



C Integers in Binary

All values in C are represented in memory as **binary**: a sequence of 0/1 bits denoting the powers of 2 that make up a number. Decimal is our own lens of interpretation.

$$7 = 2^2 + 2^1 + 2^0 \quad \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 2^2 & 2^1 & 2^0 \\ \hline \end{array}$$

$$10 = 2^3 + 2^1 \quad \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline 2^3 & 2^2 & 2^1 & 2^0 \\ \hline \end{array}$$



C offers integers in two flavors: **signed and unsigned**. Signed ints use the **2C (two's complement's) representation**: the highest bit / MSB is 1 if negative, 0 if positive.

➤ For 2C, I think about MSB as having ⚓ negative weight, e.g., -2^7 for a 8-bit char.

$$\text{CHAR\_MIN} \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline (-2^7, -128) & -2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \hline \end{array}$$

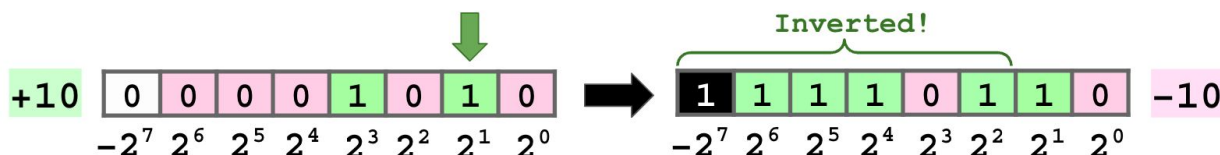
$$-1 \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline -2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \hline \end{array}$$

MSB contributes immense negativity, and no positive 1 bits to lift its spirits up 😞

MSB contributes much negativity, but just enough positivity to stay nearly afloat 😞

Casting, Two's Complements

2C Hack: To convert binary of positive integer to binary of its negative equivalent, (1) find the rightmost 1 bit, and (2) invert all bits to its left. Don't forget leading 0 bits!

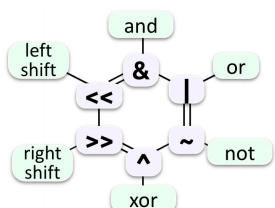


- ❑ When casting from signed to unsigned, the underlying bits don't change – just the way they're interpreted (two's complement ↔ ordinary binary)
- ❑ When comparing signed and unsigned ints, C implicitly casts signed to unsigned

Signed	-1	$\begin{array}{ c c c c c c c c } \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array}$	0xff
Unsigned	255	$\begin{array}{ c c c c c c c c } \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array}$	0xff

Bitwise Operators

- ❑ AND, OR, XOR are binary operators (2 bits as input), NOT is unary (1 bit as input)
- ❑ When applied on numbers (multiple bits), **applied to corresponding bits** in each



AND

$$\begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 0 \\ \hline \& & 1 & 0 & 1 & 0 \\ \hline \hline 1 & 0 & 0 & 0 \\ \hline \end{array}$$

OR

$$\begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 0 \\ \hline | & 1 & 0 & 1 & 0 \\ \hline \hline 1 & 1 & 1 & 0 \\ \hline \end{array}$$

XOR

$$\begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 0 \\ \hline \wedge & 1 & 0 & 1 & 0 \\ \hline \hline 0 & 1 & 1 & 0 \\ \hline \end{array}$$

NOT

$$\begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 0 \\ \hline \sim & 1 & 1 & 0 & 0 \\ \hline \hline 0 & 0 & 1 & 1 \\ \hline \end{array}$$

➔ The **AND** of two bits is **1** only if both bits are **1**, and otherwise **0**.

➔ The **OR** of two bits is **1** if either bit (or both) is a **1**, and otherwise **0**.

➔ The **XOR** (exclusive OR) of two bits is **1** if exactly one of the bits is a **1**, and **0** if both bits or neither bit is a **1**.

➔ The **NOT** of a bit is its **exact opposite**: **1** if it's a **0**, and **0** if the bit's a **1**.



More Bit Operators (Shifting)

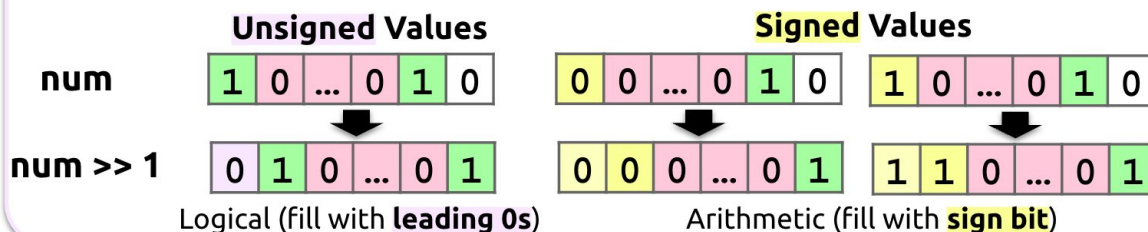


Left shifting (<<) involves sliding a bit pattern a **certain number of positions to the left** — filling in the new lower bits with 0s, and discarding any bits shifted off the left end.

$1 \ll 3 \rightarrow 1000$
3 new zero bits

$11 \ll 30 \rightarrow 11000...000$
30 new zero bits

Right shifting (>>) is similar, but slides to the right instead. However, **filling in the higher-order bits is trickier**, as we don't want to alter the sign bit / MSB.



Bitmasks to Modify Bits Strategically



To **manipulate bits in a certain way**, we often construct a help value / **bitmask**, e.g.,

Turn a bit on $\text{bit} \mid 1$

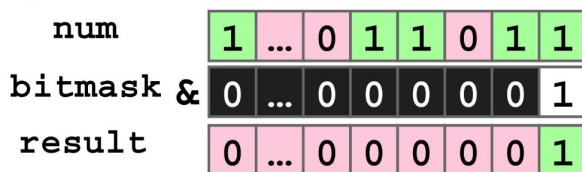
Invert/flip a bit $\text{bit} \wedge 1$

Turn a bit off $\text{bit} \& 0$

Retrieving the last bit $\text{num} \& 1$

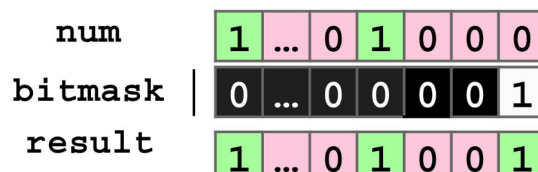
To see how a bitmask works, I recommend trying it with **concrete examples**, e.g.,

Task: Getting the lowest bit (LSB)
Bitmask: AND (&) with 1



Note that all bits are **switched off**, except for the **last one!** (preserved from original **num**)

Task: Turning on the lowest bit
Bitmask: OR (|) with 1



Note that all bits are **retained**, except the **lowest bit (LSB)** we just turned on!

Building Bitmasks



Bitmask Design

For **building bitmasks**, I recommend:

- ❖ Breaking it up in separate steps
- ❖ Constructing it **from bitmask(s) you already know** (e.g., -1 is all 1s)

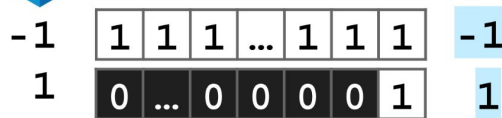
👉 On the right, some useful ones to know!

Task: Turning off the lowest 4 bits simultaneously



So a possible bitmask is AND (&) with $(-1\text{L} \ll 4)$

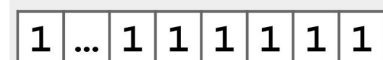
Bitmask building blocks / LEGOs



Note: 1L for signed long, 1UL for unsigned long



How do we build this mask?



Start with -1L (above) and shift 4 bits to the left!



Alternatively, the bitmask from prior slide!