# Bitwise Operations
# Floating Point Numbers

## CS 2130: Computer Systems and Organization 1

**Xinyao Yi** Ph.D.
Assistant Professor

## Announcements

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

# Quick Review

# Values of Two's Complement Numbers

**Why "invert the bits and add 1"?**

- Because in 8 bits, we have 256 total values (0–255).
- A negative number is stored as 256 − (its absolute value). $2^n - |x|$
- The "invert + 1" trick is just a fast way to compute that.

invert bits: $1\,\underbrace{0000}\,\underbrace{0000} = 2^8$

$0\,\underbrace{1111}\,\underbrace{1111} \Rightarrow 255 \quad (2^n - 1)$

$255 + 1 = 256 = 2^8$

(This is the definition of negative numbers in 2's complement).

$-2^{n-1} \le x \le 2^{n-1} - 1$

# 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

## Operations (on Integers)

Logical not: $!x$

0 means false

Any non-zero value means true.

- $!0 = 1$ and $!x = 0, \forall x \neq 0$
- Useful in C, no booleans
- Some languages name this one differently

C does not have a buit-in Boolean type in older standards.

Logical expressions return integers (0 or 1)

Example:
```
if (!ptr) {
    // ptr is NULL
}
```

Python: not

Java/C/C++: !

Bash: !

## 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

01011010 << 2

0 1 0 1 1 0 1 0 0 0

# Right Bit-shift Example

01011010 >> 3

00001011 ~~010~~

# Bit-shift

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

Consider decimal:

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

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

# Right Bit-shift Example 2

11001010 >> 1

if this is a signed number:

11100101

if this is an unsigned number:

01100101

## Right Bit-shift Example 2

For signed integers, extend the sign bit (1)

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

11001010 >> 1

→ Why "approximate"?

For positive numbers: 8 >> 1 = 4
8 / 2 = 4 ⇒ Same result

For negative numbers: -3 >> 1 = -2
-3 / 2 = -1 ⇒ Different

Different results because:

>> performs arithmetic shift (rounds toward -∞)

/ in C performs integer division (rounds toward 0)

UNIVERSITY of VIRGINIA

# Bit fiddling example

$3-5 \Rightarrow 3+(-5)$

$0110 - 0010 \Rightarrow 0110 + (-0010)$

We've learned how to do this!

So I can do substraction now!

# 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

# Non-Integer Numbers

- What about other kinds of numbers?

# Non-Integer Numbers

Floating point numbers

- Decimal: 3.14159

decimal point.

UNIVERSITY of VIRGINIA

# 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 0110101.
- 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

(We can't represent binary point).

UNIVERSITY of VIRGINIA

## Scientific Notation

Convert the following decimal to scientific notation:

2130

$2.130 \times 10^3$

## Scientific Notation

Convert the following binary to scientific notation:

101101

$1.01101 \times 2^5$

## Something to Notice

An interesting phenomenon:

- Decimal: first digit can be any number except 0

$$2.13 \times 10^3$$

$0.213 \times 10^4$ X

wrong!

## 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!

$$1.01101 \times 2^5$$

  – First digit can only be 1

We don't need to store this "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

depends on the hardware design.

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

always 1

# 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

$$\pm\ 1.\boxed{01101} \times 2^{\boxed{5}}$$

fraction

exponent

0 1 1 0 1 0 0 0 0 0 0 0

## Example

A rough example in Decimal:

$$6.42 \times 10^3$$

$$\frac{42}{100} = \frac{420}{1000} = \frac{4200}{10000}$$

$\underbrace{0}_{\text{sign}}$ $\underbrace{03}_{\text{exp}}$ $\underbrace{4200}_{\text{fraction}}$

2 bits, add 0's front

4 bits, add 0's behind.

# 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)

## 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

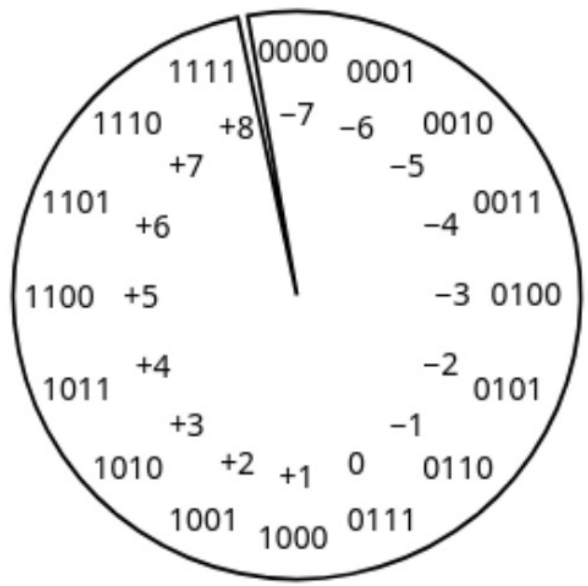
How do we store the exponent?

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

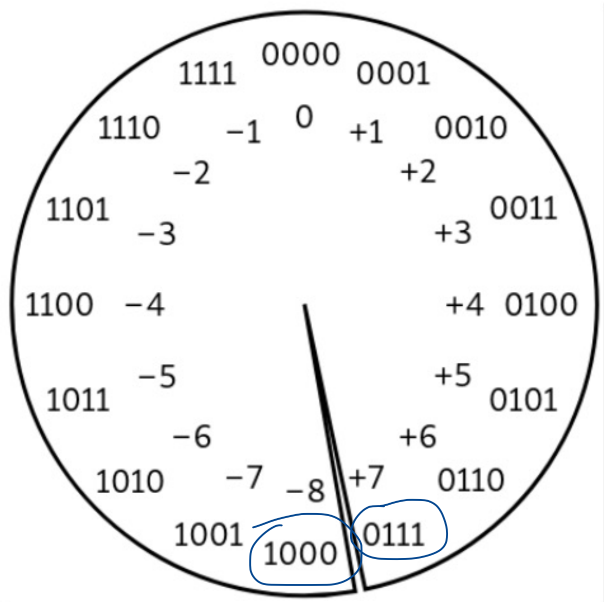
## Biased Integers

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

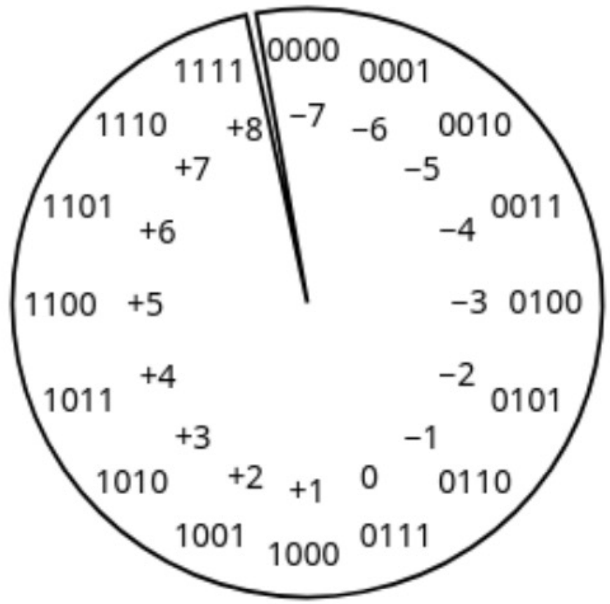
- **Two's Complement**: Define 0 as 00...0
- **Biased**: Define 0 as 0111...1
- Biased wraps from 000...0 to 111...1



# Biased Integers




Two's Complement




Biased



if we want to convert between 2's complement and biased, all we have to do is add the bias.

# Biased Integers Example

Calculate value of biased integers (4-bit example)

0010

2's comp → biased ⇒ add the bias

biased → 2's comp ⇒ substract the bias

ignore the fact that I borrowed too far.

```
  0010
- 0111 (bias)
  1011
```

2's complement: 1011

flip: 0100

"+1": 0101

0101 is +5, so 1011 is −5

# Floating Point Example

1 bit: sign
4 bits: exponent
3 bits: fraction

$$101.011_2$$

$1.01011 \times 2^2$

2's complement for 2: 0010

add the bias: 0010
+0111
$\boxed{1001}$

0 sign | 1001 exponent | 01011 fraction

only 3 bits for fraction?
We are rounding
01011
↓
011

↓

011 fraction