such as number sequences (like a counter), you want the bit patterns to be designed to help detect mistakes. One approach is called Gray Coding.[26]

---

[23] Steven J. Searle, "A Brief History of Character Codes in North America, Europe, and East Asia", Sakamura Laboratory, University Museum, University of Tokyo (1999), http://tronweb.super-nova.co.jp/characcodehist.html, Visited 02 Nov 2001.

[24] Jukka Korpela, "A tutorial on character code issues", Tampere University of Technology, Finland, http://www.cs.tut.fi/~jkorpela/chars.html, (24 October 2001). Visited 02 Nov 2001.

[25] "Character Codes", OSdata.com, http://www.osdata.com/system/physical/charcode.htm Visited 01 January 2008.

[26] "Gray code", National Institute of Standards and Technology http://www.nist.gov/dads/HTML/graycode.html Visited 01 January 2008.

You can find codes for most any type of data. For example, HTML has established accepted hexadecimal codes for colors.

Positional Number Systems

The idea of positional number systems was the topic of the first seven chapters of *Liber abaci* (1202 A.D.) by Leonardo Fibonacci[27]. Leonardo Fibonacci, also known as Leonardo of Pisa or Leonardo Pisano, was born approximately 1170 A.D., and died after 1240.

A **positional number system** is a series number system in which **only the coefficients c_k are recorded.** For most positional number systems you will work with, the coefficient in position number **k** multiplies the corresponding function numbered **k**. This is true for function or power series number systems. The common example is:

$$\ldots c_4 c_3 c_2 c_1 c_0 . c_{-1} c_{-2} \ldots$$

If the base of the number system cannot be assumed from context, it must be stated explicitly.

A positional number system requires an explicit symbol for zero as a place holder.

Ancient Greek and Hebrew, and Roman numerals, had no zero. Greek and Hebrew used their alphabets as numerals. All had to work hard to extend the range of numbers that could be represented as their applications diversified, and interest in computation became more sophisticated. What is surprising is that the Roman numeral system remained in daily use for so long.

Decimal Number System

In a decimal number, each digit has a place value. In elementary school, we learned that the position of a digit in a decimal number has meaning. In the number 9874, the digit 4 is in the one's place, the digit 7 is in the tens place, the digit 8 is in the hundred's place, and the digit 9 is in the thousand's place. We need to generalize the concept, so we write it more formally.

10^3	10^2	10^1	10^0	← place values in exponent form
1,000	**100**	**10**	**1**	← place values
9	**8**	**7**	**4**	← digits

9 8 7 4 is the same as

(**9** x **1000**) + (**8** x **100**) + (**7** x **10**) + (**4** x **1**)

This is the sum of the digits times their place values.

We also know that the number 009874 is the same number as 9874. We can write as many zeros to the left end of a number as we want and the number does not change value. This is true for all numbers in a positional number system.

---

[27] *Encyclopaedia Britannica 2003*, Deluxe Edition CD-ROM

Banfill has a nice web site for review of decimal place values.[28]

Algebra Review of Exponents

To understand the positional number system, we need a few basic rules of algebra about powers or exponents.

1. Any number raised to the power zero is equal to one.

$$3^0 = 1, \qquad 27^0 = 1 \qquad a^0 = 1 \qquad x^0 = 1$$

2. Any number raised to the power one is equal to itself.

$$3^1 = 3, \qquad 17^1 = 17 \qquad a^1 = a \qquad x^1 = x$$

3. Any number raised to a positive integer power N that is greater than one is equal to N copies of the number multiplied together.

$$3^5 = 3 \times 3 \times 3 \times 3 \times 3 = 243$$
$$a^3 = a \times a \times a$$

4. $X^{-M} = \frac{1}{X^M}$ Notice the minus sign in the left hand term.

Positional Number System General Concept

Some people like the general concept of an idea first, and then look at details. Other people like detailed examples first, and then the general concept. If you like details first, skip to the topic on the Binary Number System. Come back to this topic after studying the hexadecimal system.

In a positional number system, the position of a digit within a number determines its value. Another term for "digit" is "coefficient". Position is numbered from a reference point, which is called the *decimal point* for a base 10 number. For whole numbers without the reference point explicitly represented, this is the right side of the number. The right-most digit of a whole number is in position zero.

Let $\mathbf{c_k}$ be the coefficient in position **k**, where the position is identified by the subscript. Let the letter **b** stand for the base of the system. The general form of a number in a positional number system is

$\mathbf{\ldots c_4 c_3 c_2 c_1 c_0 \,.\, c_{-1} c_{-2} \ldots}$ The period between coefficients $\mathbf{c_0 \,.\, c_{-1}}$ is called the "radix point". In base 10, this is the decimal point.

In the decimal number 84693, $c_4 = 8$, $c_3 = 4$, $c_2 = 6$, $c_1 = 9$, $c_0 = 3$

The value of this general number $c_4 c_3 c_2 c_1 c_0 \,.\, c_{-1} c_{-2} \ldots$ is given by:

$$(c_4 \times b^4) + (c_3 \times b^3) + (c_2 \times b^2) + (c_1 \times b^1) + (c_0 \times b^0) + (c_{-1} \times b^{-1}) + (c_{-2} \times b^{-2})$$

[28] J. Banfill, "Place Value – Lessons", AAA Math, (2004). http://www.aaamath.com/B/plc.htm. Visited 01 January 2008.

Notice the minus sign in the powers to the right of b^0. The radix point is located between the term with b^0 and the term with b^{-1}. We will concentrate on whole numbers.

In the decimal number 84693, the base is 10, and we have

$(8 \times 10^4) + (4 \times 10^3) + (6 \times 10^2) + (9 \times 10^1) + (3 \times 10^0)$
$= 80{,}000 + 4{,}000 + 600 + 90 + 3$

Binary Number System

In the binary (base **2**) number system, b = **2**. The general binary form is

$\mathbf{\ldots c_4 c_3 c_2 c_1 c_0} = \ldots (c_4 \times \mathbf{2}^4) + (c_3 \times \mathbf{2}^3) + (c_2 \times \mathbf{2}^2) + (c_1 \times \mathbf{2}^1) + (c_0 \times \mathbf{2}^0)$

$= \ldots (c_4 \times 16) + (c_3 \times 8) + (c_2 \times 4) + (c_1 \times 2) + (c_0 \times 1)$

The meaning of the binary number 1011_2 is shown in the table below. Recall that $2^3 = 2 \times 2 \times 2 = 8$.

$\mathbf{2}^3$	$\mathbf{2}^2$	$\mathbf{2}^1$	$\mathbf{2}^0$	← place values as powers of 2
8	4	2	1	← place values
1	**0**	**1**	$\mathbf{1}_2$	← coefficients

In the binary number system, each place value is double the previous place value, moving from right to left.

1 0 1 1$_2$ is the same as

(**1** x 8) + (**0** x 4) + (**1** x 2) + (**1** x 1) This is the sum of the digits multiplied by the place value,

= 8 + 0 + 2 + 1
= 11 in the decimal number system.

The place values of the four right-most bits are **8 4 2 1**. Recall that any decimal number from 0 to 15 can be represented by 4 bits. To do this by computation, determine what combination of 8, 4, 2, and 1 will sum to that number. Begin with 8 and work to smaller numbers. Add a number if the current sum plus the number is still less than or equal to the goal. Record a "1" if you use that number. Record a "0" if you do not use that number.

Example, 14 = 8 + 4 + 2. The number 14 can be represented by a 4-bit code by turning on (setting to 1) only the bits in positions representing 8, 4, and 2.

		8421	← Place values.
Therefore	14 = 8 + 4 + 2 =	1110_2	= 1110**B** = **%**1110

Other examples:

0 =	0000_2
7 = 4 + 2 + 1 =	0111_2
15 = 8 + 4 + 2 + 1 =	1111_2

Example: Convert the hexadecimal C to binary.

8421 ← Place values.

Therefore $C_{16} = 12_{10} = 8 + 4 =$ 1100_2

Example 1: The number **11 0100**$_2$ is

5	**4**	**3**	**2**	**1**	**0**	Position Number
2^5	2^4	2^3	2^2	2^1	2^0	Place Value
32	16	8	4	2	1	Place Value in Decimal
1	**1**	**0**	**1**	**0**	**0**	Coefficients
(**1** x **32**)	+ (**1** x **16**)	+ (**0** x **8**)	+ (**1** x **4**)	+ (**0** x **2**)	+ (**0** x **1**)	Sum of Products
32	+ 16	+ 0	+ 4	+ 0	+ 0	
					ANSWER	**= 54**

Example 2: The number **101 1001**$_2$ is

6	**5**	**4**	**3**	**2**	**1**	**0**	Position Number
2^6	2^5	2^4	2^3	2^2	2^1	2^0	Place Value
64	32	16	8	4	2	1	Place Value in Decimal
1	**0**	**1**	**1**	**0**	**0**	**1**	Coefficients
(**1**x**64**)	(**0**x**32**)	+ (**1**x**16**)	+ (**1**x**8**)	+ (**0**x**4**)	+ (**0**x**2**)	+ (**1**x**1**)	Sum of Products
64	+ 0	+ 16	+ 8	+ 0	+ 0	+ 1	
						ANSWER	**= 89**

Example 3: Convert **1 0 1 0 1 1**$_2$ to decimal.

5	4	3	2	1	0	← position number
2^5	2^4	2^3	2^2	2^1	2^0	← place value as power of base 2
32	16	8	4	2	1	← place value
1	**0**	**1**	**0**	**1**	**1**$_2$	← coefficients
(**1**x**32**)	+ (**0**x**16**)	+ (**1** x **8**)	+ (**0** x **4**)	+ (**1** x **2**)	+(**1** x **1**)	← sum of products
32	+ 0	+ 8	+ 0	+ 2	+ 1	← perform multiplication
					ANSWER	**= 43**

Hexadecimal Number System

In the hexadecimal (base **16**) number system, b = **16**. Therefore, the general hexadecimal form is

…c$_4$c$_3$c$_2$c$_1$c$_0$ = … (c$_4$ x **16**^{4}) + (c$_3$ x **16**^{3}) + (c$_2$ x **16**^{2}) + (c$_1$ x **16**^{1}) + (c$_0$ x **16**^{0})

= … (c$_4$ x 65536) + (c$_3$ x 4096) + (c$_2$ x 256) + (c$_1$ x 16) + (c$_0$ x 1).

The following chart allows you to convert from a hexadecimal digit to its equivalent decimal value.

Hex Digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 24. Counting in Hexadecimal

ACE your knowledge of hexadecimal. If you memorize A, C, and E are 10, 12, and 14, you can mentally determine B is 11, D is 13, and F is 15.

Recall that $16^3 = 16 \times 16 \times 16 = 4096$.

Example 1: Convert the hexadecimal number **$93AF_{16}$** to decimal.

Each digit in a hexadecimal number has a place value.

3	**2**	**1**	**0**	← position number
16^3	**16^2**	**16^1**	**16^0**	← place values as powers of 16
4,096	**256**	**16**	**1**	← **place value in decimal**
9	**3**	**A**	**F_{16}**	← hexadecimal coefficients
9	**3**	**10**	**15**	← **decimal coefficients**
(**9** x **4,096**)	+ (**3** x **256**)	+ (**10** x 16)	+ (**15** x **1**)	← sum of products
36,864	+768	+ 160	+15	← perform multiplication
			ANSWER	**= 37,807**

Compute the sum of products from the **"place value in decimal"** and the **"decimal coefficients"** rows.

Example 2: Convert the hexadecimal number **$4A2D_{16}$** to decimal.

3	**2**	**1**	**0**	← position number
16^3	**16^2**	**16^1**	**16^0**	← place values as powers of 16
4,096	**256**	**16**	**1**	← place value in decimal
4	**A**	**2**	**D_{16}**	← hexadecimal coefficients
4	**10**	**2**	**13**	← decimal coefficients
(**4** x **4,096**)	+ (**10** x **256**)	+ (**2** x **16**)	+ (**13** x **1**)	← sum of products
16,384	+2560	+ 32	+13	← perform multiplication
			ANSWER	**= 18,989**

Example 3: Convert the hexadecimal number **A B C 6**$_{16}$ to decimal.

3	**2**	**1**	**0**	← position number
16^3	**16^2**	**16^1**	**16^0**	← place value as power of base 16
4096	**256**	**16**	**1**	← place value
A	**B**	**C**	**6_{16}**	← hexadecimal coefficients
10	**11**	**12**	**6**	← decimal coefficients
(**10** x **4096**)	+ (**11** x **256**)	+ (**12** x **16**)	+ (**6** x **1**)	← sum of products
40,960	+ 2,816	+ 192	+ 6	← perform multiplication
			ANSWER	**= 43,974**

Octal Number System

In the octal (base **8**) number system, b = **8**. Therefore, the general octal form is

$\ldots c_4c_3c_2c_1c_0 = (c_4 \times 8^4) + (c_3 \times 8^3) + (c_2 \times 8^2) + (c_1 \times 8^1) + (c_0 \times 8^0)$

$= (c_4 \times 4096) + (c_3 \times 512) + (c_2 \times 64) + (c_1 \times 8) + (c_0 \times 1).$

Decode the meaning of the hexadecimal number **4273_8** as shown in the table below. Recall that $8^3 = 8 \times 8 \times 8 = 512$.

Each digit in an octal number has a place value.

3	**2**	**1**	**0**	← position number
8^3	**8^2**	**8^1**	**8^0**	← place values as powers of 8
512	**64**	**8**	**1**	← place value
4	**2**	**7**	**3_8**	← coefficients
(**4** x **512**)	+ (**2** x **64**)	+ (**7** x **8**)	+ (**3** x **1**)	← sum of products
2,048	+ 128	+ 56	+ 3	← perform multiplication
			ANSWER	**= 2,235**

If you see a digit greater than 7 in a number, you know it cannot be an octal number.

Fun Questions

How many symbols are in a *base 3* system?

What is the answer to "2 + 2 = ?" in *base 3*? (You may do the arithmetic in base 10 and convert the answer to base 3.)

Sexagesimal or Sexagenary System

The sexagesimal or sexagenary system is a base 60 positional number system. We use it for time, direction, and position. We apply it in navigation and astronomy. Our implementation is a variant of a system first used by the early Sumerians, where we allow decimal numbers as coefficients. To ensure we treat the number as base 60, we use

special symbols to indicate the place value. The place values are degrees, minutes, and seconds. We allow the coefficient for degrees to exceed 60.

3	2	1	0	← position number
	°	‘	“	← place value symbol
	degrees	minutes	seconds	← place value name
	60^2	60^1	60^0	← place values as powers of 60
	3600	**60**	**1**	← place value in base 10
	179	**59**	**59**	← coefficients in base 10
	+ (**179** x **3600**)	+ (**59** x **60**)	+ (**59** x **1**)	← sum of products
	644400	+ 3540	+ 59	← perform multiplication
			ANSWER	**= 647999” = 179° 59‘ 59“**

Nothing Matters

The symbol for “nothing”, which is 0 in the Hindu-Arabic notation, is necessary for efficient recording of numbers in a positional number system. Look again at the general form for a number written in positional notation.

$$c_4 c_3 c_2 c_1 c_0 \,.\, c_{-1} c_{-2} \ldots$$

Suppose that there is no symbol for “nothing” and you have a number without any coefficient for b^2. You would have to omit the term c_2 from your record of that number.

You can still have a number system using position to write a number in a canonical form, but you would also need to write each digit with its corresponding place value, or use different number symbols as coefficients of different place values.

$$(c_4 \times b^4) + (c_3 \times b^3) + (c_2 \times b^2) + (c_1 \times b^1) + (c_0 \times b^0) + (c_{-1} \times b^{-1}) + (c_{-2} \times b^{-2})$$

Explicitly writing the place value with the coefficient does not require an ordering of terms, although it does not prohibit ordering them. The number above has the same value as

$$(c_4 \times b^4) + (c_2 \times b^2) + (c_3 \times b^3) + (c_1 \times b^1) + (c_0 \times b^0) + (c_{-1} \times b^{-1}) + (c_{-2} \times b^{-2})$$

where $(c_3 \times b^3) + (c_2 \times b^2)$ were exchanged.

A system using different symbols for different place-values was used in Attic numerals described by Herodianus. Hebrew also used a system of assigning numerical values to letters of the alphabet.[29] Use of a finite set of different symbols as coefficients of different place values limits the number of numbers that can be represented.

[29] D. E. Smith, “Numerals”, Volume 16, page 612, *Encyclopaedia Britannica* (1965)

Conversion to Decimal

To convert a binary or hexadecimal number to decimal
1) Multiply each digit by its place value.
2) Sum the products.
3) The result is the decimal equivalent.

Conversion to Decimal Problems:
Show all your work.

Problem A1. Convert 110_2 to decimal.

		ANSWER

← place values as powers of 2
← place values
← coefficients
← sum of products
← perform multiplication
=

Problem A2. Convert $1\ 0110_2$ to decimal.
Problem A3. Convert 1 1010 1101**B** to decimal.
Problem A4. Convert **%**111 1001 to decimal
Problem A5. Convert $B4_{16}$ to decimal.
Problem A6. Convert $2C5F_{16}$ to decimal.

For the abstract thinkers:

Convert **x w y w z y** $\mathbf{x}_4$ to decimal, for the number system {w, x, y, z}.

Solution: First notice that this is a base-4 system. The set {w, x, y, z}. exactly 4 symbols, and the ordering is given. We can associate "w" with "0", "x" with "1", "y" with "2", and "z" with "3".

6	**5**	**4**	**3**	**2**	**1**	**0**
$\mathbf{4^6}$	$\mathbf{4^5}$	$\mathbf{4^4}$	$\mathbf{4^3}$	$\mathbf{4^2}$	$\mathbf{4^1}$	$\mathbf{4^0}$
x	**w**	**y**	**w**	**z**	**y**	$\mathbf{x_4}$
4096	**1024**	**256**	**64**	**16**	**4**	**1**
1	0	2	0	3	2	1

← position number
← place value as power of base 16
← digits
← place value in decimal
← digits in decimal

= (**x** x 4^6) + (**w** x 4^5) + (**y** x 4^4) + (**w** x 4^3) + (**z** x 4^2) + (**y** x 4^1) + (**x** x 4^0)
= (**1** x 4096) + (**0** x 1024) + (**2** x 256) + (**0** x 64) + (**3** x 16) + (**2** x 4) + (**1** x 1)
= 40,960 + 0 + 512 + 0 + 48 + 8 + 1
= $4{,}665_{10}$

Problem A7. Convert 3011_4 to decimal.

Note: Processors on many calculators are 4-bit processors. A unit of memory consisting of 4-bits is a "nibble", which is half a byte.[30]

Problem A8. Convert 3017_8 (octal = base 8) to decimal.

Converting From Decimal: Method of Successive Division

The method of successive division is used to convert from decimal to other base number systems. If you are rusty with long division, you can review it at http://www.aaamath.com/B/div55_x2.htm.[31] We begin by dividing the original number by the base that we are going to.

What we are after is the collection of remainders after the remainders are converted to digits in the new base number system. It is important to label each remainder so that we know how to arrange the remainders into our final answer. We label each remainder with a subscripted **R**, such as $\mathbf{R_3}$. "**R**" stands for "Remainder". The subscript identifies the position number of the remainder. We begin with the position zero remainder, $\mathbf{R_0}$.

We repeat dividing the quotient from the last step by the new base. We continue this process until we have zero for a quotient.

To form the answer, we use the remainders to build the result from right-to-left. Remainder $\mathbf{R_0}$ becomes the extreme right digit for an integer answer. Remainder $\mathbf{R_1}$ goes to the left of $\mathbf{R_0}$. The process continues until all the computed remainders have been used. The result looks like

$\mathbf{R_9R_8R_7R_6R_5R_4R_3R_2R_1R_0}$.

Converting Decimal Numbers (Base 10) to Binary (Base 2)

To convert a decimal number to another base

1) Divide the decimal number by the base and determine the quotient (**Q**) and remainder (**R**).
2) Divide the resulting quotient by the base and determine its quotient (**Q**) and remainder (**R**).
3) Repeat step 2 until the quotient (**Q**) is zero.
4) Write the remainders in the new base beginning with the first remainder computed in position $\mathbf{R_0}$. …$\mathbf{R_9R_8R_7R_6R_5R_4R_3R_2R_1R_0}$
5) The result from step 4 is the number converted to the new base.

[30] John J. Donovan, *Systems Programming*, "Machine Structure – 360 and 370", page 25, McGraw-Hill (1972).

[31] J. Banfill, "Dividing a 4-digit by 2-digit numbers", AAA Math (2004). http://www.aaamath.com/B/div55_x2.htm Visited 14 January 2008.

Example 1: Convert 87 to binary.

Step 0 Base = 2

	4	3	
2	8	7	= Q_0
	8		
	0	7	
		6	
		1	**= R_0**

→ **Step 1** Base = 2

	2	1	
2	4	3	**= Q_1**
	4		
	0	3	
		2	
		1	**= R_1**

Step 2 Base = 2

	1	0	
2	2	1	= Q_2
	2		
	0	1	
		0	
		1	**= R_2**

→ **Step 3** Base = 2

		5	
2	1	0	**= Q_3**
	1	0	
		0	**= R_3**

Step 4 Base = 2

	2	
2	5	= Q_4
	4	
	1	**= R_4**

→ **Step 5** Base = 2

	1	
2	2	**= Q_5**
	2	
	0	**= R_5**

Step 6 Base = 2

	0	
2	1	= Q_6
	0	
	1	**= R_6**

Stop only when the Quotient is zero.

Record the answer using remainders, starting in the column on the right end.

6	5	4	3	2	1	0	Position Number
R_6	**R_5**	**R_4**	**R_3**	**R_2**	**R_1**	**R_0**	← **Record remainders from right to left.**
1	**0**	**1**	**0**	**1**	**1**	**1**	Final Answer

The answer is $101\ 0111_2$ = 101 0111**B** = **%**101 0111

Example 2: Convert 101 from decimal to binary.

Step 0 Base = 2

		5	0	
2	1	0	1	= Q_0
	1	0		
		0	1	
			0	
			1	**= R_0**

→ **Step 1** Base = 2

	2	5	
2	5	0	**= Q_1**
	4		
	1	0	
	1	0	
		0	**= R_1**

Step 2
Base = 2

	1	2
2	2	5
	2	
	0	5
		4
		1

$= Q_2$
$= Q_3$
$= \mathbf{R_2}$

→ **Step 3**
Base = 2

		6
2	1	2
	1	2
		0

$= Q_3$
$= Q_4$
$= \mathbf{R_3}$

Step 4
Base = 2

3	
6	6
	6
	0

$= Q_4$
$= \mathbf{Q_5}$
$= \mathbf{R_4}$

→ **Step 5**
Base = 2

1	
3	3
	2
	1

$= Q_5$
$= Q_6$
$= \mathbf{R_5}$

Step 6
Base = 2

0	
1	1
	0
	1

$= \mathbf{Q_7}$
$= Q_6$
$= \mathbf{R_6}$

Stop only after the Quotient is zero.

Record the answer using remainders, starting in the column on the right end.

6	5	4	3	2	1	0	Position Number
$\mathbf{R_6}$	$\mathbf{R_5}$	$\mathbf{R_4}$	$\mathbf{R_3}$	$\mathbf{R_2}$	$\mathbf{R_1}$	$\mathbf{R_0}$	← **Record remainders from right to left.**
1	**1**	**0**	**0**	**1**	**0**	**1**	Final Answer

The answer is $110\ 0101_2$ = 110 0101**B** = **%**110 0101 = 101_{10}

Converting Decimal Numbers (Base 10) to Hexadecimal (Base 16)

To convert a decimal number to another base

1) Divide the decimal number by the base and determine the quotient **Q** and remainder **R**. Convert the remainder to hexadecimal.
2) Divide the resulting quotient by the base and determine its quotient **Q** and remainder **R**. It is good to convert as soon as you see what remains.
3) Repeat step 2 until the quotient **Q** is zero.
4) Write the remainders in the new base beginning with the last remainder computed. …$\mathbf{R_9R_8R_7R_6R_5R_4R_3R_2R_1R_0}$
5) The result from step 4 is the number converted to the new base.

Example 1: Convert 3117 to hexadecimal.

```
Step 0              1  9  4   = Q1    →   Step 1              1  2   = Q2
Base = 16     | 3   1  1  7   = Q0        Base = 16     | 1   9  4   = Q1
                1   6                                     1   6
                ------                                    ------
                1   5  1                                      3  4
                1   4  4                                      3  2
                ---------                                     ----
                       7  7                                      2   = R1
                       6  4                                          = 2H
                       ----
                       1  3   = R0
                              = DH

Step 2              0   = Q3   Stop only when the Quotient is zero.
Base = 16     | 1   2   = Q2
                    0
                ------
                1   2   = R2   Remainder in decimal.
                ------
                        = CH   Remainder converted to hexadecimal.
```

2	1	0	Position Number
R_2	**R_1**	**R_0**	← **Record remainders from right to left.**
12	2	13	Remainders in decimal.
C	**2**	**D**	Final answer. Remainders in hexadecimal.

The answer is $C2D_{16}$ = C2DH = OxC2D = 3117_{10}

Example 2: Convert 3982 to hexadecimal.

```
Step 0              2  4  8   = Q1    →   Step 1              1  5   = Q2
Base = 16     | 3   9  8  2   = Q0        Base = 16     | 2   4  8   = Q1
                3   2                                     1   6
                ------                                    ------
                    7  8                                      8  8
                    6  4                                      8  0
                    ----                                      ----
                    1  4  2                                      8   = R1
                    1  2  8                                          = 8H
                    -------
                       1  4   = R0
                              = EH

Step 2              0   = Q3   Stop only when the Quotient is zero.
Base = 16     | 1   5   = Q2
                    0
                ------
                1   5   = R2   Remainder in decimal.
                ------
                        = FH   Remainder converted to hexadecimal.
```

2	1	0	Position Number
R_2	**R_1**	**R_0**	← **Record remainders from <u>right</u> to <u>left</u>.**
E	**8**	**F**	Final answer. Remainders in hexadecimal.

The answer is $E8F_{16}$ = E8F**H** = **Ox**E8F = 3982_{10}

Example 3: Convert 3786 to hexadecimal.

Step 0
Base = 16

```
       2  3  6     = Q1
   ------------
   | 3  7  8  6    = Q0
     3  2
   ------
        5  8
        4  8
      ------
        1  0  6
           9  6
      ---------
           1  0    = R0
                   = AH
```

→ **Step 1**
Base = 16

```
       1  4    = Q2
   ---------
   | 2  3  6   = Q1
     1  6
   ------
        7  6
        6  4
      ------
        1  2   = R1
               = CH
```

Step 2
Base = 16

```
        0    = Q3      Stop only when the Quotient is zero.
   ------
   | 1  4    = Q2
        0
   ------
     1  4    = R2      Remainder in decimal.
   ------
             = EH      Remainder converted to hexadecimal.
```

2	1	0	Position Number
R_2	**R_1**	**R_0**	← **Record remainders from <u>right</u> to <u>left</u>.**
E	**C**	**A**	Final answer. Remainders in hexadecimal.

The answer is ECA_{16} = ECA**H** = **Ox**ECA = 3786_{10}

Example 4: Convert 2748 to hexadecimal.

Step 0
Base = 16

```
       1  7  1     = Q1
   ------------
   | 2  7  4  8    = Q0
     1  6
   ------
     1  1  4
     1  1  2
   ---------
           2  8
           1  6
         ------
           1  2    = R0
                   = CH
```

→ **Step 1**
Base = 16

```
       1  0    = Q2
   ---------
   | 1  7  1   = Q1
     1  6
   ------
        1  1
           0
      ------
        1  1   = R1
               = BH
```

Step 2 0 = **Q_3** Stop only when the Quotient is zero.

Base = 16 | 1 0 = Q_2

0

1 0 = **R_2** Remainder in decimal.

= **AH** Remainder converted to hexadecimal.

2	1	0	Position Number
R_2	**R_1**	**R_0**	← **Record remainders from right to left.**
A	**B**	**C**	Final answer. Remainders in hexadecimal.

The answer is ABC_{16} = ABC**H** = **Ox**ABC = 2748_{10}

Practice: Convert the following decimal numbers to hexadecimal.

3243 = CAB_{16}	2989 = BADH	2750 = OxABE	3245 = CAD_{16}
4013 = FAD_{16}	4075 = FEBH	3564 = OxDEC	4077 = FED_{16}

Converting Decimal Numbers (Base 10) to Octal (Base 8)

To convert a decimal number to another base

1) Divide the decimal number by the base and determine the quotient **Q** and remainder **R**.
2) Divide the resulting quotient by the base and determine its quotient **Q** and remainder **R**.
3) Repeat step 2 until the quotient **Q** is zero.
4) Write the remainders in the new base beginning with the last remainder computed. …**$R_9R_8R_7R_6R_5R_4R_3R_2R_1R_0$**
5) The result from step 4 is the number converted to the new base.

Example: Convert decimal 3117 to octal.

Step 0 3 8 9 = **Q_1** → **Step 1** 4 8 = Q_2

Base = 8 | 3 1 1 7 = Q_0 Base = 8 | 3 8 9 = **Q_1**

2 4 3 2

7 1 6 9

6 4 6 4

7 7 5 = **R_1**

7 2

5 = **R_0**

Step 2		6	**= Q_3**	→	**Step 4**	0	**= Q_4**
Base = 8	4	8	= Q_2		Base = 8	6	= Q_3
	4	8				0	
		0	**= R_2**			6	**= R_3**

Stop only when the Quotient is zero.

Record the answer using remainders, starting in the column on the right end.

3	2	1	0	Position Number
R_3	**R_2**	**R_1**	**R_0**	**← Record remainders from right to left.**
6	0	5	5	Remainders in decimal.
6	**0**	**5**	**5**	Final answer. Remainders in octal.

The answer is 6055_8 = **O**6055 = 6055**Q** = 3117_{10}.

Notice the shape of the capital letter O (Oh) is slightly different than the number 0 (zero). Some printers put a slash through the number zero. The slashed zero is common world wide, except in the United States. It is common also in communications and in the military.

Converting Decimal Numbers (Base 10) to Base 4

In the base 4 example, an additional goal is to recognize that it does not matter what symbols are used in a number system as long as you know the collating sequence. In this example, the arbitrary symbol set {∃,∇,Δ,Σ} is used. You can make up your own symbols as long as you define the order.

To convert a decimal number to another base

1) Divide the decimal number by the base and determine the quotient **Q** and remainder **R**.
2) Divide the resulting quotient by the base and determine its quotient **Q** and remainder **R**.
3) Repeat step 2 until the quotient **Q** is zero.
4) Write the remainders in the new base beginning with the last remainder computed. …$R_9R_8R_7R_6R_5R_4R_3R_2R_1R_0$
5) The result from step 4 is the number converted to the new base.
6) Substitute the symbol used in the symbol set.

Example: Convert 3117 to {∃, ∇, Δ, Σ}

Step 0 Base = 4

```
      7  7  9   = Q1
3 ) 3  1  1  7   = Q0
    2  8
    ----
       3  1
       2  8
       -----
          3  7
          3  6
          ----
             1   = R0
                 = Ñ4
```

→

Step 1 Base = 4

```
      1  9  4   = Q2
7 ) 7  7  9   = Q1
    4
    ----
    3  7
    3  6
    -----
       1  9
       1  6
       -----
          3   = R1
              = S4
```

Step 2 Base = 4

```
      4  8     = Q3
1 ) 1  9  4    = Q2
    1  6
    ----
       3  4
       3  2
       -----
          2   = R2
              = D4
```

→

Step 3 Base = 4

```
      1  2     = Q4
4 ) 4  8     = Q3
    4
    ----
    0  8
       8
       -----
          0   = R3
              = $4
```

Step 4 Base = 4

```
      3     = Q5
1 ) 1  2    = Q4
    1  2
    ----
       0   = R4
           = $4
```

→

Step 5 Base = 4

```
      0     = Q6
3 ) 3     = Q5
    0
    ----
    3   = R5
        = S4
```

Stop only when the Quotient is zero. Record the answer using remainders, starting in the column on the right end.

5	4	3	2	1	0	Position Number
R_5	R_4	R_3	R_2	R_1	R_0	← **Record remainders from right to left.**
S	$	$	D	S	$Ñ_4$	Final answer.

Checklist for Converting from Decimal to Another Base Number System

- Select correct base.
- Divide by base.
- Remainder becomes a digit.
- Convert the remainder to a digit of the target number system base.
- Divide into new quotient. Do not divide into remainder.
- Don't stop dividing until quotient is zero.
- Build the final answer with the position-zero remainder at the right end.
- Identify number system base with correct notation. Allow any of the styles.
  - Binary: **%** nnnn nnnn$_2$ nnnn**B** nnnn**b**
  - Octal: **O**mmmm mmmm$_8$ mmmm**Q**
  - Hexadecimal: **Ox**NNNN NNNN$_{16}$ NNNN**H** NNNN**h**
- Final answer correct.

You can check you answer by converting the answer back to decimal.

Conversion From Decimal Problems

The purposes of this homework set are to give you experience in doing conversions, and to compare conversions of the same starting number in different bases. You will learn it is easier to convert from decimal to an intermediate large base, and then use table look-up to convert from the intermediate base to a smaller final base. This procedure results in fewer mistakes by reducing the number of divisions or multiplications.

Problem B1. Convert from decimal to binary. Show all your work. Summarize your answers in Table 27 below.

a. Convert 31 from decimal to binary. Record the answer in column A, row 1.
b. Convert 99 from decimal to binary. Record the answer in column A, row 2.
c. Convert 165 from decimal to binary. Record the answer in column A, row 3.
d. Convert 169 from decimal to binary. Record the answer in column A, row 4.

Problem B2. Convert from decimal to hexadecimal. Show all your work. Summarize your answers in Table 27 below.

a. Convert 257 from decimal to hexadecimal. Record the answer in column B, row 5.
b. Convert 445 from decimal to hexadecimal. Record the answer in column B, row 6.
c. Convert 479 from decimal to hexadecimal. Record the answer in column B, row 7.
d. Convert 708 from decimal to hexadecimal. Record the answer in column B, row 8.

Learning goal for problems 3 and 4: After doing the conversions, observe that you can convert from decimal to binary by first converting from decimal to hexadecimal, and then from hexadecimal to binary. This is a recommended shortcut for manual conversions. Similarly, you can convert from decimal to hexadecimal by first converting from decimal to binary, and then from binary to hexadecimal.

Problem B3. Use table look-up to convert from binary to hexadecimal. Summarize your answers in Table 27 below.

a. Convert the binary number found in column A, row 1 to hexadecimal using table look-up. Record the answer in column B, row 1.
b. Convert the binary number found in column A, row 2 to hexadecimal using table look-up. Record the answer in column B, row 2.
c. Convert the binary number found in column A, row 3 to hexadecimal using table look-up. Record the answer in column B, row 3.
d. Convert the binary number found in column A, row 4 to hexadecimal using table look-up. Record the answer in column B, row 4.

Problem B4. Use table look-up to from hexadecimal to binary. Summarize your answers in Table 27 below.

a. Convert the hexadecimal number found in column B, row 5 to binary using table look-up. Record the answer in column A, row 5.
b. Convert the hexadecimal number found in column B, row 6 to binary using table look-up. Record the answer in column A, row 6.
c. Convert the hexadecimal number found in column B, row 7 to binary using table look-up. Record the answer in column A, row 7.
d. Convert the hexadecimal number found in column B, row 8 to binary using table look-up. Record the answer in column A, row 8.

Problem B5. Convert from decimal to base 4. Summarize your answers in Table 27 below.

a. Convert 31 from decimal to base 4. Record the answer in column C, row 1.
b. Convert 99 from decimal to base 4. Record the answer in column C, row 2.
c. Convert 165 from decimal to base 4. Record the answer in column C, row 3.
d. Convert 169 from decimal to base 4. Record the answer in column C, row 4.

Problem B6. Convert from decimal to octal (base 8). Summarize your answers in Table 27 below.

a. Convert 31 from decimal to octal. Record the answer in column D, row 1.
b. Convert 99 from decimal to octal. Record the answer in column D, row 2.
c. Convert 165 from decimal to octal. Record the answer in column D, row 3.
d. Convert 169 from decimal to octal. Record the answer in column D, row 4.

Conversion Between Base 4, Binary, and Hexadecimal

Counting in base 4 is like counting in other number systems, except you have only 4 digits to count with, {0, 1, 2, 3}.

- Each base-4 digit corresponds exactly to **two** binary digits.
- Each hexadecimal digit corresponds exactly to **two** base-4 digits.

Do conversions by inspection (table look-up).

Table 25. Equivalent Numbers in Base 4 and Octal

Decimal	Hexadecimal	Binary	
Base 10	Base 16	Base 2	Base 4
0	0	0000	00
1	1	0001	01
2	2	0010	02
3	3	0011	03
4	4	0100	10
5	5	0101	11
6	6	0110	12
7	7	0111	13
8	8	1000	20
9	9	1001	21
10	A	1010	22
11	B	1011	23
12	C	1100	30
13	D	1101	31
14	E	1110	32
15	F	1111	33

Examples

a. Convert FAD2B**H** to base-4.

Hexadecimal	F	A	D	2	B
Base 4	33	22	31	02	23

The answer is 3322310223_4

b. Convert **%**110010011 to base-4. Note that you need to pad the binary number on the left.

Binary	01	10	01	00	11
Base 4	1	2	1	0	3

The answer is 12103_4

c. Convert 323223101_4 from base-4 to hexadecimal. Note that you need to pad the base-4 number on the left.

Base-4	03	23	22	31	01
Hexadecimal	3	B	A	D	1

The answer is **0x**3BAD1

d. Convert 211034 from base-4 to binary.

Base-4	2	1	1	0	3
Binary	10	01	01	00	11

The answer is 1001010011**B**

Problem B7. Do conversions from hexadecimal to base-4 by inspection (using table look-up). Summarize your answers in Table 27 below.

a. Convert the hexadecimal number in column B, row 5 to base-4 by inspection. Record the answer in column C, row 5.
b. Convert the hexadecimal number in column B, row 6 to base-4 by inspection. Record the answer in column C, row 6.
c. Convert the hexadecimal number in column B, row 7 to base-4 by inspection. Record the answer in column C, row 7.
d. Convert the hexadecimal number in column B, row 8 to base-4 by inspection. Record the answer in column C, row 8.

Conversion Between Octal and Binary

Octal is base 8. Counting in base 8 is like counting in other number systems, except you have only 8 digits to count with, {0, 1, 2, 3, 5, 6, 7}. Each octal digit corresponds exactly to three binary digits. Do conversions by inspection (table look-up).

Table 26. Equivalent Numbers in Octal

Decimal	Binary	Octal
Base 10	Base 2	Base 8
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7

To convert from binary to octal, the first step is to group bits into groups of **three** binary digits each. It is important to start the grouping from the right end. To convert from octal to binary, replace each octal digit with three binary digits.

Examples

1. Convert 111110001011002 to octal. Note that the binary number has to be padded on the left end to finish a group of 3 bits.

Binary	011	111	000	101	100
Octal	3	7	0	5	4

The answer is 37054_8

2. Convert 736258 to binary

Octal	1	3	5	7	2
Binary	001	011	101	111	010

The answer is %1011101111010

Problem B8. Each group of three binary digits is exactly one octal digit. Summarize your answers in Table 27 below.

a. Convert the binary number in column A, row 5, to octal by inspection. Record the answer in column D, row 5.
b. Convert the binary number in column A, row 6, to octal by inspection. Record the answer in column D, row 6.
c. Convert the binary number in column A, row 7, to octal by inspection. Record the answer in column D, row 7.
d. Convert the binary number in column A, row 8, to octal by inspection. Record the answer in column D, row 8.

Table 27. Decimal Conversion Solutions

	Part →	A	B	C	D
Row	**Decimal**	**Binary**	**Hexadecimal**	**Base 4**	**Octal**
1	**31**				
2	**99**				
3	**165**				
4	**169**				
5	**257**				
6	**445**				
7	**479**				
8	**708**				

NUMBER SYSTEM CONVERSION PROBLEMS

Problem 1. Convert the following decimal numbers to binary. Show all your work.
a. 535 **b.** 240 **c.** 61 **d.** 202

Problem 2. Convert the following numbers to hexadecimal. Show all your work.
a. 37 **b.** 222 **c.** 168 **d.** 959

Problem 3. Convert the following numbers to decimal. Show all your work.
a. 11010111_2 **b.** %01111001 **c.** EB_{16} **d.** Ox7F

Strings

A string is an ordered sequence of characters. Call the first character of a string the head, and the last character of a string the tail. (This terminology is not standardized.) One property of a string is its length. For example, the length of the string, "My favorite string?" is 19. Spaces count as characters. A very special string is the "null string", which is a string of zero length, such as "". Notice that there are no characters between the quotation marks.

A string can be either a fixed length string or a variable length string. A fixed length string is always the same length, even if all its characters are blanks. Fixed length strings are efficiently treated as an array, and they are easy to sort.

Variable length strings are handled in one of two ways. One way is to record the string length as one field of the string. The other way is to designate one character as a "null character" that can be used to identify the end of a string. The internal code used is often just the byte consisting of all zero bits, which is the null character. The null character does not count as one of the characters when computing the length of a string. A string that consists only of the null character is a string of zero length, and it is called the "null string".

For a string that includes a "length" field, that field is not shown to the outside world. The length field is the first field retrieved from memory when working with that string. This method allows use of a block transfer instruction from memory to the CPU, which is a fast way of moving data. The number is placed into a register and is used as a counter. The maximum length allowed is limited by the largest number that can fit into the length field. For many applications, this is very efficient. It is common for text fields in a data base to be limited to 255 characters. This leaves one byte out of 256 to be used as a length, and a single byte can hold numbers in the range [0,255]. It is not a useful string format if you need very long strings, such as the length of a book, or the source code for a computer program.

Example 1: "This string is here." The first field of this string contains the length of the string, which is "20" for this example. The symbol ␣ is the customary symbol used to identify a blank when it must be graphically displayed.

20	T	h	i	s	␣	s	t	r	i	n	g	␣	i	s	␣	h	e	r	e	.

Example 2: "This string is here." This is a string of length 20. The last symbol is the lower case Greek letter lambda (λ), which corresponds to the lower case Latin letter "l" (el). It is customary to use this symbol to represent the null character when documenting the end of a string. The actual code stored usually is a byte of all zero bits, not the character code for λ.

T	h	i	s	␣	s	t	r	i	n	g	␣	i	s	␣	h	e	r	e	.	λ

A language using this convention can store a sequence of strings, separating them by l . For example: Go Navy! Beat Army! (If you are Army, I expect you to reverse the example.)

G	o	␣	N	a	v	y	!	l	B	e	a	t	␣	A	r	m	y	!	l

String Operations

Operations upon strings, and symbols used to represent these operations, are defined by the programming language being used. At the hardware level, operations that affect strings are fetch, store, block move, comparison.

Length

The length operator gives the length of a string, not including the null character (if it has one). Often, this operation is abbreviated as LEN. The result is an integer value.

Example 1:

x =

H	o	n	o	r	l

LEN(x) = 5

Example 2:

y =

L	o	v	e	␣	y	o	u	r	␣	n	e	i	g	h	b	o	r	l

LEN(y) = 18

Concatenation

Concatenation is the operation of taking two separate strings, x and y, and making one string z by appending the second string to the first string. Example:

x =

B	o	y	␣	l

y =

S	c	o	u	t	s	l

Then z = x & y is

z =

B	o	y	␣	S	c	o	u	t	s	l

Truncation

Truncation is the operation of taking only the first N characters of a string, discarding all remaining characters. The operation requires specifying the length N. If the string is shorter than N to begin with, the string is unaltered. When N is counted from the left side, the operation is often called LEFT. When N is counted from the right side, the operation is often called RIGHT.

LEFT Truncation

Example 1: Let

x = | r | a | i | n | i | n | g | l |

Let y = LEFT(x,4)

y = | r | a | i | n | l |

Example 2: Let

x = | E | l | i | j | a | h | l |

Let y = LEFT(x,8)

y = | E | l | i | j | a | h | l |

Since x is shorter than the truncation length, 8, nothing is taken away from x. Therefore, y = x.

RIGHT Truncation

Example 1: Consider the palindrome

x = | M | A | D | A | M | I | M | A | D | A | M | l |

Let y = RIGHT(x,4)

y = | A | D | A | M | l |

Example 2: Let

x = | C | a | t | l |

Let y = RIGHT(x,4)

y = | C | a | t | l |

Since x < 4, no truncation takes place. y = x.

Extraction

Extraction fetches N characters from a string, beginning with a specified beginning point k. The function is often called MID, for “middle” of a string. An error condition exists if the operation cannot be performed.

Example 1: Let the MID function have the format MID(x=string,k=start,N=size), and let

x = | M | A | D | A | M | I | M | A | D | A | M | l |

Let y = MID(x,6,2). Then

y = | I | M | l |

Comparison

Strings are compared one character at a time. The primitive comparison operators are “equal” (=) and “not equal” (¹). For two strings to be equal, they must have the same length, and each character must be equal. A common format is:

“If x = y”

The outcome of a test is “True” or “False”.

Many computer languages permit unequal comparisons:

Table 28. Comparison Operators

Equal (=)	Not equal (¹)
Less than (<)	Less than or equal (£)
Greater than (>)	Greater than or equal (³)

The binary value of the character code is used for the comparison. Two strings are compared one character at a time. The default sequence is to begin the comparison at the string head, and continue toward the string tail until the comparison result is determined. The comparison of characters terminates when the first unequal comparison is found, or when the end of the shorter string is found.

String Direction

Sometimes the notion of string direction is meaningful, such as strings containing human language. The direction of a string is context dependent, such as in a crossword puzzle. European languages are normally read from left to right. Some languages, such as Hebrew and Arabic, are read from right to left. Some languages may be written from top to bottom, such as Chinese and Japanese. String direction is not usually stored with a string, but this can be included in the data structure if necessary.

Arabic influence is also seen in the way numbers are written, which have the reference point for whole numbers on the right end. In English, we determine the size and partitioning of the number by first scanning the number starting at the radix (decimal) point, and we proceed from right to left in order to select the proper words to describe the number.

Inequality Comparisons

The inequality comparisons are used for searching and sorting. Most sorting in business data processing is done on character data.

Example 1:

x	=	B	l	u	e	l
y	=	B	l	e	w	l

If x > y

The last comparison performed is x=u versus y=e. Conclude: x > y. Therefore, the answer is “True”.

Example 2:

x	=	S	w	i	m	l			
y	=	S	w	i	m	m	i	n	g	l

If x > y

Only 4 comparisons are made. Conclude: x < y because x and y are equal for the first 4 characters and LEN(x) < LEN(y). The answer is “False”.

String Search

The string search is a special case of the comparison. This operation searches string **x** to see if string **y** is contained in it. If the string is found, the starting position in string **x** is reported. This is sometimes called the FIND operation. The search stops with the first match found. Some versions allow specification of a search starting point, N. For example, k = FIND(**x**,**y**,N). If the search string is not found, the returned value depends on the language. Often, the value returned is a negative number.

Example 1: Let

x	=	B	l	e	s	s	e	d	␣	a	r	e	␣	t	h	e	␣	p	o	o	r	l
y	=	l	e	s	s	l																

k = FIND(x,y)

The result is k = 2.

Some search programs permit restriction to locating only whole words. If that restriction were applied in the above example, the result would be k = -1, indicating that the search string was not found in the target string. Restricting a search to whole words is more difficult. After locating a candidate match, the target must be examined to determine if the matched string was merely contained in a longer word, or was a complete word. What are indicators that a whole word was found?

Table 29. Detecting a Whole Word in a String

Left Side of Target Match	Right Side of Target Match
Preceded by a blank or at the head of the string.	Followed by a blank or end-of-string code.
Preceded by a quotation mark that is preceded by a blank.	Followed by punctuation mark that is followed by a blank or end-of-string code. Punctuation marks are: period, comma, colon, semicolon, quotation mark.
Preceded by a parenthesis or bracket.	Followed by a parenthesis or bracket.
Preceded by a forward or backward slash.	Followed by a forward or backward slash.
Preceded by a tab or end-of-line control code.	Followed by a tab or end-of-line control code.

This is not an exhaustive list. The search is further complicated by other factors:

- exclude or skip over non-printing control codes, such as used to control the font.
- permit matching to upper and lower case letters.
- permit use of wild card characters in the search string. A wild card character can specify an exact unknown character position (often indicated in the search string by a question mark, ?), or an indefinite number of unknown characters at a particular position (often indicated in the search string by an asterisk, *).

Example 2: Let

x =	B	l	e	s	s	e	d	␣	a	r	e	␣	t	h	e	␣	p	o	o	r	l
y =	e	␣	l																		

k = FIND(x,y,13)

The result is k = 15

Example 3: Let

x =	B	l	e	s	s	e	d	␣	a	r	e	␣	t	h	e	␣	p	o	o	r	l
y =	i	s	l																		

k = FIND(x,y)

The result is k = -1

Compound Expressions

It is common to use the results of one operation as a parameter or string argument of another expression. This is composition of functions. As in evaluation of algebra expressions, evaluate the innermost function(s) first.

Example 1: Find Again and extract.

x =	B	e	␣	b	r	a	v	e	.	␣	B	e	␣	s	t	r	o	n	g	.	␣
	B	e	␣	r	e	a	d	y	.	␣	B	e	␣	a	l	e	r	t	.	l	
y =	B	e	l																		

Find:

z = MID(x, FIND(x, y, FIND(x,y) + LEN(y)) + LEN(y) + 1, 6)

z =																					

Solution:

n = FIND(x, y, FIND(x,y) + LEN(y))

j = FIND(x,y) = 1

k = LEN(y) = 2

n = FIND(x, y, **j + k**)

n = FIND(x, y, **1 + 2**) = FIND(x, y, 3) = 11

z = MID(x, 11 + LEN(y) + 1, 6)

z = MID(x, 11 + 2 + 1, 6) = MID(x, 14, 6)

z = | s | t | r | o | n | g | l | | | | | | | | | | | | | |

String Function Problems

Problem 1. Length. Let

x = | B | l | e | s | s | e | d | ␣ | a | r | e | ␣ | t | h | e | ␣ | p | o | o | r | l |

Find k = LEN(x) =

Problem 2. Concatenation. Let

x = | B | e | ␣ | l |

y = | P | r | e | p | a | r | e | d | l |

Concatenate x and y. z = x & y.

z = | | | | | | | | | | | | | | | | | | | | |

Problem 3. Left. Let

x = | B | l | e | s | s | e | d | ␣ | a | r | e | ␣ | t | h | e | ␣ | p | o | o | r | l |

Find y = Left(x,5).

y = | | | | | | | | | | | | | | | | | | | | |

Problem 4. Right. Let

x = | B | l | e | s | s | e | d | ␣ | a | r | e | ␣ | t | h | e | ␣ | p | o | o | r | l |

Find y = Right(x,8).

y = | | | | | | | | | | | | | | | | | | | | |

Problem 5. MID. Let

x = | B | l | e | s | s | e | d | ␣ | a | r | e | ␣ | t | h | e | ␣ | p | o | o | r | l |

Find y = MID(x,14,2)

y = | | | | | | | | | | | | | | | | | | | | |

Problem 6. Comparison. Let

x = | B | l | e | s | s | e | d | ␣ | a | r | e | ␣ | t | h | e | ␣ | p | o | o | r | l |

y = | B | l | e | s | s | e | d | ␣ | a | r | e | ␣ | t | h | e | ␣ | m | e | e | k | l |

If x < y

Give answer as “True” or “False”.

Problem 7. Search.

x = | B | e | h | o | l | d | , | ␣ | I | ␣ | b | r | i | n | g | ␣ | g | l | a | d | l |

y = | h | o | l | d | l |

Find k = FIND(x,y)

Problem 8. Search.

x =	t	i	d	i	n	g	s	.	␣	Y	o	u	␣	s	h	a	l	l	l
y =	y	o	u	l															

Search only for whole words; permit upper or lower case.

Find k = FIND(x,y)

Problem 9. Compound Function

x =	B	l	e	s	s	e	d	␣	a	r	e	␣	t	h	e	␣	p	o	o	r	l
y =	P	r	e	p	a	r	e	d	l												

Find:

z = LEFT(x,5) & MID(x,13,3) & y

z =																				

Problem 10. Compound Function

x =	B	e	␣	b	r	a	v	e	.	␣	B	e	␣	s	t	r	o	n	g	.	␣
	B	e	␣	r	e	a	d	y	.	␣	B	e	␣	a	l	e	r	t	.	l	
y =	B	e	l																		

Find:

z = RIGHT(x, LEN(x)–LEN(y)–FIND(x,y, FIND(x,y, FIND(x,y)+LEN(y))+LEN(y)))

z =																				

Storage of Multibyte Words: Big Endian, Little Endian

IBM mainframes and Intel processors store multibyte data in different orders. Consider a 4-byte word written in hexadecimal: AB CD EF 01.

Table 30. Little, Middle, and Big Endian Definitions

Big Endian (Big End In)	AB CD EF 01 ‖ (Most Significant Byte) Lowest Address	IBM System 390 AIX Sun SPARC Solaris IBM PowerPC MacIntosh HP Precision Architecture HP-UX SCSI devices network protocols TCP/IP packets W3C
Little Endian (Little End In)	01 EF CD AB ‖ (Least Significant Byte) Lowest Address	Intel 486, Pentium DEC Alpha RISC Microsoft Windows IBM OS/2 VAX/VMS Digital Unix Intel ABI PCI bus
Middle Endian	BA DC FE 10 A and B are reversed.	Occasionally used with Packed Decimal (BCD)

Little Endian format could have some advantages on a stack-oriented machine. By fetching the lowest remaining order byte next and placing it on a stack, the stack ends up with the data in Big Endian order after the transfer is complete. The most significant byte is on top.

If you share data between machines, data order becomes critical. It affects storage of all multiple byte data, whether to primary storage (memory), or secondary storage (tape, disk). This includes short integers, long integers, floating point, double precision floating point, extended precision floating point, complex, double precision complex, extended precision complex, and string data. It also affects details of hardware interfaces. For example, MacIntosh has a Big Endian 16-bit RGB pixel format. There also exists a Little Endian 16-bit RGB pixel format.

These differences affect files, data structures, and arrays. It affects the storage of bytes for Unicode characters. It affects even data operations, such as searching and sorting. These require the ability to make comparisons.

Logic Operations

Logic operations accept one or two binary inputs and produce one binary output. Each operation is defined through use of a table. The table is called a truth table because a zero (0) is called a False (F) value, and a one (1) is called a True (T) value. Logic operations are also called switching operations, combinational logic, Boolean logic, or Boolean algebra. The rules for logic were developed by English mathematician George Boole (1815 – 1864).

The logic operations are used in Boolean algebra, logic, and set theory. Boolean algebra is used by binary computers. Logic is used in law, philosophy, and mathematics. Set theory is used in statistics and mathematics. Each of these disciplines has its own names and symbols for these operations. The rules of algebra are the same.

The most basic logic operations are the **AND**, **OR**, and **NOT** functions. When these functions are implemented in electronics, they are called "gates". All other functions can be synthesized from these. These are already familiar to you through your use of every day language.

For multi-bit variables, the operation is done for each bit position, independently of all other bit positions. It is important to right-align variables before beginning an operation. There are not carries or borrows, unlike arithmetic operations.

The symbols for **AND**, **OR**, and **NOT** which are given below are used in equations. The logical **AND** function in Boolean logic corresponds abstractly to the multiplication function in algebra. The logical **OR** function in Boolean logic corresponds abstractly to the addition function in algebra. The logical **NOT** function is shown by a bar above the variable. American National Standards Institute (ANSI) Circuit symbols from *OrCAD Capture* are shown.[32]

AND Operation

Suppose your friend says he is going swimming and flying. That tells you he is going to do both activities. If you ask for a peanut butter and jelly sandwich, you want peanut butter, and you also want jelly. If you are asked if you got peanut butter and jelly, and you actually got both, the answer is True. If you did not actually receive both, the answer is False. The **AND** operation is summarized in a truth table. Let peanut butter be represented by the variable X, and let jelly be represented by the variable Y.

Table 31. Truth Table for AND Operator, Version 1

AND	**X = F = 0**	**X = T = 1**
Y = F = 0	F = 0	F = 0
Y = T = 1	F = 0	T = 1

The logic symbol for the **AND** function is the arithmetic multiplication symbol. The following are equivalent:

[32] OrCAD Capture Demonstration 9.10.93 (26 April 1999).

Equation 1. $x\,AND\,y = x \bullet y = xy$

The ANSI standard circuit symbol for the **AND** function is: 

The inputs are applied at pins 1 and 2 on the left side, as the symbol is presently oriented. The output is taken at point 3 on the right side. Note the straight edge on the input end of the symbol. The output end is rounded.

In this truth table, the inputs to the **AND** operator are given in the top row and the left column. The outcome is given in the other cells.

There is another version of this same table.

Table 32. Truth Table for AND Operator, Version 2

Inputs		Output
X	**Y**	**X AND Y**
0	0	0
0	1	0
1	0	0
1	1	1

There is one more version of this same table. This version will be extended with other operators.

Table 33. Truth Table for AND Operator, Version 3

	Inputs			
Function	**XY = 00**	**XY = 01**	**XY = 10**	**XY = 11**
X AND Y	0	0	0	1

Example: Perform the operation 1100 **AND** 1010.

X	1	1	0	0
Y	1	0	1	0
X AND Y	**1**	**0**	**0**	**0**

AND Applications

The **AND** operation is used as a mask to isolate one or more bits from a larger collection of bits, such as extracting a subnet address in a network router application, or obtaining the binary value of a decimal digit represented by its ASCII, EBCDIC, or UNICODE character. It is also used to clear one or more bits in a larger collection of bits, as might be done to convert a lower case ASCII or UNICODE character to upper case, converting an upper case EBCDIC character to lower case, to clear a bit to zero in a compact data record, or to clear the sign bit to zero.

Example: Convert EBCDIC decimal character to binary.

Observe that the binary value is contained in the lower four bits of the EBCDIC code. For example, the EBCDIC code for the character printed as "**9**" is F9**H**. In binary, this code is 11111001**B**. To obtain the binary value of 9, do the **AND** operation using the mask of 00001111**B** = 0F**H**. The operation F9**H** **AND** 0F**H** = 09**H** in binary is computed bit by bit:

"9" = F9**H** =	1	1	1	1	1	0	0	1
Mask = 0F**H** =	0	0	0	0	1	1	1	1
Binary 9 = 09**H** =	0	0	0	0	1	0	0	1

Example: Convert an upper case EBCDIC letter to lower case.

Observe that bit #6 (where the right-most bit is bit #0) is "1" in the upper case character, but is "0" in the corresponding lower case character. For example, the EBCDIC code for the upper case character "**J"** is D1**H** = 11010001**B**. The EBCDIC code for the lower case character "**j"** is 91**H** = 10010001**B**. To clear bit #6, use the **AND** operation with the mask BF**H**. This is: D1**H** **AND** BF**H** = 91**H**.

EBCDIC "J" = D1**H** =	1	1	0	1	0	0	0	1
Mask = BF**H** =	1	0	1	1	1	1	1	1
EBCDIC "j" = 91**H** =	1	0	0	1	0	0	0	1

OR Operation

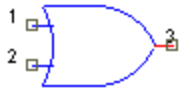
The operation "x **OR** y" is True if either x or y is True. This includes the case that x and y are both True. If you asked for red or blue crayons, this would be satisfied by getting either a red crayon, a blue crayon, or both a red and a blue crayon. If you are really thirsty, you will take Pepsi OR Coke. The truth table is:

Table 34. Truth Table for OR Operator, Version 1

OR	**X = F = 0**	**X = T = 1**
Y = F = 0	F = 0	T = 1
Y = T = 1	T = 1	T = 1

The logic symbol for the **OR** function is the arithmetic addition symbol. The following are equivalent:

Equation 2. $x\,OR\,y = x + y$

The ANSI standard circuit symbol for the **OR** function is: 

The inputs are applied at pins 1 and 2 on the left side, as the symbol is presently oriented. The output is taken at point 3 on the right side. Note the concave curve on the input side of the symbol. The output end looks like a bullet with a blunt point.

An alternate form for this same truth table is:

Table 35. Truth Table for OR Operator, Version 2

	Inputs			
Function	**XY = 00**	**XY = 01**	**XY = 10**	**XY = 11**
X OR Y	0	1	1	1

Example: Perform the operation 1100 OR 1010.

X	1	1	0	0
Y	1	0	1	0
X OR Y	**1**	**1**	**1**	**0**

OR Applications

The **OR** operation is used as a mask to set one or more bits in a collection of bits, such as obtaining the decimal ASCII, EBCDIC, or UNICODE character of a binary value between 0 and 9. It is also used to set one or more bits in a larger collection of bits, as might be done to convert an upper case ASCII or UNICODE character to lower case, converting an lower case EBCDIC character to upper case, to set a bit to one in a compact data record, or to set the sign bit to one.

Example: Convert the binary value 00000111**B** to the ASCII code for the character "7", which is 37**H**.

To do this, use the **OR** operation to insert the code 0011 in the upper half of the byte, using the mask 30**H**. 07**H** **OR** 30**H** = 37**H**.

7_{10} =	0	0	0	0	0	1	1	1
Mask = 30**H** =	0	0	1	1	0	0	0	0
ASCII "7" = 37**H** =	0	0	1	1	0	1	1	1

Example: Convert an EBCDIC character from lower case to upper case.

For example, convert EBCDIC character "t" to upper case "T". Note that bit #6 is turned off (zero) in the lower case letter, and turned on (one) in the upper case letter. The hexadecimal EBCDIC code for the character "t" is A3**H**. The code for "T" is E3**H**. The mask for conversion is 40**H**. The conversion is done using the logical **OR** operation, A3**H** **OR** 40**H** = E3**H**.

EBCDIC "t" = A3**H** =	1	0	1	0	0	0	1	1
Mask = 40**H** =	0	1	0	0	0	0	0	0
EBCDIC "T" = E3**H** =	1	1	1	0	0	0	1	1

NOT Operation

The **NOT** operator uses only one variable as an input, and is therefore called a *unary* operator. Another name for this is the *complement* operator. Whatever value the input has, the output is just the opposite. When applied to a full byte, all the zeros of the byte get changed to ones, and all the previous ones get changed to zeros. When a kid says, "I love school … NOT!", the kid is negating the sentence just spoken.

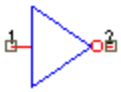
The truth table for the **NOT** operator is:

Table 36. Truth Table for NOT Operator

OR	**X = F = 0**	**X = T = 1**
NOT X	T = 1	F = 0

There are three logic symbols used for the **NOT** operator. One logic symbol for the **NOT** function is a bar over the variable. Another is an apostrophe following the variable. The third is a horizontal bar with the right end bent down, like a golf club. The following are equivalent:

Equation 3. $NOT\ x = \overline{x} = x' = \neg x$

The ANSI standard circuit symbol for the **NOT** function is:  The input is applied at pin 1 on the left side, as the symbol is presently oriented. The output is taken at point 2 on the right side. Note the small circle on the output side of the symbol. The body of the symbol is an equilateral triangle.

Example: Perform the operation **NOT** 1011.

X	1	0	1	1
NOT X	**0**	**1**	**0**	**0**

NOT Applications

The **NOT** operator can be used to treat a bit as a toggle switch. Suppose a push button light switch is "off". Pushing the switch changes it to "on". Pushing the switch again changes it back to "off". Let a bit represent the state of the switch. Let "off" be represented by zero, and let "on" be represented by one. The **NOT** operator toggles the bit between zero and one.

Applying the NOT operator to an integer word results in the one's complement of that word. It may be useful to convert a hexadecimal representation to binary before taking the complement, and then convert the result back to hexadecimal.

Example: Find the one's complement of A0F3**H**. **NOT** A0F3**H** = 5F0C**H**.

A0F3H =	1	0	1	0	0	0	0	0	1	1	1	1	0	0	1	1
NOT A0F3H =	0	1	0	1	1	1	1	1	0	0	0	0	1	1	0	0

Logic Operation Problems

Perform the following operations.

Problem 1. D6**H** AND AD**H**

Problem 2. A5**H** OR 61**H**

Problem 3. NOT 0F**H**

Boolean Algebra Theorems

Properties of Boolean algebra are summarized in the below table from Sloan.[33]

[33] Martha E. Sloan, *Computer Hardware and Organization: An Introduction*, Second Edition, Science Research Associates, Inc (1983). pp 26-27.

Table 37. Boolean Algebra Properties

Property	**OR** Version	**AND** Version
Involute	$\overline{\overline{x}} = x$	
Identity	x + 0 = x	x • 1 = x
Null element	x + 1 = 1	x • 0 = 0
Idempotent	x + x = x	x • x = x
Complement	$x + \overline{x} = 1$	$x \bullet \overline{x} = 0$
Commutative	x + y = y + x	x • y = y • x
Associative	x + (y + z) = (x + y) + z	x • (y • z) = (x • y) • z
Distributive	x + (y • z) = (x + y) • (x + z)	x • (y + z) = (x • y) + (x • z)
Absorption	x + (x • y) = x	x • (x + y) = x
Absorption	$x + (\overline{x} \bullet y) = x + y$	$x \bullet (\overline{x} + y) = x \bullet y$
Consensus	$(x \bullet y) + (\overline{x} \bullet z) + (y \bullet z)$ $= (x \bullet y) + (\overline{x} \bullet z)$	$(x + y) \bullet (\overline{x} + z)(y + z)$ $= (x + y) \bullet (\overline{x} + z)$
DeMorgan's Laws	$\overline{x + y} = \overline{x} \bullet \overline{y}$	$\overline{x \bullet y} = \overline{x} + \overline{y}$
DeMorgan's Laws	$\overline{(x \bullet z) + (y \bullet z)} = (\overline{x} + \overline{z}) \bullet (\overline{y} + \overline{z})$	$\overline{(x + z) \bullet (y + z)} = (\overline{x} \bullet \overline{z}) + (\overline{y} \bullet \overline{z})$

Exclusive OR (XOR) Operation

The Exclusive OR operation x **XOR** y is True if either x or y, but not both x and y, are True. The word "or" in "I will eat with chopsticks or a fork" probably means **XOR**. The language of logical operations is unambiguous, unlike English. The truth table is

Table 38. Truth Table for XOR Operator, Version 1

XOR	**X = F = 0**	**X = T = 1**
Y = F = 0	F = 0	T = 1
Y = T = 1	T = 1	T = 0

The logic symbol for the **XOR** function is the arithmetic addition symbol inside a small circle. The following are equivalent:

Equation 4 $x\,XOR\,y = x \oplus y = (x + y) \bullet (\overline{x \bullet y})$

The ANSI standard circuit symbol for the **XOR** function is: 

This is similar to the **OR** symbol. The inputs are applied at pins 1 and 2 on the left side, as the symbol is presently oriented. The output is taken at point 3 on the right side. Note the concave double curve on the input side of the symbol.

An alternative truth table is

Table 39. Truth Table for XOR Operator, Version 2

	Inputs			
Function	**XY = 00**	**XY = 01**	**XY = 10**	**XY = 11**
X XOR Y	0	1	1	0

Contrast the **XOR** operator with the **OR** operator. The difference between the two is in the outcome when X=1 and Y=1.

Example: Perform the operation 1100 XOR 1010.

X	1	1	0	0
Y	1	0	1	0
X XOR Y	**0**	**1**	**1**	**0**

Real Logic Components

It is easier to manufacture another family of logic components, the **NAND** and **NOR** functions. The **NAND** is the complement of the **AND** function, which is the NOT AND function. The **NOR** is the complement of the **OR** function, which is the NOT OR function. The **NOT** function can be constructed from either the **NAND** function or the **NOR** function. The circuit symbols are like the **AND** and **OR** functions, with the addition of a small circle at the output end. All logic circuits can be made with **NAND** and **NOR** gates. This will be demonstrated. In general, it is easier to learn the theory in terms of **AND**, **OR**, and **NOT**. It is easier to build chips using **NAND** and **NOR**.

NAND Operation

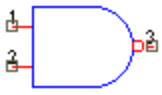
The operation "x **NAND** y" is True if either x or y is False. The truth table is:

Table 40. Truth Table for NAND Operator, Version 1

NAND	**X = F = 0**	**X = T = 1**
Y = F = 0	T = 1	T = 1
Y = T = 1	T = 1	F = 0

The logic symbol for the **NAND** function is the up arrow, $\uparrow$. The following are equivalent:

Equation 5. $x\,NAND\,y = x \uparrow y = \overline{x \bullet y}$

The ANSI standard circuit symbol for the **NAND** function is: 

This is like the **AND** function with a circle at the output end. The inputs are applied at pins 1 and 2 on the left side, as the symbol is presently oriented. The output is taken at point 3 on the right side.

An alternate form for this same truth table is:

Table 41. Truth Table for NAND Operator, Version 2

	Inputs			
Function	**XY = 00**	**XY = 01**	**XY = 10**	**XY = 11**
X NAND Y	1	1	1	0

Notice that the **NAND** gate can be used to construct the **NOT** function because 1 **NAND** y = **NOT** y.

Example: Perform the operation 1100 **NAND** 1010.

X	1	1	0	0
Y	1	0	1	0
X NAND Y	**0**	**1**	**1**	**1**

NOR Operation

The operation "x **NOR** y" is True if both x and y are False. The truth table is:

NOR	**X = F = 0**	**X = T = 1**
Y = F = 0	T = 1	T = 0
Y = T = 1	T = 0	F = 0

Table 42. Truth Table for NOR Operator, Version 1

The logic symbol for the **NOR** function is the "down" arrow, ↓. The following are equivalent:

Equation 6. $x \; NOR \; y = x \downarrow y = \overline{x + y}$

The ANSI standard circuit symbol for the **NOR** function is: 

This is like the OR function with a circle at the output end. The inputs are applied at pins 2 and 3 on the left side, as the symbol is presently oriented. The output is taken at point 1 on the right side.

An alternate form for this same truth table is:

Table 43. Truth Table for NOR Operator, Version 2

	Inputs			
Function	**XY = 00**	**XY = 01**	**XY = 10**	**XY = 11**
X NOR Y	1	0	0	0

Notice that the **NOR** gate can be used to construct the **NOT** function because 0 **NOR** y = **NOT** y.

Example: Perform the operation 1100 **NOR** 1010.

X	1	1	0	0
Y	1	0	1	0
X NOR Y	**0**	**0**	**0**	**1**

Logic Operation Summary

There are 16 combinatorial functions of two variables. These are summarized in the table below.

Table 44. Logic Operation Summary

						Output for XY Inputs			
Boolean		Logic		Set Theory		XY	X Y	XY	X Y
Operator	Symbol	Operation	Symbol	Operator	Symbol	00	01	10	11
ZERO	0	Null				0	0	0	0
AND	•	Conjunction	$X \wedge Y$	Intersection	$X \cap Y$	0	0	0	1
ONLY X		Inhibit				0	0	1	0
X	X					0	0	1	1
ONLY Y		Inhibit				0	1	0	0
Y	Y					0	1	0	1
XOR	$\oplus$	Mod 2 Addition				0	1	1	0
OR	+	Disjunction	$X \vee Y$	Union	$X \cup Y$	0	1	1	1
NOR	↓					1	0	0	0
COINCIDENCE	Á	Equivalence, Tautology	$X \Leftrightarrow Y$			1	0	0	1
NOT Y	$\overline{Y}$	Negation	$Y', \neg Y, \overline{Y}$	Complement	$Y', \overline{Y}, Y^C$	1	0	1	0
NOT ONLY Y		Imply	$Y \Rightarrow X$			1	0	1	1
NOT X	$\overline{X}$	Negation	$X', \neg X, \overline{X}$	Complement	$X, \overline{X}, X^C$	1	1	0	0
NOT ONLY X		Imply	$X \Rightarrow Y$			1	1	0	1
NAND	↑					1	1	1	0
ONE	1	Unity				1	1	1	1

Special definitions:

In the *imply* operation, $X \Rightarrow Y$, the variable on the left side is called the *antecedent*, and the variable on the right side is called the *consequent*.

Tautology: $X \Leftrightarrow Y$ Statement whose truth values are always true. This is equivalent to $(X \Rightarrow Y) \wedge (Y \Rightarrow X)$, or equivalently $(X \Rightarrow Y) \wedge (X \Leftarrow Y)$ where the implication arrow direction and order of variables are reversed in the second term.

Hexadecimal Addition

Hexadecimal addition is done just like decimal addition. The only difference is that the addition table is larger.

Table 45. Hexadecimal Addition Table

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

In the addition table above, the left digit of a two digit answer is a carry to the next column for addition. Addition is only defined for pairs of numbers.

Example 1. Add the hexadecimal numbers 7FAC and FBD using a 16-bit word. In decimal, this is 32684 + 4029 = 36713.

	"Throw Away" Column	Carry			
		1	1	1	
Original Number		**7**	**F**	**A**	**C**
Radix Complement	Add →		**F**	**B**	**D**
Final Result		**8**	**F**	**6**	**9**

Do the addition beginning from the right hand side, in column 0. That is important. Find C and D in the column and row headings of the *addition* table. Find the entry in the table where those rows intersect. The result is 19. Record the 9 at the bottom of the *calculation* table in the result row. Record the 1 at the top in the Carry row.

Move left to column 1. Find A and B in the addition table column headings. Find the entry in the table where those rows intersect. The result is 15. Record the 1 in the carry row of column 2. Add the 5 to the carry (1) at the top of column 1. Record the resulting 6 at the bottom in the result row.

Move left to column 2. Find F and F in the addition table column headings. Find the entry in the table where those rows intersect. The result is 1E. Write the 1 in the carry row of column 3. Add the E to the carry in column 2. The result is F. Record the resulting F at the bottom in the result row.

Move left to column 3. Only 7 is available. Add 7 to the carry in column 3. The result is 8. Record the resulting 8 at the bottom in the result row.

The answer is $8F69_{16}$ = 8F69**H** = **Ox**8F69. In decimal, this answer is 36713.

If any carry is generated when adding numbers in column 3, the carry goes to the "Throw Away" column. This is because we assumed our word is only 16 bits long. Anything beyond that is lost. This is called an "overflow" condition.

Example 2. Add FFFF + 0001 using a 16 bit word. In decimal, this is 255 + 1 = 256.

	"Throw Away" Column	Carry			
	1	1	1	1	
Original Number		**F**	**F**	**F**	**F**
Radix Complement	Add →				**1**
Final Result		**0**	**0**	**0**	**0**

The answer is 0000_{16} = 0000**H** = **Ox**0000 = 0_{10}. This is an example of overflow.

Hexadecimal Addition Problems

Assume a 16-bit word. Perform the following additions. Record any bits you throw away separately from the answer.

Problem 1. 8888H + 8888H
Problem 2. CAD8H + BCA1H
Problem 3. F3FAH + 51CH
Problem 4. 2CCDH + 5A1FH
Problem 5. 3333H + ACDAH
Problem 6. 710H + 553H

Binary Coded Decimal (BCD) Notation

Binary Coded Decimal (BCD) notation is a method of representing base-10 numbers in a computer exactly. Each decimal number is represented by 4 bits (nibble) corresponding to the binary representation of that decimal number. For example, the decimal number 978 is represented in two bytes in BCD as

$$0000{:}\mathbf{1001{:}0111{:}1000}_{\mathbf{BCD}}$$

The attractive feature of BCD is that all decimal fractions used in commerce can be represented internally in BCD exactly. This is not possible with binary. Because our monetary system uses base 10 arithmetic, and our accounting standards are developed using base 10 arithmetic, we expect to do accounting on computers and get the same result we would get if we do it by hand.

BCD arithmetic is a natural choice for use with calculators, and computer applications, that operate on only decimal data.

Table 46. Decimal Fractions in Base 2, 16, 10, and BCD

Fraction	Base 2	Base 16	Base 10	BCD
1/10	0.00011	0.19	**0.1**	**0.0001**
2/10	0.0011	0.3	**0.2**	**0.0010**
3/10	0.01001	0.4C	**0.3**	**0.0011**
4/10	0.0110	0.6	**0.4**	**0.0100**
5/10	**0.1**	**0.8**	**0.5**	**0.0101**
6/10	0.1001	0.9	**0.6**	**0.0110**
7/10	0.10110	0.B3	**0.7**	**0.0111**
8/10	0.1100	0.C	**0.8**	**0.1000**
9/10	0.11100	0.E6	**0.9**	**0.1001**

Conversion between decimal and BCD is as easy as conversion between binary and hexadecimal. It is done by simple substitution. Each decimal digit produces four BCD binary digits. Each group of four BCD binary digits produces one decimal digit.

Example 1. Convert 975.283 from decimal to BCD.

Decimal	9	7	5	.2	8	3
BCD	1001	0111	0101	.0010	1000	0011

Example 2. Convert 1001 0111 0011 .0101 1000 0110 from BCD to decimal.

BCD	1001	0111	0011	.0101	1000	0110
Decimal	9	7	3	.5	8	6

Not every group of 4 binary digits is a BCD digit. All hexadecimal digits above 9 are not BCD codes.

Example 3. Convert 1100 1010 1011 from BCD to decimal.

Answer: The groups of 4 bits given in this problem do not represent decimal numbers in BCD.

One of the attractive features of BCD for business applications is that conversion from ASCII or EBCDIC character code format to BCD is much simpler than from character code format to binary. This is important to business data processing which spends most of its time doing input, output, sorting, and storage, and comparatively little time doing computation. ASCII and EBCDIC are 8-bit codes with the numerical digit in the right nibble (4-bit group). Obtain the BCD representation by discarding the left nibble. Technically, do a "logical AND" of the ASCII or EBCDIC code with "0F" (hexadecimal) [00001111 (binary)] to obtain the BCD code. Using the ability to shift register contents right and left, you can pack BCD digits. This is called "packed BCD" in technical literature by people that work at the machine language level of programming a processor. Therefore, if you have the decimal number "937", you get "0000:1001:0011:0111".

Table 47. Codes for Decimal Numbers in ASCII, EBCDIC and BCD

Decimal	0	1	2	3	4	5	6	7	8	9
ASCII Hex	30	31	32	33	34	35	36	37	38	39
ASCII Binary	0011: 0000	0011: 0001	0011: 0010	0011: 0011	0011: 0100	0011: 0101	0011: 0110	0011: 0111	0011: 1000	0011: 1001
EBCDIC Hex	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9
EBCDIC Binary	1111: 0000	1111: 0001	1111: 0010	1111: 0011	1111: 0100	1111: 0101	1111: 0110	1111: 0111	1111: 1000	1111: 1001
BCD	**0000**	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**

Hardware for performing BCD arithmetic was quickly introduced for computers designed for business data processing. The IBM 1401 had BCD hardware. The IBM 360 included BCD arithmetic implemented in microcode. When microprocessors were introduced, the Z80 processor included BCD arithmetic. The calculator is a natural target for using BCD arithmetic, as the person using a calculator is most likely to be thinking in base 10. Beginning with the 80486 processor, Intel processors have BCD and binary integer arithmetic support included in the x87 Floating Point Unit (FPU).

IBM Binary Coded Decimal (BCD)

IBM binary coded decimal is used only to represent integers. Any scaling for a fractional part must be done by the programmer.

IBM provides two forms, Packed BCD and Unpacked BCD. Packed BCD is more compact for storage and is the only BCD format that can be used in computations. This is an appropriate format for accounting data. Unpacked BCD is faster to use when a number is not required for numerical computations.

The codes used for plus and minus signs depend upon the operating environment used to generate the data. The codes for the sign are different than the codes for numbers. By placing the sign nibble or byte at the end a number, the sign can therefore be used to identify the end of a BCD number. See the SAS Language Reference Dictionary for a discussion.[34]

Table 48. IBM BCD Sign Codes

Plus Sign	Minus Sign
A_{16}	B_{16}
C_{16} (preferred)	D_{16} (preferred)
E_{16}	
F_{16} (also the Zone code)	

Zero is normally accompanied with a plus sign, although a minus sign with zero is also valid.

IBM Packed BCD

Also called "Packed Decimal" format, the packed BCD format is 1 to 16 bytes long, and stores one decimal digit in each 4-bit nibble. The sign also is stored as a 4-bit nibble. This permits representation of 31 decimal digits and a sign. A packed decimal integer always has an odd number of digits, one of which might be a leading zero. The sign nibble marks the end of the Packed BCD word.

In the diagram below, "d" is the decimal digit.

0 d 3	4 d 7	8 … n	n+1 d n+4	4-bit Sign

The cost of conversion from character format to packed decimal format is small compared to the cost in time to convert to binary. By stripping off the left nibble, memory and storage are more efficiently used.

IBM Unpacked BCD

Also called "Zoned Decimal" format, the unpacked BCD format stores one decimal digit in the right nibble of each byte. The left nibble contains a zone code. In the right-most byte, the left nibble is the sign. Except for the right-most byte, each byte in zoned decimal format is the ASCII (3n) or EBCDIC (Fn) code for that number. In the diagram below, "Z" is the zone character (3 or F), "d" is the decimal digit and "S" is the code for the sign. The lower case "n" is the digit number in the Unpacked BCD word, counting from the left.

0 Z 3	4 d 7	8 Z 11	12 d 15	16 … n	S	d

The advantage of zoned decimal format is that no time is required for data conversion from character format. It reduces the storage requirements for numerical data.

[34] SAS Institute, Inc., "Working with Packed Decimal and Zoned Decimal Data", SAS Language Reference: Dictionary (1999). http://www.rdg.ac.uk/ITS/Topic/Stats/StGSAS8_01/SAS8/lgref/z1131887.htm, 09 Jun 2002.

This is good for transaction processing, which is characterized by much input, sorting, and much output. Data must be converted to Packed BCD before doing computations.

Conversion to IBM BCD Problems

Problem 1. Convert ASCII -736 to IBM Unpacked BCD.

Z	d	Z	d	S	d

Problem 2. Convert ASCII -736 to IBM Packed BCD.

d	d	d	S

Fixed Point Notation

Positional Number System Review

The positional number system used a radix point to separate the integer part from the fraction part of a number. In the decimal number system, we call the radix point by the name "decimal point". The number 835.27 means

$$(8 \times 10^{2}) + (3 \times 10^{1}) + (5 \times 10^{0}) + (2 \times 10^{-1}) + (7 \times 10^{-2})$$

The positional number system for any base can be used to represent fractions as well as integers. The positional number system represents a number in the form

$$\ldots c_5 c_4 c_3 c_2 c_1 c_0 \,.\, c_{-1} c_{-2} c_{-3} c_{-4} \ldots$$

The c_k are the digits or coefficients of the number written in fixed point notation. The subscripts in the above expression are the position numbers. These numbers locate the position of the number as measured from the radix point. The base of the number system must be known in order to be able to evaluate the number. Without any additional information, if all the digits are in the range of 0 through 9, the number is assumed to be a decimal number. Otherwise, the base must be explicitly annotated. Let **b** be the base of the number. Then the number is evaluated as

$$\ldots(c_5 b^5)+(c_4 b^4)+(c_3 b^3)+(c_2 b^2)+(c_1 b^1)+(c_0 b^0)+(c_{-1} b^{-1})+(c_{-2} b^{-2})+(c_{-3} b^{-3})+(c_{-4} b^{-4})\ldots$$

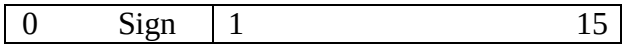
IBM Fixed Point

The IBM System 390 has four fixed point representations, which depend on length, and whether the number is signed or unsigned.

Signed fixed point (integers) use two's complement to represent negative numbers. A positive number has a sign bit of zero. Negative numbers are stored in two's complement form, and have a sign bit of one. A signed integer may be 16 or 32 bits long.

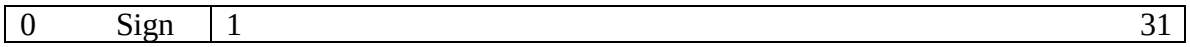
Unsigned integers are considered positive. An unsigned integer may be 8, 16, or 32 bits long. These are useful for representing logical (Boolean) data, counters, enumerated data, address offsets, and other purposes.

IBM Short Form Fixed Point



The signed short integer is a 16-bit, two's complement integer. The unsigned short integer is a 16-bit magnitude word.

IBM Long Form Fixed Point



The signed long integer is a 32-bit, two's complement integer. The unsigned long integer is a 32-bit magnitude word.

Convert Fixed Point Binary to Fixed Point Hexadecimal

Conversion from binary to hexadecimal for numbers with fractions is similar to the case with integers. Groups of four binary digits are converted to hexadecimal. The grouping begins at the radix point, in both directions from the radix point.

Example: Convert $11111110000111101011 01.00110001110001_2$ to hexadecimal.

Group the bits into groups of four digits, beginning at the radix point.

$$11\ 1111\ 1000\ 0111\ 1010\ 1101\ .\ 0011\ 0001\ 1100\ 01_2$$

If the extremities do not have four bits, pad with zeros in the direction away from the radix point. The number becomes

$$\mathbf{00}11\ 1111\ 1000\ 0111\ 1010\ 1101\ .\ 0011\ 0001\ 1100\ 01\mathbf{00}_2$$

Convert each group of 4 bits to a single hexadecimal digit.

0011	1111	1000	0111	1010	1101	. 0011	0001	1100	01**00**
3	F	8	7	A	D	. 3	1	C	4

The answer is 3F87AD.31D4

Convert a Fixed Point Hexadecimal Number to Binary

To convert a fixed point hexadecimal number to binary, merely substitute the 4-bit binary number for each hexadecimal number.

Example: Convert 3AC.6BC to binary.

3	A	C.	6	B	C
0011	1010	1100	. 0110	1011	1100

Extreme zeros may be trimmed from the result. The answer is

$$11\ 1010\ 1100.0110\ 1011\ 11_2$$

Convert a Fixed Point Binary Number to Decimal

The binary number 1101.101_2 is evaluated as

$$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3})$$

$$= 8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 = 13.625$$

3	2	1	0	-1	-2	-3	Position Number
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	Place Value
8	4	2	1	1/2	1/4	1/8	Place Value as Decimal Fraction
8	4	2	1	0.5	0.25	0.125	Place Value in Decimal
1	**1**	**0**	**1**	**. 1**	**0**	**1**	Coefficients
(1x8)	(1x4)	+ (Ox2)	+ (1x1)	+ (1x0.5)	+ (Ox0.25)	+ (1x0.125)	Sum of Products
8	+ 4	+ 0	+ 1	+ 0.5	+ 0.25	+ 0.125	
						ANSWER	**= 13.625**

Convert a Decimal Fixed Point Number to Binary

The approach for converting a decimal integer to binary is used for the whole number portion of the decimal fixed point number. The procedure for conversion of the fraction part of the decimal fixed point number to binary uses multiplication rather than division.

Example: Convert the decimal number 21.40625 to binary.

Partition the decimal number into a whole number plus a fraction.

$$21.40625 = 21 + 0.40625$$

Convert the whole number portion to binary using the method of successive division.

Step 0 Base = 2

1	0	**= Q_1**
2 | 2	1	= Q_0
2		
0	1	
	0	
	1	**= R_0**

→ **Step 1** Base = 2

	5	= Q_2
1 | 1	0	**= Q_1**
1	0	
0	**0**	**= R_1**

Step 2 Base = 2

2	**= Q_3**
5 |	= Q_2
4	
1	**= R_2**

→ **Step 3** Base = 2

1	= Q_4
2 |	= **Q_3**
2	
0	**= R_3**

Step 4 Base = 2

0	**= Q_5**
1 |	= Q_4
0	
1	**= R_4**

Stop only when the Quotient is zero.

The integer portion of the binary number is 1 0101.

Convert 0.40625 (the fraction part) by successive multiplication by the base, which is 2 in this case. The result of a multiplication that is to the left of the radix point becomes a digit in the answer.

Column Position Number		-1	-2	-3	-4	-5
Fraction of X	**0.**	**4**	**0**	**6**	**2**	**5**
b						**2**
Carry	0	0	1	0	1	
	0	8	0	2	4	0
R_{-1} = 0	**0.**	**8**	**1**	**2**	**5**	**0**

Continue repeating the process until the remaining fraction part is zero.

Column Position Number		-1	-2	-3	-4
Fraction of R_{-1}	**0.**	**8**	**1**	**2**	**5**
b					**2**
Carry	1	0	0	1	
	0	6	2	4	0
R_{-2} = 1	**1**	**6**	**2**	**5**	**0**

Column Position Number		-1	-2	-3
Fraction of R_{-2}	**0.**	**6**	**2**	**5**
b				**2**
Carry	1	0	1	
	0	2	4	0
R_{-3} = 1	**1**	**2**	**5**	**0**

Column Position Number		-1	-2
Fraction of R_{-3}	**0.**	**2**	**5**
b			**2**
Carry	0	1	
	0	4	0
R_{-4} = 0	**0**	**5**	**0**

Column Position Number		-1
Fraction of R_{-4}	**0.**	**5**
b		**2**
Carry	1	
	0	0
R_{-5} = 1	**1**	**0**

The procedure stops when the fraction of the last remainder is zero. Build the fraction part of the binary number working from left to right, away from the radix point. The fraction part of the answer is

$$\mathbf{R}_{-1}\mathbf{R}_{-2}\mathbf{R}_{-3}\mathbf{R}_{-4}\mathbf{R}_{-5} = 0.0110\ 1$$

The final answer is the sum of the whole part plus the fraction part.

$$1\ 0101 + 0.0110\ 1 = \mathbf{1\ 0101.0110\ 1_2 = 21.40625}$$

Convert a Fixed Point Hexadecimal Number to Decimal

The hexadecimal number $BAD.CE_{16}$ is evaluated as

$$(B \times 16^2) + (A \times 16^1) + (D \times 16^0) + (C \times 16^{-1}) + (E \times 16^{-2})$$
$$= (11 \times 256) + (10 \times 16) + (13 \times 1) + (12 \times 0.625) + (14 \times 0.00390625)$$
$$= 2816 + 160 + 13 + 7.5 + 0.0546875 = 2996.554688$$

2	**1**	**0**	**-1**	**-2**	Position Number
$\mathbf{16^2}$	$\mathbf{16^1}$	$\mathbf{16^0}$	$\mathbf{16^{-1}}$	$\mathbf{16^{-2}}$	Place Value
256	16	1	1/16	1/256	Place Value as Decimal Fraction
4256	16	1	0.625	0.00390625	Place Value in Decimal
B	**A**	**D**	**.C**	**E**	Coefficients
(**11**x**256**)	+ (**10x16**)	+ (**13**x**1**)	+ (**12**x**0.625**)	+ (**14x0.00390625**)	Sum of Products
+ 2816	+ 160	+ 13	+ 7.5	+ 0.0546875	
					= **2996.554688**

Convert a Decimal Fixed Point Number to Hexadecimal

The approach for converting a decimal integer to hexadecimal is used for the whole number portion of the decimal fixed point number. The procedure for conversion of the fraction part of the decimal fixed point number to hexadecimal uses multiplication rather than division.

Example: Convert the decimal number 3290.9921875 to hexadecimal.

Partition the decimal number into a whole number plus a fraction.

$$3290.9921875 = 3290 + 0.9921875$$

Convert the whole number portion to hexadecimal using the method of successive division.

Step 0 Base = 16

	2	0	5	= **Q_1**
3	2	9	0	= Q_0
3	2			
	0	9		
		0		
		9	**0**	
		8	**0**	
		1	**0**	
		=	**A_{16}**	**= R_0**

→ **Step 1** Base = 16

	1	2	= Q_2
2	0	5	**= Q_1**
1	6		
	4	5	
	3	2	
	1	**3**	
	=	**D_{16}**	**= R_1**

Step 2 Base = 16

	0		= **Q_3**
1	2		= Q_2
	0		
1	**2**		
=	**C_{16}**		**= R_2**

Stop only when the Quotient is zero.

The integer portion of the hexadecimal number is **CDA_{16}**.

Convert 0.9921875 (the fraction part) by successive multiplication by the base, which is 16 in this case. The result of a multiplication that is to the left of the radix point becomes a digit in the answer.

Column Position Number		-1	-2	-3	-4	-5	-6	-7
Fraction of X	**0**	**.9**	**9**	**2**	**1**	**8**	**7**	**5**
b							**1**	**6**
Sum Carry	1 1			2	2	1		
Carry 0	5	5	1	0	4	4	3	
Carry 1	0	0	0	0	0	0		
		4	4	2	6	8	2	0
	9	9	2	1	8	7	5	
R_{-1} = 0	**1 5**	**8**	**7**	**5**	**0**	**0**	**0**	**0**
=	**F_{16}**							

Continue repeating the process until the remaining fraction part is zero.

Column Position Number			-1	-2	-3
Fraction of R_{-1}		**0**	**.8**	**7**	**5**
b				**1**	**6**
Sum Carry	1	2	1		
Carry 0		4	4	3	0
Carry 1		0	0	0	0
			8	2	0
		8	7	5	0
R_{-2} =	**1**	**4**	**0**	**0**	**0**
=		**E_{16}**			

The procedure stops when the fraction of the last remainder is zero. Build the fraction part of the binary number working from left to right, away from the radix point. The fraction part of the answer is

$$\mathbf{R}_{-1}\mathbf{R}_{-2} = 0.FE$$

The final answer is the sum of the whole part plus the fraction part.

$$CDA + 0.FE = \mathbf{CDA.FE_{16} = 3290.9921875}$$

Fixed Point Conversion Problems

Problem 1. Convert $11\ 1011\ 0101.0011\ 01_2$ to hexadecimal.

Problem 2. Convert $FED.CAB_{16}$ to binary.

Problem 3. Convert $10\ 1100.011_2$ to decimal.

Problem 4. Convert 107.625 to binary. ($110\ 1011.101_2$)

Problem 5. Convert $AD.E_{16}$ to decimal. (173.875)

Problem 6. Convert 430.7578125 to hexadecimal. (1AE.C2**H**)

Partitioned Binary Numbers

Sometimes it is useful (especially in networking for constructing subnet numbers) to partition a binary number so the sum of the individual numbers is the value of the original number. For example, partition the binary number $10\ 1011_2$ between bit numbers 4 and 3, where bit 0 is the right-most bit. Let the left partition be $10\ 0000_2$, and the right partition be given by 1011_2. Observe that the sum is the original value.

Left partition	1	0	0	0	0	0
Right partition			1	0	1	1
Sum	1	0	1	0	1	1

Convert the left partition $10\ 0000_2$ to decimal.

5	4	3	2	1	0	← position number
2^5	2^4	2^3	2^2	2^1	2^0	← place value as power of base 2
32	16	8	4	2	1	← place value
1	0	0	0	0	0_2	← digits

$= (1 \times 32) + (0 \times 16) + (0 \times 8) + (0 \times 4) + (0 \times 2) + (0 \times 1)$
$= 32 + 0 + 0 + 0 + 0 + 0$
$= 32$

Convert the right partition 1011_2 to decimal.

3	2	1	0	← position number (bit number)
2^3	2^2	2^1	2^0	← place value as power of base 2
8	4	2	1	← place value
1	0	1	1_2	← digits

$= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)$
$= 8 + 0 + 2 + 1$
$= 11$

Add the decimal values of these partitions. $32 + 11 = 43$.

Partition Problems

Problem A1. Partition $1011\ 0110_2$ between bits number 4 and 5. What are the two resulting decimal numbers and their sum? Find:

a. Left partition
b. Right partition
c. Sum

Problem A2. Partition $0110\ 1100_2$ between bits number 4 and 5. What are the two resulting decimal numbers and their sum? Find:

a. Left partition
b. Right partition
c. Sum

Partitioning Internet Protocol (IP) Numbers

Internet Protocol (IP) addresses for Internet Protocol Version 4 (Ipv4, in use during 2002 A.D.) are 32 bits long, and are written as a **dotted decimal** number. It is a sequence of four decimal numbers that represent the value of each octet. The term **octet** refers to a group of 8 bits.[35] A typical IP address written in dotted decimal form is: **10.1.127.201** . In the table below, this IP address is converted to binary and hexadecimal. The second row contains the bit number in the 32-bit word.

10								**1**								**127**								**201**							
31							24	23							16	15							8	7							0
0	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**
0				A				0				1				7				F				6				9			

A network address is assigned to a network by the American Registry for Internet Numbers (ARIN). The administrator of a network has the option of partitioning its network into subnetworks. While there are some restrictions, the basic task is to identify which IP numbers will belong to each subnetwork. The details are taught in a networking class. This discussion will not pay attention to those details.

Suppose you have been assigned the network address 10.x.x.x. This network address gives you three octets to use, signified by x. Suppose you want to establish subnets that have at least 4000 host machines on each subnet. How many subnets can you create, and what is the block of numbers you can assign as addresses on each subnet?

In the simplest of systems, **subnet host machines are selected by use of the N least significant bits** (right-most bits) in an IP address. The problem reduces to selecting the number N. Find the smallest power of 2 that is greater than or equal to 4000. Solution: N = 12, because $2^{12} = 4096$. The right-most 12 bits, numbered 0 through 11, are used to identify individual host machines on each subnet. This leaves bits 12 through 23 (12 bits) for identifying a particular subnet. You therefore can have 4096 addresses for subnets.

Some of the subnet addresses are reserved for special purposes. An IP address having a host machine address of all zeros is the "Network Address" or "Network Number". An IP address having a host machine address of all ones is the "Broadcast Address" for that subnet, which routes messages to all computers on that subnet. Thus, **the number of computers on a classical subnet is two less than the number of numbers that can be represented by the host machine address portion of the IP address.**

[35] Debra Littlejohn Shinder, *Computer Networking Essentials*, page 233, Cisco Systems, Inc. (2001)

The allocation looks like the following.

31							24	23							16	15							8	7							0
0	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**

The portion coded with "**x**" is the **address for a subnet.** The portion coded with "**y**" is the **address for a specific host machine on that subnet.**

A necessary task is to identify the numbers that correspond to a chosen partition.

Subnet Mask

A subnet mask has 1 in all the positions in the network and subnet portion of the address field. The dotted decimal notation is 255.255.240.0 . When this is ANDed with an incoming IP address, the result is the address for the target subnet.

255								**255**								**240**								**0**							
31							24	23							16	15							8	7							0
1	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
F				F				F				F				F				0				0				0			

Subnet Problems

Problem B1. Convert the dotted decimal for IP address 17.10.127.5 to binary and hexadecimal, using the table below.

31							24	23							16	15							8	7							0

Problem B2. How many subnet addresses are possible in the subnet mask shown below?

31							24	23							16	15							8	7							0
0	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**

Problem B3. How many host machine addresses are possible for each subnet in the subnet mask shown below?

31							24	23							16	15							8	7							0
0	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**	**y**

Problem B4. How many bits in the below mask need to be set aside to make sure each subnet can have at least 283 host machines?

31							24	23							16	15							8	7							0
0	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**

Problem B5. How many subnet addresses are possible in the below mask if enough bits are set aside to make sure each subnet can have at least 283 host machines?

31							24	23							16	15							8	7							0
0	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**	**z**

Problem B6. What is the dotted decimal notation for the subnet mask needed for ensuring enough bits are set aside to ensure each subnet can have at least 283 host machines?

255								255								254								0							
31							24	23							16	15							8	7							0

(German) Roman Numeral System

Not every number system is a strict positional number system like the decimal system. The (German) Roman numeral system is a common example. We still use this for ceremonial purposes, such as dates on monuments, buildings, and currency. It is also popular on clocks. The Roman numeral system has no symbol for zero. To count in Roman numerals, we use letters of the Latin alphabet as follows.

Table 49. Counting Using (German) Roman Numerals

Decimal	Roman
1	I
2	II
3	III
4	IV
5	V
6	VI
7	VII
8	VIII
9	IX
10	X

Decimal	Roman
11	XI
12	XII
13	XIII
14	XIV
15	XV
16	XVI
17	XVII
18	XVIII
19	XIX
20	XX

Decimal	Roman
21	XXI
22	XXII
23	XXIII
24	XXIV
25	XXV
26	XXVI
27	XXVII
28	XXVIII
29	XXIX
30	XXX

Decimal	Roman
31	XXXI
32	XXXII
33	XXXIII
34	XXXIV
35	XXXV
36	XXXVI
37	XXXVII
38	XXXVIII
39	XXXIX
40	XL

Decimal	Roman
41	XLI
42	XLII
43	XLIII
44	XLIV
45	XLV
46	XLVI
47	XLVII
48	XLVIII
49	XLIX
50	L

Decimal	Roman
51	LI
52	LII
53	LIII
54	LIV
55	LV
56	LVI
57	LVII
58	LVIII
59	LIX
60	LX

Decimal	Roman
10	X
20	XX
30	XXX
40	XL
50	L
60	LX
70	LXX
80	LXXX
90	XC
100	C

Decimal	Roman
100	C
200	CC
300	CCC
400	CD
500	D
600	DC
700	DCC
800	DCCC
900	CM
1,000	M

Decimal	Roman
1,000	M
1,500	MD
2,000	MM
3,000	MMM
4,000	$\mathrm{M}\bar{\mathrm{V}}$
5,000	$\bar{\mathrm{V}}$
1,000,000	$\bar{\mathrm{M}}$

X is used for 10. V is the upper half of X, and is used for 5. L is used for 50. C is used for 100. D is 500. M is 1000. [The Romans used (|), not M.][36] A dash line above a letter multiplies the base value by 1000. There is no zero. Sometimes, a pair of horizontal bars, above and below these symbols, are used to distinguish these from mere alphabet letters.

While there is a pattern to the meaning of numbers depending on their position, it is very different from the simple pattern used with the decimal number system. As you might guess, arithmetic using the Roman numeral system is not as easy as with the decimal system. The Romans apparently did not do computation with their numerals, but rather used a counting board or abacus. A 1st century A.D. gravestone shows a person with an abacus in his hand.[37]

The Roman numeral system applied meaning to the position of numbers, but not in the same way our present day positional number system does. It evolved from a tally system. The basic tally system uses vertical lines, one line for each object counted. The next modification is a grouping operation that is applied in steps of 5. After accumulating 4 vertical marks, such as |||| , a grouping is made by placing a diagonal slash through the group of 4 vertical marks.

The Roman numeral system is the next level of sophistication. Instead of having 5 marks, the symbol V is substituted. It took less space to write IV for 4 than it did to write |||| . Writing materials were precious. When the printing press was invented, word spellings and number formats were modified to further reduce the space required.

Roman numerals are created from left-to-right and evaluated from right-to-left. The object is to express the quantity using the fewest possible symbols.

Parsing Rule. The pattern for evaluation that developed is as follows. Group adjacent identical Roman Numerals before evaluating the number. Consider a pair of adjacent Roman numerals.[38] Beginning with the right end, if the numeral to the left is less in value to the numeral on the right,(just the numeral immediately to the right, not the whole number to the right), the numeral on the left is subtracted from the numeral on the right. If the numeral on the left is greater than or equal to the numeral on the right, the numeral on the left is added to the numeral on the right. Gerard Schildberger has published a list of numbers from 1 to 3999.[39]

The following examples of years (except 1944, 1999, 2002) are from the Government Printing Office Style Manual.[40]

---

[36] Karl Menninger, *Number Words and Number Symbols: A Cultural History of Numbers*, page 281, Dover Publications (1992).
[37] Karl Menninger, *Number Words and Number Symbols: A Cultural History of Numbers*, page 306, Dover Publications (1992).
[38] Gerard Schildberger, "Roman Numerals are Nonassociative", email. (23 June 2002)
[39] Gerard Schildberger, "The numbers from 1 to 3999 expressed as Roman numerals", http://www.research.att.com/~njas/sequences/a006968.txt visited 04 April 2001.
[40] *United States Government Printing Office Style Manual 2000*, U.S. Government Printing Office (2000), pg 189. http://www.access.gpo.gov/styleman/2000/style001.html, 26 June 2002

Table 50. Roman Numeral Years in the U.S. Government Printing Office Style Manual

Years Written in Roman Numerals					
1600	MDC	1930	MCMXXX	1980	MCMLXXX
1700	MDCC	1940	MCMXL	1990	MCMXC
1800	MDCCC	1944	MCMXLIV	1999	MIM
1900	MCM	1950	MCML	2000	MM
1910	MCMX	1960	MCMLX	2002	MMII
1920	MCMXX	1970	MCMLXX	2010	MMX

You can sometimes reduce the number of symbols appearing in a Roman numeral by the following approach. If the same symbol appears on both the right side and the left side of a higher value symbol, these symbols constitute a matched pair. You can remove both symbols without changing the value of the number. Examples:

- IXI → X
- ICMLI → CML

A problem with Roman numerals is ambiguity, as illustrated with the following example. The parentheses indicate the grouping used in evaluating the example. Note that this kind of ambiguity is impossible with our own positional number system.

- (CIV)(MXVI) = -104 + 1016 = 912
- C(I(V(MX)V)I) → CMX = 910
- C(IVM)(XVI) = -100-4+1000+10+5+1 = 912
- (CI)(VM)XVI = -101 –5 + 1016 = 910
- (CIV)M(XVI) = -104 + 1000 + 16 = 912
- CIVMXVI = 910, evaluated according to the parsing rule.

Background for next example:

- (VC)I = 95 + 1 = 96
- V(CI) = 99 – 5 = 94

Example:

- V(CIM) = 899 – 5 = 894
- (VC)(IM) = 999 – 95 = 904
- ((VC)I)M = 1000 – 96 = 904
- (V(CI))M = 1000 – 94 = 906
- VCIM = 894, evaluated according to the parsing rule.

The parsing rule given here applies to Roman numerals written today, but this was not always true throughout history. To read literature of earlier times, you will need to study the conventions in use at that time.

German Roman Numeral Problems
For Romans with a lot of Gaul.

Problem 1. Write 390 in Roman numerals. (Celts defeated Rome on 18 July 390 BC, and sacked Rome 3 days later.[41])
Problem 2. Write 407 AD in Roman numerals. (Constantine withdrew from Britain that year.[42])
Problem 3. Convert 255 to Roman numerals.
Problem 4. Add the Roman numerals XI and XXXII. Convert the result to decimal after adding.
Problem 5. What time is XXV O'clock? (Humorous answer.)
Problem 6. Write zero in Roman numerals. (Trick question.)
Problem 7. Divide XXX by VI .
Problem 8. There is a number on the base of the pyramid shown on the back side of the paper One Dollar Federal Reserve Note, MDCCLXXVI. Copy that number given in Roman Numerals and translate it into Indo-Arabic numbers (the number system we use in commerce today).
Problem 9. Evaluate the numeral I(VL) , evaluating inside the parentheses first.
Problem 10. Evaluate the numeral (IV)L , evaluating inside the parentheses first.

[41] Encyclopaedia Britannica (1965)
[42] Encyclopaedia Britannica (1965)

Fibonacci Bases

Another example of a number system that is not a strict positional number system like the decimal system is a Fibonacci base. If you have a twisted mind, or really enjoy looking at the world in ways different than most people, this is the number system for you. A really nice discussion, complete with online calculators, is given by Knott.[43]

Fibonacci introduced the sequence that now bears his name in *Liber abaci* (1202) a problem he stated: "How many pairs of rabbits can be produced from a single pair in one year if it is assumed that every month each pair begets a new pair which from the second month becomes productive?"[44]

Fibonacci Numbers

The first few Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, … The first two numbers of the sequence are established by definition as $F_0 = 0$, $F_1 = 1$. The rest of the numbers are defined by the recursion formula $F_n = F_{n-1} + F_{n-2}$ where n is a variable index that takes on values 2, 3, 4, 5, … .

The sequence of Fibonacci numbers led to other questions in number theory. A short list of examples of the sequence seen in nature is listed in Encyclopaedia Britannica. Generalized Fibonacci numbers are useful in the design of efficient methods of sorting very large sets of data.[45]

Fibonacci Base Representation

The Fibonacci numbers are used as the basis of the number system in a place-value notation. A number written in the Fibonacci Base Representation has the form

$$a_nF_n + a_{n-1}F_{n-1} + \ldots + a_4F_4 + a_3F_3 + a_2F_2$$

where the coefficient a_n is an integer multiplier of the corresponding Fibonacci number F_n. For example, the Fibonacci Base Representation number $95013_{\mathbf{Fib}}$ is converted to a decimal number by:

$95013_{\mathbf{Fib}}$	$= (9 \times F_6) + (5 \times F_5) + (0 \times F_4) + (1 \times F_3) + (3 \times F_2)$
	$= (9 \times 8) + (5 \times 5) + (0 \times 3) + (1 \times 2) + (3 \times 1)$
	$= 72 + 25 + 0 + 2 + 3$
	$= 102_{\mathbf{10}}$

Notice that the lowest subscript on the Fibonacci number and the corresponding coefficient is 2 because $F_2 = 1$. $F_1 = 1$, and $F_0 = 0$. While it would be possible to extend the notation down to F_0, there is nothing to be gained by having two unit bases F_2 and F_1,

---

[43] Ron Knott, "Using Fibonacci Numbers to Represent Whole Numbers" (19 January 2007). http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibrep.html visited 15 December 2007.
[44] *Encyclopaedia Britannica 2003*, Deluxe Edition CD-ROM
[45] Donald E. Knuth, *The Art of Computer Programming*, Volume 3, Searching and Sorting, Addison-Wesley (1973).

and pointless to include F_0 since anything multiplied by zero (F0) is zero, and does not change the value of the number being represented.

Extended Fibonacci Base Representation

The Fibonacci series can be extended to negative indexes. This is useful in other work related to Fibonacci numbers. The extension is done by rearranging the equation to: $F_{n-1} = F_{n+1} - F_n$.

n	8	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
F_n	21	13	8	5	3	2	1	1	0	1	-1	2	-3	5	-8	13	-21

From this table, observe that

Result	When
$F_{-n} = F_n$	n is ODD
$F_{-n} = -F_n$	n is EVEN

Minimal Fibonacci Base Representation

Knott[46] described a Fibonacci base positional number system proposed by Edouard Zeckendorf in 1972 that uses the fewest Fibonacci numbers. An alternative name for this representation is therefore the Zeckendorf Representation. It allows a unique representation of any positive integer as a sum of Fibonacci numbers, by applying a few simple rules:

- Number position values increase from right to left.
- Use only the digits 0 and 1 as coefficients.
- No two consecutive Fibonacci numbers can be used in the same sum. This means the digit 1 cannot appear adjacent to another 1.

Example:

F_{14}	**F_{13}**	**F_{12}**	**F_{11}**	**F_{10}**	**F_9**	**F_8**	**F_7**	**F_6**	**F_5**	**F_4**	**F_3**	**F_2**	← Fibonacci Number
377	**233**	**144**	**89**	**55**	**34**	**21**	**13**	**8**	**5**	**3**	**2**	**1**	← place values
0	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	← **digits**
0	**0**	**0**	**0**	**0**	**0**	**21**	**0**	**0**	**5**	**0**	**0**	**1**	← decimal values

Check the answer to see if it is correct.

1 0 0 1 0 0 1 $_{\text{Fib}}$ is the same as

(**21** x **1**) + (**5** x **1**) + (**1** x **1**) = 21 + 5 + 1 = 27

[46] Ron Knott, "Using Fibonacci Numbers to Represent Whole Numbers" (19 January 2007). http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibrep.html visited 15 December 2007.

This is the sum of the digits times their place values.

Knott provides a program on his web site for converting decimal to minimal Fibonacci base representation, as well as for other conversions. Another name for "standard" is "canonical". A number placed into an agreed standard form is in canonical form.

Convert Decimal to Minimal Fibonacci Base Representation

1. Set Remainder **R** to the decimal number to be converted.
2. Find the largest Fibonacci number **F** equal to or less than R.
3. Place a 1 in the column of the translation table corresponding to that Fibonacci number **F**.
4. Subtract **F** from **R**, with the result being the new **R:** $\mathbf{R_{new} \leftarrow R - F}$.
5. If R is greater than 0, go to step 2 and repeat the process.

Example Conversions to Minimal Fibonacci Base Representation

1. Convert 317_{10} to Minimal Fibonacci Base (Zeckendorf) Representation.

The number in the top row, labeled "Remainder R", is the amount remaining to be coded into Zeckendorf Representation by considering subtraction of the Fibonacci number in that column. Begin the procedure on the left-most column, and proceed to the right. For the column of Fibonacci Number F_{10}, the Remainder R before subtraction is **84**. F_{10} = **55**, which is less than or equal to **84**. Therefore, perform the subtraction and record "**1**" in the "digits" row of that column. Record the remainder from the subtraction (**29**) at the top of the column of the next column to the right, above F_9.

317	317	84	84	84	29	29	8	8	0	0	0	0	Remainder **R**
F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	F_7	F_6	F_5	F_4	F_3	F_2	← Fibonacci Number
377	233	144	89	55	34	21	13	8	5	3	2	1	← place values
0	1	0	0	1	0	1	0	1	0	0	0	0	← digits
0	233	0	0	55	0	0	0	0	0	0	0	0	← decimal values
	84			29		8		0					Remainder **R**

The answer is **$1001\ 0101\ 0000_{Fib}$**

2. Convert 69_{10} to Minimal Fibonacci Base (Zeckendorf) Representation

69	69	69	69	69	14	14	14	1	1	1	1	1	Remainder **R**
F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	F_7	F_6	F_5	F_4	F_3	F_2	← Fibonacci Number
377	233	144	89	55	34	21	13	8	5	3	2	1	← place values
0	0	0	0	1	0	0	1	0	0	0	0	1	← digits
0	0	0	0	55	0	0	13	0	0	0	0	1	← decimal values
				14			1					0	Remainder **R**

The answer is **$1\ 0010\ 0001_{Fib}$**

Addition in Minimal Fibonacci Base Representation

Two Fibonacci numbers written in Zeckendorf representation can be added.

Example: The two Zeckendorf representation numbers to be added are in the rows labeled X and Y. The result of the addition is in the row labeled "Answer".

F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	F_7	F_6	F_5	F_4	F_3	F_2	← Fibonacci Number
377	233	144	89	55	34	21	13	8	5	3	2	1	← place values
0	1	0	0	1	0	1	0	1	0	0	0	0	← X
0	0	0	0	1	0	0	1	0	0	0	0	1	← Y
0	1	0	0	2	0	1	1	1	0	0	0	1	Answer
	233			110		21	13	8				1	← decimal values

The answer is **1002 0111 0001**$_{\mathbf{Fib}}$ = (233+110+21+13+8+1)$_{10}$ = **386**$_{10}$

Restore Fibonacci Base Representation to Minimal Fibonacci Base Representation

After adding, it may be necessary to rewrite the answer to restore it to Zeckendorf representation. Two rules can be helpful. The primary purpose is to have a systematic method of applying the rules until no opportunity is available to apply them again. You can modify the rules to start scanning from the opposite end. You will eventually get the same result.

- Rule 1: Start from the right end of the row and proceed to the left to find the next "2".
  - If column F_2 contains 2, set column F_2 to zero and add 1 to column F_3.
  - If column F3 contains 2,
    - Set column F_3 to zero.
    - Add 1 to column F_4 and to F_2.
  - For F_4 or higher, if column F_n contains a 2, then
    - set F_n to zero and add 1 to F_{n+1} and to F_{n-2}.
    - This is the recursion $2F_n = F_{n+1} + F_{n-2}$.
- Rule 2: Start from the right end of the row and proceed to the left. If two adjacent columns, F_n and $F_{n-1,}$ contain ones, then
  - Set F_n and F_{n-1} to zero and set F_{n+1} to 1.
  - This is the recursion $F_{n+1} = F_n + F_{n-1}$.

The number in the third row below, labeled "Answer", is the result of an addition of two numbers in Zeckendorf representation. In each subsequent row, one of the rules is applied to "simplify" the result. Repeated application of the rules eventually will transform the answer to Zeckendorf representation. The final result is in the last row.

F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	F_7	F_6	F_5	F_4	F_3	F_2	← Fibonacci Number
377	233	144	89	55	34	21	13	8	5	3	2	1	← place values
0	1	0	0	2	0	1	1	1	0	0	0	1	Answer
	1		1	0		2	1	1				1	Rule 1
	1		1		1	0	1	2				1	Rule 1
	1		1		1		2	0		1		1	Rule 1
	1		1		1	1	0	0	1	1		1	Rule 1
	1		1		1	1	0	1	0	0		1	Rule 2
	1		1	1	0	0	0	1	0	0		1	Rule 2
	1	1	0	0	0	0	0	1	0	0		1	Rule 2
1	0	0	0	0	0	0	0	1	0	0	0	1	Rule 2

The answer in Zeckendorf representation is: $\mathbf{1\ 0000\ 0001\ 0001}_{\mathbf{Fib}} = (377+8+1)_{10} = \mathbf{386}_{10}$

Fibonacci Base Representation Problems

Problem 1. Convert 50_{10} to Minimal Fibonacci Base (Zeckendorf) Representation.

50													Remainder **R**
F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	F_7	F_6	F_5	F_4	F_3	F_2	← Fibonacci Number
377	233	144	89	55	34	21	13	8	5	3	2	1	← place values
													← digits
													← decimal values
													Remainder **R**

Problem 2. Convert 53_{10} to Minimal Fibonacci Base (Zeckendorf) Representation.

53													Remainder **R**
F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	F_7	F_6	F_5	F_4	F_3	F_2	← Fibonacci Number
377	233	144	89	55	34	21	13	8	5	3	2	1	← place values
													← digits
													← decimal values
													Remainder **R**

Problem 3. Add the two Zeckendorf Representation numbers:
$1000\ 1010\ 0101_{Fib} + 1010\ 0001\ 0011_{Fib}$

F_{15}	F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	F_7	F_6	F_5	F_4	F_3	F_2	← Fibonacci Number
610	**377**	**233**	**144**	**89**	**55**	**34**	**21**	**13**	**8**	**5**	**3**	**2**	**1**	← place values
		1	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	← X
		1	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	← Y
														Answer
														← decimal values

Problem 4. Restore Fibonacci Base Representation 10110112_{Fib} to Minimal Fibonacci Base Representation.

F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	F_7	F_6	F_5	F_4	F_3	F_2	← Fibonacci Number
377	**233**	**144**	**89**	**55**	**34**	**21**	**13**	**8**	**5**	**3**	**2**	**1**	← place values
					1	**0**	**1**	**1**	**0**	**1**	**1**	**2**	Answer
													Rule
													Rule
													Rule
													Rule
													Rule
													Rule
													Rule
													Rule
													Rule
													Rule
													Rule

Phi Base Number System

The base Phi (Φ) number system is an idea closely related to the Fibonacci base representation. This is an example of a number system base that is an irrational number, yet all integers can be represented exactly in this base. Knott has a fascinating discussion[47] on this topic, which is the basis for what follows. If this topic interests you, visit his interactive website which includes calculators and diagrams.

Phi Definition

Phi is defined by the relation $\Phi - 1 = 1/\Phi = \varphi$. The solutions to this equation are: $\Phi = \frac{1}{2}\left(1 \pm \sqrt{5}\right)$. Phi is an irrational number (cannot be expressed exactly as the ratio of two integers) because of the $\sqrt{5}$ which is irrational. The solutions are approximately

$\Phi_A = 1.618034$	$\varphi_A = 0.61803399$
$\Phi_B = -0.61803399$	$\varphi_B = -1.618034$

Phi Relationship to Fibonacci Numbers

Φ and φ are related to Fibonacci numbers by the following equation:

$$\Phi^n = F_{n+1} + F_n j = (F_n + F_{n-1}) + F_n j = F_{n-1} + F_n(1 + j) = F_{n-1} + F_n \Phi$$

To show this using the Principle of Finite Induction, first let $n = 1$ and prove the equation $\Phi^n = F_{n+1} + F_n j$ reduces to $\Phi = 1 + \varphi$. Then assume that the equation $\Phi^{n-1} = F_n + F_{n-1} j$ is true for an arbitrary number n, and prove from this starting point that $\Phi^n = F_{n+1} + F_n j$ is true.

Simple Properties of Phi

There are two practical simple properties upon which other work with Phi depends. Both are not hard to show.

- $j(1 + j) = 1$
- $\Phi^n = \Phi^{n-1} + \Phi^{n-2}$

Phi Base Representation

Numbers in Phi Base representation are expressed as a weighted sum of powers of Φ, in the form

$$c_n\Phi^n + c_{n-1}\Phi^{n-1} + \ldots + c_2\Phi^2 + c_1\Phi^1 + c_0\Phi^0 + c_{-1}\Phi^{-1} + c_{-2}\Phi^{-2} + \ldots + c_{-(n-1)}\Phi^{-(n-1)} + c_{-n}\Phi^{-n}$$

where the coefficient c_n is some multiplier (possibly zero) of the n^{th} power of Φ. Numbers are expressed compactly in a manner similar to decimal notation by writing only the constant coefficients and placing a radix point to the left of the coefficient c_{-1}. When some coefficients are positive and others are negative, the negative coefficients

[47] Ron Knott, "Using Powers of Phi to represent Integers (Base Phi)" (22 December 2007). http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/phigits.html visited 24 December 2007.

must be identified. One method is to enclose the coefficient with a minus sign inside parenthesis.

Integers in Phi Base Representation

All integers can be represented in Phi Base Representation for suitable choices of the various coefficients. It is useful to look at a few examples.

$\Phi^0 = F_1+F_0\varphi = \mathbf{1}$		
$\Phi^1 = F_2+F_1\varphi = 1+\varphi$	$\Phi^{-1} = F_0+F_{-1}\varphi = \varphi$	$L_1 = \Phi^1 - \Phi^{-1} = \mathbf{1}$
$\Phi^2 = F_3+F_2\varphi = 2+\varphi$	$\Phi^{-2} = F_{-1}+F_{-2}\varphi = 1-\varphi$	$L_2 = \Phi^2 + \Phi^{-2} = \mathbf{3}$
		$\Phi^1 + \Phi^{-2} = \mathbf{2}$
$\Phi^3 = F_4+F_3\varphi = 3+2\varphi$	$\Phi^{-3} = F_{-2}+F_{-3}\varphi = -1+2\varphi$	$L_3 = \Phi^3 - \Phi^{-3} = \mathbf{4}$
$\Phi^4 = F_5+F_4\varphi = 5+3\varphi$	$\Phi^{-4} = F_{-3}+F_{-4}\varphi = 2-3\varphi$	$L_4 = \Phi^4 + \Phi^{-4} = \mathbf{7}$
$\Phi^5 = F_6+F_5\varphi = 8+5\varphi$	$\Phi^{-5} = F_{-4}+F_{-5}\varphi = -3+5\varphi$	$L_5 = \Phi^5 - \Phi^{-5} = \mathbf{11}$
$\Phi^6 = F_7+F_6\varphi = 13+8\varphi$	$\Phi^{-6} = F_{-5}+F_{-6}\varphi = 5-8\varphi$	$L_6 = \Phi^6 + \Phi^{-6} = \mathbf{18}$

Notice that the sequence $L_n = \Phi^n + (-1)^n\Phi^{-n}$ for $n > 0$ of results form a series similar to the Fibonacci series: $L_n = L_{n-1} + L_{n-2}$ where $L_1 = 1$ and $L_2 = 3$. The numbers in the set {…, 1, 3, 4, 7, 11, 18, …} are called *Lucas numbers*.

Of course, all integers can be expressed merely as an integer multiple of Φ^0, such as $13 = 13\Phi^0$. This certainly would be a unique representation, but it would miss the point that we can represent integers by combinations of numbers using an irrational base. The combinations used to express a particular integer are not necessarily unique. For example, 5 = 1+4 = 2+3.

Integers	Equation	Phi Base Representations
5	$= 5\Phi^0$	5.0
	$= 5(\Phi^1 - \Phi^{-1})$	50.(-5)
4 + 1	$= (\Phi^3 - \Phi^{-3}) + (\Phi^1 - \Phi^{-1})$	1010.(-1)0(-1)
2 + 2 + 1	$= 2(\Phi^1 + \Phi^{-2}) + \Phi^0$	21.02
3 + 2	$= (\Phi^2 - \Phi^{-2}) + (\Phi^1 + \Phi^{-2})$	110.0
	$= 3\Phi^0 + (\Phi^1 + \Phi^{-2})$	13.01
	$= (\Phi^2 - \Phi^{-2}) + 2\Phi^0$	102.0(-1)

This generates a need for restrictions to establish unique representations.

Minimal Phi Base Representation

Adapt the rules for Minimal Fibonacci Base Representation to the case of Phi Base Representation.

- Number position values increase from right to left.
- Use only the digits 0 and 1 as coefficients.

- No two consecutive Phi Base numbers can be used in the same sum. This means the digit 1 cannot appear adjacent to another 1.

Digits written in minimal Phi base representation are known as "**Phigits**".[48]

Converting Phi Base Representation to Minimal Phi Base Representation

It may be necessary to rewrite a number written in Phi Base Representation to Minimal Phi Base Representation. Two rules can be helpful. Both implement the recursion $\Phi^{n+1} = \Phi^n + \Phi^{n-1}$ The primary purpose is to have a systematic method of applying the rules until no opportunity is available to apply them again. You can modify the rules to start scanning from the opposite end. You will eventually get the same result.

- Rule 1: Start from the right end of the row and proceed to the left to find the next number greater than "1". If column Φ_n contains a number greater than 1, subtract 1 from column Φ_n, add 1 to column Φ_{n-1}, and add 1 to column Φ_{n-2}.
- Rule 2: Start from the left end of the row and proceed to the right. If two adjacent columns, Φ_n and $\Phi_{n-1,}$ contain numbers greater than zero, then subtract 1 from columns Φ_n and Φ_{n-1} and add 1 to column Φ_{n+1}.

Example 1: Write $2(\Phi^1+\Phi^{-2})$, written as $20.02_{\mathbf{Phi}}$,. and convert it to minimal phi base representation

Step	F^3	F^2	F^1	F^0	·	F^{-1}	F^{-2}	F^{-3}	F^{-4}
1 Start			**2**	0		0	**2**		
2 R#1			2	0		0	1	1	1
3 R#1			1	1		1	1	1	1
4 R#2		1	0	0		1	1	1	1
5 R#2		1	0	1		0	0	1	1
6 R#2		**1**	0	**1**		0	**1**	0	0

The minimal phi base representation is $\Phi^2 + \Phi^0 + \Phi^{-2}$ written as . $101.01_{\mathbf{Phi}}$.

When there is a negative coefficient, one approach is to propagate positive coefficients from the left to the right by reversing Rule 2. The goal is to produce a coefficient of the same power of Φ that is the same in magnitude, and opposite in sign, and then do the subtraction.

[48] Jose Glez-Regueral, Professor at Madrid, cited by Ron Knott, "Other names for Base Phi", in "Using Powers of Phi to represent Integers" (22 December 2007).
http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/phigits.html visited 24 December 2007.

Example 2: Write $\Phi^3 - \Phi^{-3}$, written as $1000.00(-1)_{Phi}$, and convert it to minimal phi base representation.

Step	F^3	F^2	F^1	F^0	.	F^{-1}	F^{-2}	F^{-3}	F^{-4}
1 Start	**1**	0	0	0		0	0	**(-1)**	
2 R^{-1} #2	0	1	1	0		0	0	(-1)	
3 R^{-1} #2	0	1	0	1		1	0	(-1)	
4 R^{-1} #2	0	1	0	1		0	1	1+(-1)	
5	0	**1**	0	**1**		0	**1**	0	

The minimal phi base representation is $\Phi^2 + \Phi^0 + \Phi^{-2}$ written as . 101.01_{Phi} .

Phi Based Addition

Perform arithmetic with Phi base numbers as you would with an general algebraic power series, and then apply the rules for transforming the result to a minimal Phi base representation.

Example: $101.01_{Phi} + 1_{Phi}$. In decimal, this is the problem: 4 + 1 = 5. These numbers represent:

$1\Phi^2 + 0\Phi^1 + 1\Phi^0 + 0\Phi^{-1} + 1\Phi^{-2} = 101.01_{Phi}$
$0\Phi^2 + 0\Phi^1 + 1\Phi^0 + 0\Phi^{-1} + 0\Phi^{-2} = 1_{Phi}$

Write each coefficient in the column corresponding to the associated power of Φ.

	F^3	F^2	F^1	F^0		F^{-1}	F^{-2}	F^{-3}	F^{-4}
c =		1	0	1	**.**	0	1		
d =				1	**.**	0	0		
c + d =		**1**	**0**	**2**	**.**	**0**	**1**		
Rule 1		1	0	1	**.**	1	2		
Rule 1		1	0	1	**.**	1	1	1	1
Rule 2		1	1	0	**.**	0	1	1	1
Rule 2	1	0	0	0	**.**	0	1	1	1
Rule 2 Answer	**1**	**0**	**0**	**0**	**.**	**1**	**0**	**0**	**1**

The coefficients for each individual power of Φ are added separately. This produced the intermediate result: c + d = 102.01_{Phi}. This is: $1\Phi^2 + 0\Phi^1 + 2\Phi^0 + 0\Phi^{-1} + 1\Phi^{-2}$

While this answer is correct, it is not in standard (canonical) form. Rules were applied to transform this result to the minimal Phi base representation.

The answer is $1000.1001_{Phi} = 1\Phi^3 + 0\Phi^2 + 0\Phi^1 + 0\Phi^0 + 1\Phi^{-1} + 0\Phi^{-2} + 0\Phi^{-3} + 1\Phi^{-4}$

This can be expanded: $1\Phi^3 + 1\Phi^{-1} + 1\Phi^{-4} = 3+2\varphi + \varphi + 2-3\varphi = 5$.

Phi Base Representation Problems

Problem 1. Express $A = \Phi^4 + \Phi^3 + \Phi^{-3}$ as phigits:

	F^6	F^5	F^4	F^3	F^2	F^1	F^0	.	F^{-1}	F^{-2}	F^{-3}	F^{-4}	F^{-5}	F^{-6}
A =								.						

Problem 2. Express $B = \Phi^4 + \Phi^1 + \Phi^{-4} + \Phi^{-6}$ as phigits:

	F^6	F^5	F^4	F^3	F^2	F^1	F^0	.	F^{-1}	F^{-2}	F^{-3}	F^{-4}	F^{-5}	F^{-6}
B =								.						

Problem 3. Add the phigits A (from Problem 1) and B (from Problem 2).

	F^6	F^5	F^4	F^3	F^2	F^1	F^0	.	F^{-1}	F^{-2}	F^{-3}	F^{-4}	F^{-5}	F^{-6}
A =								.						
B =								.						
A+B=								.						

Problem 4. Transform the sum, A+B, computed in Problem 3 into minimal Phi base representation. In the left column, write the rule number being used for each step of the transformation.

	F^6	F^5	F^4	F^3	F^2	F^1	F^0	.	F^{-1}	F^{-2}	F^{-3}	F^{-4}	F^{-5}	F^{-6}
A+B=								.						
								.						
								.						
								.						
								.						
								.						
								.						

Problem 5. In the answer to Problem 4, substitute expressions for F^n in terms of n and f. Reduce the expression to the fewest number of terms.

Continued Fractions

Another type of notation that is positional is that of continued fractions. It is useful for expressing some very important constants which have a non-repeating decimal form, such as e and $\sqrt{2}$. This is a topic that is sometimes taught in high schools as an extra topic. A continued fraction has the general form

$$X = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cfrac{b_4}{a_4 + \cfrac{b_5}{a_5 + \cdots}}}}}$$

where the constants can be written in the more compact form of

$$\left[a_0, \frac{b_1}{a_1}, \frac{b_2}{a_2}, \frac{b_3}{a_3}, \frac{b_4}{a_4}, \frac{b_5}{a_5}, \cdots\right]$$

Every real number can be written as a unique continued fraction, and every rational number can be written as a continued fraction with a finite number of terms.[49] Using this notation, we get very regular expressions for some important transcendental numbers. Here are two important examples.

$$\sqrt{2} = [1,2,2,2,\cdots] \approx 1.414215\ldots$$

$$e = 1 + \left[1, \frac{2}{2}, \frac{3}{3}, \frac{4}{4}, \frac{5}{5}, \cdots\right] \approx 2.718281828459045\ldots$$

A procedure that is repeated is called iterative. Iterative calculation makes it easy to repeat the calculation until the answer is accurate enough for your purposes. The tendency of the computation to stabilize near a final answer is called convergence. The difference between the computed answer and the ideal value is called error. When you do not know the number of steps needed for convergence, iterative methods can be very useful.

A procedure that calls itself is called recursive. Recursive methods reveal fundamental properties of a problem that might otherwise escape notice. Recursive procedures often take less code to implement than an iterative procedure that accomplishes the same end goal. This simplicity is very attractive when doing

[49] Mark Herkommer, "Continued Fractions", *Number Theory: A Programmer's Guide*, McGraw-Hill (1998).

calculations manually on a calculator. D'Souza has a nice tutorial for C++ programmers on recursion.[50]

Some very important applications of recursion occur in computer science and mathematics. Recursive methods are sometimes the most efficient approach to implementing translators. Another important application of recursion is in the topic of fractals. God seems to have a special love of recursion in applying fractals in nature, such as the growth of trees or the shape of a shore line.

The apparent ease of recursive calculation is beguiling. Recursive methods are usually not as efficient as alternative methods of computing the same quantity, when measured by time required for convergence. For a more detailed discussion, see Acton.[51]

Example 1: Evaluate the Square Root of 2.

$$\sqrt{2} = [1,2,2,2,\cdots]$$

This is in the form of a continued fraction. We obviously cannot compute an infinite number of terms. (We would be late for dinner if we tried.) However, this gives us a method for approximating some important quantities. We can compute the first N terms, where we choose N in advance. Write this as a continued fraction, and approximate it by evaluating the first 6 terms.

$$\sqrt{2} = 1+\frac{1}{2+\frac{1}{2+\frac{1}{2+\frac{1}{2+\frac{1}{2+...}}}}} \approx 1+\frac{1}{2+\frac{1}{2+\frac{1}{2+\frac{1}{2+\frac{1}{2}}}}} = 1+\frac{1}{2+\frac{1}{2+\frac{1}{2+\frac{1}{2.5}}}}$$

$$= 1+\frac{1}{2+\frac{1}{2+\frac{1}{2.454545455}}} = 1+\frac{1}{2+\frac{1}{2.407407407}} = 1+\frac{1}{2.415384615} = 1.414012739$$

This evaluation is very easy using a calculator with an reciprocal function, x^{-1}. On the TI-83 calculator, perform the following steps:

[50] Erwin D'Souza, "Recursion Programming", CoolComps Inc., http://www11.brinkster.com/erwnerve/recursion.htm (2000). 01 July 2002.
[51] Forman S. Acton, *Numerical Methods that Work*, Harper and Row (1970). pp 356-357.

(1) 2 ENTER

(2) x^{-1} + 2 ENTER

(3) Repeat step (2) until the variation of answer from one iteration to the next iteration is smaller than your stopping criteria.

(4) x^{-1} + 1 ENTER

Example 2: Evaluate **e**, approximating it by the first 6 terms of the continued fraction for **e**.

$$e = 1 + \left[1, \frac{2}{2}, \frac{3}{3}, \frac{4}{4}, \frac{5}{5}, \cdots\right] = 1 + 1 + \cfrac{2}{2 + \cfrac{3}{3 + \cfrac{4}{4 + \cfrac{5}{5 + \cfrac{6}{6 + \cdots}}}}} \cong 1 + 1 + \cfrac{2}{2 + \cfrac{3}{3 + \cfrac{4}{4 + \cfrac{5}{5 + \cfrac{6}{6}}}}}$$

$$= 1 + 1 + \cfrac{2}{2 + \cfrac{3}{3 + \cfrac{4}{4 + \cfrac{5}{5 + 1}}}} = 1 + 1 + \cfrac{2}{2 + \cfrac{3}{3 + \cfrac{4}{4 + \cfrac{5}{6}}}} = 1 + 1 + \cfrac{2}{2 + \cfrac{3}{3 + \cfrac{4}{4.833\ldots}}} \approx 1 + 1 + \cfrac{2}{2 + \cfrac{3}{3.827586207}}$$

$$\approx 1 + 1 + \cfrac{2}{2.783783784} \approx 2.71844602$$

This evaluation requires that you choose the number of steps in advance. It is very easy using a calculator with an reciprocal function, x^{-1}. For N = 6, perform the following steps:

(1) N = 6

(2) N ENTER

(3) x^{-1} * (N – 1) + (N – 2) ENTER

(4) N = N – 1

(5) Repeat steps (3) and (4) until N = 2 (that is, until N – 2 = 0).

(6) + 1 ENTER

Unfortunately, the usual arithmetic operations of addition, subtraction, multiplication, and division are not easy for continued fractions. Some tutorials on

continued fractions are: Cut-the-Knot[52], Doctor Luis[53], Edward G. Dunne[54], and Mark A. Herkommer[55].

Continued Fraction Problem

Problem 1. The *golden ratio* is the solution to the equation $r^2 = r + 1$. Divide both sides by r to get the equation $r = 1 + 1/r$. Substitute $(1 + 1/r)$ for r in the denominator. Continue this forever. The equation $r = 1 + 1/r$ is called a recursive equation. With the substitutions done, it looks like

$$X = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cdots}}}}}$$

Evaluate the first six terms. Show all your steps.

$$X = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1}}}}}$$

[52] Cut-the-Knot, http://www.cut-the-knot.com/do_you_know/fraction.shtml
[53] Doctor Luis, "Continued Fractions", *The Math Forum*, Drexel University (2002). http://www.mathforum.org/dr.math, 11 JUN 2002.
[54] Edward G. Dunne, "Primes and Continued Fractions", American Mathematical Society. http://www.research.alt.com/~njas/sequences/DUNNE/TEMPERMENT.HTML.
[55] Mark Herkommer, "Continued Fractions", *Number Theory: A Programmer's Guide*, McGraw-Hill (1998).

Finite Word Length Arithmetic

When we learned arithmetic in elementary school, we allowed numbers to grow without limit on the number of digits we were allowed to use. We were permitted to use infinite-precision arithmetic. If we needed to expand the number of digits on the right end of a number as a result of multiplication or addition, we just wrote the digit.

Today's binary computers store data in words with a finite number of binary digits. **The number of numbers that can be represented exactly is finite and fixed.** This has some important implications. The only way to increase the number of numbers that can be represented is to increase the number of bits. This can be done by increasing the number of bits per word (hardware), or by using multiple words (software or hardware).

Codes for representing numbers are designed so that arithmetic operations produce results that are usually accurate enough for applications of interest. Sometimes, we can take advantage of the finite size to do useful operations easily, such as doing subtraction by using complement arithmetic.

We cannot represent all numbers exactly. This leads to several kinds of errors due to arithmetic using finite length word size.

Errors Due to Finite Word Size
Errors due to finite length word size include the following.

- Approximation errors
  - Round off
  - Truncation
- Overflow
- Underflow

In addition to these errors, logic can detect arithmetic errors such as dividing by zero by testing the value of the denominator before or during division. This cause is not due to finite length arithmetic, but it does cause a finite length word overflow error.

Approximation

There are gaps between numbers that we represent exactly. For example, we cannot represent exactly the number 1/3 in base two. If the number we need cannot be represented exactly, the selection of a number we can represent results in an approximation error.

Evaluation of functions require time and resources. Many important functions cannot usually be computed exactly, such as square root, trigonometric functions, and logarithms. Engineers and scientists also use other Special Functions important to their work which require approximation. Such functions are evaluated using numerical methods or tabled values.

The accumulation of approximation errors can easily lead to large errors in a sequence of computations. The study of such errors and methods to minimize such errors are topics in numerical methods.

Round Off

Round off error is the difference between the exact number desired and the closest number that can be represented using selected code for the finite length word. The usual result is to increase the variance of the error as computations proceed.

Truncation

Truncation is a special case of approximation error that occurs by discarding bits without rounding to the closest approximation. This is the "Floor" operator. The usual result is to produce an answer biased closer to zero.

Overflow

Overflow is the condition of producing a result larger than can be represented with the finite length word.

Overflow can occur by adding or multiplying two very large numbers to produce a number even larger that cannot be contained within the word. A more common cause is to divide a large number by a number very close to zero.

Underflow

Underflow is the condition of producing a result that should be non-zero that is closer to zero than can be represented by the word using the selected code.

Underflow can occur by dividing a small number by a large number.

Arithmetic Examples

Strategies for performing arithmetic with finite length of computer words has been to work with integers (fixed point), an exponential notation (floating point) analogous to scientific notation, or significance arithmetic.[56] Fixed point arithmetic places the burden on the programmer of keeping track of which part of the word represents whole numbers, and which part represents fractions. Further coding and work is done if the range of numbers includes negative numbers. Significance Arithmetic explicitly includes quantifying accuracy and precision of an operation.

The purpose of the examples below is to get you thinking in terms of what has to be considered when working with only a fixed number of digits. These examples assume a fixed point notation using decimal arithmetic.

Suppose we are allowed a 5 digit number, with three digits to the left of the decimal point and two digits to the right of the decimal point. We consider addition, multiplication, and division. Traditional subtraction is not affected. (Think about why this is true.)

---

[56] Weisstein, Eric W. "Significance Arithmetic.", MathWorld--A Wolfram Web Resource (10 February 2008). http://mathworld.wolfram.com/SignificanceArithmetic.html visited 12 February 2008.

Addition example: 555.55 + 555.55

	Throw Away	Col 2	Col 1	Col 0	Col –1	Col –2
		5	5	5	.5	5
		5	5	5	.5	5
Carry		0	0	0	.0	0
Intermediate Sum	1	1	1	1	.1	
Answer	1	**1**	**1**	**1**	**.1**	**0**

Multiplication example: 689.76 x 3.2

	Col 5	Col 4	Col 3	Col 2	Col 1	Col 0	Col –1	Col – 2
A				**6**	**8**	**9**	**.7**	**6**
times B						**3**	**.2**	
Carry 1		1	1	1	1	1		
Multiply by Column (-1)			2	6	8	4	2	
Intermediate 1 Sum		1	3	7	9.	5	2	
Carry 2	1	2	2	2	1			
Multiply by Column (0)		8	4	7	1	8		
Intermediate 2 Partial Carry A	1		1	1	1			
Intermediate 2 Partial Sum A	1	1	9	6.	1	3	2	
Intermediate 2 Partial Carry B		1						
Intermediate 2 Partial Sum B	2	1	0	7	2	3	2	
Answer	**2**	**2**	**0**	**7**	**2**	**3**	**2**	

The assumed rules define which 5 digits are retained in the answer. If you retained Column 0 as the baseline, the reported answer would be 207.20. The usual practice is to retain the highest order digits, which gives us 2207.2 for this problem. If you are limited to only 5 digits, you will lose some information no matter what you do.

Notice that the number of additions needed depend upon the number of times we need to consider a carry. Designers of multiplication circuits for computers think carefully about how to reduce the number of steps involved in multiplication. Good designs require thoughtful insight into the basic steps of multiplication, and attention to common special cases.

Division example: Some processors place the dividend (numerator) initially into a register combination that is twice the length of the word being used for storage in main memory. For our 5 digit example, the dividend would be placed into a 10 digit register pair. The result consists of a quotient and a remainder, each in a storage-length word.

The notation below is similar to the format for doing division by hand in elementary schools in Italy.[57] It is a convenient notation for making explicit which quotient digit is associated with which multiplication. This results in fewer mistakes when done by hand.

	Col 2	**Col 1**	**Col 0**	**Col –1**	**Col –2**	**Col –3**	**Col –4**	**Col –5**	**Col –6**	**Col –7**	
Divisor	**Divi-dend**										
127.26)	**8**	**1**	**7**	**.4**	**5**	0	0	0	0	0	**Quotient**
Multiply	8	9	0	.6	2						**(7**
Subtract		2	6	.8	3	0	0	0	0	0	**(.**
Multiply		2	5	4	5	2					**(2**
Subtract			1	.3	7	8	0	0	0	0	
Multiply			1	2	7	2	6				**(1**
Subtract				.1	0	5	4	0	0	0	
Multiply				0	0	0	0	0			**(0**
Subtract				.1	0	5	4	0	0	0	
Multiply					8	9	7	4	5		**(7**
Subtract					1	5	6	5	5	0	

The answer is approximately **7.2107**

International Notation Systems for Arithmetic

Notation for writing arithmetic varies around the world, both the symbols used for arithmetic operators, and the layout of the problem when done by hand. For example, it is common that the symbol for multiplication is a dot (·) rather than (x). Computer programming languages often use the asterisk (*). The notation used in America for the division problem 13/7 is expressed in Germany as 13 : 7.

Notation can suggest novel or more efficient methods of solving problems. The following are from Germany.[58]

Manual Multiplication in Germany

Compared to multiplication taught in America, the German notation places the two numbers to be multiplied on the same line, separated by a centered dot used as the multiplication symbol. Multiplication begins with the left-most digit of the multiplier, and proceeds to the right. An example is presented twice. The first example is the

[57] Student from Italy (c. 2001)
[58] Karla Gaudet, example with instructions. (05 January 2008)

compact version as it would normally be done. The second version goes step by miniature step.

Multiplication example. 324 · 961

3	**2**	**4**	**·**	**9**	**6**	**1**
	2	9	1	6		
		1	9	4	4	
				3	2	1
	3	1	1	3	6	4

The multiplier is 961.

The “carry” operation is normally done mentally without writing it down. The set of tables below show the process in individual steps, with the “carry” recorded explicitly.

Step 1

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry 3				3			
Multiply 4·9					6		

Step 2

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry 1			1	3			
Multiply by 2·9				8	6		

Step 3

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry 2		2	1	3			
Multiply by 3·9			7	8	6		

Step 4

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		

Step 5

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry 2					2		
Multiply 4·6						4	

Step 6

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry 1				1	2		
Multiply 2·6					2	4	

Step 7

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry			1	1	2		
Multiply 3·6				8	2	4	

Longer Variation A:

Step 8A

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry			1	1	2		
Multiply 3·6				8	2	4	
Subtotal 2			1	9	4	4	

Step 9A

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry			1	1	2		
Multiply				8	2	4	
Subtotal 2			1	9	4	4	
Carry							
Multiply 4·1							4

Step 10A

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry			1	1	2		
Multiply				8	2	4	
Subtotal 2			1	9	4	4	
Carry							
Multiply 2·1						2	4

Step 11A

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry			1	1	2		
Multiply				8	2	4	
Subtotal 2			1	9	4	4	
Carry							
Multiply 3·1					3	2	4

Step 12A

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry			1	1	2		
Multiply				8	2	4	
Subtotal 2			1	9	4	4	
Carry							
Multiply 3·1					3	2	4
Subtotal 3					3	2	4

Step 13A. Add the subtotals.

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry			1	1	2		
Multiply				8	2	4	
Subtotal 2			1	9	4	4	
Carry							
Multiply 3·1					3	2	4
Subtotal 3					3	2	4
TOTAL		3	1	1	3	6	4

End of problem.

Shorter Variation B:

Step 8B. In this version, the next total includes the subtotal immediately above it.

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry			1	1	2		
Multiply 3·6				8	2	4	
Subtotal 2		3	1	1	0	4	
Carry							
Multiply 4·1							4

Step 9B.

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry			1	1	2		
Multiply 3·6				8	2	4	
Subtotal 2		3	1	1	0	4	
Carry							
Multiply 2·1						2	4

Step 10B.

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry			1	1	2		
Multiply 3·6				8	2	4	
Subtotal 2		3	1	1	0	4	
Carry							
Multiply 3·1					3	2	4

Step 11B. In this version, the next total includes the subtotal immediately above it.

Multiplicand.multiplier	3	2	4	·	9	6	1
Carry		2	1	3			
Multiply			7	8	6		
Subtotal 1		2	9	1	6		
Carry			1	1	2		
Multiply 3·6				8	2	4	
Subtotal 2		3	1	1	0	4	
Carry							
Multiply 3·1					3	2	4
TOTAL		3	1	1	3	6	4

Manual Division in Germany

In Germany, the symbol used for division is the colon (:) rather than the slash (/) or the division symbol (÷). The format of multiply and subtract is the same as done in America. The notation differs by

- Placing the divisor to the right of the numerator
- Placing the quotient to the right rather than on top of the numerator.
- No distinguishing division symbol to the left (v) and line above the numerator

2	**4**	**5**	**:**	**2**	**3**	**=**			**10.652173**
2	3								
	1	5	0						
	1	3	8						
		1	2	0					
		1	1	5					
				5	0				
				4	6				
					4	0			
					2	3			
					1	7	0		
					1	6	1		
							9	0	
							6	9	

Complement Arithmetic

Introduction to Complement Arithmetic

Complement arithmetic is an efficient way to perform subtraction involving numbers represented by a finite length word. A major benefit is that it permits subtraction without having to think about borrowing from adjacent columns when you use finite length arithmetic. Binary digital computers use "One's Complement" or "Two's Complement" arithmetic internally. It is very useful for computer professionals to be able to do this in hexadecimal or octal for tracing through memory dumps to examine the performance of a program for errors.

From a computer's point of view, the advantage is that subtraction can be implemented as a combination of complement and addition. No extra subtraction hardware is required. The logical complement operation is very quick to do.

If you declare that numbers having a one in the most significant bit are negative, and those with zero in the most significant bit are positive, then you can use two's complement notation to signify negative numbers in binary. With this convention, radix complement (two's complement in binary) provides a unique representation of the number zero. Without this convention, you cannot tell a positive number from a negative number by just looking at the bit pattern.

Base 10 Complement Arithmetic

We will begin with an illustration in base ten. We learned base 10 arithmetic because most humans have 10 digits (8 fingers, 2 thumbs)… the original digital calculator. If we want to perform the subtraction 724 - 538, we would do the following.

After Borrow	6	11	14
	7	**2**	**4**
-	**5**	**3**	**8**
Result	**1**	**8**	**6**

The problem with this approach is that you must determine if you need to borrow from the column immediately to the left. If so, you add the amount borrowed from the current column and subtract it from the left-adjacent column. After completing the borrow operation, you then perform the subtraction. Humans can do this without much additional attention when we work with base 10 because we have practiced it for years. It is not quite as efficient as we would like for implementation on a computer. We can make this easier.

Suppose we are only allowed to use 3-place numbers, 000 through 999. We are required to represent all positive and negative numbers we work with by using these

numbers. To represent a number using complement arithmetic using base 10, do the following.

		9	**9**	**9**
Subtract		**5**	**3**	**8**
Intermediate Result		4	6	1
Add		0	0	1
Complement		**4**	**6**	**2**

The top line consists of a finite length word filled with our highest digit in our base 10 number system, which is 9. Recall, the digits in base 10 run from 0 through 9. The next line is our traditional negative number. If our negative number were only 2 digits, it would be treated as having leading zeros to the left. The number in the second row is subtracted from the top row. Notice that no borrows are required from adjacent columns. The third row is the result of this subtraction. In base 10, this number is the "Nine's Complement".

The final operation is to add 1 to the nine's complement. This produces our "Radix Complement", where the radix of the number system is 10. This is our negative number. To complete the subtraction, add this radix complement to the original number.

Throw away any digits that carry beyond the third column.

	"Throw Away" Column			
Original Number		**7**	**2**	**4**
Radix Complement	Add →	**4**	**6**	**2**
Final Result	1	**1**	**8**	**6**

The result is 186.

Try some examples. Using 4 digits, compute: 3928 - 489 = 3439.

Radix Complement of -489 is 9511. 3928 + 9511 = 1|3439 = 3439. The vertical bar separates the 4 digit number (which we keep) from the fifth digit (which we throw away).

Try: -239 - 398 using 4 digits.

Base 16 (Hexadecimal) Complement Arithmetic

The principles with base 16 are the same as for base 10, except that we have 16 digits (0 through F) to use. Instead of subtracting from 9999, we subtract from FFFF.

The problem is to compute:

	7	**D**	**4**
-	**5**	**C**	**E**
Result	**2**	**0**	**6**

Begin by finding the 15's Complement of 5CE.

	F	**F**	**F**
-	**5**	**C**	**E**
Result	A	3	1

Find the Radix Complement by adding 1 to the 15's Complement.

	A	3	1
Add			1
Radix Complement	**A**	**3**	**2**

To perform the original subtraction, we now do addition. Refer to the Hexadecimal Addition Table.

	"Throw Away" Column			
Carry	1	1		
Original Number		**7**	**D**	**4**
Radix Complement		**A**	**3**	**2**
Result	1	**2**	**0**	**6**

Drop the digit carried into position 4.

The answer is 206**H**, where the suffix **H** identifies the number as a hexadecimal number.

Try another one!

		1	**F**	**3**
Subtract	**-**	**4**	**6**	**7**
Result				

Hmm… How do we do this? The negative number is bigger than the positive number! Let's try.

Find the radix complement of 467H.

	F	**F**	**F**
-	**4**	**6**	**7**
	B	9	8
+			1
Radix Complement	**B**	**9**	**9**

Add the radix complement to 1F3H.

Carry	1		
	1	**F**	**3**
+	**B**	**9**	**9**
	D	**8**	**C**

The difference between 1F3H and 467H is D8CH.

Convert this back to minus a positive number.

	F	**F**	**F**
-	**D**	**8**	**C**
	2	7	3
			1
	2	**7**	**4**

Append the minus sign. The answer in signed-magnitude format is **-274H**.

Base 2 (Binary) Complement Arithmetic

Consider the subtraction problem: 1100_2 - 0111_2. The first step is to convert the negative number to radix complement.

	1	**1**	**1**	**1**
-	**0**	**1**	**1**	**1**
	1	0	0	0
+				1
Radix Complement	**1**	**0**	**0**	**1**

There is an historical anomaly in the name of this representation. The number “1000” is called the “one’s complement of 0111”. The number “1001” is called the “two’s complement of 0111”. This is the concept that people nation-wide think of when you mention “two’s complement”. The problem occurs when you work with base 3. It leads to an ambiguity of terminology. The term “radix complement” is unambiguous, but many people do not understand the term “radix”. The next step is to add the radix complement.

	"Throw Away" Column				
Carry					
Original Number		1	1	0	0
Add		**1**	**0**	**0**	**1**
Final Result	1	**0**	**1**	**0**	**1**

The answer is 0101_2.

Try the problem in decimal to see if the answer makes sense.

$1100_2 = 12_{10}$.
$0111_2 = 7_{10}$.

$12 - 7 = 5_{10} = 0101_2$.

Oops… it worked! ☺

Complement Arithmetic Problems

Problem 1. 3-digit Base 10: Compute 33 – 15.

		9	**9**	**9**			"Throw Away" Column			
Subtract			**1**	**5**		Original Number			**3**	**3**
Intermediate Result						Radix Complement	Add →			
Add		0	0	1		Final Result				
Radix Complement										

Problem 2. 3-digit Base 10: Compute 33 – 45.

		9	**9**	**9**			"Throw Away" Column			
Subtract			**4**	**5**		Original Number			**3**	**3**
Intermediate Result						Radix Complement	Add →			
Add		0	0	1		Final Result				
Radix Complement										

Convert this back to minus a positive number.

		9	**9**	**9**
Subtract				
Intermediate Result				
Add		0	0	1
Final Result				

Append the minus sign to get the signed-magnitude representation.

Problem 3. 3-digit Hexadecimal. Compute $FAD_{16} - DA1_{16}$

Use the Hexadecimal Addition Table to perform hexadecimal addition.

		F	**F**	**F**			"Throw Away" Column			
Subtract		**D**	**A**	**1**		Original Number		**F**	**A**	**D**
Intermediate Result						Radix Complement	Add →			
Add		0	0	1		Final Result				
Complement										

Problem 4. 3-digit Hexadecimal. Compute $FAD_{16} - FEA_{16}$

		F	**F**	**F**			"Throw Away" Column			
Subtract		**F**	**E**	**A**		Original Number		**F**	**A**	**D**
Intermediate Result						Radix Complement	Add →			
Add		0	0	1		Result				
Radix Complement										

Convert this back to minus a positive number.

	"Throw Away" Column	**F**	**F**	**F**
Subtract				
Intermediate Result				
Add		0	0	1
Final Result				

Notice that all the numbers used in computation are positive numbers. The purpose of complement arithmetic is to handle negative numbers without using signed arithmetic. It is possible since we are using finite sized words to represent numbers.

We attach the minus sign to the result as a notation that the number is a negative number. Notice that FFF (as a positive number) is greater than FC3 (as a positive number), so FFF – F3C > 0. Therefore 03C + 1 > 0. We attach a minus sign to 03D because we started with FAD – FEA. We note that FAD < FEA, so FAD – FEA < 0 .

Problem 5. 8-digit Binary. Subtract $11\ 0110_2 - 1\ 1101_2$

	"Throw Away" Column	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Subtract					**1**	**1**	**1**	**0**	**1**
Intermediate Result									
Add		0	0	0	0	0	0	0	1
Radix Complement									

	"Throw Away" Column								
Original Number				**1**	**1**	**0**	**1**	**1**	**0**
Radix Complement	Add →								
Final Result									

Problem 6. 8-digit Binary. Subtract $11\ 0110_2 - 1011\ 1101_2$

		1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Subtract		**1**	**0**	**1**	**1**	**1**	**1**	**0**	**1**
Intermediate Result									
Add		0	0	0	0	0	0	0	1
Radix Complement									

	"Throw Away" Column								
Original Number				**1**	**1**	**0**	**1**	**1**	**0**
Radix Complement	Add →								
Result									

Convert this back to minus a positive number.

		1	1	1	1	1	1	1	1
Subtract									
Intermediate Result									
Add		0	0	0	0	0	0	0	1
Final Result									

Again, notice that all the numbers used in computations are positive numbers. We attach a minus sign to the result to tag it as a negative number since our original problem is $11\ 0110_2 - 1011\ 1101_2$ where $11\ 0110_2 < 1011\ 1101_2$.

See if your answers are correct.

Fast Two's Complement

There is a fast method for doing two's complement without doing an addition. Locate the position of the right-most 1-bit. For example, in the word below, the right-most bit is in position 3.

Bit Position	7	6	5	4	3	2	1	0
Example:	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**

To take the two's complement of this example, leave bits in positions 3, 2, 1, and 0 alone. Take the one's complement of bits in positions 7, 6, 5, and 4. The answer for this example is:

Bit Position	7	6	5	4	3	2	1	0
Example:	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**

In general, find the position of the right-most 1-bit. Take the one's complement of all the higher order bits. This approach for finding two's complement is good for hardware design.

Fast Two's Complement Problems

Problem 1. Consider the binary number 00110100_2.

a. What bit position is the right-most 1-bit in?
b. What bit positions are left alone when writing the fast two's complement?
c. Write the two's complement of this binary number.

Problem 2. Consider the binary number 11010000_2.

a. What bit position is the right-most 1-bit in?
b. What bit positions are left alone when writing the fast two's complement?
c. Write the two's complement of this binary number.

Binary Coded Decimal Excess-3 Notation

George Robert Stibitz of Bell Labs in 1939 proposed representing decimal numbers by adding 3 to the Binary Coded Decimal representation.[59] It is called "Excess-3" because the code is greater than the reference value by 3. Instead of using the word "excess", the shorter notation "XS" is used.

This simplified the hardware implementation. It is much easier to take the complement for subtraction. It is easier to detect a "carry" during addition. In the table below, the binary code is given for BCD and BCD XS-3 for comparison.

Table 51. Complements in Decimal, BCD, and BCD Excess-3

Decimal	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Complement of Decimal (9 – x)	9	8	7	6	5	4	3	2	1	0

BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
Complement of BCD	1111	1110	1101	1100	1011	1010	1001	1000	0111	0110

BCD XS-3	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**	**1010**	**1011**	**1100**
Complement of BCD XS-3	**1100**	**1011**	**1010**	**1001**	**1000**	**0111**	**0110**	**0101**	**0100**	**0011**

Notice that the complement of BCD XS-3 produces the BCD XS-3 code for the decimal complement. For example, the BCD XS-3 code for decimal 5 is binary 1000. The complement is 0111, which is the BCD XS-3 code for decimal 4. The complement of decimal 5 is 4 (ie, 9 – 5 = 4). The BCD XS-3 code produces a useful result.

On the other hand, the BCD code (not BCD XS-3 code) for decimal 5 is binary 0101. The complement is 1010, which is not a code for any decimal digit in the BCD (not XS-3) code.

Addition with BCD XS-3

The concept of addition using BCD Excess-3 is simple. Implementation is easier to handle in hardware than BCD without Excess-3 coding because of the symmetric placement of the plain text within the code.

The details that need attention are how to handle situations that result in a sum exceeding the code. There are two ways for the sum to overflow: code overflow and word overflow. In both cases, a carry bit C is set to 1. The value of C is added to the next higher significant digit.

---

[59] Paul E. Ceruzzi , "Building the Complex Number Computer", *Reckoners: The Prehistory of the Digital Computer, from Relays to the Stored Program Concept, 1935-1945*, Chapter 4: Number, Please: Computers at Bell Labs, pages 80 – 84, Greenwood Press (1983). http://ed-thelen.org/comp-hist/Reckoners-ch-4.html visited 05 February 2008.

- A code overflow exists when the sum falls outside of the boundaries of the code. In the table below, valid codes for decimal numbers are from 0011_2 to 1100_2. Any answer that falls outside this range produces a code overflow.
- A word overflow exists when the sum produces a value that exceeds the largest value a word is able to represent. For the 4-bit word, this value is 1111_2.

Table 52. Binary Coded Decimal Excess-3

				D=3				X=4 P=7
Decimal				0	1	2	3	4
Code	0	1	2	3	4	5	6	7
Binary	0000	0001	0010	**0011**	**0100**	**0101**	**0110**	**0111**
	Code Overflow							
	Y=5 Q=8					**V=13**		
Decimal	5	6	7	8	9			
Code	8	9	10	11	12	13	14	15
Binary	**1000**	**1001**	**1010**	**1011**	**1100**	1101	1110	1111
						Code Overflow		

Let X and Y be two single digit decimal numbers. Let D be the Displacement of the Excess code, which is 3 for this system. The BCD XS-3 codes are

- P = X + 3 = X + Excess
- Q = Y + 3 = Y + Excess

To get the coded sum, S = (P + Q) – Excess = (P + Q) – D = (P + Q) – 3 = (X + Y) + Excess. If the value of S exceeds the valid code, a carry value C must be set to 1 and the value of S adjusted. There are several special cases as illustrated in the table below.

Table 53. Code and Word Overflow Carry Zones for BCD Excess-3 Addition

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	00	01	02	03	04	05	06	07	08	09
1	01	02	03	04	05	06	07	08	09	**10**
2	02	03	04	05	06	07	08	09	**10**	**11**
3	03	04	05	06	07	08	09	**10**	**11**	**12**
4	04	05	06	07	08	09	**10**	**11**	**12**	**13**
5	05	06	07	08	09	**10**	**11**	**12**	**13**	**14**
6	06	07	08	09	**10**	**11**	**12**	**13**	**14**	**15**
7	07	08	09	**10**	**11**	**12**	**13**	**14**	**15**	**16**
8	08	09	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
9	09	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**

In the addition table, the right digit is the coded sum, the left digit is the carry digit to be added to the next higher significant digit. Since the arithmetic is being done in binary, the carry digit is either a zero or one, and is a single binary digit (bit).

The sum of the BCD XS-3 values ((P – D) + Q) may result in an overflow condition which produces a wrong answer prior to adjustment. There is no code or word overflow for the upper left corner. The band that includes (5 + 5) has code overflow but no word overflow. The band that includes (6 + 7) has both word and code overflow. The bottom right corner that includes (8 + 8) has word overflow, but no code overflow. When either overflow condition occurs, the carry bit C is set to 1.

In the (Nassi-Schneiderman) flow chart for BCD Excess-3, the final Sum is S. Variables E and R are intermediate temporary variables. The value of E incorporates the value of the carry bit, which allows this addition procedure to apply for higher order positions of a number. V is the first code position past the decimal table. The quantity (D – V) adjusts the intermediate sum Result R to wrap around the fixed length table to obtain the correct BCD XS-3 Sum.

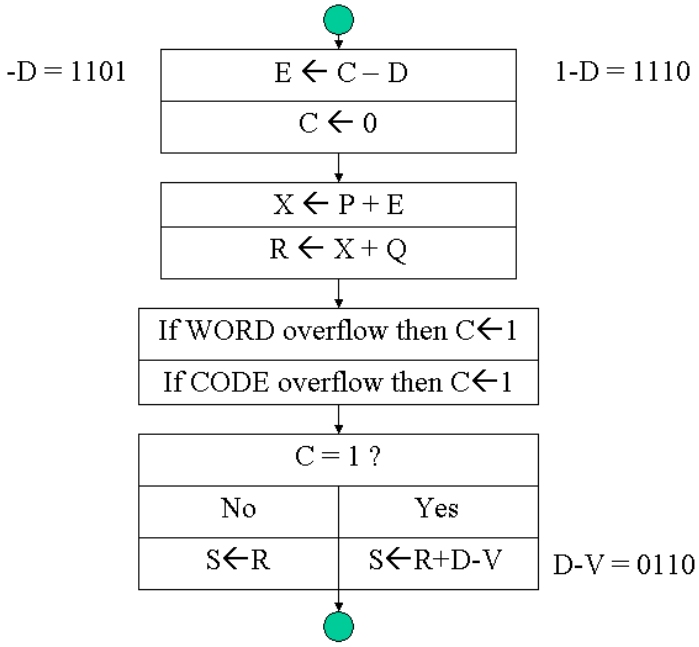


Figure 7. BCD Excess-3 Addition Flow Chart

BCD XS-3 Addition Examples with Input Carry = 0

When you look at examples given in tables below, also look at the table of Binary Coded Decimal Excess-3. Beginning with the code entry for the first number of an addition problem, count to the right in the table for the number of columns given by the second number of the addition problem. Note what happens if the result falls outside the range of permitted codes.

Table 54. Example Set 1 of BCD XS-3 Addition with Input Carry C = 0

Row		**5 + 2**			**5 + 4**			**5 + 7**			**5 + 9**
1	5	1000		5	1000		5	1000		5	1000
2	- D	1101		- D	1101		- D	1101		- D	1101
3		**10101**			**10101**			**10101**			**10101**
4	+ 2	0101		+ 4	0111		+ 7	1010		+ 9	1100
5	NWO NCO	**1010**		NWO NCO	**1100**		NWO **CO** **C←1**	**1111**		**WO** **CO** **C←1**	**10001**
6							+ D - V	0110			0110
7								**10101**			**0111**
8	C = 0	S = 7		C = 0	S = 9		C = 1	S = 2		C = 1	S = 4
9	Sum	**7**			**9**			**12**			**14**

In Row 2, D = Displacement of code = 3. Thus, - D = - 3, which is coded in two's complement in Row 2. In Row 5, WO = Word Overflow, NWO = No Word Overflow, CO = Code Overflow, NCO = No Code Overflow. In Row 8, C = Carry, S = Sum.

The overflow bit (left-most bit) in Row 3 is discarded and not used in the addition of the value shown in Row 4.

Problem (5+2) illustrates the case of no word overflow and no code overflow. It is a case that is not near a boundary of the code overflow identification table. Other examples are just before a boundary (5 + 4), (5 + 7), and (5 + 9). A later example will repeat these problems, but with an input carry C = 1, which will push the result over the boundary.

Problem (5 + 4), like (5 + 2), has no overflow.

Problem (5 + 7) shows the case of a code overflow, but no word overflow. Notice that the carry bit is set, C ← 1, in row 5. The overflow in Row 7 is ignored and does not affect setting the carry bit.

Problem (5 + 9) illustrates the case of both a word overflow and a code overflow. Anytime either overflow condition occurs, the carry bit is set, C ← 1.

Another example set includes the case of word overflow, but no carry overflow (9 + 9).

Table 55. Example Set 2 of BCD XS-3 Addition with Input Carry C = 0

Row		**9 + 0**			**9 + 3**			**9 + 6**			**9 + 9**
1	9	1100		9	1100		9	1100		9	1100
2	- D	1101		- D	1101		- D	1101		- D	1101
3		**11001**			**11001**			**11001**			**11001**
4	+ 0	0011		+ 3	0110		+ 6	1001		+ 9	1100
5	NWO NCO	**1100**		NWO **CO** **C←1**	**1111**		**WO** **CO** **C←1**	**10010**		**WO** NCO **C←1**	**10101**
6				+ D - V	0110		+ D - V	0110		+ D - V	0110
7					**10101**			**1000**			**1011**
8	C = 0	S = 9		C = 1	S = 2		C = 1	S = 5		C = 1	S = 8
9	Sum	**9**			**12**			**15**			**18**

Problem (9 + 9) illustrates the case of word overflow, but no code overflow.

BCD XS-3 Addition Examples with Input Carry = 1

The difference between this example and the case of Carry = 0 is that the adjustment in Row 2 has 1 added to it. The value of (1 – D = – D + 1) in binary is 1110_2. The effect of this change ripples through the examples for cases on the change boundary shown in the table for Code and Word Overflow Carry Zones for BCD Excess-3 Addition.

Table 56. Example Set 1 of BCD XS-3 Addition with Input Carry C = 1

Row		**5 + 2** **+ 1**			**5 + 4** **+ 1**			**5 + 7** **+ 1**			**5 + 9** **+ 1**
1	5	1000		5	1000		5	1000		5	1000
2	**1 - D**	1110		**1 - D**	1110		**1 - D**	1110		**1 - D**	1110
3		**10110**			**10110**			**10110**			**10110**
4	+ 2	0101		+ 4	0111		+ 7	1010		+ 9	1100
5	NWO NCO	**1011**		NWO **CO** **C←1**	**1101**		**WO** **CO** **C←1**	**10000**		**WO** **CO** **C←1**	**10010**
6				+ D - V	0110		+ D - V	0110		+ D - V	0110
7					**10011**			**0110**			**1000**
8	C = 0	S = 8		C = 1	S = 0		C = 1	S = 3		C = 1	S = 5
9	Sum	**8**			**10**			**13**			**15**

Compared to the same example with Carry C = 0, the Sum for this example with Carry = 1 is greater by 1.

The interesting case of the next example is the sum of (9 + 9) with input Carry C = 1. This is the problem of (9 + 10).

Table 57. Example Set 2 of BCD XS-3 Addition with Input Carry C = 1

Row		**9 + 0 + 1**			**9 + 3 + 1**			**9 + 6 + 1**			**9 + 9 + 1**
1	9	1100		9	1100		9	1100		9	1100
2	**1 - D**	1110		**1 - D**	1110		**1 - D**	1110		**1 - D**	1110
3		**11010**			**11010**			**11010**			**11010**
4	+ 0	0011		+ 3	0110		+ 6	1001		+ 9	1100
5	NWO **CO** **C←1**	**1101**		**WO** **CO** **C←1**	**10000**		**WO** NCO **C←1**	**10011**		**WO** NCO **C←1**	**10110**
6	+ D - V	0110		+ D - V	0110		+ D - V	0110		+ D - V	0110
7		**10011**			**0110**			**1001**			**1100**
8	C = 1	S = 0		C = 1	S = 3		C = 1	S = 6		C = 1	S = 9
9	Sum	**10**			**13**			**16**			**19**

Subtraction with BCD XS-3

BCD subtraction (X – Y) is done by adding the radix complement of the smaller magnitude number to the larger magnitude number, and applying the correct sign. The radix complement is taken of the whole number, not of individual digits! To do this, take the binary complement of each BCD XS-3 digit. Then, add 1 to the number, doing a carry to the next significant digit if necessary.

Table 58. Radix Complement of XS-3 Without Code Overflow

	Decimal	Excess-3	Remarks
x	302	0110 0011 0101	
$\overline{x}$	697	1001 1100 1010	1's Complement
$\overline{x}+1$	698	1001 1100 1011	Radix Complement

Table 59. Radix Complement of XS-3 With Code Overflow

	Decimal	Excess-3	Remarks
x	290	0101 1100 0011	
$\overline{x}$	709	1010 0011 1100	1's Complement
$\overline{x}+1$		1010 0011 1101	Add 1 Code Overflow.
		0000 0000 0110	XS-3 adjust: add 6
	710	1010 0100 0011	Note carry to next significant digit

BCD numbers are stored in signed magnitude format rather than storing negative numbers in radix complement format. This approach is useful for variable length numbers. This requires determination of the sign of the result.

Even if negative BCD numbers are stored in radix complement format, a sign is required. Recall for negative binary numbers in complement form, the range of numbers

is almost evenly split between positive and negative numbers. The most significant bit serves as a sign bit. The problem for BCD is that 10 numbers are coded into a 16 number code. At least a 19 number code is required to represent 9 positive and 9 negative numbers, and a signless zero. A sign or flag is needed to identify a number

Complement arithmetic can still be used for subtraction. One approach is to always prepare both BCD numbers in radix complement format before performing the operation. A hybrid approach tests signs and magnitudes, and take radix complements only if necessary. This is shown in the modified Nassi-Schneiderman chart below.

Subtraction (X – Y) of Signed Magnitude Represented Numbers

<table>
<tr><td colspan="3">Compare Sign(X) to Sign(Y)</td></tr>
<tr><td colspan="2">Sign(X) = Sign(Y)</td><td>Sign(X) ≠ Sign(Y)</td></tr>
<tr><td colspan="2">Compare Magnitudes</td><td rowspan="3"></td></tr>
<tr><td>|X| = |Y|</td><td>|X| < |Y|</td><td>|X| > |Y|</td></tr>
<tr><td rowspan="2">Z ← 0</td><td>Radix Complement (X)</td><td>Radix Complement (Y)</td></tr>
<tr><td>XS-3 Add</td><td colspan="2">XS-3 Add</td></tr>
<tr><td>Sign(Z) ← +</td><td>Sign(Z) ← Complement [Sign(X)]</td><td colspan="2">Sign(Z) ← Sign(X)</td></tr>
</table>

Figure 8. BCD Excess-3 Subtraction Flow Chart

If signs are fetched first, it is sometimes possible to avoid having to compare magnitudes.

Magnitude comparisons can be done while fetching digits. If lengths of numbers are known before scanning, it may be possible to determine the relative magnitudes before completely scanning both numbers by starting comparisons with the most significant digits. This requires storing the length of the number. This is an advantage if comparisons, without subsequent arithmetic, are a common need.

Using a finite length word for storing length of a BCD number necessarily places a limit on the number of digits the BCD number can have. For most business applications, this is not a practical problem. It is a problem for tasks that produce a number with many digits, such as computing pi.

BCD XS-3 Subtraction Example Same Sign and |X| = |Y|

Let X = 37 and Y = 37. Find Z = X – Y.

Then set Z to zero, and set the sign of Z to "positive". Problem finished.

BCD XS-3 Subtraction Example Same Sign and |X| < |Y|

Let X = 302 and Y = 608. Find Z = X – Y = – 306. Let the finite word length be 3 decimal digits long.

Let the internal BCD XS-3 code be P = X + 3, Q = Y + 3. Let the BCD XS-3 code for the sum be S = Z + 3. Let T = Radix Complement, and W = T in BCD XS-3 Code.

Find the radix complement of X.

	Decimal		BCD XS-3 Binary
X	302	P=	0110 0011 0101$_{\text{BCD XS-3}}$
$\overline{X}$	697		1001 1100 1010$_{\text{BCD XS-3}}$
T= $\overline{X}+1$	698	W=	1001 1100 1011$_{\text{BCD XS-3}}$

Add [Radix Complement(X)] + Y.

	Decimal			W_2	W_1	W_0
T= $\overline{X}+1$	698		W	1001	1100	1011
				Q_2	Q_1	Q_0
Y	608		Q	1001	0011	1011

Using a finite length word of 3 decimal digits, discard the overflow digit for Z to produce the result of addition as Z = 306.

Perform the binary computation according to the BCD Addition flow chart, working from right to left, one BCD XS-3 digit at a time. To avoid ambiguity, use the same subscript of digit also for a Carry In to belong to that digit. The subscript for a Carry Out is for the next higher significant digit.

Table 60. BCD XS-3 Subtraction for Same Sign and |X|<|Y|

698 +608		T_2+Y_2 $+C_2$	**6 + 6** **+ 1**		T_1+Y_1 $+C_1$	**9 + 0** **+ 1**		T_0+Y_0 $+C_0$	**8 + 8** **+ 0**
1	W	W_2	1001		W_1	1100		W_0	1011
2		**1 - D**	1110		**1 - D**	1110		**- D**	1101
3	T	T_2	**10111**		T_1	**11010**		T_0	**11000**
4	Q	Q_2	1001		Q_1	0011		Q_0	1011
5	**R**	**WO** R_1**: CO** **$C_3 \leftarrow 1$**	**10000**		NWO R_1**: CO** **$C_2 \leftarrow 1$**	**1101**		**WO** R_0: NCO **$C_1 \leftarrow 1$**	**10011**
6		+ D - V	0110		+ D - V	0110		+ D - V	0110
7	S		**0110**			**10011**			**1001**
8		$C_3 = 1$	$S_2 = 3$		$C_2 = 1$	$S_1 = 0$		$C_1 = 1$	$S_0 = 6$
9		Sum	**13**			**10**			**16**

Finally, the sign of the answer is the complement of the sign of X.

Sign(Z) = Complement[Sign(X)] = negative sign: – .

The final answer is **Z = : – 306**.

BCD XS-3 Subtraction Example Same Sign and |X| > |Y|

Let X = – 461 and Y = – 320. Find Z = X – Y = – 141. Let the finite word length be 3 decimal digits long.

Let the internal BCD XS-3 code be P = |X| + 3, Q = |Y| + 3. Let the BCD XS-3 code for the sum be S = Z + 3. Let T = Radix Complement, and W = T in BCD XS-3 Code.

Find the radix complement of Y.

	Decimal		BCD XS-3 Binary
|Y|	320	Q=	$0110\ 0101\ 0011_{\text{BCD XS-3}}$
$\overline{\lvert Y \rvert}$	679	$\overline{Q}$ =	$1001\ 1010\ 1100_{\text{BCD XS-3}}$
T= $\overline{\lvert Y \rvert}$ +1	680	W'=	$1001\ 1010\ 1101_{\text{BCD XS-3}}$ **Notes: Code Overflow in T_0. Generate C_1 to next digit.**

Concentrate on the right-most digit. Apply the rules for BCD XS-3 Addition for the case of Code Overflow to complete the process of obtaining the radix complement of Y.

1001	1010	1101	W' from above
		0110	Add Code Overflow adjustment of + D – V
		1 0011	C_1 and W_0. Add carry C1 to next significant BCD XS-3 digit.
1001	1011	0011	W

Add X + [Radix Complement(Y)]. Work on right column completely, then move to the next column to the left, applying the proper displacement –D or 1–D if a carry C=1 was generated from the previous column.

Table 61. BCD XS-3 Subtraction for Same Sign and |X|>|Y|

	Decimal								
|X|	461			P_2	0111	P_1	1001	P_0	0100
Displacement				1 – D	1110	– D	1101	– D	1101
			|X|	$\|X\|_2$	10101	$\|X\|_1$	10110	$\|X\|_0$	10001
$T=\left\|\overline{Y}\right\|+1$	680		W	W_2	1001	W_1	1011	W_0	0011
R←|X| + W			R	R_2	1110	R_1	10001	R_0	0100
					Code Overflow C_3←1		Word & Code Overflow C_2←1		No Overflow C_1←0
				D – V	0110	D – V	0110		
				R_2+D-V	10100	R_1+D-V	0111	R_0	0100
			S	S_2	4	S_1	7	S_0	4
|Z|	**141**			$\|Z\|_2$	**1**	$\|Z\|_1$	**4**	$\|Z\|_0$	**1**

Using a finite length word of 3 decimal digits, discard the overflow digit for Z.

Sign(Z) ←Sign(X) = negative = –.

Z = – 141.

Binary Multiplication

Multiplication in binary follows the same procedural rules as multiplication in base ten. It is simpler because the multiplication table is much smaller. Remember how long it took to memorize the multiplication table? The binary table is much easier! The inputs are X and Y.

Multiply		X	
		0	**1**
Y	**0**	0	0
	1	0	1

Example 1: Multiply 11010_2 x 10111_2 = 26 x 23 = 598. In the example below, reserve the row immediately below the multiplier to hold the carry when adding the columns. Label the columns with the position number, beginning from the right hand side. The row labeled "Column 0" is the result of multiplying X by the digit found in Column 0 of Y. The row labeled "Column 1" is the result of multiplying X by the digit found in Column 1 of Y. After multiplying X by each digit of Y, add the resulting rows. Be sure to account for any carries from the previous column. The number of digits in the answer will be the total number of digits in the individual multipliers.

Column Position Number						4	3	2	1	0
X						**1**	**1**	**0**	**1**	**0**
Y						**1**	**0**	**1**	**1**	**1**
Carry	1	1	1	10	1	1	0	0	0	
Column 0						1	1	0	1	0
Column 1					1	1	0	1	0	
Column 2				1	1	0	1	0		
Column 3			0	0	0	0	0			
Column 4		1	1	0	1	0				
Sum	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**

Check the answer by converting 1001010110_2 to decimal.

Place Value	512	256	128	64	32	16	8	4	2	1
Sum	1	0	0	1	0	1	0	1	1	0

The answer is 598.

Example 2. Multiply **11011.011_2** by **101.110_2** = 27.375 x 5.75 = 157.40625.

Column Position Number							4	3	2	1	0	-1	-2	-3
X							**1**	**1**	**0**	**1**	**1.**	**0**	**1**	**1**
Y									**1**	**0**	**1.**	**1**	**1**	**0**
Carry	1	1	1	10	10	10	10	10	1	1	1	0	0	
Column -3							0	0	0	0	0	0	0	0
Column -2						1	1	0	1	1	0	1	1	
Column -1					1	1	0	1	1	0	1	1		
Column 0				1	1	0	1	1	0	1	1			
Column 1			0	0	0	0	0	0	0	0				
Column 2		1	1	0	1	1	0	1	1					
Sum	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1.**	**0**	**1**	**1**	**0**	**1**	**0**

Check the answer by converting 1001 1101.0110 10_2 to decimal.

Place Value	128	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	1/32	1/64
Sum	1	0	0	1	1	1	0	1	0	1	1	0	1	0

The answer is 157.40625 .

Just as with decimal multiplication, the number of digits to the right of the radix point is the same as the number of digits to the right of the radix point in both original numbers. In this example, X has 3 digits to the right of the radix point (which you can call the binary point), and Y has 3 digits to the right of the radix point. The answer has 6 digits to the right of the radix point.

Similarly, the number of digits to the left of the radix point is the same as the number of digits to the left of the radix point in both original numbers. In this example, X has 5 digits to the left of the radix point, and Y has 3 digits to the left of the radix point. The answer has 8 digits to the left of the radix point.

Some methods of multiplication are faster than others. One example is the Booth Algorithm, which is used in some designs of arithmetic logic units.[60] You can view a simulation of multiplication using this method at the U. Massachusetts web site.[61]

[60] Yap Siong Chua, CDA 4101 class notes, U. North Florida, Jacksonville, FL (1981).
[61] Israel Koren, "Booth's Algorithm", (09 December 2005).
http://www.ecs.umass.edu/ece/koren/arith/simulator/Booth/ Visited 04 January 2008.

Binary Multiplication

Multiply 11101_2 by 10110_2. = 29 x 22. Show all your work. The answer is 638.

Column Position Number						4	3	2	1	0
X						**1**	**1**	**1**	**0**	**1**
Y						**1**	**0**	**1**	**1**	**0**
Carry										
Column 0										
Column 1										
Column 2										
Column 3										
Column 4										
Sum										

Check the answer by converting it to decimal.

Place Value	512	256	128	64	32	16	8	4	2	1
Sum										

Answer =

Hexadecimal Multiplication

Remember memorizing the times-ten table? You can do the same for hexadecimal. The multiplication table for hexadecimal is given below. Use this for hexadecimal multiplication just as you use the decimal multiplication table for decimal multiplication.

Table 62. Hexadecimal Multiplication Table

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Example: Multiply **X** = **0x**A4B2 times **Y** = $C3_{16}$.

Sum Carry		1	1			
Carry term 0		1		2		
Carry term 1	7	3	8	1		
X			**A**	**4**	**B**	**2**
times Y					**C**	**3**
			E	C	1	6
		8	0	4	8	
Answer	**7**	**D**	**7**	**3**	**9**	**6**

The answer is 7D7396**H**.

In the above example, “Carry term 0” contains the carry terms from multiplying by 3. “Carry term 1” contains the carry terms from multiplying by C. “Sum Carry” contains the carry terms from addition in the adjacent right column.

Hexadecimal Multiplication Problem

Problem 1. Multiply X = FADH times EB_{16}.

Sum Carry				
Carry term 0				
Carry term 1				
X		**F**	**A**	**D**
times Y			**E**	**B**

Answer

Answer =

Fixed Point Binary Division

Division in binary is similar to division in decimal.

Example: Divide 10011101.01101_2 by 11011.011_2 . The first step is to normalize the division problem to make the divisor an integer. This is a nice, but not necessary, step. This transforms the problem into dividing

$$10011101011.01_2 \text{ by } 11011011_2$$

								1	**0**	**1.**	**1**	**1**
11011011	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**1.**	**0**	**1**
		1	1	0	1	1	0	1	1				
			1	0	1	1	1	1	1	1	1		
				1	1	0	1	1	0	1	1		
				1	0	1	0	0	1	0	0	0	
					1	1	0	1	1	0	1	1	
						1	1	0	1	1	0	1	1
						1	1	0	1	1	0	1	1
						0	0	0	0	0	0	0	0

The answer is 101.11_2 . This result is consistent with Example 2 of fixed point multiplication.

Fixed Point Binary Division Problems

Problem 1. Divide 101.110011_2 by 11.0101_2 .

								.	
110101	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**.1**	**1**

Common Fractions in Different Bases

Fractions cannot be exactly represented in a finite number of digits in all number systems. For some kinds of work, one number system may be better than another number system. For example, the fraction 1/3 cannot be represented in base 10 using a finite number of digits. Some fractions can be represented by repeating decimals, and a bar is placed on top of that group of digits that repeats infinitely. We therefore represent 1/3 in decimal as

$$\frac{1}{3} = 0.333333333\ldots = 0.\overline{3} = 0.\underline{3}$$

Not all numbers with fractional parts can be represented in this way because the fractional part is not a repeating decimal. The numbers e and π are famous examples.

When you look at the stock market, you see the fractional part of prices listed in multiples of negative powers of two for the high, low, last, and change, such as 5/8 = 0.101, 3/4 = 0.11, 1/2 = 0.1. Use of these fractions permitted stock price data to be stored in binary on computers exactly, before the invention of Binary Coded Decimal. Stock markets in the United Stated converted from fractional to decimal systems in early 2001. You can learn more about the stock market on the Internet at http://www.nyse.com.

Fraction	**1/2**	**1/4**	**1/8**	**1/16**	**1/32**
Power of 2	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
Binary Fraction	0.10000	0.01000	0.00100	0.00010	0.00001

Here are some examples of fractions expressed in different bases. Fractions with exact representation with a finite number of digits are indicated in bold. Underlined groups of digits are repeating digits. The proper notation is to have a solid over-bar. Neither MS Equation 3.0, nor Corel Equation 2.0, can do that for groups of selected digits.

Table 63. Ordinal Fractions in Different Bases

Ordinal Fractions							
Fraction	Base 2	Base 3	Base 5	Base 7	Base 10	Base 12	Base 16
1/2	**0.1**	0.1	0.2	0.3	**0.5**	**0.6**	**0.8**
1/3	0.01	**0.1**	0.13	0.2	0.3	**0.4**	0.5
1/4	**0.01**	0.02	0.1	0.15	**0.25**	**0.3**	**0.4**
1/5	0.0011	0.0121	**0.1**	0.1254	**0.2**	0.2497	0.3
1/6	0.001	0.01	0.04	0.1	0.6	**0.2**	0.2A
1/7	0.001	0.012	0.032412	**0.1**	0.1428571	0.186A35	0.249
1/8	**0.001**	0.01	0.03	0.06	**0.125**	**0.16**	**0.2**
1/9	0.000111	**0.01**	0.021	0.053	0.1	**0.14**	0.15
1/10	0.00011	0.0022	0.02	0.0462	**0.1**	0.12497	0.19

Table 64. Decimal Fractions in Different Bases

Decimal Fractions							
Fraction	Base 2	Base 3	Base 5	Base 7	Base 10	Base 12	Base 16
1/10	0.00011	0.0022	0.02	0.0462	**0.1**	0.12497	0.19
2/10	0.0011	0.0121	**0.1**	0.1254	**0.2**	0.2497	0.3
3/10	0.01001	0.0220	0.12	0.2046	**0.3**	0.37249	0.4C
4/10	0.0110	0.1012	**0.2**	0.2543	**0.4**	0.4972	0.6
5/10	**0.1**	0.1	0.2	0.3	**0.5**	**0.6**	**0.8**
6/10	0.1001	0.1210	**0.3**	0.43	**0.6**	0.7249	0.9
7/10	0.10110	0.2002	0.32	0.4620	**0.7**	0.84972	0.B3
8/10	0.1100	0.2101	**0.4**	0.543	**0.8**	0.9724	0.C
9/10	0.11100	0.2200	0.42	0.6204	**0.9**	0.A9724	0.E6

Failure to account for the inability to exactly represent decimal fractions using fixed length binary words in the design of the Patriot missile software resulted in the loss of 28 lives in a Scud attack on 25 FEB 1991 on the American barracks in Dharan, Saudi Arabia during the Gulf War. See the article by Arnold.[62]

Beware of thinking that base 10 is necessarily the best based upon observing that so many of the fractions above can be expressed exactly with a finite length number. If God had given us 7 digits, instead of 10, at the end of our upper appendages, we would be more inclined to prefer the fractions 1/7, 2/7. 3/7, 4/7, 5/7, 6/7, and 1 = 7/7.

[62] Douglas N. Arnold, "The Patriot Missile Failure" (23 AUG 2000). http://www.ima.umn.edu/~arnold/disasters/patriot.html Visited 04 JUN 2002.

Prime Numbers and Number System Base

One integer is considered "divisible" by another integer only if the remainder from the division is zero. A positive integer is called "prime" if it is divisible only by itself and one. One is declared to not be a prime number.

The properties of the set of integers are independent of the base of the number system used to express them. The base of the number system in use does not affect the set of prime numbers. It does not alter which numbers are prime. It does not alter the number of prime numbers.

There is no "largest" prime number. For every number, there is a larger prime number. The "infinitude of primes" was proven by Euclid in 300 B.C., and is called "Euclid's Second Theorem". There is an infinite number of prime numbers. These facts keep the National Security Agency in business, and keep cryptologists busy and happy (or unhappy). A statement and proof of Euclid's Second Theorem can be found on the Internet.[63]

Historical context of important discoveries are interesting. Euclid lived approximately 325 BC to 265 BC. Euclid of Alexandria was born 2 years before Alexander the Great died in Babylon (325 B.C.), and 3 years after Aristotle died (322 B.C.). Aristotle had been the personal tutor of Alexander the Great, and was a student of Plato.[64] There is a strong conjecture that Euclid studied at Plato's Academy in Athens.[65] Euclid was alive during the reigns of Ptolemy I and Ptolemy II in Egypt. Alexandria, Egypt, was the intellectual capital of the world. During this period, Ptolemy II ordered the translation of Jewish Law, producing the Septuagint, one of the most influential translations ever to be accomplished. Euclid died in Alexandria. Elsewhere, it was during this period that the Roman Aqueduct was built.

Here are the first five prime numbers, expressed in different bases.

---

[63] Chris K. Caldwell, "Euclid's Proof of the Infinitude of Primes (c. 300 BC)", The University of Tennessee at Martin. http://primes.utm.edu/notes/proofs/infinite/euclids.html Visited 04 January 2008.

[64] Aine Donovan, David E. Johnson, George R. Lucas, Jr., and Paul E. Roush (eds.), *Ethics for Military Leaders*, Simon and Schuster (1998).

[65] J J O'Connor and E F Robertson, "Euclid of Alexandria", School of Mathematics and Statistics, University of St. Andrews, Scotland (January 1999). http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Euclid.html, 13 September 2002.

Table 65. First 12 Prime Numbers in Different Bases

First Dozen Prime Numbers in Different Bases						
Base 10	Base 2	Base 3	Base 5	Base 7	Base 12	Base 16
2	10	2	2	2	2	2
3	11	10	3	3	3	3
5	101	12	10	5	5	5
7	111	21	12	10	7	7
11	1011	102	21	14	B	B
13	1101	111	23	16	11	D
17	10001	122	32	23	15	11
19	10011	201	34	25	17	13
23	10111	212	43	32	1B	17
29	11101	1002	104	41	25	19
31	11111	1011	111	43	27	1F
37	100101	1101	122	41	31	25

If an integer is not a prime number, it can be divided by a smaller prime number. Consider the number 19. Try the division in different bases.

Base 2 Demonstration that 19 is prime.

Divide by 2

```
          1  0  0  1
      ---------------
10 | 1  0  0  1  1
     1  0
     ----------
        0  0  1  1
           1  0
           ------
        R=    1
```

Divide by 3

```
             1  1  0
         ------------
11 | 1  0  0  1  1
        1  1
        ---------
           1  1
           1  1
           ------
              0  1
                 0
                 ---
           R=    1
```

Divide by 5

```
             1  1
         ---------
101 | 1  0  0  1  1
        1  0  1
        ---------
           1  0  0  1
              1  0  1
              -------
           R2= 1  0  0
           R10 =4
```

Divide by 7

```
                1  0
            ---------
111 | 1  0  0  1  1
        1  1  1
        ---------
              1  0  1
                 0
                 ---
           R2= 1  0  1
           R10 =5
```

11

					1
1011	**1**	**0**	**0**	**1**	**1**
		1	0	1	1
	R_2=	**1**	**0**	**0**	**0**
	R_{10}	=8			

13

					1
1101	**1**	**0**	**0**	**1**	**1**
		1	1	0	1
	R_2=	**1**	**1**	**0**	
	R_{10}	=6			

17

					1
10001	**1**	**0**	**0**	**1**	**1**
	1	0	0	0	1
			R_2=	**1**	**0**
			R_{10}	=2	

19

					1
10011	**1**	**0**	**0**	**1**	**1**
	1	0	0	1	1
				R_2=	**0**
				R_{10}	=0

The first divisor not equal to one that results in a remainder of zero is $10011_2 = 19_{10}$. Therefore, this is a prime number.

Base 3 Demonstration that 19 is prime.

Divide by

2

	1	**0**	**0**
2	**2**	**0**	**1**
	2		
			1
			0
	R_3=		**1**
	R_{10}		=1

3

		2	**0**
10	**2**	**0**	**1**
	2	0	
			1
			0
		R_3=	**1**
		R_{10}	=1

Divide by

5

		1	0
12	**2**	**0**	**1**
	1	2	
		1	1
			0
	R_3=	**1**	**1**
	R_{10}	=4	

7

			2
21	**2**	**0**	**1**
	1	0	2
	R_3=	**1**	**2**
		R_{10}	=5

11

			1
102	**2**	**0**	**1**
	1	0	2
	R_3=	**2**	**2**
	R_{10}	=8	

13

111	**2**	**0**	**1**
	1	1	1
	R_2=	**2**	**0**
	R_{10}	=6	

17

			1
122	**2**	**0**	**1**
	1	2	2
		R_2=	**2**
		R_{10}	=2

19

			1
201	**2**	**0**	**1**
	2	0	1
		R_2=	**0**
		R_{10}	=0

The first divisor not equal to one that results in a remainder of zero is $201_3 = 19_{10}$. Therefore, this is a prime number.

Base 5 Demonstration that 19 is prime.

Divide by

2

		1	**4**
2	**3**	**4**	
	2		
	1	4	
	1	3	
	R_5=	**1**	
	R_{10}	=1	

3

		1	**1**
3	**3**	**4**	
	3		
	0	4	
		3	
	R_3=	**1**	
	R_{10}	=1	

5

			3
5	3	4	
	3	0	
	R_3=	**4**	
	R_{10}	=4	

Divide by

7

			2
12	**3**	**4**	
	2	4	
R_3=	**1**	**0**	
	R_{10}	=5	

11

			1
21	**3**	**4**	
	2	1	
R_3=	**1**	**3**	
R_{10}	=8		

Divide by

13

		1
23	**3**	**4**
	2	3
R_3=	**1**	**1**
	R_{10}	=6

17

		1
32	**3**	**4**
	3	2
	R_3=	**2**
	R_{10}	=2

19

		1
34	3	4
	3	4
	R_3=	**0**
	R_{10}	=0

The first divisor not equal to one that results in a remainder of zero is $34_5 = 19_{10}$. Therefore, this is a prime number.

Prime Number Problems

Problem 1. a. Show that 7 is a prime number using base 2.
b. Show that 7 is a prime number using base 3.
c. Compare the results.

Floating Point Arithmetic

The general concept is called "exponential notation". There is an extensive collection of links to web sites on the subject of exponential notation.[66]

Fixed Point Numbers with Fractions

Define a **fixed point number** as a number written in positional notation that contains a **radix point** (a period in the United States, or a comma in Europe). The number immediately to the left of the radix point is in the units position. In the decimal system, the special name of the radix point is "decimal point". When the fixed point number has no fractional part, we call it an integer. Fixed point notation is the notation normally used by most people in their every-day life, and in business and finance. Examples of some approximated numbers written in fixed point notation are:

14.5939 kg = 1 slug
$\pi \approx$ **3.1415 92653 5897 32384 62643**
e $\approx$ **2.7 1828 1828 45 90 45 2353 60287**
$\gamma \approx$ **0.5772 15664 90153 28606 06512**

Some useful numbers are very large or very small, and writing such numbers in fixed point notation is inefficient and inconvenient. There are other ways of writing very large and very small numbers: scientific notation, and floating point notation.

Business data processing often uses fixed point numbers with fractions for currency. The internal representation for fixed point numbers on computers usually does not provide for recording the position of a radix point. The program must keep track of the proper position, and use formatting statements to position the radix point properly.

Scientific Notation

Scientific notation permits us to write very large numbers and very small numbers using a compact notation. A number written in scientific notation consists of several parts. An example of a number written in scientific notation is

$$\mathbf{6.02216\ 9} \times 10^{\mathbf{23}}\ \text{mol}^{-1}$$

This number is known as Avogadro's number.[67] It is the number of elementary entities (molecules) in one mole of a substance (referenced to the number of atoms in 12 grams of Carbon-12).[68]

In the example above, the "**6.022169**" portion of the number is called the **mantissa,** significand, or coefficient. At least one author called it the "base", but that name can cause confusion with some other concepts. Except for the number zero, the

---

[66] Mitchell N. Charity, "An Exponential Notation Meta Page" (10 August 2003). http://www.vendian.org/mncharity/export1/exponential_notation_meta/ Visited 04 January 2008.
[67] *Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables*, Applied Mathematics Series, AMS-55, Milton Abramowitz and Irene A. Stegun (eds.), National Bureau of Standards, U.S. Department of Commerce, U.S. Government Printing Office (June 1964).
[68] "Unit of Amount of Substance (Mole)", *The International System of Units (SI)*, Section 2.1.1.6:, Page 8, NIST Special Publication 330, 2001 Edition, Barry N. Taylor (editor), National Institute of Standards and Technology (July 2001).

mantissa of a number written in scientific notation has one non-zero number to the left of the decimal point.

The "**23**" portion of the number is called the **characteristic** or exponent. The exponent can be a positive or negative integer. When the exponent is zero, the number is written without the exponent term. The number "10" is the exponent base.

Avagadro's number is

$$6022\ 16900\ 00000\ 00000\ 00000 = 6.02216\ 9 \times 10^{23}$$

Very small numbers written in scientific notation have negative powers of the base, such as the gravitational constant,

$$G = 6.6732 \times 10^{-11}\ \text{N m}^2 / \text{kg}^2$$

The number

$$-\ q = -\ 1.60219\ 17 \times 10^{-19}\ \text{C}$$

is the elementary charge of an electron, measured in Coulombs. The mantissa is "-1.6021917". The characteristic is "-19". The charge of an electron is negative, as indicated by the minus sign of the mantissa. The value of the charge of a single electron is very much less than one, as indicated by minus sign in the characteristic. Written in fixed point notation, this number is

$$-\ 0.\ 00000\ 00000\ 00000\ 00016\ 02191\ 7$$

Scientific Notation Conversion Problems

Problem 1. Write $h = 0.00000\ 00000\ 00000\ 00000\ 00000\ 00000\ 00066\ 26196$ J/s in scientific notation. This is the Plank constant.

Problem 2. Write $c = 2997\ 92500$ m/s in scientific notation. This is the speed of light in a vacuum.

Scientific Notation Multiplication

Multiplication and division using scientific notation is easy. The charge of electrons in one mole of hydrogen is the product of the charge per electron times the number of electrons in a mole of hydrogen. The mantissas are multiplied, and the characteristics are algebraically added.

$$(6.02216\ 9 \times 10^{23}\ \text{mol}^{-1}) \times (-\ 1.60219\ 17 \times 10^{-19}\ \text{C})$$
$$= (6.02216\ 9) \times (-\ 1.60219\ 17) \times 10^{23\ -\ 19}\ \text{C/mol}$$
$$= -\ 9.\ 64866\ 91879\ 797 \times 10^{4}\ \text{C/mol}$$

That was much easier than multiplying

$$6022\ 16900\ 00000\ 00000\ 00000 \times 0.\ 00000\ 00000\ 00000\ 00016\ 02191\ 7$$

Here is another example for multiplication.

$$(3.0 \times 10^{3}) \times (6.0 \times 10^{2}) = (3.0 \times 6.0) \times 10^{3\ +\ 2} = 18.0 \times 10^{5}$$

Scientific Notation Multiplication Problems

Problem 1. Approximate the distance of one light-year.[69]

(2.998 x 10^{8} m/s) x (3.147 x 10^{7} s/yr) =

Problem 2. Naively estimate the kinetic energy of the earth revolving around the Sun, assuming a circular orbit (ignoring Kepler).[70]

(1/2) x (5.98 x 10^{24} kg) x (2.99 x 10^{4} m/s)^{2} =

Scientific Notation Normalization

Notice that the form of 18.0 x 10^{5} is not quite in scientific notation. The mantissa has more than one digit to the left of the decimal point. The process of reformatting the number into scientific notation is called **normalization**.

$$18.0 \times 10^{5} = (1.8 \times 10^{1}) \times 10^{5} = 1.8 \times (10^{1+5}) = 1.8 \times 10^{6}$$

Scientific Notation Division

Division is almost as easy. Consider dividing 1.5 x 10^{4} by 3.0 x 10^{5}. The procedure is to divide the mantissas, algebraically add the characteristics [(+4) + (-5) in the example below], and normalize the result.

$$\frac{1.5 \bullet 10^{4}}{3.0 \bullet 10^{5}} = \left(\frac{1.5}{3.0}\right) \bullet 10^{4-5} = \left(\frac{1}{2}\right) \bullet 10^{-1} = 0.5 \bullet 10^{-1} = \left(5.0 \bullet 10^{-1}\right) \bullet 10^{-1} = 5.0 \bullet 10^{-2}$$

Scientific Notation Addition and Subtraction

Addition and subtraction using scientific notation require more effort. Numbers must be written in a form having the same exponent before performing addition or subtraction. For example:

$$(2.1 \times 10^{3}) + (1.4 \times 10^{2}) = (21.0 \times 10^{2}) + (1.4 \times 10^{2})$$
$$= 22.4 \times 10^{2} = 2.24 \times 10^{3}$$

It is also correct to write

$$(2.1 \times 10^{3}) + (1.4 \times 10^{2}) = (2.1 \times 10^{3}) + (0.14 \times 10^{3}) = 2.24 \times 10^{3}$$

Here is another example:

$$(3.2 \times 10^{3}) - (1.1 \times 10^{4}) = (0.32 \times 10^{4}) - (1.10 \times 10^{4})$$
$$= -0.78 \times 10^{4} = -7.8 \times 10^{3}$$

It is easy to lose precision when adding or subtracting numbers with very different exponents. For example:

$$2 \times 10^{10} - 0.5 \times 10^{-10}. = 2\ 00000\ 00000 - 0.00000\ 00000\ 5$$
$$= 1\ 99999\ 99999\ .99999\ 99999\ 5$$

---

[69] Encyclopedia.com, "Light Year", http://www.encyclopedia.com. 01 July 2002.

[70] George Goth, "The Magnitudes of Physics", Skyline College, San Bruno CA (08 Aug 2001) http://www.smccd.net/accounts/goth/MainPages/magphys.htm, 01 July 2002.

In physics, it is common to be interested in very small differences of very large numbers. Consider the pressure of the atmosphere on the surface of the earth. When we hear sound, we are sensing very small variations in that pressure.

Engineering Notation

Engineering notation is similar to scientific notation, except that the characteristic is always a multiple of 3. Because of this, the mantissa may have one to three digits to the left of the decimal point.

Examples:

Fixed Point	Engineering	Scientific
13,200,000,000.0	13.2×10^{9}	1.32×10^{10}
0.000 000 325	325×10^{-9}	3.25×10^{-7}
102,200,000,000,000,000,000,000	102.2×10^{21}	1.022×10^{23}

The rule for choosing multiples of three is a normalization rule. Notice that engineering notation is naturally related to the practice in the United States to form groups of three digits to aid in quickly determining the value of a number. This is reinforced both in engineering and the SI units of measure by the language adopted for number prefixes.

Table 66. SI Units of Measure Prefixes

tera	giga	mega	kilo		milli	micro	nano	pico
$\times 10^{12}$	$\times 10^{9}$	$\times 10^{6}$	$\times 10^{3}$	$\times 10^{0}$	$\times 10^{-3}$	$\times 10^{-6}$	$\times 10^{-9}$	$\times 10^{-12}$

Floating Point Notation

Floating point numbers are like numbers in scientific notation. It is all in binary. The radix point (the period) is either immediately to the right or to the left of the most significant digit, depending on the specific processor. An example of a floating point number is:

$$0.11 \times 2^{110}$$

The binary fraction 0.11_2 is the mantissa. Convert its value to decimal as follows:

$$0.11_2 = 2^{-1} + 2^{-2} = \frac{1}{2} + \frac{1}{4} = \frac{3}{4} = 0.75$$

The binary exponent 110_2 equals decimal 6. The value of the exponent term is

$$2^{6} = 64_{10}$$

The decimal value of the original number is

$$0.75 \times 64 = 48$$

On a different processor, this same number might have a floating point representation as

$$1.1 \times 2^{109}$$

Floating Point Multiplication

Multiplication with floating point numbers is similar to multiplication with scientific notation. Multiply the mantissas and algebraically add the characteristics.

Example:

$$(0.11 \times 2^{110}) \times (0.1001 \times 2^{100}) = (0.11 \times 0.1001) \times (2^{110 + 100})$$

Perform binary addition for the exponents using the binary addition table.

Carry	1	0	0	
		1	**1**	**0**
+		**1**	**0**	**0**
Answer	1	0	1	0

When arithmetic is done with floating point numbers, the answer is placed back into standard form after the operation is finished. Just before normalization, the answer is 0.011011×2^{1010}. The process of normalization adjusts the characteristic in order to place the most significant digit immediately to the right of the radix point.

$$0.011011 \times 2^{1010} = (0.11011 \times 2^{-1}) \times 2^{1010} = 0.11011 \times 2^{1001}$$

Floating Point Division

Floating point division is similar to scientific notation division. The procedure is to divide the mantissas, algebraically add the characteristics, and normalize the result.

Example 1. Divide (0.1001×2^{1001}) by (0.1101×2^{101}).

$$= \left(\frac{0.1001}{0.1101} \right) \bullet 2^{1001-101}$$

Adding the characteristics algebraically is the easy part.

$$2^{1001-101} = 2^{100}$$

The division is more tedious.

```
0.1101 | 0. 1 0 0 1 0 0 0 0 0 0 0 0
```

The first step is to shift the radix point in both terms to the right by the same number of places until the divisor is an integer. In this example, shift the radix point by 4 places. In fractional notation, this is

$$\frac{0.1001}{0.1101} = \frac{1001}{1101}$$

```
                        .1 0 1 1 0 0 0 1 0 0 1 0 0 1
1101 | 1 0 0 1 .0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
         1 1 0 1
       ---------
           1 0 1 0 0
           1 1 0 1
         ---------
               1 1 1 0
               1 1 0 1
             -----------
                   1 0 0 0 0
                     1 1 0 1
                   -----------
                         1 1 0 0 0
                           1 1 0 1
                         -----------
                               1 1 0 0 0
```

The answer is

$$0.1011\ 0001\ 0010\ 01 \times 2^{100}$$
$$= \mathbf{0.1011\ \underline{0001} \times 2^{100}}$$
$$= 1011.\underline{0001}$$

Notes:

$0.1001 \times 2^{1001} = 1001 \times 2^{101} = 9 \times 2^5 = 9 \times 32 = 288$
$0.1101 \times 2^{101} = 1101 \times 2^1 = 13 \times 2 = 26$
$288/26 = 11.\underline{076923}$

Example 2. Divide (0.11101×2^{110}) by (0.101×2^{1101}),

$$\left(\frac{0.11101 \bullet 2^{110}}{0.101 \bullet 2^{1101}}\right) = \left(\frac{0.10111}{0.101}\right) \bullet 2^{110-1101}$$

The characteristic is $110 - 1101 = -111$.
For the mantissa:

0.101 ⟌ **0. 1 1 1 0 1 0 0 0 0 0 0 0**

Shift the radix point in both terms to the right by the same number of places until the divisor is an integer. In this example, shift the radix point by 3 places. Perform the division.

```
                  1  .0  1  1  1  0  0  1
          ┌──────────────────────────────
101       │ 1  1  1  .0  1  0  0  0  0  0
          1  0  1
          ──────
             1  0  0  1
             1  0  1
             ──────
             1  0  0  0
                1  0  1
                ──────
                   1  1  0
                   1  0  1
                   ──────
                      1  0  0  0
                         1  0  1
                         ──────
                         1  1  1
```

The result of the division is $1.0111\ 00 \times 2^{-101}$. Normalize the resulting floating point notation:

$$1.0111\ 00 \times 2^{-101} = (0.1011\ 100 \times 2^{1}) \times 2^{-101}$$
$$= 0.1011\ 100 \times (2^{1} \times 2^{-101})$$
$$= \mathbf{0.1011\ 100 \times 2^{-100}}$$
$$= 0.0000\ 1011\ 100$$

Example 3. Divide $(0.10111 \times 2^{1011})$ by (0.11×2^{10}).

$$\left(\frac{0.10111 \bullet 2^{1011}}{0.11 \bullet 2^{10}}\right) = \left(\frac{0.10111}{0.11}\right) \bullet 2^{1011-10}$$

The characteristic is $1011 - 10 = 1001$.

The mantissa requires more work.

0.11 ⟌ **0 .1 1 1 0 0 0 0 0 0 0 0 0**

Shift the radix point in both terms to the right by the same number of places until the divisor is an integer. In this example, shift the radix point by 2 places. Perform the division.

```
                0  .1  1  1  1  0  1  0  1
          ┌──────────────────────────────────
       11 │ 1  0  .1  1  1  0  0  0  0  0
             1  1
          ──────
             1  0  1
                1  1
             ──────
                1  0  1
                   1  1
                ──────
                   1  0  0
                      1  1
                   ──────
                      1  0  0
                         1  1
                      ──────
                         1  0  0
```

The answer in binary is

$$\mathbf{0.1111\ \underline{01} \times 2^{1001}}$$
$$= 1\ 1110\ 1010.\underline{10}$$
$$= 256 + 128 + 64 + 32 + 8 + 2 + (2/3) = \mathbf{490.\underline{6}}$$

Representation of Floating Point Numbers

Representing numbers in a computer is a design problem. Computers do not have infinite length words. Words in a computer are finite in length. This means that not all numbers can be represented exactly. Rhetorical questions:

How should you represent zero?
How should you represent a negative number?
Should you have a representation of infinity? If so, how?
How should you represent a mantissa?
How should you represent a characteristic?
What other representations are desirable? Error conditions? Status indicators?

Several technical references for the following discussion are valuable for future computer scientists and computer engineers.[71] [72] [73] [74]

Negative Numbers

In the discussion that follows, mention will occasionally be made that some methods of representing numbers result in two representations for zero, +0 and –0. For testing equality, you usually want a test that +0 = -0 to be true, and it would seem that a unique value for zero would always be desirable. Alas, such is not always the case. Distinguishing between these is useful when dealing with something called "branch cuts", which is a topic in complex variables.[75] Scientists, engineers, and mathematicians worry about such things. Computer scientists that do not work with engineers and scientists, and people involved in business data processing, do not need to consider this. More will be said about this when discussing IEEE formats for floating point numbers.

How do you provide for negative numbers? There are three major ways of doing this.

- signed magnitude
- complement
- bias or Excess (XS)

For purpose of discussion, pretend that we are restricted to a 4-bit word. It is important to start numbering bits from the right hand side, starting with zero. The left-most bit is called the Most Significant Bit (MSB), and the right-most bit is called the Least Significant Bit (LSB). For example:

3	2	1	0
MSB			LSB

Signed Magnitude

The sign bit approach is to reserve one bit for use to indicate the sign of the number. When we compare two numbers, a positive number should appear to be larger than a negative number. It makes sense to assign the most significant bit to be the sign bit. If a sign bit equal to zero represents a negative sign, and a sign bit equal to one represents a positive sign, we get the right outcome when we do a comparison.

---

[71] P. H. Abbott, D. G. Brush, C. W. Clark III, C. J. Crone, J. R. Ehrman, G. W. Ewart, C. A. Goodrich, M. Hack, J. S. Kapernick, B. J. Minchau, W. C. Shepard, R. M. Smith, Sr., R. Tallman, S. Walkowiak, A. Watanabe, and W. R. White, "Architecture and software support in IBM S/390 Parallel Enterprise Servers for IEEE Floating-Point arithmetic" IBM Journal of Research and Development, Volume 43, Numbers 5/6 (1999), http://www.ibm.com, 08 June 2002

[72] ESA/390 Principles of Operation, International Business Machines, Document Number SA22-7201-06 (1999), http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/DZ9AR006/CCONTENTS , 10 Jun 2002.

[73] Intel® Itanium™ Architecture Software Developer's Manual, Volume 1, *Application Architecture*, Revision 2.0 (December 2001).

[74] David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *Computing Surveys*, Association for Computing Machinery (March 1991).

[75] W. Kahan, "Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit"; Chapter 7 in *The State of the Art in Numerical Analysis* ed. by M. Powell and A. Iserles (1987) Oxford.

±	2	1	0
MSB			LSB

A complication to using the sign bit approach is that there are two possible representations of the number zero: a plus-zero, and a minus-zero. The trichotomy test classifies a number as being

- less than zero
- equal to zero
- greater than zero

When there are two possible representations for zero, the test becomes more complicated. You must test for both a positive and a negative zero. Two representations for zero means that one combination of bits is being wasted that could be used to represent another number. For our example signed nibble, we can represent only 14 unique numbers.

Complement

Negative numbers can be represented in binary using either one's complement or two's complement. Complement arithmetic has the advantage of using positive integer addition for adding positive and negative numbers. Like the signed magnitude approach, one's complement representation has the disadvantage of having two representations for the number zero.

Two's complement (radix complement) arithmetic does not have the disadvantage of ambiguous representations of zero. If you declare that numbers having a one in the most significant bit is a negative number, you can use two's complement to unambiguously represent negative numbers. If you do not agree to that convention, you have ambiguity. Consider the bit pattern in an example nibble:

0	1	1	0

Is the intended number plus six, or minus ten?

Bias or Excess

Rather than interpreting the most significant bit as an indicator of sign, the bias approach is to declare that the B smallest numbers are negative numbers. Let X be the value we want to represent. The number Y is the stored value, which is computed as $Y = X + B$. In our 4-bit word, suppose we declare a bias of 4. We have 16 possible numbers, with the range [-4, 11].

0	0	1	0	is -2

while the pattern

0	1	1	0	is $+2$

and

1	1	1	1	is $+11$

This approach allows you to partition your set of numbers in an arbitrary fashion. Bias representation imposes additional procedural requirements on arithmetic operations.

Suppose you want to add -1 plus -1 using a bias of 4. Naïve addition gives us

-1+4=3	0	0	1	1
+	0	0	1	1
Sum = 6	0	1	1	0

The sum of 6 corresponds to +2, which is not what we want. We want to compute $X_1 + X_2 = (-1) + (-1) = -2$. To get the desired answer, add the two numbers and subtract the bias. The required computation is

$$Y_{answer} = Y_1 + Y_2 - B$$

Addition is commutative; you can exchange order of addition without changing the result. Subtracting the bias before doing the addition can preserve precision in the result at intermediate steps when using a finite length word for arithmetic, such as when using a real computer.

$$Y_{answer} = Y_1 - B + Y_2$$

Bias representation of negative numbers turns out to be very useful when using exponents, with a desire to limit the number of places reserved for fractions (negative exponents). The amount of bias is designated in the name. The notation above would be called XS-4 notation, where "**XS**" means "**Excess**".

It is common on binary computers to allocate half the range of numbers that can be represented to be negative numbers, though you could design a system differently. The bias is usually set to be some power of two, or just one less than a power of two, depending on the processor.

IEEE excess notation declares that when the most significant bit is on, the represented number is a positive integer. The bit pattern of the excess is all ones, except for the most significant bit which is zero. When this is done, the most significant bit can be interpreted as a sign bit, with a negative number having the sign bit turned off. For an 8-bit number, this is called **XS**-127. The range of possible numbers for this situation is [-127, 128]. A zero characteristic is represented by $0111\ 1111_2$.

IBM hexadecimal floating point excess notation declares that when the most significant bit is turned on, the represented number is a positive integer or zero. The bit pattern of the excess is all zeros, except for the most significant bit which is a one. For a 7-bit number (which IBM uses), this is called XS-64. The range of possible numbers for this situation is [-64, 63]. A zero characteristic is represented by $100\ 0000_2$.

	IEEE		**IBM**		
# Bits	**Decimal Excess**	**Hex Excess**	**Decimal Excess**	**Hex Excess**	**Equivalent Binary XS**
7			64	80	256
8	127	7F			
11	1023	7FF			
15	16383	7FFF			
17	65535	1FFFF			

Mantissa Normalization and the Characteristic

The details of representing the mantissa and characteristic depend on the normalization scheme chosen. We saw a precursor to the variation in rules when we considered the differences between scientific notation and engineering notation.

In a fixed length word computer, you are limited to how many different numbers you can represent. You can represent a greater range of numbers if you are willing to sacrifice how densely packed the representable numbers are. Suppose you have only 8 bits to represent all numbers, with allocation of bits as shown below. Because of using one bit for a sign, only 255 different numbers can be represented using these 8 bits, rather than 256, because +0 = -0.

7	6	5	4	3	2	1	0
±	Mantissa				Characteristic		

With binary normalization, the characteristic tells you how many places the radix point of the fixed point binary number was shifted to satisfy the normalization rule. For example, using two's complement in the characteristic, the number

7	6	5	4	3	2	1	0
0	**1**	**1**	**0**	**0**	**0**	**1**	**1**
±	Mantissa				Characteristic		

has a characteristic of $011_2 = 3$. The value in binary of $0.11 \times 2^3 = 110.0_2$. The number

7	6	5	4	3	2	1	0
0	**1**	**1**	**0**	**0**	**1**	**0**	**1**
±	Mantissa				Characteristic		

has a two's complement characteristic of 101_2. In signed-magnitude, this number is $-011_2 = -3$. The value of the floating point number is
$0.11 \times 2^{-3} = 0.0001\ 1_2 = (1/16) + (1/32) = 0.09375$.

Floating Point Implicit Binary Normalization

Effective use of every bit is zealously sought. Recall that the normalization scheme required shifting until a 1 arrived in the most significant position in the mantissa. If that bit is always 1, then there is no need to physically store it. Just pretend it is there, and use the most significant bit for another digit that you do not know *a priori*.

This is called implicit normalization, and it is the usual method of binary normalization. For a fixed number of bits, this extends the range of numbers that can be represented.

Block Normalization

Consider a different normalization rule, Block-2. For each increment of the characteristic, shift the mantissa 2 places instead of 1. Apply this new rule to the example bit patterns above. The first one,

7	6	5	4	3	2	1	0
0	**1**	**1**	**0**	**0**	**0**	**1**	**1**
±	Mantissa				Characteristic		

now has a characteristic of 2 x 3 = 6. The value in binary of $0.11 \times 2^{6} = 110000.0_2$. The number

7	6	5	4	3	2	1	0
0	**1**	**1**	**0**	**0**	**1**	**0**	**1**
±	Mantissa				Characteristic		

has a two's complement characteristic of 2 x (-3) = -6 . The value of the floating point number is $0.11 \times 2^{-6} = 0.0000\ 0011_2 = 0.0117\ 1875$.

Notice that the range of numbers you can represent has dramatically increased. You can represent numbers much closer to one, and much farther from one. This also reduces the frequency of having to do normalization. However, you can still only represent 255 different numbers. For our 8-bit floating point number, here are some of the properties for different decisions for normalization. The binary number closest to one that you can represent is given by

7	6	5	4	3	2	1	0
0	See table below.				**1**	**0**	**0**
±	Mantissa				Characteristic		

The positive binary number farthest from one that you can represent is given by

7	6	5	4	3	2	1	0
0	**1**	**1**	**1**	**1**	**0**	**1**	**1**
±	Mantissa				Characteristic		

Number of bits shifted for each increment of the characteristic	Binary mantissa of smallest number you can represent	Positive normalized number closest to zero you can represent	Positive number farthest from zero you can represent
Block-1 = Binary Normalization	0.1000	$2^{-1} \times 2^{-4} = 1/32 = 0.3125$	$(2^4 - 1) \times 2^{-1} = 7.5$
Block-2	0.0100	$2^{-2} \times 2^{-8} = 1/1024 \approx 9.7656 \times 10^{-4}$	$(2^4 - 1) \times 2^{2} = 48$
Block-3	0.0010	$2^{-3} \times 2^{-12} = 2^{-15} = 1/32768 \approx 3.05176 \times 10^{-5}$	$(2^4 - 1) \times 2^{5} = 480$
Block-4 = Hexadecimal Normalization	0.0001	$2^{-4} \times 2^{-16} = 2^{-20} = 1/1048576 \approx 9.53674 \times 10^{-7}$	$(2^4 - 1) \times 2^{8} = 3840$

As the number of bits shifted for each increment of the characteristic increased, the range of numbers also increased. In going from a shift of 1 bit to a shift of 4 bits for normalization, the range of values changed from A = [0.3125, 7.5] to approximately B = [9.54×10^{-7} , 3840], with only 255 different values on each interval. The holes between large numbers in set B are bigger than the holes between numbers in set A.

Floating Point Overflow and Underflow

Floating Point Overflow is the condition of attempting to compute a characteristic that is larger than the largest characteristic that a floating point word can store. This condition often is the result of attempting to divide by zero, or dividing a very large number by a very small number. Often, the floating point overflow is the result of a mistake in program logic or a bad data input.

Floating Point Underflow is the condition of attempting to compute a characteristic that is less than zero. This is equivalent to computing a number that is very close to zero. The importance of floating point underflow depends upon the application. For many applications, assuming the result to be zero is an adequate assumption. This can be catastrophic for applications that investigate the importance of small differences between resulting computations, as is common in physics and engineering.

Subnormal Floating Point Numbers

A subnormal (or denormalized) floating point number is a number with a zero characteristic, but a non-zero mantissa. A subnormal floating point number does not have an implicit bit. Ability to accommodate subnormal floating point numbers extends the range of numbers closer to zero that can be represented, after all the shifting that can be counted by the characteristic has been done. Permitting subnormal floating point numbers delays the onset of underflow. Not all floating point processors support subnormal floating point numbers.

Introduction to IEEE Standards and Implementation

For a fixed length representation (a fixed number of bits), there is a tradeoff between the number of bits used for the characteristic and the number of bits used for the mantissa. What is the best allocation of bits for the selected purpose?

Floating point arithmetic can be implemented in either software or hardware. Floating point hardware is useful if you must do a lot of floating point operations. Floating point hardware increases the cost of the processor, so this capability is a waste if you do not use it. For floating point hardware, the tradeoffs are implemented by the engineer and cannot be altered.

When floating point arithmetic is done in software, the programmer has the option of performing these tradeoffs to design a floating point format that is optimum for the application. Most programmers do not think about this. Instead, they usually mimic the structures implemented in hardware for other processors that include floating point hardware. IEEE Standards 754 and 854 specify floating point formats.

These standards were constructed so that a person who does floating point arithmetic using one compiler and processor combination will get about the same results as a person doing floating point arithmetic using another compiler and processor combination. While standards have helped, they still do not guarantee performance independent of compiler and platform. The standards only specify format for storage in memory. The standards do not specify the layout of special purpose registers used during computation, nor the algorithms for performing arithmetic. For critical applications, identical precision and accuracy cannot be taken for granted for applications ported to other hardware environments or compilers. Trust, but verify. Special situations can benefit from custom designed floating point software.

Sometimes programmers need to convert floating point numbers to fixed point numbers for use in real-time automatic control systems. When this is not carefully done, it can result in disaster. Lack of attention to detail in conversion of numbers from one representation to another led to the explosion of the European Space Agency's Ariane 5 rocket on 04 June 1996. The problem was traced to a conversion from a 64-bit floating point number to a 16-bit fixed point number.[76]

Zucker has a nice interactive web site for converting decimal to floating point.[77]

IEEE Standards 754 (1985) and 854 for Floating Point Arithmetic

The Institute of Electrical and Electronics Engineers (IEEE) standard for floating point arithmetic is the industry standard for design.[78] It defines the format for floating point numbers and the methods for doing computation.

---

[76] Douglas N. Arnold, "The Explosion of the Ariane 5" (23 AUG 2000). http://www.ima.umn.edu/~arnold/disasters/ariane.html visited 04 JUN 2002.

[77] Ron Zucker, "Floating Point Conversion Using Excess Notation", University of North Florida. http://www.unf.edu/~rzucker/cot3100dir/excess.html visited 07 January 2008.

[78] P. H. Abbott, D. G. Brush, C. W. Clark III, C. J. Crone, J. R. Ehrman, G. W. Ewart, C. A. Goodrich, M. Hack, J. S. Kapernick, B. J. Minchau, W. C. Shepard, R. M. Smith, Sr., R. Tallman, S. Walkowiak, A. Watanabe, and W. R. White, "Architecture and software support in IBM S/390 Parallel Enterprise Servers

IEEE Zero, Subnormals, Infinity, and Not A Number (NAN)

IEEE floating point zero is defined by setting all bits in both the characteristic and the mantissa to zero. The value is considered zero regardless of the state of the sign bit. IEEE identifies a number as subnormal when all bits of the characteristic are zero, but the mantissa is nonzero. IEEE identifies a number as infinity when all bits of the characteristic are one, but the mantissa is zero.

In addition to numerical values, the IEEE floating point specification defines two other objects, "not a number" (NaN) or "not a value". There are two types of NaNs: Signaling NAN (SnaN), and Quiet NAN (QnaN). NANs have all bits of the characteristic set to one, but the mantissa is non-zero. The Quiet NAN mantissa has a one in the most significant bit. The Signaling NAN mantissa has a zero in the most significant bit.

IEEE Single Precision Floating Point

0	Sign	1	Characteristic	8	9	Mantissa	31

IEEE single precision floating point has an 8-bit characteristic using Excess-127 ($2^7 - 1$) notation, and a 23-bit mantissa in signed magnitude form, with implicit-bit binary normalization (24-bit precision), where the implicit bit is in the 2^0 position (just to the left of the binary point). The sign bit is the sign of the mantissa, with 0 indicating positive and 1 indicating negative.

$$X = \pm 2^{(Characteristic - Excess)} \bullet (1 + Mantissa)$$

The maximum number that can be represented with this format is approximately $2 \times 2^{127} \approx 3.4 \times 10^{38}$. The smallest normalized number that can be represented is approximately $2 \times 2^{-127} \approx 1.2 \times 10^{-38}$. The smallest subnormal number that can be represented is approximately $2 \times 2^{-127} \times 2^{-23} = 2 \times 2^{-150} \approx 1.4 \times 10^{-45}$.

IEEE Double Precision Floating Point

0	Sign	1	Characteristic	11	12	Mantissa	63

IEEE double precision floating point has an 11-bit characteristic using Excess-1023 ($2^{10} - 1$) notation, and 52-bit mantissa in signed magnitude form, with implicit-bit binary normalization, where the implicit bit is in the 2^0 position. The sign bit is the sign of the mantissa, with 0 indicating positive and 1 indicating negative. Therefore, the mantissa has 53-bit precision.

$$X = \pm 2^{(Characteristic - Excess)} \bullet (1 + Mantissa)$$

The maximum number that can be represented is approximately $2 \times 2^{1023} \approx 1.8 \times 10^{308}$. The smallest normalized number that can be represented is

---

for IEEE Floating-Point arithmetic" IBM Journal of Research and Development, Volume 43, Numbers 5/6 (1999), http://www.ibm.com, 08 June 2002.

approximately $2 \times 2^{-1023} \approx 2.2 \times 10^{-308}$. The smallest subnormal number that can be represented is approximately

$2 \times 2^{-1023} \times 2^{-52} = 2 \times 2^{-1075} \approx 4.9 \times 10^{-324}$.

IEEE Double Extended Precision Floating Point

79	Sign	78	Characteristic	64	63	Mantissa	0

IEEE extended precision floating point has a 15-bit characteristic using Excess-16383 ($2^{14} - 1$) notation, and 64-bit mantissa, with implicit-bit binary normalization, where the implicit bit is in the 2^0 position. The sign bit is the sign of the mantissa, with 0 indicating positive and 1 indicating negative. Therefore, the mantissa has 65-bit precision.

$$X = \pm 2^{(Characteristic - Excess)} \bullet (1 + Mantissa)$$

The maximum number that can be represented is approximately $2 \times 2^{16383} \approx 1.2 \times 10^{4932}$. This is more than the number of molecules in the universe (assuming a finite universe and a big bang). The smallest normalized number that can be represented is approximately $2 \times 2^{-16383} \approx 3.4 \times 10^{-4932}$. The smallest subnormal number that can be represented is approximately

$2 \times 2^{-16383} \times 2^{-64} = 2 \times 2^{-16447} \approx 1.8 \times 10^{-4951}$.

Intel Floating Point Unit Formats

The Intel Floating Point Unit (FPU) uses the IEEE format for Single Real and Double Real floating point numbers.

Intel Extended Real Floating Point

79	Sign	78	Characteristic	64	63	Mantissa	0

The 80-bit floating point register format has a 15-bit characteristic and 64-bit mantissa, with explicit-bit binary normalization (64-bit precision) where the most significant bit is in the 2^0 position. The sign bit is the sign of the mantissa, with 0 indicating positive and 1 indicating negative. The 15-bit characteristic is biased by $2^{14} - 1 = 16383$.

$$X = \pm 2^{(Characteristic - Excess)} \bullet (Mantissa)$$

The largest number that can be represented with this format is approximately $2 \times 2^{16383} - 1 \approx 1.18 \times 10^{4932}$. The smallest normalized number that can be represented is approximately $2 \times 2^{-16383} \approx 3.4 \times 10^{-4932}$. The smallest subnormal number that can be represented is approximately $2 \times 2^{-16383} \times 2^{-64} = 2 \times 2^{-16447} \approx 1.8 \times 10^{-4951}$.

Itanium and Pentium 4 Extended Double Precision Floating Point Register

The Itanium IA-64 processor by Intel fully complies with ANSI/IEEE Standard 754 for floating point arithmetic.[79] In addition, the Intel processors have an Extended Double Precision floating point register format.

81	Sign	80	Characteristic	64	63	Mantissa	0

The 82-bit floating point register format has a 17-bit characteristic and 64-bit mantissa, with explicit-bit binary normalization (64-bit precision) where the most significant bit is in the 2^0 position. The sign bit is the sign of the mantissa, with 0 indicating positive and 1 indicating negative. The 17-bit characteristic is biased by $2^{16} - 1 = 65{,}535$.

$$X = \pm 2^{(Characteristic - Excess)} \bullet (Mantissa)$$

The largest number that can be represented with this format is approximately $2 \times 2^{65536} - 1 \approx 4 \times 10^{19728}$.

IBM Floating Point Arithmetic

The IBM System 390 supports the IEEE Floating Point Arithmetic standard, which IBM calls "Binary Floating Point (BFP)". In addition, S/390 continues to support the IBM proprietary hexadecimal normalization architecture. IBM calls this "Hexadecimal Floating Point (HFP)".[80]

Hexadecimal Floating Point means that the characteristic is a power of 16 rather than a power of 2. Each increment by 1 of the characteristic represents a shift of 4 bits of the mantissa when it is being normalized. Excess notation for IBM HFP also differs; the excess for IBM HFP is a power of 2, the excess for IEEE and IBM BFP is one less than a power of 2. Excess for IBM HFP is $64 = 2^6$ for all sizes of Hexadecimal Floating Point words. Excesses for IEEE and IBM short and long Binary Floating Point are $127 = 2^7 - 1$ and $1023 = 2^{10} - 1$. The IA-64 processor Extended Double Precision excess is $65535 = 2^{32} - 1$, and the IBM Binary Extended Floating Point excess is $16383 = 2^{14} - 1$.

Hexadecimal Short Floating Point

0	Sign	1	Characteristic	7	8	Mantissa	31

The 32-bit short floating point word has a 7-bit characteristic using XS-64 notation. The 24-bit mantissa uses explicit hexadecimal normalization to extend the range of numbers and reduce the frequency of shifting. The IBM hexadecimal format does not have an implicit bit to the left of the radix point. The sign bit is zero for a plus sign, and it is one for a negative sign.

$$X = 16^{(Characteristic - Excess)} \bullet (Mantissa)$$

---

[79] Intel® Itanium™ Architecture Software Developer's Manual, Volume 1, *Application Architecture*, Revision 2.0 (December 2001).

[80] ESA/390 Principles of Operation, International Business Machines, Document Number SA22-7201-06 (1999). http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/DZ9AR006/CCONTENTS , Visited 10 Jun 2002.

Hexadecimal Long Floating Point

0	Sign	1	Characteristic	7	8	Mantissa	63

The 64-bit short floating point word has a 7-bit characteristic using XS-64 notation. The 56-bit mantissa uses explicit hexadecimal normalization to extend the range of numbers and reduce the frequency of shifting. The IBM hexadecimal format does not have an implicit bit to the left of the radix point. The sign bit is zero for a plus sign, and it is one for a negative sign.

Hexadecimal Extended Floating Point

Extended floating point is represented with two 64-bit words. The characteristic uses Excess-64 (2^6) notation.

0	Sign	1	High Order Characteristic	7	8	Left 14 Hex Digits of Mantissa	63

64	Sign	65	Low Order Characteristic	71	72	Right 14 Hex Digits of Mantissa	127

The mantissa has a total of 112 bits. The high order characteristic belongs with the high order word. The low order characteristic belongs to the low order word. The sign bit of both words are the same, which is the sign of the mantissa.

Let X be the value in the high order word. Let Y be the value in the low order word. Let the value of the extended floating point variable be

$$Z = X + Y$$

Let the sign of the high order word be Xsign. Let the characteristic be Xcharacteristic. Let the mantissa be Xmantissa. Then

$$X = 16^{(Xcharacteristic - Excess)} \bullet (Xmantissa)$$

Let the sign of the low order word be Ysign. Note that Ysign = Xsign. Let the characteristic be Ycharacteristic. Let the mantissa be Ymantissa.

$$Y = 16^{(Ycharacteristic - Excess)} \bullet (Ymantissa)$$

Note that the low order mantissa is shifted by 14 hexadecimal digits in position from the high order mantissa. That means the low order characteristic is always 14 less than the high order characteristic.

$$Ycharacteristic = Xcharacteristic - 14$$

Therefore,

$$Y = 16^{(Xcharacteristic - 14 - Excess)} \bullet (Ymantissa)$$

Evaluating,

$$Z = 16^{(Xcharacteristic - Excess)} \left(Xmantissa + 16^{-14} \bullet (Ymantissa)\right)$$

Because the minimum value that can be stored in the low order characteristic field is zero, the minimum characteristic that can be used in the high order characteristic is 14. The range of values that can be used in the high order characteristic is [-50, 63]. Remember that these are powers of 16. IBM Hexadecimal Extended Floating Point therefore cannot represent numbers as close to zero as IBM Hexadecimal Long Floating Point. To help keep this in perspective, the moment of a Proton is approximately 1.4×10^{-26} J/T (Joules / Tesla).

Binary Short Floating Point

The IBM Binary Short Floating Point word conforms to the IEEE Single Precision Floating Point word specifications.

Binary Long Floating Point

The IBM Binary Long Floating Point word conforms to the IEEE Double Precision Floating Point word specifications.

Binary Extended Floating Point

0	Sign	1	Characteristic	15	16	Mantissa	127

The 128-bit binary extended floating point has a 15-bit characteristic using XS-16383 $(2^{14} - 1)$ notation. The 112-bit mantissa (113-bit precision) uses implicit normalization. Note that IEEE does not have a specification for a floating point word format greater than Double Precision. The IBM Binary Extended Floating Point word gives greater precision (more bits in the mantissa) than the Intel IA-64 (Pentium 4 and Itanium) Extended Double Precision word, but with reduced range (2 fewer bits in the characteristic). The IBM Hexadecimal Floating Point characteristic, for all its formats, produce a greater range of numbers that can be represented $((1/16) \times 16^{-64}, 16^{63})$, or approximately $(5.4 \times 10^{-79}, 7.2 \times 10^{75})$. Chris Impey reported that one estimate of the number of photons in the universe is 10^{80}.[81]

The maximum number that can be represented is approximately $2 \times 2^{16383} \approx 1.2 \times 10^{4932}$. The smallest normalized number that can be represented is approximately $2 \times 2^{-16383} \approx 3.4 \times 10^{-4932}$. The smallest subnormal number that can be represented is approximately $2 \times 2^{-16383} \times 2^{-112} = 2 \times 2^{-16495} \approx 6.5 \times 10^{-4966}$.

[81] Chris Impey, "Reacting to the Size and Shape of the Universe", Mercury Magazine Vol. 30 No. 1 (January/February 2001). http://www.aspsky.org/mercury/mercury/30_01/universe.html Visited 05 July 2002.

Floating Point Conversions

Convert IEEE Single Precision Floating Point to IBM Hexadecimal Short Floating Point

Procedure:

Step 1: Let M = (IEEE Single Precision Floating Point characteristic) – (IEEE Single Precision Floating Point Excess).

Note: The IEEE format uses implicit normalization. The radix point is immediately to the left of the IEEE mantissa most significant bit.

Step 2: Insert the prefix bit pattern 0001 immediately to the left of the radix point.

Step 3: Let MM = M MOD 4. This is the remainder obtained by dividing M by 4.

Step 4: Compute the number of bits (J) to right-shift the mantissa and prefix bit pattern.

- If MM = 0, then J = 4
- If MM < 0, then J = |MM|. |MM| is the absolute value of MM.
- If MM > 0, then J = 4 – MM.

The number of bits of precision in the mantissa lost is $J - 1$.

Step 5: Copy the shifted bit pattern into the IBM Short HFP mantissa.

Step 6: Compute the IBM HFP characteristic, without excess.

- $K = \left\lfloor \frac{M+4}{4} \right\rfloor$. The brackets with feet at the bottom, and no roof, is the FLOOR function. The FLOOR of an exact integer is that integer. The FLOOR of a number with a fraction is the next LOWER integer. Examples: $\lfloor 3.4 \rfloor = 3$, $\lfloor -3.4 \rfloor = -4$.

Step 7: Add the IBM HFP excess to K and store it in the HFP characteristic field.

Step 8: Copy the sign bit from the IEEE Single Precision FP word to the sign bit of the IBM HFP word.

Example: Convert the following Convert IEEE Single Precision Floating Point to IBM Hexadecimal Short Floating Point

IEEE Single Precision Floating Point

S	Characteristic								Mantissa						
0	1			4				8	9			12			15
0	1	0	0	1	0	0	0	1	0	0	1	1	0	0	1

Mantissa (Continued)															
16				20				24				28			31
0	0	1	1	1	1	0	1	0	0	0	1	0	0	0	1

Step 1: Let M = (IEEE Single Precision Floating Point characteristic) – (IEEE Single Precision Floating Point Excess). M = 145 – 127 = 18.

Note: The IEEE format uses implicit normalization. The radix point is immediately to the left of the IEEE mantissa most significant bit.

Step 2: Insert the prefix bit pattern 0001 immediately to the left of the radix point. The appended mantissa is 0001.001 1001 0011 1101 0001 0001

Step 3: Let MM = M MOD 4. This is the remainder obtained by dividing M by 4. MM = 18 MOD 4 = 2

Step 4: Compute the number of bits (J) to right-shift the mantissa and prefix bit pattern.

- If MM = 0, then J = 4
- If MM < 0, then J = |MM|. |MM| is the absolute value of MM.
- If MM > 0, then J = 4 – MM = 4 – 2 = 2.

The number of bits of precision in the mantissa lost is J – 1 = 2 – 1 = 1.

Step 5: Copy the shifted bit pattern into the IBM Short HFP mantissa. The pattern copied is: 01001 1001 0011 1101 0001 0001

Step 6: Compute the IBM HFP characteristic, without excess.

- $K = \left\lfloor \frac{M+4}{4} \right\rfloor = \left\lfloor \frac{18+4}{4} \right\rfloor = \left\lfloor \frac{22}{4} \right\rfloor = \lfloor 5.5 \rfloor = 5$.

Step 7: Add the IBM HFP excess to K and store it in the HFP characteristic field. K + 64 = 5 + 64 = 69 = 45_{16} = 100 0101_2 .

Step 8: Copy the sign bit from the IEEE Single Precision FP word to the sign bit of the IBM HFP word.

IBM Hexadecimal Short Floating Point

S	Characteristic							Mantissa							
0	1			4			7	8				12			15
0	1	0	0	0	1	0	1	0	1	0	0	1	1	0	0

Mantissa (Continued)															
16				20				24				28			31
1	0	0	1	1	1	1	0	1	0	0	0	1	0	0	0

Convert IBM Hexadecimal Extended Floating Point to IEEE Double Extended Precision Floating Point

Procedure:

Step 1: Subtract the IBM Hexadecimal Floating Point Excess from the high order characteristic.
Step 2: Let K = the numerical value of the high order characteristic.
Step 3: Let N = 4 K. Each positive increment of the IBM Hexadecimal Floating Point characteristic represents a shift of the radix by 4 bit positions to the left to perform the normalization.
Step 4: Let J = the number of bit positions from the left end of the mantissa occupied by the first one-bit.
Step 5: Let M = N – J .
Step 6: Convert M to binary.
Step 7: Add the IEEE Double Extended Precision Floating Point Excess to M.
Step 8: Record the result in the characteristic field of the IEEE Double Extended Precision Floating Point word.
Step 9: The most significant bit of the mantissa of the IBM Hexadecimal Floating Point becomes the implicit bit in the IEEE Double Extended Precision Floating Point word, and therefore does not explicitly get recorded.
Step 10: Beginning with the bit immediately to the right of the most significant bit, copy the remaining bits from the IBM Hexadecimal Floating Point mantissa into the IEEE Double Extended Precision Floating Point mantissa.
Step 11: Stop after the IEEE Double Extended Precision Floating Point mantissa is filled. Note that 37 to 40 least-significant-bits of precision in the mantissa were lost in the conversion.

Convert IEEE Double Extended Precision Floating Point to IBM Hexadecimal Extended Floating Point

Procedure:

Step 1: Set all IBM Hexadecimal Extended Floating Point bits to zero.
Step 2: Copy the IEEE Double Extended Precision Floating Point sign bit into the IBM Hexadecimal Extended Floating Point High Order sign bit and Low Order sign bit.
Step 3: Evaluate the IEEE Double Extended Precision Floating Point Excess from the characteristic. Call this G.
Step 4: Subtract the IEEE Double Extended Precision Floating Point Excess from the characteristic. Call this $P = G - XS_{\text{IEEE DE FP}}$.
Step 5: Compute the IBM Hexadecimal Floating Point Characteristic without the IBM HFP Excess. $K = \left\lfloor \frac{P+4}{4} \right\rfloor$
Step 6: Compute the IBM Hexadecimal Floating Point Characteristic with the IBM HFP Excess. N = K + 64.
Step 7: Write N into the IBM HFP High Order Characteristic.
Step 8: Append the implicit most significant bit to the left of the radix point for the IEEE DEP FP mantissa.

Step 9: Let MM = P MOD 4. This is the remainder obtained by dividing P by 4.
Step 10: Compute the number of places to shift the appended mantissa to obtain the IBM HFP mantissa.
- If MM = 0, then J = 4
- If MM > 0, then J = 4 – MM
- If MM < 0, then J = | MM | where the vertical bars identify the absolute value function.

Step 11: Beginning with the most significant bit of the shifted mantissa, copy the remaining bits from the IEEE Double Extended Precision Floating Point mantissa into the IBM Hexadecimal Floating Point Mantissa.
Step 12: Compute the IBM HFP Low Order Characteristic. Call this Q.
Step 13: Write Q into the IBM HFP Low Order Characteristic.

Floating Point Conversion Problems

Problem 1. Convert the following IEEE Single Precision Floating Point word to an IBM Hexadecimal Short Floating Point word.

IEEE Single Precision Floating Point

S	Characteristic								Mantissa						
0	1			4				8	9			12			15
0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0

Mantissa (Continued)															
16				20				24				28			31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

IBM Hexadecimal Short Floating Point

S	Characteristic							Mantissa							
0	1			4			7	8				12			15

Mantissa (Continued)															
16				20				24				28			31

Problem 2. Convert the following IEEE Single Precision Floating Point word to a IBM Hexadecimal Short Floating Point word.

IEEE Single Precision Floating Point

S	Characteristic								Mantissa						
0	1			4				8	9			12			15
0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0

Mantissa (Continued)															
16				20				24				28			31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IBM Hexadecimal Short Floating Point

S	Characteristic							Mantissa							
0	1			4			7	8				12			15

Mantissa (Continued)															
16				20				24				28			31

Problem 3. Convert the following IEEE Single Precision Floating Point word to a IBM Hexadecimal Short Floating Point word.

IEEE Single Precision Floating Point

S	Characteristic								Mantissa						
0	1			4				8	9			12			15
1	0	1	1	1	1	1	0	1	0	0	1	0	0	0	0

Mantissa (Continued)															
16				20				24				28			31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IBM Hexadecimal Short Floating Point

S	Characteristic							Mantissa							
0	1			4			7	8				12			15

Mantissa (Continued)															
16				20				24				28			31

Problem 4. Convert the following IBM Hexadecimal Extended Floating Point word to an IEEE Double Extended Precision Floating Point word.

IBM Hexadecimal Extended Floating Point

High Order Word

S	High Order Characteristic							High Order Mantissa							
0	1			4			7	8			12				15
0	1	0	0	1	0	1	0	0	0	0	1	0	1	1	0

High Order Mantissa (Continued 1)															
16				20				24				28			31
0	1	0	1	1	1	0	0	1	1	1	1	1	0	0	1

High Order Mantissa (Continued 2)															
32				36				40				44			47
0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	1

High Order Mantissa (Continued 3)															
48				52				56				60			63
1	1	1	0	1	1	1	0	0	1	0	1	0	0	1	0

Low Order Word

S	Low Order Characteristic							Low Order Mantissa							
64	65			68			71	72			76				79
0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0

Low Order Mantissa (Continued 1)															
80				84				88				92			95
0	0	0	0	1	0	1	1	0	1	1	1	0	0	1	0

Low Order Mantissa (Continued 2)															
96				100				104				108			111
1	1	0	0	0	1	0	1	1	0	0	0	1	0	1	0

Low Order Mantissa (Continued 3)															
112				116				120				124			127
1	1	1	0	0	1	0	0	0	1	0	0	0	1	0	1

IEEE Double Extended Precision Floating Point

S	Characteristic														
79	78			75			72	71				67			64

Mantissa															
63			60				56				52				48

Mantissa (Continuation 1)														
47			44				40				36				32

Mantissa (Continuation 2)														
31			28				24				20				16

Mantissa (Continuation 3)														
15			12				8				4				0

Problem 5. Convert the following IEEE Double Extended Precision Floating Point word to an IBM Hexadecimal Extended Floating Point word.

IEEE Double Extended Precision Floating Point

S	Characteristic														
79	78			75			72	71				67			64
1	0	1	1	1	1	1	1	0	1	0	0	1	0	1	0

Mantissa															
63			60				56				52				48
0	1	1	1	0	0	1	0	1	1	1	1	1	0	1	1

Mantissa (Continuation 1)														
47			44				40				36				32
1	1	0	0	0	1	1	1	0	0	1	1	1	1	0	0

Mantissa (Continuation 2)														
31			28				24				20				16
1	1	1	1	1	0	0	0	0	1	1	0	1	1	0	0

Mantissa (Continuation 3)														
15			12				8				4				0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1

IBM Hexadecimal Extended Floating Point

High Order Word

S	High Order Characteristic							High Order Mantissa							
0	1			4			7	8			12				15

High Order Mantissa (Continued 1)															
16				20				24				28			31

High Order Mantissa (Continued 2)															
32				36				40				44			47

High Order Mantissa (Continued 3)															
48				52				56				60			63

Low Order Word

S	Low Order Characteristic							Low Order Mantissa							
64	65			68			71	72			76				79

Low Order Mantissa (Continued 1)															
80				84				88				92			95

Low Order Mantissa (Continued 2)															
96				100				104				108			111

Low Order Mantissa (Continued 3)															
112				116				120				124			127

Problem 6. Convert the following IEEE Double Extended Precision Floating Point word to an IBM Hexadecimal Extended Floating Point word.

IEEE Double Extended Precision Floating Point

S	Characteristic														
79	78			75			72	71				67			64
1	0	1	1	1	1	1	1	0	0	0	0	1	0	1	1

Mantissa															
63			60				56				52				48
0	1	1	1	0	0	1	0	1	1	1	1	1	0	1	1

Mantissa (Continuation 1)															
47			44				40				36				32
1	1	0	0	0	1	1	1	0	0	1	1	1	1	0	0

Mantissa (Continuation 2)															
31			28				24				20				16
1	1	1	1	1	0	0	0	0	1	1	0	1	1	0	0

Mantissa (Continuation 3)															
15			12				8				4				0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1

IBM Hexadecimal Extended Floating Point

High Order Word

S	High Order Characteristic							High Order Mantissa							
0	1			4			7	8				12			15

High Order Mantissa (Continued 1)															
16				20				24				28			31

High Order Mantissa (Continued 2)															
32				36				40				44			47

High Order Mantissa (Continued 3)															
48				52				56				60			63

Low Order Word

S	Low Order Characteristic							Low Order Mantissa							
64	65			68			71	72				76			79

Low Order Mantissa (Continued 1)															
80				84				88				92			95

Low Order Mantissa (Continued 2)															
96				100				104				108			111

Low Order Mantissa (Continued 3)															
112				116				120				124			127

Problem 7. Convert the following IEEE Double Extended Precision Floating Point word to an IBM Hexadecimal Extended Floating Point word.

IEEE Double **Extended** Precision Floating Point

S	Characteristic														
79	78			75			72	71				67			64
1	0	1	1	1	0	1	1	1	0	0	1	0	1	1	0

Mantissa															
63			60				56				52				48
0	1	1	1	0	0	1	0	1	1	1	1	1	0	1	1

Mantissa (Continuation 1)															
47			44				40				36				32
1	1	0	0	0	1	1	1	0	0	1	1	1	1	0	0

Mantissa (Continuation 2)															
31			28				24				20				16
1	1	1	1	1	0	0	0	0	1	1	0	1	1	0	0

Mantissa (Continuation 3)															
15			12				8				4				0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1

IBM Hexadecimal Extended Floating Point

High Order Word

S	High Order Characteristic							High Order Mantissa							
0	1			4			7	8			12				15

High Order Mantissa (Continued 1)															
16				20				24				28			31

High Order Mantissa (Continued 2)															
32				36				40				44			47

High Order Mantissa (Continued 3)															
48				52				56				60			63

Low Order Word

S	Low Order Characteristic							Low Order Mantissa							
64	65			68			71	72			76				79

Low Order Mantissa (Continued 1)															
80				84				88				92			95

Low Order Mantissa (Continued 2)															
96				100				104				108			111

Low Order Mantissa (Continued 3)															
112				116				120				124			127

Numerical Methods

Numerical methods are procedures for solving mathematical problems numerically, including problems for which no closed form solutions exist. Attention is given to designing procedures that make efficient use of a computer while controlling computational errors. Analysis includes quantifying errors, approximation or convergence to solutions, and sensitivity of solutions to variations of inputs.

Numerical methods have been applied to algebra, calculus, differential equations, and matrix algebra. Computational fluid dynamics, weather forecasting, underwater sound propagation, and analysis of vibration of complex structures (cars, aircraft, space station) all require use of numerical methods.

The accumulation of approximation errors can render a solution useless. The study of such errors and methods to minimize such errors are introductory topics in numerical analysis. Some of the approaches used to minimize these errors include

- Thoughtful selection of the order of performing arithmetic operations. For example, doing arithmetic between pairs of small quantities before combining those results with larger quantities. Another strategy is to seek to alternate multiplications and divisions to keep results as close to 1 as possible.
- Rearranging the computation of a quantity to reduce the number of operations that produce large errors. For example, evaluating $f(x) = a_2 x^2 + a_1 x + a_0$ as $f(x) = (a_2 x + a_1) x + a_0$ which reduces the number of multiplications from 4 to 2.
- Selection of numerical methods that require fewer operations that produce large errors.

Numerical Methods include Numerical Analysis and Approximation Theory.

- Selection of mathematical approaches that converge to an adequate solution (not necessarily ever capable of reaching an ideal exact solution) with fewer operations.
- Application of asymptotic approximations.[82]

You must first understand the problem you want to solve! A perfectly computed solution to the wrong problem is still a wrong solution.

- Identify and solve the right problem.
- Express the problem in simplest forms.

Approximation theory[83] is the mathematical study of how given quantities can be approximated by other (usually simpler) ones under appropriate conditions. Approximation theory also studies the size and properties of the error introduced by approximation. Approximations are often obtained by power series expansions in which the higher order terms are dropped.

---

[82] James P. Keener, *Principles of applied mathematics: Transformation and approximation*, Pages 424 – 428. Addison-Wesley (1988).

[83] Weisstein, Eric W. "Approximation Theory.", MathWorld--A Wolfram Web Resource (10 February 2008). http://mathworld.wolfram.com/ApproximationTheory.html visited 12 February 2008.

More Computer Arithmetic

Basic arithmetic is the foundation for much sophisticated work in science and engineering. Doing a good job with arithmetic is prerequisite to doing a good job with other computations. What constitutes "good" depends upon the application. Digital Engineering Institute has a nice web site.[84] Some of the considerations that lead to different designs include: required accuracy, required precision, attention to circuit propagation delay, importance of speed, number of processors, control of fan-out, chip area required for circuit design, physical mass and volume of resulting circuit, and importance of power consumption.

[84] Richard B. Katz, "Arithmetic", NASA Office of Logic Design (15 January 2002). http://www.klabs.org/DEI/Arithmetic/ Visited 14 January 2008.

Sound, Smell and Color

Other kinds of data exist besides generic numbers. Some standards are emerging through common use.

Sound

Sounds may be represented either as processed signals or by abstract representations. Digitized waveforms often require large files that use much storage space and take much time to transmit over a network. Compression techniques have been developed to extract parameters that describe and permit an approximate reconstruction of the original sound. Encryption methods are also used to protect royalties from sale of copyrighted materials. These methods are used by the entertainment industry.

Another method for representing sounds is to describe the method used to generate sounds. An example of such an abstract method is a musical score that a performer uses when playing music. This specifies the sequence of musical notes, along with directions of how they are to be played. This is a compact notation that can be edited easily by a composer, and used by automated musical instruments to produce music. This approach is used in synthesized music.

The MIDI Manufacturers Association has a tutorial, links to specifications, and other information related to Musical Instrument Digital Interface (MIDI).[85] Codes of the MIDI specification are not brief enough to include in this tutorial. Another approach is to use MusiXTeX.[86]

Smell

How can you implement a computer version of "scratch and sniff"? The detection, analysis, and information display has been automated. One example is Cyranose. It measures the resistance of 32 polymer composite sensors when exposed to a vapor. The data is digitized and analyzed. Parameters that describe the vapor (smell) constitute an encoding of the data.[87] For an entrepreneurial and light-hearted exploration, see Charles Platt's article on Digiscent.[88] Even Amazon is getting into the act.[89]

Color

Consider codes for color used on web pages on the internet. This assumes that a particular code generates the same color on different monitors. While that assumption is not strictly true, we expect it to be close.

---

[85] Jim Heckroth (Crystal Semiconductor Corp), "Tutorial on MIDI and Music Synthesis", MIDI Manufacturers Association (2001). http://www.midi.org/about-midi/tutorial/tutor.shtml Visited 14 January 2008.

[86] Daniel Taupin, MusiXTeX, "Directory: CTAN home / tex-archive/ macros/ musixtex/ taupin " CTAN: The Comprehensive TeX Archive Network. http://www.ctan.org/tex-archive/macros/musixtex/taupin/ Visited 14 January 2008.

[87] http://cyranosciences.com/technology/onboard.html Visited 2001. Not available on 14 January 2008.

[88] Charles Platt, "You've Got Smell!", Wired News (November 1999). http://www.wired.com/wired/archive/7.11/digiscent_pr.html Visited 14 January 2008.

[89] "Amazon To Market Online Smells" (1998). http://www.rayno.com/amasmell.htm Visited 14 January 2008.

RGB color monitors represent colors by using combinations of three primary colors: red, green, and blue. It is common today for the intensity of each primary color to be represented by one byte, which allows 256 different levels of each primary color. The total number of colors represented by this scheme using three bytes is 256 x 256 x 256 = 16,777,216. Colors are coded as three pairs of hexadecimal digits such as OxF040FF, or as a triplet of decimal numbers such as (240, 64, 255). Internally, these codes are the same. The format used for input depends upon the software.

Hexadecimal Color Codes

The color table below shows 85 of the 16,777,216 colors that can be represented using 3 bytes per color.

OxFF0000	Ox00FF00	Ox0000FF
OxF00000	Ox00F000	Ox0000F0
OxC00000	Ox00C000	Ox0000C0
Ox800000	Ox008000	Ox000080
Ox400000	Ox004000	Ox000040
Ox0F0000	Ox000F00	Ox000008
OxFF00FF	OxFFFF00	Ox00FFFF
OxF000F0	OxF0F000	Ox00F0F0
OxC000C0	OxC0C000	Ox00C0C0
Ox800080	Ox808000	Ox008080
Ox400040	Ox404000	Ox004040
OxFFFFFF	OxFFFFFF	OxFFFFFF
OxFFF0FF	OxFFFFF0	OxF0FFFF
OxFFC0FF	OxFFFFC0	OxC0FFFF
OxFF80FF	OxFFFF80	Ox80FFFF
OxFF40FF	OxFFFF40	Ox40FFFF
OxFF00F0	OxFFF000	OxFFF0F0
OxFF00C0	OxFFC000	OxFFC0C0
OxFF0080	OxFF8000	OxFF8080
OxFF0040	OxFF4000	OxFF4040
OxF0FFF0	OxF0FF00	Ox00FFF0
OxC0FFC0	OxC0FF00	Ox00FFC0
Ox80FF80	Ox80FF00	Ox00FF80
Ox40FF40	Ox40FF00	Ox00FF40
OxF000FF	Ox00F0FF	OxF0F0FF
OxC000FF	Ox00C0FF	OxC0C0FF
Ox8000FF	Ox0080FF	Ox8080FF
Ox4000FF	Ox0040FF	Ox4040FF

Obviously, if you are looking at a black and white copy of this, you are not going to see the colors. You can see the colors by viewing an electronic copy. There are several very nice web sites with color code charts for use with HTML.

Analog Computers

The term “analog” refers to devices that are not binary digital computers. Analog computers rely on electromechanical, electronic, and other components to represent, retain, and transform continuous values. These values may be used for subsequent computation, displayed, or used as control signals to other devices. Analog computers tend to be special purpose devices, sometimes quite sophisticated.

Digital electronic computers are all analog devices at the most basic level. Computers that are part digital and part analog are called hybrid computers.

While digital computers have made a tremendous contribution, they are still limited by the speed at which inputs can be digitized and functions computed. This is a significant limitation in some applications of data acquisition and signal processing.

Critical issues for analog devices include the accuracy of converting data, repeatability, and reliability. Such devices are sensitive to its environment: heat, electromagnetic fields, vibration, humidity, gravity, etc.

We continue to push for extreme performance in computation and storage speed, small size, low power consumption, and other properties. Scientists and engineers are pushing the boundaries of physics, materials science and biological sciences to find new and reliable ways to compute.