Number Systems

Contents

Articles

References

Article Licenses

Two's complement

Two's complement is a mathematical operation on binary numbers, as well as a binary signed number representation based on this operation.

The two's complement of an N-bit number is defined as the complement with respect to 2^N, in other words the result of subtracting the number from 2^N. This is also equivalent to taking the ones' complement and then adding one, since the sum of a number and its ones' complement is all 1 bits. The two's complement of a number behaves like the negative of the original number in most arithmetic, and positive and negative numbers can coexist in a natural way.

In two's-complement representation, negative numbers are represented by the two's complement of their absolute value;[1] in general, negation (reversing the sign) is performed by taking the two's complement. This system is the most common method of representing signed integers on computers.[2] An N-bit two's-complement numeral system can represent every integer in the range $-(2^{N-1})$ to $+(2^{N-1}-1)$ while ones' complement can only represent integers in the range $-(2^{N-1}-1)$ to $+(2^{N-1}-1)$.

The two's-complement system has the advantage that the fundamental arithmetic operations of addition, subtraction, and multiplication are identical to those for unsigned binary numbers (as long as the inputs are represented in the same number of bits and any overflow beyond those bits is discarded from the result). This property makes the system both simpler to implement and capable of easily handling higher precision arithmetic. Also, zero has only a single representation, obviating the subtleties associated with negative zero, which exists in ones'-complement systems.

The method of complements can also be applied in base-10 arithmetic, using ten's complements by analogy with two's complements.

Bits	Unsigned value	2's complement value
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127
1000 0000	128	−128
1000 0001	129	−127
1000 0010	130	−126
1111 1110	254	−2
1111 1111	255	−1

Potential ambiguities of terminology

One should be cautious when using the term *two's complement*, as it can mean either a number format or a mathematical operator. For example, 0111 represents decimal 7 in two's-complement *notation*, but the two's complement of 7 in a 4-bit register is actually the "1001" bit string (the same as represents $9 = 2^4 - 7$ in unsigned arithmetics) which is the two's complement *representation* of −7. The statement "convert x to two's complement" may be ambiguous, since it could describe either the process of representing x in two's-complement notation without changing its value, or the calculation of the two's complement, which is the arithmetic negative of x if two's complement representation is used.

Converting from two's complement representation

A two's-complement number system encodes positive and negative numbers in a binary number representation. The weight of each bit is a power of two, except for the most significant bit, whose weight is the negative of the corresponding power of two.

The value `w` of an `N`-bit integer $a_{N-1}a_{N-2}\ldots a_0$ is given by the following formula:

$$w = -a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i$$

The most significant bit determines the sign of the number and is sometimes called the sign bit. Unlike in sign-and-magnitude representation, the sign bit also has the weight $-(2^{N-1})$ shown above. Using `N` bits, all integers from $-(2^{N-1})$ to $2^{N-1} - 1$ can be represented.

Converting to two's complement representation

In two's complement notation, a *non-negative* number is represented by its ordinary binary representation; in this case, the most significant bit is 0. Though, the range of numbers represented is not the same as with unsigned binary numbers. For example, an 8-bit unsigned number can represent the values 0 to 255 (11111111). However a two's complement 8-bit number can only represent positive integers from 0 to 127 (01111111), because the rest of the bit combinations with the most significant bit as '1' represent the negative integers −1 to −128.

The two's complement operation is the additive inverse operation, so negative numbers are represented by the two's complement of the absolute value.

From the ones' complement

To get the two's complement of a binary number, the bits are inverted, or "flipped", by using the bitwise NOT operation; the value of 1 is then added to the resulting value, ignoring the overflow which occurs when taking the two's complement of 0.

For example, using 1 byte (= 2 nibbles = 8 bits), the decimal number 5 is represented by

$$0000\ 0101_2$$

The most significant bit is 0, so the pattern represents a non-negative value. To convert to −5 in two's-complement notation, the bits are inverted; 0 becomes 1, and 1 becomes 0:

1111 1010

At this point, the numeral is the ones' complement of the decimal value 5. To obtain the two's complement, 1 is added to the result, giving:

1111 1011

The result is a signed binary number representing the decimal value −5 in two's-complement form. The most significant bit is 1, so the value represented is negative.

The two's complement of a negative number is the corresponding positive value. For example, inverting the bits of −5 (above) gives:

0000 0100

And adding one gives the final value:

0000 0101

The two's complement of zero is zero: inverting gives all ones, and adding one changes the ones back to zeros (since the overflow is ignored). Furthermore, the two's complement of the most negative number representable (e.g. a one as the most-significant bit and all other bits zero) is itself. Hence, there appears to be an 'extra' negative number.

Subtraction from 2^N

The sum of a number and its ones' complement is an N-bit word with all 1 bits, which is $2^N - 1$. Then adding a number to its two's complement results in the N lowest bits set to 0 and the carry bit 1, where the latter has the weight 2^N. Hence, in the unsigned binary arithmetic the value of two's-complement negative number x^* of a positive x satisfies the equality $x^* = 2^N - x$.[3]

For example, to find the 4-bit representation of −5 (subscripts denote the base of the representation):

$x = 5_{10}$ therefore $x = 0101_2$

Hence, with $N = 4$:

$$x^* = 2^N - x = 2^4 - 5_{10} = 10000_2 - 0101_2 = 1011_2$$

The calculation can be done entirely in base 10, converting to base 2 at the end:

$$x^* = 2^N - x = 2^4 - 5_{10} = 11_{10} = 1011_2$$

Working from LSB towards MSB

A shortcut to manually convert a binary number into its two's complement is to start at the least significant bit (LSB), and copy all the zeros (working from LSB toward the most significant bit) until the first 1 is reached; then copy that 1, and flip all the remaining bits. This shortcut allows a person to convert a number to its two's complement without first forming its ones' complement. For example: the two's complement of "0011 1100" is "1100 0100", where the underlined digits were unchanged by the copying operation (while the rest of the digits were flipped).

In computer circuitry, this method is no faster than the "complement and add one" method; both methods require working sequentially from right to left, propagating logic changes. The method of complementing and adding one can be sped up by a standard carry look-ahead adder circuit; the LSB towards MSB method can be sped up by a similar logic transformation.

Sign extension

Decimal	7-bit notation	8-bit notation
−42	1010110	1101 0110
42	0101010	0010 1010

sign-bit repetition in 7 and 8-bit integers using two's-complement

When turning a two's-complement number with a certain number of bits into one with more bits (e.g., when copying from a 1 byte variable to a two byte variable), the most-significant bit must be repeated in all the extra bits and lower bits.

Some processors have instructions to do this in a single instruction. On other processors a conditional must be used followed with code to set the relevant bits or bytes.

Similarly, when a two's-complement number is shifted to the right, the most-significant bit, which contains magnitude and the sign information, must be maintained. However when shifted to the left, a 0 is shifted in. These rules preserve the common semantics that left shifts multiply the number by two and right shifts divide the number by two.

Both shifting and doubling the precision are important for some multiplication algorithms. Note that unlike addition and subtraction, precision extension and right shifting are done differently for signed vs unsigned numbers.

The most negative number

With only one exception, started with any number in two's-complement representation, if all the bits are flipped and 1 added, the two's-complement representation of the negative of that number is obtained. Positive 12 becomes negative 12, positive 5 becomes negative 5, zero becomes zero(+overflow), etc.

−128	1000 0000
invert bits	0111 1111
add one	1000 0000

The two's complement of −128 results in the same 8-bit binary number.

The two's complement of the minimum number in the range will not have the desired effect of negating the number. For example, the two's complement of −128 in an 8-bit system results in the same binary number. This is because a positive value of 128 cannot be represented with an 8-bit signed binary numeral. Note that this is detected as an overflow condition since there was a carry into but not out of the most-significant bit. This can lead to unexpected bugs in that an unchecked implementation of absolute value could return a negative number in the case of the minimum negative. The *abs* family of integer functions in C typically has this behaviour. This is also true for Java.[4] In this case it is for the developer to decide if there will be a check for the minimum negative value before the call of the function.

The most negative number in two's complement is sometimes called "the weird number," because it is the only exception.[5][6]

Although the number is an exception, it is a valid number in regular two's complement systems. All arithmetic operations work with it both as an operand and (unless there was an overflow) a result.

Why it works

Given a set of all possible `N`-bit values, we can assign the lower (by binary value) half to be the integers from 0 to $(2^{N-1} - 1)$ inclusive and the upper half to be -2^{N-1} to -1 inclusive. The upper half can be used to represent negative integers from -2^{N-1} to -1 because, under addition modulo 2^N they behave the same way as those negative integers. That is to say that because $i + j \bmod 2^N = i + (j + 2^N) \bmod 2^N$ any value in the set $\{\ j + k\ 2^N \mid k \text{ is an integer}\ \}$ can be used in place of `j`.

For example, with eight bits, the unsigned bytes are 0 to 255. Subtracting 256 from the top half (128 to 255) yields the signed bytes −128 to −1.

The relationship to two's complement is realised by noting that 256 = 255 + 1, and $(255 - x)$ is the ones' complement of `x`.

Decimal	Two's complement
127	0111 1111
64	0100 0000
1	0000 0001
0	0000 0000
−1	1111 1111
−64	1100 0000
−127	1000 0001
−128	1000 0000

Some special numbers to note

Example

−95 modulo 256 is equivalent to 161 since

−95 + 256

= −95 + 255 + 1

= 255 − 95 + 1

= 160 + 1

= 161

```
  1111 1111                        255
- 0101 1111                      - 95
===========                       =====
  1010 0000  (ones' complement)    160
+         1                      +   1
===========                       =====
  1010 0001  (two's complement)    161
```

Two's complement	Decimal
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	−1
1110	−2
1101	−3
1100	−4
1011	−5
1010	−6
1001	−7
1000	−8

Two's complement using a 4-bit integer

Fundamentally, the system represents negative integers by counting backward and wrapping around. The boundary between positive and negative numbers is arbitrary, but the de facto rule is that all negative numbers have a left-most bit (most significant bit) of one. Therefore, the most positive 4-bit number is 0111 (7) and the most negative is 1000 (−8). Because of the use of the left-most bit as the sign bit, the absolute value of the most negative number ($|-8| = 8$) is too large to represent. For example, an 8-bit number can only represent every integer from −128 to 127 ($2^{8-1} = 128$) inclusive. Negating a two's complement number is simple: Invert all the bits and add one to the result. For example, negating 1111, we get 0000 + 1 = 1. Therefore, 1111 must represent −1.

The system is useful in simplifying the implementation of arithmetic on computer hardware. Adding 0011 (3) to 1111 (−1) at first seems to give the incorrect answer of 10010. However, the hardware can simply ignore the left-most bit to give the correct answer of 0010 (2). Overflow checks still must exist to catch operations such as summing 0100 and 0100.

The system therefore allows addition of negative operands without a subtraction circuit and a circuit that detects the sign of a number. Moreover, that addition circuit can also perform subtraction by taking the two's complement of a number (see below), which only requires an additional cycle or its own adder circuit. To perform this, the circuit merely pretends an extra left-most bit of 1 exists.

Arithmetic operations

Addition

Adding two's-complement numbers requires no special processing if the operands have opposite signs: the sign of the result is determined automatically. For example, adding 15 and −5:

```
 11111 111   (carry)
  0000 1111  (15)
+ 1111 1011  (−5)
==================
  0000 1010  (10)
```

This process depends upon restricting to 8 bits of precision; a carry to the (nonexistent) 9th most significant bit is ignored, resulting in the arithmetically correct result of 10_{10}.

The last two bits of the carry row (reading right-to-left) contain vital information: whether the calculation resulted in an arithmetic overflow, a number too large for the binary system to represent (in this case greater than 8 bits). An overflow condition exists when these last two bits are different from one another. As mentioned above, the sign of the number is encoded in the MSB of the result.

In other terms, if the left two carry bits (the ones on the far left of the top row in these examples) are both 1s or both 0s, the result is valid; if the left two carry bits are "1 0" or "0 1", a sign overflow has occurred. **Conveniently, an XOR operation on these two bits can quickly determine if an overflow condition exists.** As an example, consider the signed 4-bit addition of 7 and 3:

```
 0111   (carry)
  0111  (7)
+ 0011  (3)
=============
  1010  (−6)  invalid!
```

In this case, the far left two (MSB) carry bits are "01", which means there was a two's-complement addition overflow. That is, $1010_2 = 10_{10}$ is outside the permitted range of −8 to 7.

In general, any two `N`-bit numbers may be added *without* overflow, by first sign-extending both of them to $N + 1$ bits, and then adding as above. The $N + 1$ bits result is large enough to represent any possible sum ($N = 5$ two's complement can represent values in the range −16 to 15) so overflow will never occur. It is then possible, if desired, to 'truncate' the result back to `N` bits while preserving the value if and only if the discarded bit is a proper sign extension of the retained result bits. This provides another method of detecting overflow—which is equivalent to the method of comparing the carry bits—but which may be easier to implement in some situations, because it does not require access to the internals of the addition.

Subtraction

Computers usually use the method of complements to implement subtraction. Using complements for subtraction is closely related to using complements for representing negative numbers, since the combination allows all signs of operands and results; direct subtraction works with two's-complement numbers as well. Like addition, the advantage of using two's complement is the elimination of examining the signs of the operands to determine if addition or subtraction is needed. For example, subtracting −5 from 15 is really adding 5 to 15, but this is hidden by the two's-complement representation:

```
 11110 000   (borrow)
  0000 1111  (15)
− 1111 1011  (−5)
===========
  0001 0100  (20)
```

Overflow is detected the same way as for addition, by examining the two leftmost (most significant) bits of the borrows; overflow has occurred if they are different.

Another example is a subtraction operation where the result is negative: 15 − 35 = −20:

```
 11100 000   (borrow)
  0000 1111  (15)
− 0010 0011  (35)
===========
  1110 1100  (−20)
```

As for addition, overflow in subtraction may be avoided (or detected after the operation) by first sign-extending both inputs by an extra bit.

Multiplication

The product of two N-bit numbers requires 2*N* bits to contain all possible values. If the precision of the two, two's complement operands is doubled before the multiplication, direct multiplication (discarding any excess bits beyond that precision) will provide the correct result. For example, take 6 × −5 = −30. First, the precision is extended from 4 bits to 8. Then the numbers are multiplied, discarding the bits beyond 8 (shown by 'x'):

```
     00000110  (6)
 *   11111011  (−5)
 ============
          110
         1100
        00000
       110000
      1100000
     11000000
    x10000000
   xx00000000
 ============
   xx11100010
```

This is very inefficient; by doubling the precision ahead of time, all additions must be double-precision and at least twice as many partial products are needed than for the more efficient algorithms actually implemented in computers.

Some multiplication algorithms are designed for two's complement, notably Booth's multiplication algorithm. Methods for multiplying sign-magnitude numbers don't work with two's-complement numbers without adaptation. There isn't usually a problem when the multiplicand (the one being repeatedly added to form the product) is negative; the issue is setting the initial bits of the product correctly when the multiplier is negative. Two methods for adapting algorithms to handle two's-complement numbers are common:

- First check to see if the multiplier is negative. If so, negate (*i.e.*, take the two's complement of) both operands before multiplying. The multiplier will then be positive so the algorithm will work. Because both operands are negated, the result will still have the correct sign.
- Subtract the partial product resulting from the MSB (pseudo sign bit) instead of adding it like the other partial products. This method requires the multiplicand's sign bit to be extended by one position, being preserved during the shift right actions.[7]

As an example of the second method, take the common add-and-shift algorithm for multiplication. Instead of shifting partial products to the left as is done with pencil and paper, the accumulated product is shifted right, into a second register that will eventually hold the least significant half of the product. Since the least significant bits are not changed once they are calculated, the additions can be single precision, accumulating in the register that will eventually hold the most significant half of the product. In the following example, again multiplying 6 by −5, the two registers and the extended sign bit are separated by "|":

```
 0 0110  (6)   (multiplicand with extended sign bit)
 × 1011 (−5)   (multiplier)
 =|====|====
 0|0110|0000   (first partial product (rightmost bit is 1))
 0|0011|0000   (shift right, preserving extended sign bit)
 0|1001|0000   (add second partial product (next bit is 1))
 0|0100|1000   (shift right, preserving extended sign bit)
 0|0100|1000   (add third partial product: 0 so no change)
 0|0010|0100   (shift right, preserving extended sign bit)
 1|1100|0100   (subtract last partial product since it's from sign bit)
 1|1110|0010   (shift right, preserving extended sign bit)
  |1110|0010   (discard extended sign bit, giving the final answer, −30)
```

Comparison (ordering)

Comparison is often implemented with a dummy subtraction, where the flags in the computer's status register are checked, but the main result is ignored. The zero flag indicates if two values compared equal. If the exclusive-or of the sign and overflow flags is 1, the subtraction result was less than zero, otherwise the result was zero or greater. These checks are often implemented in computers in conditional branch instructions.

Unsigned binary numbers can be ordered by a simple lexicographic ordering, where the bit value 0 is defined as less than the bit value 1. For two's complement values, the meaning of the most significant bit is reversed (i.e. 1 is less than 0).

The following algorithm (for an n-bit two's complement architecture) sets the result register R to −1 if A < B, to +1 if A > B, and to 0 if A and B are equal:

```
Reversed comparison of sign bit:
if A(n-1) == 0 and B(n-1) == 1 then
     R := +1
     break
  else if A(n-1) == 1 and B(n-1) == 0 then
```

```
     R := -1
     break
  end

Comparison of remaining bits:
for i = n-2...0 do
  if A(i) == 0 and B(i) == 1 then
     R := -1
     break
  else if A(i) == 1 and B(i) == 0 then
     R := +1
     break
  end
end

R := 0
```

Two's complement and universal algebra

In a classic *HAKMEM* published by the MIT AI Lab in 1972, Bill Gosper noted that whether or not a machine's internal representation was two's-complement could be determined by summing the successive powers of two. In a flight of fancy, he noted that the result of doing this algebraically indicated that "algebra is run on a machine (the universe) which is two's-complement."[8]

Gosper's end conclusion is not necessarily meant to be taken seriously, and it is akin to a mathematical joke. The critical step is "...110 = ...111 − 1", i.e., "$2X = X - 1$", and thus $X = ...111 = -1$. This presupposes a method by which an infinite string of 1s is considered a number, which requires an extension of the finite place-value concepts in elementary arithmetic. It is meaningful either as part of a two's-complement notation for all integers, as a typical 2-adic number, or even as one of the generalized sums defined for the divergent series of real numbers 1 + 2 + 4 + 8 + ⋯.[9] Digital arithmetic circuits, idealized to operate with infinite (extending to positive powers of 2) bit strings, produce 2-adic addition and multiplication compatible with two's complement representation.[10] Continuity of binary arithmetical and bitwise operations in 2-adic metric also has some use in cryptography.[11]

References

[1] David J. Lilja and Sachin S. Sapatnekar, *Designing Digital Computer Systems with Verilog*, Cambridge University Press, 2005 online (http://books.google.com/books?vid=ISBN052182866X&id=5BvW0hYhxkQC&pg=PA37&lpg=PA37&ots=l-E0VjyPt8&dq="two's+complement+arithmetic"&sig=sS5_swrfrzcQI2nHWest75sIjgg)

[2] E.g. "Signed integers are two's complement binary values that can be used to represent both positive and negative integer values.", Section 4.2.1 in Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, November 2006

[3] For $x = 0$ we have $2^N - 0 = 2^N$, which is equivalent to 0* = 0 modulo 2^N (i.e. after restricting to `N` least significant bits).

[4] "Math (Java Platform SE 7)" (http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html). .

[5] Reynald Affeldt and Nicolas Marti. "Formal Verification of Arithmetic Functions in SmartMIPS Assembly" (http://www.ipl.t.u-tokyo.ac.jp/jssst2006/papers/Affeldt.pdf). .

[6] google.com (http://books.google.com/books?id=5X7JV5-n0FIC&pg=PA19&dq="weird+number"+binary); "Digital Design and Computer Architecture", by David Harris, David Money Harris, Sarah L. Harris. 2007. Page 18.

[7] Wakerly, John F. (2000). *Digital Design Principles & Practices* (3rd ed.). Prentice Hall. p. 47. ISBN 0-13-769191-2.

[8] Hakmem - Programming Hacks - Draft, Not Yet Proofed (http://www.inwap.com/pdp10/hbaker/hakmem/hacks.html#item154)

[9] For the summation of 1 + 2 + 4 + 8 + ⋯ without recourse to the 2-adic metric, see Hardy, G.H. (1949). *Divergent Series*. Clarendon Press. LCC QA295 .H29 1967. (pp. 7–10)

[10] Vuillemin, Jean (1993). *On circuits and numbers* (http://www.hpl.hp.com/techreports/Compaq-DEC/PRL-RR-25.pdf). Paris: Digital Equipment Corp.. p. 19. . Retrieved 2012-01-24., Chapter 7, especially 7.3 for multiplication.

[11] Anashin, Vladimir; Bogdanov, Andrey; Kizhvatov, Ilya (2007). "ABC Stream Cipher" (http://crypto.rsuh.ru/). Russian State University for the Humanities. . Retrieved 24 January 2012.

Further reading

- Koren, Israel (2002). *Computer Arithmetic Algorithms*. A.K. Peters. ISBN 1-56881-160-8.
- Flores, Ivan (1963). *The Logic of Computer Arithmetic*. Prentice-Hall.

External links

- Tutorial: Two's Complement Numbers (http://www.vb-helper.com/tutorial_twos_complement.html)
- Two's complement array multiplier JavaScript simulator (http://www.ecs.umass.edu/ece/koren/arith/simulator/ArrMlt/)
- Javascript converter for 2's complement to decimal and vice versa (http://prozessorsimulation.klickagent.ch/?lang=en&convertor=true)

Ones' complement

The **ones' complement** of a binary number is defined as the value obtained by inverting all the bits in the binary representation of the number (swapping 0's for 1's and vice-versa). The ones' complement of the number then behaves like the negative of the original number in some arithmetic operations. To within a constant (of −1), the ones' complement behaves like the negative of the original number with binary addition. However, unlike two's complement, these numbers have not seen widespread use because of issues such as the offset of −1, that negating zero results in a distinct **negative zero** bit pattern, less simplicity with arithmetic borrowing, etc.

A **ones' complement system** or **ones' complement arithmetic** is a system in which negative numbers are represented by the arithmetic negative of the value. In such a system, a number is negated (converted from positive to negative or vice versa) by computing its ones' complement. An N-bit ones' complement numeral system can only represent integers in the range $-(2^{N-1}-1)$ to $2^{N-1}-1$ while two's complement can express -2^{N-1} to $2^{N-1}-1$.

The **ones' complement binary** numeral system is characterized by the bit complement of any integer value being the arithmetic negative of the value. That is, inverting all of the bits of a number (the logical complement) produces the same result as subtracting the value from 0.

History

The early days of digital computing were marked by a lot of competing ideas about both hardware technology and mathematics technology (numbering systems). One of the great debates was the format of negative numbers, with some of the era's most expert people having very strong and different opinions. One camp supported two's complement, the system that is dominant today. Another camp supported ones' complement, where any positive value is made into its negative equivalent by inverting all of the bits in a word. A third group supported "sign & magnitude", where a value is changed from positive to negative simply by toggling the word's sign (high order) bit.

There were arguments for and against each of the systems. Sign & magnitude allowed for easier tracing of memory dumps (a common process 40 years ago) as numeric values tended to use fewer 1 bits. Internally, these systems did ones' complement math so numbers would have to be converted to ones' complement values when they were transmitted from a register to the math unit and then converted back to sign-magnitude when the result was transmitted back to the register. The electronics required more gates than the other systems – a key concern when the cost and packaging of discrete transistors was critical. IBM was one of the early supporters of sign-magnitude, with their 7090 (709x series) computers perhaps the best known architecture to use it.

Ones' complement allowed for somewhat simpler hardware designs as there was no need to convert values when passed to/from the math unit. But it also shared an undesirable characteristic with sign-magnitude – the ability to represent negative zero (–0). Negative zero behaves exactly like positive zero; when used as an operand in any calculation, the result will be the same whether an operand is positive or negative zero. The disadvantage, however, is that the existence of two forms of the same value necessitates two rather than a single comparison when checking for equality with zero. Ones' complement subtraction can also result in an end-around borrow (described below). It can be argued that this makes the addition/subtraction logic more complicated or that it makes it simpler as a subtraction requires simply inverting the bits of the second operand as it's passed to the adder. The CDC 6000 series and UNIVAC 1100 series computers were based on ones' complement.

Two's complement is the easiest to implement in hardware, which may be the ultimate reason for its widespread popularity. Remember that processors on the early mainframes often consisted of thousands of transistors – eliminating a significant number of transistors was a significant cost savings. The architects of the early integrated circuit based CPUs (Intel 8080, etc.) chose to use two's complement math. As IC technology advanced, virtually all adopted two's complement technology. Intel, AMD, and IBM POWER chips are all two's complement.

Number representation

Positive numbers are the same simple, binary system used by two's complement and sign-magnitude. Negative values are the bit complement of the corresponding positive value. The largest positive value is characterized by the sign (high-order) bit being off (0) and all other bits being on (1). The smallest negative value is characterized by the sign bit being 1, and all other bits being 0. The table below shows all possible values in a 4-bit system, from –7 to +7.

```
    +      –
0   0000   1111   ––Note that +0 and –0 return TRUE when tested for zero, FALSE when tested for non-zero.
1   0001   1110
2   0010   1101
3   0011   1100
4   0100   1011
5   0101   1010
6   0110   1001
7   0111   1000
```

Basics

Adding two values is straight forward. Simply align the values on the least significant bit and add, propagating any carry to the bit one position left. If the carry extends past the end of the word it is said to have "wrapped around", a condition called an "end-around carry". When this occurs, the bit must be added back in at the right-most bit. This phenomenon does not occur in two's complement arithmetic.

```
  0001 0110     22
+ 0000 0011      3
===========   ====
  0001 1001     25
```

Subtraction is similar, except that borrows, rather than carries, are propagated to the left. If the borrow extends past the end of the word it is said to have "wrapped around", a condition called an "end-around borrow". When this occurs, the bit must be subtracted from the right-most bit. This phenomenon does not occur in two's complement arithmetic.

```
  0000 0110      6
- 0001 0011     19
===========   ====
1 1111 0011    -12    ←An end-around borrow is produced, and the sign bit of the intermediate result is 1.
- 0000 0001      1    ←Subtract the end-around borrow from the result.
===========   ====
  1111 0010    -13    ←The correct result (6 - 19 = -13)
```

It is easy to demonstrate that the bit complement of a positive value is the negative magnitude of the positive value. The computation of 19 + 3 produces the same result as 19 − (−3).

Add 3 to 19.

```
  0001 0011     19
+ 0000 0011      3
===========   ====
  0001 0110     22
```

Subtract −3 from 19.

```
  0001 0011     19
- 1111 1100     -3
===========   ====
1 0001 0111     23    ←An end-around borrow is produced.
- 0000 0001      1    ←Subtract the end-around borrow from the result.
===========   ====
  0001 0110     22    ←The correct result (19 - (-3) = 22).
```

Negative zero

Negative zero is the condition where all bits in a signed word are 1. This follows the ones' complement rules that a value is negative when the left-most bit is 1, and that a negative number is the bit complement of the number's magnitude. The value also behaves as zero when computing. Adding or subtracting negative zero to/from another value produces the original value.

Adding negative zero:

```
  0001 0110     22
+ 1111 1111     -0
===========   ====
1 0001 0101     21    ←An end-around carry is produced.
+ 0000 0001      1
===========   ====
  0001 0110     22    ←The correct result (22 + (-0) = 22)
```

Subtracting negative zero:

```
  0001 0110     22
- 1111 1111     -0
===========   ====
1 0001 0111     23    ←An end-around borrow is produced.
- 0000 0001      1
===========   ====
```

```
  0001 0110     22    ←The correct result (22 – (–0) = 22)
```

Negative zero is easily produced in a 1's complement adder. Simply add the positive and negative of the same magnitude.

```
  0001 0110     22
+ 1110 1001    –22
============   ====
  1111 1111     –0    ←Negative zero.
```

Although the math always produces the correct results, a side effect of negative zero is that software must test for negative zero.

Avoiding negative zero

The generation of negative zero becomes a non-issue if addition is achieved with a complementing subtractor. The first operand is passed to the subtract unmodified, the second operand is complemented, and the subtraction generates the correct result, avoiding negative zero. The previous example added 22 and –22 and produced –0.

```
  0001 0110     22          0001 0110     22                 1110 1001   –22          1110 1001   –22
+ 1110 1001    –22        – 0001 0110     22               + 0001 0110    22        – 1110 1001   –22
============   ====  but   ============   ====   likewise,   ============   ===  but   ============   ===
  1111 1111     –0          0000 0000      0                  1111 1111    –0          0000 0000      0
```

The interesting "corner cases" are when one or both operands are zero and/or negative zero.

```
  0001 0010     18            0001 0010     18
– 0000 0000      0          – 1111 1111    –0
============   ====          ============   ====
  0001 0010     18           1 0001 0011    19
                             – 0000 0001     1
                             ============   ====
                               0001 0010    18
```

Subtracting +0 is trivial (as shown above). If the second operand is negative zero it is inverted and the original value of the first operand is the result. Subtracting –0 is also trivial. The result can be only 1 of two cases. In case 1, operand 1 is –0 so the result is produced simply by subtracting 1 from 1 at every bit position. In case 2, the subtraction will generate a value that is 1 larger than operand 1 and an end around borrow. Completing the borrow generates the same value as operand 1.

The only really interesting case is when both operands are plus or minus zero. Look at this example:

```
  0000 0000      0          0000 0000      0          1111 1111     –0          1111 1111     –0
+ 0000 0000      0        + 1111 1111     –0        + 0000 0000      0        + 1111 1111     –0
============   ====       ============   ====       ============   ====       ============   ====
  0000 0000      0          1111 1111     –0          1111 1111     –0        1 1111 1110     –1
                                                                             + 0000 0001      1
                                                                             ==================
                                                                               1111 1111     –0
```

```
  0000 0000      0          0000 0000      0          1111 1111     –0          1111 1111     –0
– 1111 1111     –0        – 0000 0000      0        – 1111 1111     –0        – 0000 0000      0
============   ====       ============   ====       ============   ====       ============   ====
```

```
1 0000 0001      1         0000 0000      0         0000 0000      0         1111 1111     −0
− 0000 0001      1
===========    ====
  0000 0000      0
```

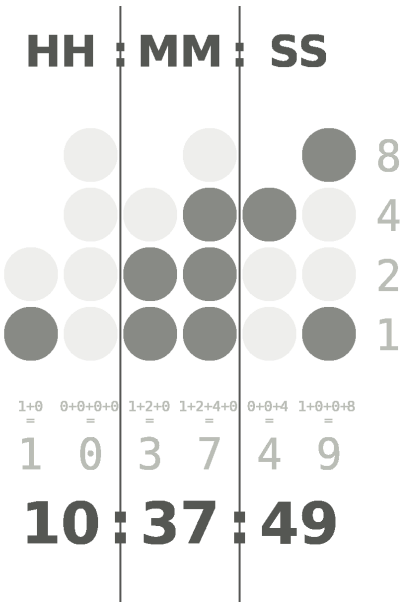
This example shows that of the 4 possible conditions when adding only ±0, an adder will produce −0 in three of them. A complementing subtractor will produce −0 only when both operands are −0.

References

Donald Knuth: *The Art of Computer Programming*, Volume 2: Seminumerical Algorithms, chapter 4.1

Binary-coded decimal

In computing and electronic systems, **binary-coded decimal** (**BCD**) is a class of binary encodings of decimal numbers where each decimal digit is represented by a fixed number of bits, usually four or eight, although other sizes (such as six bits) have been used historically. Special bit patterns are sometimes used for a sign or for other indications (e.g., error or overflow).



A binary clock might use LEDs to express binary values. In this clock, each column of LEDs shows a binary-coded decimal numeral of the traditional sexagesimal time.

In byte-oriented systems (i.e. most modern computers), the term *uncompressed* BCD usually implies a full byte for each digit (often including a sign), whereas *packed* BCD typically encodes two decimal digits within a single byte by taking advantage of the fact that four bits are enough to represent the range 0 to 9. The precise 4-bit encoding may vary however, for technical reasons, see Excess-3 for instance.

BCD's main virtue is a more accurate representation and rounding of decimal quantities as well as an ease of conversion into human-readable representations. As compared to binary positional systems, BCD's principal drawbacks are a small increase in the complexity of the circuits needed to implement basic arithmetics and a slightly less dense storage.

BCD was used in many early decimal computers. Although BCD is not as widely used as in the past, decimal fixed-point and floating-point formats are still important and continue to be used in financial, commercial, and industrial computing, where subtle conversion and rounding errors that are inherent to floating point binary representations cannot be tolerated.[1]

Basics

As described in the introduction, BCD takes advantage of the fact that any one decimal numeral can be represented by a four bit pattern:

Decimal Digit	BCD 8 4 2 1
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

As most computers store data in 8-bit bytes, it is possible to use one of the following methods to encode a BCD number:

- **Uncompressed**: each numeral is encoded into one byte, with four bits representing the numeral and the remaining bits having no significance.
- **Packed**: two numerals are encoded into a single byte, with one numeral in the least significant nibble (bits 0 through 3) and the other numeral in the most significant nibble (bits 4 through 7).

As an example, encoding the decimal number **`91`** using uncompressed BCD results in the following binary pattern of two bytes:

```
Decimal:          9          1
Binary :  0000 1001  0000 0001
```

In packed BCD, the same number would fit into a single byte:

```
Decimal:     9    1
Binary :  1001 0001
```

Hence the numerical range for one uncompressed BCD byte is zero through nine inclusive, whereas the range for one packed BCD is zero through ninety-nine inclusive.

To represent numbers larger than the range of a single byte any number of contiguous bytes may be used. For example, to represent the decimal number **`12345`** in packed BCD, using big-endian format, a program would encode as follows:

```
Decimal:          1     2    3     4    5
Binary :  0000 0001  0010 0011  0100 0101
```

Note that the most significant nibble of the most significant byte is zero, implying that the number is in actuality **`012345`**. Also note how packed BCD is more efficient in storage usage as compared to uncompressed BCD; encoding the same number in uncompressed format would consume 100 percent more storage.

Shifting and masking operations are used to pack or unpack a packed BCD digit. Other logical operations are used to convert a numeral to its equivalent bit pattern or reverse the process.

BCD in Electronics

BCD is very common in electronic systems where a numeric value is to be displayed, especially in systems consisting solely of digital logic, and not containing a microprocessor. By utilizing BCD, the manipulation of numerical data for display can be greatly simplified by treating each digit as a separate single sub-circuit. This matches much more closely the physical reality of display hardware—a designer might choose to use a series of separate identical seven-segment displays to build a metering circuit, for example. If the numeric quantity were stored and manipulated as pure binary, interfacing to such a display would require complex circuitry. Therefore, in cases where the calculations are relatively simple working throughout with BCD can lead to a simpler overall system than converting to binary.

The same argument applies when hardware of this type uses an embedded microcontroller or other small processor. Often, smaller code results when representing numbers internally in BCD format, since a conversion from or to binary representation can be expensive on such limited processors. For these applications, some small processors feature BCD arithmetic modes, which assist when writing routines that manipulate BCD quantities.

Packed BCD

A common variation of the two-digits-per-byte encoding is called **packed BCD** (or simply **packed decimal**), which has been in use since the 1960s or earlier and implemented in all IBM mainframe hardware since then. In most representations, one or more bytes hold a decimal integer, where each of the two nibbles of each byte represent a decimal digit, with the more significant digit in the upper half of each byte, and with leftmost byte (residing at the lowest memory address) containing the most significant digits of the packed decimal value. The lower nibble of the rightmost byte is usually used as the sign flag (although in some representations this nibble may be used as the least significant digit if the packed decimal value does not have a sign at all, i.e., is purely unsigned). As an example, a 4-byte value consists of 8 nibbles, wherein the upper 7 nibbles store the digits of a 7-digit decimal value and the lowest nibble indicates the sign of the decimal integer value.

Standard sign values are 1100 (hex C) for positive (+) and 1101 (D) for negative (−). This convention was derived from abbreviations for accounting terms (Credit and Debit), as packed decimal coding was widely used in accounting systems. Other allowed signs are 1010 (A) and 1110 (E) for positive and 1011 (B) for negative. Some implementations also provide unsigned BCD values with a sign nibble of 1111 (F). ILE RPG uses 1111 (F) for positive and 1101 (D) for negative.[2] In packed BCD, the number 127 is represented by 0001 0010 0111 1100 (127C) and −127 is represented by 0001 0010 0111 1101 (127D). Burroughs systems used 1101 (D) for negative, and any other value was considered a positive sign value (the processors would normalize a positive sign to 1100 (C)).

Sign Digit	BCD 8 4 2 1	Sign	Notes
A	1 0 1 0	**+**	
B	1 0 1 1	**−**	
C	1 1 0 0	**+**	Preferred
D	1 1 0 1	**−**	Preferred
E	1 1 1 0	**+**	
F	1 1 1 1	**+**	Unsigned

No matter how many bytes wide a word is, there are always an even number of nibbles because each byte has two of them. Therefore, a word of n bytes can contain up to $(2n)-1$ decimal digits, which is always an odd number of digits. A decimal number with d digits requires $\frac{1}{2}(d+1)$ bytes of storage space.

For example, a 4-byte (32-bit) word can hold seven decimal digits plus a sign, and can represent values ranging from ±9,999,999. Thus the number −1,234,567 is 7 digits wide and is encoded as:

```
0001 0010 0011 0100 0101 0110 0111 1101
1    2    3    4    5    6    7    −
```

(Note that, like character strings, the first byte of the packed decimal − with the most significant two digits − is usually stored in the lowest address in memory, independent of the endianness of the machine.)

In contrast, a 4-byte binary two's complement integer can represent values from −2,147,483,648 to +2,147,483,647.

While packed BCD does not make optimal use of storage (about $^{1}/_{6}$ of the memory used is wasted), conversion to ASCII, EBCDIC, or the various encodings of Unicode is still trivial, as no arithmetic operations are required. The extra storage requirements are usually offset by the need for the accuracy and compatibility with calculator or hand calculation that fixed-point decimal arithmetic provides. Denser packings of BCD exist which avoid the storage penalty and also need no arithmetic operations for common conversions.

Packed BCD is supported in the COBOL programming language as the "COMPUTATIONAL-3" (an IBM extension adopted by many other compiler vendors) or "PACKED-DECIMAL" (part of the 1985 COBOL standard) data type. Besides the IBM System/360 and later compatible mainframes, packed BCD was implemented in the native instruction set of the original VAX processors from Digital Equipment Corporation and was the native format for the Burroughs Corporation Medium Systems line of mainframes (descended from the 1950s Electrodata 200 series).

Fixed-point packed decimal

Fixed-point decimal numbers are supported by some programming languages (such as COBOL and PL/I). These languages allow the programmer to specify an implicit decimal point in front of one of the digits. For example, a packed decimal value encoded with the bytes 12 34 56 7C represents the fixed-point value +1,234.567 when the implied decimal point is located between the 4th and 5th digits:

```
12 34 56 7C
12 34.56 7+
```

The decimal point is not actually stored in memory, as the packed BCD storage format does not provide for it. Its location is simply known to the compiler and the generated code acts accordingly for the various arithmetic operations.

Higher-density encodings

If a decimal digit requires four bits, then three decimal digits require 12 bits. However, since 2^{10} (1,024) is greater than 10^{3} (1,000), if three decimal digits are encoded together, only 10 bits are needed. Two such encodings are *Chen-Ho encoding* and *Densely Packed Decimal*. The latter has the advantage that subsets of the encoding encode two digits in the optimal seven bits and one digit in four bits, as in regular BCD.

Zoned decimal

Some implementations, for example IBM mainframe systems, support **zoned decimal** numeric representations. Each decimal digit is stored in one byte, with the lower four bits encoding the digit in BCD form. The upper four bits, called the "zone" bits, are usually set to a fixed value so that the byte holds a character value corresponding to the digit. EBCDIC systems use a zone value of 1111 (hex F); this yields bytes in the range F0 to F9 (hex), which are the EBCDIC codes for the characters "0" through "9". Similarly, ASCII systems use a zone value of 0011 (hex 3), giving character codes 30 to 39 (hex).

For signed zoned decimal values, the rightmost (least significant) zone nibble holds the sign digit, which is the same set of values that are used for signed packed decimal numbers (see above). Thus a zoned decimal value encoded as the hex bytes F1 F2 D3 represents the signed decimal value −123:

```
F1 F2 D3
 1  2 -3
```

EBCDIC zoned decimal conversion table

BCD Digit	Hexadecimal				EBCDIC Character			
0+	C0	A0	E0	F0	{ (*)		\ (*)	0
1+	C1	A1	E1	F1	A	~ (*)		1
2+	C2	A2	E2	F2	B	s	S	2
3+	C3	A3	E3	F3	C	t	T	3
4+	C4	A4	E4	F4	D	u	U	4
5+	C5	A5	E5	F5	E	v	V	5
6+	C6	A6	E6	F6	F	w	W	6
7+	C7	A7	E7	F7	G	x	X	7
8+	C8	A8	E8	F8	H	y	Y	8
9+	C9	A9	E9	F9	I	z	Z	9
0−	D0	B0			} (*)	^ (*)		
1−	D1	B1			J			
2−	D2	B2			K			
3−	D3	B3			L			
4−	D4	B4			M			
5−	D5	B5			N			
6−	D6	B6			O			
7−	D7	B7			P			
8−	D8	B8			Q			
9−	D9	B9			R			

(*) *Note: These characters vary depending on the local character code page setting.*

Fixed-point zoned decimal

Some languages (such as COBOL and PL/I) directly support fixed-point zoned decimal values, assigning an implicit decimal point at some location between the decimal digits of a number. For example, given a six-byte signed zoned decimal value with an implied decimal point to the right of the fourth digit, the hex bytes F1 F2 F7 F9 F5 C0 represent the value +1,279.50:

```
F1 F2 F7 F9 F5 C0
 1  2  7  9. 5 +0
```

IBM and BCD

IBM used the terms **binary-coded decimal** and **BCD** for 6-bit *alphanumeric* codes that represented numbers, upper-case letters and special characters. Some variation of BCD *alphamerics* was used in most early IBM computers, including the IBM 1620, IBM 1400 series, and non-Decimal Architecture members of the IBM 700/7000 series.

The IBM 1400 series were character-addressable machines, each location being six bits labeled *B, A, 8, 4, 2* and *1,* plus an odd parity check bit (*C*) and a word mark bit (*M*). For encoding digits *1* through *9*, *B* and *A* were zero and the digit value represented by standard 4-bit BCD in bits *8* through *1*. For most other characters bits *B* and *A* were derived simply from the "12", "11", and "0" "zone punches" in the punched card character code, and bits *8* through *1* from the *1* through *9* punches. A "12 zone" punch set both *B* and *A*, an "11 zone" set *B*, and a "0 zone" (a 0 punch combined with any others) set *A*. Thus the letter **A**, *(12,1)* in the punched card format, was encoded *(B,A,1)* and the currency symbol **$**, *(11,8,3)* in the punched card, as *(B,8,3)*. This allowed the circuitry to convert between the punched card format and the internal storage format to be very simple with only a few special cases. One important special case was digit *0*, represented by a lone *0* punch in the card, and *(8,2)* in core memory. [3]

The memory of the IBM 1620 was organized into 6-bit addressable digits, the usual *8, 4, 2, 1* plus *F*, used as a flag bit and *C*, an odd parity check bit. BCD *alphamerics* were encoded using digit pairs, with the "zone" in the even-addressed digit and the "digit" in the odd-addressed digit, the "zone" being related to the *12*, *11*, and *0* "zone punches" as in the 1400 series. Input/Output translation hardware converted between the internal digit pairs and the external standard 6-bit BCD codes.

In the Decimal Architecture IBM 7070, IBM 7072, and IBM 7074 *alphamerics* were encoded using digit pairs (using two-out-of-five code in the digits, **not** BCD) of the 10-digit word, with the "zone" in the left digit and the "digit" in the right digit. Input/Output translation hardware converted between the internal digit pairs and the external standard 6-bit BCD codes.

With the introduction of System/360, IBM expanded 6-bit BCD *alphamerics* to 8-bit EBCDIC, allowing the addition of many more characters (e.g., lowercase letters). A variable length Packed BCD *numeric* data type was also implemented, providing machine instructions that performed arithmetic directly on packed decimal data.

On the IBM 1130 and 1800, packed BCD was supported in software by IBM's Commercial Subroutine Package.

Today, BCD data is still heavily used in IBM processors and databases, such as IBM DB2, mainframes, and Power6. In these products, the BCD is usually zoned BCD (as in EBCDIC or ASCII), Packed BCD (two decimal digits per byte), or "pure" BCD encoding (one decimal digit stored as BCD in the low four bits of each byte). All of these are used within hardware registers and processing units, and in software.

Other computers and BCD

The Digital Equipment Corporation VAX-11 series included instructions that could perform arithmetic directly on packed BCD data and convert between packed BCD data and other integer representations. The VAX's packed BCD format was compatible with that on IBM System/360 and IBM's later compatible processors. The MicroVAX and later VAX implementations dropped this ability from the CPU but retained code compatibility with earlier machines by implementing the missing instructions in an operating system-supplied software library. This was invoked automatically via exception handling when the no longer implemented instructions were encountered, so that programs using them could execute without modification on the newer machines.

In more recent computers such capabilities are almost always implemented in software rather than the CPU's instruction set, but BCD numeric data is still extremely common in commercial and financial applications.

Addition with BCD

It is possible to perform addition in BCD by first adding in binary, and then converting to BCD afterwards. Conversion of the simple sum of two digits can be done by adding 6 (that is, 16 − 10) when the five-bit result of adding a pair of digits has a value greater than 9. For example:

```
1001 + 1000 = 10001
   9 +    8 =    17
```

Note that 10001 is the binary, not decimal, representation of the desired result. In BCD as in decimal, there cannot exist a value greater than 9 (1001) per digit. To correct this, 6 (0110) is added to that sum and then the result is treated as two nibbles:

```
10001 + 0110 = 00010111 => 0001 0111
   17 +    6 =       23       1    7
```

The two nibbles of the result, 0001 and 0111, correspond to the digits "1" and "7". This yields "17" in BCD, which is the correct result.

This technique can be extended to adding multiple digits by adding in groups from right to left, propagating the second digit as a carry, always comparing the 5-bit result of each digit-pair sum to 9.

Subtraction with BCD

Subtraction is done by adding the ten's complement of the subtrahend. To represent the sign of a number in BCD, the number 0000 is used to represent a positive number, and 1001 is used to represent a negative number. The remaining 14 combinations are invalid signs. To illustrate signed BCD subtraction, consider the following problem: 357 − 432.

In signed BCD, 357 is 0000 0011 0101 0111. The ten's complement of 432 can be obtained by taking the nine's complement of 432, and then adding one. So, 999 − 432 = 567, and 567 + 1 = 568. By preceding 568 in BCD by the negative sign code, the number −432 can be represented. So, −432 in signed BCD is 1001 0101 0110 1000.

Now that both numbers are represented in signed BCD, they can be added together:

```
0000 0011 0101 0111 + 1001 0101 0110 1000 = 1001 1000 1011 1111
   0    3    5    7 +    9    5    6    8 =    9    8   11   15
```

Since BCD is a form of decimal representation, several of the digit sums above are invalid. In the event that an invalid entry (any BCD digit greater than 1001) exists, 6 is added to generate a carry bit and cause the sum to become a valid entry. The reason for adding 6 is that there are 16 possible 4-bit BCD values (since $2^4 = 16$), but only 10 values are valid (0000 through 1001). So adding 6 to the invalid entries results in the following:

```
1001 1000 1011 1111 + 0000 0000 0110 0110 = 1001 1001 0010 0101
   9    8   11   15 +    0    0    6    6 =    9    9    2    5
```

Thus the result of the subtraction is 1001 1001 0010 0101 (-925). To check the answer, note that the first bit is the sign bit, which is negative. This seems to be correct, since 357 − 432 should result in a negative number. To check the rest of the digits, represent them in decimal. 1001 0010 0101 is 925. The ten's complement of 925 is 1000 − 925 = 999 − 925 + 1 = 074 + 1 = 75, so the calculated answer is −75. To check, perform standard subtraction to verify that 357 − 432 is −75.

Note that in the event that there are a different number of nibbles being added together (such as 1053 − 122), the number with the fewest number of digits must first be padded with zeros before taking the ten's complement or subtracting. So, with 1053 − 122, 122 would have to first be represented as 0122, and the ten's complement of 0122 would have to be calculated.

Background

The binary-coded decimal scheme described in this article is the most common encoding, but there are many others. The method here can be referred to as *Simple Binary-Coded Decimal* (*SBCD*) or *BCD 8421*. In the headers to the table, the '8 4 2 1', *etc.*, indicates the weight of each bit shown; note that in the fifth column two of the weights are negative. Both ASCII and EBCDIC character codes for the digits are examples of zoned BCD, and are also shown in the table.

The following table represents decimal digits from 0 to 9 in various BCD systems:

Digit	BCD 8 4 2 1	Excess-3 or Stibitz Code	BCD 2 4 2 1 or Aiken Code	BCD 8 4 −2 −1	IBM 702 IBM 705 IBM 7080 IBM 1401 8 4 2 1	ASCII 0000 8421	EBCDIC 0000 8421
0	0000	0011	0000	0000	1010	0011 0000	1111 0000
1	0001	0100	0001	0111	0001	0011 0001	1111 0001
2	0010	0101	0010	0110	0010	0011 0010	1111 0010
3	0011	0110	0011	0101	0011	0011 0011	1111 0011
4	0100	0111	0100	0100	0100	0011 0100	1111 0100
5	0101	1000	1011	1011	0101	0011 0101	1111 0101
6	0110	1001	1100	1010	0110	0011 0110	1111 0110
7	0111	1010	1101	1001	0111	0011 0111	1111 0111
8	1000	1011	1110	1000	1000	0011 1000	1111 1000
9	1001	1100	1111	1111	1001	0011 1001	1111 1001

Legal history

In the 1972 case Gottschalk v. Benson, the U.S. Supreme Court overturned a lower court decision which had allowed a patent for converting BCD encoded numbers to binary on a computer. This was an important case in determining the patentability of software and algorithms.

Comparison with pure binary

Advantages

- Many non-integral values, such as decimal 0.2, have an infinite place-value representation in binary (.001100110011...) but have a finite place-value in binary-coded decimal (0.0010). Consequently a system based on binary-coded decimal representations of decimal fractions avoids errors representing and calculating such values.
- Scaling by a factor of 10 (or a power of 10) is simple; this is useful when a decimal scaling factor is needed to represent a non-integer quantity (e.g., in financial calculations)
- Rounding at a decimal digit boundary is simpler. Addition and subtraction in decimal does not require rounding.
- Alignment of two decimal numbers (for example 1.3 + 27.08) is a simple, exact, shift.
- Conversion to a character form or for display (e.g., to a text-based format such as XML, or to drive signals for a seven-segment display) is a simple per-digit mapping, and can be done in linear (O(n)) time. Conversion from pure binary involves relatively complex logic that spans digits, and for large numbers no linear-time conversion algorithm is known (see Binary numeral system).

Disadvantages

- Some operations are more complex to implement. Adders require extra logic to cause them to wrap and generate a carry early. 15–20 percent more circuitry is needed for BCD add compared to pure binary. Multiplication requires the use of algorithms that are somewhat more complex than shift-mask-add (a binary multiplication, requiring binary shifts and adds or the equivalent, per-digit or group of digits is required)
- Standard BCD requires four bits per digit, roughly 20 percent more space than a binary encoding (the ratio of 4 bits to $\log_2 10$ bits is 1.204). When packed so that three digits are encoded in ten bits, the storage overhead is greatly reduced, at the expense of an encoding that is unaligned with the 8-bit byte boundaries common on existing hardware, resulting in slower implementations on these systems.
- Practical existing implementations of BCD are typically slower than operations on binary representations, especially on embedded systems, due to limited processor support for native BCD operations.

Application

The BIOS in many personal computers stores the date and time in BCD because the MC6818 real-time clock chip used in the original IBM PC AT motherboard provided the time encoded in BCD. This form is easily converted into ASCII for display.[4]

The Atari 8-bit family of computers used BCD to implement floating-point algorithms. The MOS 6502 processor used has a BCD mode that affects the addition and subtraction instructions.

Early models of the PlayStation 3 store the date and time in BCD. This led to a worldwide outage of the console on 1 March 2010. The last two digits of the year stored as BCD were misinterpreted as 16 causing a paradox in the unit's date, rendering most functionalities inoperable.

Representational variations

Various BCD implementations exist that employ other representations for numbers. Programmable calculators manufactured by Texas Instruments, Hewlett-Packard, and others typically employ a floating-point BCD format, typically with two or three digits for the (decimal) exponent. The extra bits of the sign digit may be used to indicate special numeric values, such as infinity, underflow/overflow, and error (a blinking display).

Signed variations

Signed decimal values may be represented in several ways. The COBOL programming language, for example, supports a total of five zoned decimal formats, each one encoding the numeric sign in a different way:

Type	Description	Example
Unsigned	No sign nibble	`F1 F2 F3`
Signed trailing *(canonical format)*	Sign nibble in the last (least significant) byte	`F1 F2 C3`
Signed leading *(overpunch)*	Sign nibble in the first (most significant) byte	`C1 F2 F3`
Signed trailing separate	Separate sign character byte (`'+'` or `'−'`) following the digit bytes	`F1 F2 F3 2B`
Signed leading separate	Separate sign character byte (`'+'` or `'−'`) preceding the digit bytes	`2B F1 F2 F3`

Telephony Binary Coded Decimal (TBCD)

GSM developed **TBCD**, an expansion to BCD where the remaining (unused) bit combinations are used to add specific telephony characters.[5] It is backward compatible to BCD.

Decimal Digit	BCD 8 4 2 1
*	1 0 1 0
#	1 0 1 1
a	1 1 0 0
b	1 1 0 1
c	1 1 1 0
Used as filler when there is an odd number of digits	1 1 1 1

Alternative encodings

If errors in representation and computation are more important than the speed of conversion to and from display, a scaled binary representation may be used, which stores a decimal number as a binary-encoded integer and a binary-encoded signed decimal exponent. For example, 0.2 can be represented as 2×10^{-1}.

This representation allows rapid multiplication and division, but may require shifting by a power of 10 during addition and subtraction to align the decimal points. It is appropriate for applications with a fixed number of decimal places that do not then require this adjustment— particularly financial applications where 2 or 4 digits after the decimal point are usually enough. Indeed this is almost a form of fixed point arithmetic since the position of the radix point is implied.

Chen-Ho encoding provides a boolean transformation for converting groups of three BCD-encoded digits to and from 10-bit values that can be efficiently encoded in hardware with only 2 or 3 gate delays. Densely Packed Decimal is a similar scheme that is used for most of the significand, except the lead digit, for one of the two alternative decimal encodings specified in the IEEE 754-2008 standard.

References

[1] "General Decimal Arithmetic" (http://speleotrove.com/decimal/). .
[2] "ILE RPG Reference" (http://publib.boulder.ibm.com/iseries/v5r2/ic2924/books/c0925083170.htm). .
[3] IBM BM 1401/1440/1460/1410/7010 Character Code Chart in BCD Order (http://ed-thelen.org/1401Project/Van1401-CodeChart.pdf)
[4] http://www.se.ecu.edu.au/units/ens1242/lectures/ens_Notes_08.pdf
[5] "Signalling Protocols and Switching (SPS) Guidelines for using Abstract Syntax Notation One (ASN.1) in telecommunication application protocols" (http://www.etsi.org/deliver/etsi_etr/001_099/060/02_60/etr_060e02p.pdf). .

Further reading

- Mackenzie, Charles E. (1980). *Coded Character Sets: History and Development*. Addison-Wesley. ISBN 0-201-14460-3.
- *Arithmetic Operations in Digital Computers*, R. K. Richards, 397pp, D. Van Nostrand Co., NY, 1955
- Schmid, Hermann, *Decimal computation*, ISBN 0-471-76180-X, 266pp, Wiley, 1974
- *Superoptimizer: A Look at the Smallest Program*, Henry Massalin, ACM Sigplan Notices, Vol. 22 #10 (Proceedings of the Second International Conference on Architectural support for Programming Languages and Operating Systems), pp122–126, ACM, also IEEE Computer Society Press #87CH2440-6, October 1987
- *VLSI designs for redundant binary-coded decimal addition*, Behrooz Shirazi, David Y. Y. Yun, and Chang N. Zhang, IEEE Seventh Annual International Phoenix Conference on Computers and Communications, 1988,

pp52–56, IEEE, March 1988
- *Fundamentals of Digital Logic* by Brown and Vranesic, 2003
- *Modified Carry Look Ahead BCD Adder With CMOS and Reversible Logic Implementation*, Himanshu Thapliyal and Hamid R. Arabnia, Proceedings of the 2006 International Conference on Computer Design (CDES'06), ISBN 1-60132-009-4, pp64–69, CSREA Press, November 2006
- *Reversible Implementation of Densely-Packed-Decimal Converter to and from Binary-Coded-Decimal Format Using in IEEE-754R*, A. Kaivani, A. Zaker Alhosseini, S. Gorgin, and M. Fazlali, 9th International Conference on Information Technology (ICIT'06), pp273–276, IEEE, December 2006.
- Decimal Arithmetic Bibliography (http://speleotrove.com/decimal/decbibindex.html)

External links

- IBM: Chen-Ho encoding (http://speleotrove.com/decimal/chen-ho.html)
- IBM: Densely Packed Decimal (http://speleotrove.com/decimal/DPDecimal.html).
- Convert BCD to decimal, binary and hexadecimal and vice versa (http://www.unitjuggler.com/convert-numbersystems-from-decimal-to-bcd.html)
- BCD for Java (https://code.google.com/p/bcd4j/)

Gray code

Gray code by bit width

2-bit	4-bit
00	0000
01	0001
11	0011
10	0010
	0110
3-bit	0111
	0101
000	0100
001	1100
011	1101
010	1111
110	1110
111	1010
101	1011
100	1001
	1000

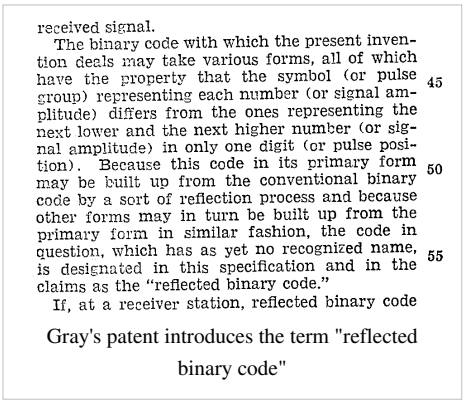
The **reflected binary code**, also known as **Gray code** after Frank Gray, is a binary numeral system where two successive values differ in only one bit. It is a non-weighted code.

The reflected binary code was originally designed to prevent spurious output from electromechanical switches. Today, Gray codes are widely used to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems.

Name

Bell Labs researcher Frank Gray introduced the term *reflected binary code* in his 1947 patent application, remarking that the code had "as yet no recognized name".[1] He derived the name from the fact that it "may be built up from the conventional binary code by a sort of reflection process".

The code was later named after Gray by others who used it. Two different 1953 patent applications give "Gray code" as an alternative name for the "reflected binary code";[2][3] one of those also lists "minimum error code" and "cyclic permutation code" among the names.[3] A 1954 patent application refers to "the Bell Telephone Gray code".[4]



received signal.
The binary code with which the present invention deals may take various forms, all of which have the property that the symbol (or pulse group) representing each number (or signal amplitude) differs from the ones representing the next lower and the next higher number (or signal amplitude) in only one digit (or pulse position). Because this code in its primary form may be built up from the conventional binary code by a sort of reflection process and because other forms may in turn be built up from the primary form in similar fashion, the code in question, which has as yet no recognized name, is designated in this specification and in the claims as the "reflected binary code."
If, at a receiver station, reflected binary code

Gray's patent introduces the term "reflected binary code"

Motivation

Many devices indicate position by closing and opening switches. If that device uses natural binary codes, these two positions would be right next to each other:

```
...
011
100
...
```

The problem with natural binary codes is that, with real (mechanical) switches, it is very unlikely that switches will change states exactly in synchrony. In the transition between the two states shown above, all three switches change state. In the brief period while all are changing, the switches will read some spurious position. Even without keybounce, the transition might look like 011 — 001 — 101 — 100. When the switches appear to be in position 001, the observer cannot tell if that is the "real" position 001, or a transitional state between two other positions. If the output feeds into a sequential system (possibly via combinational logic) then the sequential system may store a false value.

The reflected binary code solves this problem by changing only one switch at a time, so there is never any ambiguity of position,

Dec	Gray	Binary
0	000	000
1	001	001
2	011	010
3	010	011
4	110	100
5	111	101
6	101	110
7	100	111

Notice that state 7 can roll over to state 0 with only one switch change. This is called the "cyclic" property of a Gray code. In the standard Gray coding the least significant bit follows a repetitive pattern of 2 on, 2 off (… 11001100 …); the next digit a pattern of 4 on, 4 off; and so forth.

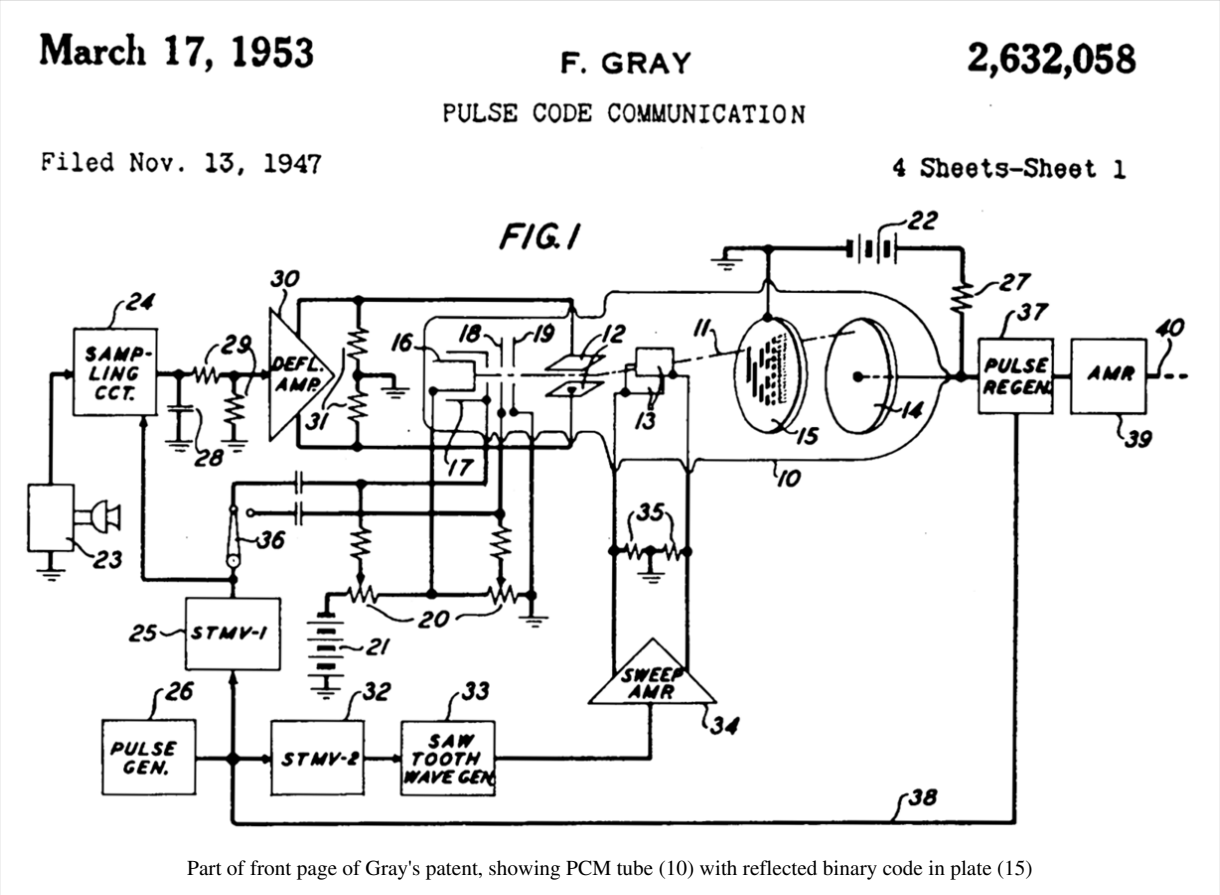
More formally, a **Gray code** is a code assigning to each of a contiguous set of integers, or to each member of a circular list, a word of symbols such that each two adjacent code words differ by one symbol. These codes are also

known as *single-distance codes*, reflecting the Hamming distance of 1 between adjacent codes. There can be more than one Gray code for a given word length, but the term was first applied to a particular binary code for the non-negative integers, the *binary-reflected Gray code*, or **BRGC**, the three-bit version of which is shown above.

History and practical application

Reflected binary codes were applied to mathematical puzzles before they became known to engineers. The French engineer Émile Baudot used Gray codes in telegraphy in 1878. He received the French Legion of Honor medal for his work. The Gray code is sometimes attributed, incorrectly,[5] to Elisha Gray (in *Principles of Pulse Code Modulation*, K. W. Cattermole,[6] for example).

Frank Gray, who became famous for inventing the signaling method that came to be used for compatible color television, invented a method to convert analog signals to reflected binary code groups using vacuum tube-based apparatus. The method and apparatus were patented in 1953 and the name of Gray stuck to the codes. The "PCM tube" apparatus that Gray patented was made by Raymond W. Sears of Bell Labs, working with Gray and William M. Goodall, who credited Gray for the idea of the reflected binary code.[7]

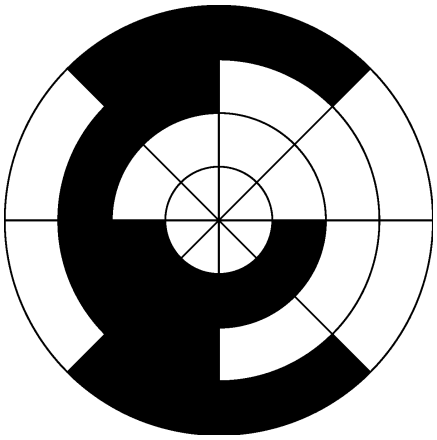


Part of front page of Gray's patent, showing PCM tube (10) with reflected binary code in plate (15)

The use of his eponymous codes that Gray was most interested in was to minimize the effect of error in the conversion of analog signals to digital; his codes are still used today for this purpose, and others.

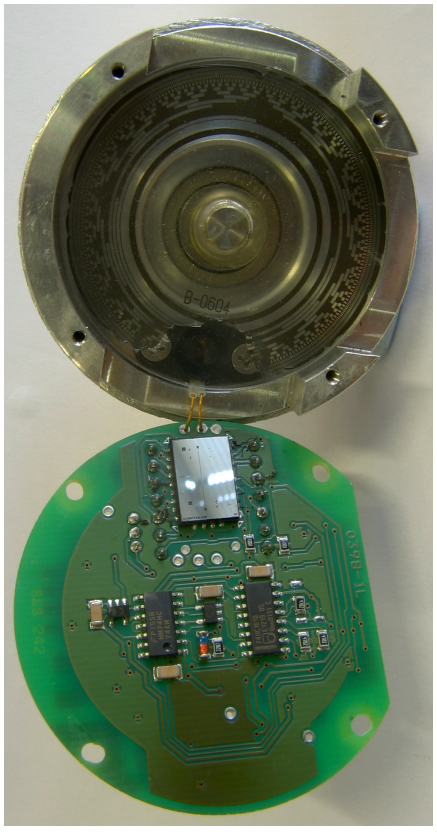
Position encoders

Gray codes are used in position encoders (linear encoders and rotary encoders), in preference to straightforward binary encoding. This avoids the possibility that, when several bits change in the binary representation of an angle, a misread will result from some of the bits changing before others. Originally, the code pattern was electrically conductive, supported (in a rotary encoder) by an insulating disk. Each track had its own stationary metal spring contact; one more contact made the connection to the pattern. That common contact was connected by the pattern to whichever of the track contacts were resting on the conductive pattern. However, sliding contacts wear out and need maintenance, which favors optical encoders.



Rotary encoder for angle-measuring devices marked in 3-bit binary-reflected Gray code (BRGC)

Regardless of the care in aligning the contacts, and accuracy of the pattern, a natural-binary code would have errors at specific disk positions, because it is impossible to make all bits change at exactly the same time as the disk rotates. The same is true of an optical encoder; transitions between opaque and transparent cannot be made to happen simultaneously for certain exact positions. Rotary encoders benefit from the cyclic nature of Gray codes, because consecutive positions of the sequence differ by only one bit. This means that, for a transition from state A to state B, timing mismatches can only affect when the A→B transition occurs, rather than inserting one or more (up to N-1 for an N-bit codeword) false intermediate states, as would occur if a standard binary code were used.



A Gray code absolute rotary encoder with 13 tracks. At the top can be seen the housing, interrupter disk, and light source; at the bottom can be seen the sensing element and support components.

Towers of Hanoi

The binary-reflected Gray code can also be used to serve as a solution guide for the Towers of Hanoi problem, as well as the classical Chinese rings puzzle, a sequential mechanical puzzle mechanism.[5] It also forms a Hamiltonian cycle on a hypercube, where each bit is seen as one dimension.

Genetic algorithms

Due to the Hamming distance properties of Gray codes, they are sometimes used in genetic algorithms. They are very useful in this field, since mutations in the code allow for mostly incremental changes, but occasionally a single bit-change can cause a big leap and lead to new properties.

Karnaugh maps

Gray codes are also used in labelling the axes of Karnaugh maps.[8]

Error correction

In modern digital communications, Gray codes play an important role in error correction. For example, in a digital modulation scheme such as QAM where data is typically transmitted in symbols of 4 bits or more, the signal's constellation diagram is arranged so that the bit patterns conveyed by adjacent constellation points differ by only one bit. By combining this with forward error correction capable of correcting single-bit errors, it is possible for a receiver to correct any transmission errors that cause a constellation point to deviate into the area of an adjacent point. This makes the transmission system less susceptible to noise.

Communication between clock domains

Digital logic designers use Gray codes extensively for passing multi-bit count information between synchronous logic that operates at different clock frequencies. The logic is considered operating in different "clock domains". It is fundamental to the design of large chips that operate with many different clocking frequencies.

Gray code counters and arithmetic

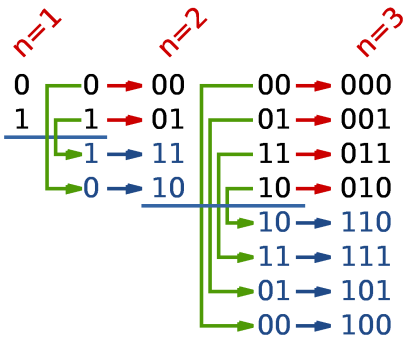
A typical use of Gray code counters is building a FIFO (first-in, first-out) data buffer that has read and write ports that exist in different clock domains. The input and output counters inside such a dual-port FIFO are often stored using Gray code to prevent invalid transient states from being captured when the count crosses clock domains.[9] The updated read and write pointers need to be passed between clock domains when they change, to be able to track FIFO empty and full status in each domain. Each bit of the pointers is sampled non-deterministically for this clock domain transfer. So for each bit, either the old value or the new value is propagated. Therefore, if more than one bit in the multi-bit pointer is changing at the sampling point, a "wrong" binary value (neither new nor old) can be propagated. By guaranteeing only one bit can be changing, Gray codes guarantee that the only possible sampled values are the new or old multi-bit value. Typically Gray codes of power-of-two length are used.

Sometimes digital buses in electronic systems are used to convey quantities that can only increase or decrease by one at a time, for example the output of an event counter which is being passed between clock domains or to a digital-to-analog converter. The advantage of Gray codes in these applications is that differences in the propagation delays of the many wires that represent the bits of the code cannot cause the received value to go through states that are out of the Gray code sequence. This is similar to the advantage of Gray codes in the construction of mechanical encoders, however the source of the Gray code is an electronic counter in this case. The counter itself must count in Gray code, or if the counter runs in binary then the output value from the counter must be reclocked after it has been converted to Gray code, because when a value is converted from binary to Gray code, it is possible that differences in the arrival times of the binary data bits into the binary-to-Gray conversion circuit will mean that the code could go briefly through states that are wildly out of sequence. Adding a clocked register after the circuit that converts the count value to Gray code may introduce a clock cycle of latency, so counting directly in Gray code may be advantageous. A Gray code counter was patented in 1962 US3020481 [10], and there have been many others since. In recent times a Gray code counter can be implemented as a state machine in Verilog. In order to produce the next count value, it is necessary to have some combinational logic that will increment the current count value that is stored in Gray code. Probably the most obvious way to increment a Gray code number is to convert it into ordinary binary code, add one to it with a standard binary adder, and then convert the result back to Gray code. This approach was discussed in a paper in 1996 [11] and then subsequently patented by someone else in 1998 US5754614 [12]. Other methods of counting in Gray code are discussed in a report by R. W. Doran, including taking the output from the first latches of the master-slave flip flops in a binary ripple counter.[13]

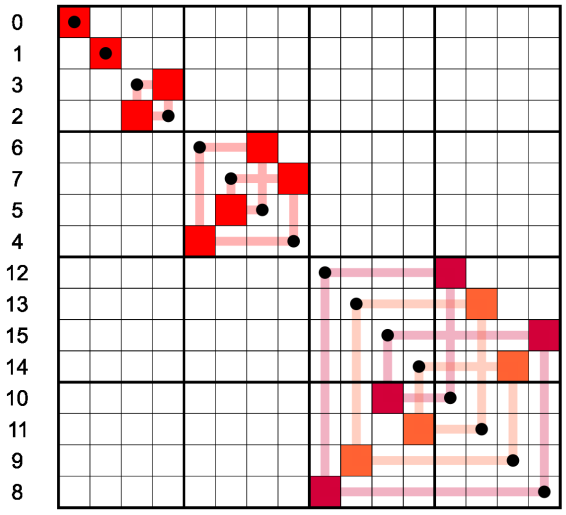
Perhaps the most common electronic counter with the "only one bit changes at a time" property is the Johnson counter.

Constructing an *n*-bit Gray code

The binary-reflected Gray code list for *n* bits can be generated recursively from the list for *n*−*1* bits by reflecting the list (i.e. listing the entries in reverse order), concatenating the original list with the reversed list, prefixing the entries in the original list with a binary 0, and then prefixing the entries in the reflected list with a binary 1. For example, generating the *n* = 3 list from the *n* = 2 list:



The first few steps of the reflect-and-prefix method.



4-bit Gray code permutation

2-bit list:	00, 01, 11, 10	
Reflected:		10, 11, 01, 00
Prefix old entries with *0*:	000, 001, 011, 010,	
Prefix new entries with *1*:		110, 111, 101, 100
Concatenated:	000, 001, 011, 010,	110, 111, 101, 100

The one-bit Gray code is G_1 = (0, 1). This can be thought of as built recursively as above from a zero-bit Gray code G_0 = { Λ } consisting of a single entry of zero length. This iterative process of generating G_{n+1} from G_n makes the following properties of the standard reflecting code clear:

- G_n is a permutation of the numbers 0, ..., 2^n−1. (Each number appears exactly once in the list.)
- G_n is embedded as the first half of G_{n+1}.
- Therefore the coding is *stable*, in the sense that once a binary number appears in G_n it appears in the same position in all longer lists; so it makes sense to talk about *the* reflective Gray code value of a number: $G(m)$ = the *m*-th reflecting Gray code, counting from 0.
- Each entry in G_n differs by only one bit from the previous entry. (The Hamming distance is 1.)

- The last entry in G_n differs by only one bit from the first entry. (The code is cyclic.)

These characteristics suggest a simple and fast method of translating a binary value into the corresponding Gray code. Each bit is inverted if the next higher bit of the input value is set to one. This can be performed in parallel by a bit-shift and exclusive-or operation if they are available: the *n*th Gray code is obtained by computing $n \oplus \lfloor n/2 \rfloor$

A similar method can be used to perform the reverse translation, but the computation of each bit depends on the computed value of the next higher bit so it cannot be performed in parallel. Assuming g_i is the i th gray-coded bit ($g_0$being the most significant bit), and b_i is the i th binary-coded bit ($b_0$being the most-significant bit), the reverse translation can be given recursively: $b_0 = g_0$, and $b_i = g_i \oplus b_{i-1}$. Alternatively, decoding a Gray code into a binary number can be described as a prefix sum of the bits in the Gray code, where each individual summation operation in the prefix sum is performed modulo two.

To construct the binary-reflected Gray code iteratively, start with the code 0, and at step *i* find the bit position of the least significant '1' in the binary representation of *i* - flip the bit at that position in the previous code to get the next code. The bit positions start 0, 1, 0, 2, 0, 1, 0, 3, ... (sequence A007814 in OEIS). See find first set for efficient algorithms to compute these values.

Converting to and from Gray code

The following functions in C convert between binary numbers and their associated Gray codes.

```
/*
        The purpose of this function is to convert an unsigned
        binary number to reflected binary Gray code.

        The operator >> is shift right. The operator ^ is exclusive or.
*/
unsigned int binaryToGray(unsigned int num)
{
        return (num >> 1) ^ num;
}

/*
        The purpose of this function is to convert a reflected binary
        Gray code number to a binary number.
*/
unsigned int grayToBinary(unsigned int num)
{
    unsigned int numBits = 8 * sizeof(num);
    unsigned int shift;
    for (shift = 1; shift < numBits; shift = 2 * shift)
    {
        num = num ^ (num >> shift);
    }
    return num;
}
```

Special types of Gray codes

In practice, a "Gray code" almost always refers to a binary-reflected Gray code (BRGC). However, mathematicians have discovered other kinds of Gray codes. Like BRGCs, each consists of a lists of words, where each word differs from the next in only one digit (each word has a Hamming distance of 1 from the next word).

n-ary Gray code

Ternary number → ternary Gray code
0 → 000
1 → 001
2 → 002
10 → 012
11 → 010
12 → 011
20 → 021
21 → 022
22 → 020
100 → 120
101 → 121
102 → 122
110 → 102
111 → 100
112 → 101
120 → 111
121 → 112
122 → 110
200 → 210
201 → 211
202 → 212
210 → 222
211 → 220
212 → 221
220 → 201
221 → 202
222 → 200

There are many specialized types of Gray codes other than the binary-reflected Gray code. One such type of Gray code is the ***n*-ary Gray code**, also known as a **non-Boolean Gray code**. As the name implies, this type of Gray code uses non-Boolean values in its encodings.

For example, a 3-ary (ternary) Gray code would use the values {0, 1, 2}. The (*n*, *k*)-*Gray code* is the *n*-ary Gray code with *k* digits.[14] The sequence of elements in the (3, 2)-Gray code is: {00, 01, 02, 12, 10, 11, 21, 22, 20}. The (*n*, *k*)-Gray code may be constructed recursively, as the BRGC, or may be constructed iteratively. An algorithm to iteratively generate the (*N*, *k*)-Gray code is presented (in C):

```
// inputs: base, digits, value
// output: gray
// Convert a value to a graycode with the given base and digits.
// Iterating through a sequence of values would result in a sequence
// of Gray codes in which only one digit changes at a time.
void to_gray(unsigned base, unsigned digits, unsigned value, unsigned
gray[digits])
{
      unsigned baseN[digits];      // Stores the ordinary base-N
```

```
number, one digit per entry
      unsigned i;             // The loop variable

      // Put the normal baseN number into the baseN array. For base 10,
 109
      // would be stored as [9,0,1]
      for (i = 0; i < digits; i++) {
            baseN[i] = value % base;
            value    = value / base;
      }

      // Convert the normal baseN number into the graycode equivalent.
Note that
      // the loop starts at the most significant digit and goes down.
      unsigned shift = 0;
      while (i--) {
            // The gray digit gets shifted down by the sum of the
higher
            // digits.
            gray[i] = (baseN[i] + shift) % base;
            shift = shift + base - gray[i];      // Subtract from base
so shift is positive
      }
}
// EXAMPLES
// input: value = 1899, base = 10, digits = 4
// output: baseN[] = [9,9,8,1], gray[] = [0,1,7,1]
// input: value = 1900, base = 10, digits = 4
// output: baseN[] = [0,0,9,1], gray[] = [0,1,8,1]
```

There are other graycode algorithms for *(n,k)*-Gray codes. It is important to note that the *(n,k)*-Gray codes produced by the above algorithm is always cyclical; some algorithms, such as that by Guan,[14] lack this property when k is odd. On the other hand, while only one digit at a time changes with this method, it can change by wrapping (looping from n-1 to 0). In Guan's algorithm, the count alternately rises and falls, so that the numeric difference between two graycode digits is always one.

Gray codes are not uniquely defined, because a permutation of the columns of such a code is a Gray code too. The above procedure produces a code in which the lower the significance of a digit, the more often it changes, making it similar to normal counting methods.

Balanced Gray code

Although the binary reflected Gray code is useful in many scenarios, it is not optimal in certain cases because of a lack of "uniformity".[15] In *balanced Gray codes*, the number of changes in different coordinate positions are as close as possible. To make this more precise, let *G* be a *R*-ary complete Gray cycle having transition sequence (δ_k); the *transition counts (spectrum) of* G are the collection of integers defined by

$$\lambda_k = \left|\{j \in \mathbb{Z}_{R^n} : \delta_j = k\}\right|, \text{ for } k \in \mathbb{Z}_R$$

A Gray code is *uniform* or *uniformly balanced* if its transition counts are all equal, in which case we have $\lambda_k = R^n/n$ for all *k*. Clearly, when $R = 2$, such codes exist only if *n* is a power of 2. Otherwise, if *n* does not divide R^n evenly, it is possible to construct *well-balanced* codes where every transition count is either $\lfloor R^n/n \rfloor$ or $\lceil R^n/n \rceil$. Gray codes can also be *exponentially balanced* if all of their transition counts are adjacent powers of two, and such codes exist for every power of two.[16]

We will now show a construction for well-balanced binary Gray codes which allows us to generate a *n*-digit balanced Gray code for every *n*.[17] The main principle is to inductively construct a $(n+2)$-digit Gray code G' given an *n*-digit Gray code *G* in such a way that the balanced property is preserved. To do this, we consider partitions of $G = g_0, \cdots, g_{2^n-1}$ into an even number *L* of non-empty blocks of the form

$$\{g_0\}, \{g_1, \cdots, g_{k_2}\}, \{g_{k_2+1}, \cdots, g_{k_3}\}, \cdots, \{g_{k_{L-2}+1}, \cdots, g_{-2}\}, \{g_{-1}\}$$

where $k_1 = 0, k_{L-1} = -2$, and $k_L = -1 \pmod{2^n}$. This partition induces a $(n+2)$-digit Gray code given by

$$00g_0,$$
$$00g_1, \cdots, 00g_{k_2}, 01g_{k_2}, \cdots, 01g_1, 11g_1, \cdots, 11g_{k_2},$$
$$11g_{k_2+1}, \cdots, 11g_{k_3}, 01g_{k_3}, \cdots, 01g_{k_2+1}, 00g_{k_2+1}, \cdots, 00g_{k_3}, \cdots,$$
$$00g_{-2}, 00g_{-1}, 10g_{-1}, 10g_{-2}, \cdots, 10g_0, 11g_0, 11g_{-1}, 01g_{-1}, 01g_0$$

If we define the *transition multiplicities* $m_i = \left|\{j : \delta_{k_j} = i, 1 \le j \le L\}\right|$ to be the number of times the digit in position *i* changes between consecutive blocks in a partition, then for the $(n+2)$-digit Gray code induced by this partition the transition spectrum λ'_k is

$$\lambda'_k = \begin{cases} 4\lambda_k - 2m_k, & \text{if } 0 \le k < n \\ L, & \text{otherwise} \end{cases}$$

The delicate part of this construction is to find an adequate partitioning of a balanced *n*-digit Gray code such that the code induced by it remains balanced. Uniform codes can be found when $R \equiv 0 \mod 4$ and $R^n \equiv 0 \mod n$, and this construction can be extended to the *R*-ary case as well.[17]

Monotonic Gray codes

Monotonic codes are useful in the theory of interconnection networks, especially for minimizing dilation for linear arrays of processors.[18] If we define the *weight* of a binary string to be the number of 1's in the string, then although we clearly cannot have a Gray code with strictly increasing weight, we may want to approximate this by having the code run through two adjacent weights before reaching the next one.

We can formalize the concept of monotone Gray codes as follows: consider the partition of the hypercube $Q_n = (V_n, E_n)$ into *levels* of vertices that have equal weight, i.e.

$$V_n(i) = \{v \in V_n : v \text{ has weight } i\}$$

for $0 \le i \le n$. These levels satisfy $|V_n(i)| = \binom{n}{i}$. Let $Q_n(i)$ be the subgraph of Q_n induced by $V_n(i) \cup V_n(i+1)$, and let $E_n(i)$ be the edges in $Q_n(i)$. A monotonic Gray code is then a Hamiltonian path in Q_n such that whenever $\delta_1 \in E_n(i)$ comes before $\delta_2 \in E_n(j)$ in the path, then $i \le j$.

An elegant construction of monotonic *n*-digit Gray codes for any *n* is based on the idea of recursively building subpaths $P_{n,j}$ of length $2\binom{n}{j}$ having edges in $E_n(j)$.[18] We define $P_{1,0} = (0, 1)$, $P_{n,j} = \emptyset$ whenever $j < 0$ or $j \geq n$, and

$$P_{n+1,j} = 1P_{n,j-1}^{\pi_n}, 0P_{n,j}$$

otherwise. Here, π_n is a suitably defined permutation and P^π refers to the path *P* with its coordinates permuted by π. These paths give rise to two monotonic *n*-digit Gray codes $G_n^{(1)}$ and $G_n^{(2)}$ given by

$$G_n^{(1)} = P_{n,0}P_{n,1}^R P_{n,2}P_{n,3}^R \cdots \text{ and } G_n^{(2)} = P_{n,0}^R P_{n,1}P_{n,2}^R P_{n,3} \cdots$$

The choice of π_n which ensures that these codes are indeed Gray codes turns out to be $\pi_n = E^{-1}(\pi_{n-1}^2)$. The first few values of $P_{n,j}$ are shown in the table below.

Subpaths in the Savage-Winkler algorithm

$P_{n,j}$	*j* = 0	*j* = 1	*j* = 2	*j* = 3
***n* = 1**	0, 1			
***n* = 2**	00, 01	10, 11		
***n* = 3**	000, 001	100, 110, 010, 011	101, 111	
***n* = 4**	0000, 0001	1000, 1100, 0100, 0110, 0010, 0011	1010, 1011, 1001, 1101, 0101, 0111	1110, 1111

These monotonic Gray codes can be efficiently implemented in such a way that each subsequent element can be generated in *O*(*n*) time. The algorithm is most easily described using coroutines.

Monotonic codes have an interesting connection to the Lovász conjecture, which states that every connected vertex-transitive graph contains a Hamiltonian path. The "middle-level" subgraph $Q_{2n+1}(n)$ is vertex-transitive (that is, its automorphism group is transitive, so that each vertex has the same "local environment"" and cannot be differentiated from the others, since we can relabel the coordinates as well as the binary digits to obtain an automorphism) and the problem of finding a Hamiltonian path in this subgraph is called the "middle-levels problem", which can provide insights into the more general conjecture. The question has been answered affirmatively for $n \leq 15$, and the preceding construction for monotonic codes ensures a Hamiltonian path of length at least 0.839*N* where *N* is the number of vertices in the middle-level subgraph.[19]

Beckett–Gray code

Another type of Gray code, the **Beckett–Gray code**, is named for Irish playwright Samuel Beckett, who was interested in symmetry. His play "Quad" features four actors and is divided into sixteen time periods. Each period ends with one of the four actors entering or leaving the stage. The play begins with an empty stage, and Beckett wanted each subset of actors to appear on stage exactly once.[20] Clearly the set of actors currently on stage can be represented by a 4-bit binary Gray code. Beckett, however, placed an additional restriction on the script: he wished the actors to enter and exit so that the actor who had been on stage the longest would always be the one to exit. The actors could then be represented by a first in, first out queue, so that (of the actors onstage) the actor being dequeued is always the one who was enqueued first.[20] Beckett was unable to find a Beckett–Gray code for his play, and indeed, an exhaustive listing of all possible sequences reveals that no such code exists for *n* = 4. It is known today that such codes do exist for *n* = 2, 5, 6, 7, and 8, and do not exist for *n* = 3 or 4. An example of an 8-bit Beckett–Gray code can be found in Knuth's *Art of Computer Programming*.[5] According to Sawada and Wong, the search space for *n* = 6 can be explored in 15 hours, and more than 9,500 solutions for the case *n* = 7 have been found.[21]

Snake-in-the-box codes

Snake-in-the-box codes, or *snakes*, are the sequences of nodes of induced paths in an *n*-dimensional hypercube graph, and coil-in-the-box codes, or *coils*, are the sequences of nodes of induced cycles in a hypercube. Viewed as Gray codes, these sequences have the property of being able to detect any single-bit coding error. Codes of this type were first described by W. H. Kautz in the late 1950s;[22] since then, there has been much research on finding the code with the largest possible number of codewords for a given hypercube dimension.

Single-track Gray code

Yet another kind of Gray code is the **single-track Gray code** (STGC) developed by N. B. Spedding (NZ Patent 264738 - October 28, 1994)[23] and refined by Hiltgen, Paterson and Brandestini in "Single-track Gray codes" (1996).[24] The STGC is a cyclical list of *P* unique binary encodings of length n such that two consecutive words differ in exactly one position, and when the list is examined as a *P x n* matrix, each column is a cyclic shift of the first column.[25]

The name comes from their use with rotary encoders, where a number of tracks are being sensed by contacts, resulting for each in an output of 0 or 1. To reduce noise due to different contacts not switching at exactly the same moment in time, one preferably sets up the tracks so that the data output by the contacts are in Gray code. To get high angular accuracy, one needs lots of contacts; in order to achieve at least 1 degree accuracy, one needs at least 360 distinct positions per revolution, which requires a minimum of 9 bits of data, and thus the same number of contacts.

If all contacts are placed at the same angular position, then 9 tracks are needed to get a standard BRGC with at least 1 degree accuracy. However, if the manufacturer moves a contact to a different angular position (but at the same distance from the center shaft), then the corresponding "ring pattern" needs to be rotated the same angle to give the same output. If the most significant bit (the inner ring in Figure 1) is rotated enough, it exactly matches the next ring out. Since both rings are then identical, the inner ring can be cut out, and the sensor for that ring moved to the remaining, identical ring (but offset at that angle from the other sensor on that ring). Those 2 sensors on a single ring make a quadrature encoder. That reduces the number of tracks for a "1 degree resolution" angular encoder to 8 tracks. Reducing the number of tracks still further can't be done with BRGC.

For many years, Torsten Sillke and other mathematicians believed that it was impossible to encode position on a single track such that consecutive positions differed at only a single sensor, except for the 2-sensor, 1-track quadrature encoder. So for applications where 8 tracks were too bulky, people used single-track incremental encoders (quadrature encoders) or 2-track "quadrature encoder + reference notch" encoders.

N. B. Spedding, however, registered a patent in 1994 with several examples showing that it was possible.[23] Although it is not possible to distinguish 2^n positions with *n* sensors on a single track, it *is* possible to distinguish close to that many. For example, when *n* is itself a power of 2, *n* sensors can distinguish $2^n - 2n$ positions. Hiltgen and Paterson published a paper in 2001 exhibiting a single-track gray code with exactly 360 angular positions, constructed using 9 sensors.[26] Since this number is larger than $2^8 = 256$, more than 8 sensors are required by any code, although a BRGC could distinguish 512 positions with 9 sensors. An STGC for $P = 30$ and $n = 5$ is reproduced here:

```
10000
10100
11100
11110
11010
11000
01000
01010
01110
01111
01101
01100
00100
00101
00111
10111
10110
00110
00010
10010
10011
11011
01011
00011
00001
01001
11001
11101
10101
10001
```

Note that each column is a cyclic shift of the first column, and from any row to the next row only one bit changes.[27] The single-track nature (like a code chain) is useful in the fabrication of these wheels (compared to BRGC), as only one track is needed, thus reducing their cost and size. The Gray code nature is useful (compared to chain codes, also called De Bruijn sequences), as only one sensor will change at any one time, so the uncertainty during a transition

between two discrete states will only be plus or minus one unit of angular measurement the device is capable of resolving.[28]

Notes

[1] F. Gray. *Pulse code communication*, March 17, 1953 (filed Nov. 1947). U.S. Patent 2,632,058 (http://www.google.com/patents?vid=2632058)

[2] J. Breckman. *Encoding Circuit*, Jan 31, 1956 (filed Dec. 1953). U.S. Patent 2,733,432 (http://www.google.com/patents?vid=2733432)

[3] E. A. Ragland et al. *Direction-Sensitive Binary Code Position Control System*, Feb. 11, 1958 (filed Oct. 1953). U.S. Patent 2,823,345 (http://www.google.com/patents?vid=2823345)

[4] S. Reiner et al. *Automatic Rectification System*, Jun 24, 1958 (filed Jan. 1954). U.S. Patent 2,839,974 (http://www.google.com/patents?vid=2839974)

[5] Knuth, Donald E. "Generating all *n*-tuples." *The Art of Computer Programming, Volume 4A: Enumeration and Backtracking*, pre-fascicle 2a, October 15, 2004. (http://www-cs-faculty.stanford.edu/~knuth/fasc2a.ps.gz)

[6] Cattermole, K. W. (1969). *Principles of Pulse Code Modulation*. New York: American Elsevier. ISBN 0-444-19747-8.

[7] Goodall, W. M. (1951). "Television by Pulse Code Modulation". *Bell Sys. Tech. J.* **30**: 33–49.

[8] Wakerly, John F (1994). *Digital Design: Principles & Practices*. New Jersey: Prentice Hall. pp. 222, 48–49. ISBN 0-13-211459-3. Note that the two page sections taken together say that K-maps are labeled with Gray code. The first section says that they are labeled with a code that changes only one bit between entries and the second section says that such a code is called Gray code.

[9] "Synchronization in Digital Logic Circuits (http://www.stanford.edu/class/ee183/handouts_spr2003/synchronization_pres.pdf) by Ryan Donohue

[10] http://www.google.com/patents?vid=3020481

[11] Mehta, H.; Owens, R.M. & Irwin, M.J. (1996), Some issues in gray code addressing (http://ieeexplore.ieee.org/xpls/abs_all.jsp?tp=&arnumber=497616&isnumber=10625), in the Proceedings of the 6th Great Lakes Symposium on VLSI (GLSVLSI 96), IEEE Computer Society,pp. 178

[12] http://www.google.com/patents?vid=5754614

[13] The Gray Code by R. W. Doran (http://www.cs.auckland.ac.nz/CDMTCS//researchreports/304bob.pdf)

[14] Guan, Dah-Jyh (1998). "Generalized Gray Codes with Applications" (http://nr.stpi.org.tw/ejournal/ProceedingA/v22n6/841-848.pdf) (PDF). *Proc. Natl. Sci. Counc. Repub. Of China (A)* **22**: 841–848. .

[15] Girish S. Bhat and Carla D. Savage (1996). "Balanced Gray codes" (http://www.combinatorics.org/Volume_3/Abstracts/v3i1r25.html). *Electronic Journal of Combinatorics* **3** (1): R25. .

[16] I. N Suparta (2005). "A simple proof for the existence of exponentially balanced Gray codes". *Electronic Journal of Combinatorics* **12**.

[17] M. Flahive and B. Bose (2007). "Balancing cyclic R-ary Gray codes". *Electronic Journal of Combinatorics* **14**.

[18] C. D Savage and P. Winkler (1995). "Monotone Gray codes and the middle levels problem". *Journal of Combinatorial Theory, Series A* **70** (2): 230–248. doi:10.1016/0097-3165(95)90091-8. ISSN 0097-3165.

[19] C. D Savage (1997). *Long cycles in the middle two levels of the Boolean lattice*.

[20] Goddyn, Luis (1999). "MATH 343 Applied Discrete Math Supplementary Materials" (http://www.math.sfu.ca/~goddyn/Courses/343/supMaterials.pdf) (PDF). Dept. of Math, Simon Fraser U. .

[21] Wong, J. (2007). "A Fast Algorithm to generate Beckett-Gray codes". *Electronic Notes in Discrete Mathematics* **29**: 571–577. doi:10.1016/j.endm.2007.07.091.

[22] Kautz, W. H. (1958). "Unit-distance error-checking codes". *IRE Trans. Elect. Comput.* **7**: 177–180.

[23] *A position encoder* (http://www.winzurf.co.nz/Single_Track_Grey_Code_Patent/Single_track_Grey_code_encoder_patent.pdf). 1994. .

[24] Hiltgen, Alain P.; Kenneth G. Paterson, Marco Brandestini (1996). "Single-Track Gray Codes" (http://ieeexplore.ieee.org/iel1/18/11236/00532900.pdf) (PDF). *IEEE Transactions on Information Theory* **42** (5): 1555–1561. doi:10.1109/18.532900. .

[25] Etzion, Tuvi; Moshe Schwartz (1999). "The Structure of Single-Track Gray Codes" (http://www.cs.technion.ac.il/~etzion/PUB/Gray2.pdf) (PDF). *IEEE Transactions on Information Theory* **45** (7): 2383–2396. doi:10.1109/18.796379. .

[26] Hiltgen, Alain P.; Kenneth G. Paterson (2001). "Single-Track Circuit Codes" (http://www.hpl.hp.com/techreports/2000/HPL-2000-81.pdf) (PDF). *IEEE Transactions on Information Theory* **47** (6): 2587–2595. doi:10.1109/18.945274. .

[27] "Venn Diagram Survey — Symmetric Diagrams" (http://www.combinatorics.org/Surveys/ds5/VennSymmEJC.html). *Electronic Journal of Combinatorics*. 2001. .

[28] Alciatore, David G.; Michael B. Histand (1999). McGraw-Hill Education - Europe. ISBN 978-0-07-131444-2. http://mechatronics.colostate.edu/.

References

- Black, Paul E. *Gray code* (http://www.nist.gov/dads/HTML/graycode.html). 25 February 2004. NIST.
- Press, WH; Teukolsky, SA; Vetterling, WT; Flannery, BP (2007). "Section 22.3. Gray Codes" (http://apps.nrbook.com/empanel/index.html#pg=1166). *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). New York: Cambridge University Press. ISBN 978-0-521-88068-8.
- Savage, Carla (1997). "A Survey of Combinatorial Gray Codes" (http://www.csc.ncsu.edu/faculty/savage/AVAILABLE_FOR_MAILING/survey.ps). *SIAM Rev.* **39** (4): 605–629. doi:10.1137/S0036144595295272. JSTOR 2132693.
- Wilf, Herbert S. (1989). "Chapters 1-3". *Combinatorial algorithms: an update*. SIAM. ISBN 0-89871-231-9.

External links

- "Gray Code" demonstration (http://demonstrations.wolfram.com/BinaryGrayCode/) by Michael Schreiber, Wolfram Demonstrations Project (with Mathematica implementation). 2007.
- NIST Dictionary of Algorithms and Data Structures: Gray code (http://www.nist.gov/dads/HTML/graycode.html)
- Hitch Hiker's Guide to Evolutionary Computation, Q21: What are Gray codes, and why are they used? (http://www.aip.de/~ast/EvolCompFAQ/Q21.htm), including C code to convert between binary and BRGC
- Subsets or Combinations (http://www.theory.cs.uvic.ca/~cos/gen/comb.html) Can generate BRGC strings
- "The Structure of Single-Track Gray Codes" (http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi?1998/CS/CS0937) by Moshe Schwartz, Tuvi Etzion
- Single-Track Circuit Codes (http://www.hpl.hp.com/techreports/2000/HPL-2000-81.html) by Hiltgen, Alain P.; Paterson, Kenneth G.
- Dragos A. Harabor uses Gray codes in a 3D digitizer (http://www.ugcs.caltech.edu/~dragos/3DP/coord.html).
- single-track gray codes, binary chain codes (Lancaster 1994 (http://tinaja.com/text/chain01.html)), and linear feedback shift registers are all useful in finding one's absolute position on a single-track rotary encoder (or other position sensor).
- Computing Binary Combinatorial Gray Codes Via Exhaustive Search With SAT Solvers (http://ieeexplore.ieee.org/xpls/abs_all.jsp?isnumber=4475352&arnumber=4475394&count=44&index=39) by Zinovik, I.; Kroening, D.; Chebiryak, Y.
- AMS Column: Gray codes (http://www.ams.org/featurecolumn/archive/gray.html)
- Optical Encoder Wheel Generator (http://www.bushytails.net/~randyg/encoder/encoderwheel.html)
- ProtoTalk.net - Understanding Quadrature Encoding (http://prototalk.net/forums/showthread.php?t=78) - Covers quadrature encoding in more detail with a focus on robotic applications

Hexadecimal

In mathematics and computer science, **hexadecimal** (also **base 16**, or **hex**) is a positional numeral system with a radix, or base, of 16. It uses sixteen distinct symbols, most often the symbols **0–9** to represent values zero to nine, and **A**, **B**, **C**, **D**, **E**, **F** (or alternatively **a–f**) to represent values ten to fifteen. For example, the hexadecimal number 2AF3 is equal, in decimal, to $(2 \times 16^3) + (10 \times 16^2) + (15 \times 16^1) + (3 \times 16^0)$, or 10995.

Each hexadecimal digit represents four binary digits (bits), and the primary use of hexadecimal notation is a human-friendly representation of binary-coded values in computing and digital electronics. One hexadecimal digit represents a nibble, which is half of an octet or byte (8 bits). For example, byte values can range from 0 to 255 (decimal), but may be more conveniently represented as two hexadecimal digits in the range 00 to FF. Hexadecimal is also commonly used to represent computer memory addresses.

Representation

Written representation

Using 0-9 and A-F

0_{hex}	=	0_{dec}	=	0_{oct}	0	0	0	0
1_{hex}	=	1_{dec}	=	1_{oct}	0	0	0	1
2_{hex}	=	2_{dec}	=	2_{oct}	0	0	1	0
3_{hex}	=	3_{dec}	=	3_{oct}	0	0	1	1
4_{hex}	=	4_{dec}	=	4_{oct}	0	1	0	0
5_{hex}	=	5_{dec}	=	5_{oct}	0	1	0	1
6_{hex}	=	6_{dec}	=	6_{oct}	0	1	1	0
7_{hex}	=	7_{dec}	=	7_{oct}	0	1	1	1
8_{hex}	=	8_{dec}	=	10_{oct}	1	0	0	0
9_{hex}	=	9_{dec}	=	11_{oct}	1	0	0	1
A_{hex}	=	10_{dec}	=	12_{oct}	1	0	1	0
B_{hex}	=	11_{dec}	=	13_{oct}	1	0	1	1
C_{hex}	=	12_{dec}	=	14_{oct}	1	1	0	0
D_{hex}	=	13_{dec}	=	15_{oct}	1	1	0	1
E_{hex}	=	14_{dec}	=	16_{oct}	1	1	1	0
F_{hex}	=	15_{dec}	=	17_{oct}	1	1	1	1

In situations where there is no context, hexadecimal numbers can be ambiguous and confused with numbers expressed in other bases. There are several conventions for expressing values unambiguously. A numerical subscript (itself written in decimal) can give the base explicitly: 159_{10} is decimal 159; 159_{16} is hexadecimal 159, which is equal to 345_{10}. Other authors prefer a text subscript, such as $159_{decimal}$ and 159_{hex}, or 159_{d} and 159_{h}.

In linear text systems, such as those used in most computer programming environments, a variety of methods have arisen:

- In URIs (including URLs), character codes are written as hexadecimal pairs prefixed with `%`: `http://www.example.com/name%20with%20spaces` where `%20` is the space (blank) character (code value 20 in hex, 32 in decimal).
- In XML and XHTML, characters can be expressed as hexadecimal numeric character references using the notation `&#x`*code*`;`, where *code* is the 1- to 6-digit hex number assigned to the character in the Unicode standard. Thus `&#x2019;` represents the curled right single quote (Unicode value 2019 in hex, 8217 in decimal).
- Color references in HTML and CSS and X Window can be expressed with six hexadecimal digits (two each for the red, green, and blue components, in that order) prefixed with `#`: white, for example, is represented `#FFFFFF`.[1] CSS allows 3-hexdigit abbreviations with one hexdigit per component: #FA3 abbreviates #FFAA33 (a golden orange:).
- *nix (Unix and related) shells, AT&T assembly language, and likewise the C programming language, which was designed for Unix (and the syntactic descendants of C[2]) use the prefix `0x` for numeric constants represented in hex: `0x5A3`. Character and string constants may express character codes in hexadecimal with the prefix `\x` followed by two hex digits: `'\x1B'` represents the Esc control character; `"\x1B[0m\x1B[25;1H"` is a string containing 11 characters (plus a trailing NUL to mark the end of the string) with two embedded Esc characters.[3] To output an integer as hexadecimal with the printf function family, the format conversion code `%X` or `%x` is used.
- In the Unicode standard, a character value is represented with `U+` followed by the hex value: `U+20AC` is the Euro sign (€).
- In MIME (e-mail extensions) quoted-printable encoding, characters that cannot be represented as literal ASCII characters are represented by their codes as two hexadecimal digits (in ASCII) prefixed by an *equal to* sign =, as in `Espa=F1a` to send "España" (Spain). (Hexadecimal F1, equal to decimal 241, is the code number for the lower case n with tilde in the ISO/IEC 8859-1 character set.)
- In Intel-derived assembly languages, hexadecimal is denoted with a suffixed `H` or `h`: `FFh` or `05A3H`. Some implementations require a leading zero when the first hexadecimal digit character is not a decimal digit: `0FFh`
- Other assembly languages (6502, Motorola), Pascal, Delphi, some versions of BASIC (Commodore), GML and Forth use `$` as a prefix: `$5A3`.
- Some assembly languages (Microchip) use the notation `H'ABCD'` (for $ABCD_{16}$).
- Ada and VHDL enclose hexadecimal numerals in based "numeric quotes": `16#5A3#`. For bit vector constants VHDL uses the notation `x"5A3"`.[4]
- Verilog represents hexadecimal constants in the form `8'hFF`, where 8 is the number of bits in the value and FF is the hexadecimal constant.
- Modula-2 and some other languages use # as a prefix: `#05A3`
- The Smalltalk language uses the prefix `16r`: `16r5A3`
- PostScript and the Bourne shell and its derivatives denote hex with prefix `16#`: `16#5A3`. For PostScript, binary data (such as image pixels) can be expressed as unprefixed consecutive hexadecimal pairs: `AA213FD51B3801043FBC`...
- In early systems when a Macintosh crashed, one or two lines of hexadecimal code would be displayed under the Sad Mac to tell the user what went wrong.
- Common Lisp uses the prefixes `#x` and `#16r`.
- MSX BASIC,[5] QuickBASIC, FreeBASIC and Visual Basic prefix hexadecimal numbers with `&H`: `&H5A3`
- BBC BASIC and Locomotive BASIC use `&` for hex.[6]
- TI-89 and 92 series uses a `0h` prefix: `0h5A3`
- The most common format for hexadecimal on IBM mainframes (zSeries) and midrange computers (IBM System i) running the traditional OS's (zOS, zVSE, zVM, TPF, IBM i) is `X'5A3'`, and is used in Assembler, PL/I, COBOL, JCL, scripts, commands and other places. This format was common on other (and now obsolete) IBM

systems as well. Occasionally quotation marks were used instead of apostrophes.

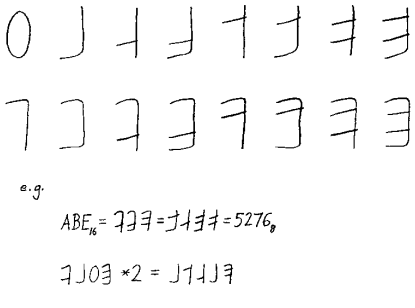
- Donald Knuth introduced the use of a particular typeface to represent a particular radix in his book *The TeXbook*.[7] Hexadecimal representations are written there in a typewriter typeface: `5A3`
- Any IPv6 address can be written as eight groups of four hexadecimal digits, where each group is separated by a colon (`:`). This, for example, is a valid IPv6 address: 2001:0db8:85a3:0000:0000:8a2e:0370:7334
- ALGOL 68 uses the prefix `16r` to denote hexadecimal numbers: `16r5a3`. Binary, quaternary (base-4) and octal numbers can be specified similarly.

There is no universal convention to use lowercase or uppercase for the letter digits, and each is prevalent or preferred in particular environments by community standards or convention.

Early written representations

The choice of the letters *A* through *F* to represent the digits above nine was not universal in the early history of computers.

- During the 1950s, some installations favored using the digits 0 through 5 with a macron character ("¯") to denote the values 10–15.
- Bendix G-15 computers used the letters *U* through *Z*.
- The Librascope LGP-30 used the letters *F*, *G*, *J*, *K*, *Q* and *W*.[8]
- Bruce Alan Martin of Brookhaven National Laboratory considered the choice of A–F "ridiculous" and in a 1968 letter to the editor of the CACM proposed an entirely new set of symbols based on the bit locations, which did not gain much acceptance.[9]



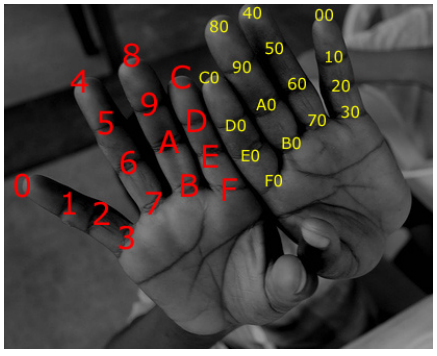
Bruce Alan Martin's hexadecimal notation proposal

- Soviet programmable calculators Б3-34 and similar used the symbols "–", "L", "C", "Г", "E", " " (space) on their displays.

Verbal and digital representations

There are no traditional numerals to represent the quantities from ten to fifteen — letters are used as a substitute — and most European languages lack non-decimal names for the numerals above ten. Even though English has names for several non-decimal powers (*pair* for the first binary power, *score* for the first vigesimal power, *dozen*, *gross*, and *great gross* for the first three duodecimal powers), no English name describes the hexadecimal powers (decimal 16, 256, 4096, 65536, ...). Some people read hexadecimal numbers digit by digit like a phone number: *4DA* is "four-dee-ay". However, the letter *A* sounds like "eight", *C* sounds like "three", and *D* can easily be mistaken for the "-ty" suffix: Is it *4D* or *forty*? Other people avoid confusion by using the NATO phonetic alphabet: *4DA* is "four-delta-alfa", the Joint Army/Navy Phonetic Alphabet ("four-dog-able"), or a similar ad hoc system.

Systems of counting on digits have been devised for both binary and hexadecimal. Arthur C. Clarke suggested using each finger as an on/off bit, allowing finger counting from zero to 1023_{10} on ten fingers. Another system for counting up to FF_{16} (255_{10}) is illustrated on the right; it seems to be an extension of an existing system for counting in twelves (dozens and grosses), that is common in South Asia and elsewhere.



Hexadecimal finger-counting scheme.

Signs

The hexadecimal system can express negative numbers the same way as in decimal: –2A to represent -42_{10} and so on.

However, some prefer instead to use the hexadecimal notation to express the exact bit patterns used in the processor, so a sequence of hexadecimal digits may represent a signed or even a floating point value. This way, the negative number -42_{10} can be written as FFFF FFD6 in a 32-bit CPU register (in two's-complement), as C228 0000 in a 32-bit FPU register or C045 0000 0000 0000 in a 64-bit FPU register (in the IEEE floating-point standard).

Hexadecimal exponential notation

Just as decimal numbers can be represented in exponential notation so too can hexadecimal. By convention, the letter *p* represents *times two raised to the power of*, whereas *e* serves a similar purpose in decimal. The number after the *p* is **decimal** and represents the **binary** exponent.

Usually the number is normalised: that is, the leading hexadecimal digit is 1 (unless the value is exactly 0).

Example: 1.3DEp42 represents $1.3DE_{16} \times 2^{42}$.

Hexadecimal exponential notation is required by the IEEE 754 binary floating-point standard. This notation can be produced by some versions of the *printf* family of functions by using the **%a** conversion.

Conversion

Binary conversion

Most computers manipulate binary data, but it is difficult for humans to work with the large number of digits for even a relatively small binary number. Although most humans are familiar with the base 10 system, it is much easier to map binary to hexadecimal than to decimal because each hexadecimal digit maps to a whole number of bits (4_{10}). This example converts 1111_2 to base ten. Since each position in a binary numeral can contain either a 1 or a 0, its value may be easily determined by its position from the right:

- $0001_2 = 1_{10}$
- $0010_2 = 2_{10}$
- $0100_2 = 4_{10}$
- $1000_2 = 8_{10}$

Therefore:

$$\begin{aligned} 1111_2 &= 8_{10} + 4_{10} + 2_{10} + 1_{10} \\ &= 15_{10} \end{aligned}$$

With little practice, mapping 1111_2 to F_{16} in one step becomes easy: see table in Written representation. The advantage of using hexadecimal rather than decimal increases rapidly with the size of the number. When the number becomes large, conversion to decimal is very tedious. However, when mapping to hexadecimal, it is trivial to regard the binary string as 4-digit groups and map each to a single hexadecimal digit.

This example shows the conversion of a binary number to decimal, mapping each digit to the decimal value, and adding the results.

$$\begin{aligned} 01011110101101010010_2 &= 262144_{10} + 65536_{10} + 32768_{10} + 16384_{10} + 8192_{10} + 2048_{10} + 512_{10} + 256_{10} + 64_{10} + 16_{10} + 2_{10} \\ &= 387922_{10} \end{aligned}$$

Compare this to the conversion to hexadecimal, where each group of four digits can be considered independently, and converted directly:

$$\begin{aligned} 010111101011010100 10_2 &= 0101 \quad 1110 \quad 1011 \quad 0101 \quad 0010_2 \\ &= 5 \quad E \quad B \quad 5 \quad 2_{16} \\ &= 5EB52_{16} \end{aligned}$$

The conversion from hexadecimal to binary is equally direct.

The octal system can also be useful as a tool for people who need to deal directly with binary computer data. Octal represents data as three bits per character, rather than four.

Division-remainder in source base

As with all bases there is a simple algorithm for converting a representation of a number to hexadecimal by doing integer division and remainder operations in the source base. In theory, this is possible from any base, but for most humans only decimal and for most computers only binary (which can be converted by far more efficient methods) can be easily handled with this method.

Let d be the number to represent in hexadecimal, and the series $h_i h_{i-1} ... h_2 h_1$ be the hexadecimal digits representing the number.

1. i := 1
2. h_i := d mod 16
3. d := (d–h_i) / 16
4. If d = 0 (return series h_i) else increment i and go to step 2

"16" may be replaced with any other base that may be desired.

The following is a JavaScript implementation of the above algorithm for converting any number to a hexadecimal in String representation. Its purpose is to illustrate the above algorithm. To work with data seriously, however, it is much more advisable to work with bitwise operators.

```
function toHex(d) {
  var r = d % 16;
  var result;
  if (d-r == 0)
    result = toChar(r);
  else
    result = toHex( (d-r)/16 ) + toChar(r);
  return result;
}

function toChar(n) {
  const alpha = "0123456789ABCDEF";
  return alpha.charAt(n);
}
```

Addition and multiplication

It is also possible to make the conversion by assigning each place in the source base the hexadecimal representation of its place value and then performing multiplication and addition to get the final representation. That is, to convert the number B3AD to decimal one can split the hexadecimal number into its digits: B (11_{10}), 3 (3_{10}), A (10_{10}) and D (13_{10}), and then get the final result by multiplying each decimal representation by 16^p, where p is the corresponding hex digit position, counting from right to left, beginning with 0. In this case we have B3AD = $(11 \times 16^3) + (3 \times 16^2) + (10 \times 16^1) + (13 \times 16^0)$, which is 45997 base 10.

x	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E	20
3	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D	30
4	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C	40
5	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B	50
6	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A	60
7	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69	70
8	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78	80
9	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87	90
A	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96	A0
B	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5	B0
C	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4	C0
D	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3	D0
E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2	E0
F	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1	F0
10	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0	100

A hexadecimal multiplication table

Tools for conversion

Most modern computer systems with graphical user interfaces provide a built-in calculator utility, capable of performing conversions between various radices, in general including hexadecimal.

In Microsoft Windows, the Calculator utility can be set to Scientific mode (called Programmer mode in some versions), which allows conversions between radix 16 (hexadecimal), 10 (decimal), 8 (octal) and 2 (binary), the bases most commonly used by programmers. In Scientific Mode, the on-screen numeric keypad includes the hexadecimal digits A through F, which are active when "Hex" is selected. In hex mode, however, the Windows Calculator supports only integers.

Real numbers

As with other numeral systems, the hexadecimal system can be used to represent rational numbers, although recurring digits are common since sixteen (10h) has only a single prime factor (two):

1/2	**=**	**0.8**	1/6	=	0.2A	1/A	=	0.19	1/E	=	0.1249
1/3	=	0.5	1/7	=	0.249	1/B	=	0.1745D	1/F	=	0.1
1/4	**=**	**0.4**	**1/8**	**=**	**0.2**	1/C	=	0.15	**1/10**	**=**	**0.1**
1/5	=	0.3	1/9	=	0.1C7	1/D	=	0.13B	1/11	=	0.0F

where an overline denotes a recurring pattern.

For any base, 0.1 (or "1/10") is always equivalent to one divided by the representation of that base value in its own number system: Counting in base 3 is 0, 1, 2, 10 (three). Thus, whether dividing one by two for binary or dividing one by sixteen for hexadecimal, both of these fractions are written as `0.1`. Because the radix 16 is a perfect square (4^2), fractions expressed in hexadecimal have an odd period much more often than decimal ones, and there are no cyclic numbers (other than trivial single digits). Recurring digits are exhibited when the denominator in lowest terms has a prime factor not found in the radix; thus, when using hexadecimal notation, all fractions with denominators that are not a power of two result in an infinite string of recurring digits (such as thirds and fifths). This makes hexadecimal (and binary) less convenient than decimal for representing rational numbers since a larger proportion lie outside its range of finite representation.

All rational numbers finitely representable in hexadecimal are also finitely representable in decimal, duodecimal, and sexagesimal: that is, any hexadecimal number with a finite number of digits has a finite number of digits when expressed in those other bases. Conversely, only a fraction of those finitely representable in the latter bases are finitely representable in hexadecimal. For example, decimal 0.1 corresponds to the infinite recurring representation

0.199999999999... in hexadecimal. However, hexadecimal is more efficient than bases 12 and 60 for representing fractions with powers of two in the denominator (e.g., decimal one sixteenth is 0.1 in hexadecimal, 0.09 in duodecimal, 0:3:45 in sexagesimal and 0.0625 in decimal).

In decimal Prime factors of the base: 2, 5			**In hexadecimal** Prime factors of the base: 2		
Fraction	Prime factors of the denominator	Positional representation	Positional representation	Prime factors of the denominator	Fraction
1/2	2	**0.5**	**0.8**	2	1/2
1/3	3	**0.**3333... = **0.**3	**0.**5555... = **0.**5	3	1/3
1/4	2	**0.25**	**0.4**	2	1/4
1/5	5	**0.2**	**0.**3	5	1/5
1/6	2, 3	**0.1**6	**0.2**A	2, 3	1/6
1/7	7	**0.**142857	**0.**249	7	1/7
1/8	2	**0.125**	**0.2**	2	1/8
1/9	3	**0.**1	**0.**1C7	3	1/9
1/10	2, 5	**0.1**	**0.1**9	2, 5	1/A
1/11	11	**0.**09	**0.**1745D	B	1/B
1/12	2, 3	**0.08**3	**0.1**5	2, 3	1/C
1/13	13	**0.**076923	**0.**13B	D	1/D
1/14	2, 7	**0.0**714285	**0.1**249	2, 7	1/E
1/15	3, 5	**0.0**6	**0.**1	3, 5	1/F
1/16	2	**0.0625**	**0.1**	2	1/10
1/17	17	**0.**0588235294117647	**0.**0F	11	1/11
1/18	2, 3	**0.0**5	**0.0**E38	2, 3	1/12
1/19	19	**0.**052631578947368421	**0.**0D79435E5	13	1/13
1/20	2, 5	**0.05**	**0.0**C	2, 5	1/14
1/21	3, 7	**0.**047619	**0.**0C3	3, 7	1/15
1/22	2, 11	**0.0**45	**0.0**BA2E8	2, B	1/16
1/23	23	**0.**04347826086956521739 13	**0.**0B21642C859	17	1/17
1/24	2, 3	**0.041**6	**0.0**A	2, 3	1/18
1/25	5	**0.04**	**0.**0A3D7	5	1/19
1/26	2, 13	**0.0**384615	**0.0**9D8	2, B	1/1A
1/27	3	**0.**037	**0.**097B425ED	3	1/1B
1/28	2, 7	**0.03**571428	**0.0**924	2, 7	1/1C
1/29	29	**0.**03448275862068965517241379 31	**0.**08D3DCB	1D	1/1D
1/30	2, 3, 5	**0.0**3	**0.0**8	2, 3, 5	1/1E
1/31	31	**0.**032258064516129	**0.**08421	1F	1/1F
1/32	2	**0.03125**	**0.08**	2	1/20
1/33	3, 11	**0.**03	**0.**07C1F	3, B	1/21
1/34	2, 17	**0.0**2941176470588235	**0.0**78	2, 11	1/22

1/35	5, 7	**0.0**285714	**0.**075	5, 7	1/23
1/36	2, 3	**0.02**7	**0.0**71C	2, 3	1/24

Algebraic irrational number	In decimal	**In hexadecimal**
√2 (the length of the diagonal of a unit square)	1.41421356237309...	1.6A09E667F3BCD...
√3 (the length of the diagonal of a unit cube)	1.73205080756887...	1.BB67AE8584CAA...
√5 (the length of the diagonal of a 1×2 rectangle)	2.2360679774997...	2.3C6EF372FE95...
φ (phi, the golden ratio = (1+√5)/2	1.6180339887498...	1.9E3779B97F4A...
Transcendental irrational number		
π (pi, the ratio of circumference to diameter)	3.14159265358979323846264338327950288419716939937510...	3.243F6A8885A308D313198A2E03707344A4093822299F31D008...
e (the base of the natural logarithm)	2.7182818284590452...	2.B7E151628AED2A6B...
τ (the Thue–Morse constant)	0.412454033640...	0.6996 9669 9669 6996 ...
γ (the limiting difference between the harmonic series and the natural logarithm)	0.5772156649015328606...	0.93C467E37DB0C7A4D1B...

Powers

Possibly the most widely used powers, powers of two, are easier to show using base 16. The first sixteen powers of two are shown below.

2^x	value
2^0	1
2^1	2
2^2	4
2^3	8
2^4	10_{hex}
2^5	20_{hex}
2^6	40_{hex}
2^7	80_{hex}
2^8	100_{hex}
2^9	200_{hex}
2^A ($2^{10_{dec}}$)	400_{hex}
2^B ($2^{11_{dec}}$)	800_{hex}
2^C ($2^{12_{dec}}$)	1000_{hex}
2^D ($2^{13_{dec}}$)	2000_{hex}
2^E ($2^{14_{dec}}$)	4000_{hex}
2^F ($2^{15_{dec}}$)	8000_{hex}

2^{10} ($2^{16_{dec}}$)	10000_{hex}

Since four squared is sixteen, powers of four have an even easier relation:

4^x	value
4^0	1
4^1	4
4^2	10_{hex}
4^3	40_{hex}
4^4	100_{hex}
4^5	400_{hex}
4^6	1000_{hex}
4^7	4000_{hex}
4^8	10000_{hex}

This also makes tetration easier when using two and four since:
${}^3 2 = 2^4 = 10_{\text{hex}}$,
${}^4 2 = 2^{16} = 10000_{\text{hex}}$ and
${}^5 2 = 2^{65536} = (1 \text{ followed by } 16384 \text{ zeros})_{\text{hex}}$.

Cultural

Etymology

The word *hexadecimal* is composed of *hexa-*, derived from the Greek ἑξ (hex) for "six", and *-decimal*, derived from the Latin for "tenth". Webster's Third New International online derives "hexadecimal" as an alteration of the all-Latin "sexadecimal" (which appears in the earlier Bendix documentation). The earliest date attested for "hexadecimal" in Merriam-Webster Collegiate online is 1954, placing it safely in the category of international scientific vocabulary (ISV). It is common in ISV to mix Greek and Latin combining forms freely. The word "sexagesimal" (for base 60) retains the Latin prefix. Donald Knuth has pointed out that the etymologically correct term is "senidenary", from the Latin term for "grouped by 16". (The terms "binary", "ternary" and "quaternary" are from the same Latin construction, and the etymologically correct term for "decimal" arithmetic is "denary".)[10] Alfred B. Taylor used "senidenary" in his mid 19th century work on alternative number bases, although he rejected base 16 because of its "incommodious number of digits."[11][12] Schwartzman notes that the expected form from usual Latin phrasing would be "sexadecimal", but computer hackers would be tempted to shorten that word to "sex".[13] The etymologically proper Greek term would be *hexadecadic* (although in Modern Greek *deca-hexadic (δεκαεξαδικός)* is more commonly used).

Use in Chinese culture

The traditional Chinese units of weight were base-16. For example, one jīn (斤) in the old system equals sixteen taels. The suanpan (Chinese abacus) could be used to perform hexadecimal calculations.

Common patterns and humor

Hexadecimal is sometimes used in programmer jokes because some words can be formed using hexadecimal digits. Some of these words are "dead", "beef", "babe", and with appropriate substitutions "c0ffee". Since these are quickly recognizable by programmers, debugging setups sometimes initialize memory to them to help programmers see when something has not been initialized.

An example is the magic number in Universal Mach-O files and java class file structure, which is "`CAFEBABE`". Single-architecture 32-bit big-endian Mach-O files have the magic number "`FEEDFACE`" at their beginning. "`DEADBEEF`" is sometimes put into uninitialized memory. Microsoft Windows XP clears its locked index.dat files with the hex codes: "`0BADF00D`". The Visual C++ remote debugger uses "`BADCAB1E`" to denote a broken link to the target system.

Two common bit patterns often employed to test hardware are `01010101` and `10101010` in binary (their corresponding hex values are 55h and AAh, respectively). The reason for their use is to alternate between *off* ('0') to *on* ('1') or vice versa when switching between these two patterns. These two values are often used together as *signatures* in critical PC system sectors (e.g., the hex word, `0xAA55`, which on little-endian systems is 55h followed by AAh, must be at the end of a valid Master Boot Record).

The following table shows a joke referencing hexadecimal:

```
3x12 = 36
2x12 = 24
1x12 = 12
0x12 = 18
```

The first three lines are interpreted as decimal multiplication, but in the last, "0x" signals the Hexadecimal interpretation of "12", which is 18.

Another joke based on the use of a word containing only letters from the first six in the alphabet (and thus those used in hexadecimal) is...

> If only dead people understand hexadecimal, how many people understand hexadecimal?

In this case, "dead" refers to a hexadecimal number DEAD (57005 base 10), as opposed to the state of being deceased.

A Knuth reward check is one hexadecimal dollar, or $2.56.

Primary numeral system

Similar to dozenal advocacy, there have been occasional attempts to promote hexadecimal as the preferred numeral system. These attempts usually propose pronunciation and/or symbology.[14] Sometimes the proposal unifies standard measures so that they are multiples of 16.[15][16][17]

An example of unifying standard measures is Hexadecimal time, which subdivides a day by 16 so that there are 16 "hexhours" in a day.[17]

Key to number base notation

Simple key for notations used in article:

Full Text Notation	Abbreviation	Number Base
binary	bin	2
octal	oct	8
decimal	dec	10
hexadecimal	hex	16

References

[1] "Hexadecimal web colors explained" (http://www.web-colors-explained.com/hex.php). .

[2] Some of C's syntactic descendants are C++, C#, Java, JavaScript, Python and Windows PowerShell

[3] The string `"\x1B[0m\x1B[25;1H"` specifies the character sequence `Esc [ 0 m Esc [ 2 5 ; 1 H Nul`. These are the escape sequences used on an ANSI terminal that reset the character set and color, and then move the cursor to line 25.

[4] The VHDL MINI-REFERENCE: VHDL IDENTIFIERS, NUMBERS, STRINGS, AND EXPRESSIONS (http://www.eng.auburn.edu/department/ee/mgc/vhdl.html#numbers)

[5] MSX is Coming — Part 2: Inside MSX (http://www.atarimagazines.com/compute/issue56/107_1_MSX_IS_COMING.php) Compute!, issue 56, January 1985, p. 52

[6] BBC BASIC programs are not fully portable to Microsoft BASIC (without modification) since the latter takes `&` to prefix octal values. (Microsoft BASIC primarily uses `&O` to prefix octal, and it uses `&H` to prefix hexadecimal, but the ampersand alone yields a default interpretation as an octal prefix.

[7] Donald E. Knuth. *The TeXbook* (Computers and Typesetting, Volume A). Reading, Massachusetts: Addison-Wesley, 1984. ISBN 0-201-13448-9. The source code of the book in TeX (http://www.ctan.org/tex-archive/systems/knuth/tex/texbook.tex) (and a required set of macros CTAN.org (ftp://tug.ctan.org/pub/tex-archive/systems/knuth/lib/manmac.tex)) is available online on CTAN.

[8] This somewhat odd sequence was from the next six sequential numeric keyboard codes in the LGP-30's 6-bit character code. LGP-30 PROGRAMMING MANUAL (http://ed-thelen.org/comp-hist/lgp-30-man.html#R4.13)

[9] *Letters to the editor: On binary notation*, Bruce Alan Martin, Associated Universities Inc., Communications of the ACM, Volume 11, Issue 10 (October 1968) Page: 658 doi:10.1145/364096.364107

[10] Knuth, Donald. (1969). *Donald Knuth, in The Art of Computer Programming, Volume 2*. ISBN 0-201-03802-1. (Chapter 17.)

[11] A.B. Taylor, Report on Weights and Measures (http://books.google.com/books?id=X7wLAAAAYAAJ&pg=PP5), Pharmaceutical Association, 8th Annual Session, Boston, Sept. 15, 1859. See pages and 33 and 41.

[12] Alfred B. Taylor, Octonary numeration and its application to a system of weights and measures, Proc Amer. Phil. Soc. Vol XXIV (http://books.google.com/books?id=KsAUAAAAYAAJ&pg=PA296), Philadelphia, 1887; pages 296-366. See pages 317 and 322.

[13] Schwartzman, S. (1994). *The Words of Mathematics: an etymological dictionary of mathematical terms used in English*. ISBN 0-88385-511-9.

[14] "Base 4^2 Hexadecimal Symbol Proposal" (http://www.hauptmech.com/base42). .

[15] "Intuitor Hex Headquarters" (http://www.intuitor.com/hex/). .

[16] "A proposal for addition of the six Hexadecimal digits (A-F) to Unicode" (http://std.dkuug.dk/jtc1/sc2/wg2/docs/n2677). .

[17] Nystrom, John William (1862). *Project of a New System of Arithmetic, Weight, Measure and Coins: Proposed to be called the Tonal System, with Sixteen to the Base* (http://books.google.com/books?id=aNYGAAAAYAAJ). Philadelphia. .

Octal

The **octal** numeral system, or **oct** for short, is the base-8 number system, and uses the digits 0 to 7. Octal numerals can be made from binary numerals by grouping consecutive binary digits into groups of three (starting from the right). For example, the binary representation for decimal 74 is 1001010, which can be grouped into (00)1 001 010 – so the octal representation is 112.

In the decimal system each decimal place is a power of ten. For example:

$$\mathbf{74}_{10} = \mathbf{7} \times 10^1 + \mathbf{4} \times 10^0$$

In the octal system each place is a power of eight. For example:

$$\mathbf{112}_{8} = \mathbf{1} \times 8^2 + \mathbf{1} \times 8^1 + \mathbf{2} \times 8^0$$

By performing the calculation above in the familiar decimal system we see why 112 in octal is equal to 64+8+2 = 74 in decimal.

Usage

By Native Americans

The Yuki language in California and the Pamean languages[1] in Mexico have octal systems because the speakers count using the spaces between their fingers rather than the fingers themselves.[2]

By Europeans

- In 1716 King Charles XII of Sweden asked Emanuel Swedenborg to elaborate a number system based on 64 instead of 10. Swedenborg however argued that for people with less intelligence than the king such a big base would be too difficult and instead proposed 8 as the base. In 1718 Swedenborg wrote (but did not publish) a manuscript: "En ny räknekonst som omväxlas vid talet 8 istället för det vanliga vid talet 10" ("A new arithmetic (or art of counting) which changes at the Number 8 instead of the usual at the Number 10"). The numbers 1-7 are there denoted by the consonants l, s, n, m, t, f, u (v) and zero by the vowel o. Thus 8 = "lo", 16 = "so", 24 = "no", 64 = "loo", 512 = "looo" etc. Numbers with consecutive consonants are pronounced with vowel sounds between in accordance with a special rule.[3]
- Writing under the pseudonym "Hirossa Ap-Iccim" in The Gentleman's Magazine, (London) July 1745, Hugh Jones proposed an octal system for British coins, weights and measures. "Whereas reason and convenience indicate to us an uniform standard for all quantities; which I shall call the *Georigan standard*; and that is only to divide every integer in each *species* into eight equal parts, and every part again into 8 real or imaginary particles, as far as is necessary. For tho' all nations count universally by *tens* (originally occasioned by the number of digits on both hands) yet 8 is a far more complete and commodious number; since it is divisible into halves, quarters, and half quarters (or units) without a fraction, of which subdivision *ten* is uncapable...." In a later treatise on Octave computation (1753) Jones concluded: "Arithmetic by *Octaves* seems most agreeable to the Nature of Things, and therefore may be called Natural Arithmetic in Opposition to that now in Use, by Decades; which may be esteemed Artificial Arithmetic."[4]
- In 1801, James Anderson criticized the French for basing the Metric system on decimal arithmetic. He suggested base 8 for which he coined the term *octal*. His work was intended as recreational mathematics, but he suggested a purely octal system of weights and measures and observed that the existing system of English units was already, to a remarkable extent, an octal system.[5]
- In the mid 19th century, Alfred B. Taylor concluded that "Our octonary [base 8] radix is, therefore, beyond all comparison the "*best possible one*" for an arithmetical system." The proposal included a graphical notation for the

digits and new names for the numbers, suggesting that we should count "*un*, *du*, *the*, *fo*, *pa*, *se*, *ki*, *unty*, *unty-un*, *unty-du*" and so on, with successive multiples of eight named "*unty*, *duty*, *thety*, *foty*, *paty*, *sety*, *kity* and *under*." So, for example, the number 65 would be spoken in octonary as *under-un*.[6][7] Taylor also republished some of Swedenborg's work on octonary as an appendix to the above-cited publications.

In fiction

In the 2009 film *Avatar*, the language of the extraterrestrial Na'vi race employs an octal numeral system, probably due to the fact that they have four fingers on each hand.[8]

In the TV series *Stargate SG-1*, the Ancients, a race of beings responsible for the invention of the Stargates, used an octal system of mathematics.

In computers

Octal became widely used in computing when systems such as the PDP-8, ICL 1900 and IBM mainframes employed 12-bit, 24-bit or 36-bit words. Octal was an ideal abbreviation of binary for these machines because their word size is divisible by three (each octal digit represents three binary digits). So four, eight or twelve digits could concisely display an entire machine word. It also cut costs by allowing Nixie tubes, seven-segment displays, and calculators to be used for the operator consoles, where binary displays were too complex to use, decimal displays needed complex hardware to convert radices, and hexadecimal displays needed to display more numerals.

All modern computing platforms, however, use 16-, 32-, or 64-bit words, further divided into eight-bit bytes. On such systems three octal digits per byte would be required, with the most significant octal digit representing two binary digits (plus one bit of the next significant byte, if any). Octal representation of a 16-bit word requires 6 digits, but the most significant octal digit represents (quite inelegantly) only one bit (0 or 1). This representation offers no way to easily read the most significant byte, because it's smeared over four octal digits. Therefore, hexadecimal is more commonly used in programming languages today, since two hexadecimal digits exactly specify one byte. Some platforms with a power-of-two word size still have instruction subwords that are more easily understood if displayed in octal; this includes the PDP-11 and Motorola 68000 family. The modern-day ubiquitous x86 architecture belongs to this category as well, but octal is rarely used on this platform, although certain properties of the binary encoding of opcodes become more readily apparent when displayed in octal, e.g. the ModRM byte, which is divided into fields of 2, 3, and 3 bits, so octal can be useful in describing these encodings.

Octal is sometimes used in computing instead of hexadecimal, perhaps most often in modern times in conjunction with file permissions under Unix systems (see chmod). It has the advantage of not requiring any extra symbols as digits (the hexadecimal system is base-16 and therefore needs six additional symbols beyond 0–9). It is also used for digital displays.

In programming languages, octal literals are typically identified with a variety of prefixes, including the digit *0*, the letters *o* or *q*, or the digit–letter combination *0o*. For example, the literal 73 (base 8) might be represented as *073*, *o73*, *q73*, or *0o73* in various languages. Newer languages have been abandoning the prefix *0*, as decimal numbers are often represented with leading zeroes. The prefix *q* was introduced to avoid the prefix *o* being mistaken for a zero, while the prefix *0o* was introduced to avoid starting a numerical literal with an alphabetic character (like *o* or *q*), since these might cause the literal to be confused with a variable name. The prefix *0o* also follows the model set by the prefix *0x* used for hexadecimal literals in the C language.[9][10][11]

Octal numbers that are used in some programming languages (C, Perl, PostScript…) for textual/graphical representations of byte strings when some byte values (unrepresented in a code page, non-graphical, having special meaning in current context or otherwise undesired) have to be to escaped as `\nnn`. Octal representation of non-ASCII bytes may be particularly handy with UTF-8, where any start byte has octal value `\3nn` and any continuation byte has octal value `\2nn`.

Conversion between bases

Decimal to octal conversion

Method of successive division by 8

To convert integer decimals to octal, divide the original number by the largest possible power of 8 and successively divide the remainders by successively smaller powers of 8 until the power is 1. The octal representation is formed by the quotients, written in the order generated by the algorithm.

For example, to convert 125_{10} to octal:

$125 / 8^2 = \mathbf{1}$

$125 - 8^2 \times 1 = 61$

$61 / 8^1 = \mathbf{7}$

$61 - 8^1 \times 7 = 5$

$5 / 8^0 = \mathbf{5}$

Therefore, $125_{10} = 175_8$.

Another example:

$900 / 8^3 = \mathbf{1}$

$900 - 8^3 \times 1 = 388$

$388 / 8^2 = \mathbf{6}$

$388 - 8^2 \times 6 = 4$

$4 / 8^1 = \mathbf{0}$

$4 - 8^1 \times 0 = 4$

$4 / 8^0 = \mathbf{4}$

Therefore, $900_{10} = 1604_8$.

Method of successive multiplication by 8

To convert a decimal fraction to octal, multiply by 8; the integer part of the result is the first digit of the octal fraction. Repeat the process with the fractional part of the result, until it is null or within acceptable error bounds.

Example: Convert 0.1640625 to octal:

$0.1640625 \times 8 = 1.3125 = \mathbf{1} + 0.3125$

$0.3125 \times 8 = 2.5 = \mathbf{2} + 0.5$

$0.5 \times 8 = 4.0 = \mathbf{4} + 0$

Therefore, $0.1640625_{10} = 0.124_8$.

These two methods can be combined to handle decimal numbers with both integer and fractional parts, using the first on the integer part and the second on the fractional part.

Octal to decimal conversion

To convert a number `k` to decimal, use the formula that defines its base-8 representation:

$$k = \sum_{i=0}^{n} \left(a_i \times 8^i \right)$$

In this formula, a_i is an individual octal digit being converted, where `i` is the position of the digit (counting from 0 for the right-most digit).

Example: Convert 764_8 to decimal:

$$764_8 = 7 \times 8^2 + 6 \times 8^1 + 4 \times 8^0 = 448 + 48 + 4 = 500_{10}$$

For double-digit octal numbers this method amounts to multiplying the lead digit by 8 and adding the second digit to get the total.

Example: $65_8 = 6 \times 8 + 5 = 53_{10}$

Octal to binary conversion

To convert octal to binary, replace each octal digit by its binary representation. Example: Convert 51_8 to binary:

$5_8 = 101_2$

$1_8 = 001_2$

Therefore, $51_8 = 101\ 001_2$.

Binary to octal conversion

The process is the reverse of the previous algorithm. The binary digits are grouped by threes, starting from the least significant bit and proceeding to the left and to the right. Add leading 0s (or trailing zeros to the right of decimal point) to fill out the last group of three if necessary. Then replace each trio with the equivalent octal digit.

For instance, convert binary 1010111100 to octal:

001	010	111	100
1	2	7	4

Therefore, $1010111100_2 = 1274_8$.

Convert binary 11100.01001 to octal:

011	100	.	010	010
3	4	.	2	2

Therefore, $11100.01001_2 = 34.22_8$.

Octal to hexadecimal conversion

The conversion is made in two steps using binary as an intermediate base. Octal is converted to binary and then binary to hexadecimal, grouping digits by fours, which correspond each to a hexadecimal digit.

For instance, convert octal 1057 to hexadecimal:

To binary:

1	0	5	7
001	000	101	111

then to hexadecimal:

0010	0010	1111
2	2	F

Therefore, $1057_8 = 22F_{16}$.

Hexadecimal to octal conversion

Hexadecimal to octal conversion proceeds by first converting the hexadecimal digits to 4-bit binary values, then regrouping the binary bits into 3-bit octal digits.

For example, to convert $3FA5_{16}$:

To binary:

3	F	A	5
0011	1111	1010	0101

then to octal:

0	011	111	110	100	101
0	3	7	6	4	5

Therefore, $3FA5_{16} = 37645_8$.

References

[1] Avelino, Heriberto (2006). "The typology of Pame number systems and the limits of Mesoamerica as a linguistic area" (http://linguistics.berkeley.edu/~avelino/Avelino_2006.pdf). *Linguistic Typology* **10** (1): 41–60. doi:10.1515/LINGTY.2006.002.

[2] Marcia Ascher. "Ethnomathematics: A Multicultural View of Mathematical Ideas" (http://links.jstor.org/sici?sici=0746-8342(199209)23:4<353:EAMVOM>2.0.CO;2-#&size=LARGE). The College Mathematics Journal. . Retrieved 2007-04-13.

[3] Donald Knuth, *The Art of Computer Programming*

[4] See H.R. Phalen, "Hugh Jones and Octave Computation," *The American Mathematical Monthly* 56 (August–September 1949): 461-65.

[5] James Anderson, On Octal Arithmetic [title appears only in page headers], Recreations in Agriculture, Natural-History, Arts, and Miscellaneous Literature (http://books.google.com/books?id=olhHAAAAYAAJ&pg=PA437), Vol. IV, No. 6 (Feb. 1801), T. Bensley, London; pages 437-448.

[6] A.B. Taylor, Report on Weights and Measures (http://books.google.com/books?id=X7wLAAAAYAAJ&pg=PP5), Pharmaceutical Association, 8th Annual Session, Boston, Sept. 15, 1859. See pages and 48 and 53.

[7] Alfred B. Taylor, Octonary numeration and its application to a system of weights and measures, Proc. Amer. Phil. Soc. Vol XXIV (http://books.google.com/books?id=KsAUAAAAYAAJ&pg=PA296), Philadelphia, 1887; pages 296-366. See pages 327 and 330.

[8] Counting in Na'vi (http://www.languagesandnumbers.com/how-to-count-in-navi/en/navi/)

[9] ECMAScript 5th Edition: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[10] Perl 6: http://svn.pugscode.org/pugs/docs/Perl6/Spec/S02-bits.pod

[11] Python 3: http://docs.python.org/3.1/reference/lexical_analysis.html#literals

External links

- Octomatics (http://www.octomatics.org) is a numeral system enabling simple visual calculation in octal.

Binary number

In mathematics and computer science, the **binary numeral system**, or **base-2 numeral system**, represents numeric values using two symbols: 0 and 1. More specifically, the usual base-2 system is a positional notation with a radix of 2. Numbers represented in this system are commonly called **binary numbers**. Because of its straightforward implementation in digital electronic circuitry using logic gates, the binary system is used internally by almost all modern computers and computer-based devices such as mobile phones.

History

The Indian scholar Pingala (around 5th–2nd centuries BC) developed mathematical concepts for describing prosody, and in doing so presented the first known description of a binary numeral system.[1][2] He used binary numbers in the form of short and long syllables (the latter equal in length to two short syllables), making it similar to Morse code.[3][4]

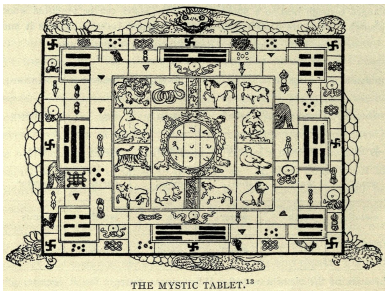
Pingala's Hindu classic titled Chandaḥśāstra (8.23) describes the formation of a matrix in order to give a unique value to each meter. An example of such a matrix is as follows (note that these binary representations are "backwards" compared to modern, Western positional notation):[5][6]

0 0 0 0 numerical value 1_{10}
1 0 0 0 numerical value 2_{10}
0 1 0 0 numerical value 3_{10}
1 1 0 0 numerical value 4_{10}

A set of eight trigrams (Bagua) and a set of 64 hexagrams ("sixty-four" gua), analogous to the three-bit and six-bit binary numerals, were in usage at least as early as the Zhou Dynasty of ancient China through the classic text *Yijing*.

Daoist Bagua

In the 11th century, scholar and philosopher Shao Yong developed a method for arranging the hexagrams which corresponds, albeit unintentionally, to the sequence 0 to 63, as represented in binary, with yin as 0, yang as 1 and the least significant bit on top. The ordering is also the lexicographical order on sextuples of elements chosen from a two-element set.[7]



Tibetan Buddhist "Mystic Tablet"

Similar sets of binary combinations have also been used in traditional African divination systems such as Ifá as well as in medieval Western geomancy. The base-2 system utilized in geomancy had long been widely applied in sub-Saharan Africa.

In 1605 Francis Bacon discussed a system whereby letters of the alphabet could be reduced to sequences of binary digits, which could then be encoded as scarcely visible variations in the font in any random text.[8] Importantly for the general theory of binary encoding, he added that this method could be used with any objects at all: "provided those objects be capable of a twofold difference only; as by Bells, by Trumpets, by Lights and Torches, by the report of Muskets, and any instruments of like nature".[8] (See Bacon's cipher.)


Gottfried Leibniz

The modern binary number system was studied by Gottfried Leibniz in 1679. See his article:*Explication de l'Arithmétique Binaire*[9](1703). Leibniz's system uses 0 and 1, like the modern binary numeral system. As a Sinophile, Leibniz was aware of the Yijing (or I-Ching) and noted with fascination how its hexagrams correspond to the binary numbers from 0 to 111111, and concluded that this mapping was evidence of major Chinese accomplishments in the sort of philosophical mathematics he admired.[10]

In 1854, British mathematician George Boole published a landmark paper detailing an algebraic system of logic that would become known as Boolean algebra. His logical calculus was to become instrumental in the design of digital electronic circuitry.[11]

In 1937, Claude Shannon produced his master's thesis at MIT that implemented Boolean algebra and binary arithmetic using electronic relays and switches for the first time in history. Entitled *A Symbolic Analysis of Relay and Switching Circuits*, Shannon's thesis essentially founded practical digital circuit design.[12]

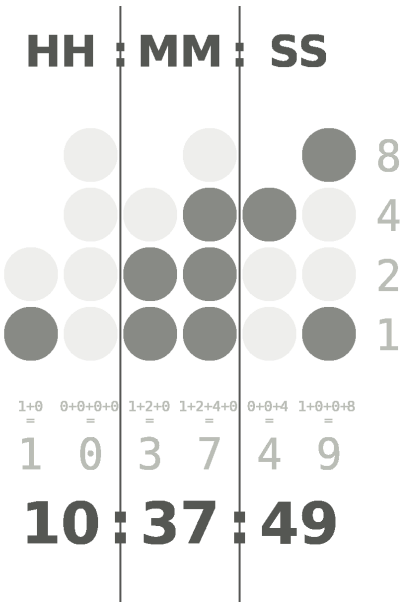
In November 1937, George Stibitz, then working at Bell Labs, completed a relay-based computer he dubbed the "Model K" (for "**K**itchen", where he had assembled it), which calculated using binary addition.[13] Bell Labs thus authorized a full research programme in late 1938 with Stibitz at the helm. Their Complex Number Computer, completed 8 January 1940, was able to calculate complex numbers. In a demonstration to the American Mathematical Society conference at Dartmouth College on 11 September 1940, Stibitz was able to send the Complex Number Calculator remote commands over telephone lines by a teletype. It was the first computing machine ever used remotely over a phone line. Some participants of the conference who witnessed the demonstration were John Von Neumann, John Mauchly and Norbert Wiener, who wrote about it in his memoirs.[14][15][16]

Representation

Any number can be represented by any sequence of bits (binary digits), which in turn may be represented by any mechanism capable of being in two mutually exclusive states. The following sequence of symbols could all be interpreted as the binary numeric value of 667:

```
1 0 1 0 0 1 1 0 1 1
| – | – – | | – | |
x o x o o x x o x x
y n y n n y y n y y
```

The numeric value represented in each case is dependent upon the value assigned to each symbol. In a computer, the numeric values may be represented by two different voltages; on a magnetic disk, magnetic polarities may be used. A "positive", "yes", or "on" state is not necessarily equivalent to the numerical value of one; it depends on the architecture in use.



A binary clock might use LEDs to express binary values. In this clock, each column of LEDs shows a binary-coded decimal numeral of the traditional sexagesimal time.

In keeping with customary representation of numerals using Arabic numerals, binary numbers are commonly written using the symbols **0** and **1**. When written, binary numerals are often subscripted, prefixed or suffixed in order to indicate their base, or radix. The following notations are equivalent:

100101 binary (explicit statement of format)

100101b (a suffix indicating binary format)

100101B (a suffix indicating binary format)

bin 100101 (a prefix indicating binary format)

100101_2 (a subscript indicating base-2 (binary) notation)

%100101 (a prefix indicating binary format)

0b100101 (a prefix indicating binary format, common in programming languages)

6b100101 (a prefix indicating number of bits in binary format, common in programming languages)

When spoken, binary numerals are usually read digit-by-digit, in order to distinguish them from decimal numerals. For example, the binary numeral 100 is pronounced *one zero zero*, rather than *one hundred*, to make its binary nature explicit, and for purposes of correctness. Since the binary numeral 100 represents the value four, it would be confusing to refer to the numeral as *one hundred* (a word that represents a completely different value, or amount). Alternatively, the binary numeral 100 can be read out as "four" (the correct *value*), but this does not make its binary nature explicit.

Counting in binary

Decimal pattern (Hex Value)	Binary numbers
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10 – (A)	1010
11 – (B)	1011
12 – (C)	1100
13 – (D)	1101
14 – (E)	1110
15 – (F)	1111
16 – (10)	10000

Counting in binary is similar to counting in any other number system. Beginning with a single digit, counting proceeds through each symbol, in increasing order. Decimal counting uses the symbols **0** through **9**, while binary only uses the symbols **0** and **1**.

When the symbols for the first digit are exhausted, the next-higher digit (to the left) is incremented, and counting starts over at 0. In decimal, counting proceeds like so:

000, 001, 002, ... 007, 008, 009, (rightmost digit starts over, and next digit is incremented)

0**1**0, 011, 012, ...

...

090, 091, 092, ... 097, 098, 099, (rightmost two digits start over, and next digit is incremented)

100, 101, 102, ...

After a digit reaches 9, an increment resets it to 0 but also causes an increment of the next digit to the left. In binary, counting is the same except that only the two symbols 0 and 1 are used. Thus after a digit reaches 1 in binary, an increment resets it to 0 but also causes an increment of the next digit to the left:

0000,

000**1**, (rightmost digit starts over, and next digit is incremented)

00**1**0, 0011, (rightmost two digits start over, and next digit is incremented)

0**1**00, 0101, 0110, 0111, (rightmost three digits start over, and the next digit is incremented)

1000, 1001, ...

Since binary is a base-2 system, each digit represents an increasing power of 2, with the rightmost digit representing 2^0, the next representing 2^1, then 2^2, and so on. To determine the decimal representation of a binary number simply

take the sum of the products of the binary digits and the powers of 2 which they represent. For example, the binary number:

100101

is converted to decimal form by:

$[(\mathbf{1}) \times 2^5] + [(\mathbf{0}) \times 2^4] + [(\mathbf{0}) \times 2^3] + [(\mathbf{1}) \times 2^2] + [(\mathbf{0}) \times 2^1] + [(\mathbf{1}) \times 2^0] =$

$[\mathbf{1} \times 32] + [\mathbf{0} \times 16] + [\mathbf{0} \times 8] + [\mathbf{1} \times 4] + [\mathbf{0} \times 2] + [\mathbf{1} \times 1] = 37_{10}$

To create higher numbers, additional digits are simply added to the left side of the binary representation.

Fractions

Fractions in binary only terminate if the denominator has 2 as the only prime factor. As a result, 1/10 does not have a finite binary representation, and this causes 10×0.1 not to be precisely equal to 1 in floating point arithmetic. As an example, to interpret the binary expression for 1/3 = .010101..., this means: $1/3 = 0 \times \mathbf{2^{-1}} + 1 \times \mathbf{2^{-2}} + 0 \times \mathbf{2^{-3}} + 1 \times \mathbf{2^{-4}} + ... = 0.3125 + ...$ An exact value cannot be found with a sum of a finite number of inverse powers of two, and zeros and ones alternate forever.

Fraction	Decimal	Binary	Fractional approximation
1/1	1 or 0.999...	1 or 0.111...	1/2 + 1/4 + 1/8...
1/2	0.5 or 0.4999...	0.1 or 0.0111...	1/4 + 1/8 + 1/16 . . .
1/3	0.333...	0.010101...	1/4 + 1/16 + 1/64 . . .
1/4	0.25 or 0.24999...	0.01 or 0.00111...	1/8 + 1/16 + 1/32 . . .
1/5	0.2 or 0.1999...	0.00110011...	1/8 + 1/16 + 1/128 . . .
1/6	0.1666...	0.0010101...	1/8 + 1/32 + 1/128 . . .
1/7	0.142857142857...	0.001001...	1/8 + 1/64 + 1/512 . . .
1/8	0.125 or 0.124999...	0.001 or 0.000111...	1/16 + 1/32 + 1/64 . . .
1/9	0.111...	0.000111000111...	1/16 + 1/32 + 1/64 . . .
1/10	0.1 or 0.0999...	0.000110011...	1/16 + 1/32 + 1/256 . . .
1/11	0.090909...	0.0001011101000101 1101...	1/16 + 1/64 + 1/128 . . .
1/12	0.08333...	0.00010101...	1/16 + 1/64 + 1/256 . . .
1/13	0.076923076923...	0.000100111011000100111011...	1/16 + 1/128 + 1/256 . . .
1/14	0.0714285714285...	0.0001001001...	1/16 + 1/128 + 1/1024 . . .
1/15	0.0666...	0.00010001...	1/16 + 1/256 . . .
1/16	0.0625 or 0.0624999...	0.0001 or 0.0000111...	1/32 + 1/64 + 1/128 . . .

Binary arithmetic

Arithmetic in binary is much like arithmetic in other numeral systems. Addition, subtraction, multiplication, and division can be performed on binary numerals.

Addition

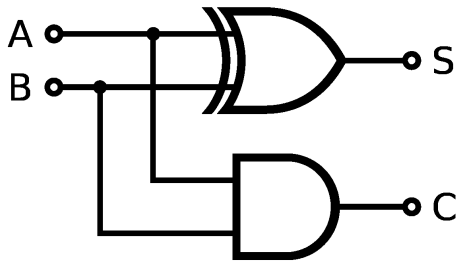
The simplest arithmetic operation in binary is addition. Adding two single-digit binary numbers is relatively simple, using a form of carrying:

$0 + 0 \rightarrow 0$

$0 + 1 \rightarrow 1$

$1 + 0 \rightarrow 1$

$1 + 1 \rightarrow 0$, carry 1 (since $1 + 1 = 0 + 1 \times$ binary 10)



The circuit diagram for a binary half adder, which adds two bits together, producing sum and carry bits.

Adding two "1" digits produces a digit "0", while 1 will have to be added to the next column. This is similar to what happens in decimal when certain single-digit numbers are added together; if the result equals or exceeds the value of the radix (10), the digit to the left is incremented:

$5 + 5 \rightarrow 0$, carry 1 (since 5 + 5 = 10 carry 1)

$7 + 9 \rightarrow 6$, carry 1 (since 7 + 9 = 16 carry 1)

This is known as *carrying*. When the result of an addition exceeds the value of a digit, the procedure is to "carry" the excess amount divided by the radix (that is, 10/10) to the left, adding it to the next positional value. This is correct since the next position has a weight that is higher by a factor equal to the radix. Carrying works the same way in binary:

```
  1 1 1 1 1  (carried digits)
    0 1 1 0 1
+   1 0 1 1 1
-------------
= 1 0 0 1 0 0 = 36
```

In this example, two numerals are being added together: 01101_2 (13_{10}) and 10111_2 (23_{10}). The top row shows the carry bits used. Starting in the rightmost column, $1 + 1 = 10_2$. The 1 is carried to the left, and the 0 is written at the bottom of the rightmost column. The second column from the right is added: $1 + 0 + 1 = 10_2$ again; the 1 is carried, and 0 is written at the bottom. The third column: $1 + 1 + 1 = 11_2$. This time, a 1 is carried, and a 1 is written in the bottom row. Proceeding like this gives the final answer 100100_2 (36 decimal).

When computers must add two numbers, the rule that: x xor y = (x + y) mod 2 for any two bits x and y allows for very fast calculation, as well.

A simplification for many binary addition problems is the Long Carry Method or Brookhouse Method of Binary Addition. This method is generally useful in any binary addition where one of the numbers has a long string of “1” digits. For example the following large binary numbers can be added in two simple steps without multiple carries from one place to the next.

```
  1 1 1   1 1 1 1 1  (carried digits)  (Long Carry Method)
    1 1 1 0 1 1 1 1 1 0                          1 1 1  0  1 1 1 1 1  0
+   1 0 1 0 1 1 0 0 1 1  Versus:       +            1 0  1  0 1 1 0 0  1  1  add crossed out digits first
----------------------                         + 1 0 0 0 1 0 0 0 0 0 0  = sum of crossed out digits
= 1 1 0 0 1 1 1 0 0 0 1                        -----------------------  now add remaining digits
```

```
  1 1 0 0 1 1 1 0 0 0 1
```

In this example, two numerals are being added together: $1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 0_2$ (958_{10}) and $1\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1_2$ (691_{10}). The top row shows the carry bits used. Instead of the standard carry from one column to the next, the lowest place-valued "1" with a "1" in the corresponding place value beneath it may be added and a "1" may be carried to one digit past the end of the series. These numbers must be crossed off since they are already added. Then simply add that result to the uncanceled digits in the second row. Proceeding like this gives the final answer $1\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 1_2$ (1649_{10}).

Addition table

	0	**1**
0	0	1
1	1	10

The binary addition table is similar, but not the same, as the truth table of the logical disjunction operation $\vee$. The difference is that $1 \vee 1 = 1$, while $1 + 1 = 10$.

Subtraction

Subtraction works in much the same way:

$0 - 0 \rightarrow 0$

$0 - 1 \rightarrow 1$, borrow 1

$1 - 0 \rightarrow 1$

$1 - 1 \rightarrow 0$

Subtracting a "1" digit from a "0" digit produces the digit "1", while 1 will have to be subtracted from the next column. This is known as *borrowing*. The principle is the same as for carrying. When the result of a subtraction is less than 0, the least possible value of a digit, the procedure is to "borrow" the deficit divided by the radix (that is, 10/10) from the left, subtracting it from the next positional value.

```
    *   * * *   (starred columns are borrowed from)
  1 1 0 1 1 1 0
-   1 0 1 1 1
---------------
= 1 0 1 0 1 1 1
```

Subtracting a positive number is equivalent to *adding* a negative number of equal absolute value; computers typically use two's complement notation to represent negative values. This notation eliminates the need for a separate "subtract" operation. Using two's complement notation subtraction can be summarized by the following formula:

A − B = A + not B + 1

For further details, see two's complement.

Multiplication

Multiplication in binary is similar to its decimal counterpart. Two numbers *A* and *B* can be multiplied by partial products: for each digit in *B*, the product of that digit in *A* is calculated and written on a new line, shifted leftward so that its rightmost digit lines up with the digit in *B* that was used. The sum of all these partial products gives the final result.

Since there are only two digits in binary, there are only two possible outcomes of each partial multiplication:

- If the digit in *B* is 0, the partial product is also 0
- If the digit in *B* is 1, the partial product is equal to *A*

For example, the binary numbers 1011 and 1010 are multiplied as follows:

```
          1 0 1 1   (A)
        × 1 0 1 0   (B)
        ---------
          0 0 0 0   ← Corresponds to a zero in B
  +     1 0 1 1     ← Corresponds to a one in B
  +   0 0 0 0
  + 1 0 1 1
  ---------------
  = 1 1 0 1 1 1 0
```

Binary numbers can also be multiplied with bits after a binary point:

```
              1 0 1.1 0 1  (A) (5.625 in decimal)
            × 1 1 0.0 1  (B) (6.25  in decimal)
            -------------
                  1.0 1 1 0 1   ← Corresponds to a one in B
    +           0 0.0 0 0 0     ← Corresponds to a zero in B
    +         0 0 0.0 0 0
    +       1 0 1 1.0 1
    +     1 0 1 1 0.1
    ----------------------
    =   1 0 0 0 1 1.0 0 1 0 1  (35.15625 in decimal)
```

See also Booth's multiplication algorithm.

Multiplication table

	0	**1**
0	0	0
1	0	1

The binary multiplication table is the same as the Truth table of the Logical conjunction operation $\wedge$.

Division

Binary division is again similar to its decimal counterpart:

Here, the divisor is 101_2, or 5 decimal, while the dividend is 11011_2, or 27 decimal. The procedure is the same as that of decimal long division; here, the divisor 101_2 goes into the first three digits 110_2 of the dividend one time, so a "1" is written on the top line. This result is multiplied by the divisor, and subtracted from the first three digits of the dividend; the next digit (a "1") is included to obtain a new three-digit sequence:

```
              1
          ___________
1 0 1    ) 1 1 0 1 1
         - 1 0 1
           -----
             0 1 1
```

The procedure is then repeated with the new sequence, continuing until the digits in the dividend have been exhausted:

```
              1 0 1
          ___________
1 0 1    ) 1 1 0 1 1
         - 1 0 1
           -----
             0 1 1
           - 0 0 0
             -----
               1 1 1
             - 1 0 1
               -----
                 1 0
```

Thus, the quotient of 11011_2 divided by 101_2 is 101_2, as shown on the top line, while the remainder, shown on the bottom line, is 10_2. In decimal, 27 divided by 5 is 5, with a remainder of 2.

Square root

Binary square root is similar to its decimal counterpart too. But, it's simpler than that in decimal.

$$(10x+y)^{10} - 100x^{10} = y^{10} + 100xy = \begin{cases} 0, & y = 0 \\ 100x+1, & y = 1 \end{cases}$$

for example

```
              1 0 0 1
            ----------
          √ 1010001
              1
            ----------
      101     01
                0
             ---------
      1001      100
```

```
              0
          --------
  10001    10001
           10001
           -------
                0
```

Bitwise operations

Though not directly related to the numerical interpretation of binary symbols, sequences of bits may be manipulated using Boolean logical operators. When a string of binary symbols is manipulated in this way, it is called a bitwise operation; the logical operators AND, OR, and XOR may be performed on corresponding bits in two binary numerals provided as input. The logical NOT operation may be performed on individual bits in a single binary numeral provided as input. Sometimes, such operations may be used as arithmetic short-cuts, and may have other computational benefits as well. For example, an arithmetic shift left of a binary number is the equivalent of multiplication by a (positive, integral) power of 2.

Conversion to and from other numeral systems

Decimal

To convert from a base-10 integer numeral to its base-2 (binary) equivalent, the number is divided by two, and the remainder is the least-significant bit. The (integer) result is again divided by two, its remainder is the next least significant bit. This process repeats until the quotient becomes zero.

Conversion from base-2 to base-10 proceeds by applying the preceding algorithm, so to speak, in reverse. The bits of the binary number are used one by one, starting with the most significant (leftmost) bit. Beginning with the value 0, repeatedly double the prior value and add the next bit to produce the next value. This can be organized in a multi-column table. For example to convert 10010101101_2 to decimal:

Prior value	× 2 +	Next bit	Next value
0	× 2 +	**1**	= 1
1	× 2 +	**0**	= 2
2	× 2 +	**0**	= 4
4	× 2 +	**1**	= 9
9	× 2 +	**0**	= 18
18	× 2 +	**1**	= 37
37	× 2 +	**0**	= 74
74	× 2 +	**1**	= 149
149	× 2 +	**1**	= 299
299	× 2 +	**0**	= 598
598	× 2 +	**1**	= **1197**

The result is 1197_{10}. Note that the first Prior Value of 0 is simply an initial decimal value. This method is an application of the Horner scheme.

Binary	1	0	0	1	0	1	0	1	1	0	1

Decimal $1\times2^{10} + 0\times2^{9} + 0\times2^{8} + 1\times2^{7} + 0\times2^{6} + 1\times2^{5} + 0\times2^{4} + 1\times2^{3} + 1\times2^{2} + 0\times2^{1} + 1\times2^{0} = 1197$

The fractional parts of a number are converted with similar methods. They are again based on the equivalence of shifting with doubling or halving.

In a fractional binary number such as 0.11010110101_2, the first digit is $\frac{1}{2}$, the second $\left(\frac{1}{2}\right)^2 = \frac{1}{4}$, etc. So if there is a 1 in the first place after the decimal, then the number is at least $\frac{1}{2}$, and vice versa. Double that number is at least 1.

This suggests the algorithm: Repeatedly double the number to be converted, record if the result is at least 1, and then throw away the integer part.

For example, $\left(\frac{1}{3}\right)_{10}$, in binary, is:

Converting	Result
$\frac{1}{3}$	0.
$\frac{1}{3} \times 2 = \frac{2}{3} < 1$	0.0
$\frac{2}{3} \times 2 = 1\frac{1}{3} \geq 1$	0.01
$\frac{1}{3} \times 2 = \frac{2}{3} < 1$	0.010
$\frac{2}{3} \times 2 = 1\frac{1}{3} \geq 1$	0.0101

Thus the repeating decimal fraction 0.3... is equivalent to the repeating binary fraction 0.01... .

Or for example, 0.1_{10}, in binary, is:

Converting	Result
0.1	0.
0.1 × 2 = **0.2** < 1	0.0
0.2 × 2 = **0.4** < 1	0.00
0.4 × 2 = **0.8** < 1	0.000
0.8 × 2 = **1.6** ≥ 1	0.0001
0.6 × 2 = **1.2** ≥ 1	0.00011
0.2 × 2 = **0.4** < 1	0.000110
0.4 × 2 = **0.8** < 1	0.0001100
0.8 × 2 = **1.6** ≥ 1	0.00011001
0.6 × 2 = **1.2** ≥ 1	0.000110011
0.2 × 2 = **0.4** < 1	0.0001100110

This is also a repeating binary fraction 0.00011... . It may come as a surprise that terminating decimal fractions can have repeating expansions in binary. It is for this reason that many are surprised to discover that 0.1 + ... + 0.1, (10 additions) differs from 1 in floating point arithmetic. In fact, the only binary fractions with terminating expansions are of the form of an integer divided by a power of 2, which 1/10 is not.

The final conversion is from binary to decimal fractions. The only difficulty arises with repeating fractions, but otherwise the method is to shift the fraction to an integer, convert it as above, and then divide by the appropriate power of two in the decimal base. For example:

$$
\begin{aligned}
x &= 1100.1\overline{01110}\ldots \\
x \times 2^6 &= 1100101110.\overline{01110}\ldots \\
x \times 2 &= 11001.\overline{01110}\ldots \\
x \times (2^6 - 2) &= 1100010101 \\
x &= 1100010101/111110 \\
x &= (789/62)_{10}
\end{aligned}
$$

Another way of converting from binary to decimal, often quicker for a person familiar with hexadecimal, is to do so indirectly—first converting (x in binary) into (x in hexadecimal) and then converting (x in hexadecimal) into (x in decimal).

For very large numbers, these simple methods are inefficient because they perform a large number of multiplications or divisions where one operand is very large. A simple divide-and-conquer algorithm is more effective asymptotically: given a binary number, it is divided by 10^k, where k is chosen so that the quotient roughly equals the remainder; then each of these pieces is converted to decimal and the two are concatenated. Given a decimal number, it can be split into two pieces of about the same size, each of which is converted to binary, whereupon the first converted piece is multiplied by 10^k and added to the second converted piece, where k is the number of decimal digits in the second, least-significant piece before conversion.

Hexadecimal

0$_{\text{hex}}$	=	0$_{\text{dec}}$	=	0$_{\text{oct}}$	0	0	0	0
1$_{\text{hex}}$	=	1$_{\text{dec}}$	=	1$_{\text{oct}}$	0	0	0	1
2$_{\text{hex}}$	=	2$_{\text{dec}}$	=	2$_{\text{oct}}$	0	0	1	0
3$_{\text{hex}}$	=	3$_{\text{dec}}$	=	3$_{\text{oct}}$	0	0	1	1
4$_{\text{hex}}$	=	4$_{\text{dec}}$	=	4$_{\text{oct}}$	0	1	0	0
5$_{\text{hex}}$	=	5$_{\text{dec}}$	=	5$_{\text{oct}}$	0	1	0	1
6$_{\text{hex}}$	=	6$_{\text{dec}}$	=	6$_{\text{oct}}$	0	1	1	0
7$_{\text{hex}}$	=	7$_{\text{dec}}$	=	7$_{\text{oct}}$	0	1	1	1
8$_{\text{hex}}$	=	8$_{\text{dec}}$	=	10$_{\text{oct}}$	1	0	0	0
9$_{\text{hex}}$	=	9$_{\text{dec}}$	=	11$_{\text{oct}}$	1	0	0	1
A$_{\text{hex}}$	=	10$_{\text{dec}}$	=	12$_{\text{oct}}$	1	0	1	0
B$_{\text{hex}}$	=	11$_{\text{dec}}$	=	13$_{\text{oct}}$	1	0	1	1
C$_{\text{hex}}$	=	12$_{\text{dec}}$	=	14$_{\text{oct}}$	1	1	0	0
D$_{\text{hex}}$	=	13$_{\text{dec}}$	=	15$_{\text{oct}}$	1	1	0	1
E$_{\text{hex}}$	=	14$_{\text{dec}}$	=	16$_{\text{oct}}$	1	1	1	0
F$_{\text{hex}}$	=	15$_{\text{dec}}$	=	17$_{\text{oct}}$	1	1	1	1

Binary may be converted to and from hexadecimal somewhat more easily. This is because the radix of the hexadecimal system (16) is a power of the radix of the binary system (2). More specifically, $16 = 2^4$, so it takes four digits of binary to represent one digit of hexadecimal, as shown in the table to the right.

To convert a hexadecimal number into its binary equivalent, simply substitute the corresponding binary digits:

$3A_{16} = 0011\ 1010_2$

$E7_{16} = 1110\ 0111_2$

To convert a binary number into its hexadecimal equivalent, divide it into groups of four bits. If the number of bits isn't a multiple of four, simply insert extra **0** bits at the left (called padding). For example:

$1010010_2 = 0101\ 0010$ grouped with padding $= 52_{16}$

$11011101_2 = 1101\ 1101$ grouped $= DD_{16}$

To convert a hexadecimal number into its decimal equivalent, multiply the decimal equivalent of each hexadecimal digit by the corresponding power of 16 and add the resulting values:

$C0E7_{16} = (12 \times 16^3) + (0 \times 16^2) + (14 \times 16^1) + (7 \times 16^0) = (12 \times 4096) + (0 \times 256) + (14 \times 16) + (7 \times 1) = 49{,}383_{10}$

Octal

Binary is also easily converted to the octal numeral system, since octal uses a radix of 8, which is a power of two (namely, 2^3, so it takes exactly three binary digits to represent an octal digit). The correspondence between octal and binary numerals is the same as for the first eight digits of hexadecimal in the table above. Binary 000 is equivalent to the octal digit 0, binary 111 is equivalent to octal 7, and so forth.

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

Converting from octal to binary proceeds in the same fashion as it does for hexadecimal:

$65_8 = 110\ 101_2$

$17_8 = 001\ 111_2$

And from binary to octal:

$101100_2 = 101\ 100_2$ grouped $= 54_8$

$10011_2 = 010\ 011_2$ grouped with padding $= 23_8$

And from octal to decimal:

$65_8 = (6 \times 8^1) + (5 \times 8^0) = (6 \times 8) + (5 \times 1) = 53_{10}$

$127_8 = (1 \times 8^2) + (2 \times 8^1) + (7 \times 8^0) = (1 \times 64) + (2 \times 8) + (7 \times 1) = 87_{10}$

Representing real numbers

Non-integers can be represented by using negative powers, which are set off from the other digits by means of a radix point (called a decimal point in the decimal system). For example, the binary number 11.01_2 thus means:

$$\mathbf{1} \times 2^{1} \quad (1 \times 2 = \mathbf{2}) \qquad \text{plus}$$

$$\mathbf{1} \times 2^{0} \quad (1 \times 1 = \mathbf{1}) \qquad \text{plus}$$

$$\mathbf{0} \times 2^{-1} \quad (0 \times ½ = \mathbf{0}) \qquad \text{plus}$$

$$\mathbf{1} \times 2^{-2} \quad (1 \times ¼ = \mathbf{0.25})$$

For a total of 3.25 decimal.

All dyadic rational numbers $\frac{p}{2^a}$ have a *terminating* binary numeral—the binary representation has a finite number of terms after the radix point. Other rational numbers have binary representation, but instead of terminating, they *recur*, with a finite sequence of digits repeating indefinitely. For instance

$$\frac{1_{10}}{3_{10}} = \frac{1_2}{11_2} = 0.0101010101\ldots_2$$

$$\frac{12_{10}}{17_{10}} = \frac{1100_2}{10001_2} = 0.10110100\ 10110100\ 10110100\ldots_2$$

The phenomenon that the binary representation of any rational is either terminating or recurring also occurs in other radix-based numeral systems. See, for instance, the explanation in decimal. Another similarity is the existence of alternative representations for any terminating representation, relying on the fact that 0.111111… is the sum of the geometric series $2^{-1} + 2^{-2} + 2^{-3} + \ldots$ which is 1.

Binary numerals which neither terminate nor recur represent irrational numbers. For instance,

- 0.1010010001000010000010 0… does have a pattern, but it is not a fixed-length recurring pattern, so the number is irrational
- 1.0110101000001001111001100110011111111 0… is the binary representation of $\sqrt{2}$, the square root of 2, another irrational. It has no discernible pattern. See irrational number.

Notes

[1] Sanchez, Julio; Canton, Maria P. (2007). *Microcontroller programming : the microchip PIC*. Boca Raton, Florida: CRC Press. p. 37. ISBN 0-8493-7189-9

[2] W. S. Anglin and J. Lambek, *The Heritage of Thales*, Springer, 1995, ISBN 0-387-94544-X

[3] Binary Numbers in Ancient India (http://home.ica.net/~roymanju/Binary.htm)

[4] Math for Poets and Drummers (http://www.sju.edu/~rhall/Rhythms/Poets/arcadia.pdf) (pdf, 145KB)

[5] "Binary Numbers in Ancient India" (http://home.ica.net/~roymanju/Binary.htm). .

[6] Stakhov, Alexey; Stakhov, Alekseĭ; Olsen, Scott (2009). *The mathematics of harmony: from Euclid to contemporary mathematics and computer science* (http://books.google.com/books?id=K6fac9RxXREC). ISBN 978-981-277-582-5. .

[7] Ryan, James A. (January 1996). "Leibniz' Binary System and Shao Yong's "Yijing"". *Philosophy East and West* (University of Hawaii Press) **46** (1): 59–90. doi:10.2307/1399337. JSTOR 1399337.

[8] Bacon, Francis (1605). "The Advancement of Learning" (http://home.hiwaay.net/~paul/bacon/advancement/book6ch1.html). London. pp. Chapter 1.

[9] Leibniz G., Explication de l'Arithmétique Binaire, Die Mathematische Schriften, ed. C. Gerhardt, Berlin 1879, vol.7, p.223; Engl. transl. (http://www.leibniz-translations.com/binary.htm)

[10] Aiton, Eric J. (1985). *Leibniz: A Biography*. Taylor & Francis. pp. 245–8. ISBN 0-85274-470-6

[11] Boole, George (2009) [1854]. *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities* (http://www.gutenberg.org/etext/15114) (Macmillan, Dover Publications, reprinted with corrections [1958] ed.). New York: Cambridge University Press. ISBN 978-1-108-00153-3. .

[12] Shannon, Claude Elwood (1940). *A symbolic analysis of relay and switching circuits* (http://hdl.handle.net/1721.1/11173). Cambridge: Massachusetts Institute of Technology. .

[13] "National Inventors Hall of Fame – George R. Stibitz" (http://www.invent.org/hall_of_fame/140.html). 20 August 2008. . Retrieved 5 July 2010.
[14] "George Stibitz : Bio" (http://stibitz.denison.edu/bio.html). Math & Computer Science Department, Denison University. 30 April 2004. . Retrieved 5 July 2010.
[15] "Pioneers – The people and ideas that made a difference – George Stibitz (1904–1995)" (http://www.kerryr.net/pioneers/stibitz.htm). Kerry Redshaw. 20 February 2006. . Retrieved 5 July 2010.
[16] "George Robert Stibitz – Obituary" (http://ei.cs.vt.edu/~history/Stibitz.html). Computer History Association of California. 6 February 1995. . Retrieved 5 July 2010.

References

- Sanchez, Julio; Canton, Maria P. (2007). *Microcontroller programming: the microchip PIC*. Boca Raton, FL: CRC Press. p. 37. ISBN 0-8493-7189-9.

External links

- A brief overview of Leibniz and the connection to binary numbers (http://www.kerryr.net/pioneers/leibniz.htm)
- Binary System (http://www.cut-the-knot.org/do_you_know/BinaryHistory.shtml) at cut-the-knot
- Conversion of Fractions (http://www.cut-the-knot.org/blue/frac_conv.shtml) at cut-the-knot
- Binary Digits (http://www.mathsisfun.com/binary-digits.html) at Math Is Fun (http://www.mathsisfun.com/)
- How to Convert from Decimal to Binary (http://www.wikihow.com/Convert-from-Decimal-to-Binary) at wikiHow
- Learning exercise for children at CircuitDesign.info (http://www.circuitdesign.info/blog/2008/06/the-binary-number-system-part-2-binary-weighting/)
- Binary Counter with Kids (http://gwydir.demon.co.uk/jo/numbers/binary/kids.htm)
- “Magic” Card Trick (http://gwydir.demon.co.uk/jo/numbers/binary/cards.htm)
- Quick reference on Howto read binary (http://www.mycomputeraid.com/networking-support/general-networking-support/howto-read-binary-basics/)
- Binary converter to HEX/DEC/OCT with direct access to bits (http://calc.50x.eu/)
- From one to another number system (https://www.codeproject.com/Articles/350252/From-one-to-another-number-system/), *article related to creating computer program for conversion of number from one to another number system with source code written in C#*
- From one to another number system (https://sites.google.com/site/periczeljkosmederevoenglish/matematika/conversion-from-one-to-another-number-system/From one to another number system.zip?attredirects=0/), *free computer program for conversion of number from one to another number system written in C#, it is necessary .NET framework 2.0*
- From one to another number system (https://sites.google.com/site/periczeljkosmederevoenglish/matematika/conversion-from-one-to-another-number-system/Solution with source code From one to another number system.zip?attredirects=0&d=1/), *full solution with open source code for conversion of number from one to another number system written in IDE SharpDevelop ver 4.1, C#*

Article Sources and Contributors

Two's complement *Source*: http://en.wikipedia.org/w/index.php?oldid=542375324 *Contributors*: !melquiades, 2001:44B8:41AB:A600:B9C7:F091:1D3:DD2E, 4pq1injbok, AS, Aberglaube, Abhineetnazi, Addps4cat, Adrianwn, Aesopos, Ahy1, Ailurophobia, Ajblue98, Amit man, Andrei Stroe, Aninhumer, Anonymous Dissident, Apurba saitech, Arnies, ArnoldReinhold, Ashenai, Avono, BD2412, BazookaJoe, Bdesham, BeIsKr, BenFrantzDale, Bender235, BiT, Bkkbrad, Bolisho, Booyabazooka, Brian Kendig, C xong, Cardboardbox, Charles Matthews, Chelmite, Chinju, Chris Roy, ChrisGualtieri, Copyeditor42, Couturier, Cybercobra, DARTH SIDIOUS 2, Dan Granahan, David-Sarah Hopwood, DavidCary, Davron, Dcoetzee, Deadkid dk, Decrypt3, Dicklyon, Dirk gently, Disavian, Discospinster, Dissident, Dysprosia, ESkog, Eeekster, EnJx, EncMstr, Eregli bob, Eric119, Escape Orbit, Etu, Evergreen9, Firoz Pervez, Frencheigh, Fresheneesz, Giftlite, Gnowor, GraemeMcRae, Graham87, Grover cleveland, Halo2, Hao2lian, HappyDog, Hephaestos, Highway Hitchhiker, Holger Blasum, Horacelamb, Hydryad, I dream of horses, Iain.mcclatchie, Incnis Mrsi, J.delanoy, J04n, Jackelfive, Jake Nelson, Jangirke, Jasonbdaniels, Jcdonelson, Jmmonde, Jostikas, Jruderman, Jth299, Kbdank71, Kcarnold, Kingpin13, Klickagent, Kvng, LOL, Laogeodritt, Lavajoe, Loadmaster, LuisFilipeM, Luna Santin, Lyoko is Cool, Machine Elf 1735, Magioladitis, Markus Kuhn, Martin451, Maximaximax, Maxis ftw, McSly, Mekong Bluesman, Melchoir, Michael Hardy, Mindbuilder, Mindmatrix, Miquonranger03, Mitjak, Mrscrith, Mysid, NMocho, Nathanael Bar-Aur L., Nilfanion, Nornagon, Nwbeeson, Nxavar, OsamaBinLogin, Paul August, Pengo, Phgao, Pkrecker, Plugwash, Positivecharge, Prashantgonarkar, Quuxplusone, Qwerty271828, Rahulchandra, Reaper Eternal, Red Thrush, RedWolf, Redfearnb, RexNL, Richellis333, Rick Sidwell, Robert K S, Rohanmittal, Rootless, Ryk, Scottcraig, Sdedeo, Shmoib, Stalinbulldog, Supaari, SwirlBoy39, Synergy, Tata2007, Teimu.tm, The Thing That Should Not Be, Thecheesykid, Tobias Bergemann, Toothgrinder, Trotter, Uncompetence, Vadmium, Vic226, Walor, Wapcaplet, Wasdavid, WimdeValk, Ximalas, Yamamoto Ichiro, Yaron K., Yayay, Zanzabarr80, Zero sharp, ZeroOne, Zhjesse, Zundark, 488 anonymous edits

Ones' complement *Source*: http://en.wikipedia.org/w/index.php?oldid=540708703 *Contributors*: 2620:0:1000:1301:BE30:5BFF:FEDB:2AE2, Bajsejohannes, BiT, Black.jeff, Bobrayner, CBM, ChrisCPearson, Cybercobra, DavidCary, Delusion23, Demonkoryu, Derschmidt, Deryck Chan, Dmcq, Entomy, Gandalf61, GermanX, Gwen-chan, Johnuniq, KentOlsen, Mgnt, Michael Hardy, Mirzasehr, Mormegil, NawlinWiki, Nullzero, Oleg Alexandrov, Pageman, Pie4all88, Prashantgonarkar, Tgeairn, The Interior, Tuhertz, Vadmium, WestwoodMatt, 26 anonymous edits

Binary-coded decimal *Source*: http://en.wikipedia.org/w/index.php?oldid=540878791 *Contributors*: 137.111.131.xxx, AVRS, Acerperi, Adi.mmmec, AgadaUrbanit, Ale jrb, Alphadriven, AndrewHowse, Anomie, Arabic Pilot, ArnoldReinhold, Audriusa, Barvinok, Behco, Bigdumbdinosaur, Blahma, Bobfran, Brian0918, BrucebWiki, CRGreathouse, Camw, ChadCloman, Choster, Ciaran H, Ciphers, Clone53421, Comet--berkeley, Conversion script, DARTH SIDIOUS 2, DMG413, DanielEng, Davehi1, Dcoetzee, Dicklyon, Dragana666, Dzubint, Eliz81, EncMstr, FlyHigh, Fresheneesz, Fswarbrick, Furrykef, GRAHAMUK, Gazilion, Ghettoblaster, Giftlite, Gilliam, GoingBatty, Hashar, Henrygb, Igiffin, Intgr, JaGa, Jaw2jaw, JayC, Jef-Infojef, Jeh, Jim.belk, Joe Decker, Kaini, Kbdank71, Keka, Kencf0618, Kevleyski, Kri, Krischik, Krypper, Kvdveer, Kwamikagami, Lbs6380, Leuko, Loadmaster, Lugia2453, Matusz, Mcapdevila, MelbourneStar, Mfc, Michael Hardy, Mikhail Ryazanov, Miraceti, Mirror Vax, Mr Elmo, MrOllie, Murray Langton, Musiphil, Neelix, Nimur, Niteowlneils, Nnp, Obradovic Goran, Octahedron80, Oli Filth, PacoMarkE, Piano non troppo, Plugwash, Pointillist, Potaco99, Qllach, Quota, R. S. Shaw, RTC, Raidon Kane, Reach Out to the Truth, RexNL, Rich Farmbrough, Rominandreu, Rwwww, Sabri76, Securiger, Shadowjams, ShelfSkewed, Shlomital, SimonP, SimonTrew, Sobreira, Speight, Superm401, Swerdnaneb, Tesi1700, Tharos, Theresa knott, Thevenerablez, Tim Parenti, Tombomp, Tsunanet, Vadmium, VampWillow, Vrenator, Vwollan, Warut, Wbm1058, Wikiborg2, Wstorr, X!, ZeroOne, كاشف عقيل, आशीष भटनागर, 253 anonymous edits

Gray code *Source*: http://en.wikipedia.org/w/index.php?oldid=542555463 *Contributors*: A876, AJP, Acerperi, Ahoerstemeier, Ahseaton, AileTheAlien, Andy Dingley, Animagi1981, BAxelrod, BD2412, Bender235, Bggoldie, Bobblehead, Bogdangiusca, BrotherE, Bubba73, Bunyk, Cburnett, Charles Matthews, Charvest, Cholling, Chris the speller, ChrisEich, CosineKitty, Cyp, David Eppstein, David.Monniaux, DavidCary, Daxx wp, Dcoetzee, Deacs33, Demi, Denelson83, Dicklyon, Dina, Dispenser, Djfeldman, Dr0b3rts, Droll, Edratzer, Egg, El C, Elias, Elphion, Emvee, Fcdesign, Fresheneesz, Gerbrant, Giftlite, Glrx, Gnorthup, GoingBatty, Haham hanuka, Hariva, Heron, Hgrosser, Hv, Iamfscked, Inductiveload, Iridescent, Isaac Dupree, Isiah Schwartz, JLaTondre, Jaiguru42, Jan olieslagers, Jason Recliner, Esq., Jeff 113, Jim1138, Jjbeard, Johnmorgan, Johnuniq, Jos.koot, Jturkia, Kanesue, Kateshortforbob, Kilom691, Kjkjava, Law, Lbaralgeen, Leonard G., LilHelpa, Linas, MER-C, MSGJ, Machine Elf 1735, MarkSweep, Mate2code, MattGiuca, Matusz, Max711, Maxal, Maxalot, Mellum, Michael Hardy, Mike1024, Mild Bill Hiccup, MooMan1, MrOllie, MyrddinE, Nick Pisarro, Jr., Nikevich, Nneonneo, Noe, Oashi, Ocatecir, Ohiosoil, Pacifier, Pakaran, Pgimeno, PierreAbbat, Piet Delport, Plamka, Pleasantville, Plugwash, Prashantgonarkar, PuerExMachina, Qwyrxian, RTC, RayKiddy, Requestion, Rich Farmbrough, Ricky81682, Rjwilmsi, RobH (2004 account), Ronz, Sakurambo, SciCompTeacher, Sciyoshi, Seantellis, Seraphimblade, Shellreef, Sjock, Smalljim, Snowolf, Sun Creator, SuneJ, Suruena, Svick, Tedickey, TheNightFly, Tomo, TutterMouse, Vanish2, Vanka5, Wapcaplet, Westley Turner, Wikkrockiana, Winzurf, Wwoods, XJamRastafire, Yahya Abdal-Aziz, Yoshigev, Yworo, ZeroOne, Zeycus, 210 anonymous edits

Hexadecimal *Source*: http://en.wikipedia.org/w/index.php?oldid=542146703 *Contributors*: 25or6to4, A D Monroe III, A4, AJRobbins, Acroterion, Adhemar, Aditya, Ae.davies1992, Aforencich, AgentPeppermint, Ahoerstemeier, AirdishStraus, Ajraddatz, Alansohn, Alekjds, Alethiareg, AlexWaelde, Alfio, Alfvaen, Andre Engels, AndreCapaGarcia, Andrejj, Android Mouse, Angela, AnnaFrance, Anomie, Anwar saadat, Arthur Rubin, Ashenai, Attys, Aulis Eskola, AxelBoldt, B4hand, Barium, Bart133, Bearboat, Belamp, Beland, BenRG, Beofluff, Bernard Ladenthin, Bevo, Binadot, Biot, BlakeCS, BlastOButter42, Blobglob, Bobo192, Bodinagamin, Bongwarrior, Bornhj, Bufdaemon, Bunnyhop11, Calmer Waters, Calmypal, Can't sleep, clown will eat me, Carre, Cassivs, Chridd, Chris 73, Chris j wood, Chris the speller, Chrislk02, Chubbles, CloneDeath, Colonies Chris, Concordia, Conversion script, Cool halo 2, Copyeditor42, Corpcon, Courcelles, Cronholm144, Curps, Cyp, D, DV8 2XL, Daggerstab, Dantheman4114, DarkJXD, Darklilac, DarthGanon, Daverocks, David Woodward, Dcoetzee, Defproc, DerHexer, Discospinster, Divineword, Dmillard10, Don4of4, DookieDungeon, Doradus, Douglas W. Jones, Dr.Luke.sc, Drphilharmonic, Duoservo, DylanW, Dzogchenpa, ESkog, Easyguyevo, Edward, Egil, Elektron, Elium2, Elockid, Elphion, EncMstr, Equazcion, Eric119, Errorx666, Excirial, Exert, Extransit, Ezeu, Fastilysock, Felixaldonso, Fireaxe888, FlyingPenguins, Fran Rogers, FrankHamersley, Fredrik, Fubaz, Fudoreaper, Furrykef, Gandalf44, Gclinkscales, Gene Nygaard, Gerbrant, Gfoley4, Ghettoblaster, Giftlite, Gindar, Glenn, Glenn L, Gniw, Golbez, GorillazFanAdam, Graham87, Guy M, Guyalsfere, HTML2011, HYC, Haakon, Hamerbro, Hanacy, Hans Adler, Hauptmech, Havarhen, HenkeB, Henkt, Heron, Hibou8, Hirzel, Hmrox, Huppybanny, IMSoP, IanOsgood, Idefix76, Incnis Mrsi, Indiana State, Inkypaws, IntrigueBlue, Isarra (HG), J'raxis, J.delanoy, J7, JIP, JTN, JaGa, James175, Jao, Jeronimo, Jerryobject, Jerryseinfeld, Jh51681, Jiddisch, Jndrline, JoanneB, John Vandenberg, Johnuniq, Josh Parris, Jovianeye, Jur123, Jvr725, KMcD, Kaihsu, Karl E. V. Palmen, Karl Palmen, Kbdank71, Keka, Kelly Martin, Kevmitch, Kim Bruning, Kluonius, Ktsquare, Kukini, Kuru, Lament, Lanukkunal, LeoNomis, Leotolstoy, Liftarn, Light current, Lightminute, Linas, Livajo, LobStoR, Lotje, Lugia2453, M.O.X, MBerrill, MER-C, MK8, Mac, Maelnuneb, Magister Mathematicae, Malo, MarkSweep, Marnanel, Masciare, Masterofpsi, Match 213, Mate2code, MathsIsFun, MattGiuca, MatthewMastracci, Mausy5043, Meaghan, Meco, Mellum, Memorized128, Mendalus, Michael Hardy, MichaelBillington, Mike Rosoft, Minesweeper, Minna Sora no Shita, Modest Genius, Mooiehoed, Moonwolf24, Mr.briancochran, Mschel, Msikma, Mtruch, Munthe, Myanw, Mythsearcher, Nageh, Nateho, NawlinWiki, Nharipra, Nickptar, Nicolas1981, Nihiltres, Noe, Nominaladversary, Norm mit, NrDg, Nucleosynthesis, Nø, OKeh, Oalders, Obradovic Goran, Oleg Alexandrov, Oli Filth, Omegatron, Opelio, Orderud, Oxymoron83, PCHS-NJROTC, PZ, Pakaran, Palfrey, Panchoy, Pascal666, Patrick, Paul August, Paul Martin, Paul Stansifer, Paulmmn, Pausch, Peter 2005, Pharaoh of the Wizards, Philip Trueman, Phluid61, PierreAbbat, Pkchan, Plugwash, Pmanderson, Poweroid, Prodego, Pseudomonas, Psr12, Quarkington, Quarl, Quercus solaris, R3m0t, RJASE1, RTC, Radix, Radon210, Raul654, Rbonvall, RedWolf, Reswobslc, Rettetast, ReyBrujo, Rfsmit, Ricardo Cancho Niemietz, Rich Farmbrough, Rintrah, Robbe, Robert, Robo37, Rorro, Rrburke, Rts.bn.vs, Rwessel, Rünno, S91by, Saraphim, Scepia, Scottmsg, Sdoking, Sege1701, Sesu Prime, Shadow1, Shishirmital, SimonP, Sjakkalle, Slady, Sligocki, Smart people USA, Smaug123, SoCalSuperEagle, Specs112, Spiff, Splibubay, Splintax, Spyforthemoon, Srleffler, Starbuck-2, StaticVision, Stephen B Streater, Stephenb, Stmrlbs, StuartBrady, StuartMurray, Super-Magician, Swpb, TAKASUGI Shinji, THEN WHO WAS PHONE?, TakuyaMurata, Tassedethe, Tcsetattr, Techman224, Ted Longstaffe, Tenbaset, Terrorist of bush, The Anome, The Mysterious Gamer, The Son of Man, The zoro, TheStarman, Thewikicontributor, Thingg, Think outside the box, Thorwald, Thrane, Tide rolls, Tikiwont, Timmeh77, TobyDZ, Tomasf, Tomdobb, Tony Fox, Trang Oul, Trevorparsons, Tripodics, Trishm, Triviator, Ttam, TwilligToves, Uaxuctum, UberScienceNerd, Unkx80, Unused0022, Uriyan, Vadmium, VampWillow, Versus22, Vsion, Vwollan, Waldir, Wapcaplet, Wayfarer, Waylonbutler, Weeliljimmy, Wereon, Wickey-nl, Wiki alf, Wimt, Wizardist, Wknight94, Wrp103, Wutsje, Wyatt915, Wysprgr2005, Wzwz, Yksyksyks, Zac439, Zfr, Zzedar, ماني, 에멜무지로, 824 anonymous edits

Octal *Source*: http://en.wikipedia.org/w/index.php?oldid=542871073 *Contributors*: 198.92.68.xxx, 28bytes, ATMarsden, Abeam89, Adam1213, Ahruman, Alansohn, Angr, Arthur Rubin, Billinghurst, Binksternet, Bjankuloski06en, BjörnBergman, Black Falcon, Bluerasberry, Bobyllib, Burn, CBM, CJLL Wright, Carson grey, Carsrac, CattleGirl, Caue.cm.rego, Chadders, Colonies Chris, CondeNasty, Conversion script, Cyp, Damicatz, Dan Hoey, Delirium, Der.Archivar, Dirac1933, Dodgerdave, Donfbreed, Douglas W. Jones, DrZarkov, Dwandelt, Eakin, EdBever, Egil, Elphion, Evil Monkey, Fabimator, Fanatix, Fastfission, Feezo, Frap, FrenchIsAwesome, Furrykef, Giftlite, Graham87, Grenavitar, Hpa, Icairns, Incnis Mrsi, Jag164, Jjunken, Jonathan Baker, Karl E. V. Palmen, Kbdank71, Keka, Kjoonlee, Kungfuadam, Kuru, Laurentius, Letter Ezh, Lexusuns, Lfcdolbear, Librarian2, LolJIZMASTERlol, Lucas.Yamanishi, MER-C, Madmardigan53, Magetoo, Markjreed, Marlow4, MatthewMastracci, Meco, Mendalus, Merovingian, Michael Angelkovich, Michael Hardy, Mika1h, Mild Bill Hiccup, MrOllie, Mulad, Mwtoews, Myrvin, Nevyn, Nic.stage, Nixdorf, Noe, Nonagonal Spider, OKeh, Omicronpersei8, Orphan Wiki, Paul August, PhilKnight, Pictureuploader, Pinethicket, Pmlineditor, Pne, Pseudoanonymous, Pseudomonas, RTC, Rachel1, Raistlin11325, RedWolf, Regenspaziergang, Reisio, Robo37, Rodrigo Novaes, Rorro, RoseParks, SebastianHelm, SimonP, Sirius nst, Skalman, Slowking Man, Stephen MUFC, TAKASUGI Shinji, Tgeairn, The dom martin, Thewikicontributor, Thorwald, Tobias Bergemann, Trackstar789, Tristostan, Tunguuz, TwoTwoHello, Ugncreative Usergname, Vadmium, Vishnava, Wapcaplet, Wernher, Whoever blocks me has no life!, Wikitiki89, YaltaC, 296 anonymous edits

Binary number *Source*: http://en.wikipedia.org/w/index.php?oldid=542231905 *Contributors*: .Absolution., 1exec1, 2001:630:63:192:358C:C707:2254:3889, 2602:306:CDC3:90B0:57:C3BE:A533:28E2, 28421u2232nfenfcenc, 2D, 2andrewknyazev, 4, 4twenty42o, 999retard, A.kamburov, ALE!, Abc518, Abu-Fool Danyal ibn Amir al-Makhiri, Adashiel, Addshore, AdjustShift, Aerographer1981, Ahoerstemeier, Airplaneman, Aitias, Alansohn, Aleniko17, AlexJ, Alexf, Amesville, Amorymeltzer, Amplitude101, Andrejj, AndrewKepert, Andy Dingley, Andypandyjay, Anomie, Anonymous Dissident, Anonymous editor, Antandrus, Apteva, Arabic Pilot, Arichnad, Arjun024, Arthur Rubin, Asdfg1234, Assassin15, Astral, AubreyEllenShomo, Auroranorth, Aursani, Avant Guard, Avazelda13, Azcolvin429, Azreal Umbra, Aztects, Baa, Baiji, Bart133, Baseball Watcher, Bbourne20, Beetstra, Belovedfreak, Ben Kidwell, Bigbluefish, Bjankuloski06en, Blackworm, BlaenkDenum, Blanche of King's Lynn, Blue520, Bluerasberry, Bobo192, Bookandcoffee, Brews ohare, Britannic124, Bulletd, Buzzlite101, C. A. Russell, CWenger, CWii, Calmer Waters, Caltas, Camw, Can't sleep, clown will eat me, Capitalist, Capricorn42, CaribDigita, Carsrac, Castedo, Catinator, Causa sui, Chairman S., Charles Matthews, CharlesDexterWard, Charleschuck, Chris 73, Chrislk02, Christian List, Chromaticity, Cimorcus, Citizen, CityOfSilver, Closedmouth, Cncmaster, Cometstyles, ContinueWithCaution,

Controls.freq, Courcelles, Crohnie, Cronholm144, CryptoDerk, Cureden, Cutiepie17881, D. F. Schmidt, DVdm, Da nuke, Daganboy, Dan6hell66, Daniel Quinlan, DanielCD, Dante Shamest, Darth Panda, David Eppstein, David McIlvenna, David n m bond, DavidCary, Dcljr, Dcoetzee, Debresser, Deeptrivia, Deggert, Dejan Jovanović, Delirium, DeniabilityPlausible, DerHexer, Derouch, Dicklyon, Dkleeman, Dodgerdave, Dodo bird, Dogcow, Dorkenhavvon, Doug Bell, Drilnoth, Drmies, Dryman, Dungodung, Dysepsion, Dysprosia, E0steven, EEng, ESkog, Ed Poor, EdBever, EdC, Edmarriner, Egmontaz, El C, Elegost5555, Epbr123, Erkan Yilmaz, Euryalus, Evercat, Excirial, Fabio479, Falcon8765, FallingGravity, Farosdaughter, Flewis, Florian Blaschke, Fredrik, Fresheneesz, Fritz Jörn, Frozenevolution, Fæ, GRAHAMUK, Gandalf61, Gav89, Gco, Gdo01, Giftlite, Gilliam, Gimboid13, GinaDana, Glane23, Gogo Dodo, GorillazFanAdam, Grafen, GregAsche, Gregbard, GroveGuy, Gurch, Gurt Posh, Gwalla, Gwen Gale, Gwernol, Haakon, Haham hanuka, Hakufu Sonsaku, HalfShadow, HallwayGiant, Ham Pastrami, Hans Adler, Hatredto, Headbomb, Helix84, HexenX, Honza Záruba, Hrishikesh0111, II MusLiM HyBRiD II, Iamunknown, Includeiostream, Incnis Mrsi, Infaredz, Infinity0, Insanephantom, Interiot, InverseHypercube, Iridescence, Iridescent, Ishan.beckham, J.delanoy, JForget, JNW, JSR, JaGa, Jaan513, Jackfork, Jacob grace, Jacob.jose, Jafet, Jagged 85, JamesBWatson, Jaq2013, Jarek Duda, Jasper Deng, Jdk42, Jeffreyarcand, JesseW, JiFish, Jim.belk, Jimpaz, Jiri 1984, Jmabel, Joel B. Lewis, JoeliusCeasar, Jogers, Johnsmitzerhoven, Jon Awbrey, Jonatan Lindstrom, Jonathan de Boyne Pollard, Jonik, Jorgepblank, Josh Parris, Jshadias, Jstew87, Juliancolton, Junglecat, Jusdafax, Justin W Smith, Jwoodger, Kappa, Karl Palmen, Katalaveno, Kbdank71, Kbh3rd, Kbrose, Keka, Kevinalewis, Kim Bruning, Kingo1234, Kingpin13, Kirill Lokshin, KnowledgeOfSelf, Kotra, Krackpipe, Kraftlos, Krishnaupas, Kubigula, Kukini, Kuru, Kurykh, LLarson, Lambiam, Legalboard, LiDaobing, Liempt, LightAnkh, Linas, LinkWalker, Linuxwikiuser, Logan, LokiClock, Lost, Luigi30, Lupo, MER-C, Machine Elf 1735, Macjohn2, Majopius, Majorly, Malo, Maniaque27, Marco Polo, Margin1522, MarkKB, Martarius, Martinap98, Martinman11, Marudubshinki, Mate2code, Math MisterY, MathsIsFun, MatthewMastracci, Mblumber, Mckoch, Meco, Meekywiki, Mendalus, Meno25, MetroMan4, Mets501, Mhdc2003, Michael Hardy, Michael93555, Michaeldadmum, Midnight Madness, Mike4ty4, Mikeo, Mild Bill Hiccup, Milogardner, Mipadi, Miquonranger03, Mlpearc, Monty845, Mouse Nightshirt, Mr Stephen, Mr. Stradivarius, MrOllie, MrPrada, Mschlindwein, MuZemike, Mushroom, Myanw, N5iln, Naohiro19, Nedge123, Nerd272, Netralized, NewEnglandYankee, Newton2, Ninly, Nitomatik, Nivix, Nixdorf, Noe, Noetica, NoobTheShow24, NrDg, Nsaa, Numbo3, Nø, OKeh, Ohconfucius, Ohnoitsjamie, Oleg Alexandrov, Omegatron, Opelio, Orienomesh-w, Oxfordwang, Patrick0Moran, Patstuart, Paul August, Peace keeper, Pedalist, Perić Željko, Peter 2005, Phantomsteve, Philip Trueman, Piano non troppo, Pikujs, Pinethicket, Pit, Platypus222, Poccil, Polpolpol4, PoojanWagh, Poopship75, Poor Yorick, Pooresd, Prolog, Przo, Psiphiorg, Psychotic Midget, Purgatory Fubar, Quarl, Quatrinauta, Queen Spiral, Qwerty112233, R-Joe, R. S. Shaw, R00m c, RHaworth, RJASE1, RJGray, RJaguar3, Raf1qu3, Rama, Ravi12346, Reach Out to the Truth, Regancy42, Reno171, Retired username, RexNL, Rfl, Rjanag, Rjd0060, Rjwilmsi, Rl, Rlevse, Rob Hooft, Robert K S, RobertG, Rohanpol, Rorro, RoyBoy, Rsm99833, Ruhrfisch, SCZenz, SEWilco, SQGibbon, SWAdair, Samadam, Sandeep Kalshaniya, Sandor rawks, Sango123, Scarian, Scohoust, Scottcraig, Scottywong, Shadowjams, Shanes, Shanken, Shenme, Siddhant, Sigma 7, SimonArlott, SimonP, Sir Nicholas de Mimsy-Porpington, Sjakkalle, Slakr, Smith14333, Smyth, Snehalbhai, Snoyes, SohanDsouza, Someguy1221, Sopoforic, Sorisos, Sorw, Soupystar, SpAwNaGeZ, SpacePirate59, Spangineer, Speuler, Spinningspark, Spug, SpuriousQ, Ssassddd, Stwalkerster, Sue Rangell, Suisui, Supaari, Superfrowny, Superiority, Superm401, Supertouch, Sweet xx, Syntaxerrorz, THEN WHO WAS PHONE?, TakuyaMurata, Tanweer Morshed, Tasc, Techbeats123, Technical math, The Illusive Man, The PIPE, The Rambling Man, The Thing That Should Not Be, The demiurge, The sunder king, The undertow, TheWeakWilled, Thecheesykid, Thelazyleo, Tiddly Tom, Tide rolls, Tintenfischlein, Titoxd, Tobby72, Tobias Bergemann, Tom harrison, Tombomp, Tomislavlac, Tomtad, Travelbird, TrippingTroubadour, Trovatore, Trurle, Twaz, Twsx, Ubergeekguy, UncleDouggie, Uquiffquiff, Utcursch, Vadmium, Valhalla, Vanished user x10, Velvetron, Vgy7ujm, Violetriga, Violinbecky76543, Vipinhari, Viznut, VoidLurker, Vwollan, WODUP, WadeSimMiser, Wapcaplet, Waterbender kara, WatermelonPotion, Wavelength, Wayne Slam, Wayward, Wernher, Westnest, WikiLaurent, Wikipop, Willking1979, WimdeValk, Winchelsea, Wipeoutman, WoollyMind, Wyzzard, Wzwz, Xander756, Yansa, Yayay, Yegorm, Z1nk666, Zac439, Zackfox, ZeroOne, Zinc2005, Zippokovich, Zundark, Zvn, ساجد امجد ساجد, كاشف عقيل, सुभाष राऊत, 1600 anonymous edits

Image Sources, Licenses and Contributors

Image:Binary clock.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Binary_clock.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: Alexander Jones & Eric Pierce

Image:Reflected binary Gray 2632058.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Reflected_binary_Gray_2632058.png *License*: Public Domain *Contributors*: Frank Gray

Image:US02632058 Gray.png *Source*: http://en.wikipedia.org/w/index.php?title=File:US02632058_Gray.png *License*: Public Domain *Contributors*: Original uploader was Dicklyon at en.wikipedia

Image:Encoder Disc (3-Bit).svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Encoder_Disc_(3-Bit).svg *License*: Public Domain *Contributors*: jjbeard

File:Gray code rotary encoder 13-track opened.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Gray_code_rotary_encoder_13-track_opened.jpg *License*: Public Domain *Contributors*: Mike1024

File:Binary-reflected Gray code construction.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Binary-reflected_Gray_code_construction.svg *License*: Public Domain *Contributors*: Inductiveload

File:Gray code permutation matrix 16.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Gray_code_permutation_matrix_16.svg *License*: Public Domain *Contributors*: Mate2code

Image:Enkelspoors-Graycode.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Enkelspoors-Graycode.svg *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: Gerbrant

Image:Bruce Martin hexadecimal notation proposal.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Bruce_Martin_hexadecimal_notation_proposal.png *License*: Attribution *Contributors*: Bruce A. Martin, Applied Mathematics Department, Brookhaven National Laboratory

File:Hexadecimal-counting.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Hexadecimal-counting.jpg *License*: Public Domain *Contributors*: Lipedia

Image:Hexadecimal multiplication table.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Hexadecimal_multiplication_table.svg *License*: Public Domain *Contributors*: Bernard Ladenthin

File:Bagua-name-earlier.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Bagua-name-earlier.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: Pakua_with_name.svg: 2006-09-23T21:16:47Z BenduKiwi 547x547 (101558 Bytes) derivative work: Machine Elf 1735 (talk)

File:Carus-p48-Mystic-table.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Carus-p48-Mystic-table.jpg *License*: Public Domain *Contributors*: An unknown Tibetan artist

File:Gottfried Wilhelm von Leibniz.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Gottfried_Wilhelm_von_Leibniz.jpg *License*: Public Domain *Contributors*: AndreasPraefcke, Auntof6, Beria, Beyond My Ken, Boo-Boo Baroo, Cirt, Davidlud, Ecummenic, Eusebius, Factumquintus, FalconL, Gabor, Luestling, Mattes, Schaengel89, Shakko, Svencb, Tomisti, 5 anonymous edits

Image:Half Adder.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Half_Adder.svg *License*: Public Domain *Contributors*: inductiveload

License