

Lecture 04


Floating Point

CS213 – Intro to Computer Systems
Branden Ghen a – Winter 2023

Slides adapted from:

St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Administrivia

- Homework 1 due today! (11:59 pm Central)
 - Submit on Gradescope
 - About 2/3s of the class has submitted so far 
- Pack Lab is released
 - Partnerships have been formed from survey too
 - You can still fill it out until end-of-day tomorrow, but no promises
- Monday: MLK Day – No lecture

Today's Goals

- Explore representing real (decimal) numbers with binary
- Understand IEEE754 encoding
- Discuss encoding impacts on floating-point arithmetic

What is hard about floating point?

- LOTS OF RULES
 - No, more than that
- Homework 2 will give you a chance to practice
- Plus on exams you'll have a notes sheet to write down rules on

Outline

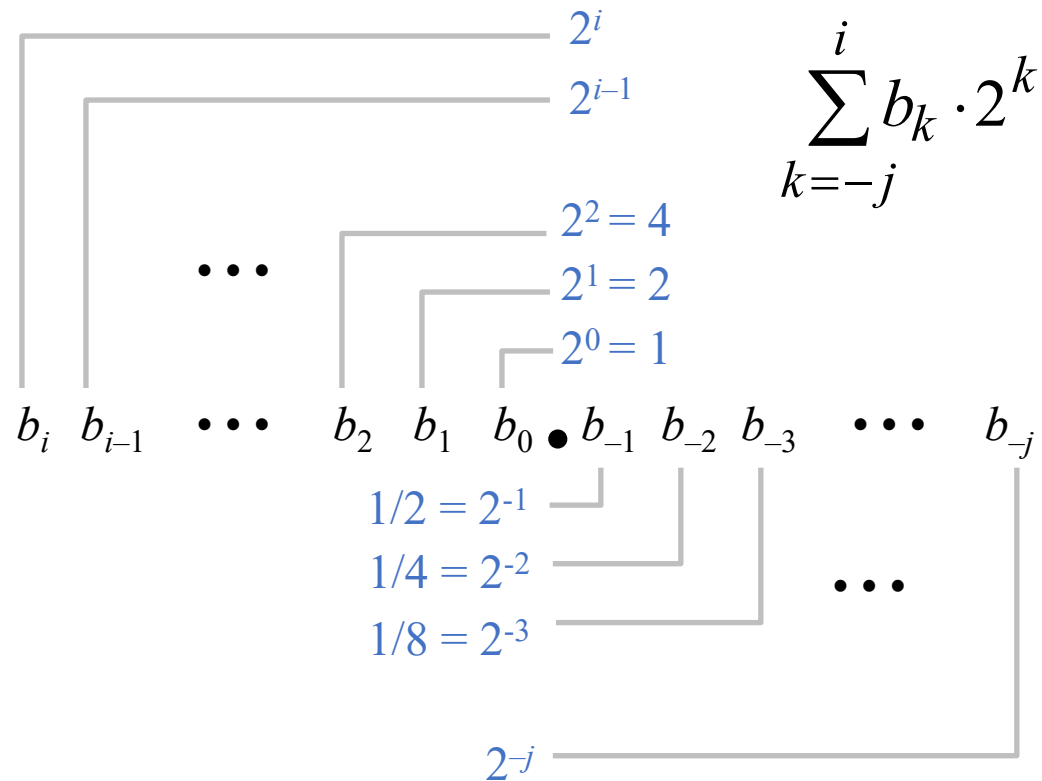
- **Fractional Binary Numbers**
- Representing Floating Point
- Smaller Floating Point
- Floating Point Arithmetic

Floating point numbers

- In decimal:
 - 123450_{10}
 - 123.450_{10}
 - 1.23450_{10}
- We can use this same system in binary as well:
 - 1010110_2 (86_{10})
 - 1010.110_2 ($10.75_{10} = \frac{86}{2^3}$)
 - 1.010110_2 ($1.34375_{10} = \frac{86}{2^6}$)

Fractional Binary Numbers

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



Example binary conversion

1010.110

Before the binary point:

$$1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 1*2^3 + 1*2^1 = 8+2 = 10$$

After the binary point:

$$1*2^{-1} + 1*2^{-2} + 0*2^{-3} = 1*2^{-1} + 1*2^{-2} = \frac{1}{2} + \frac{1}{4} = \frac{3}{4} = 0.75$$

Fractional Binary Number Examples

- $5 + 3/4 = 0b101.\boxed{11}$ Note:
This is the number 3!
- $2 + 7/8 = 0b10.\boxed{111}$ This is the number 7!
- $63/64 = 0b0.\boxed{1111111}$ This is the number 63!

Binary point is part of the solution, but not an entire encoding

- Some problems remain:

1. Computers are finite, but real numbers are not

- Need to choose how many bits to use
- Many decimal numbers would take infinite binary bits to represent perfectly
 - $3.14_{10} = 11.0010001111010111_2$ (we could keep going)

2. We also need to represent where the “binary point” is located

- We’ll use some of our bits to do so

3. Should do signed numbers while we’re at it

Outline

- Fractional Binary Numbers
- **Representing Floating Point**
- Smaller Floating Point
- Floating Point Arithmetic

Floating Point Standard – IEEE754

- Floating point representations
 - Encodes rational numbers of the form $V = m \times 2^e$
 - Base 2 scientific notation!
- IEEE Standard 754 (IEEE floating point)
 - Established in 1985 as uniform standard for floating point arithmetic
 - Before that, many idiosyncratic formats
 - Headed by William Kahan, CS prof. at UC Berkeley (later won Turing Award)
 - Supported by all major CPUs
- Driven by numerical concerns and numerical analysts
 - Nice standards for rounding, overflow, underflow
 - Had to be implementable in fast hardware as well and support many languages

Floating Point Representation

Numerical form:

$$V = (-1)^S * M * 2^E$$

Sign bit Significand (Mantissa) Exponent

- Sign bit **S** determines whether number is negative or positive
- Significand **M** normally a fractional value in range [1.0,2.0) or [0.0,1.0)
 - Called ***mantissa*** or ***significand***
- Exponent **E** weights value by power of two

Floating Point Encoding

Numerical form:

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

Diagram illustrating the numerical form of a floating point number: $V = (-1)^s * M * 2^E$. The components are labeled with blue arrows: s is the **Sign bit**, M is the **Significand (Mantissa)**, and E is the **Exponent**.

- Encoding

- MSb is sign bit (can still look at most-significant bit alone to determine sign!)
- **exp** field encodes E , k -bits (note: "*encodes E*" != "*is E*")
- **frac** field encodes M , n -bits



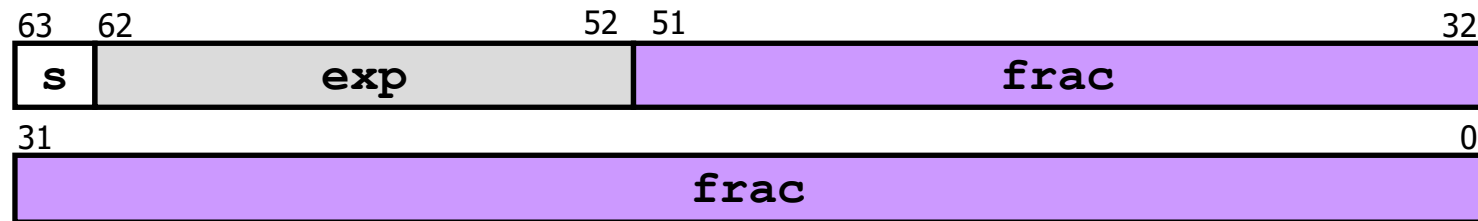
Floating Point Precision

- Sizes

- Single precision: $k = 8$ exp bits, $n = 23$ frac bits (32b total). `float` in C



- Double precision: $k = 11$ exp bits, $n = 52$ frac bits (64b total). `double` in C



Categories for Encoded Values

- Value encoded – three cases, depending on value of **exp**
 1. Normalized, the most common



2. Denormalized (very small values)



3. Special values – infinity and NaN



Categories for Encoded Values

- Value encoded – three cases, depending on value of **exp**

1. Normalized, the most common



2. Denormalized



3. Special values – infinity and NaN



Normalized, Significand

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

- Condition: not a special exponent (all zeros or ones)
- Significand coded with implied leading 1
 - $M = 1.x_1x_2\dots x_n$ ($1+f$ where $f = 0.x_1x_2\dots x_n$)
 - $x_1x_2\dots x_n$: bits of **frac**
- Idea: every normalized number is $1.xxxx$
 - So we're not going to include the leading 1 in the frac
 - We'll just know it's there when we convert to decimal
 - Saves one extra bit in the encoding!



Normalized, Exponent

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

- Condition: not a special exponent (all zeros or ones)
- Exponent coded as biased value
 - $E = \text{Exp} - \text{Bias}$
 - Exp : unsigned value denoted by **exp**
 - Bias : Bias value = $2^{k-1} - 1$, k is number of exponent bits
 - Single precision (8-bit exp): 127 (Exp: 1...254, E: -126...127)
 - Double precision (11-bit exp): 1023 (Exp: 1...2046, E: -1022...1023)
- Exponent really just pushes the binary point around
 - $1.11 * 2^2 = 11.1 * 2^1 = 111.0 * 2^0 = 111$
 - $111 * 2^{-2} = 11.1 * 2^{-1} = 1.11 * 2^0 = 1.11$



Decoding example for normalized floating point (32-bit)

- $0x41900000 = 0b01000001100100000000000000000000$
 - Group bits **s**: 0 **exp**: 10000011 **frac**: 001000000000000000000000
 - **exp** is not all zeros or all ones => not a special case
- $M = 1.001000000000000000000000 = 1.001$
- $E = \mathbf{exp} - \mathbf{bias} = 131 - 127 = 4$
 - $\mathbf{bias} = 2^{k-1} - 1, k=8 \rightarrow 2^7 - 1 = 127$
- $\mathbf{Result} = (-1)^0 * 1.001_2 * 2^4 = 10.01_2 * 2^3 = 10010.2 = 18$

$V = (-1)^s * M * 2^E$	s	exp	frac
------------------------	----------	------------	-------------

Normalized Encoding Example

- **Value**

- `float F = 15213.0; // single precision: 8 exp bits, 23 frac bits`
- $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$

- **Significand**

- $M = 1.\underline{1101101101101}_2$
- $\text{frac} = \underline{1101101101101}0000000000$ pad with 0s *on the right*. (example: 1.5 = 1.500)

- **Exponent**

- $E = 13$
- Bias = 127
- $\text{exp} = E + \text{Bias} = 140 = 10001100_2$

More examples and practice in the bonus slides after the end

Floating Point Representation:

Hex:	4	6	6	D	B	4	0	0
Binary:	0100	0110	0110	1101	1011	0100	0000	0000
exp:	100	0110	0					
frac:				110	1101	1011	0100	0000 0000

Normalized Numbers: Why These Choices?

- Significand coded with **implied leading 1**
 - Any non-zero integer will start with a 1 bit somewhere
 - Leading 1 carries no information, so don't need to store it!
 - Can express mantissas between:
 - 1.0 when frac is all 0s
 - 2.0 (nearly) when frac is all 1s
 - Want smaller? Use a smaller exponent!
- Exponent coded as biased value
 - $E = Exp - Bias$
 - Alternative to using two's complement to represent signed integers
 - Reasons are a bit tricky
 - Floating point binary values increase in the same order as unsigned = share comparisons!
 - Bias provides a more useful range (when considering denormalized)

Question + Break

- $0x3F800000 = 0b00111111000000000000000000000000$
 - Group bits **s**: 0 **exp**: 01111111 **frac**: 0000000000000000000000000000
 - **exp** is not 0...0 or 1...1 => not a special case
- **M** =
- **E** = **exp** - **bias** =
 - **bias** = $2^{k-1} - 1$, $k=8 \rightarrow 2^7 - 1 = 127$

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



Question + Break

- $0x3F800000 = 0b00111111100000000000000000000000$
 - Group bits **s**: 0 **exp**: 01111111 **frac**: 0000000000000000000000000000
 - **exp** is not 0...0 or 1...1 => not a special case
- $M = 1.00000000000000000000000000000000 = 1.0$
- $E = \mathbf{exp} - \mathbf{bias} = 127 - 127 = 0$
 - $\mathbf{bias} = 2^{k-1} - 1, k=8 \rightarrow 2^7 - 1 = 127$
- $\mathbf{Result} = (-1)^0 * 1.0_2 * 2^0 = 1$



Live Practice – pick a hex number

- 0xXXXX0000 =

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



Categories for Encoded Values

- Value encoded – three cases, depending on value of **exp**
 1. Normalized, the most common



2. Denormalized

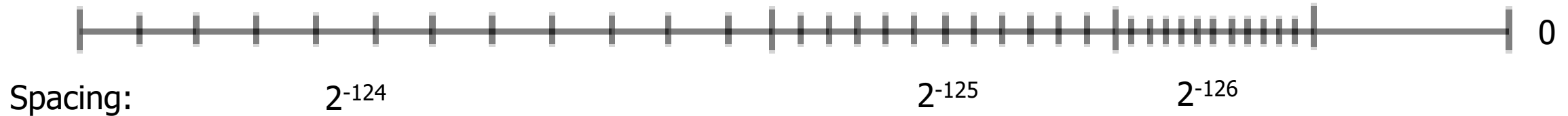


3. Special values – infinity and NaN



Normalized floating point leaves a gap around zero

- Gap is the size of $1.0000 * 2^{\text{Min Exponent}}$ (due to leading 1 bit)
 - And how do we encode "zero" anyways?




- Solution: fill in numbers between 0 and $1 * 2^{\text{Min Exponent}}$
 - Using same spacing as the previous range, in the form **0**.XXXXX



Denormalized Values

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

- Purpose: gracefully represent numbers approaching ± 0
- Condition: $\text{exp} = 000\dots 0_2$ 
- Value
 - Exponent value $E = \mathbf{1 - Bias}$
 - Note: not simply $E = 0 - \text{Bias}$ as it would be if we followed the previous rules
 - This means we're re-using the spacing from smallest normalized numbers
 - Significand value $M = \mathbf{0.xxx\dots x}_2$ (*0.frac*)
 - xxx...x: bits of frac. Leading 0 instead of leading 1
- Cases
 - $\text{exp} = 000\dots 0, \text{frac} = 000\dots 0 \Rightarrow$ Represents value 0
 - Note that we have distinct values +0 and -0
 - $\text{exp} = 000\dots 0, \text{frac} \neq 000\dots 0 \Rightarrow$ Numbers very close to 0.0

Categories for Encoded Values

- Value encoded – three cases, depending on value of **exp**
 1. Normalized, the most common



2. Denormalized



3. Special values – infinity and NaN



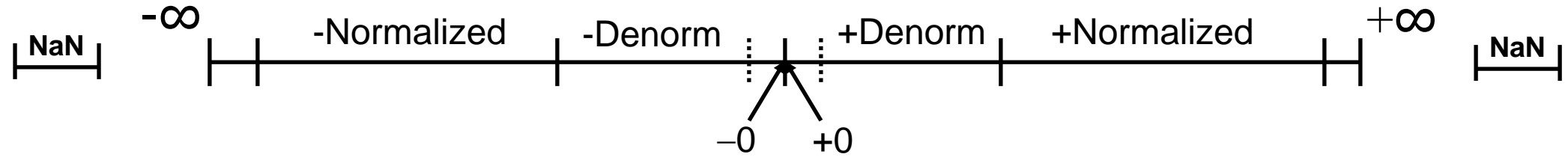
Special Values

- Purpose: represent quantities that $(-1)^s * M * 2^E$ cannot
- Condition: $\text{exp} = 111\dots 1_2$
- Cases
 - $\text{exp} = 111\dots 1_2, \text{frac} = 000\dots 0_2$
 - Represents value ∞ (infinity)
 - Both positive and negative infinity (sign bit to tell apart)
 - Operation that overflows: nicer mathematical behavior than modulo!
 - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty, -1.0/0.0 = -\infty$
 - $\text{exp} = 111\dots 1_2, \text{frac} \neq 000\dots 0_2$
 - Not-a-Number (NaN)
 - Represents case when no numeric value can be determined
 - Fraction could be used to distinguish sources (rarely used in practice)
 - E.g., $\sqrt{-1}, \infty - \infty, \infty * 0$

Floating Point in C

- C guarantees two levels
 - `float` single precision
 - `double` double precision
- Conversions
 - `int` → `float`
 - maybe rounded
 - less bits for actual value (32 → 23)
 - `int` or `float` → `double`
 - exact value preserved
 - double has greater range and higher precision (52 bits for `frac`)
 - `double` → `float`
 - may overflow, underflow (too small to represent), or be rounded (IEEE 754)
 - C99 standard says **undefined** if value out of range
 - `double` or `float` → `int`
 - rounded toward zero (-1.999 → -1)
 - C99 standard says **undefined** if value out of range

Break + Summary of FP Real Number Encodings



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

	Normalized	Denormalized
s	0/1 means +/-	0/1 means +/-
exp	exp \neq 000...0 ₂ and exp \neq 111...1 ₂	exp = 000...0 ₂
frac	x ₁ x ₂ x ₃ ...x _j	x ₁ x ₂ x ₃ ...x _j
Bias=	2 ^(k-1) - 1, for k exponent bits	2 ^(k-1) - 1, for k exponent bits
E=	exp - Bias	1 - Bias
M=	1. x ₁ x ₂ x ₃ ...x _j a.k.a. 1.frac	0. x ₁ x ₂ x ₃ ...x _j a.k.a. 0.frac
V=	(-1) ^s × (1.frac) × 2 ^(exp - Bias)	(-1) ^s × (0.frac) × 2 ^(1 - Bias)

Outline

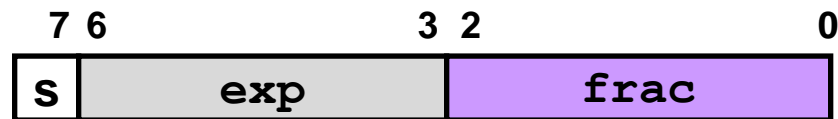
- Fractional Binary Numbers
- Representing Floating Point
- **Smaller Floating Point**
- Floating Point Arithmetic

Floating point examples

- We'll often do floating point in custom bit widths
 - Rather than 32-bit (float) or 64-bit (double)
- Reasons
 1. 64 is just too many bits to write out and think about
 2. Make sure you understand the concepts of floating point
 - Smaller versions still demonstrate concepts! (e.g., 8-bit)

Example: Tiny Floating Point

- 8-bit Floating Point Representation
 - Sign bit is in the most significant bit.
 - Next four (k) bits are exp, with a bias of 7 ($2^{k-1}-1$)
 - Last three (n) bits are frac
- Same general form as IEEE 754 format
 - normalized, denormalized numbers
 - representation of 0, NaN, infinity



Sidebar: increasingly useful for Machine Learning use!

- Models often don't need 32-bits of precision

Exponents for 8-bit tiny floats

$$\text{Bias} = 2^{4-1} - 1 = 7$$

(4-bit exp)

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

	exp	exp	E	2^E	
→	0	0000	-6	1/64	(denorms)
	1	0001	-6	1/64	
	2	0010	-5	1/32	
	3	0011	-4	1/16	
	4	0100	-3	1/8	
	5	0101	-2	1/4	
	6	0110	-1	1/2	
Normalized $E = \text{exp} - \text{Bias}$	→	7	0111	0	1
		8	1000	+1	2
		9	1001	+2	4
		10	1010	+3	8
		11	1011	+4	16
		12	1100	+5	32
		13	1101	+6	64
		14	1110	+7	128
Special	→	15	1111	n/a	(inf, NaN)

Dynamic Range of 8-bit tiny float

```
0 0000 000
0 0000 001
0 0000 010
...
0 0000 110
0 0000 111
0 0001 000
0 0001 001
...
0 0110 110
0 0110 111
0 0111 000
0 0111 001
0 0111 010
...
0 1110 110
0 1110 111
0 1111 000
0 1111 001
...
0 1111 111
```

Dynamic Range of 8-bit tiny float

s	exp	frac
0	0000	000
0	0000	001
0	0000	010
...		
0	0000	110
0	0000	111
0	0001	000
0	0001	001
...		
0	0110	110
0	0110	111
0	0111	000
0	0111	001
0	0111	010
...		
0	1110	110
0	1110	111
0	1111	000
0	1111	001
...		
0	1111	111

Dynamic Range of 8-bit tiny float

Bias = 7

	s	exp	frac
$V = (-1)^s$	0	0000	000
$\times (0.\text{frac})$	0	0000	001
$\times 2^{(1 - \text{Bias})}$	0	0000	010
Denormalized numbers	...	0 0000	110
		0 0000	111
.....			
Normalized numbers	0	0001	000
	0	0001	001
	...	0 0110	110
$V = (-1)^s$	0	0110	111
$\times (1.\text{frac})$	0	0111	000
$\times 2^{(\text{exp} - \text{Bias})}$	0	0111	001
	0	0111	010
	...	0 1110	110
	0	1110	111
.....			
Special values	0	1111	000
	0	1111	001
	...	0 1111	111

Dynamic Range of 8-bit tiny float

Bias = 7

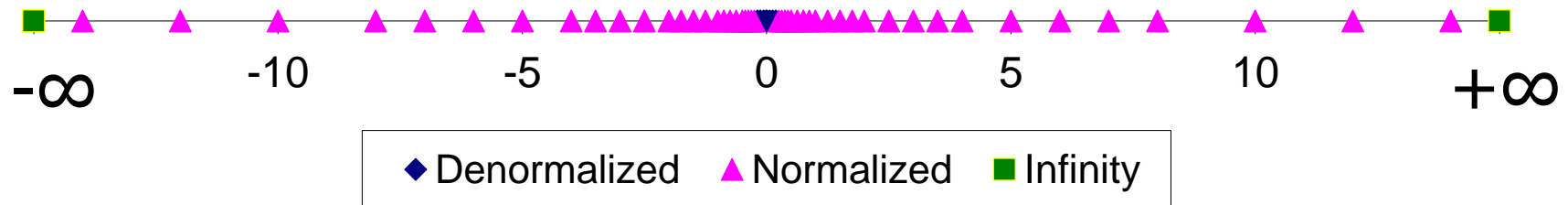
	s	exp	frac	E	Value
$V = (-1)^s$	0	0000	000	-6	0
$\times (0.\text{frac})$	0	0000	001	-6	$1/8 * 1/64 (2^{-6}) = 1/512$
$\times 2^{(1 - \text{Bias})}$	0	0000	010	-6	$2/8 * 1/64 = 2/512$
	...				
Denormalized numbers	0	0000	110	-6	$6/8 * 1/64 = 6/512$
	0	0000	111	-6	$7/8 * 1/64 = 7/512$
<hr/>					
Normalized numbers	0	0001	000	-6	$8/8 * 1/64 = 8/512$
	0	0001	001	-6	$9/8 * 1/64 = 9/512$
	...				
	0	0110	110	-1	$14/8 * 1/2 = 14/16$
$V = (-1)^s$	0	0110	111	-1	$15/8 * 1/2 = 15/16$
$\times (1.\text{frac})$	0	0111	000	0	$8/8 * 1 = 1$
$\times 2^{(\text{exp} - \text{Bias})}$	0	0111	001	0	$9/8 * 1 = 9/8$
	0	0111	010	0	$10/8 * 1 = 10/8$
	...				
	0	1110	110	7	$14/8 * 128 = 224$
	0	1110	111	7	$15/8 * 128 = 240$
<hr/>					
Special values	0	1111	000	n/a	inf
	0	1111	001	n/a	NaN
	...				
	0	1111	111	n/a	NaN

Dynamic Range of 8-bit tiny float

Bias = 7	s	exp	frac	E	Value	Notes of Interest
$V = (-1)^s$	0	0000	000	-6	0	
$\times (0.\text{frac})$	0	0000	001	-6	$1/8 * 1/64 (2^{-6}) = 1/512$	closest to zero
$\times 2^{(1 - \text{Bias})}$	0	0000	010	-6	$2/8 * 1/64 = 2/512$	
Denormalized numbers	...					
	0	0000	110	-6	$6/8 * 1/64 = 6/512$	
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	largest denorm
<hr/>						
Normalized numbers	0	0001	000	-6	$8/8 * 1/64 = 8/512$	smallest norm > 0
	0	0001	001	-6	$9/8 * 1/64 = 9/512$	
	...					
	0	0110	110	-1	$14/8 * 1/2 = 14/16$	
$V = (-1)^s$	0	0110	111	-1	$15/8 * 1/2 = 15/16$	closest to 1 below
$\times (1.\text{frac})$	0	0111	000	0	$8/8 * 1 = 1$	
$\times 2^{(\text{exp} - \text{Bias})}$	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
<hr/>						
Special values	0	1111	000	n/a	inf	
	0	1111	001	n/a	NaN	
	...					
	0	1111	111	n/a	NaN	

Distribution of Values

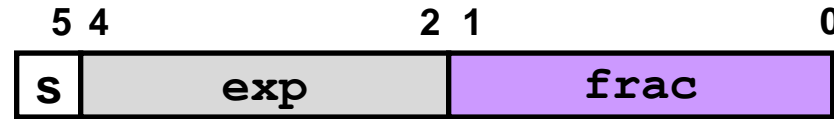
- 6-bit IEEE-like format
 - exp = 3 exponent bits
 - frac = 2 fraction bits
 - Bias is 3 ($2^{3-1}-1$)
- Notice how the distribution gets denser toward zero.



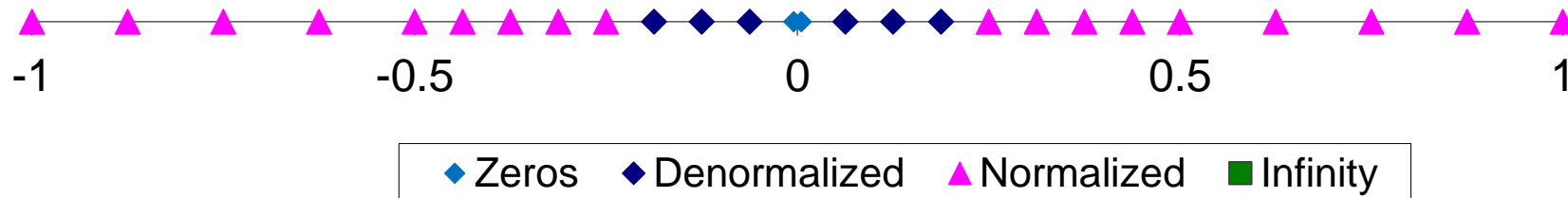
Distribution of Values (Close-up View)

- 6-bit IEEE-like format

- exp = 3 exponent bits
- frac = 2 fraction bits
- Bias is 3 ($2^{3-1}-1$)



- Smooth transition between normalized and de-normalized numbers due to definition $E = 1 - \text{Bias}$ for denormalized values
 - Zeros are denormalized numbers too! (+0 and -0)



Outline

- Fractional Binary Numbers
- Representing Floating Point
- Smaller Floating Point
- **Floating Point Arithmetic**

Floating Point Operations

- Conceptual view

- $x +_{\text{float}} y = \text{Fit}(x +_{\text{math}} y)$
- $x *_{\text{float}} y = \text{Fit}(x *_{\text{math}} y)$

- First compute exact, mathematical result

- Compute the numerical value of the operands
- Do the operation as in grade school arithmetic

- Then make it fit into desired precision

- **Step 1:** Determine frac, exp
 - Frac must be of the form 1.xxxx (0.xxx if denormalized)
 - Change exp if needed to get frac to that form (e.g., result is 101.xxx)
- **Step 2:** Possibly overflow if exponent too is large
 - Unlike integer overflow, result is mathematically reasonable: infinity
- **Step 3:** Possibly round to fit into frac if we have too many mantissa bits

Rounding

- Default rounding mode for IEEE floating point is Round-to-even
 - Other methods are statistically biased (round up, round down, round-to-zero)
 - Sum of set of positive numbers will consistently be over- or under- estimated
 - Round to nearest number
 - If **exactly** in between, round to nearest **even** number
- Round-to-even example
 - Illustrated with rounding of money

	\$1.40	\$1.60
Rounded	\$1	\$2

Rounding

- Default rounding mode for IEEE floating point is Round-to-even
 - Other methods are statistically biased (round up, round down, round-to-zero)
 - Sum of set of positive numbers will consistently be over- or under- estimated
 - Round to nearest number
 - If **exactly** in between, round to nearest **even** number
- Round-to-even example
 - Illustrated with rounding of money

	\$1.40	\$1.60	\$1.50	\$2.50	-\$1.50
Rounded	\$1	\$2			

Rounding

- Default rounding mode for IEEE floating point is Round-to-even
 - Other methods are statistically biased (round up, round down, round-to-zero)
 - Sum of set of positive numbers will consistently be over- or under- estimated
 - Round to nearest number
 - If **exactly** in between, round to nearest **even** number
- Round-to-even example
 - Illustrated with rounding of money

	\$1.40	\$1.60	\$1.50	\$2.50	-\$1.50
Rounded	\$1	\$2	\$2	\$2	-\$2

Closer Look at Round-to-even

- Rounding to other decimal places than the decimal point
 - When exactly halfway between two possible values
 - Round so that least significant digit is even
- E.g., round to nearest hundredth (i.e., 2 decimal digits in fractional part)
 - $1.23\textbf{49999}$ \Rightarrow 1.23 (Less than half way)
 - $1.23\textbf{50001}$ \Rightarrow 1.24 (Greater than half way)
 - $1.23\textbf{50000}$ \Rightarrow 1.24 (Half way—round to even)
 - $1.24\textbf{50000}$ \Rightarrow 1.24 (Half way—round to even)

Rounding Binary Numbers

- Binary fractional numbers

- Are "even" when least significant bit is 0
- Are half-way when bits to right of rounding position = $100...0_2$
 General form $XX...X.YY...Y100...0_2$
 last Y is the position to which we want to round

- Examples

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

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

Mathematical Properties of FP Arithmetic

- Mathematical properties of FP Addition
 - Addition is Associative? **NO**
 - $(x + y) + z = x + (y + z)$
 - Possibility of overflow and inexactness of rounding
 - $(3.14 + 1e10) - 1e10 = 0$ (rounding)
 - $3.14 + (1e10 - 1e10) = 3.14$
- Mathematical properties of FP Multiplication
 - Multiplication is Associative? **NO**
 - $(x \times y) \times z = x \times (y \times z)$
 - Possibility of overflow, inexactness of rounding
 - Multiplication distributes over addition? **NO**
 - $x \times (y + z) = (x \times y) + (x \times z)$
 - Possibility of overflow, inexactness of rounding
- More in bonus slides

Floating Point Summary

- IEEE Floating point (IEEE 754) has clear mathematical properties
 - But not always the ones you may expect!
- Represents numbers of form $(-1)^S \times M \times 2^E$
- One can reason about operations independent of implementation
 - As if computed with perfect precision and then rounded
- Not the same as arithmetic on real numbers
 - Violates associativity/distributivity
 - Makes life difficult for compilers & serious numerical applications programmers

Outline

- Fractional Binary Numbers
- Representing Floating Point
- Smaller Floating Point
- Floating Point Arithmetic

Outline

- Bonus slides
 - Use these for additional practice
 - And if you're interested in additional topics

Interesting Numbers for `float/double`

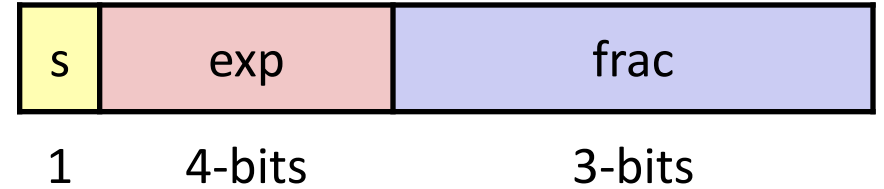
Description	exp	frac	Numeric Value ^{single prec., double prec.}
Zero	00...00	00...00	0.0
Smallest Pos. Denorm. <ul style="list-style-type: none"> • Single $\sim 1.4 \times 10^{-45}$ • Double $\sim 4.9 \times 10^{-324}$ 	00...00	00...01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
Largest Denormalized 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

Normalized Encoding Example

- Value
 - `float F = 12345.0; // single precision: k=8, n=23`
 - $12345_{10} = 11000000111001_2 = 1.1000000111001_2 \times 2^{13}$
- Significand
 - $M = 1.1000000111001_2$
 - $\text{frac} = 1000000111001$ **0000000000**
(drop leading 1, add 10 zeros)
- Exponent
 - $E = 13$
 - Bias = 127
 - $E = \text{exp} - \text{Bias} \rightarrow \text{exp} = E + \text{Bias} = 140 = 10001100_2$

Floating Point Representation:								
Hex:	4	6	4	0	E	4	0	0
Binary:	0100	0110	0100	0000	1110	0100	0000	0000

Creating a Floating Point Number



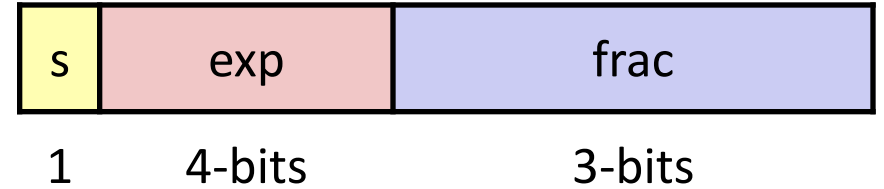
- Steps

- Is the number within the range $(-2^{1-\text{Bias}}, +2^{1-\text{Bias}})$?
 - If yes, “denormalize” to have a leading 0
 - otherwise, normalize to have leading 1
- Round to fit within fraction
- Postnormalize to deal with effects of rounding

- QUIZ in next three slides

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

Step 1: Normalize

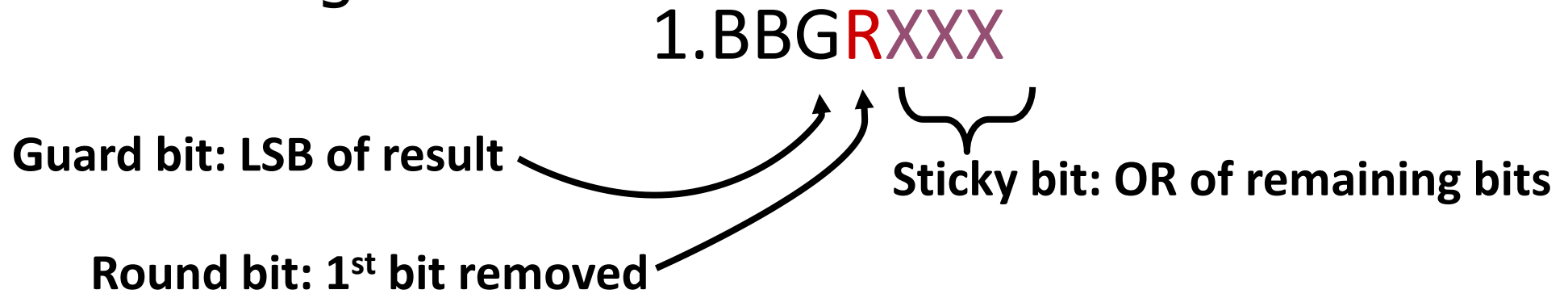


- Requirement

- 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	1.0000000	7
13	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Step 2: Rounding



- Round up conditions

- round up if $\langle \text{Guard, Round, Sticky} \rangle = \langle x11 \rangle$ because >0.5
- round up if $\langle \text{Guard, Round, Sticky} \rangle = \langle 110 \rangle$ as per round to even rules

<i>Value</i>	<i>Fraction</i>	<i>GRS</i>	<i>Incr?</i>	<i>Rounded</i>
128	1.000 0 000	000	N	1.000
13	1.101 0 000	100	N	1.101
17	1.000 1 000	010	N	1.000
19	1.001 1 000	110	Y	1.010
138	1.000 1 010	011	Y	1.001
63	1.111 1 100	111	Y	10.000

Step 3: 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
13	1.101	3		13
17	1.000	4		16
19	1.010	4		20
138	1.001	7		144
63	10.000	5	M=1.000 exp=6	64

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) (double) x
```

```
x == (int) (float) x
```

```
d == (double) (float) d
```

```
f == (float) (double) f
```

```
f == -(-f);
```

```
1.0/2 == 1/2.0
```

```
d*d >= 0.0
```

```
(f+d) - f == d
```

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**

<code>x == (int) (double) x</code>	<i>Yes</i>
<code>x == (int) (float) x</code>	<i>No (x = TMax)</i>
<code>d == (double) (float) d</code>	<i>No (d = 1e40)</i>
<code>f == (float) (double) f</code>	<i>Yes</i>
<code>f == -(-f);</code>	<i>Yes</i>
<code>1.0/2 == 1/2.0</code>	<i>Yes</i>
<code>d*d >= 0.0</code>	<i>Yes</i>
<code>(f+d) - f == d</code>	<i>No (f = 1.0e20, d = 1.0; f+d rounded to 1.0e20)</i>

Floating-Point Multiplication, Directly

- For cases where you can't work with exact results
 - E.g., when doing it in hardware

- Operands

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

- Exact result

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

- Sign s: $s1 \wedge s2$

- Significand M: $M1 * M2$

- 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

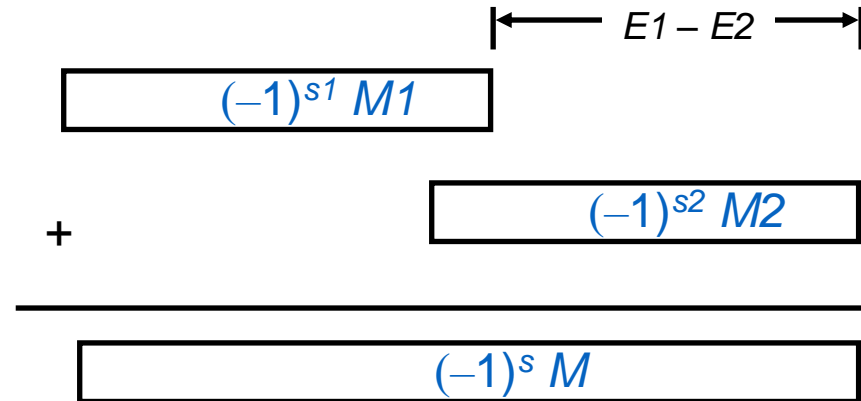
E1=3	M1=1.11010010
E2=5	M2=1.11001110

E=8	M=11.01001000111111
E=8+1	M=1.101001000111111
E=9	M=1.1010010010

Floating-Point Addition, Directly

- Operands

- $(-1)^{s1} M1 2^{E1}$
- $(-1)^{s2} M2 2^{E2}$
- Assume $E^1 > E^2$



- Exact Result

- $(-1)^s M 2^E$
- Sign s , significand M : Result of signed align & add
- Exponent E : E^1

- Fixing

- If $M \geq 2$, shift M right, increment E
- if $M < 1$, shift M left k places, decrement E by k
- Overflow if E out of range
- Round M to fit frac precision

```

E1=5 M1=1.11010010
E2=2 M2=1.11001110
E2=2 M2=0001.11001110
-----
E1=5 M1=1.11010010
E2=5 M2=0.00111001110
-----
E =5  M =10.00001011110
E =6  M =1.000001011110
    
```


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
 - $(3.14+1e10)-1e10=0$ (rounding)
 - $3.14+(1e10-1e10)=3.14$
 - 0 is additive identity? YES
 - Every element has additive inverse? ALMOST
 - Except for infinities & NaNs
- Monotonicity
 - $a \geq b \Rightarrow a+c \geq b+c$? ALMOST
 - Except for NaNs

Mathematical Properties of FP Multiplication

- 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
 - 1 is multiplicative identity? YES
 - Multiplication distributes over addition? NO
 - Possibility of overflow, inexactness of rounding
- Monotonicity
 - $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c$? ALMOST
 - Except for NaNs