# Ones' complement

**8-bit ones'-complement integers**

| Bits | Unsigned value | Ones' complement value |
|---|---|---|
| 0111 1111 | 127 | 127 |
| 0111 1110 | 126 | 126 |
| 0000 0010 | 2 | 2 |
| 0000 0001 | 1 | 1 |
| 0000 0000 | 0 | 0 |
| 1111 1111 | 255 | −0 |
| 1111 1110 | 254 | −1 |
| 1000 0010 | 130 | −125 |
| 1000 0001 | 129 | −126 |
| 1000 0000 | 128 | −127 |

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 0s for 1s 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 inverse of the binary representations of their corresponding positive numbers. 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.

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, continue to use it, but the majority of modern computers use 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 both +0 and -0 return TRUE when tested for zero
1   0001   1110   — and FALSE when tested for non-zero.
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

*Main article: Signed 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
```

"Corner cases" arise 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 next example shows what happens when both operands are plus or minus zero:

```
  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.

# See also

- Signed number representations
- Two's complement
- IEEE floating point

# References


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

# External links

- One's complement calculator