

Floating Point

Instructors:

Yuan Tang

Adapted from CMU course 15-213

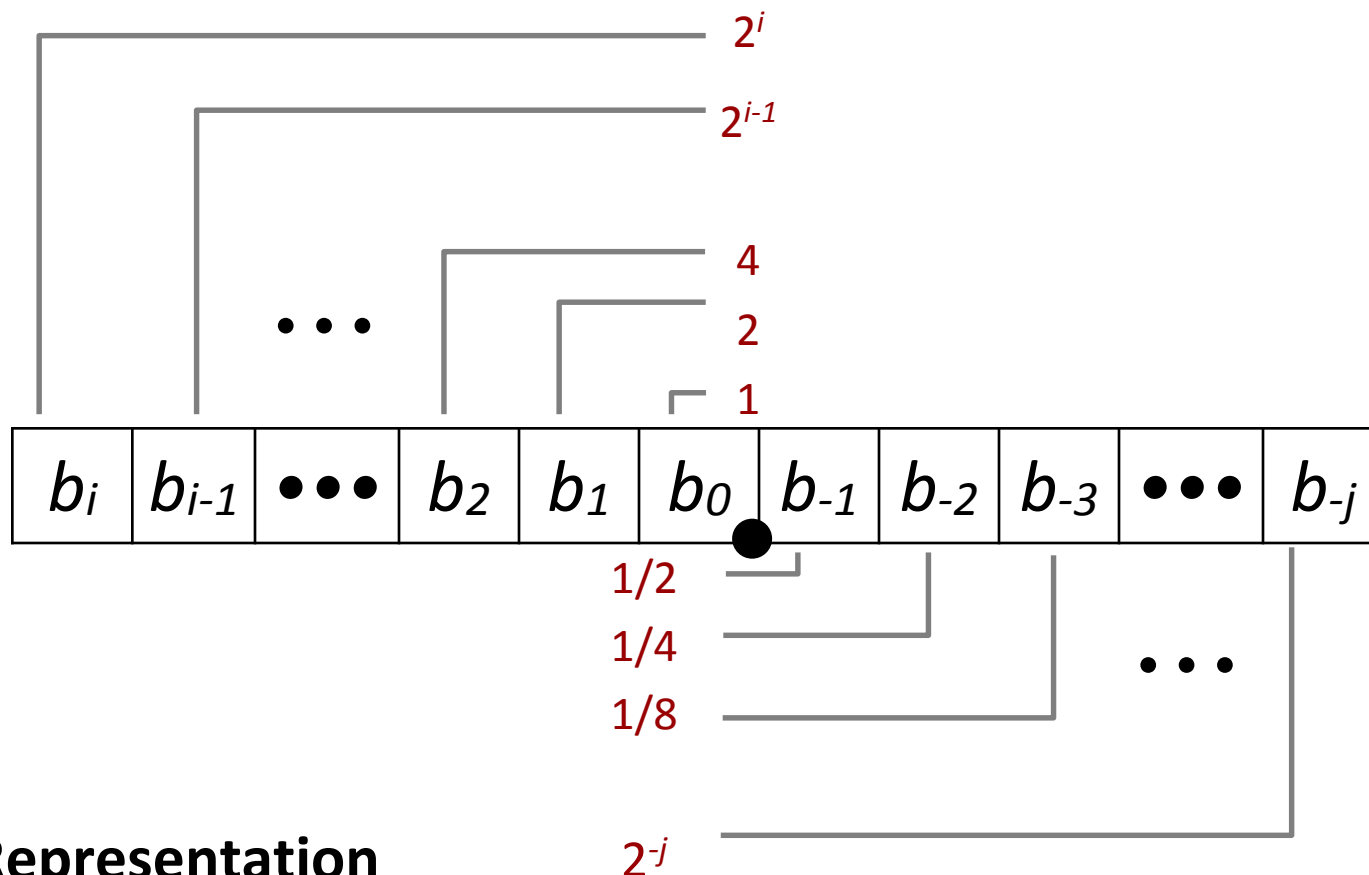
Today: Floating Point

- 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

Value	Representation
$5 \frac{3}{4}$	101.11_2
$2 \frac{7}{8}$	10.111_2
$\frac{63}{64}$	1.0111_2

Observations

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form $0.111111..._2$ 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

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

■ Value

Representation

- | | |
|----------|----------------------------------|
| ■ $1/3$ | $0.0101010101 [01] \dots_2$ |
| ■ $1/5$ | $0.001100110011 [0011] \dots_2$ |
| ■ $1/10$ | $0.0001100110011 [0011] \dots_2$ |

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

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

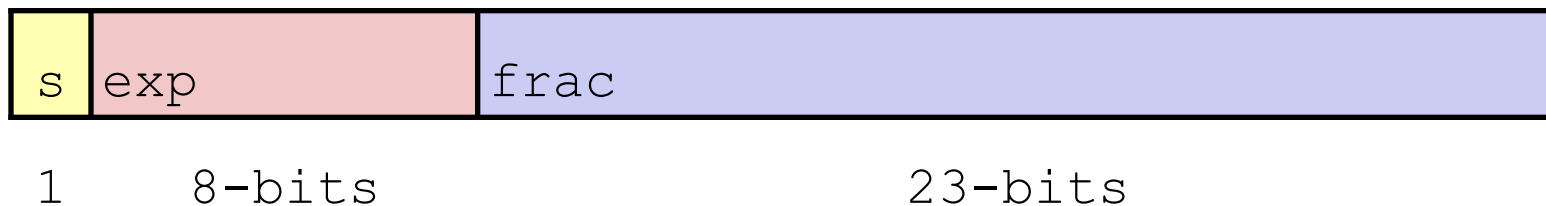
■ Encoding

- 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)



Precisions

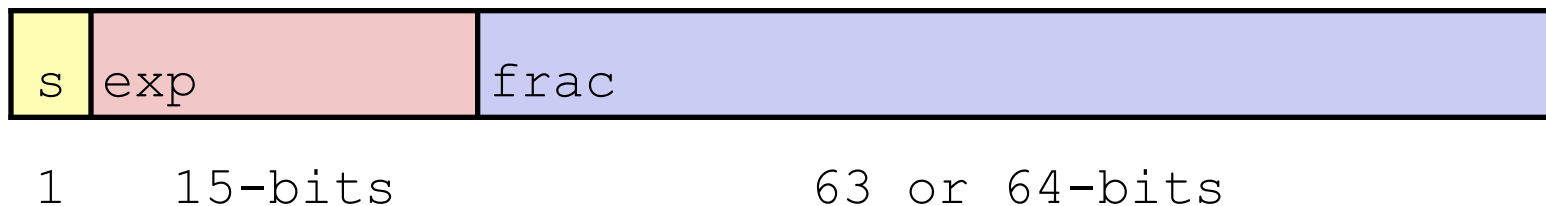
■ Single precision: 32 bits



■ Double precision: 64 bits



■ Extended precision: 80 bits (Intel only)



Normalized Values

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

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

Normalized Encoding Example

■ Value: `Float F = 15213.0;`

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

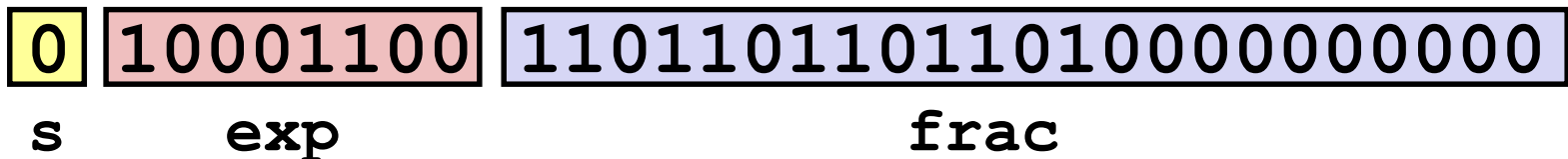
■ Significand

$$\begin{aligned} M &= 1.\underline{1101101101101}_2 \\ \text{frac} &= \underline{1101101101101}0000000000_2 \end{aligned}$$

■ Exponent

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

■ Result:



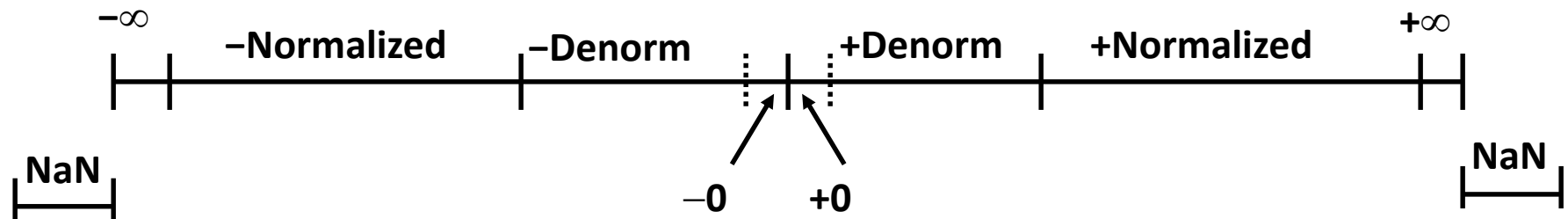
Denormalized Values

- **Condition:** $\text{exp} = 000\dots 0$
- **Exponent value:** $E = -\text{Bias} + 1$ (instead of $E = 0 - \text{Bias}$)
- **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 very close to 0.0
 - Lose precision as get smaller
 - Equispaced

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., $1.0/0.0 = -1.0/-0.0 = +\infty$ ($s=0$), $1.0/-0.0 = -\infty$ ($s=1$)
- **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., $\text{sqrt}(-1)$, $\infty - \infty$, $\infty \times 0$

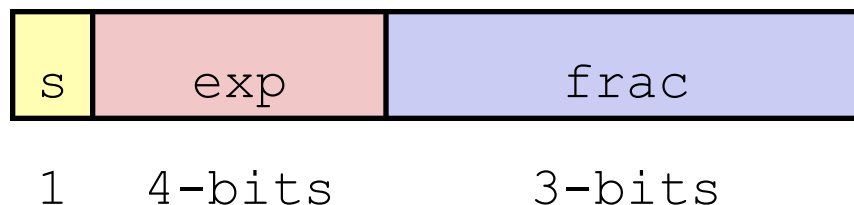
Visualization: Floating Point Encodings



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

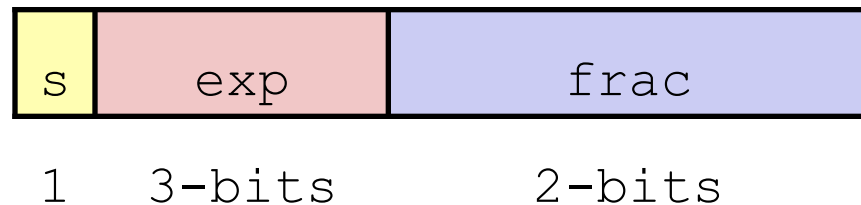
Dynamic Range (Positive Only)

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

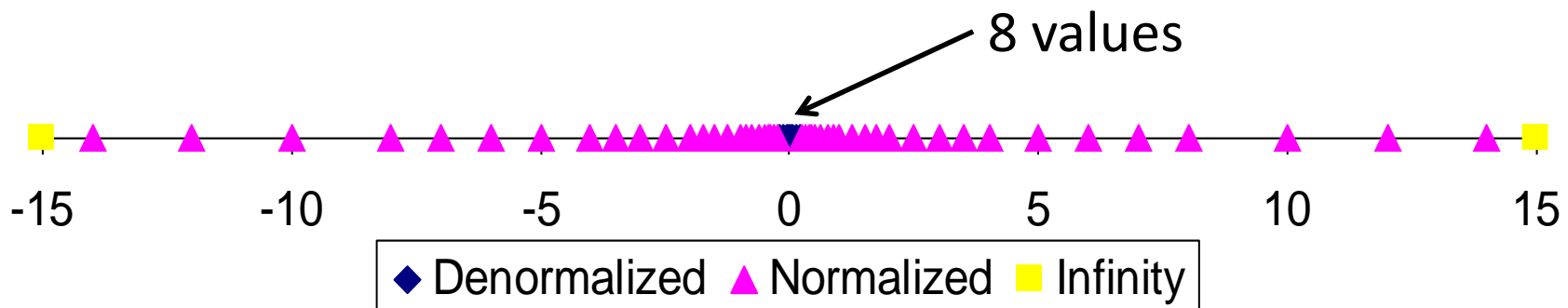
Distribution of Values

■ 6-bit IEEE-like format

- $e = 3$ exponent bits
- $f = 2$ fraction bits
- Bias is $2^{3-1} - 1 = 3$



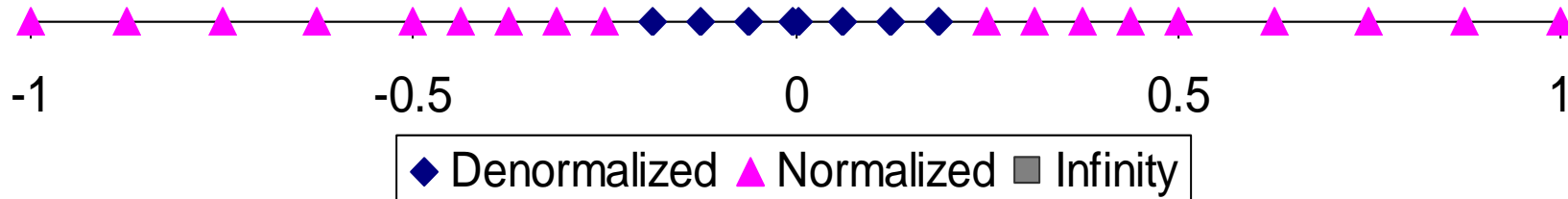
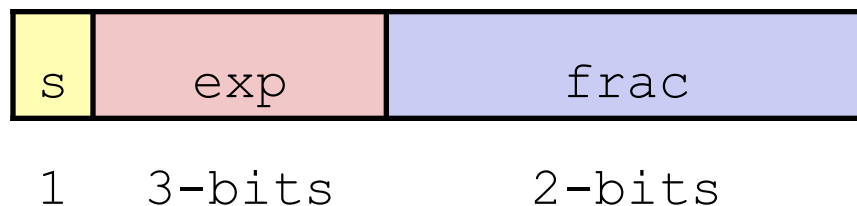
■ Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

■ 6-bit IEEE-like format

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



Interesting Numbers

{single, double}

<i>Description</i>	<i>exp</i>	<i>frac</i>	<i>Numeric Value</i>
■ Zero	00...00	00...00	0.0
■ Smallest Pos. Denorm.	00...00	00...01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
<ul style="list-style-type: none"> ■ Single $\approx 1.4 \times 10^{-45}$ ■ Double $\approx 4.9 \times 10^{-324}$ 			
■ Largest Denormalized	00...00	11...11	$(1.0 - \epsilon) \times 2^{-\{126,1022\}}$
<ul style="list-style-type: none"> ■ Single $\approx 1.18 \times 10^{-38}$ ■ Double $\approx 2.2 \times 10^{-308}$ 			
■ Smallest Pos. Normalized	00...01	00...00	$1.0 \times 2^{-\{126,1022\}}$
<ul style="list-style-type: none"> ■ Just larger than largest denormalized 			
■ One	01...11	00...00	1.0
■ Largest Normalized	11...10	11...11	$(2.0 - \epsilon) \times 2^{\{127,1023\}}$
<ul style="list-style-type: none"> ■ Single $\approx 3.4 \times 10^{38}$ ■ Double $\approx 1.8 \times 10^{308}$ 			

Special Properties of 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?
- 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

$$\blacksquare \mathbf{x} +_{\mathbf{f}} \mathbf{y} = \text{Round}(\mathbf{x} + \mathbf{y})$$

$$\blacksquare \mathbf{x} \times_{\mathbf{f}} \mathbf{y} = \text{Round}(\mathbf{x} \times \mathbf{y})$$

■ 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

■ What are the advantages of the modes?

Closer Look at Round-To-Even

■ Default Rounding Mode

- Hard to get any other kind without dropping into assembly
- 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 = $100..._2$

■ Examples

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

Value	Binary	Rounded	Action	Rounded Value
$2 \frac{3}{32}$	$10.00\textcolor{red}{011}_2$	10.00_2	($<1/2$ —down)	2
$2 \frac{3}{16}$	$10.00\textcolor{red}{110}_2$	10.01_2	($>1/2$ —up)	$2 \frac{1}{4}$
$2 \frac{7}{8}$	$10.11\textcolor{red}{100}_2$	11.00_2	($=1/2$ —up)	3
$2 \frac{5}{8}$	$10.10\textcolor{red}{100}_2$	10.10_2	($=1/2$ —down)	$2 \frac{1}{2}$

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

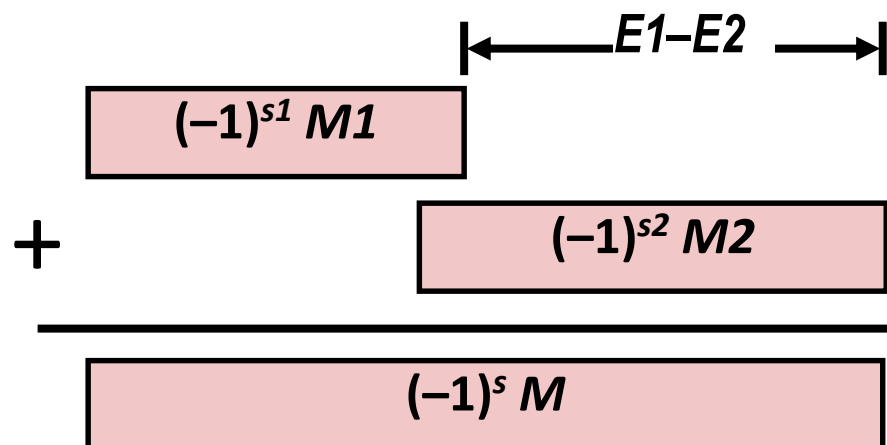
Floating Point Addition

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

- 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

Mathematical Properties of FP Add

■ Compare to those of Abelian Group

- Closed under addition?
 - But may generate infinity or NaN
- Commutative?
- Associative?
 - Overflow and inexactness of rounding
- 0 is additive identity?
- Every element has additive inverse
 - Except for infinities & NaNs

■ Monotonicity

- $a \geq b \Rightarrow a+c \geq b+c$?
 - Except for infinities & NaNs

Mathematical Properties of FP Mult

■ Compare to Commutative Ring

- Closed under multiplication?
 - But may generate infinity or NaN
- Multiplication Commutative?
- Multiplication is Associative?
 - Possibility of overflow, inexactness of rounding
- 1 is multiplicative identity?
- Multiplication distributes over addition?
 - Possibility of overflow, inexactness of rounding

■ Monotonicity

- $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c$?
 - 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 == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0` \Rightarrow `((d*2) < 0.0)`
- `d > f` \Rightarrow `-f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Today: Floating Point

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

Summary

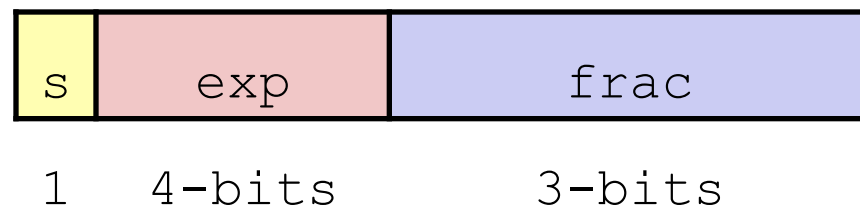
- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $M \times 2^E$
- One can reason about operations independent of implementation
 - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
 - Violates associativity/distributivity
 - Makes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

More Slides

Creating Floating Point Number

■ Steps

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



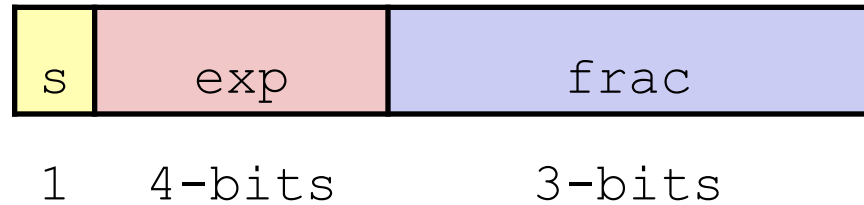
■ Case Study

- 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

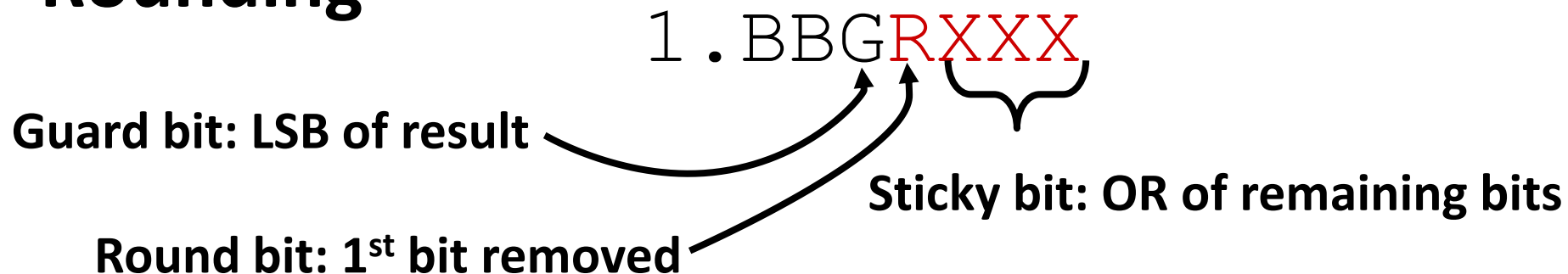


■ Requirement

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

<i>Value</i>	<i>Binary</i>	<i>Fraction</i>	<i>Exponent</i>
128	10000000	1.00000000	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

Rounding



■ Round up conditions

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

<i>Value</i>	<i>Fraction</i>	<i>GRS</i>	<i>Incr?</i>	<i>Rounded</i>
128	1.000 0000	000	N	1.000
15	1.101 0000	100	N	1.101
17	1.000 1000	010	N	1.000
19	1.001 1000	110	Y	1.010
138	1.000 1010	011	Y	1.001
63	1.111 1100	111	Y	10.000

Postnormalize

■ Issue

- Rounding may have caused overflow
- Handle by shifting right once & incrementing exponent

<i>Value</i>	<i>Rounded</i>	<i>Exp</i>	<i>Adjusted</i>	<i>Result</i>
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