

The background of the slide features a dark, textured pattern. At the top, there are horizontal bands of binary code (0s and 1s) in a light green color. Below this, there are faint, light-colored circuit diagrams and logic gate symbols, including what appears to be a bus system and various logic components like multiplexers and decoders. The overall aesthetic is technical and digital.

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:



A hand-drawn diagram consisting of a horizontal line with several vertical tick marks above it, resembling a bit pattern. Below the line, the word "integer" is written in a cursive script.

- 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: $\sim x$ - 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: $x \wedge y$ - set bit to 1 if x, y bit differs

$0110 \rightarrow 1001$

$x = 1011$
↓ ↓ ↓

$\& y = 0110$

 0010

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

! 0000000000000000 ..



00000 --- 01

! 000 1000 1000



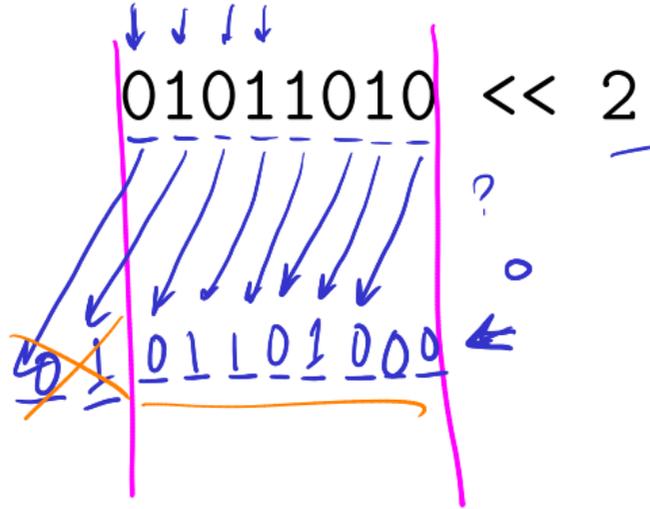
0

!0 ≈ 1 . !1 ≈ 0

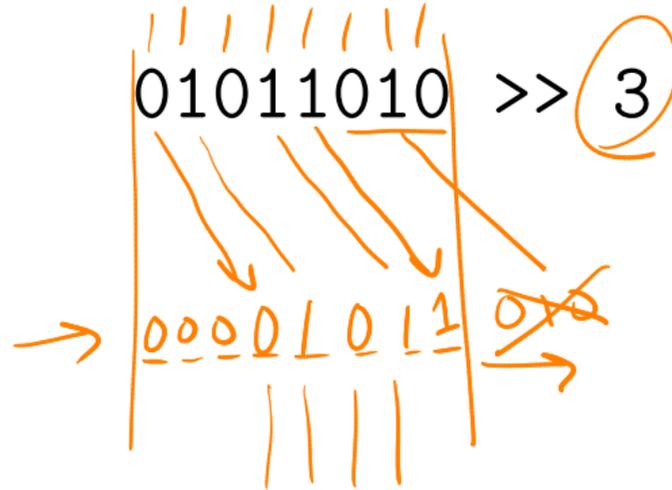
Operations (on Integers)

- Left shift: $x \ll y$ - move bits to the left
- Effectively multiply by powers of 2
- Right shift: $x \gg y$ - 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

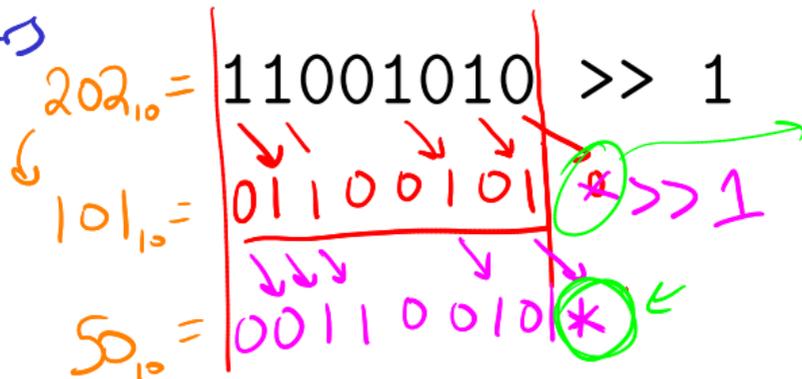
Consider decimal:

$$\underline{2130} \ll_{10} \textcircled{2} = \underline{213000} = \underline{2130} \times \underline{100} = 2130 \times 10^2$$

$$\underline{2130} \gg_{10} \underline{1} = \underline{213} = \underline{2130} / \underline{10} = 2130 \times 10^{-1}$$

Right Bit-shift Example 2

8-bit numbers
unsigned



Right Bit-shift Example 2

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

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

$$\begin{array}{l} -54 = 11001010 \ggg 1 \\ \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \\ -27 = 11001010 \end{array}$$

~ + 1

$$\begin{array}{r} 00110101 \\ + \\ \hline 110110 = 54 \end{array}$$

$$\begin{array}{r} 00011010 \\ + \\ \hline 11011 = 27 \end{array}$$

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

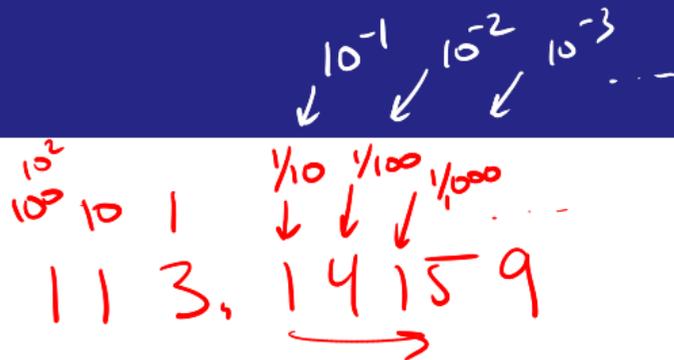
- Decimal: 3.14159

 decimal point

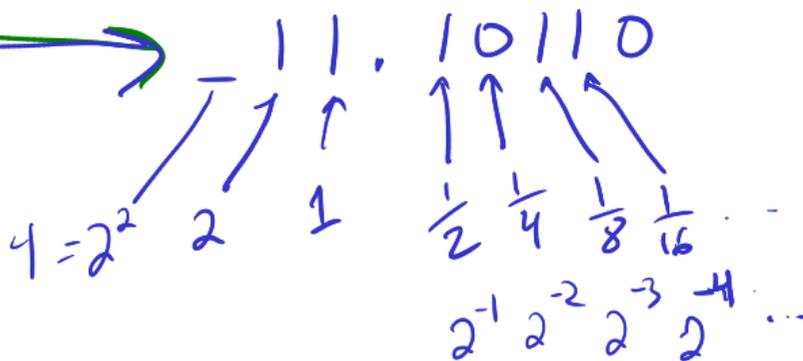
Non-Integer Numbers

Floating point numbers

- Decimal: 3.14159
- Binary: 11.10110



binary point



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.

$$\underline{2.130} \times 10^{\underline{3}}$$

↑ decimal

Scientific Notation

Convert the following binary to scientific notation:

unsigned
↓
101101.

$$1.01101 \times 2^5$$

↑ binary

Something to Notice

An interesting phenomenon:

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

$$\underline{2.13} \times 10^3$$

~~$$0.213 \times 10^4$$~~

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!**

A diagram illustrating a binary floating-point number: 1.01101×2^5 . The number is annotated with handwritten red circles and lines. The sign is a plus sign (+) in a circle, labeled "sign". The mantissa is "1.01101", with the "1" circled in orange and labeled "frac" (fraction). The exponent is "5", circled in red and labeled "EXP". A yellow arrow points from the "1" in the mantissa to the "1" in the exponent.

- First digit can only be 1

Floating Point in Binary

We must store 3 components

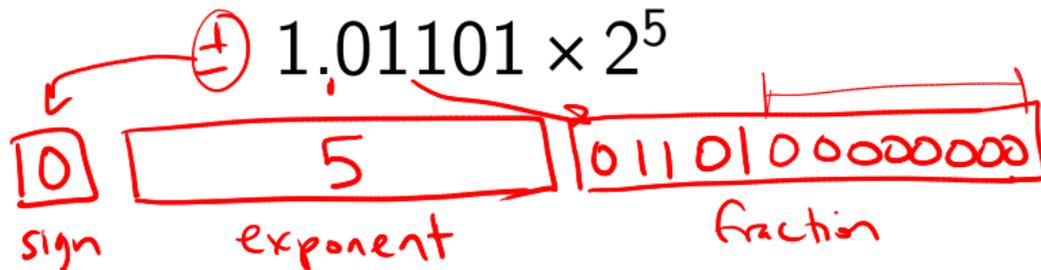
- **sign** (1-bit): 1 if negative, 0 if positive
- **fraction** or **mantissa**: (?-bits): bits after binary point
- **exponent** (?-bits): how far to move binary point

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



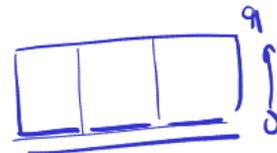
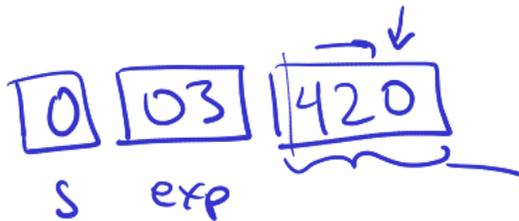
Example

A rough example in Decimal:

$$6.42 \times 10^3$$

(Handwritten annotations: a red arrow points to the decimal point, a blue circle highlights '42', and a blue arrow points from '42' to the exponent '3')

1 sign
2 exp
3 frac



$$\frac{42}{1000}$$

(Handwritten note: This fraction is crossed out with a large 'X')

$$\frac{420}{1000}$$

1000 \approx 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

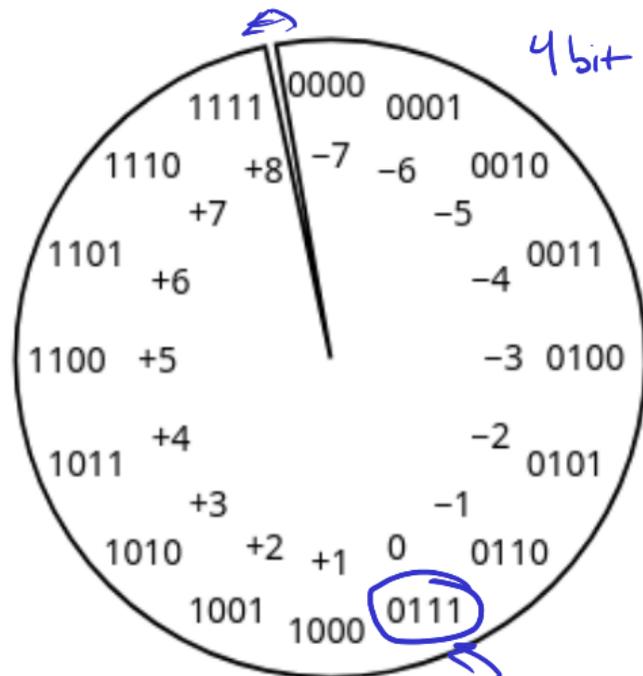
How do we store the exponent?

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

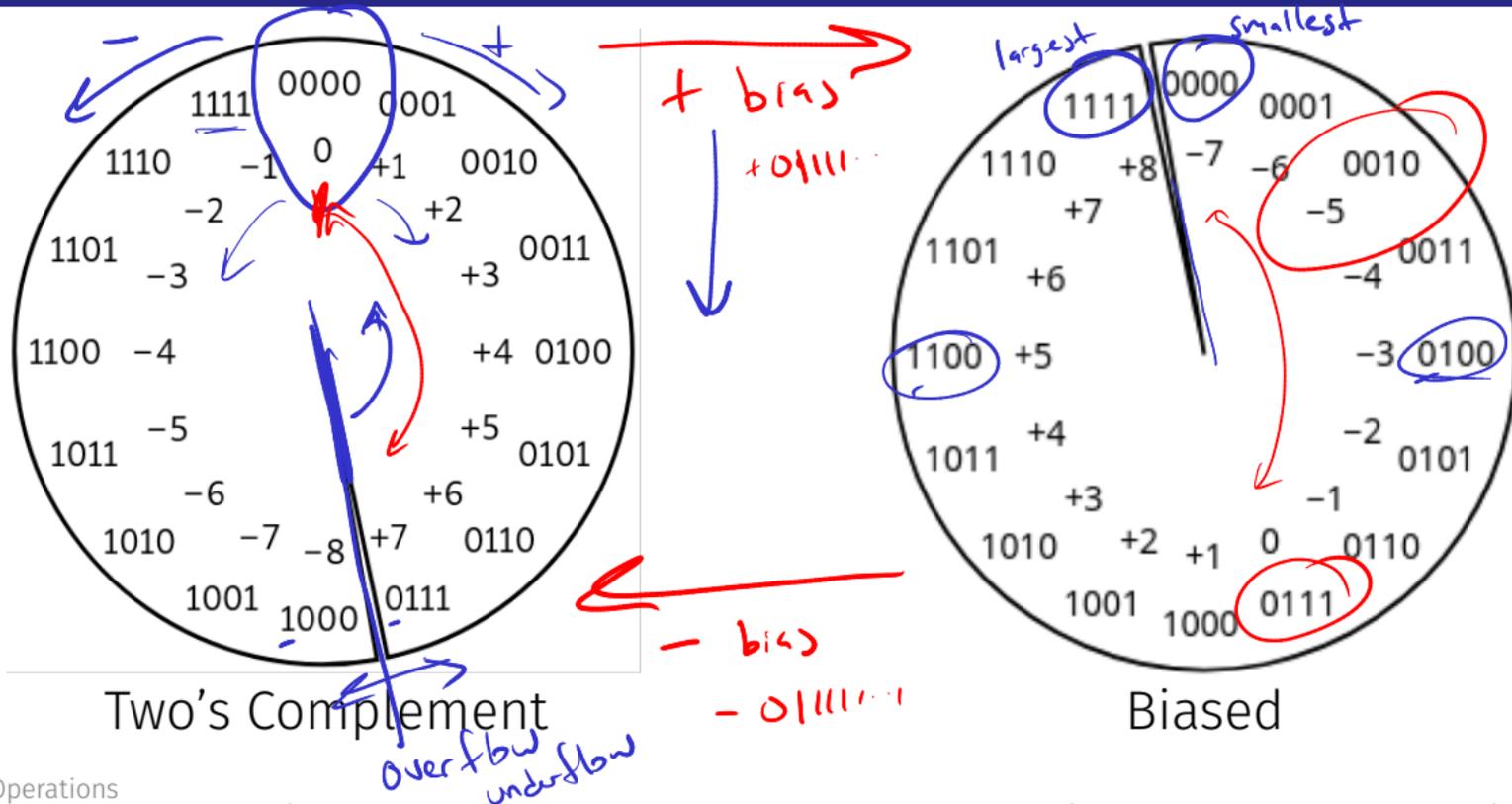
Biased Integers

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

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



Biased Integers



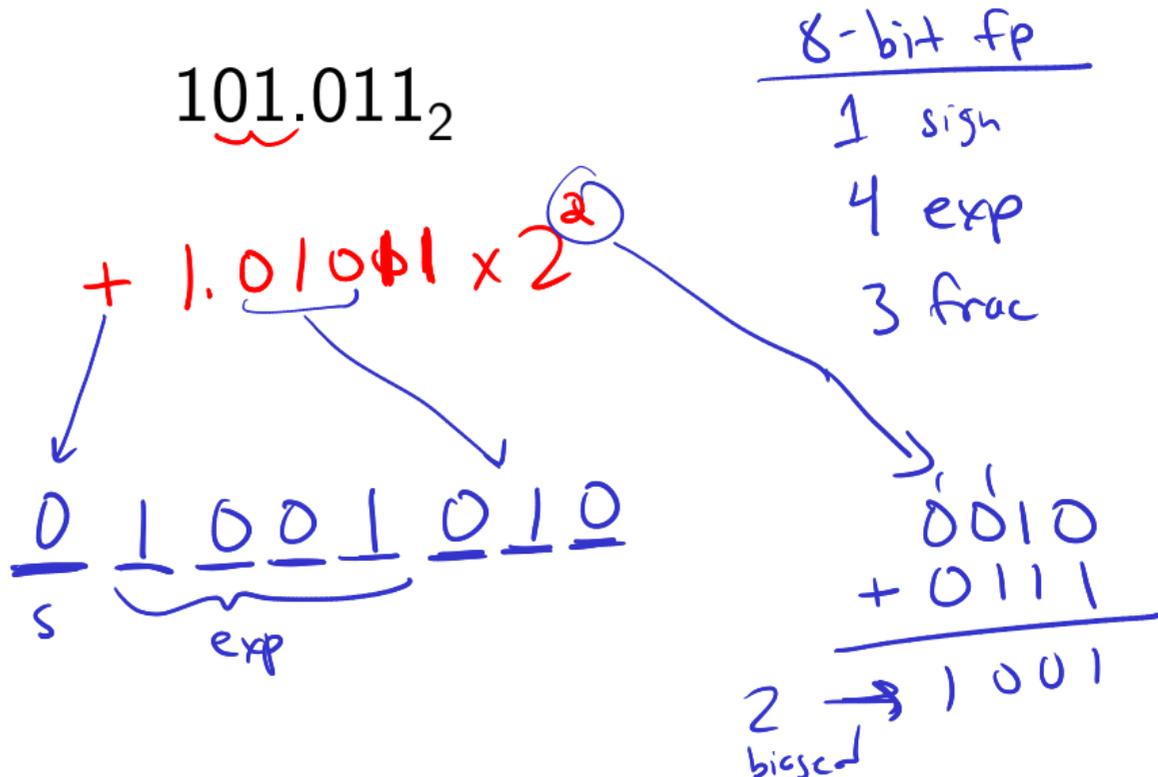
Biased Integers Example

Calculate value of biased integers (4-bit example)

$$\begin{array}{r} \\ \\ \\ * \begin{array}{cccc} & 10 & 10 & 10 & 10 \\ & 1 & 1 & 1 & \\ 0 & 0 & 1 & 0 & \end{array} \leftarrow \text{biased} = \textcircled{-5} \\ - 0111 \\ \hline \text{neg } \textcircled{1}011 \leftarrow \text{two's complement} = -5 \\ \sim 0100 \\ \# \\ \hline 0101 = 5 \end{array}$$

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 0?

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