Floating Point

How to represent floating numbers?

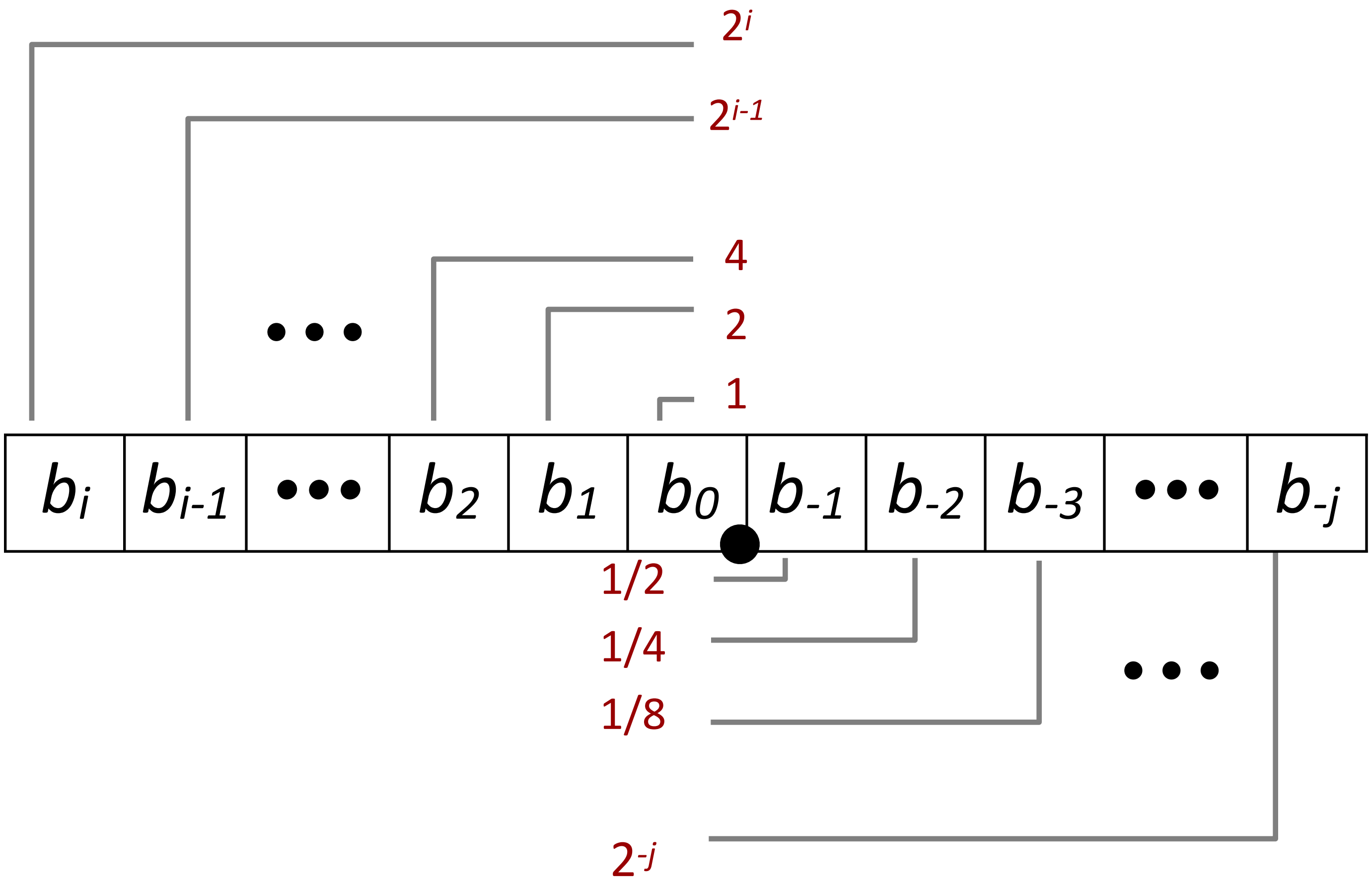
Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding
- Summary

Fractional Binary Numbers

- What is 1011.101_2?

Fractional Binary Numbers



Representation

- Bits to right of "binary point" represent fractional powers of 2

Fractional Binary Numbers: Examples

- **Value** **Representation**

Value	Representation
5 3/4	101.11_2
2 7/8	10.111_2
1 7/16	1.0111_2

- **Observations**
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - 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
  - 1/3 `0.0101010101[01]`$\ldots_2$
  - 1/5 `0.001100110011[0011]`$\ldots_2$
  - 1/10 `0.0001100110011[0011]`$\ldots_2$

Limitation #2

- Just one setting of binary 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
- 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 (inspired from the scientific notation):**

$$(-1)^s \, M \; 2^E$$

  - **Sign bit *s*** determines whether number is negative or positive
  - **Significand *M***
  - **Exponent *E***
- **Example (scientific notation in the decimal system)**
  - $-52000 = -1 \times 52 \times 10^3$
- **Encoding**
  - Most significant bit 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)

s	exp	frac

Precision options

- **Single precision: 32 bits**

s	exp	frac
1	8-bits	23-bits

- **Double precision: 64 bits**

s	exp	frac
1	11-bits	52-bits

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

s	exp	frac
1	15-bits	63 or 64-bits

Exponent and Fraction Encoding

$$\boxed{v = (-1)^s \, M \, 2^E}$$

- How to encode E and M in Exp and Frac?

s	exp	frac

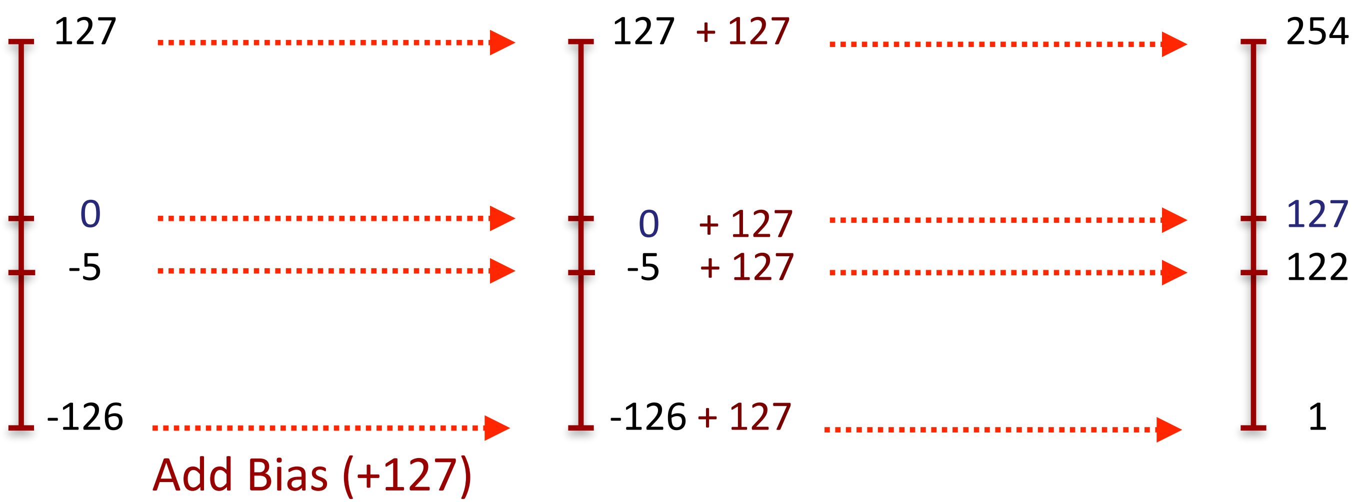
- If the floating number is <u>not</u> close to 0
  - Use the normalized encoding
- Else
  - Use the denormalized encoding

How to Represent Negative Values in Exp?

- There are many methods to represent negative numbers
  - Two's complement
  - Bias method

Representing Negative Numbers Using Bias

- **Bias method:**
  - **Let's assume you have the following range of values [-126, 127]**
  - **Let's assume you want to represent (-5) in that range**
  - **You can shift the starting point of the range by adding +127 to make it start from 1 (i.e., make the range [1, 254])**
  - **(-5) will become +122**



Normalized Encoding Example

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

$E = Exp - Bias$

- **Value:** **`float F = 15213.0;`**
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$
- **You can also write F in other ways**
  - `F` $= 0.11101101101101_2 \times 2^{14}$
    $= 11.101101101101_2 \times 2^{12}$
    $= 111.01101101101_2 \times 2^{11}$

Normalized Encoding Example

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

$E = Exp - Bias$

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**

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

  $\texttt{frac} = \underline{1101101101101}0000000000_2$

- **Exponent**

  $E = 13$

  $Bias = 127$

  $Exp = 140 = 10001100_2$

- **Result:**

0	10001100	11011011011010000000000
`s`	`exp`	`frac`

“Normalized” Values

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

$E = Exp - Bias$

- **When: exp ≠ 000…0 and exp ≠ 111…1**

- **Exponent coded as a *biased* value: $E = Exp - Bias$**
  - *Exp*: unsigned value of exp field
  - *Bias* = $2^{k-1} - 1$, where *k* is number of exponent bits
    - Single precision: Bias = 127
      - This bias shifts the range of Exp from -126…127 to 1…254
    - Double precision: Bias = 1023
      - This bias shifts the range of Exp from 1…2046 to -1022…1023

- **Significand coded with implied leading 1: $M = 1.\text{xxx…x}_2$**
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Get extra leading bit for “free”

Practice

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

$E = Exp - Bias$

- Represent 3.625 in the normalized encoding of floating points
- $3.625_{10} = 11.101_2$

- exp = ?
- frac = ?

Practice

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

$E = Exp - Bias$

- $3.625_{10} = 11.101_2$

s	exp	frac
1	8-bits	8-bits

- $11.101 = 1.\underline{1101} \times 2^1$
  - $M = 1.\underline{1101}$ ===> frac = $\underline{1101}$
  - $E = 1$
    - $Exp = E + Bias$
    - $Bias = 2^{k-1} - 1 = 2^{8-1} - 1 = 2^7 - 1 = 127$
    - ===> $Exp = 1 + 127 = 128_{10} = 10000000_2$

0	1000 0000	1101 0000
1	8-bits	8-bits

How to represent values close to 0?

- Normalized representation cannot represent 0, why?

- You can only represent 1.xxxxxx

- Solution
  - For values close to 0, use the denormalized representation

Denormalized Values

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

$E = 1 - Bias$

- **Condition:** exp = 000…0
- **Exponent value:** $E = 1 - \text{Bias}$ **(instead of** $E = 0 - Bias$**)**
- **Significand coded with implied leading 0:** $M = 0.\text{xxx}\ldots\text{x}_2$
  - **`xxx`**…**`x`**: bits of **`frac`**
- **Cases**
  - **`exp`** = 000…0, **`frac`** = 000…0
    - Represents zero value
    - Note distinct values: +0 and –0
  - **`exp`** = 000…0, **`frac`** ≠ 000…0
    - Numbers closest to 0.0

Special Values

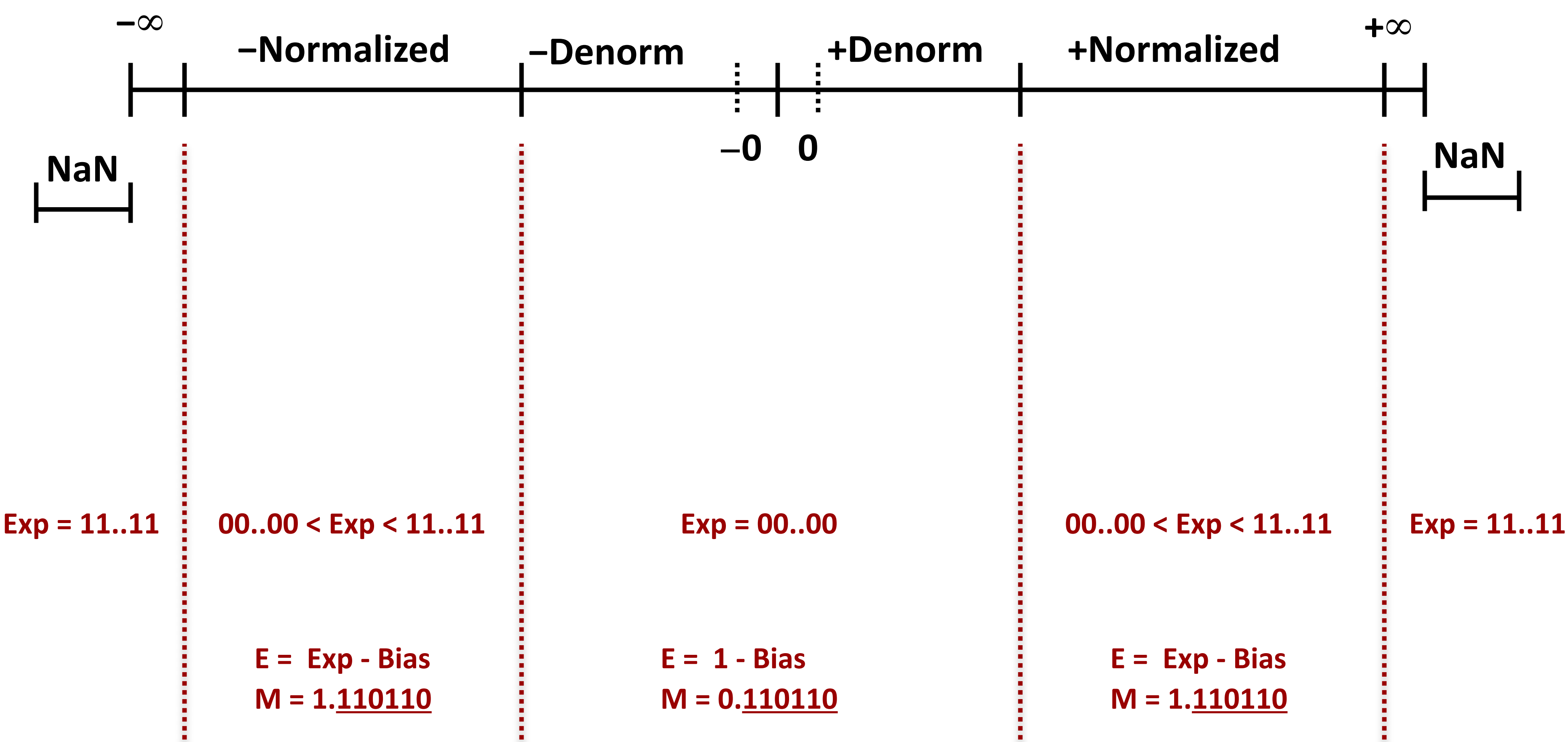
- **Condition: `exp` = `111`…`1`**

- **Case: `exp` = `111`…`1`, `frac` = `000`…`0`**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

- **Case: `exp` = `111`…`1`, `frac` ≠ `000`…`0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

Visualization: Floating Point Encodings

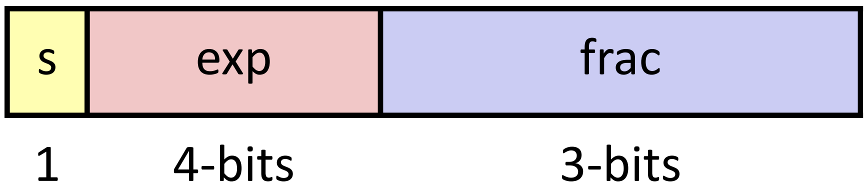
$$v = (-1)^s \, M \, 2^E$$



Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- **Example and properties**
- Rounding
- 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)

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

n: E = Exp – Bias

d: E = 1 – Bias

	s	exp	frac	E	Value	
	0	0000	000	-6	0	
Denormalized numbers	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	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
Normalized numbers	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	

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- **Rounding**
- Summary

Floating Point Operations: Basic Idea

- $x +_f y = \mathrm{Round}(x + y)$

- $x \times_f y = \mathrm{Round}(x \times y)$

- **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 (−∞)	$1	$1	$1	$2	–$2
Round up (+∞)	$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 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

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	10.00011_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 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
- **`int`** → **`double`**
  - Exact conversion, as long as **`int`** has ≤ 53 bit word size
- **`int`** → **`float`**
  - Will round according to rounding mode

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding
- **Summary**

Summary

- **IEEE Floating Point has clear mathematical properties**
- **Represents numbers of form M x 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