CS107 Lecture 3

Bits and Bytes, Integer Representations

Reading: Bryant & O'Hallaron, Ch. 2.2-2.3 (skim)

Data Types: Then and Now

- **Early 2000s:** most computers were **32-bit.** That meant `long`s and **pointers** were **32 bits** or **4 bytes**.
- 32-bit pointers store memory addresses ranging from 0 to 2^{32} - 1, for a total of **2^{32} bytes of addressable memory**. That's **4 gigabytes**, meaning 32-bit computers could address **up to 4GB** of memory! That was a lot back then!
- Now, most computers are **64-bit.** Some data types (especially **pointers**, and oftentimes `long`s) were given more memory.
- 64-bit pointers can distinguish between addresses 0 to 2^{64} - 1, equaling **2^{64} bytes**. This equals **16 exabytes**, meaning that 64-bit computers could theoretically address up to **16 * 1024 * 1024 * 1024 GB** of memory!

Unsigned Integers

An **unsigned** integer is either 0 or some positive whole number. There is no support for negative numbers in an unsigned world.

We've already discussed the conversion between decimal and binary, and we've implicitly assumed all numbers are nonnegative.

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

$$1011_2 = 11_{10}$$

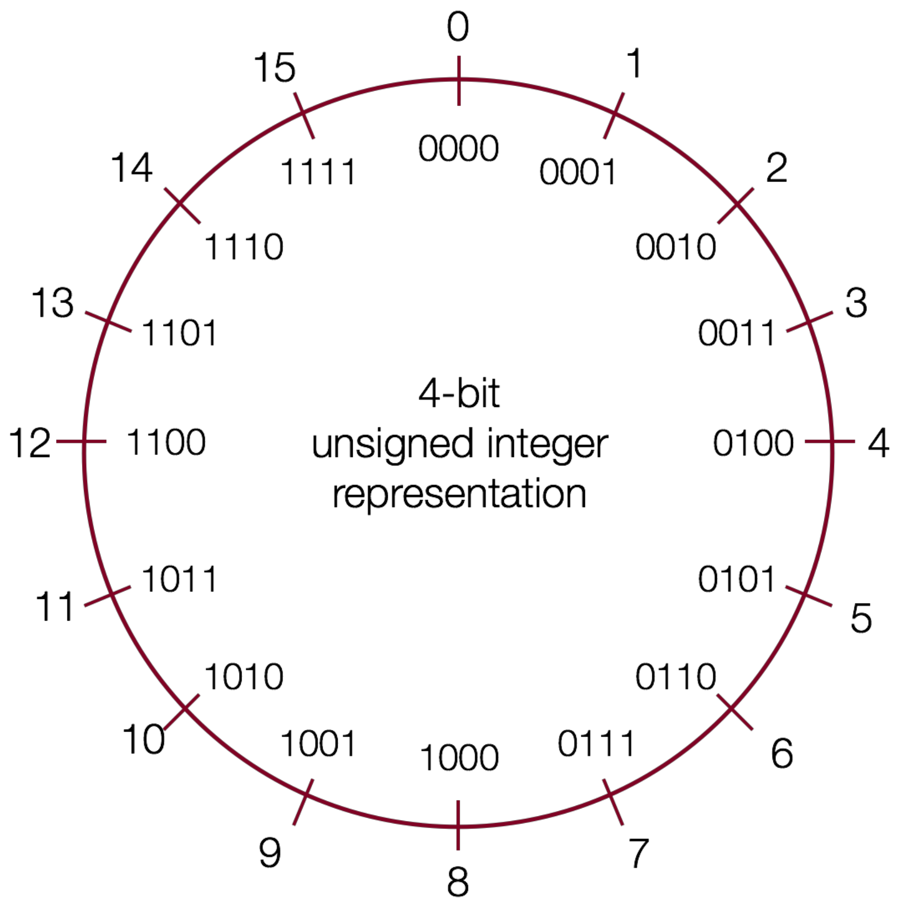
$$1111_2 = 15_{10}$$

The range of an unsigned integer is understood to be $0 \rightarrow 2^{\mathbf{w}} - 1$, where **w** is the bit count—e.g., 32-bit figures can represent 0 to $2^{32} - 1$.

Unsigned Integers

Our little number wheel to the right tells us a little about the range of values an imagined, 4-bit **`unsigned mini`** would be.

The rotary nature of the wheel implies that 15 precedes 0 in the same way that 0 precedes 1. In many ways, that's accurate.



Think of the wheel as a binary odometer that tracks a count and turns over from 1111 to 0000 in the same way that real odometers turn over from 999 999 to 000 000.

Signed Integers

A **signed** integer can be either positive, negative, or zero.

Dilemma: How do we represent both negative and positive numbers?

Proposal: use the **most significant bit** to represent sign and let all others represent magnitude.

Happy side effect: for every positive number there's a corresponding negative number. That suggests a 50/50 split between positive and negative.

Proposal: MSB represents + vs -

$$\underbrace{0}_{\text{positive}}\underbrace{110}_{6}$$

$$\underbrace{1}_{\text{negative}}\underbrace{110}_{6}$$

Proposal: MSB represents + vs -

$$\underbrace{0}_{\text{positive}}\underbrace{011}_{3}$$

$$\underbrace{1}_{\text{negative}}\underbrace{011}_{3}$$

Proposal: MSB represents + vs -

$$\underbrace{0}_{\text{positive}}\underbrace{000}_{0}$$

$$\underbrace{1}_{\text{negative}}\underbrace{000}_{0}$$

Proposal: MSB represents + vs -

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

We’re only representing 15 different values via 16 different patterns.
#sadness

Proposal: MSB represents + vs -

- **Pro:** easy to represent, and easy for humans to convert to and from decimal.
- **Con:** +/-0 is
- **Con:** we lose a bit that could be used to represent more numbers
- **Legit Con:** arithmetic is tricky: we need to find the sign, perhaps subtract (borrow and carry, etc.), maybe change the sign, maybe not. This complicates how hardware implements something as fundamental as addition. **This is the disadvantage we really care about.**

Can we do better?

Of course we can, else I wouldn't have asked.

Proposal: Optimize for Addition

Ideally, binary addition should work the same whether the numbers are positive or negative.

What pattern can be paired with 0101 so that bit-by-bit addition produces all zeros?

$$\begin{array}{r} 0101 \\ +\,???? \\ \hline 0000 \end{array}$$

Proposal: Optimize for Addition

Ideally, binary addition should work the same whether the numbers are positive or negative.

$$\begin{array}{r} 0101 \\ +\,1011 \\ \hline 0000 \end{array}$$

yes, it's really **1 0000**, but there’s no fifth bit available to store that leftmost one, so we let it fall away. The binary odometer analogy comes back.

Proposal: Optimize for Addition

Ideally, binary addition should work the same whether the numbers are positive or negative.

```
  0011
+ ????
------
  0000
```

Proposal: Optimize for Addition

Ideally, binary addition should work the same whether the numbers are positive or negative.

$$\begin{array}{r} 0011 \\ +\,1101 \\ \hline 0000 \end{array}$$

The CPU adds numbers by column-wise adding bits and throwing away anything that doesn't fit. This works for negative numbers too, without any special cases.

Proposal: Optimize for Addition

Ideally, binary addition should work the same whether the numbers are positive or negative.

```
  0000
+ ????
------
  0000
```

Proposal: Optimize for Addition

Ideally, binary addition should work the same whether the numbers are positive or negative.

$$\begin{array}{r} 0000 \\ +\,0000 \\ \hline 0000 \end{array}$$

Proposal: Optimize for Addition

$$\begin{array}{r} 0101 \\ +\,1011 \\ \hline 0000 \end{array} \qquad \begin{array}{r} 0011 \\ +\,1101 \\ \hline 0000 \end{array} \qquad \begin{array}{r} 0000 \\ +\,0000 \\ \hline 0000 \end{array}$$

The negated number is the original number **bitwise inverted**, **plus one more**.

Proposal: Optimize for Addition

A binary number plus its inverse is all 1s.

$$\begin{array}{r} 0101 \\ +\,1010 \\ \hline 1111 \end{array}$$

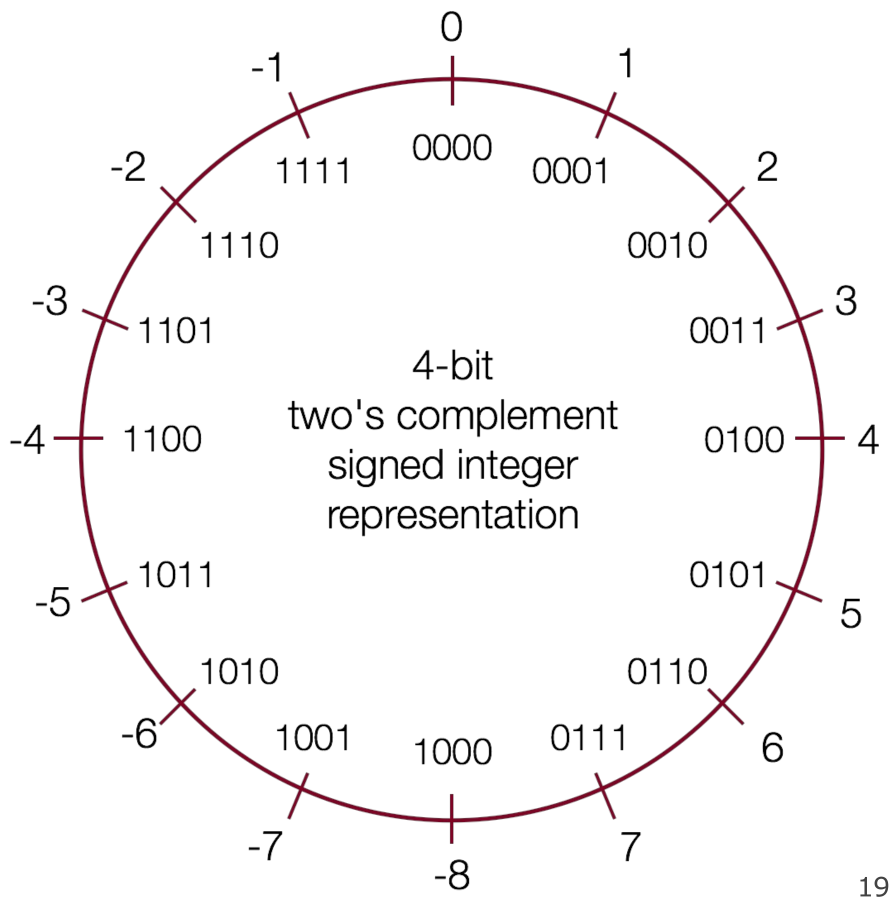
Add 1 to this to carry over all 1s and get 0!

$$\begin{array}{r} 1111 \\ +\,0001 \\ \hline 0000 \end{array}$$

This "proposal" is in fact the scheme used in practice, and it’s called **two’s complement**.

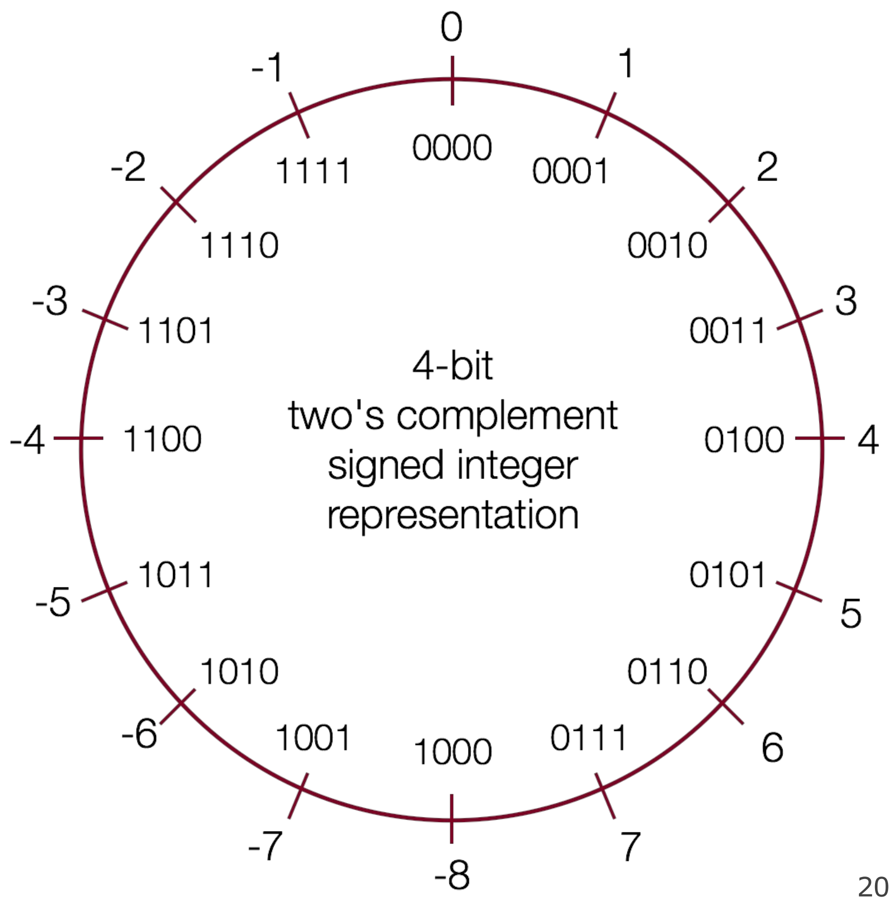
Redux: Two’s Complement

- We represent a positive number as we did before, and its negative counterpart via its **two's complement**.
- The **two’s complement** of a number is formed by inverting all bits and then adding 1.
- This works to convert from positive to negative **and** back from negative to positive!



Redux: Two's Complement

- **Con**: more difficult to represent than unsigned, and difficult to convert to and from decimal, between positive and negative.
- **Pro**: only 1 representation for 0. 😍
- **Pro**: the most significant bit still indicates the sign of a number.
- **Pro**: addition now works uniformly for any combination of positive and negative numbers.



Binary Representation and Overflow

If you exceed the **maximum** **unsigned** value representable with a fixed number of bits n, the result **wraps around**, modulo 2^n. This is called **overflow**.

$$111111_2 + 000001_2 = 000000_2$$

If you go below the **minimum** **unsigned** value—i.e., 0—representable with a fixed number of bits, the result **wraps around** from the top (also modulo 2^n).

$$000000000_2 - 000000001_2 = 111111111_2$$

Aside: With signed numbers, the hardware still wraps modulo 2^n, but C doesn't really define signed overflow, so you're not **supposed** to rely on that behavior—even though, for n = 4, +7 typically becomes −8 on real machines.

Binary Representation and Overflow

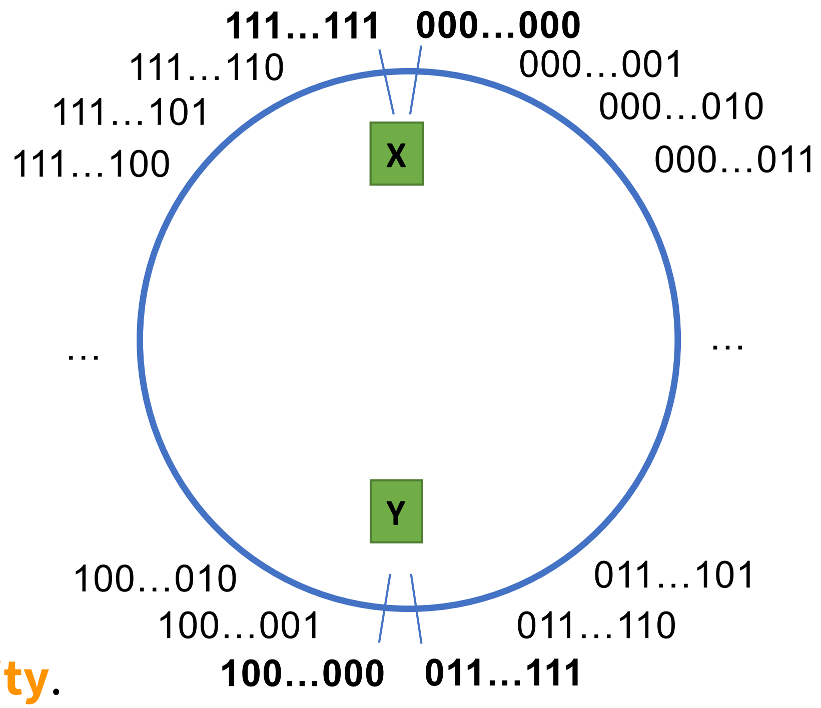
At which points can overflow occur for `signed` and `unsigned` `int`? *(assume binary values shown are 32-bit and that signed overflow is legit)*

A. signed and unsigned can both overflow at points X and Y

B. signed can overflow only at X, unsigned only at Y

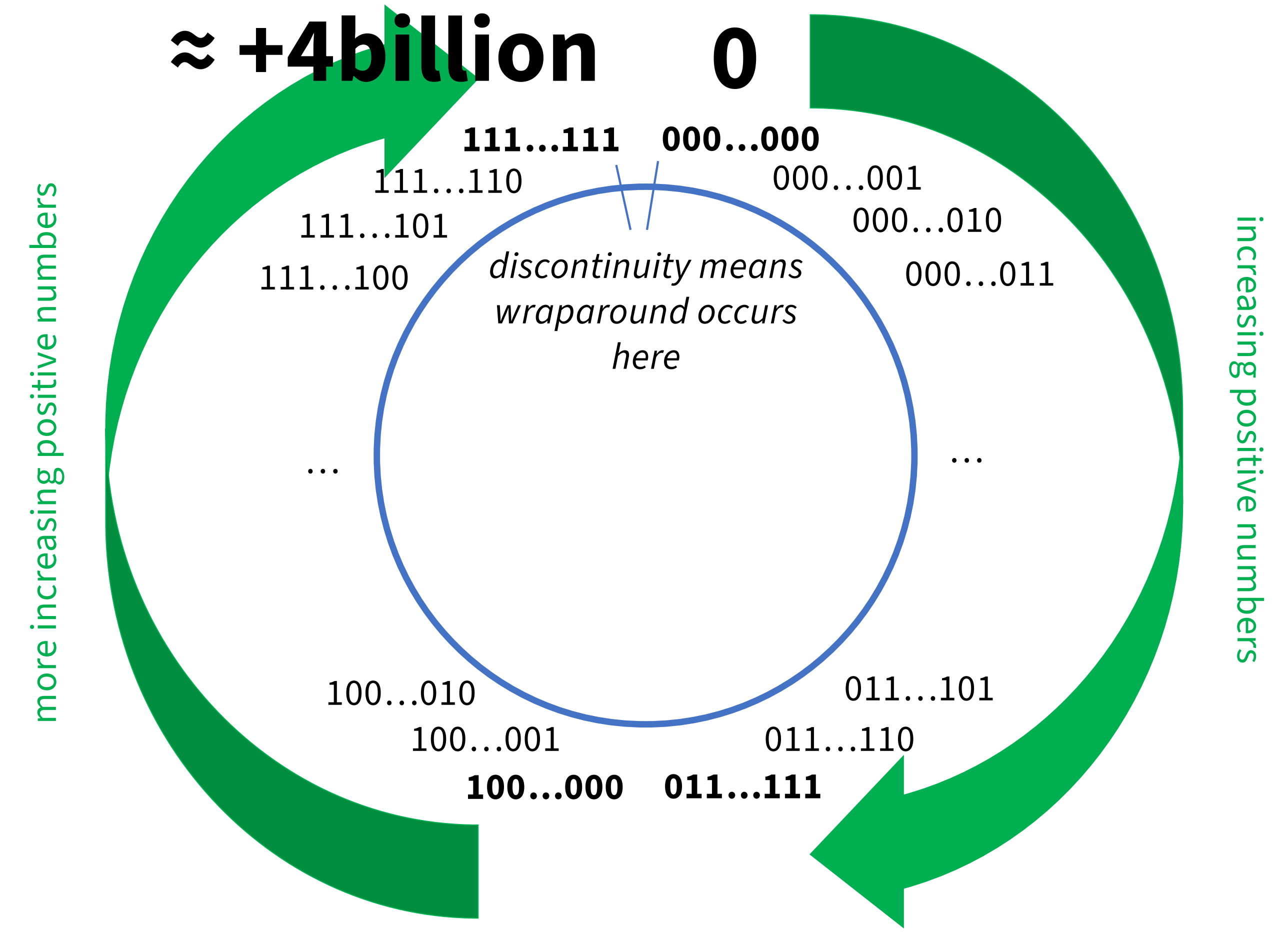
[✓ C. signed can overflow only at Y, unsigned only at X]

D. signed can overflow at X and Y, unsigned only at X

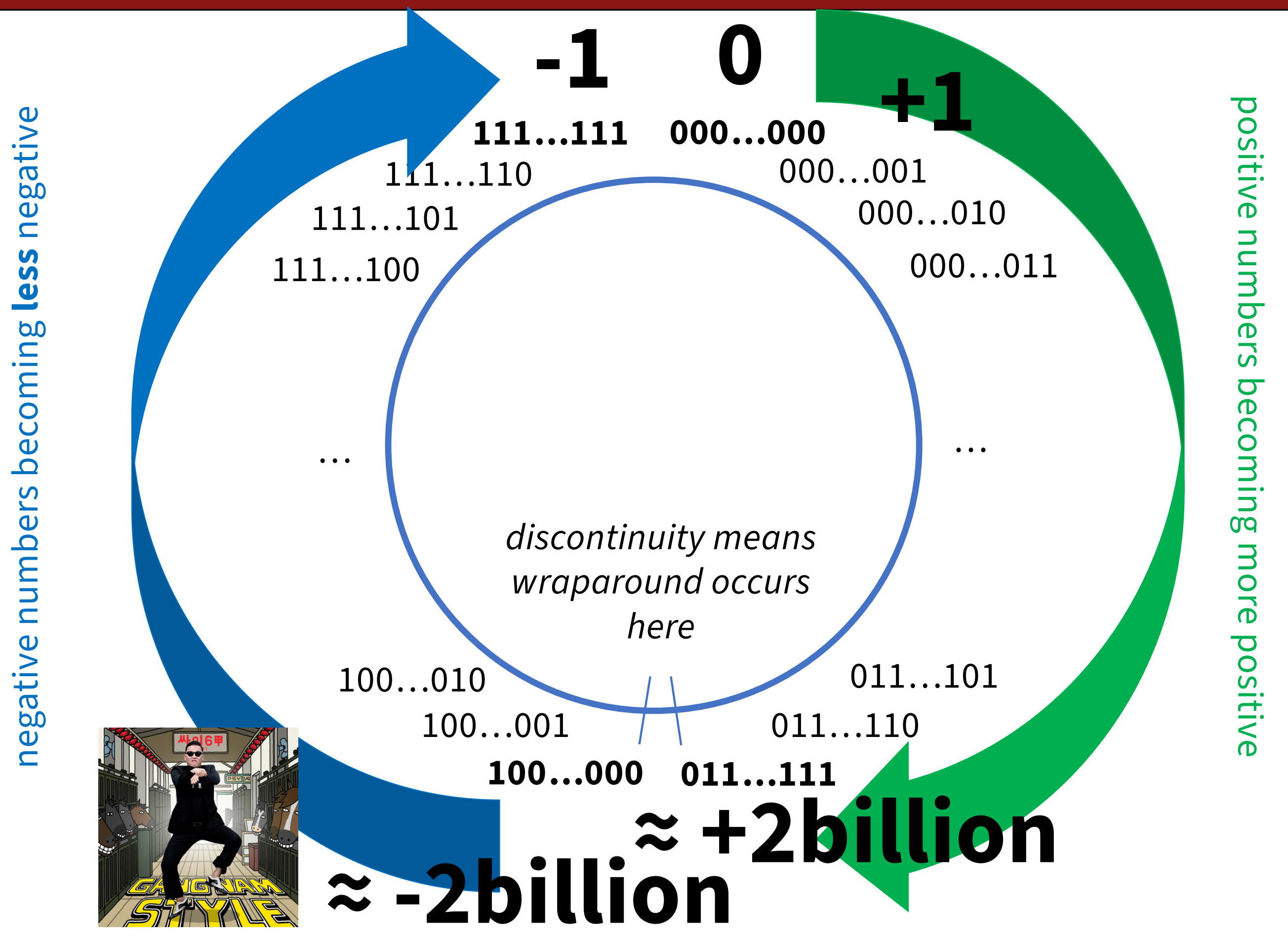


Key Idea: Assume **overflow** means **discontinuity**.

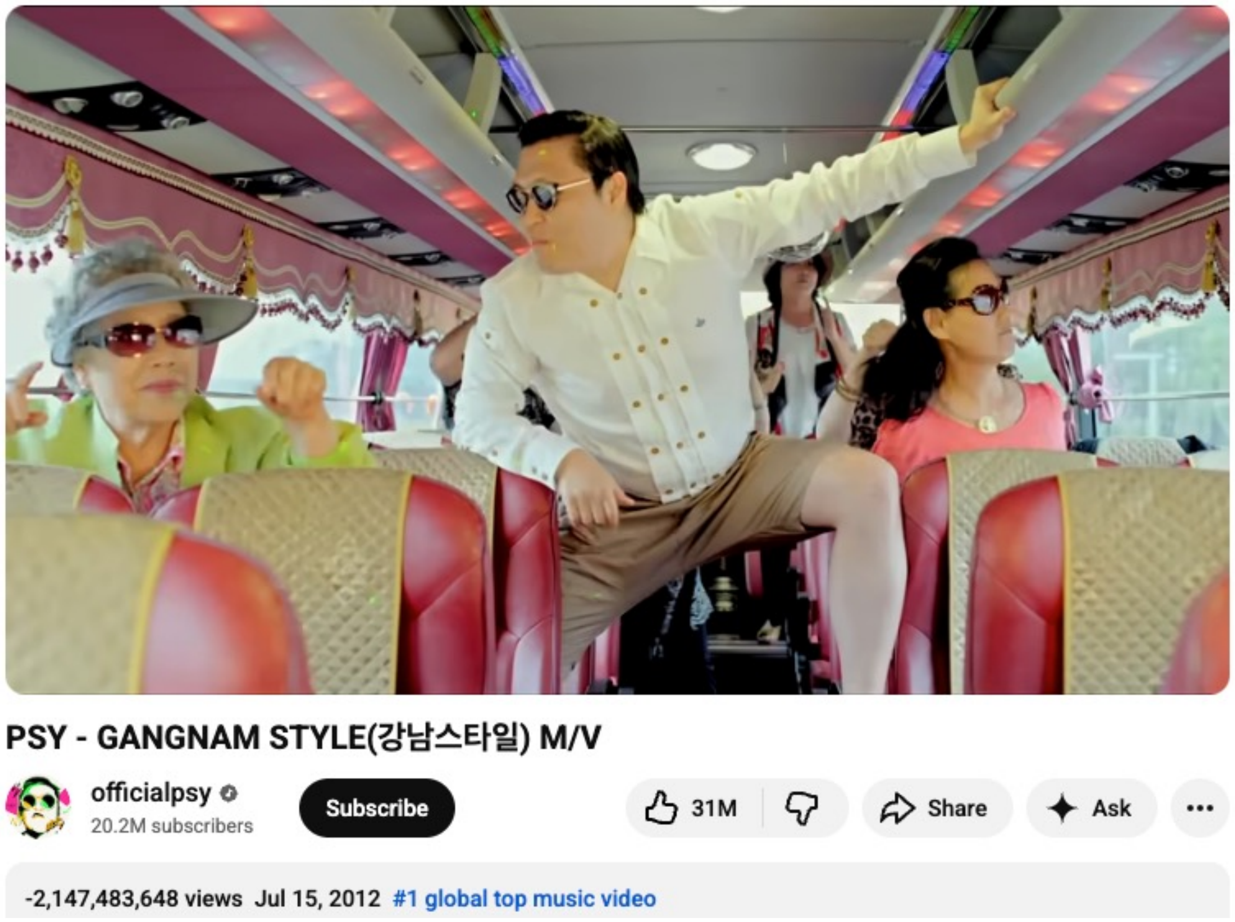
Unsigned Integers and Overflow



Signed Integers and "Overflow"



Overflow In The Wild: PSY


PSY - GANGNAM STYLE(강남스타일) M/V

-2,147,483,648 views Jul 15, 2012 #1 global top music video

YouTube: "We never thought a video would be watched in numbers greater than a 32-bit integer (up to 2,147,483,647 views), but that was before we met PSY. 'Gangnam Style' has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!" [link]

"We saw this coming a couple months ago and updated our systems to prepare for it." [link]

Overflow In The Wild: Pac-Man

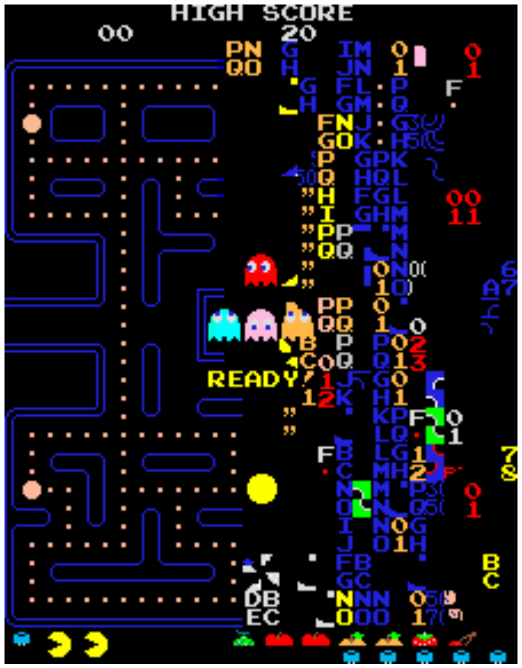
Pac-Man is a 1980 arcade game where players guide a yellow character through a maze, eating dots while avoiding four ghosts with distinct movement patterns. Power pellets briefly let Pac-Man eat the ghosts for bonus points.

Pac-Man hit market saturation in 1981 and not only became the highest-grossing arcade game ever—it became a full-on cultural phenomenon, with hit songs, TV shows, and massive merchandising.

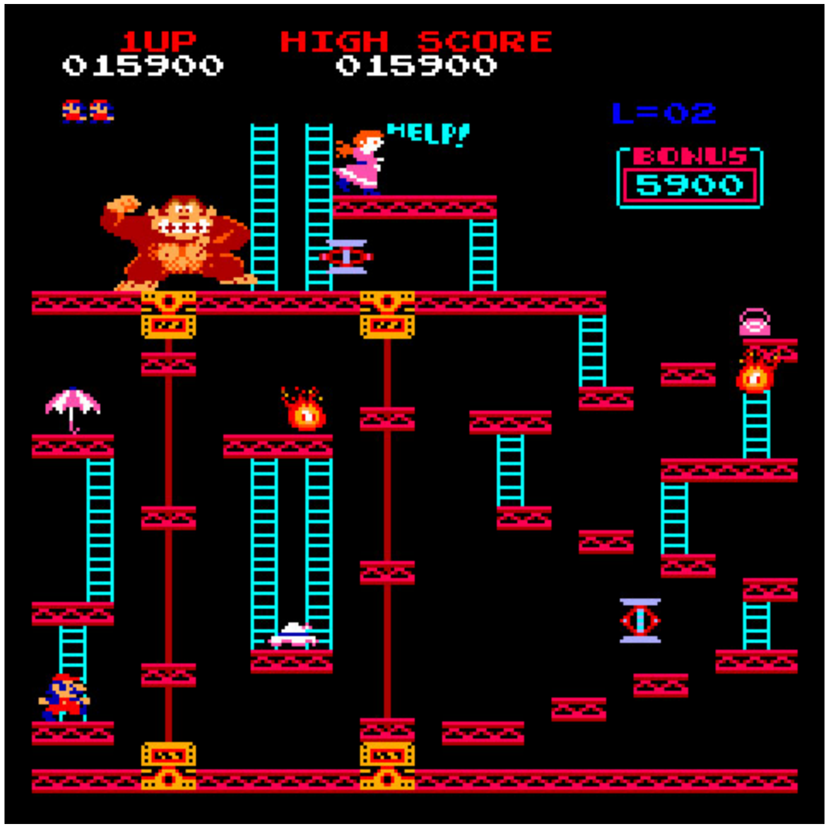
Jerry was in 7th grade in 1981. He stole quarters from his mom's purse and went to **Palombo's** to play for hours after school almost every day.

In the original game, the **Map 256 Glitch** [link] happened because the game's level counter **overflowed an 8-bit byte**. When you reach level 256—you know, after clearing two cherry levels, two strawberries, two oranges, two apples, two melons, two spaceships, two bells, and 241 keys—the game draws too many pieces of fruit and corrupts the right half of the maze with garbled letters, tiles, and symbols, making the game almost unplayable. At level 256, half the dots aren't displayed properly (though Pac-Man is expected to eat them), so progress beyond level 256 becomes impossible.

Jerry never got to level 56, much less 256.



Overflow In The Wild: Donkey Kong



Donkey Kong is a 1981 Nintendo arcade game where players guide **Mario** to climb ladders, scale platforms, and ultimately rescue a captive woman from a giant gorilla. Gameplay centers on jumping obstacles and avoiding runaway barrels and Brownian-motion fireballs.

Donkey Kong surfed the wake of Pac-Man, became absurdly popular in 1982, and launched both Mario (originally named **Jumpman**) and Donkey Kong as enduring icons of arcade history.

Jerry was in 8th grade in 1982, carried on his life of change purse crime, and went to that same Palombo's three blocks from home to play for hours after school almost every day.

In the original Donkey Kong, a timer is stored as an 8-bit unsigned value. On level 22, the timer calculation **overflowed**, because the number of time units Mario is granted to finish any level—including Level 22—is computed as 10 x (level + 4) = 260. 260 is really **1 0000 0100** in binary, but only **0000 0100** fits. That means Mario was given just 4 time units to complete a level designed to take 260.

Guess what level Jerry never got to.

Overflow In The Wild: Real Problems

Back in 2015, in **Boeing 787 Dreamliners** [link], a counter in each of **four generator control units** stored as a signed, 32-bit integer. This counter would **overflow** after $2^{31} - 1$ centiseconds (or about 248.5 days) of continuous power, triggering a shutdown of the GCU. **If all four GCUs hit this overflow together**—not unlikely if all four are powered up at the same time—**the aircraft could lose all electrical power**.

In December 2004, **Delta Air Lines** experienced an **operational collapse** [link] because of an **overflow** bug in software used to schedule flight crew. A counter tracking crew changes was stored as a **signed, 16-bit `int`**—the size of **`int`**s on most systems in 2004—with a maximum value of **32,767**.

Severe weather disruptions pushed this count above that limit, prompting it to wrap around to **-32,768**. This compromised the system's ability to properly count crew members available to work.

The result? Thousands of delays and cancellations.