Announcement

- Updated lab hours / locations are posted on course website!
- Course grade cutoffs

Floating Points

B&O Readings: 2.4

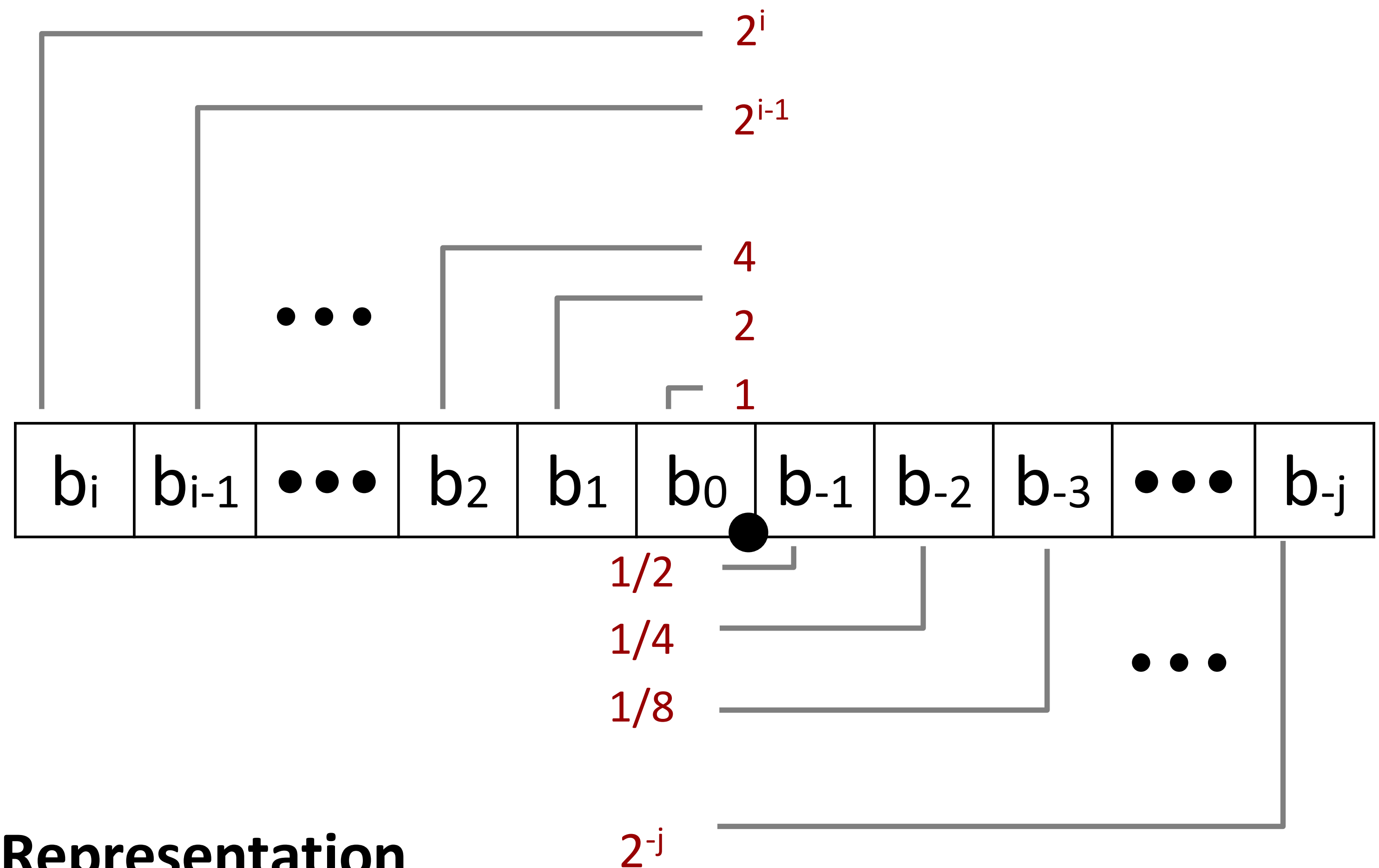
CSE 361: Introduction to Systems Software

Instructor:

I-Ting Angelina Lee

Q: How Might A Computer Represent a Real Number?

Fractional Binary Numbers



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

Examples of Fractional Binary Numbers

- Value — Representation

Value	Representation
5 3/4	$\mathtt{101.11}_2$
2 7/8	$\mathtt{10.111}_2$
1 7/16	$\mathtt{1.0111}_2$

- Observations
  - Divide by 2 by shifting right (unsigned)
  - 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 $1.0 - \varepsilon$

Fixed Point Representation

- **We might try representing fractional binary numbers by picking a fixed place for an implied binary point**
  - "fixed point binary numbers"

- **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

- **Problem: 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.

Fixed Point vs. Floating Point

Example: 4-digit positive decimal fixed point versus floating point:

- **Fix point, say fixed at xxx.x:**
  - range: 0.1 – 999.9
- **Floating point:**
  - $x_1x_2x_3y_1$ that encodes $x \cdot 10^y$
  - X can range between 0 – 999 and
    y can range between -4 – 5
  - You can choose between range versus precision.

Representable Numbers: Limitation Due to Fixed Width

- 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]…`$_2$
  - 1/5 `0.001100110011[0011]…`$_2$
  - 1/10 `0.0001100110011[0011]…`$_2$

Recap: What We Learned Thus Far

- **The difference between fixed point and floating point**
  - Floating point allows one to trade off between range and precision
- **Fundamental limitation of the fixed-width binary representation of real values:**
  - Some values cannot be represented precisely!

Today: Floating Point

- Background: Fractional binary numbers
- **IEEE floating point standard: Definition**
- Floating point operations and rounding
- Floating point in C

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

- **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 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)
  - Mantissa **M** (or Significand) represents a fractional value.
  - 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)

s	exp	frac

Precisions

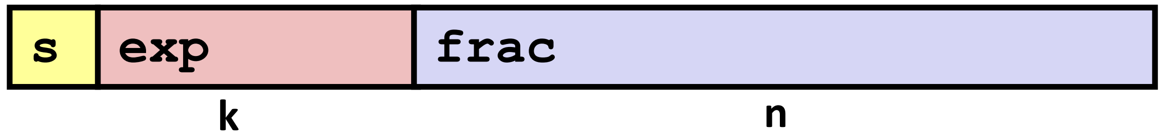
- **Single precision: 32 bits**

- **Double precision: 64 bits**

- **Extended precision: 80 bits**

Three Kinds of Floating Point Values

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



- **"Normalized" values**
  - most values
- **"Denormalized" values:**
  - special **exp** field
  - for values close to 0 or equals to 0
- **Special values: reserved for values +/- infinity, NaN**
  - special **exp** field
  - +/- infinity: when results overflow (including dividing by 0)
    - e.g. $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -1.0/0.0 = -\infty$
  - **NaN** (Not a Number): from operations with undefined results
    - e.g. $\text{sqrt}(-1)$, $\infty - \infty$, $\infty \cdot 0$

Case #1: Normalized Values

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

s	exp	frac
	k	n

- **Condition: `exp` $\neq$ `000…0` and `exp` $\neq$ `111…1`**
- **Mantissa coded with implied leading 1: M = $1.xxx…x_2$**
  - 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
  - `xxx…x`: bits of `frac` (don't bother to store the leading 1)
  - Range from [1, 2.0)
- **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)
- **We can almost compare floating points using integer comparison**

Normalized Encoding Example (32 bits)

- **Value:** `float f = 12345.0;`
  - $12345_{10} = 11000000111001_2$

- **Mantissa**

  M =

  `frac` =

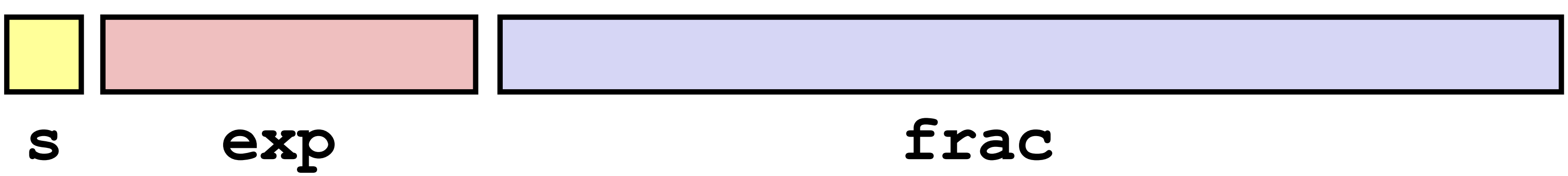
- **Exponent, E = exp – bias (bias = 127)**

  E =

  bias =

  exp =

- **Result:**



Normalized Encoding Example (32 bits)

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

- **Mantissa, M**

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

  $\texttt{frac} = \underline{\texttt{10000001110010000000000}}_2$ (implied leading 1)

- **Exponent, E = exp – bias (bias = 127)**

  $E = 13$

  $bias = 127$

  $exp = 13+127 = 140 = \texttt{10001100}_2$

- **Result:**

s	exp	frac
0	10001100	10000001110010000000000

Recap: Normalized Values

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

s	exp	frac
	k	n

- **Condition:** `exp` $\neq$ `000…0` and `exp` $\neq$ `111…1`
- **Exponent coded as biased value: E = exp – bias**
  - exp: interpret as unsigned value with bias = 127
  - E = -126 … 127 for single precision, -1022..1023 for double precision
- **Mantissa coded with implied leading 1: M = `1.xxx…x`$_2$**

Q: Given the normalized encoding, what is the smallest positive normalized value a float in C can represent?

A: with M = 1.0…0, and E = 1-127 = -126, V = 2^{-126}

Want more precision when we get closer to 0!

Case #2: Denormalized Values (For Zero & Numbers REALLY Close to Zero)

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

s	exp	frac
	k	n

- **Condition:** `exp = 000…0`
- **Special Case:** `exp = 000…0, frac = 000…0`
  - Represents zero value
  - Note distinct values: +0 and –0
- **Exponent value: E = 1 – Bias (instead of E = exp (0) – bias)**
  - E is always -126 for single precision and -1022 for double precision
- **Mantissa coded with implied leading 0: $M = 0.xxx…x_2$**
  - Max M = 0.111…1, which is 1-Ɛ
  - Combining with E = -126, this provides smooth transition from normalized values to denormalized values.

Case #3: Special Values

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

s	exp	frac
	k	n

- **Condition:** `exp = 111…1`

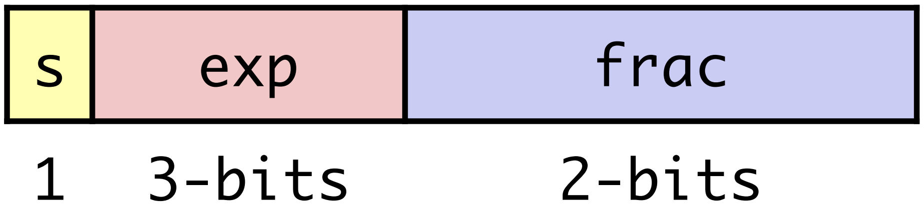
- **Case #3A:** `exp = 111…1, frac = 000…0`
  - Represents value ∞ (infinity)
  - Operation that overflows (positive and negative)
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
- **Case #3B:** `exp = 111…1, frac ≠ 000…0`
  - Not-a-Number (NaN)
  - Represents case when no numeric value cannot be determined
  - Bits in `frac` are used to store reasons for NaN
  - E.g., sqrt(–1), $\infty - \infty$, $\infty \times 0$

Special Properties of Encoding

- **FP Zero Same as Integer Zero**
  - All bits = 0
  - There is a -0.0 and a +0.0

- **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? (False)
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Tiny Floating Point Example

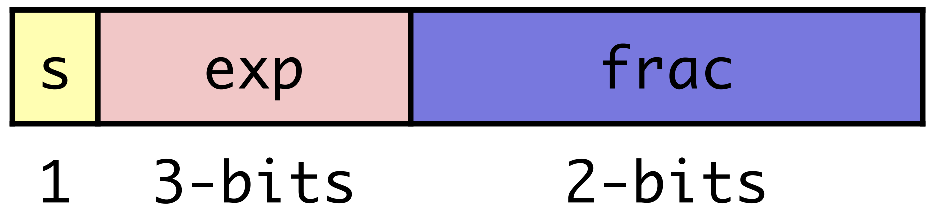


- **6-bit Floating Point Representation**
  - the sign bit is in the most significant bit
  - the next three bits are the exponent, with a bias of 3
  - the last two bits are the `frac`

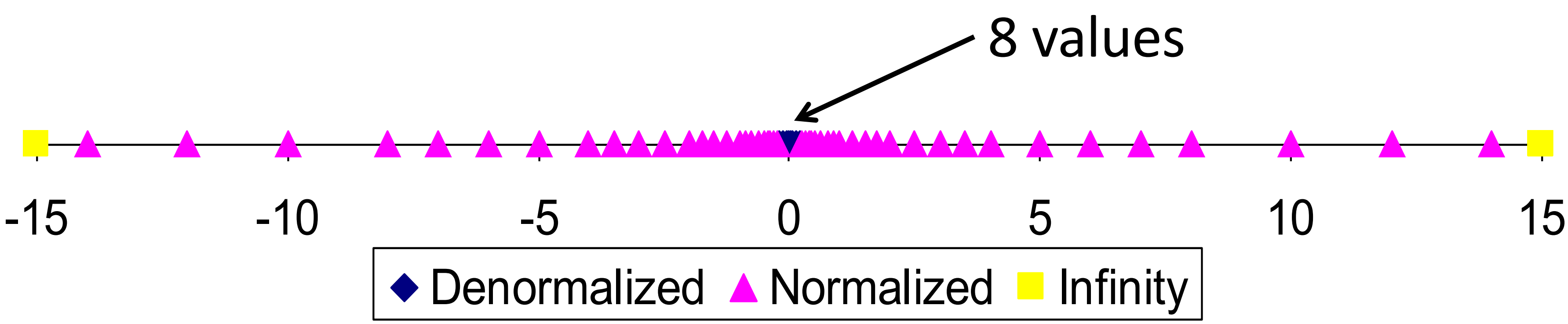
- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

Distribution of Values

- **6-bit IEEE-like format**
  - 3 exponent bits
  - 2 fraction bits
  - Bias is 2^{3-1}-1 = 3

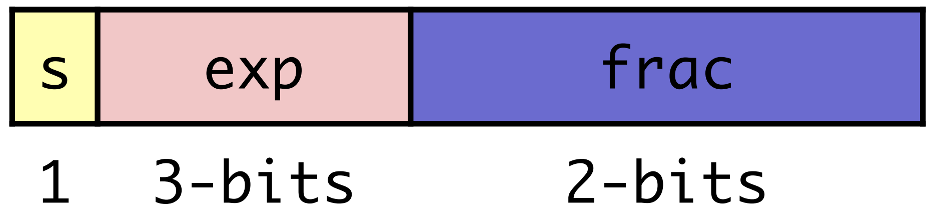


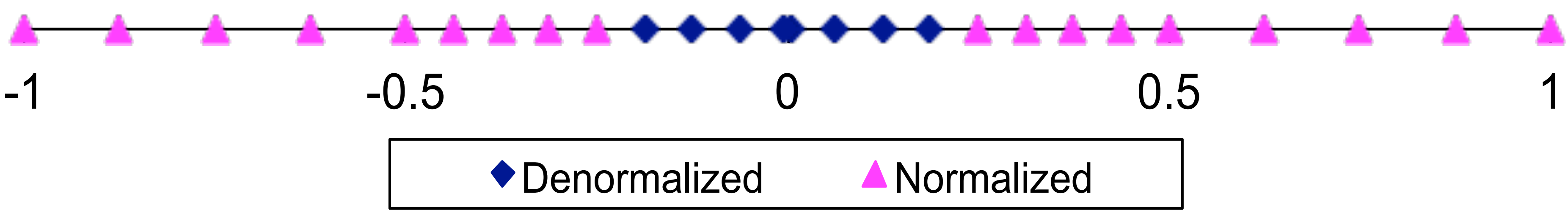
- **Notice how the distribution gets denser toward zero.**



Distribution of Values (close-up view)

- **6-bit IEEE-like format**
  - 3 exponent bits
  - 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$





Gradual Underflow

Recap: What We Learned Thus Far

- **The IEEE floating point representation:**
  - Normalized (most values)
  - Denormalized (0s or values very close to 0)
  - Special values (+/- infinity and NaN)

- **Understanding the floating point representation helps you understand the mathematical properties of floating point operations and how they interact with other integer data types.**

Puzzle

s	exp	frac
1	8-bits	23-bits

- **What's the smallest positive integer value that cannot be represented precisely using `float` in C?**
- **Answer: $2^{24} + 1$**
- **In generally, assuming we have enough `exp` bits (i.e., within range), the answer would be $2^{(n+1)} + 1$ for n-bit `frac`.**

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- **Floating point operations and rounding**
- Floating point in C

Floating Point Multiplication

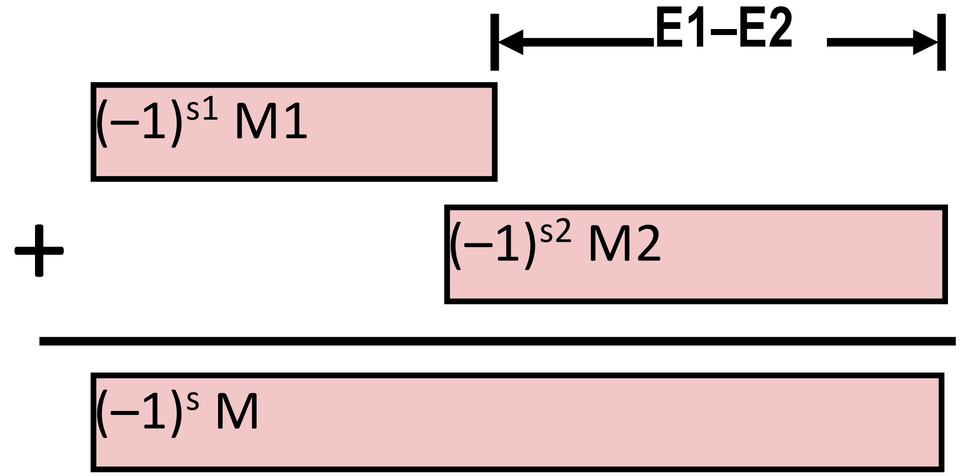
- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \times\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s: s1 ^ s2
  - Mantissa M: M1 x M2
  - Exponent E: E1 + E2

- **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 the Mantissas

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, mantissa 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

Floating Point Operations: Basic Idea

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

s	exp	frac
	8	23

- $\mathtt{x} +_f \mathtt{y} = \mathtt{Round(x + y)}$
  - E could be very different
  - the binary point needs to line up
- $\mathtt{x} \times_f \mathtt{y} = \mathtt{Round(x \times y)}$
  - 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`**

IEEE Rounding Modes

- **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

- **Round to nearest Even:**
  - When more than halfway, round up; when less than halfway, round down.
  - When exactly halfway between two possible values, round it so that least significant digit is even
  - The default rounding mode.
  - Why? So that we don't introduce statistical bias.
  - All others are statistically biased

Rounding Binary Numbers

- **When exactly halfway between two possible values**
  - Round so that least significant digit is even
- **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