WIKIPEDIA

Two's complement

Two's complement is a mathematical operation on binary numbers, and is an example of a radix complement. It is used in computing as a method of signed number representation.

The two's complement of an N-bit number is defined as its complement with respect to 2^N; the sum of a number and its two's complement is 2^N. For instance, for the three-bit number 010_2, the two's complement is 110_2, because $010_2 + 110_2 = 1000_2 = 8_{10}$ which is equal to 2^3. The two's complement is calculated by inverting the bits and adding one.

Two's complement is the most common method of representing signed integers on computers,[1] and more generally, fixed point binary values. In this scheme, if the binary number 010_2 encodes the signed integer 2_{10}, then its two's complement, 110_2, encodes the inverse: -2_{10}. In other words, to reverse the sign of most integers (all but one of them) in this scheme, you can take the two's complement of its binary representation.[2] The tables at right illustrate this property.

Three-bit signed integers

Decimal value	Two's-complement representation
0	000
1	001
2	010
3	011
−4	100
−3	101
−2	110
−1	111

Eight-bit signed integers

Decimal value	Two's-complement representation
0	0000 0000
1	0000 0001
2	0000 0010
126	0111 1110
127	0111 1111
−128	1000 0000
−127	1000 0001
−126	1000 0010
−2	1111 1110
−1	1111 1111

Compared to other systems for representing signed numbers (*e.g.,* ones' complement), two's complement 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 as the output, and any overflow beyond those bits is discarded from the result). This property makes the system simpler to implement, especially for higher-precision arithmetic. Unlike ones' complement systems, two's complement has no representation for negative zero, and thus does not suffer from its associated difficulties.

Conveniently, another way of finding the two's complement of a number is to take its ones' complement and add one: the sum of a number and its ones' complement is all '1' bits, or $2^N - 1$; and by definition, the sum of a number and its *two's* complement is 2^N.

Contents

History

The method of complements had long been used to perform subtraction in decimal adding machines and mechanical calculators. John von Neumann suggested use of two's complement binary representation in his 1945 *First Draft of a Report on the EDVAC* proposal for an electronic stored-program digital computer.[3] The 1949 EDSAC, which was inspired by the *First Draft*, used two's complement representation of binary numbers.

Many early computers, including the CDC 6600, the LINC, the PDP-1, and the UNIVAC 1107, use ones' complement notation; the descendants of the UNIVAC 1107, the UNIVAC 1100/2200 series, continued to do so. The IBM 700/7000 series scientific machines use sign/magnitude notation, except for the index registers which are two's complement. Early commercial two's complement computers include the Digital Equipment Corporation PDP-5 and the 1963 PDP-6. The System/360, introduced in 1964 by IBM, then the dominant player in the computer industry, made two's complement the most widely used binary representation in the computer industry. The first minicomputer, the PDP-8 introduced in 1965, uses two's complement arithmetic as do the 1969 Data General Nova, the 1970 PDP-11, and almost all subsequent minicomputers and microcomputers.

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 negative 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 (=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, first, the bits are inverted, that is: 0 becomes 1 and 1 becomes 0:

$$1111\ 1010_2$$

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

$$1111\ 1011_2$$

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, except in one special case. For example, inverting the bits of −5 (above) gives:

$$0000\ 0100_2$$

And adding one gives the final value:

$$0000\ 0101_2$$

Likewise, 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).

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 is an 'extra' negative number for which two's complement does not give the negation, see § Most negative number below.

Subtraction from 2^N

The sum of a number and its ones' complement is an N-bit word with all 1 bits, which is (reading as an unsigned binary number) $2^N - 1$. 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 (reading it as an unsigned

binary number) of 2^N. Hence, in the unsigned binary arithmetic the value of two's-complement negative number x^* of a positive x satisfies the equality $x^* = 2^N - x$.[4]

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

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

Hence, with $N = 4$:

$$x^* = 2^N - x = 2^4 - 5_{10} = 16_{10} - 5_{10} = 10000_2 - 0101_2 = 1011_2$$

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 (MSB) until the first 1 is reached; then copy that 1, and flip all the remaining bits (Leave the MSB as a 1 if the initial number was in sign-and-magnitude representation). This shortcut allows a person to convert a number to its two's complement without first forming its ones' complement. For example: in two's complement representation, the negation 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

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 2-byte variable), the most-significant bit must be repeated in all the extra bits. Some processors do this in a single instruction; on other processors, a conditional must be used followed by code to set the relevant bits or bytes.

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

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

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, width extension and right shifting are done differently for signed and unsigned numbers.

Most negative number

With only one exception, starting 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.

Taking 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 is −128. Although the expected result from negating −128 is +128, there is no representation of +128 with an 8 bit two's complement system and thus it is in fact impossible to represent the negation. Note that the two's complement being the same number is detected as an overflow condition since there was a carry into but not out of the most-significant bit.

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

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

This phenomenon is fundamentally about the mathematics of binary numbers, not the details of the representation as two's complement. Mathematically, this is complementary to the fact that the negative of 0 is again 0. For a given number of bits k there is an even number of binary numbers 2^k, taking negatives is a group action (of the group of order 2) on binary numbers, and since the orbit of zero has order 1, at least one other number must have an orbit of order 1 for the orders of the orbits to add up to the order of the set. Thus some other number must be invariant under taking negatives (formally, by the orbit-stabilizer theorem). Geometrically, one can view the k-bit binary numbers as the cyclic group $\mathbb{Z}/2^k$, which can be visualized as a circle (or properly a regular 2^k-gon), and taking negatives is a reflection, which fixes the elements of order dividing 2: 0 and the opposite point, or visually the zenith and nadir.

The presence of the most negative number can lead to unexpected programming bugs where the result has an unexpected sign, or leads to an unexpected overflow exception, or leads to completely strange behaviors. For example,

- the unary negation operator may not change the sign of a nonzero number. e.g., −(−128) → −128.
- an implementation of absolute value may return a negative number.[5] e.g., abs(−128) → −128.
- Likewise, multiplication by −1 may fail to function as expected. e.g., (−128) × (−1) → −128.
- Division by −1 may cause an exception (like that caused by dividing by 0).[6] Even calculating the remainder (or modulo) by −1 can trigger this exception.[7] e.g., (−128) ÷ (−1) → crash, (−128) % (−1) → crash.

In the C and C++ programming languages, the above behaviours are undefined and not only may they return strange results, but the compiler is free to assume that the programmer has ensured that undefined computations never happen, and make inferences from that assumption.[7] This enables a number of optimizations, but also leads to a number of strange bugs in such undefined programs.

The most negative number in two's complement is sometimes called "the weird number", because it is the only exception.[8][9] 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 the 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 (again, by the binary value) 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 $i + j \mod 2^N = i + (j + 2^N) \mod 2^N$ any value in the set $\{j + k\,2^N \mid k \text{ is an integer}\}$ can be used in place of j.[10]

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.

Example

> *In this subsection, decimal numbers are suffixed with a decimal point "."*

For example, an 8 bit number can only represent every integer from −128. to 127., inclusive, since $(2^{8-1} = 128.)$. −95. modulo 256. is equivalent to 161. since

−95. + 256.
= −95. + 255. + 1
= 255. − 95. + 1
= 160. + 1.
= 161.

Some special numbers to note

Decimal	Binary
127	0111 1111
64	0100 0000
1	0000 0001
0	0000 0000
−1	1111 1111
−64	1100 0000
−127	1000 0001
−128	1000 0000

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

Fundamentally, the system represents negative integers by counting backward and wrapping around. The boundary between positive and negative numbers is arbitrary, but by convention 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. 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 in binary must represent −1 in decimal.[11]

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 or 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 operates as if there were an extra left-most bit of 1.

Arithmetic operations

Two's complement 4 bit integer values

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.

Addition

Adding twos-complement numbers requires no special processing even 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)
```

Or the computation of 5 − 15 = 5 + (−15)"

```
         1   (carry)
   0000 0101  (  5)
+  1111 0001  (-15)
===========
   1111 0110  (-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. The result would be correct if treated as unsigned integer.

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 whether 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 $2N$ bits to contain all possible values.[12]

If the precision of the two operands using two's complement is doubled before the multiplication, direct multiplication (discarding any excess bits beyond that precision) will provide the correct result.[13] For example, take $6 \times (-5) = -30$. First, the precision is extended from four bits to eight. Then the numbers are multiplied, discarding the bits beyond the eighth bit (as 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.[14]

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 the sign bit

if A(n-1) == 0 and B(n-1) == 1 then
    return +1
else if A(n-1) == 1 and B(n-1) == 0 then
    return -1
end

// comparison of remaining bits

for i = n-2...0 do
    if A(i) == 0 and B(i) == 1 then
        return -1
    else if A(i) == 1 and B(i) == 0 then
```

```
        return +1
    end
end

return 0
```

Two's complement and 2-adic numbers

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."[15]

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 $1 + 2 + 4 + 8 + \cdots$.[16] 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.[17] Continuity of binary arithmetical and bitwise operations in 2-adic metric also has some use in cryptography.[18]

Fractions conversion

To convert a fraction, for instance; .0101 you must convert starting from right to left the 1s to decimal as in a normal conversion. In this example 0101 is equal to 5 in decimal. Each digit after the floating point represents a fraction where the denominator is a multiplier of 2. So, the first is 1/2, the second is 1/4 and so on. Having already calculated the decimal value as mentioned above, you use only the denominator of the LSB (LSB = starting from right). As a result, we have 5/16.

For instance, having the floating value of .0110 for this method to work, one should not consider the last 0 from the right. Hence, instead of calculating the decimal value for 0110, we calculate the value 011, which is 3 in decimal (by leaving the "0" in the end, the result would have been 6, together with the denominator $2^4 = 16$ reduces to 3/8). So the denominator is 8. So, the final result is 3/8.

See also

- Division algorithm, including restoring and non-restoring division in two's-complement representations
- Offset binary
- p-adic number
- Method of complements, generalisation to other number bases, used on mechanical calculators

References

1. 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

2. David J. Lilja and Sachin S. Sapatnekar, *Designing Digital Computer Systems with Verilog*, Cambridge University Press, 2005 online (https://books.google.com/books?vid=ISBN052182866X&id=5BvW0hYhxkQC&pg=PA37&lpg=PA37&ots=l-E0VjyPt8&dq=%22two%27s+complement+arithmetic%22&sig=sS5_swrfrzcQI2nHWest75sIjgg)
3. von Neumann, John (1945), *First Draft of a Report on the EDVAC* (http://web.mit.edu/STS.035/www/PDFs/edvac.pdf) (PDF), retrieved February 20, 2021
4. 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).
5. "Math" (http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html). Java Platform SE 7 API specification.
6. Regehr, John (2013). "Nobody Expects the Spanish Inquisition, or INT_MIN to be Divided by -1" (https://blog.regehr.org/archives/887). *Regehr.org* (blog).
7. Seacord, Robert C. (2020). "Rule INT32-C. Ensure that operations on signed integers do not result in overflow" (https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow). *wiki.sei.cmu.edu*. SEI CERT C Coding Standard.
8. Affeldt, Reynald & Marti, Nicolas. "Formal Verification of Arithmetic Functions in SmartMIPS Assembly" (https://web.archive.org/web/20110722080531/http://www.ipl.t.u-tokyo.ac.jp/jssst2006/papers/Affeldt.pdf) (PDF). Archived from the original (http://www.ipl.t.u-tokyo.ac.jp/jssst2006/papers/Affeldt.pdf) (PDF) on 2011-07-22.
9. Harris, David; Harris, David Money; Harris, Sarah L. (2007). "Weird binary number" (https://books.google.com/books?id=5X7JV5-n0FIC&q=%22weird+number%22+binary&pg=PA19). *Digital Design and Computer Architecture*. p. 18 – via Google Books.
10. "3.9. Two's Complement" (https://web.archive.org/web/20131031093811/http://www.cs.uwm.edu/~cs151/Bacon/Lecture/HTML/ch03s09.html). *Chapter 3. Data Representation*. cs.uwm.edu. 2012-12-03. Archived from the original (http://www.cs.uwm.edu/~cs151/Bacon/Lecture/HTML/ch03s09.html) on 31 October 2013. Retrieved 2014-06-22.
11. Finley, Thomas (April 2000). "Two's Complement" (http://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html). Computer Science. Class notes for CS 104. Ithaca, NY: Cornell University. Retrieved 2014-06-22.
12. Bruno Paillard. *An Introduction To Digital Signal Processors*, Sec. 6.4.2. Génie électrique et informatique Report, Université de Sherbrooke, April 2004.
13. Karen Miller (August 24, 2007). "Two's Complement Multiplication" (https://web.archive.org/web/20150213203512/http://pages.cs.wisc.edu/~cs354-1/beyond354/int.mult.html). *cs.wisc.edu*. Archived from the original (http://pages.cs.wisc.edu/~cs354-1/beyond354/int.mult.html) on February 13, 2015. Retrieved April 13, 2015.
14. Wakerly, John F. (2000). *Digital Design Principles & Practices* (3rd ed.). Prentice Hall. p. 47. ISBN 0-13-769191-2.
15. Hakmem - Programming Hacks - Draft, Not Yet Proofed (http://www.inwap.com/pdp10/hbaker/hakmem/hacks.html#item154)
16. 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 (https://catalog.loc.gov/vwebv/search?searchCode=CALL%2B&searchArg=QA295+.H29+1967&searchType=1&recCount=25). (pp. 7–10)
17. Vuillemin, Jean (1993). *On circuits and numbers* (http://www.hpl.hp.com/techreports/Compaq-DEC/PRL-RR-25.pdf) (PDF). Paris: Digital Equipment Corp. p. 19. Retrieved 2012-01-24., Chapter 7, especially 7.3 for multiplication.
18. 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

- Two's Complement Explanation, (Thomas Finley, 2000) (https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html)
- 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

- Two's complement array multiplier JavaScript simulator (http://www.ecs.umass.edu/ece/koren/arith/simulator/ArrMlt/)