[CSED211] Introduction to Computer Software Systems

Lecture 3: Floating point

Prof. Jisung Park

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.09.13

Lecture Agenda: 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`$_{(2)}$?
  - The same principle as base 10 numbers

2^i	2^{i-1}	•••	2^2	2^1	2^0					
b_i	b_{i-1}	•••	b_2	b_1	b_0	b_{-1}	b_{-2}	b_{-3}	•••	b_{-j}
						2^{-1}	2^{-2}	2^{-3}	•••	2^{-j}

- 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

Value	Representation
$5\frac{3}{4}$	`101.11`$_{(2)}$
$2\frac{7}{8}$	`10.111`$_{(2)}$
$1\frac{7}{16}$	`1.0111`$_{(2)}$

- Observations
  - Divide by 2: shifting right
  - Multiply by 2: by shifting left
  - `0.111111`…$_{(2)}$ is just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots = 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

- Limitation#1
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations
  - Value / Representation

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?)

Lecture Agenda: 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 (Institute of Electrical and Electronics Engineers) Standard 754 (IEEE754-1985)
    - Established in 1985 as uniform standard for floating point arithmetic
        - Before that, many idiosyncratic formats
    - Supported by all major CPUs
    - Merged with IEEE854-1987 in 2008 (IEEE754-2008)
    - Recently published with minor revision (IEEE754-2019)

- Driven by numerical concerns
    - Nice standards for rounding, overflow, underflow, etc.
    - Hard to make fast in hardware
        - Numerical analysts predominated over hardware designers in defining standard

Floating Point Representation (Format)

- Numerical form: $(-1)^s \times M \times 2^E$
  - 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 *e* encodes E (not exactly the same as E)
  - Frac field *f* encodes M (not exactly the same as M)

s	***e* (exp)**	***f* (frac)**

Precisions

- Single precision: 32 bits

s	e	f
1	8 bits	23 bits

- Double precision: 64 bits

s	e	f
1	11 bits	52 bits

- **Extended precision** (Intel only): 80 bits

s	e	f
1	15 bits	63 or 64 bits

Normalized Values

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

- When: $e \neq$ `000…0` and $e \neq$ `111…1`

- Exponent coded as biased value: $E = Exp - Bias$
  - Exp: unsigned value e
  - $Bias$: $2^{k-1} - 1$, where k is the number of exponent bits
    - Single precision (8-bit exp): 127 (Exp: 1, …, 254 map to E: -126, …, 127)
    - Double precision (11-bit exp): 1023 (Exp: 1, …, 2046 map to E: -1022, …, 1023)

- Significand coded with implied leading 1: $M = \mathtt{1.xxx...x}_{(2)}$
  - $f_{i-1}, f_{i-2}, \ldots, f_0$ represent the mantissa part **xxx…x**
  - Minimum when `000…0` (M = 1.0)
  - Maximum when `111…1` ($M = 2.0 - \varepsilon$)
  - 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 = $1.\underline{1101101101101}_{(2)}$
  - f = $\underline{1101101101101}0000000000_{(2)}$
- Exponent
  - E = 13
  - *Bias* = 127
  - *Exp* (*e*) = 140 = $10001100_{(2)}$
- Result

0	1000 1100	110 1101 1011 0100 0000 0000
s	*e* (exp)	*f* (frac)

$v = (-1)^s \times M \times 2^E$
$E = Exp - Bias$

Denormalized Values

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

- When e = `000…0`
- Exponent $E = 1 - Bias$ (instead of $E = -Bias$)
- Significand coded with implied leading 0: M = `0.xxx…x`$_{(2)}$
    - $f_{i-1}, f_{i-2}, \ldots, f_0$ represent the mantissa part `xxx…x`
- Two purposes
    - To represent zero value: f = `000…0`
        - Note distinct values: +0 and −0 (why?)
    - To represent numbers very close to 0: $f \neq$ `000…0`
        - Lose precision as get smaller
        - Gradual underflow: for numbers smaller than the minimum normalized value

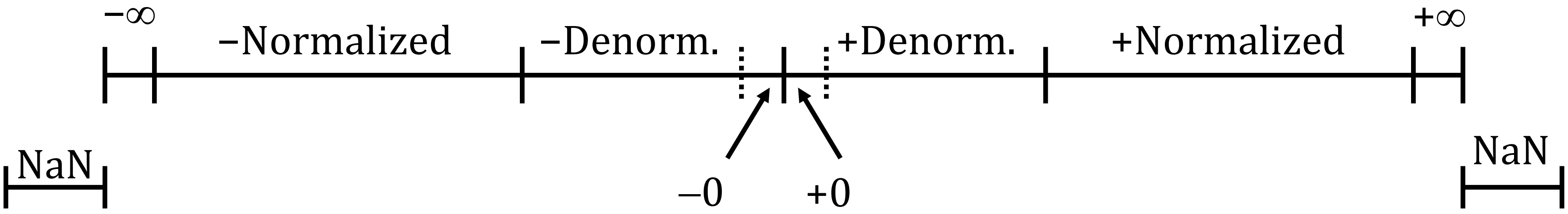
Special Values

- When e = **`111…1`**

- If f = **`000…0`**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - e.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

- If $f \neq$ **`000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - e.g., **`sqrt(-1)`**, $\infty - \infty$, $\infty \times 0$

Visualization: Floating Point Encodings



Lecture Agenda: 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	*e*	*f*
1	4 bits	3 bits

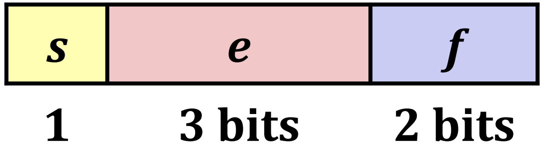
- 8-bit floating point representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the fraction part
- The same general format as IEEE format
  - For normalized and denormalized numbers
  - For special values to represent 0, NaN, and infinity

Dynamic Range (Positive Only)

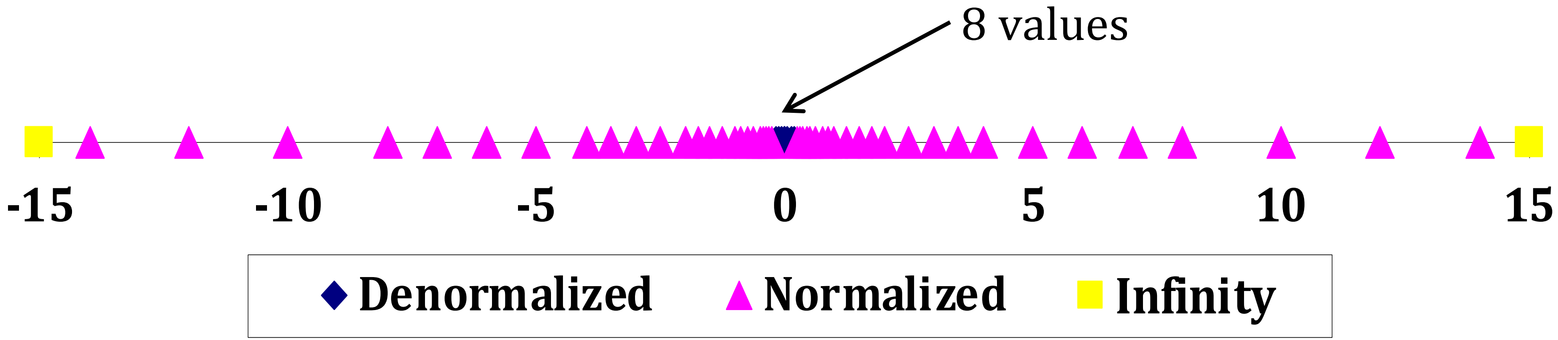
	s	exp	frac	E	Value	
	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	**Closest to zero**
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	**Largest denorm.**
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	**Smallest norm.**
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	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	
	0	1111	000	n/a	inf	**Largest norm**

Distribution of Values

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

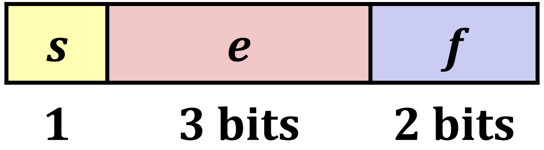


- The closer the values to the origin (0), the denser the distribution

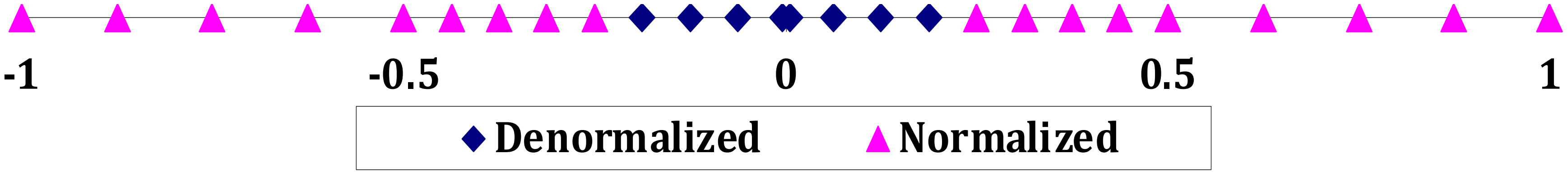


Distribution of Values (Close-Up View)

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



- Gradual underflow: the same distances b/w adjacent denormalized values as adjacent normalized values in the previous ranges (i.e., equispace)



Special Properties of Encoding

- Floating-point zero is the same as the integer zero
  - All bits are ‘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
      - Denormalized vs. normalized
      - Normalized vs. infinity

Creating Floating Point Number

- Three steps
  - Step 1: Normalize to have leading 1
  - Step 2: Round to fit within fraction
  - Step 3: Post-normalize to deal with effects of rounding

s	***e***	***f***
1	**4 bits**	**3 bits**

- Case Study: convert 8-bit unsigned numbers to tiny floating-point format

Value	**Numbers**
$128_{(10)}$	$10000000_{(2)}$
$15_{(10)}$	$00001101_{(2)}$
$17_{(10)}$	$00010001_{(2)}$
$19_{(10)}$	$00010011_{(2)}$
$138_{(10)}$	$10001010_{(2)}$
$63_{(10)}$	$00111111_{(2)}$

Step 1: Normalization

s	e	f
1	4 bits	3 bits

- Set the binary point so that the number to have leading one
  - i.e., to be in a form of `1.xxxxx`...
  - While incrementing the exponent as shift the binary point to the left

- Case Study: convert 8-bit unsigned numbers to tiny floating-point format

Value	Numbers	Fraction	Exponent
$128_{(10)}$	$10000000_{(2)}$	$1.0000000_{(2)}$	7
$15_{(10)}$	$00001101_{(2)}$	$1.1010000_{(2)}$	3
$17_{(10)}$	$00010001_{(2)}$	$1.0001000_{(2)}$	4
$19_{(10)}$	$00010011_{(2)}$	$1.0011000_{(2)}$	4
$138_{(10)}$	$10001010_{(2)}$	$1.0001010_{(2)}$	7
$63_{(10)}$	$00111111_{(2)}$	$1.1111100_{(2)}$	5

Rounding

`1.BBGRXXX`

Guard bit `G`: LSB of result

Round bit `R`: 1st bit to remove

Sticky bit `S`: OR of remaining bits

- Principle
  - `R = 0` → Discard the remaining bits (∵ remainders < 0.5)
  - `R = 1` and `S = 1` → Increase G (∵ remainders > 0.5)
  - `R = 1` and `S = 0` (i.e., remainders = 0.5) → Round to even

Value	Fraction	GRS	Increase?	Rounded
$128_{(10)}$	$1.0000000_{(2)}$	000	N	1.000
$15_{(10)}$	$1.1010000_{(2)}$	100	N	1.101
$17_{(10)}$	$1.0001000_{(2)}$	010	N	1.000
$19_{(10)}$	$1.0011000_{(2)}$	110	Y	1.010
$138_{(10)}$	$1.0001010_{(2)}$	011	Y	1.001
$63_{(10)}$	$1.1111100_{(2)}$	111	Y	10.00

Post-Normalization

- Issue: rounding may have caused overflow
  - Handle by shifting right once & incrementing exponent

Value	Rounded	Exp	Adjusted	Result
$128_{(10)}$	1.000	7		$128_{(10)}$
$15_{(10)}$	1.101	3		$15_{(10)}$
$17_{(10)}$	1.000	4		$16_{(10)}$
$19_{(10)}$	1.010	4		$20_{(10)}$
$138_{(10)}$	1.001	7		$142_{(10)}$
$63_{(10)}$	10.00	5	1.000/6	$64_{(10)}$

Lecture Agenda: 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 *f*

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 (to $-\infty$)	\$1	\$1	\$1	\$2	–\$2
Round up (to $+\infty$)	\$2	\$2	\$2	\$3	–\$1
Nearest even (default)	\$1	\$2	\$2	\$2	–\$2

- What are the advantages of each mode?

Closer Look at Round-to-Even

- Default rounding mode
  - All other modes are statistically biased: sum of a 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 the least significant bit is 0
  - Half-way when bits to right of rounding position = `100`…$_{(2)}$

- Examples: round to nearest 1/4 (2-bit 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

Floating Point Multiplication

- $\{(-1)^{s1} \times M1 \times 2^{E1}\} \times \{(-1)^{s2} \times M2 \times 2^{E2}\} = (-1)^{s} \times M \times 2^{E}$, where
  - Sign s: s1 ^ s2
  - Significand M: $M1 \times M2$
  - Exponent E: $E1 + E2$

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

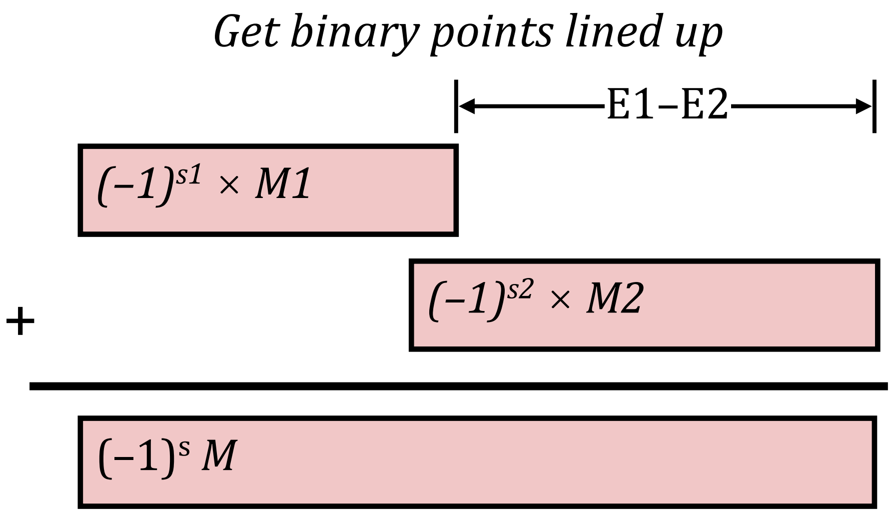
- Implementation
  - Biggest chore is multiplying significands

Floating Point Addition

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

- Exact Result: $(-1)^{s} \times M \times 2^{E}$, where
  - Sign s, significand M:
    Result of signed align & add
  - Exponent E: E1

- Adjustment
  - If $M \geq 2$, shift M right, increment E
  - If $M < 1$, shift M left, decrement E
  - Overflow if E out of range
  - Round M to fit the given precision



Mathematical Properties of Floating Point Add.

- Compare to those of Abelian Group
  - Closed under addition? Yes: but may results in infinity or NaN
  - Associative? No: Overflow and inexactness of rounding
  - Identity element? 0
  - Inverse element? Yes, except for infinity or NaN
  - Commutative? Yes

- Monotonicity
  - $a \geq b \Rightarrow a+c \geq b+c$? Yes, except for infinity or NaN

Mathematical Properties of FP Mult

- Compare to Commutative Ring
  - Closed under multiplication? Yes: but may results in infinity or NaN
  - Associative? No: Overflow and inexactness of rounding
  - Identity element? 1
  - Inverse element? Yes, except for infinity or NaN
  - Commutative? Yes
  - Multiplication distributes over addition? No: Overflow and inexactness of rounding

`1e20*(1e20-1e20)=0.0,  1e20*1e20-1e20*1e20=NaN`

- Monotonicity
  - $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$? Yes, except for infinity or NaN

Lecture Agenda: 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 supports two precision levels
  - `float` single precision
  - `double` double precision

- Conversions and 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`
    - May be rounded according to rthe ounding 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

```c
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 ⇒ ((d*2) < 0.0)` ✓
- `d > f ⇒ -f > -d` ✓
- `d*d >= 0.0` ✓
- `(d+f)-d == f` ✗

Interesting Numbers

Description	*e*	*f*	*Numeric Value*
• Zero	`00…00`	`00…00`	0.0
• Smallest Pos. Denorm.	`00…00`	`00…01`	$2^{-\{23, 52\}} \times 2^{-\{126, 1022\}}$
○ Single ≈ 1.4×10^{-45}			
○ Double ≈ 4.9×10^{-324}			
• Largest Denormalized.	`00…00`	`11…11`	$(1.0 - \varepsilon) \times 2^{-\{126, 1022\}}$
○ Single ≈ 1.18×10^{-38}			
○ Double ≈ 2.2×10^{-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 ≈ 3.4×10^{38}			
○ Double ≈ 1.8×10^{308}			

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $(-1)^s \times M \times 2E$
- 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

[CSED211] Introduction to Computer Software Systems

Lecture 3: Floating point

Prof. Jisung Park

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.09.13