

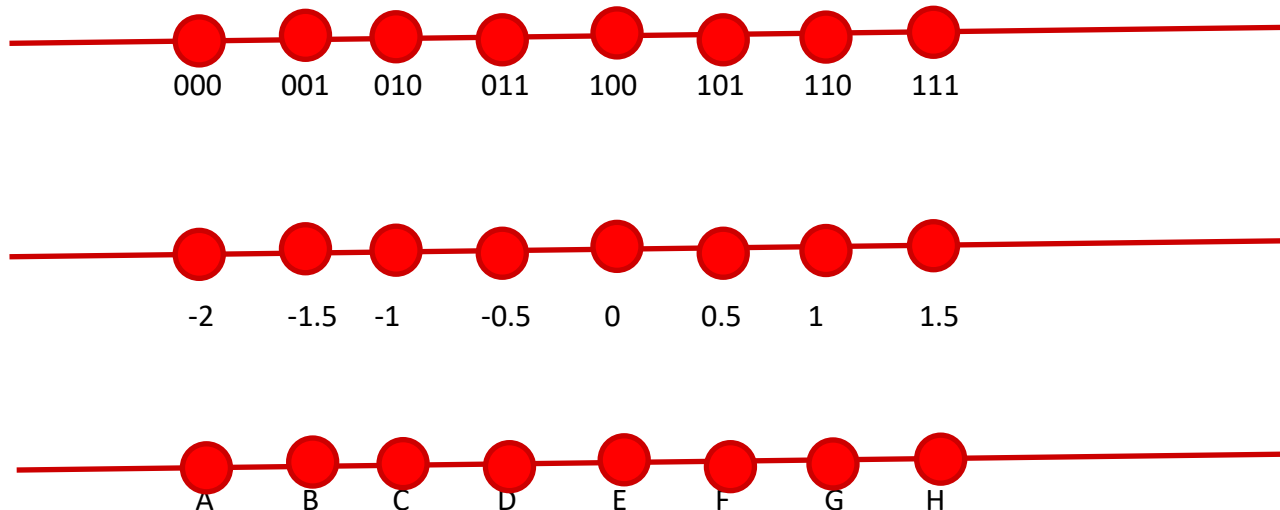


Floating Point

14-513/18-613: Introduction to Computer Systems
3rd Lecture, May 21, 2020

Recall the “Number Line Challenge”

- Given a specific bit-width, we get only a specific number of points on our number line. We can assign them whatever meaning we like, but that’s all we get.
- Consider some examples for a 3-bit number line:

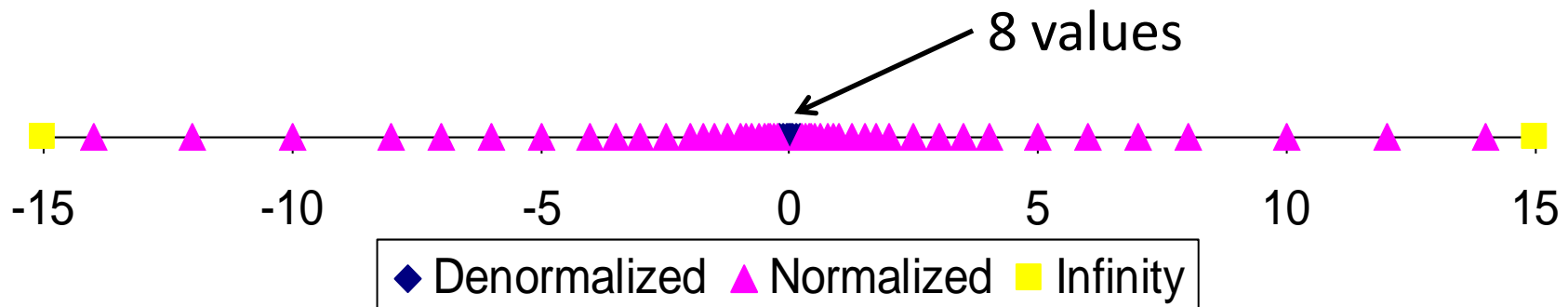


So, Where Do Fractions Fit In?

- **For some applications range is important – things are large and details don't matter**
 - Think telescope
- **For some applications resolution is important and nothing is large**
 - Thing microscope
- **We could solve this problem by using different types, e.g. one for large things and one for small things**
 - But, then what happens when math involved both?
- **We want a unified numbering system where we have more resolution for small things and more range for large things.**

IEEE's Answer

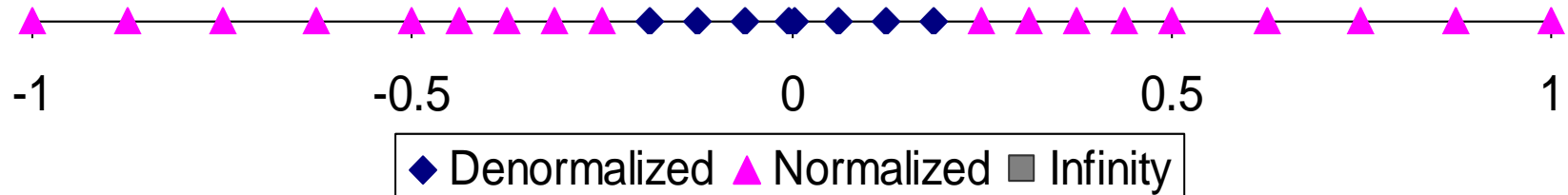
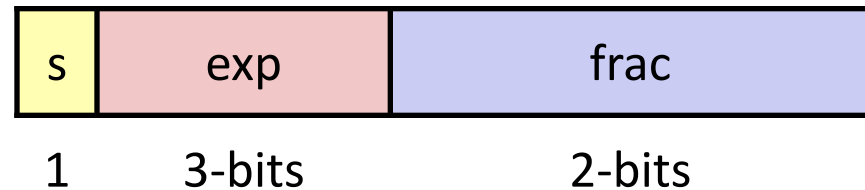
- Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

■ 6-bit IEEE-like format

- $e = 3$ exponent bits
- $f = 2$ fraction bits
- Bias is 3



IEEE Floating Point

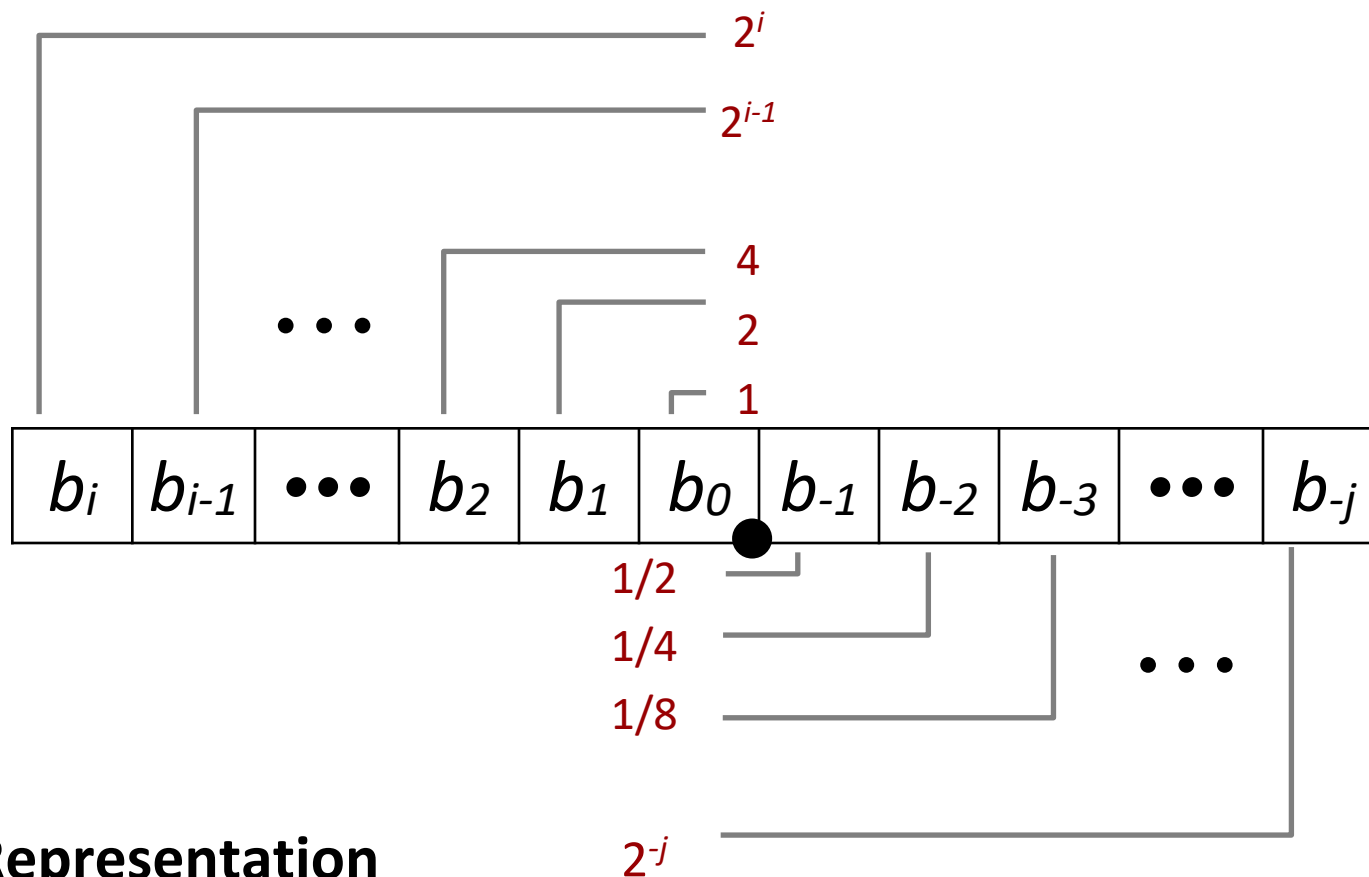
■ IEEE Standard 754

- 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

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	
$5 \frac{3}{4} = \frac{23}{4}$	101.11_2	$= 4 + 1 + \frac{1}{2} + \frac{1}{4}$
$2 \frac{7}{8} = \frac{23}{8}$	10.111_2	$= 2 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}$
$1 \frac{7}{16} = \frac{23}{16}$	1.0111_2	$= 1 + \frac{1}{4} + \frac{1}{8} + \frac{1}{16}$

Observations

- Divide by 2 by shifting right (unsigned)
- Multiply by 2 by shifting left
- Numbers of form $0.111111\dots_2$ are just below 1.0
 - $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{2^i} + \dots \rightarrow 1.0$
 - Use notation $1.0 - \epsilon$

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?)

Floating Point Representation

■ 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

Example:

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

■ 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)



Precision options

- **Single precision: 32 bits**

≈ 7 decimal digits, $10^{\pm 38}$



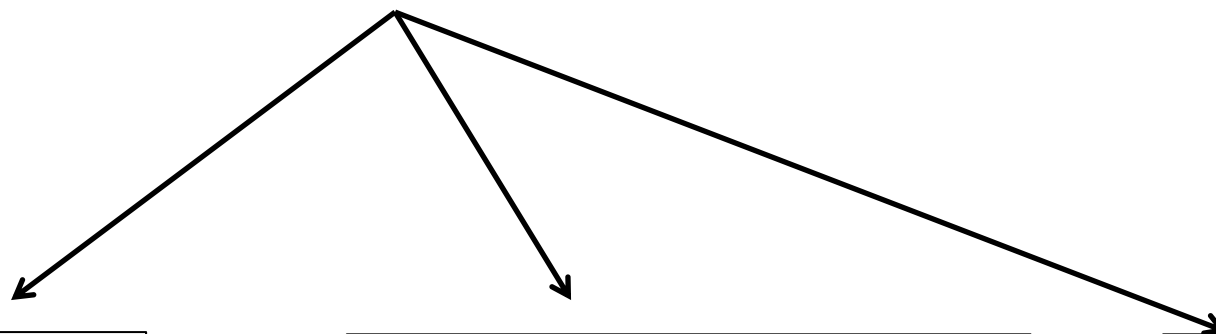
- **Double precision: 64 bits**

≈ 16 decimal digits, $10^{\pm 308}$



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

Three “kinds” of floating point numbers



00...00

denormalized

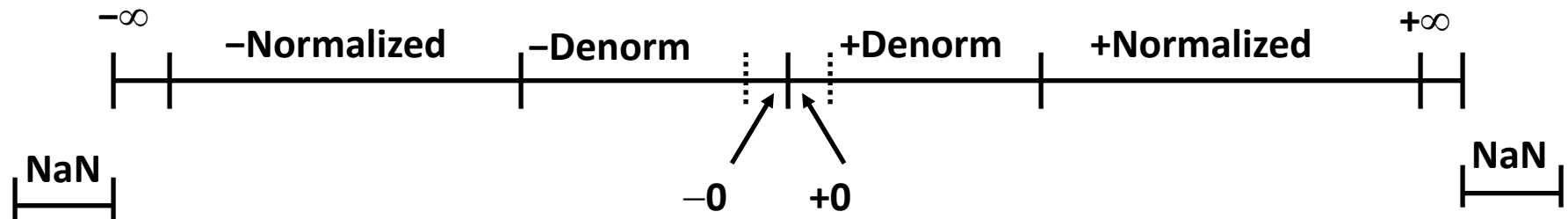
exp ≠ 0 and exp ≠ 11...11

normalized

11...11

special

Visualization: Floating Point Encodings



“Normalized” Values

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

- **When: $\text{exp} \neq 000\dots 0$ and $\text{exp} \neq 111\dots 1$**

- **Exponent coded as a *biased* value: $E = \text{exp} - \text{Bias}$**
 - exp : unsigned value of exp field
 - $\text{Bias} = 2^{k-1} - 1$, 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: $M = 1.\text{xxx}\dots\text{x}_2$**
 - xxx...x: bits of frac field
 - Minimum when **frac**=000...0 ($M = 1.0$)
 - Maximum when **frac**=111...1 ($M = 2.0 - \epsilon$)
 - Get extra leading bit for “free”

Normalized Encoding Example

■ Value: float $F = 15213.0$;

$$\begin{aligned} \blacksquare 15213_{10} &= 11101101101101_2 \\ &= 1.1101101101101_2 \times 2^{13} \end{aligned}$$

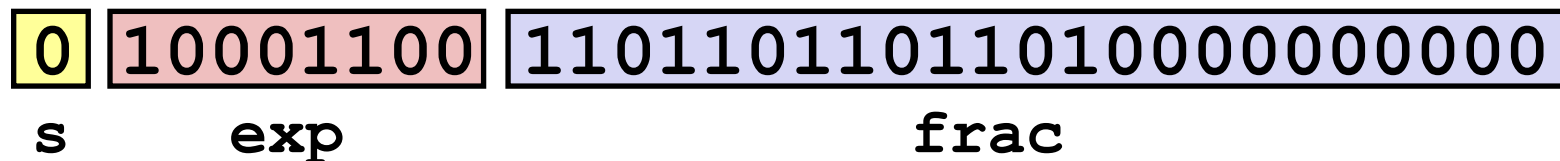
■ Significand

$$\begin{aligned} M &= 1.\underline{1101101101101}_2 \\ \text{frac} &= \underline{11011011011010000000000}_2 \end{aligned}$$

■ Exponent

$$\begin{aligned} E &= 13 \\ \text{Bias} &= 127 \\ \text{exp} &= 140 = 10001100_2 \end{aligned}$$

■ Result:



$$\begin{aligned} v &= (-1)^s M 2^E \\ E &= \text{exp} - \text{Bias} \\ \text{Bias} &= 2^{k-1} - 1 \end{aligned}$$

Denormalized Values

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

$$E = 1 - \text{Bias}$$

$$\text{Bias} = 2^{k-1} - 1$$

- **Condition:** $\text{exp} = 000\dots 0$
- **Bias is the same as for normalized numbers**
 - $\text{Bias} = 2^{k-1} - 1$, where k is number of exponent bits
- **Exponent value: $E = 1 - \text{Bias}$ (instead of $\text{exp} - \text{Bias}$)** (why?)
- **Significand coded with implied leading 0: $M = 0.\text{xxx}\dots\text{x}_2$**
 - $\text{xxx}\dots\text{x}$: bits of frac
- **Cases**
 - $\text{exp} = 000\dots 0$, $\text{frac} = 000\dots 0$
 - Represents zero value
 - Note distinct values: $+0$ and -0 (why?)
 - $\text{exp} = 000\dots 0$, $\text{frac} \neq 000\dots 0$
 - Numbers closest to 0.0
 - Equispaced

Denormalized Encoding Example

- Value: float $F = 0.5;$
- Base 2 Scientific Notation
 - $0.5_{10} = 1.0 \times 2^{-1}$
- Converting to a leading 0
 - $1.0 \times 2^{-1} = 0.1 \times 2^0$

■ Exponent

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

$$E = 1 - Bias = 1 - 1 = 0$$

$$Exp = 00$$

$$Frac = 0.1_2$$

■ Result:

0 00 100

s exp frac

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

$$E = 1 - Bias$$

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

Special Values

- **Condition: $\text{exp} = 111\dots 1$**
- **Case: $\text{exp} = 111\dots 1, \text{frac} = 000\dots 0$**
 - **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: $\text{exp} = 111\dots 1, \text{frac} \neq 000\dots 0$**
 - **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

float: 0xC0A00000

binary: _____



E =

S =

M =

$v = (-1)^S M 2^E =$

$$v = (-1)^S M 2^E$$

$$E = \text{exp} - \text{Bias}$$

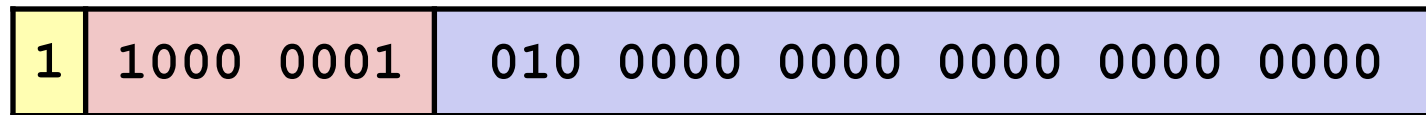
$$\text{Bias} = 2^{k-1} - 1 = 127$$

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 #1

float: 0xC0A00000

binary: 1100 0000 1010 0000 0000 0000 0000 0000



1

8-bits

23-bits

E =

S =

M = 1.

$v = (-1)^S M 2^E =$

$$v = (-1)^S M 2^E$$

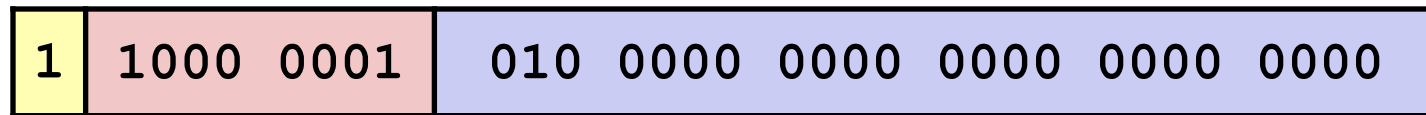
$$E = \text{exp} - \text{Bias}$$

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 #1

float: 0xC0A00000

binary: 1100 0000 1010 0000 0000 0000 0000 0000



1

8-bits

23-bits

$$E = \text{exp} - \text{Bias} = 129 - 127 = 2 \text{ (decimal)}$$

$S = 1$ -> negative number

$$M = 1.010 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000$$

$$= 1 + 1/4 = 1.25$$

$$v = (-1)^S M 2^E = (-1)^1 * 1.25 * 2^2 = -5$$

$$v = (-1)^S M 2^E$$

$$E = \text{exp} - \text{Bias}$$

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

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

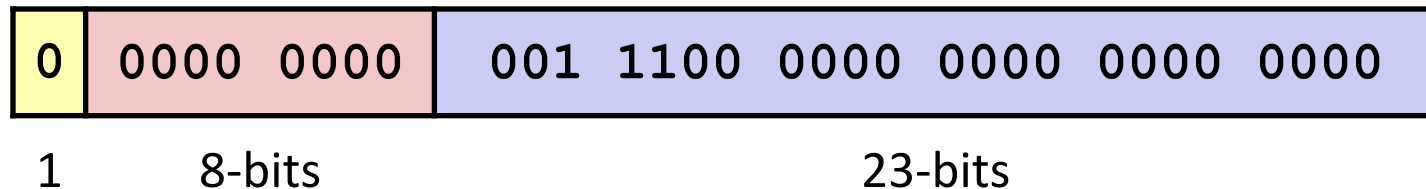
C float Decoding Example #2

$$v = (-1)^S M 2^E$$

$$E = 1 - \text{Bias}$$

float: 0x001C0000

binary: 0000 0000 0001 1100 0000 0000 0000 0000



E =

S =

M = 0 .

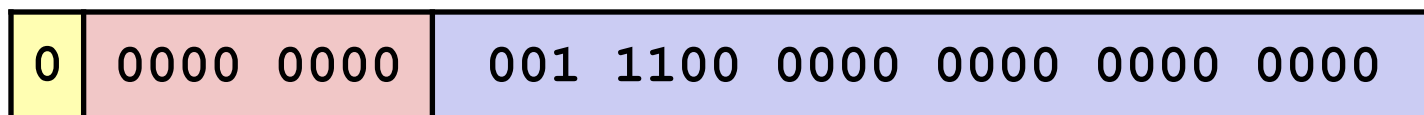
$$v = (-1)^S M 2^E =$$

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 #2

float: 0x001C0000

binary: 0000 0000 0001 1100 0000 0000 0000 0000



1

8-bits

23-bits

$$E = 1 - \text{Bias} = 1 - 127 = -126 \text{ (decimal)}$$

$S = 0$ -> positive number

$$M = 0.001\ 1100\ 0000\ 0000\ 0000\ 0000$$

$$= 1/8 + 1/16 + 1/32 = 7/32 = 7 * 2^{-5}$$

$$v = (-1)^S M 2^E = (-1)^0 * 7 * 2^{-5} * 2^{-126} = 7 * 2^{-131}$$

$$\approx 2.571393892 \times 10^{-39}$$

$$v = (-1)^S M 2^E$$

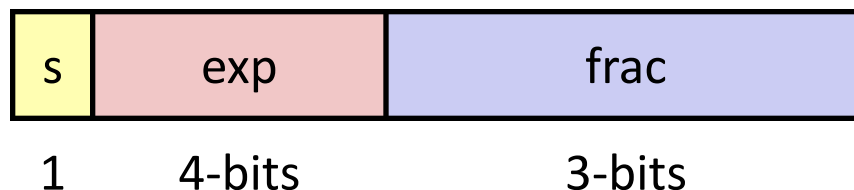
$$E = 1 - \text{Bias}$$

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

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

Tiny Floating Point Example



■ 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	
Denormalized numbers	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	$14/8 * 1/2 = 14/16$	
Normalized numbers	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	

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

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

■ $x +_f y = \text{Round}(x + y)$

■ $x \times_f y = \text{Round}(x \times y)$

■ 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 ($-\infty$)	\$1 ↓	\$1 ↓	\$1 ↓	\$2 ↓	-\$2 ↓
■ Round up ($+\infty$)	\$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 underestimated

■ 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

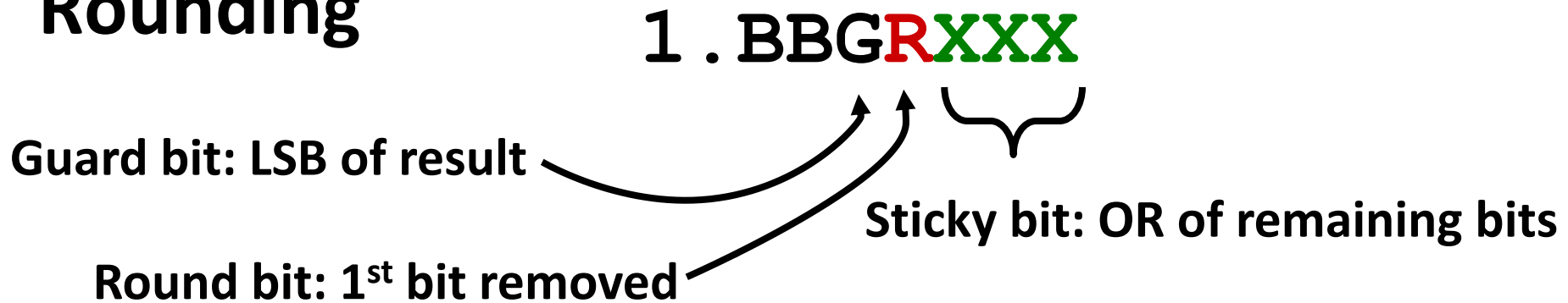
- “Even” when least significant bit is 0
- “Half way” when bits to right of rounding position = 100...₂

■ Examples

- Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00 011 ₂	10.00 ₂	(<1/2—down)	2
2 3/16	10.00 110 ₂	10.01 ₂	(>1/2—up)	2 1/4
2 7/8	10.11 100 ₂	11.00 ₂	(1/2—up)	3
2 5/8	10.10 100 ₂	10.10 ₂	(1/2—down)	2 1/2

Rounding



■ Round up conditions

- Round = 1, Sticky = 1 \rightarrow > 0.5
- Guard = 1, Round = 1, Sticky = 0 \rightarrow Round to even

<i>Fraction</i>	<i>GRS</i>	<i>Incr?</i>	<i>Rounded</i>
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

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

- **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

$$\begin{aligned}
 \text{4 bit significand: } 1.010 * 2^2 \times 1.110 * 2^3 &= 10.0011 * 2^5 \\
 &= 1.00011 * 2^6 = 1.001 * 2^6
 \end{aligned}$$

Floating Point Addition

$$\blacksquare (-1)^{s1} M1 2^{E1} + (-1)^{s2} M2 2^{E2}$$

- Assume $E1 > E2$

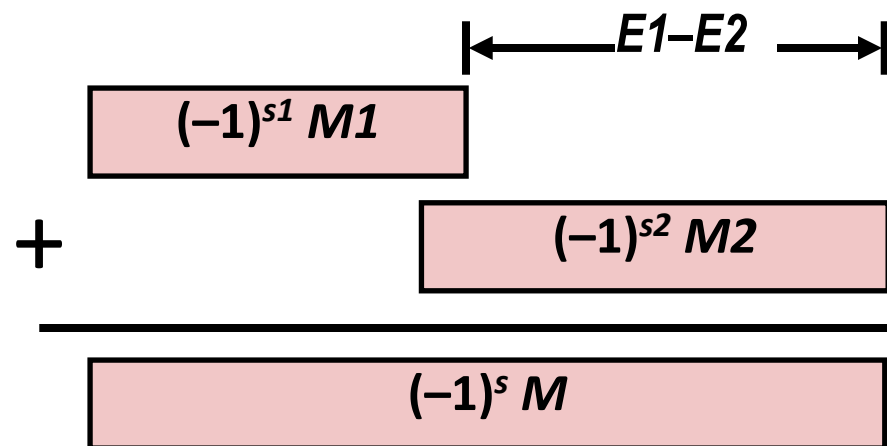
$$\blacksquare \text{Exact Result: } (-1)^s M 2^E$$

- 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



$$1.010 * 2^2 + 1.110 * 2^3 = (0.1010 + 1.1100) * 2^3$$

$$= 1.0110 * 2^3 = 1.00110 * 2^4 = 1.010 * 2^4$$

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

- $a \geq b \Rightarrow a+c \geq b+c$ *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: $(1e20 * 1e20) * 1e-20 = \text{inf}$, $1e20 * (1e20 * 1e-20) = 1e20$
- 1 is multiplicative identity? *Yes*
- Multiplication distributes over addition? *No*
 - Possibility of overflow, inexactness of rounding
 - $1e20 * (1e20 - 1e20) = 0.0$, $1e20 * 1e20 - 1e20 * 1e20 = \text{NaN}$

■ 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`

✗

✓

✓

✗

✓

✗

✓

✓

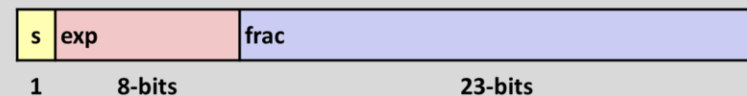
✓

✗

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $M \times 2^E$
- 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

Single precision: 32 bits



Double precision: 64 bits

