Section 2: Integer & Floating Point Numbers

- Representation of integers: unsigned and signed
- Unsigned and signed integers in C
- Arithmetic and shifting
- Sign extension

- Background: fractional binary numbers
- <u>IEEE floating-point standard</u>
- Floating-point operations and rounding
- Floating-point in C

IEEE Floating Point

- **Analogous to scientific notation**
  - Not 12000000 but 1.2×10^{7}; not 0.0000012 but 1.2×10^{-6}
    - (write in C code as: 1.2e7; 1.2e-6)
- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs today
- **Driven by numerical concerns**
  - Standards for handling rounding, overflow, underflow
  - Hard to make fast in hardware but numerically well-behaved

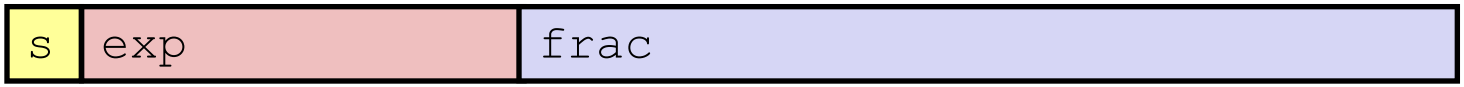
Floating Point Representation

- **Numerical form:**

$$V_{10} = (-1)^{s} * M * 2^{E}$$

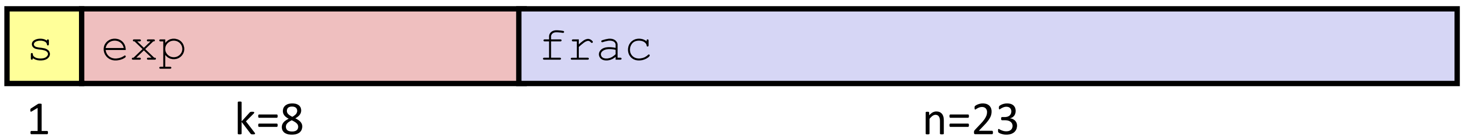
  - Sign bit ***s*** determines whether number is negative or positive
  - Significand (mantissa) ***M*** normally a fractional value in range [1.0,2.0)
  - Exponent ***E*** weights value by a (possibly negative) power of two

- **Representation in memory:**
  - MSB **`s`** is sign bit ***s***
  - **`exp`** field encodes ***E*** (but is *not equal* to E)
  - **`frac`** field encodes ***M*** (but is *not equal* to M)

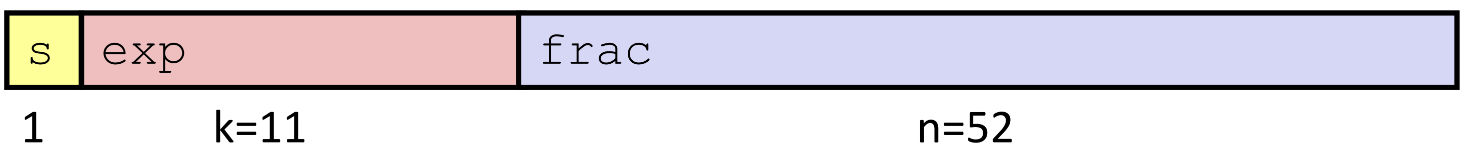


Precisions

- **Single precision: 32 bits**



- **Double precision: 64 bits**



Normalization and Special Values

$V = (-1)^{S} * M * 2^{E}$

s	exp	frac
	k	n

- **"Normalized" means the mantissa M has the form 1.xxxxx**
  - 0.011 x 2^5 and 1.1 x 2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it

- **How do we represent 0.0? Or special / undefined values like 1.0/0.0?**

Normalization and Special Values

$$V = (-1)^{S} * M * 2^{E}$$

s	exp	frac
	k	n

- **"Normalized" means the mantissa M has the form 1.xxxxx**
  - 0.011 x 2^5 and 1.1 x 2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it

- **Special values:**
  - The bit pattern 00...0 represents zero
  - If **`exp`** == 11...1 and **`frac`** == 00...0, it represents ∞
    - e.g. 1.0/0.0 = –1.0/–0.0 = +∞, 1.0/–0.0 = –1.0/0.0 = –∞
  - If **`exp`** == 11...1 and **`frac`** != 00...0, it represents NaN: "Not a Number"
    - Results from operations with undefined result, e.g. sqrt(–1), $\infty - \infty$, $\infty * 0$

Normalized Values

$V = (-1)^{s} * M * 2^{E}$

s	exp	frac
	k	n

- **Condition: `exp` ≠ `000…0` and `exp` ≠ `111…1`**
- **Exponent coded as *biased* value: *E* = *exp* - *Bias***
  - **`exp`** is an *unsigned* value ranging from 1 to 2^k-2 (k == # bits in **`exp`**)
  - *Bias* = 2^{k-1} - 1
    - Single precision: 127 (so *exp*: 1…254, *E*: -126…127)
    - Double precision: 1023 (so *exp*: 1…2046, *E*: -1022…1023)
  - These enable negative values for E, for representing very small values
- **Significand coded with implied leading 1: *M* = `1.xxx…x`$_2$**
  - **`xxx…x`**: the n bits of **`frac`**
  - Minimum when **`000…0`** (*M* = 1.0)
  - Maximum when **`111…1`** (*M* = 2.0 – ε)
  - Get extra leading bit for "free"

Normalized Encoding Example

$V = (-1)^{s} * M * 2^{E}$

s	exp	frac
	k	n

- **Value:** `float f = 12345.0;`
  - $12345_{10} = 11000000111001_2$
    $= 1.1000000111001_2 \times 2^{13}$ (normalized form)

- **Significand:**

  $M = 1.\underline{1000000111001}_2$

  frac$= \underline{1000000111001}0000000000_2$

- **Exponent:** E = exp - Bias, so exp = E + Bias

  $E = 13$

  $Bias = 127$

  exp $= 140 = 10001100_2$

- **Result:**

s	exp	frac
0	10001100	10000001110010000000000