

Integer representations

Computers store data as bits. Each bit can have one of two values: 0 or 1. A byte consists of 8 bits, so it has $2^8 = 256$ possible bit patterns.

A bit pattern does not tell us its meaning by itself. We also need to know how the bits should be interpreted.

For example, the same eight bits can represent an unsigned integer or a signed integer. An **unsigned integer** can be zero or positive. A **signed integer** can also be negative.

We will use 8-bit examples throughout this section. The same principles apply to larger groups of bits.

Unsigned integers

In ordinary decimal notation, digit positions have weights of 1, 10, 100, and so on, starting from the right.

In binary notation, the weights are powers of two:

Bit position	7	6	5	4	3	2	1	0
Weight	128	64	32	16	8	4	2	1

Bit positions are numbered from zero, starting on the right. Bit 7 is therefore the leftmost bit in a byte. It is called the **most significant bit**, because it has the greatest weight in unsigned representation.

To find the value of an unsigned binary number, add the weights of the positions containing a 1.

Example: interpret the bit pattern 10011100.

Bits:	1	0	0	1	1	1	0	0
Weights:	128	64	32	16	8	4	2	1
Value = 128 + 16 + 8 + 4 = 156								

The smallest unsigned 8-bit value is 00000000, which represents 0. The largest is 11111111:

$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

Therefore, the range of an **8-bit unsigned integer is 0 to 255**.

Programming-language names for integer types vary. A type called `unsigned short int` should not be assumed to have 8 bits.

Converting a decimal integer to binary

To convert a non-negative decimal integer to binary, repeatedly divide it by 2. At each step, record the integer quotient and the remainder.

The remainder is always 0 or 1.

Example: convert 156 to binary.

Division	Integer quotient	Remainder
$156 \div 2$	78	0
$78 \div 2$	39	0
$39 \div 2$	19	1
$19 \div 2$	9	1
$9 \div 2$	4	1
$4 \div 2$	2	0
$2 \div 2$	1	0
$1 \div 2$	0	1

Stop when the quotient becomes zero. Read the remainders **from bottom to top**:

156 in decimal = 10011100 in binary

We can check the result using the positional weights:

$128 + 16 + 8 + 4 = 156$

Binary addition and the available number of bits

Binary addition follows rules similar to decimal addition. However, a binary digit can only be 0 or 1.

$0 + 0 = 0$
 $0 + 1 = 1$
 $1 + 0 = 1$
 $1 + 1 = 10$

In the last case, we write 0 in the current position and carry 1 to the next position on the left.

Example: add 182 and 109.

```
  10110110  (182)
+  01101101  (109)
-----
  100100011  (291)
```

The correct mathematical result needs **9 bits**. It cannot fit in an unsigned 8-bit value.

If only the lowest 8 bits are retained, the stored pattern is:

00100011 (35)

This is an example of **unsigned overflow**: the mathematical result is outside the available range.

Here, retaining only 8 bits gives $291 - 256 = 35$. The extra bit on the left is called the **carry-out**.

Representing negative integers: sign and magnitude

How can we include negative numbers without adding a separate minus-sign character?

One possible approach is to reserve the leftmost bit for the sign:

- 0 means non-negative.
- 1 means negative.

The remaining seven bits store the **magnitude**, which is the absolute value of the number.

This method is called **sign-magnitude representation**.

Example: represent +5 and -5.

	Sign	Magnitude
+5:	0	0000101
-5:	1	0000101

Seven magnitude bits can represent values from 0 to 127. Therefore, the signed range is **-127 to +127**.

There are two bit patterns for zero:

00000000	(+0)
10000000	(-0)

The 256 bit patterns therefore represent only 255 different numerical values.

Sign-magnitude representation is easy to understand, but arithmetic requires special handling of the signs and magnitudes.

For example, ordinary binary addition does not directly give the correct answer for $-5 + 7$:

10000101	(-5 in sign-magnitude representation)
+ 00000111	(+7)

10001100	(-12 in sign-magnitude representation)

The expected result is +2. To get it, an arithmetic procedure for sign-magnitude numbers must account for the signs and subtract the magnitudes.

Two's complement representation

A widely used alternative for signed integers is **two's complement representation**.

Its main advantage is that you can add positive and negative integers using the same binary addition

procedure as with unsigned integers.

Let us first see what bit pattern would make this possible.

We want:

$$-5 + 5 = 0$$

When only 8 bits are retained, the value 256 becomes 0 because its binary representation needs a ninth bit:

256 = 100000000 in binary
Lowest 8 bits: 00000000

Therefore, we can use the bit pattern for $256 - 5 = 251$ to represent -5 :

```
  11111011   (251 when interpreted as unsigned)
+ 00000101   (5)
-----
 100000000

Retain 8 bits:
  00000000   (0)
```

This is the key idea behind 8-bit two's complement representation.

For a negative integer $-m$, the unsigned value of its 8-bit two's complement pattern is $256 - m$.

For this representation, m can range from 1 to 128. Positive integers and zero use their ordinary binary patterns, with a leftmost bit of 0.

Encoding a negative integer

A convenient way to obtain the two's complement representation of a negative integer is:

1. Write its positive magnitude in binary, using the chosen number of bits.
2. Invert every bit: replace 0 with 1 and 1 with 0.
3. Add 1, keeping the chosen number of bits.

Example: encode -5 using 8 bits.

```
+5:           00000101
Invert all bits: 11111010
Add 1:         11111011
```

Therefore, **-5 is represented by 11111011.**

The number of bits matters. Always establish the width before applying this procedure.

Interpreting a two's complement bit pattern

In 8-bit two's complement representation, the positional weights are:

Bit position	7	6	5	4	3	2	1	0
Weight	−128	64	32	16	8	4	2	1

Only the leftmost bit's weight differs from the unsigned case.

Consequently:

- If the leftmost bit is 0, the number is zero or positive.
- If the leftmost bit is 1, the number is negative.

However, **the remaining seven bits do not represent the magnitude of a negative number.**

Example: interpret 11111011.

$$-128 + 64 + 32 + 16 + 8 + 2 + 1 = -5$$

Another method is to read the entire pattern as an unsigned value. If the leftmost bit is 1, subtract 256:

Unsigned interpretation:	251
Two's complement interpretation:	$251 - 256 = -5$

The interpretation changes the value; the stored bits remain the same.

Range and important bit patterns

Bit pattern	Unsigned value	Two's complement value
00000000	0	0
00000001	1	1
01111111	127	127
10000000	128	−128
11111011	251	−5
11111111	255	−1

The range of an **8-bit two's complement integer is −128 to +127.**

There is only one representation of zero. The range contains one more negative value than positive values.

In general, with n bits:

- Unsigned range: 0 to $2^n - 1$.
- Two's complement range: -2^{n-1} to $2^{n-1} - 1$.

Adding positive and negative integers

Example: calculate $-5 + 7$ using 8-bit two's complement representation.

```
  11111011  (-5)
+ 00000111  (+7)
-----
 10000010
```

Retain the lowest 8 bits:

```
00000010  (+2)
```

The result is correct.

This fixed-width calculation discards the carry-out. **A carry-out does not, by itself, indicate signed overflow.**

Two's complement makes addition convenient, but the result must still fit in the available signed range.

Signed overflow

Signed overflow occurs when the mathematical result is outside the representable signed range.

For example, 100 and 60 both fit in an 8-bit signed integer, but their sum, 160, does not:

```
  01100100  (+100)
+ 00111100  (+60)
-----
 10100000  (-96 when interpreted as two's complement)
```

Adding two positive numbers has produced a pattern representing a negative number. This indicates signed overflow.

For two's complement addition, overflow occurs when:

- Two positive operands produce a negative result.
- Two negative operands produce a non-negative result.

Adding operands with opposite signs cannot cause signed overflow.

Notice that the example has no carry-out, yet signed overflow occurs. In the earlier example, $-5 + 7$ had a carry-out but no signed overflow.

Carry-out and signed overflow describe different conditions.

These examples describe fixed-width binary arithmetic. Programming languages may handle out-of-

range calculations differently, so you must also check their rules when writing a program.

Check your understanding

1. What range can an unsigned 8-bit integer represent?
2. Why does 8-bit sign-magnitude representation have two zeros?
3. Encode -12 using 8-bit two's complement representation.
4. Interpret 11110110 as an unsigned integer and as a two's complement integer.
5. Can $+156$ be represented as an 8-bit two's complement integer?
6. What happens when $+127$ and $+1$ are added using 8-bit two's complement arithmetic?

Answers

1. 0 to 255.
2. The sign bit can be either 0 or 1 while all magnitude bits are zero.
3. $+12$ is 00001100. Inverting gives 11110011; adding 1 gives **11110100**.
4. Unsigned: **246**. Two's complement: $246 - 256 = -10$.
5. No. The greatest positive value is $+127$. The unsigned pattern for 156, 10011100, represents -100 when interpreted as two's complement.
6. The resulting pattern is **10000000**, which represents -128 . The mathematical result, $+128$, is outside the range, so signed overflow occurs.

```
.tc-lab { max-width: 900px; margin: 24px auto; padding: 24px; border: 1px solid #cbd5e1; border-radius: 16px; background: #f8f9fc; color: #172033; font: 16px/1.6 system-ui, sans-serif; }
.tc-lab * { box-sizing: border-box; }
.tc-lab h2, .tc-lab h3 { color: #172033; margin: 0 0 12px; }
.tc-lab p { margin: 10px 0; }
.tc-lab button, .tc-lab input { font: inherit; }
.tc-lab button { padding: 8px 14px; border: 1px solid #94a3b8; border-radius: 8px; background: #fff; color: #172033; cursor: pointer; }
.tc-lab button:hover { background: #e2e8f0; }
.tc-lab button:focus-visible, .tc-lab input:focus-visible { outline: 3px solid #2563eb; outline-offset: 3px; }
.tc-lab .tc-card { padding: 20px; margin-top: 18px; border: 1px solid #dbe3ee; border-radius: 12px; background: #fff; }
.tc-lab .tc-controls { display: flex; gap: 10px; align-items: center; flex-wrap: wrap; margin: 12px 0; }
.tc-lab input[type="number"] { width: 110px; padding: 7px 10px; border: 1px solid #94a3b8; border-radius: 7px; background: #fff; color: #172033; }
.tc-lab input[type="range"] { width: 100%; margin: 12px 0; }
.tc-lab .tc-bits { display: grid; grid-template-columns: repeat(8, minmax(0, 1fr)); gap: 6px; margin: 16px 0; }
.tc-lab .tc-bit-cell { text-align: center; min-width: 0; }
.tc-lab .tc-bit { display: block; width: 100%; padding: 10px 0; font: bold 25px/1.2 monospace; border: 2px solid #cbd5e1; background: #f1f5f9; }
.tc-lab .tc-bit[aria-pressed="true"] { background: #dbeafe; border-color: #2563eb; color: #1e40af; }
.tc-lab .tc-sign .tc-bit[aria-pressed="true"] { background: #ffedd5; border-color: #c2410c; color: #9a3412; }
.tc-lab .tc-weight { display: block; font-size: 13px; padding-top: 5px; color: #475569; }
.tc-lab .tc-results { display: flex; flex-wrap: wrap; gap: 12px; margin: 12px 0; }
.tc-lab .tc-result { flex: 1 1 180px; padding: 12px; border-radius: 8px; background: #eff6ff; }
.tc-lab .tc-value { display: block; font-size: 28px; font-weight: bold; }
.tc-lab .tc-mono { font-family: ui-monospace, monospace; white-space: pre-wrap; overflow-wrap: anywhere; }
.tc-lab .tc-note { color: #475569; font-size: 14px; }
.tc-lab .tc-feedback { margin-top: 12px; padding: 12px; border-left: 4px solid #64748b; background: #f1f5f9; }
.tc-lab .tc-feedback[data-kind="success"] { border-color: #15803d; background: #f0fdf4; }
.tc-lab .tc-feedback[data-kind="error"] { border-color: #b45309; background: #fff2cc; }
.tc-lab .tc-primary { background: #1d4ed8; color: #fff; border-color: #1d4ed8; }
.tc-lab .tc-primary:hover { background: #1e40af; }
.tc-lab .tc-target { font-size: 30px; font-weight: bold; color: #1d4ed8; }
.tc-lab [hidden] { display: none !important; }
@media (max-width: 520px) { .tc-lab { padding: 12px; }
```

```
.tc-lab .tc-card { padding: 12px; } .tc-lab .tc-bits { gap: 3px; } .tc-lab .tc-bit { font-size: 21px; } .tc-lab .tc-weight { font-size: 11px; } }
```

Two's Complement: Explore and Play

Eight bits can represent signed integers from **−128 to +127**. Click the bits to see how the same pattern can have different meanings.

1. Explore the bits

Signed decimal value: −128 −5 −1 0 127

The weights below the bits apply to two's complement. The leftmost bit has weight **−128**.

Two's complement value

Unsigned value

Negate: invert bits and add 1 Add 1

Try negating −5, or adding 1 to 127.

2. Build the negative number

Set the eight bits to encode this number in two's complement:

Click a bit to switch between 0 and 1. Then check your answer.

Check answer Show a hint Show solution Next number

From:

<https://edu.iit.uni-miskolc.hu/> - Institute of Information Science - University of Miskolc

Permanent link:

https://edu.iit.uni-miskolc.hu/tanszek:oktatas:techcomm:encoding_integers?rev=1791829949

Last update: 2026/10/12 18:32

