Floating Point

15-213/18-213/14-513/15-513: Introduction to Computer Systems
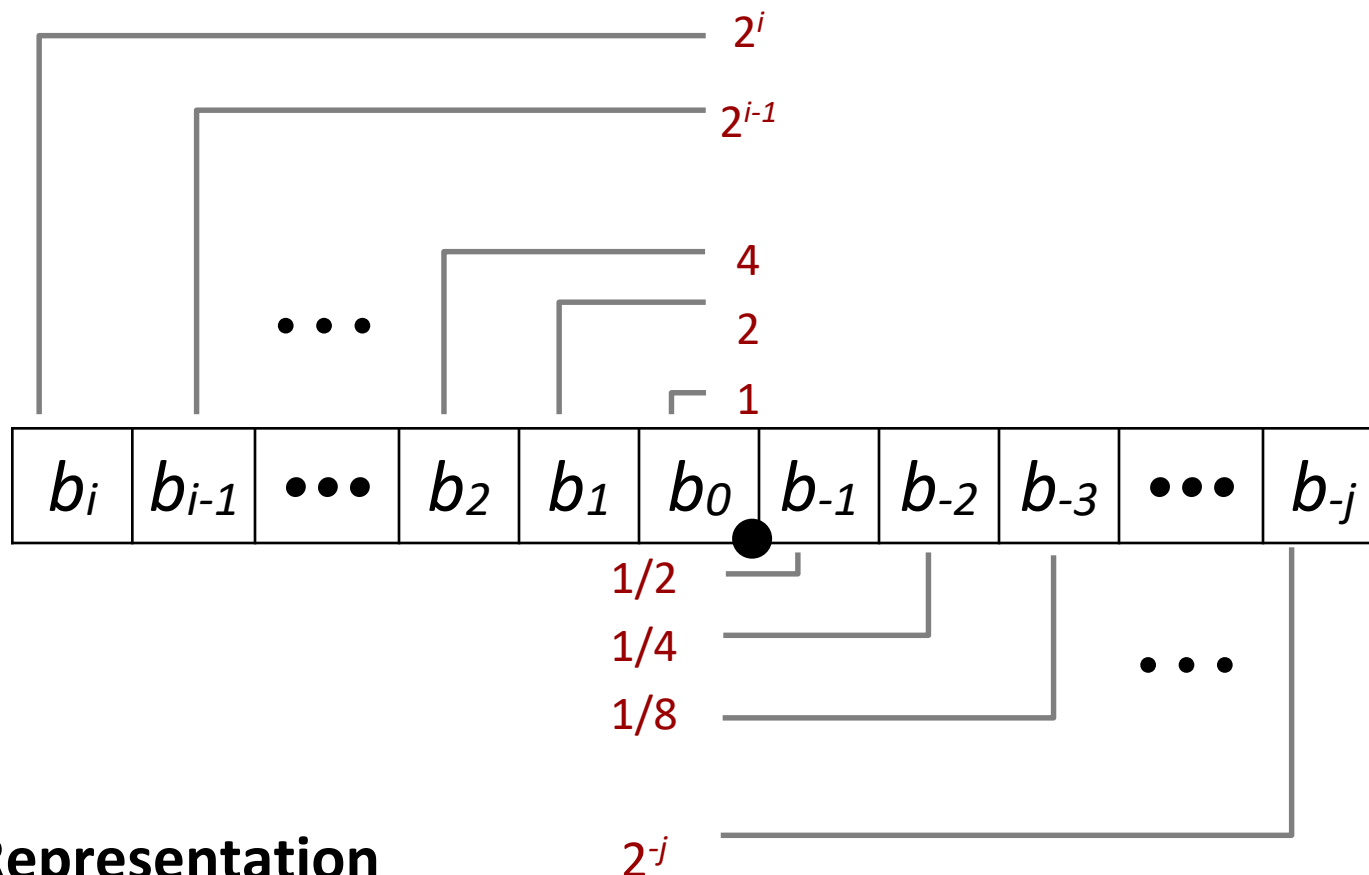4th Lecture, Sept. 6, 2018

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?**

Fractional Binary Numbers

- **Representation**
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \times 2^k$$

Fractional Binary Numbers: Examples

- **Value**   **Representation**

  5 3/4   = 23/4 101.11_2 = 4 + 1 + 1/2 + 1/4

  2 7/8   = 23/8 10.111_2 = 2 + 1/2 + 1/4 + 1/8

  1 7/16 = 23/16 1.0111_2 = 1 + 1/4 + 1/8 + 1/16

- **Observations**

  - Divide by 2 by shifting right (unsigned)

  - Multiply by 2 by shifting left

  - Numbers of form $0.111111..._2$ are just below 1.0

    - $1/2 + 1/4 + 1/8 + ... + 1/2^i + ...$ ➞ 1.0

    - Use notation 1.0 − ε

Representable Numbers

- **Limitation #1**
  - 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]`…$_2$

- **Limitation #2**
  - Just one setting of binary point within the *w* bits
    - Limited range of numbers (very small values? very large?)

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
  - Some CPUs don't implement IEEE 754 in full
    e.g., early GPUs, Cell BE processor

- **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

Floating Point Representation

Example:
$15213_{10} = (-1)^0 \times 1.1101101101101_2 \times 2^{13}$

- **Numerical Form:**

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

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

- **Encoding**

  - 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)

s	exp	frac

Precision options

- **Single precision: 32 bits**
  ≈ 7 decimal digits, $10^{\pm 38}$

s	exp	frac
1	8-bits	23-bits

- **Double precision: 64 bits**
  ≈ 16 decimal digits, $10^{\pm 308}$

s	exp	frac
1	11-bits	52-bits

- **Other formats: half precision, quad precision**

Three "kinds" of floating point numbers

s	exp	frac

1   **e-bits**   **f-bits**

00…00	exp ≠ 0 and exp ≠ 11…11	11…11

denormalized   normalized   special
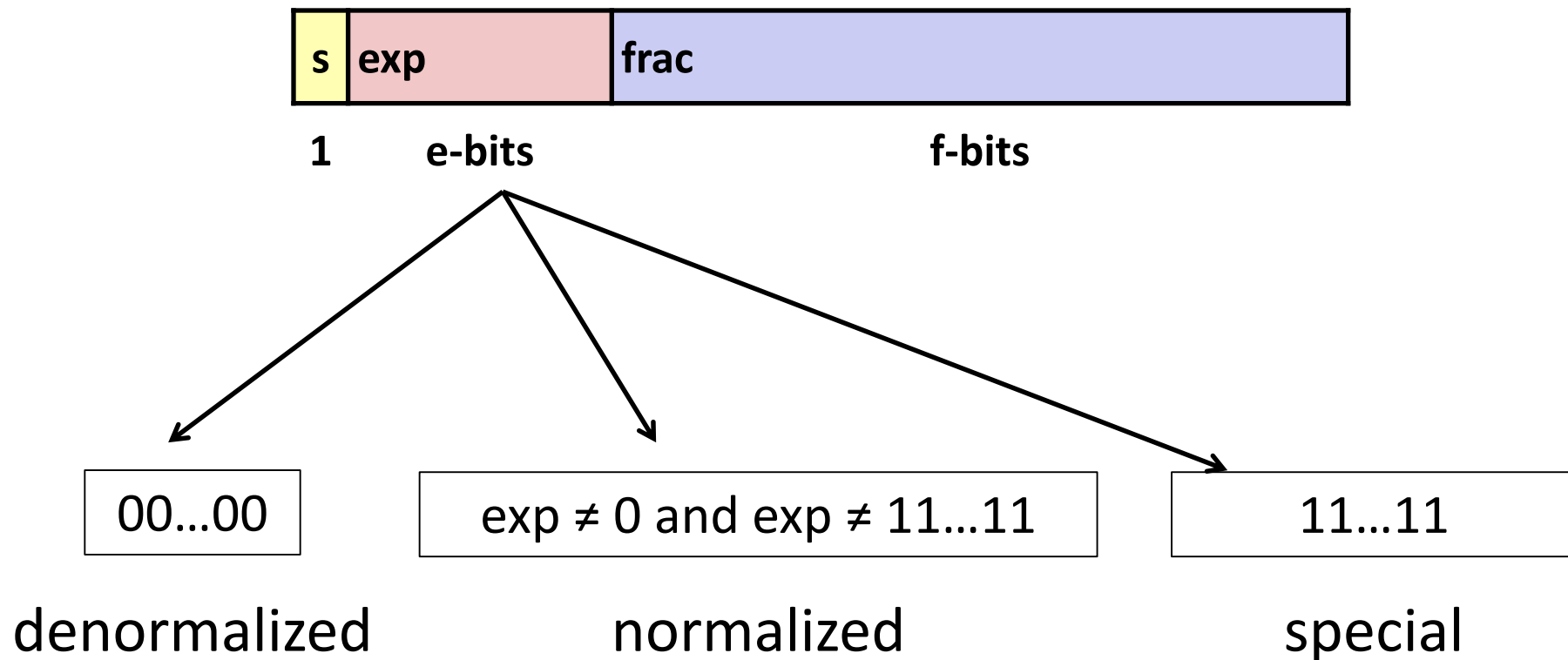
"Normalized" Values

$$v = (-1)^s \, M \, 2^E$$

- **When: `exp` ≠ 000…0 and `exp` ≠ 111…1**

- **Exponent coded as a *biased* value: E = `exp` − *Bias***

  - *exp*: unsigned value of exp field

  - *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)

- **Significand coded with implied leading 1: M = 1.xxx…x$_2$**

  - xxx…x: bits of frac field

  - Minimum when `frac`=000…0 (M = 1.0)

  - Maximum when `frac`=111…1 (M = 2.0 − ε)

  - Get extra leading bit for "free"

Normalized Encoding Example

$$v = (-1)^s \, M \, 2^E$$
$$E = \text{exp} - \textit{Bias}$$

- **Value: `float F = 15213.0;`**
  - 15213_{10} = 11101101101101_2
    - = $1.1101101101101_2 \times 2^{13}$

- **Significand**

  M = $1.\underline{1101101101101}_2$
  `frac=` $\underline{1101101101101}0000000000_2$
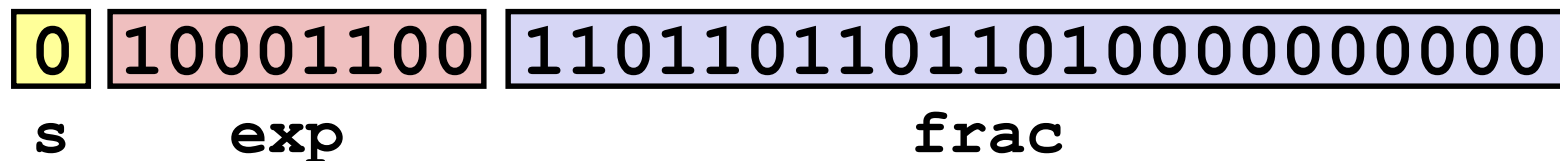
- **Exponent**

  E =   13
  $Bias$ =   127
  `exp` =   140   = 10001100_2

- **Result:**

0	10001100	11011011011010000000000
s	exp	frac

Denormalized Values

$$v = (-1)^s \, M \, 2^E$$
$$E \;=\; 1 - Bias$$

- **Condition:** exp = 000…0

- **Exponent value:** E = 1 − Bias (instead of `exp` − *Bias*)   (why?)

- **Significand coded with implied leading 0:** M = 0.xxx…x$_2$

  - `xxx`…`x`: bits of `frac`

- **Cases**

  - `exp` = 000…0, `frac` = 000…0

    - Represents zero value

    - Note distinct values: +0 and −0 (why?)

  - `exp` = 000…0, `frac` ≠ 000…0

    - Numbers closest to 0.0

    - Equispaced

Special Values

- **Condition: `exp` = 111…1**

- **Case: `exp` = 111…1, `frac` = 000…0**
  - **Represents value ∞ (infinity)**
  - Operation that overflows
  - Both positive and negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

- **Case: `exp` = 111…1, `frac` ≠ 000…0**
  - **Not-a-Number (NaN)**
  - Represents case when no numeric value can be determined
  - E.g., $\text{sqrt}(-1)$, $\infty - \infty$, $\infty \times 0$

C float Decoding Example

$$v = (-1)^s \, M \, 2^E$$
$$E = \texttt{exp} - Bias$$

float: `0xC0A00000`

$Bias = 2^{k-1} - 1 = 127$

binary: ___ ___ ___ ___ ___ ___ ___ ___

| 1 | 8-bits | 23-bits |

E =

S =

M =

v = (−1)ˢ M 2ᴱ =

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

C float Decoding Example

$$v = (-1)^S \, M \, 2^E$$
$$E = \texttt{exp} - Bias$$

float: `0xC0A00000`

binary: <u>1</u><u>100</u> <u>0000</u> <u>1</u><u>010</u> <u>0000</u> <u>0000</u> <u>0000</u> <u>0000</u> <u>0000</u>

1	1000 0001	010 0000 0000 0000 0000 0000
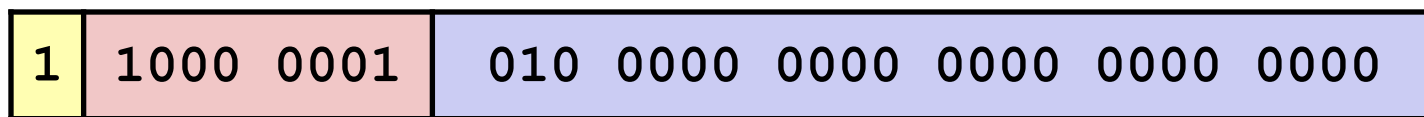
1   8-bits   23-bits

E =

S =

M = 1.

v = (−1)^S M 2^E =

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

C float Decoding Example

$$v = (-1)^s \, M \, 2^E$$
$$E = \texttt{exp} - Bias$$

float: `0xC0A00000`

$Bias = 2^{k-1} - 1 = 127$

binary: `1100 0000 1010 0000 0000 0000 0000 0000`

1	1000 0001	010 0000 0000 0000 0000 0000
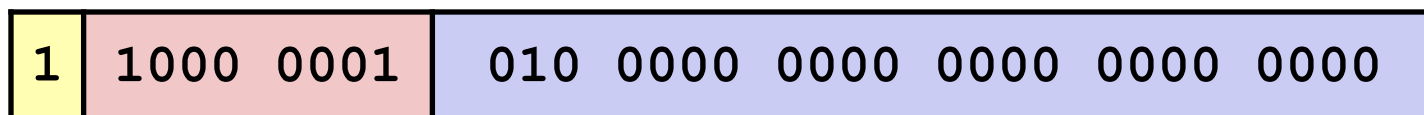
1   8-bits   23-bits

E = `exp` − Bias = 129 − 127 = 2 (decimal)

S = 1 -> negative number

M = `1.010 0000 0000 0000 0000 0000`

    = `1 + 1/4 = 1.25`

v = (−1)^s M 2^E = (-1)^1 * 1.25 * 2^2 = -5

| Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Visualization: Floating Point Encodings
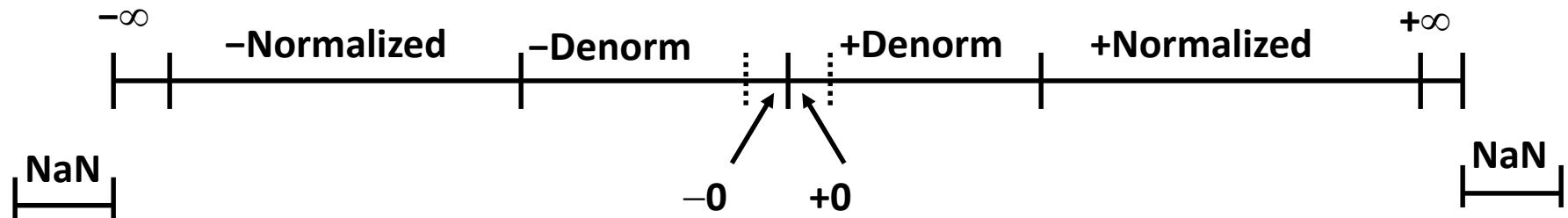
Today: Floating Point

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

Tiny Floating Point Example

s	exp	frac
1	4-bits	3-bits
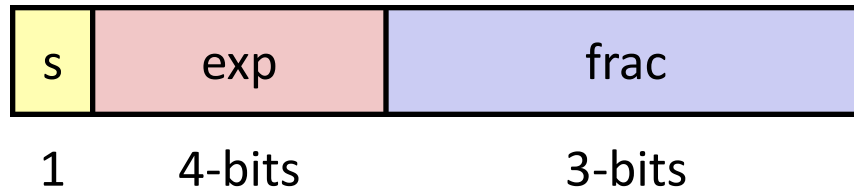
- **8-bit Floating Point Representation**
  - the sign bit is in the most significant bit
  - the next four bits are the `exp`, with a bias of 7
  - the last three bits are the `frac`

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

Dynamic Range (s=0 only)

$$v = (-1)^s \, M \, 2^E$$
norm: E = exp − Bias
denorm: E = 1 − Bias

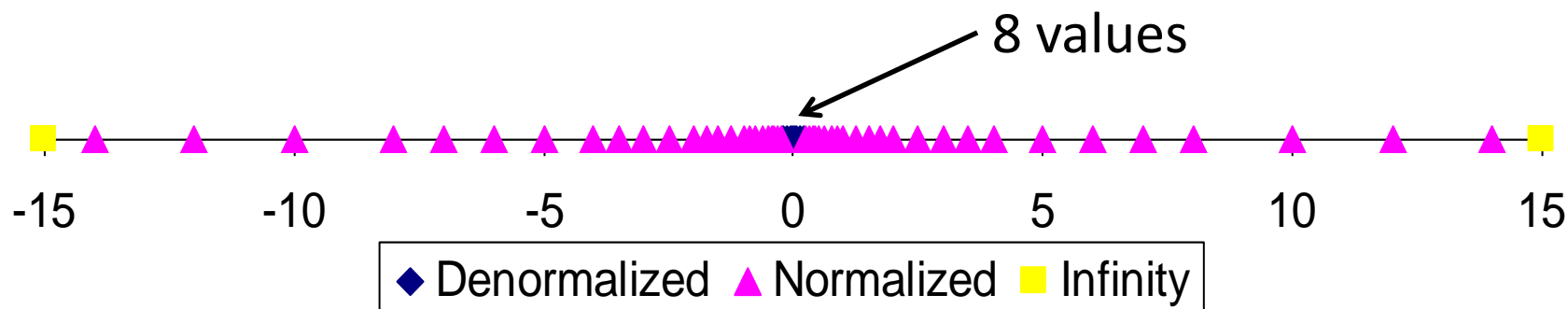| s exp frac	E	Value	
0 0000 000	−6	0	
0 0000 001	−6	1/8*1/64 = 1/512	closest to zero
0 0000 010	−6	2/8*1/64 = 2/512	$(-1)^0(0+1/4)*2^{-6}$
…			
0 0000 110	−6	6/8*1/64 = 6/512	
0 0000 111	−6	7/8*1/64 = 7/512	largest denorm
0 0001 000	−6	8/8*1/64 = 8/512	smallest norm
0 0001 001	−6	9/8*1/64 = 9/512	$(-1)^0(1+1/8)*2^{-6}$
…			
0 0110 110	−1	14/8*1/2 = 14/16	
0 0110 111	−1	15/8*1/2 = 15/16	closest to 1 below
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	

Denormalized numbers

Normalized numbers

Distribution of Values

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

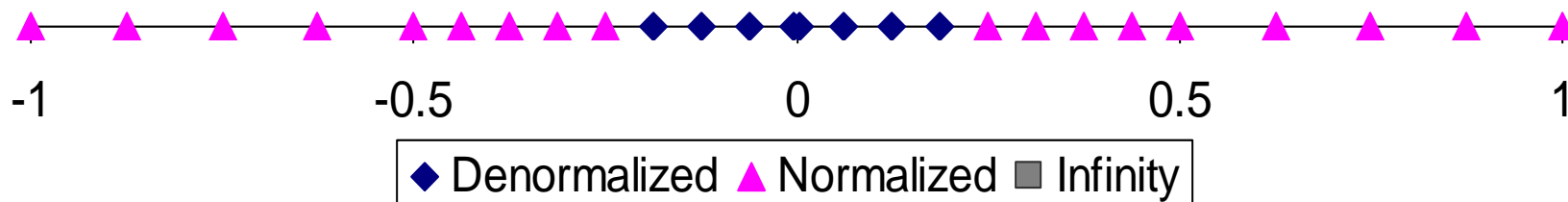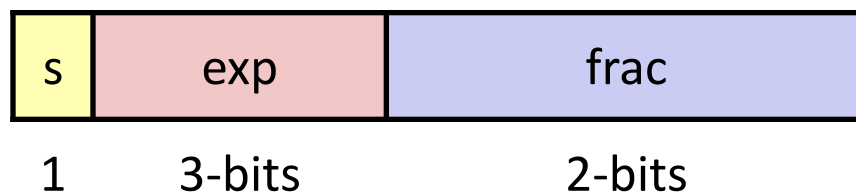| s	exp	frac
1	3-bits	2-bits
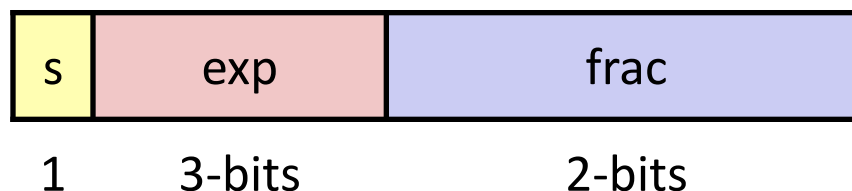
- **Notice how the distribution gets denser toward zero.**

8 values

◆ Denormalized ▲ Normalized ■ Infinity

Distribution of Values (close-up view)

■ **6-bit IEEE-like format**

  ■ e = 3 exponent bits

  ■ f = 2 fraction bits

  ■ Bias is 3

s	exp	frac
1	3-bits	2-bits

◆ Denormalized ▲ Normalized ■ Infinity

Special Properties of the IEEE Encoding

- **FP Zero Same as Integer Zero**
  - All bits = 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? The answer is complicated.
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Quiz Time!

Check out:

https://canvas.cmu.edu/courses/5835

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

- **`x +`$_f$` y = Round(x + y)`**

- **`x ×`$_f$` y = Round(x × y)`**

- **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 ↓

*Round to nearest, but if half-way in-between then round to nearest even

Closer Look at Round-To-Even

- **Default Rounding Mode**
  - Hard to get any other kind without dropping into assembly
  - C99 has support for rounding mode management
  - 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

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)

Rounding Binary Numbers

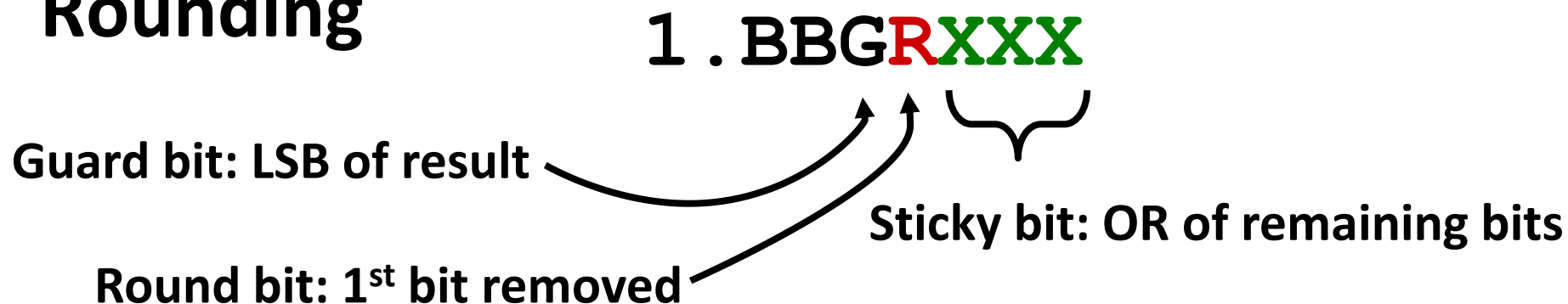- ## Binary Fractional Numbers

  - "Even" when least significant bit is 0
  - "Half way" when bits to right of rounding position = $100..._2$

- ## 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

Rounding

1.BBG**R**XXX

Guard bit: LSB of result

Sticky bit: OR of remaining bits

Round bit: 1st bit removed

- **Round up conditions**
  - Round = 1, Sticky = 1 ⇀ > 0.5
  - Guard = 1, Round = 1, Sticky = 0 ⇀ Round to even

Fraction	GRS	Incr?	Rounded
1.0000000	000	N	1.000
1.1010000	100	N	1.101
1.0001000	010	N	1.000
1.0011000	110	Y	1.010
1.0001010	011	Y	1.001
1.1111100	111	Y	10.000

FP Multiplication

- $(-1)^{s1} \, M1 \, 2^{E1} \quad x \quad (-1)^{s2} \, M2 \, 2^{E2}$

- **Exact Result: $(-1)^{s} \, M \, 2^{E}$**

  - Sign s: $s1 \wedge s2$
  - Significand M: $M1 \times M2$
  - Exponent E: $E1 + E2$

- **Fixing**

  - If $M \geq 2$, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit `frac` precision

- **Implementation**

  - Biggest chore is multiplying significands

4 bit significand: `1.010*2`^{2} **`x`** `1.110*2`^{3} `= 1``0``.0011*2`^{5}
                        `= 1.000``11``*2`^{6} `= 1.00``1``*2`^{6}

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 $M \geq 2$, 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

Get binary points lined up
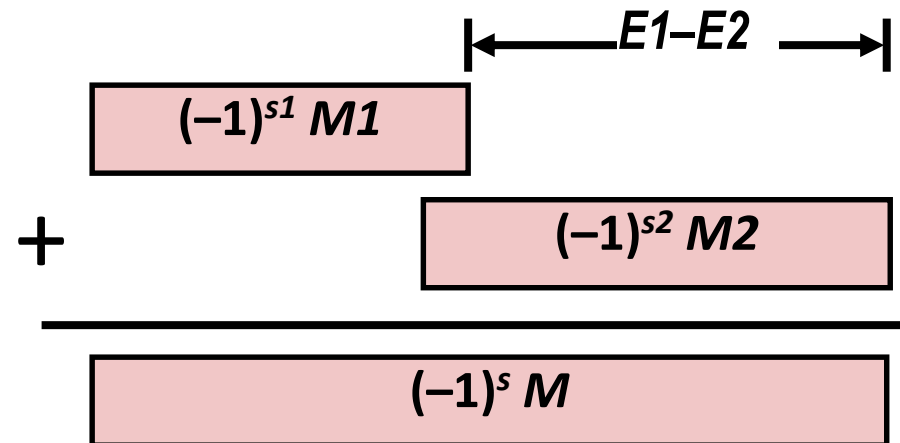

```
1.010*2² + 1.110*2³ = (0.1010 + 1.1100)*2³
= 10.0110 * 2³ = 1.00110 * 2⁴ = 1.010 * 2⁴
```

Mathematical Properties of FP Add

- **Compare to those of Abelian Group**
  - Closed under addition?   *Yes*
    - But may generate infinity or NaN
  - 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?   *Almost*
    - Yes, except for infinities & NaNs

- **Monotonicity**
  - a ≥ b ⇒ a+c ≥ b+c?   *Almost*
    - Except for infinities & NaNs

Mathematical Properties of FP Mult

- **Compare to Commutative Ring**
  - Closed under multiplication?   *Yes*
    - But may generate infinity or NaN
  - 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 Guarantees Two Levels**
    - `float`   single precision
    - `double`   double precision

- **Conversions/Casting**
    - Casting between `int`, `float`, and `double` changes bit representation
    - `double`/`float` → `int`
        - Truncates fractional part
        - Like rounding toward zero
        - Not defined when out of range or NaN: Generally sets to TMin
    - `int` → `double`
        - Exact conversion, as long as `int` has ≤ 53 bit word size
    - `int` → `float`
        - Will round according to rounding mode

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

- `x == (int)(float) x` ✗
- `x == (int)(double) x` ✓
- `f == (float)(double) f` ✓
- `d == (double)(float) d` ✗
- `f == -(-f);` ✓
- `2/3 == 2/3.0` ✗
- `d < 0.0    ⇒    ((d*2) < 0.0)` ✓
- `d > f      ⇒    -f > -d` ✓
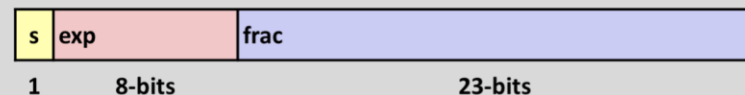- `d * d >= 0.0` ✓
- `(d+f)-d == f` ✗

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**

  - Violates associativity/distributivity

  - Makes life difficult for compilers & serious numerical applications programmers
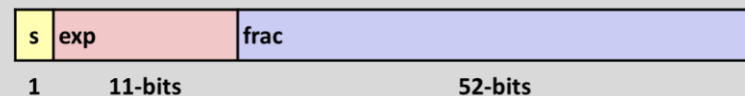
Additional Slides

Creating Floating Point Number

- **Steps**

s	exp	frac

1   4-bits   3-bits

  - Normalize to have leading 1
  - Round to fit within fraction
  - Postnormalize to deal with effects of rounding
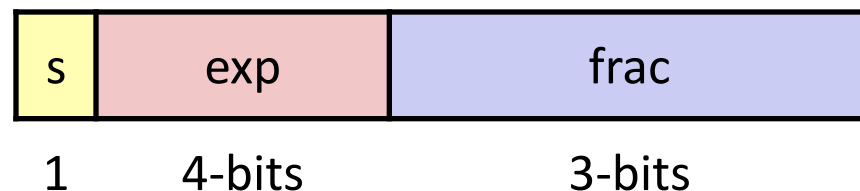
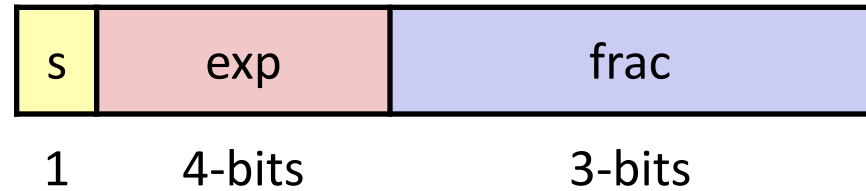- **Case Study**

  - Convert 8-bit unsigned numbers to tiny floating point format

Example Numbers

```
128        10000000
 15        00001101
 33        00010001
 35        00010011
138        10001010
 63        00111111
```

Normalize

s	exp	frac
1	4-bits	3-bits

- **Requirement**
  - Set binary point so that numbers of form 1.xxxxx
  - Adjust all to have leading one
    - Decrement exponent as shift left

Value	Binary	Fraction	Exponent
128	10000000	1.0000000	7
15	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Postnormalize

■ **Issue**

▪ Rounding may have caused overflow

▪ Handle by shifting right once & incrementing exponent

Value	Rounded	Exp	Adjusted	Numeric Result
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

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 $\approx 1.4 \times 10^{-45}$
- Double $\approx 4.9 \times 10^{-324}$

■ **Largest Denormalized**	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$

- Single $\approx 1.18 \times 10^{-38}$
- Double $\approx 2.2 \times 10^{-308}$

■ **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 $\approx 3.4 \times 10^{38}$
- Double $\approx 1.8 \times 10^{308}$