Floating Point

B&O Readings: 2.4
CSE 361: Introduction to Systems Software

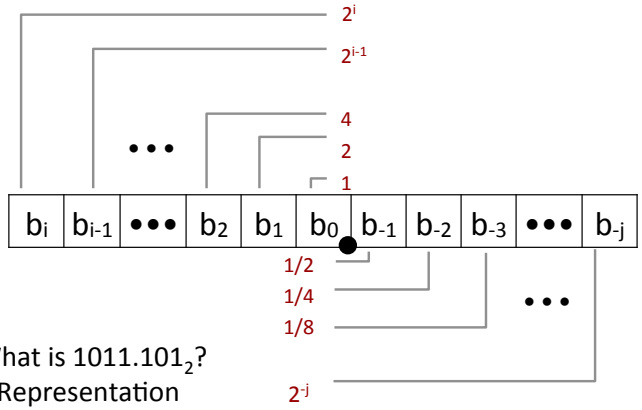
Instructor:
I-Ting Angelina Lee

Note: these slides were originally created by Markus Püschel at Carnegie Mellon University

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Fractional Binary Numbers



What is 1011.101_2?

- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \times 2^k$

Fractional Binary Numbers: Examples

- Value Representation
  5 3/4
  2 7/8
  1 63/64

- Observations
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Fractional Binary Numbers: Examples

Value	Representation
5 3/4	101.11_2
2 7/8	10.111_2
1 63/64	1.111111_2

- Observations
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

- Limitation
  - Can only exactly represent numbers of the form $x/2^k$
  - Other rational numbers have repeating bit representations

Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

Fixed Point Representation

- We might try representing fractional binary numbers by picking a fixed place for an implied binary point
  - "fixed point binary numbers"
- Let's do that, using 8-bit fixed point numbers as an example
  - #1: the binary point is between bits 2 and 3
    $b_7\ b_6\ b_5\ b_4\ b_3\ [.]\ b_2\ b_1\ b_0$
  - #2: the binary point is between bits 4 and 5
    $b_7\ b_6\ b_5\ [.]\ b_4\ b_3\ b_2\ b_1\ b_0$
- The position of the binary point affects the *range* and *precision* of the representation
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers

Fixed Point Pros and Cons

- Pros
  - It's simple. The same hardware that does integer arithmetic can do fixed point arithmetic
    - In fact, the programmer can use ints with an implicit fixed point
    - ints are just fixed point numbers with the binary point to the right of b_0
- Cons
  - There is no good way to pick where the fixed point should be
    - Sometimes you need range, sometimes you need precision – the more you have of one, the less of the other.

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

IEEE Floating Point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs
- Driven by numerical concerns
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard
- 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)

Floating Point Representation

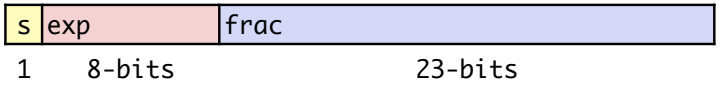
- Numerical Form:

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

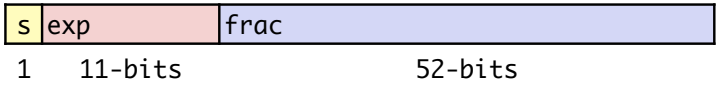
  - Sign bit **s** determines whether number is negative (s=1) or positive (s=0)
  - Significand (mantissa) **M** normally a fractional value in range [1.0,2.0).
  - Exponent **E** weights value by a (possibly negative) power of two
- Encoding
  - MSB s is sign bit **s**
  - exp field *encodes* **E** (is *not equal* to E)
  - frac field *encodes* **M** (is *not equal* to M)

Precisions

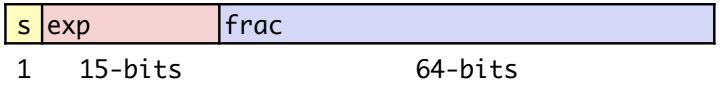
- Single precision: 32 bits



- Double precision: 64 bits

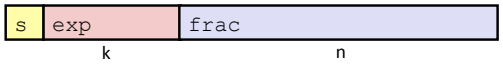


- Extended precision: 80 bits (Intel only)



Normalization and Special Values

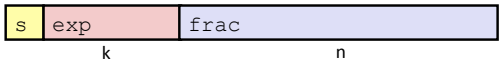
$V = (-1)^s \cdot M \cdot 2^E$



- **"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

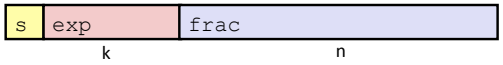
$V = (-1)^s \cdot M \cdot 2^E$



- **"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), ∞ - ∞, ∞·0

Case #1: Normalized Values

$V = (-1)^s \cdot M \cdot 2^E$



- **Condition:** `exp` ≠ `000…0` and `exp` ≠ `111…1`
- Exponent coded as biased value: E = Exp – Bias
  - Exp: unsigned value `exp`
  - Bias = $2^{k-1} - 1$, where k is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: -126…127)
    - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)
  - The bias enables negative values for E, for representing very small values
- Significand coded with implied leading 1: M = $1.xxx\ldots x_2$
  - xxx…x: 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

- **Value:** `Float F = 12345.0;`
  - $12345_{10} = 11000000111001_2$
- **Significand**

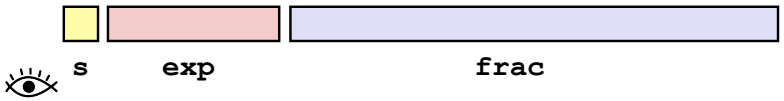
  M = `1.`$\underline{1000000111001}_2$

  `frac` =
- **Exponent, E = Exp – Bias**

  E =

  Bias =

  Exp =
- **Result:**



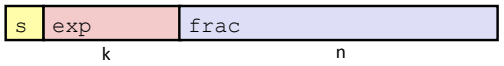
Normalized Encoding Example

- **Value:** `Float F = 12345.0;`
  - $12345_{10} = 11000000111001_2$
    $= 1.1000000111001_2 \times 2^{13}$ (normalized form)
- **Significand**
  - *M* = $1.\underline{1000000111001}_2$
  - **frac** = $\underline{10000001110010000000000}_2$ (implied leading 1)
- **Exponent, E = Exp – Bias**
  - *E* = 13
  - *Bias* = 127
  - *Exp* = 140 = 10001100_2
- **Result:**

0	10001100	10000001110010000000000
s	exp	frac

Case #2: Denormalized Values

$V = (-1)^s \cdot M \cdot 2^E$

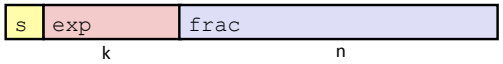


- **Condition:** `exp = 000…0`
- Exponent value: E = 1 – Bias (instead of E = 0 – Bias)
- Significand coded with implied leading 0: M = 0.xxx…x_2
  - `xxx…x`: bits of `frac`
- **Cases 2A:** `exp = 000…0, frac = 000…0`
  - Represents zero value
  - Note distinct values: +0 and –0 (why?)
- **Case 2B:** `exp = 000…0, frac ≠ 000…0`
  - Numbers very close to 0.0

Numbers really close to 0

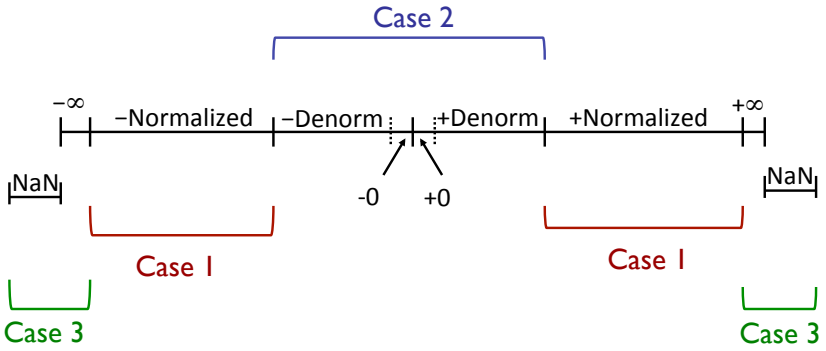
Case #3: Special Values

$V = (-1)^s \cdot M \cdot 2^E$



- **Condition:** `exp = 111…1`
- **Case #3A:** `exp = 111…1, frac = 000…0`
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞
- **Case #3B:** `exp = 111…1, frac ≠ 000…0`
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

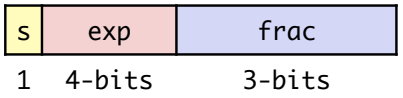
Visualization: Floating Point Encodings



Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Tiny Floating Point Example



- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the `frac`

- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity

Dynamic Range (Positive Only)

$(-1)^s\ M\ 2^E$

	s	exp	frac	E	Value	
	0	0000	000			
	0	0000	001			closest to zero
Denormalized numbers	0	0000	010			
	…					
	0	0000	110			
	0	0000	111			largest denorm
	0	0001	000			smallest norm
	0	0001	001			
	…					
	0	0110	110			
	0	0110	111			closest to 1 below
Normalized numbers	0	0111	000			
	0	0111	001			closest to 1 above
	0	0111	010			
	…					
	0	1110	110			
	0	1110	111			largest norm
	0	1111	000			

Dynamic Range (Positive Only)

$(-1)^s\ M\ 2^E$

M = 0.frac
E = 1 – Bias

	s	exp	frac	E	Value	
	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
Denormalized numbers	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
	0	0001	000			smallest norm
	0	0001	001			
	…					
	0	0110	110			
	0	0110	111			closest to 1 below
Normalized numbers	0	0111	000			
	0	0111	001			closest to 1 above
	0	0111	010			
	…					
	0	1110	110			
	0	1110	111			largest norm
	0	1111	000			

Dynamic Range (Positive Only)

$(-1)^s\ M\ 2^E$

	s	exp	frac	E	Value	
	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
Denormalized numbers	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
M = 1.frac	0	0001	001	-6	9/8*1/64 = 9/512	
E = exp – Bias	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
Normalized numbers	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest norm
	0	1111	000			

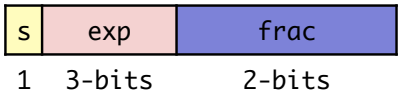
Dynamic Range (Positive Only)

$(-1)^s\ M\ 2^E$

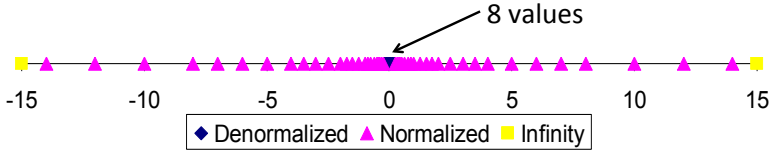
	s	exp	frac	E	Value	
	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
Denormalized numbers	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
M = 1.frac	0	0001	001	-6	9/8*1/64 = 9/512	
E = exp – Bias	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
Normalized numbers	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest norm
	0	1111	000	n/a	inf	

Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 23-1-1 = 3

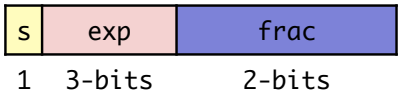


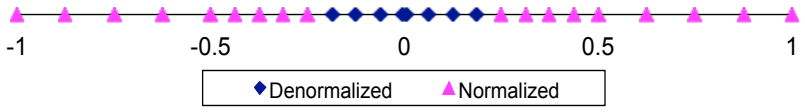
- Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

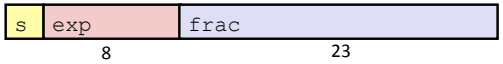
- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3





Interesting Numbers (Single Precision)

$V = (-1)^s \cdot M \cdot 2^E$



Description	*exp*	*frac*	*Numeric Value*
■ Zero	00…00	00…00	0.0
■ Smallest Pos. Denorm.			
■ Largest Denormalized			
■ Smallest Pos. Normalized			
■ One			
■ Largest Normalized			

Interesting Numbers {single,double}

Description	*exp*	*frac*	*Numeric Value*
■ Zero	00…00	00…00	0.0
■ Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
▪ Single ≈ 1.4×10^{-45}			
▪ Double ≈ 4.9×10^{-324}			
■ Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
▪ Single ≈ 1.18×10^{-38}			
▪ Double ≈ 2.2×10^{-308}			
■ Smallest Pos. Normalized	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
▪ Just larger than largest denormalized			
■ One	01…11	00…00	1.0
■ Largest Normalized	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
▪ Single ≈ 3.4×10^{38}			
▪ Double ≈ 1.8×10^{308}			

Special Properties of Encoding

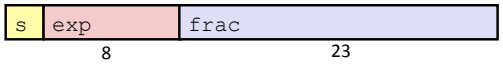
- FP Zero Same as Integer Zero
  - All bits = 0
  - There is a -0.0 and a 0.
- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider −0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Floating Point Operations: Basic Idea

$V = (-1)^s \cdot M \cdot 2^E$



- $x +_f y = \text{Round}(x + y)$
  - E could be very different
  - the binary point needs to line up
- $x \times_f y = \text{Round}(x \times y)$
  - need to ensure that the resulting exponent is still in range

- Basic idea
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into `frac`

Rounding

- Rounding Modes (illustrate with $ rounding)

	$1.40	$1.60	$1.50	$2.50	–$1.50
Towards zero	$1	$1	$1	$2	–$1
Round down (–∞)	$1	$1	$1	$2	–$2
Round up (+∞)	$2	$2	$2	$3	–$1
Nearest Even (default)	$1	$2	$2	$2	–$2

- Floating point calculation uses round-to-nearest even.
  - Why?

Closer Look at Round-To-Even

- Default Rounding Mode
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated

- Applying to Other Decimal Places / Bit Positions
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth → 1.2X

    1.2349999
    1.2350001
    1.2350000
    1.2450000

Closer Look at Round-To-Even

- Default Rounding Mode
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated

- Applying to Other Decimal Places / Bit Positions
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

Rounding Binary Numbers

- Binary Fractional Numbers
  - "Even" when least significant bit is 0
  - "Half way" when bits to right of rounding position = $100\ldots_2$
- Examples
  - Round to nearest 1/4 (2 bits right of binary point)

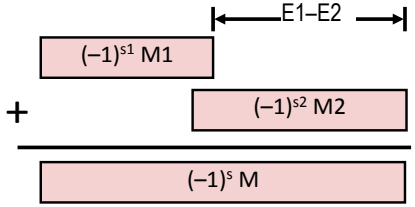
Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

FP Multiplication

- $(-1)^{s1}$ M1 2^{E1} x $(-1)^{s2}$ M2 2^{E2}
- Exact Result: $(-1)^s$ M 2^E
  - Sign s: s1 ^ s2
  - Significand M: M1 x M2
  - Exponent E: E1 + E2
- Fixing
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit `frac` precision
- Implementation
  - Biggest chore is multiplying significands

Floating Point Addition

- $(-1)^{s1}$ M1 2^{E1} + $(-1)^{s2}$ M2 2^{E2}
  - Assume E1 > E2
- Exact Result: $(-1)^s$ M 2^E
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: E1



- Fixing
  - If M ≥ 2, shift M right, increment E
  - if M < 1, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit `frac` precision

Mathematical Properties of FP Add

- Compare to those of Abelian Group
  - Commutative? Yes
  - Associative? No
    - Overflow and inexactness of rounding
    - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`
  - 0 is additive identity? Yes
  - Every element has additive inverse? Yes
    - Yes, except for infinities & NaNs Almost
- Monotonicity
  - a ≥ b ⇒ a+c ≥ b+c? Almost
    - Except for infinities & NaNs

Mathematical Properties of FP Mult

- Compare to Commutative Ring
  - Multiplication Commutative? Yes
  - Multiplication is Associative? No
    - Possibility of overflow, inexactness of rounding
    - Ex: `(1e20*1e20)*1e-20= inf, 1e20*(1e20*1e-20)= 1e20`
  - 1 is multiplicative identity? Yes
  - Multiplication distributes over addition? No
    - Possibility of overflow, inexactness of rounding
    - `1e20*(1e20-1e20)= 0.0, 1e20*1e20 – 1e20*1e20 = NaN`
- Monotonicity
  - $a \geq b \;\&\; c \geq 0 \Rightarrow a * c \geq b * c$? Almost
    - Except for infinities & NaNs

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Floating Point in C

- C offers two levels of precision

`float`	single precision (32-bit)
`double`	double precision (64-bit)

- Default rounding mode is round-to-even
- `#include <math.h>` to get `INFINITY` and `NAN` constants
- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results
  - Just avoid them!

Floating Point in C

- Conversions between data types:
  - Casting between `int`, `float`, and `double` changes the bit representation!!
  - `int` → `float`
    - May be rounded; overflow not possible
  - `int` → `double` or `float` → `double`
    - Exact conversion, as long as int has ≤ 53-bit word size
  - `double` or `float` → `int`
    - Truncates fractional part (rounded toward zero)
    - Not defined when out of range or NaN: generally sets to Tmin

Floating Point Puzzles

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NaN

```
A. x == (int)(float) x
B. x == (int)(double) x
C. f == (float)(double) f
D. d == (float) d
E. f == -(-f);
F. 2.0/3 == 2/3.0
G. d * d >= 0.0
H. (f+d)-f == d
```


Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form M x 2^E
- One can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
  - Makes life difficult for compilers & serious numerical applications programmers