Floating Point Numbers

CS 2130: Computer Systems and Organization 1

January 30, 2023

Announcements

- TA Office Hours starting Wednesday
  - Wednesdays, Rice 011
  - Thurs-Sun, Olsson 001
- Please join our Discord server
- Lab tomorrow: hex editor
- Homework 1 due Feb 6 (Mon)

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

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
- 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)

1000 0000 <<1

00000000 >>1

00000000

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

  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

Scientific Notation

Convert the following decimal to scientific notation:

2130

2.130×10^3

Scientific Notation

Convert the following binary to scientific notation:

101101.

1.01101×2^5

Something to Notice

An interesting phenomenon:

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

$$2.13 \times 10^3 \qquad \cancel{0.213 \times 10^4}$$

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!

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

  - 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?

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

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

s	exponent	
	5	01101 00000000

Example

A rough example in Decimal:

$\frac{42}{100}$

$$\cancel{6}.\underline{42} \times 10^3$$

1 sign
2 exp
3 frac

$$\underset{s}{\boxed{0}} \quad \underset{exp}{\boxed{03}} \quad \underset{frac}{\boxed{420}}$$

$\frac{420}{1000}$

Exponent

How do we store the exponent?

- Exponents *can* be negative

.000

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

- 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?

- Exponents *can* be negative

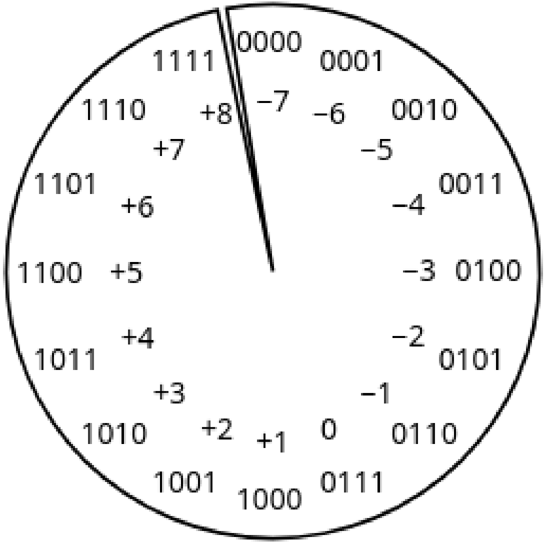
$$2^{-3} = \frac{1}{2^3} = \frac{1}{8}$$

- Need positive and negative ints (but no minus sign)
- *Don't we always use Two's Complement?* Unfortunately Not
- Biased integers
  - Make comparison operations run more smoothly
  - Hardware more efficient to build
  - Other valid reasons

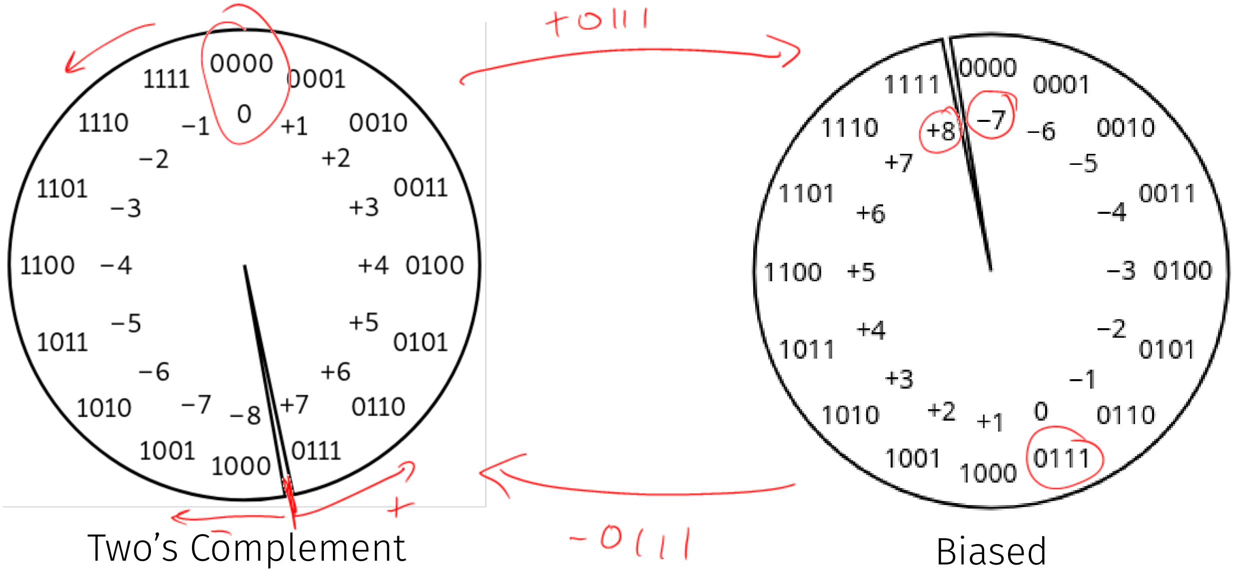
Biased Integers

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

Biased Integers Example

Calculate value of biased integers (4-bit example)

0010 = -5

biased → 0010 = -5
- 0111
1011 = -5
2's complement →
$\boxed{\sim \\ +1}$
0100
1
binary → 0101 = 5

bias ⇔ 0010 = 2
- 0111 = -7
2's → 1011 -5

Biased Integers

borrow 1

$$
\begin{array}{r}
0010 \\
-\;0111 \\
\hline
1011
\end{array}
\qquad\qquad
\begin{array}{r}
\overset{3}{\cancel{4}}\overset{1}{2} \\
-\;4 \\
\hline
\end{array}
$$

Floating Point Example

8-bit

1 sign

4 exp

3 fraction

$+\ 101.011_2$

$+\ 1.01011 \times 2^2$

$$
\begin{array}{r}
1\ 0\ 0\ 0 \\
1.0000111
\end{array}
$$

$$
\begin{array}{r}
11 \\
0010 \\
+\ 0111 \\
\hline
1001
\end{array}
$$

sign	exponent	frac
0	1001	011

Floating Point Example

$$101.011_2$$

Floating Point Example

What does the following encode?

s	exp	frac
1	001110	1010101

$$
\begin{array}{r l}
001110 & \leftarrow \# \text{ in binary} \\
-\ 011111 & \leftarrow \text{bias} \\
\hline
= -17 &
\end{array}
$$

$$-1.1010101 \times 2^{-17}$$

$$0.00\ldots\ 011010101$$

Floating Point Example

What does the following encode?

$\boxed{1}\ \boxed{001110}\ \boxed{1010101}$

What about 0?

Floating Point Numbers

Four cases:

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

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

- **Denormalized**: Exponent bits all 0

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

- **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