

Floating Point

15-213: Introduction to Computer Systems
4th Lecture, Jan 26, 2012

Instructors:

Todd C. Mowry & Anthony Rowe

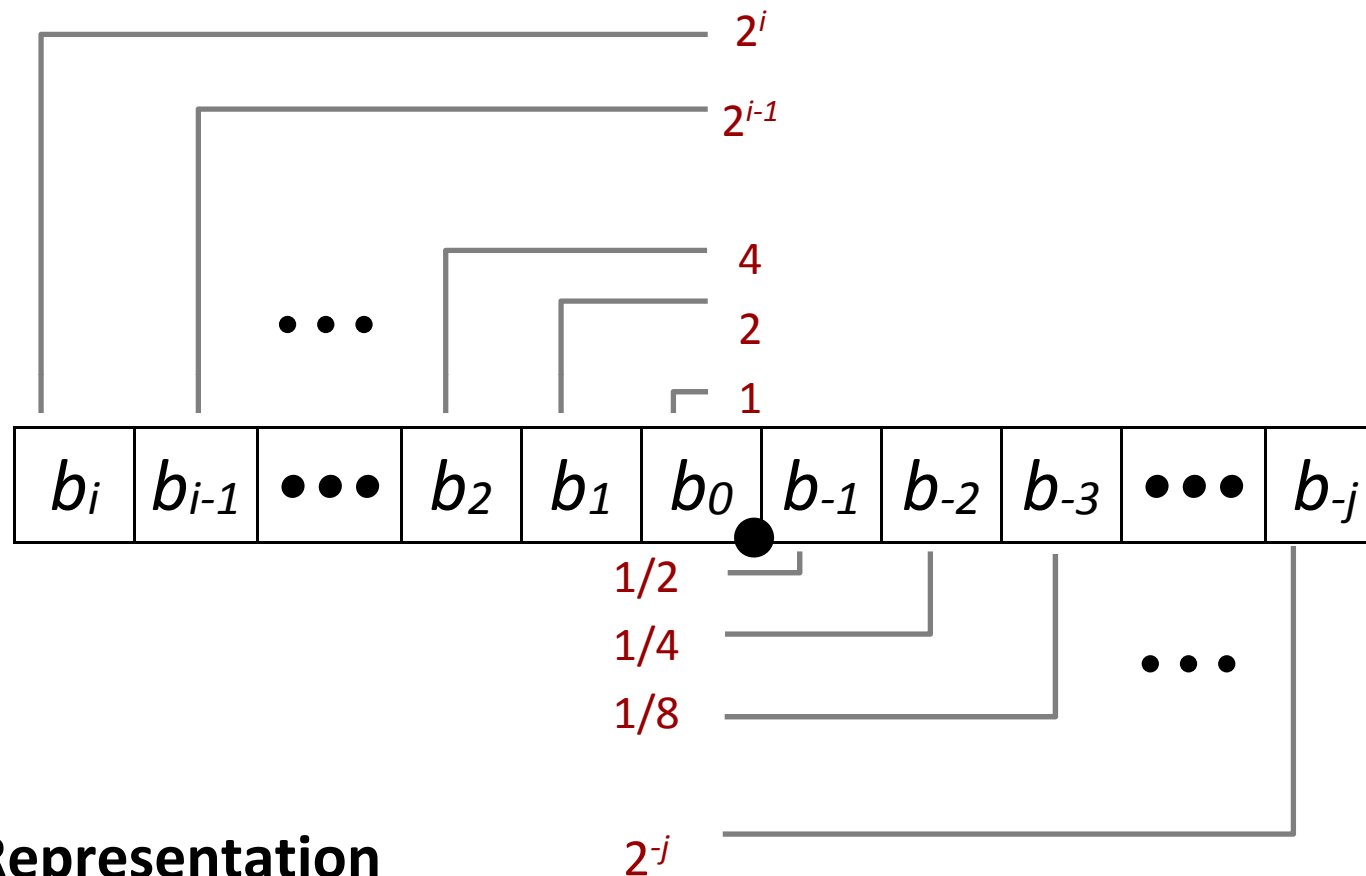
Today: Floating Point

- **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

$5 \frac{3}{4}$	101.11_2
$2 \frac{7}{8}$	10.111_2
$\frac{63}{64}$	0.111111_2

■ 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 decimal 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

■ 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)



Precision options

■ Single precision: 32 bits



■ Double precision: 64 bits



■ Extended precision: 80 bits (Intel only)



“Normalized” Values

- **When: $\text{exp} \neq 000\dots 0$ 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: $M = 1.\text{XXX}\dots\text{X}_2$**
 - xxx...x: bits of frac
 - Minimum when $\text{frac}=000\dots 0$ ($M = 1.0$)
 - Maximum when $\text{frac}=111\dots 1$ ($M = 2.0 - \epsilon$)
 - Get extra leading bit for “free”

Normalized Encoding Example

- Value: Float $F = 15213.0$;

- $15213_{10} = 11101101101101_2$
 $= 1.1101101101101_2 \times 2^{13}$

- Significand

$$M = 1.\underline{1101101101101}_2$$

$$\text{frac} = \underline{11011011011010000000000}_2$$

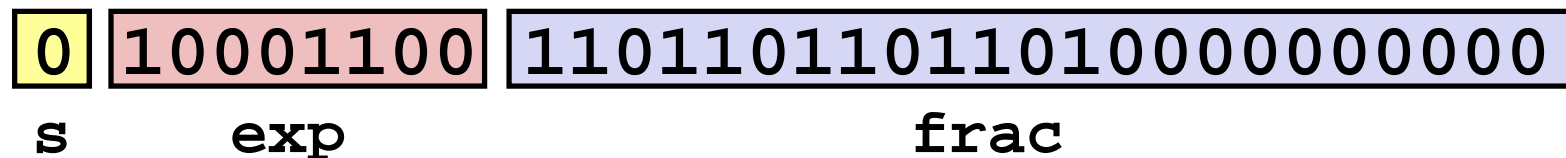
- Exponent

$$E = 13$$

$$\text{Bias} = 127$$

$$\text{Exp} = 140 = 10001100_2$$

- Result:



Denormalized Values

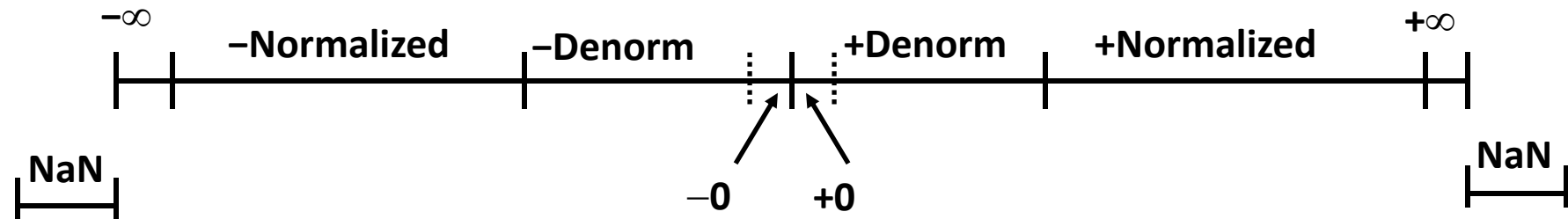
- **Condition:** $\text{exp} = 000\dots 0$

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

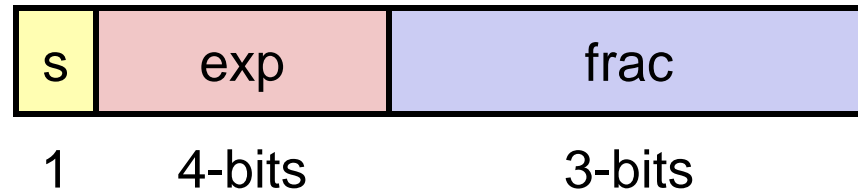
Visualization: Floating Point Encodings



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

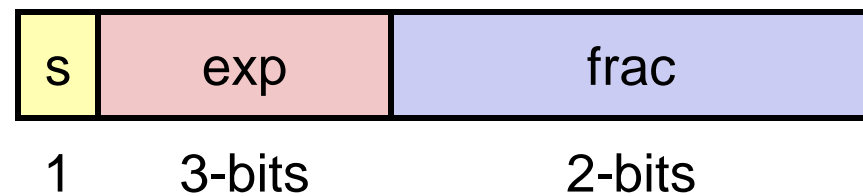
Dynamic Range (Positive Only)

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	closest to zero
	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
	0	0001	001	-6	$9/8 * 1/64 = 9/512$	
	...					
	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		

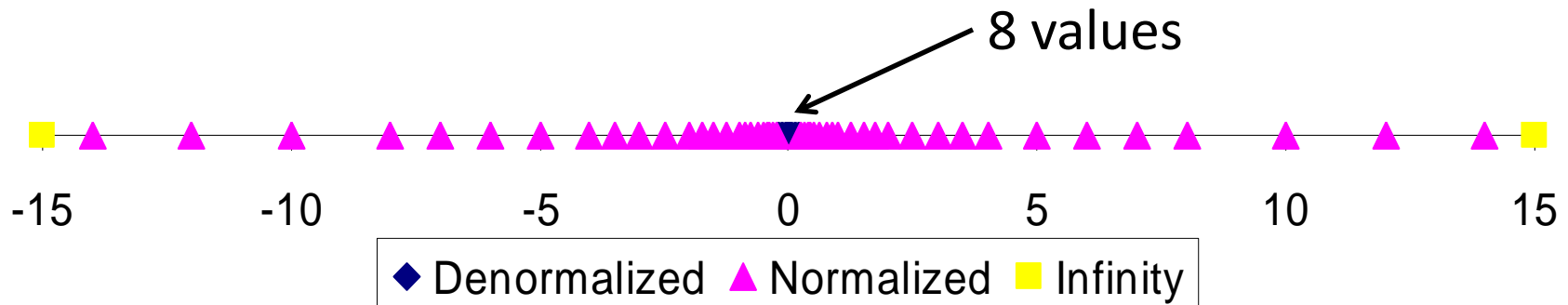
Distribution of Values

■ 6-bit IEEE-like format

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



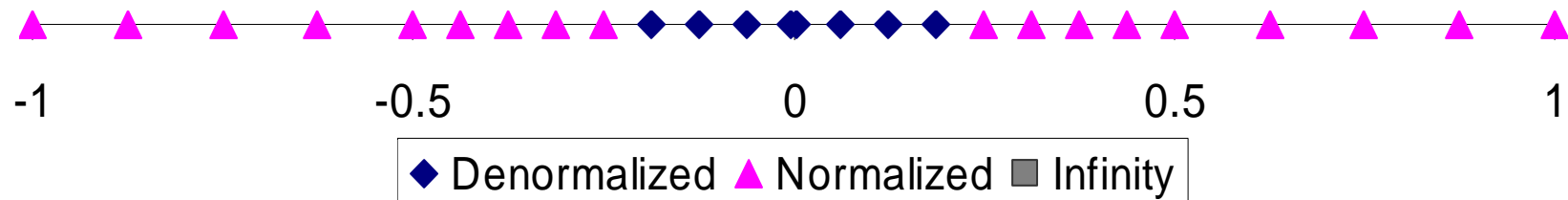
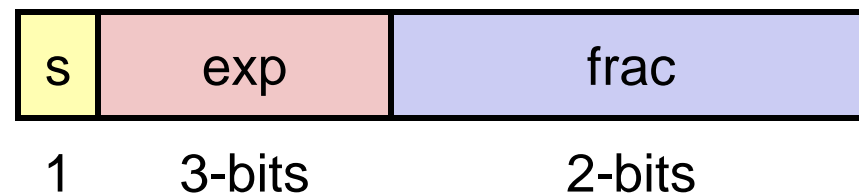
■ Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

■ 6-bit IEEE-like format

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



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

■ $x +_f y = \text{Round}(x + y)$

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

■ 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

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 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

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 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

FP Multiplication

■ $(-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2}$

■ **Exact Result:** $(-1)^s M 2^E$

- Sign s : $s_1 \wedge s_2$
- Significand M : $M_1 \times M_2$
- Exponent E : $E_1 + E_2$

■ 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

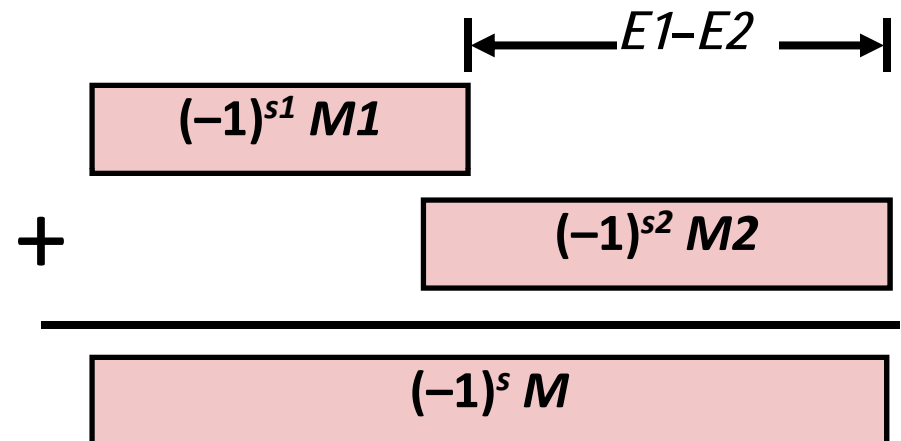
Floating Point Addition

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

- 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

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

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

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

- $x == (\text{int})(\text{float}) x$
- $x == (\text{int})(\text{double}) x$
- $f == (\text{float})(\text{double}) f$
- $d == (\text{float}) d$
- $f == -(-f);$
- $2/3 == 2/3.0$
- $d < 0.0 \quad \Rightarrow \quad ((d*2) < 0.0)$
- $d > f \quad \Rightarrow \quad -f > -d$
- $d * d \geq 0.0$
- $(d+f)-d == f$