Bitwise Operations
Floating Point Numbers

CS 2130: Computer Systems and Organization 1
January 26, 2026

Announcements

- Homework 1 due February 2, 2026
- Lab 3 tomorrow! (watch Canvas announcements for updates)

Operations

So far, we have discussed:

- Addition: `x + y`
  - Can get multiplication
- Subtraction: `x - y`
  - Can get division, but more difficult
- Unary minus (negative): `-x`
  - Flip the bits and add 1

Operations (on Integers)

Bit vector: fixed-length sequence of bits (ex: bits in an integer)

- Manipulated by bitwise operations

Bitwise operations: operate over the bits in a bit vector

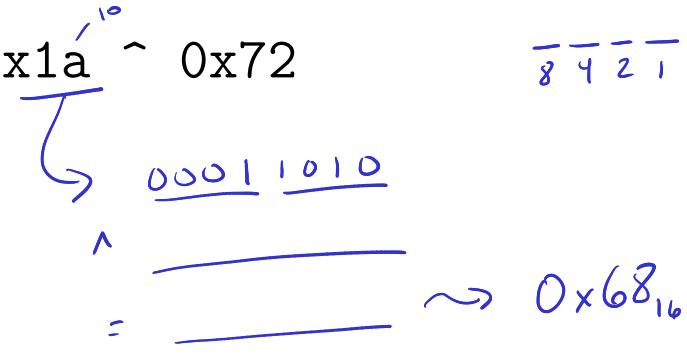
- Bitwise not: `~x` - flips all bits (unary)
- Bitwise and: `x & y` - set bit to 1 if x, y have 1 in same bit
- Bitwise or: `x | y` - set bit to 1 if either x or y have 1
- Bitwise xor: `x ^ y` - set bit to 1 if x, y bit differs

0110 → 1001

x = 1011
& y = 0110
0010

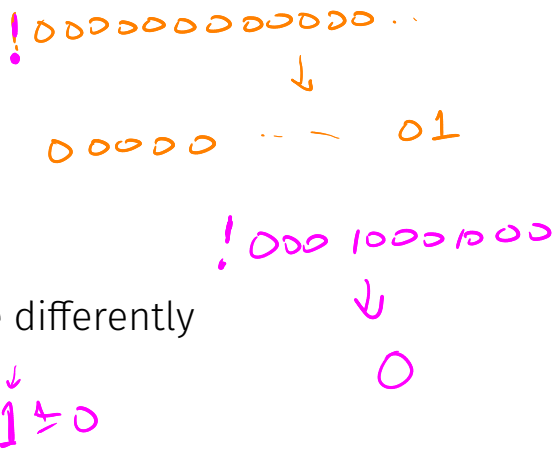
Your Turn!

What is: `0x1a ^ 0x72`



Operations (on Integers)

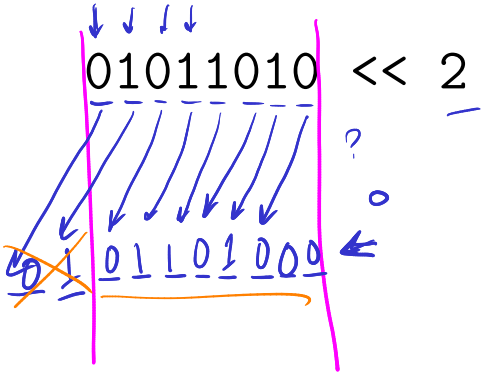
- Logical not: $!x$
  - $!0 = 1$ and $!x = 0, \forall x \neq 0$
  - Useful in C, no booleans
  - Some languages name this one differently



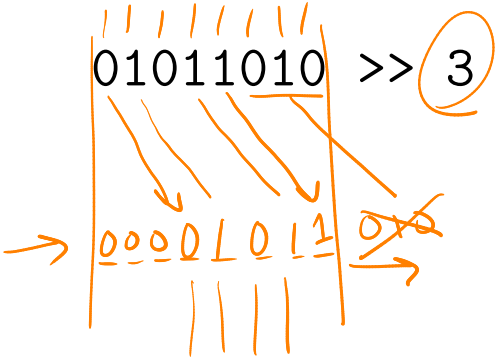
Operations (on Integers)

- Left shift: $x << y$ - move bits to the left
  - Effectively multiply by powers of 2
- Right shift: $x >> y$ - move bits to the right
  - Effectively divide by powers of 2
  - Signed (extend sign bit) vs unsigned (extend 0)

Left Bit-shift Example



Right Bit-shift Example



Bit-shift

Computing bit-shift effectively multiplies/divides by powers of 2

binary

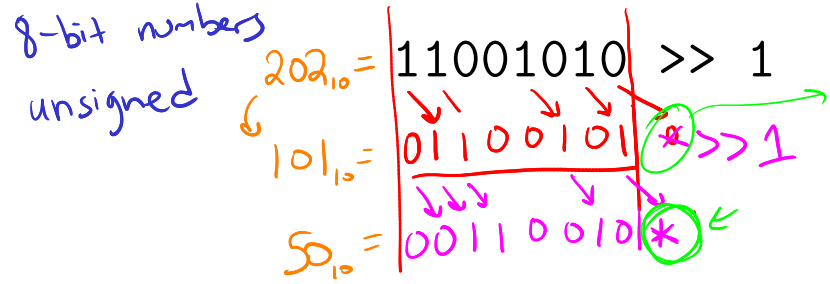
Consider decimal:

$$2130 <<_{10} 2 = 213000 = 2130 \times 100 = 2130 \times 10^2$$

$$2130 >>_{10} 1 = 213 = 2130 / 10$$

$$2130 \times 10^{-1}$$

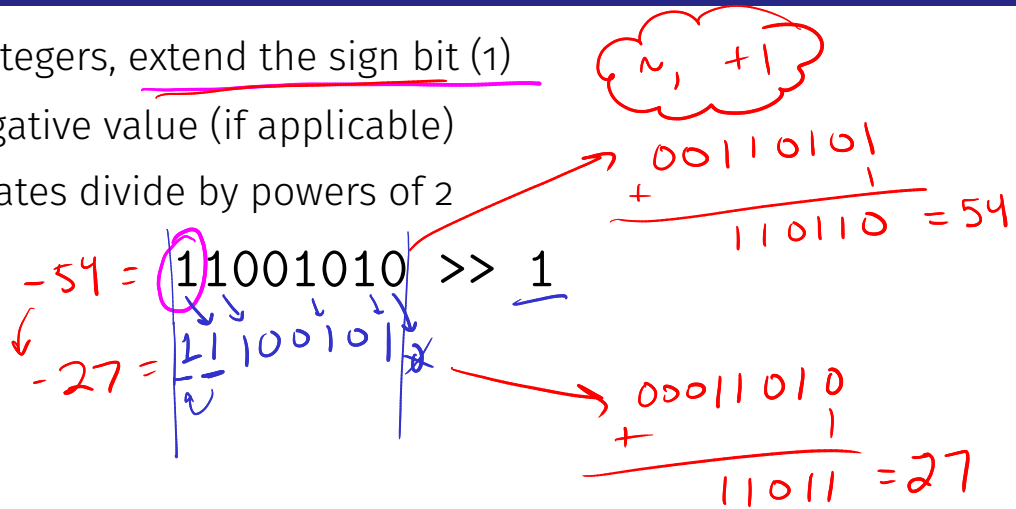
Right Bit-shift Example 2



Right Bit-shift Example 2

For **signed** integers, extend the sign bit (1)

- Keeps negative value (if applicable)
- Approximates divide by powers of 2



Bit fiddling example

subtract: Given x and y, set z to x - y without using - or multi-bit constants.

$z = x - y$

What about other kinds of numbers?

Non-Integer Numbers

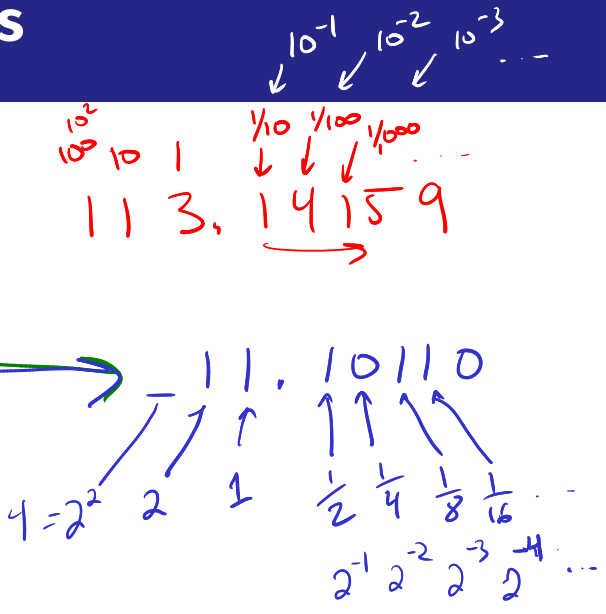
Floating point numbers

- Decimal: 3.14159

Non-Integer Numbers

Floating point numbers

- Decimal: 3.14159
- Binary: 11.10110



Non-Integer Numbers

Floating point numbers

- Decimal: 3.14159
- Binary: 11.10110
- With integers, the point is always fixed after all digits
- With floating point numbers, the point can move!

Non-Integer Numbers

Floating point numbers

- Decimal: 3.14159
- Binary: 11.10110
- With integers, the point is always fixed after all digits
- With floating point numbers, the point can move!

Challenge! only 2 symbols in binary

Scientific Notation

Convert the following decimal to scientific notation:

2130.

2.130×10^{3}

decimal

Scientific Notation

Convert the following binary to scientific notation:

unsigned

101101.

1.01101×2^{5}

5 ← binary

Something to Notice

An interesting phenomenon:

- Decimal: first digit can be any number *except* 0

$$2.13 \times 10^3$$

0.213×10^4

Something to Notice

An interesting phenomenon:

- Decimal: first digit can be any number *except* 0

$$2.13 \times 10^3$$

- Binary: first digit can be any number *except* 0 Wait!

$$1.01101 \times 2^5$$

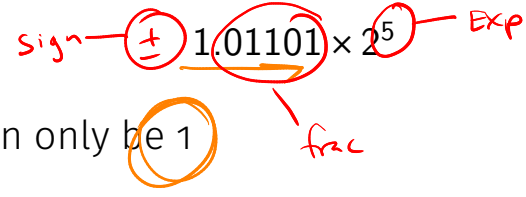
Something to Notice

An interesting phenomenon:

- Decimal: first digit can be any number *except* 0

$$2.13 \times 10^3$$

- Binary: first digit can be any number *except* 0 Wait!



  - First digit can only be 1

Floating Point in Binary

We must store 3 components

- **sign** (1-bit): 1 if negative, 0 if positive
- **fraction** or **mantissa**: (?-bits): bits after binary point
- **exponent** (?-bits): how far to move binary point

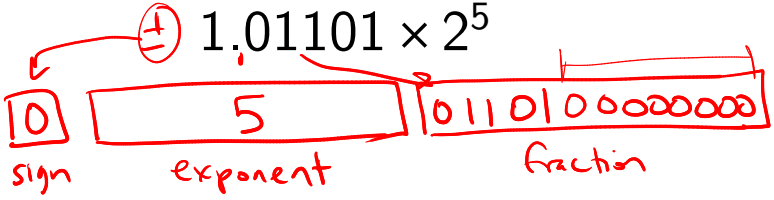
We do not need to store the value before the binary point. Why?

Floating Point in Binary

How do we store them?

- Originally many different systems
- IEEE standardized system (IEEE 754 and IEEE 854)
- Agreed-upon order, format, and number of bits for each

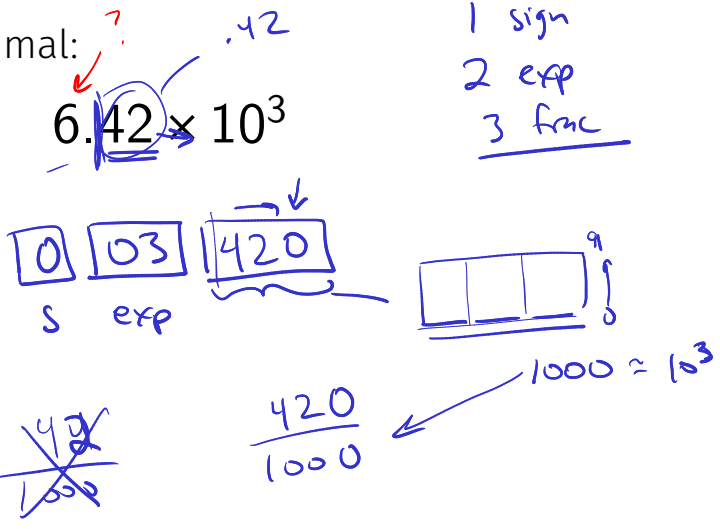
64-bit



Example

A rough example in Decimal:

6.42×10^3



Exponent

How do we store the exponent?

- Exponents *can* be negative

$$2^{-3} = \frac{1}{2^3} = \frac{1}{8}$$

1.01101×2^{-3}

0.00101101

- Need positive and negative ints (but no minus sign)

Exponent

How do we store the exponent?

- Exponents *can* be negative

$$2^{-3} = \frac{1}{2^3} = \frac{1}{8}$$

- Need positive and negative ints (but no minus sign)
- *Don't we always use Two's Complement?*

Exponent

How do we store the exponent?

- Exponents *can* be negative

$$2^{-3} = \frac{1}{2^3} = \frac{1}{8}$$

- Need positive and negative ints (but no minus sign)
- *Don't we always use Two's Complement?* Unfortunately Not

Exponent

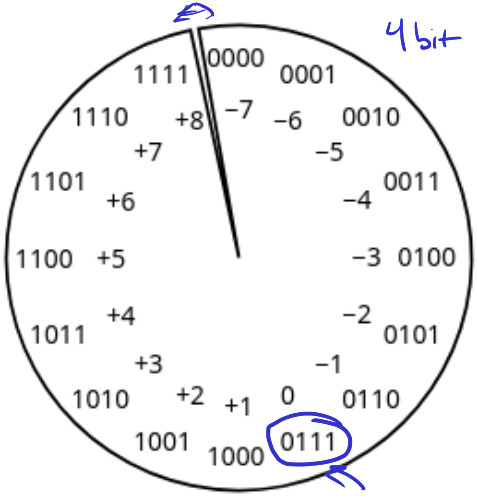
How do we store the exponent?

- Biased integers
  - Make comparison operations run more smoothly
  - Hardware more efficient to build
  - Other valid reasons

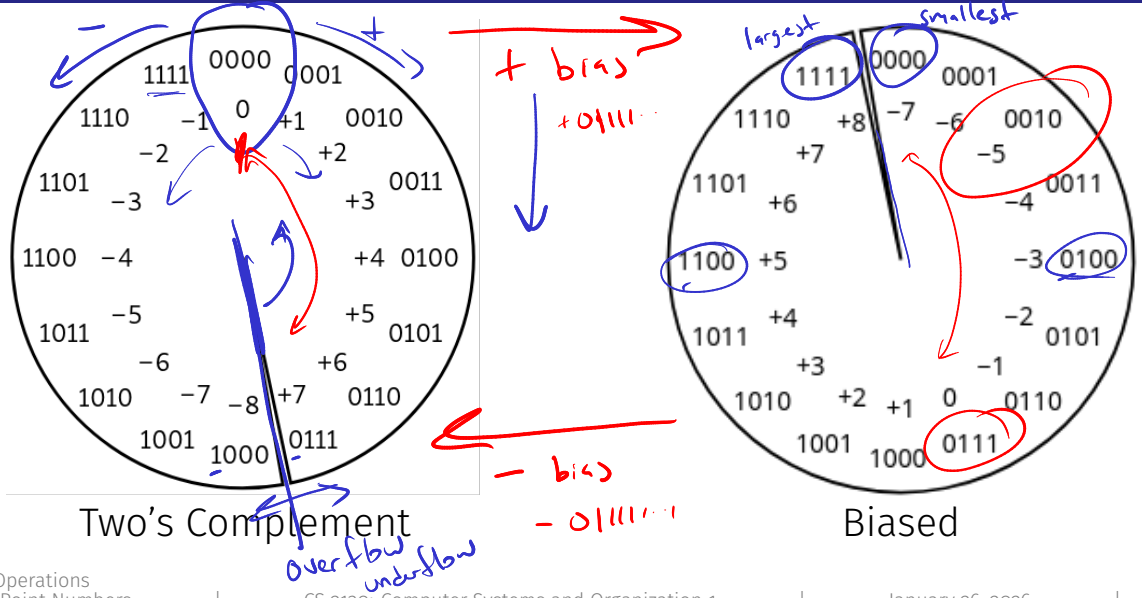
Biased Integers

Similar to Two's Complement, but add **bias**

- **Two's Complement**: Define 0 as 00...0
- **Biased**: Define 0 as 0111...1
- Biased wraps from 000...0 to 111...1

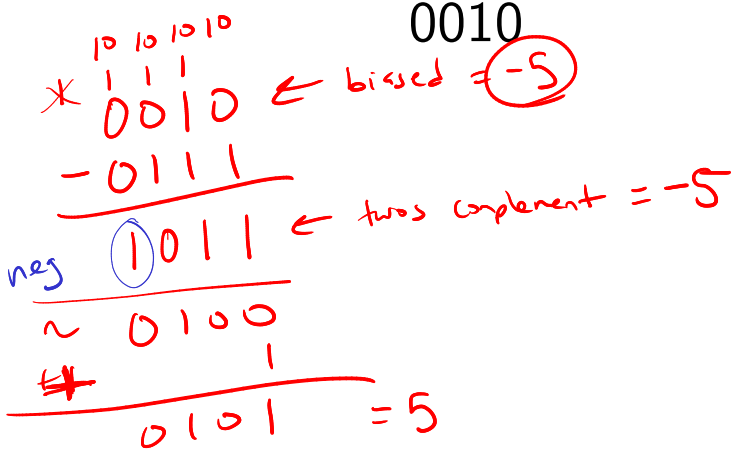


Biased Integers



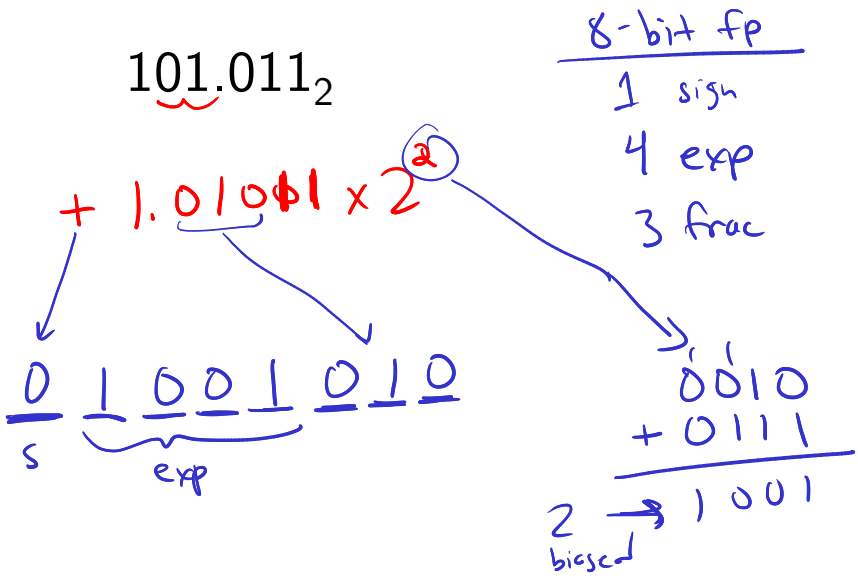
Biased Integers Example

Calculate value of biased integers (4-bit example)



Biased Integers

Floating Point Example



Floating Point Example

$$101.011_2$$

Floating Point Example

What does the following encode?

1	001110	1010101

Floating Point Example

What does the following encode?

1	001110	1010101

What about o?

Floating Point Numbers

Four cases:

- **Normalized**: What we have seen today

$$s\ eeee\ ffff = \pm 1.ffff \times 2^{eeee - \text{bias}}$$

- **Denormalized**: Exponent bits all 0

$$s\ eeee\ ffff = \pm 0.ffff \times 2^{1 - \text{bias}}$$

- **Infinity**: Exponent bits all 1, fraction bits all 0 (i.e., $\pm\infty$)
- **Not a Number (NaN)**: Exponent bits all 1, fraction bits not all 0