Floating Point

CSE 238/2038/2138: Systems Programming

Instructor:

Fatma CORUT ERGİN

Slides adapted from Bryant & O'Hallaron's slides

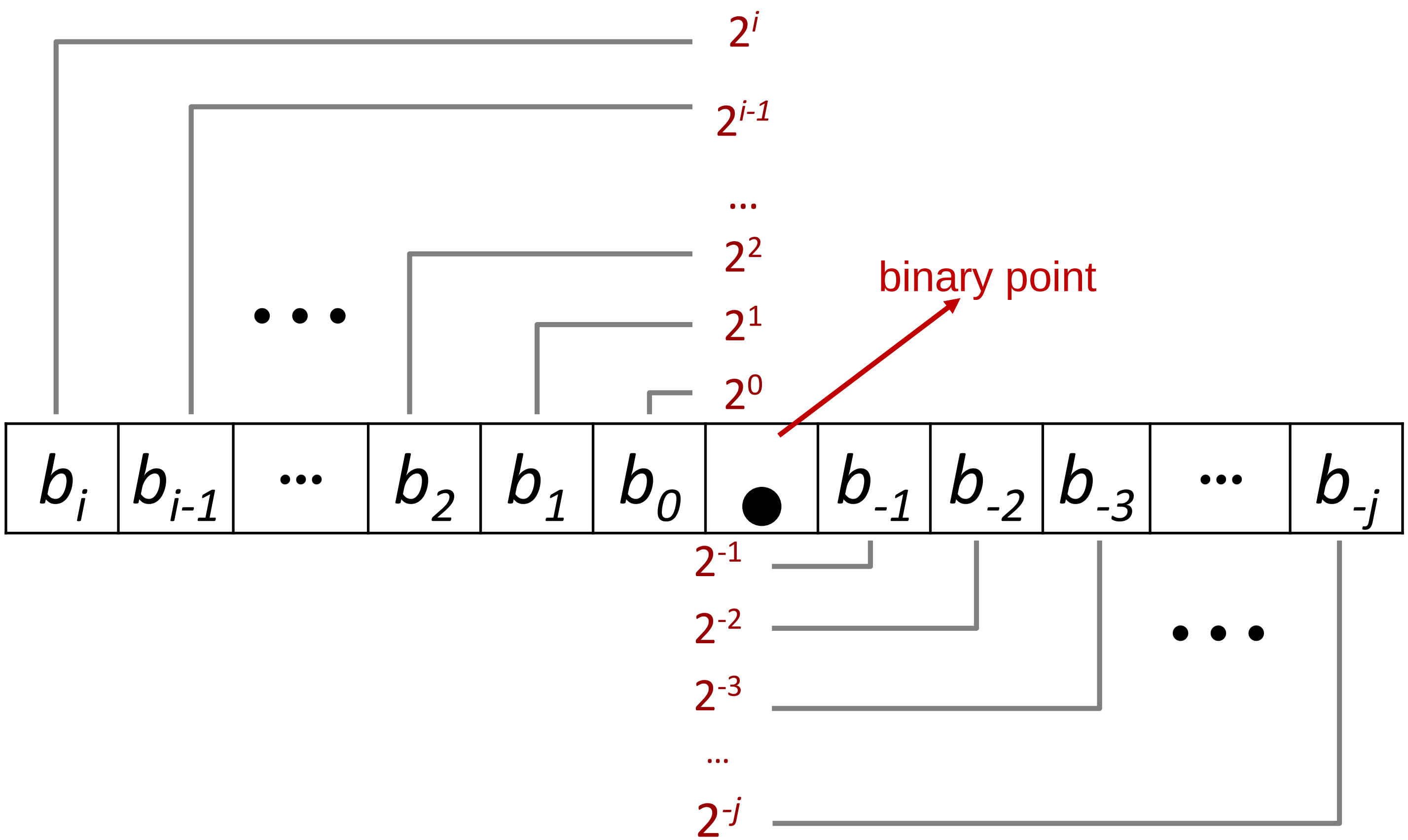
Today: Floating Point

- **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

What is X = 1011.101_2?

$$\sum_{k=-j}^{i} b_k \times 2^k$$

X = $1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3}$

X = $11\frac{5}{8}$

Fractional Binary Numbers: Examples

Value	Representation
$5\frac{3}{4}$	101.11_2
$2\frac{7}{8}$	010.111_2
$1\frac{7}{16}$	001.0111_2

- **Observations**
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{2^i}+\ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

- **Limitation #1**
  - Only numbers of the form $\frac{x}{2^k}$ can be represented exactly
    - Other rational numbers have repeating bit representations

Value	Representation
$\frac{1}{3}$	$0.0101010101[01]..._2$
$\frac{1}{5}$	$0.001100110011[0011]..._2$
$\frac{1}{10}$	$0.0001100110011[0011]..._2$

- **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, 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 machine specific 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

IEEE Floating Point Representation

Example: $(12345_{10} = 0011000000111001_2)$
$12345.0_{10} = (-1)^0 * 1.1000000111001_2 * 2^{13}$

- **Numerical Form:**

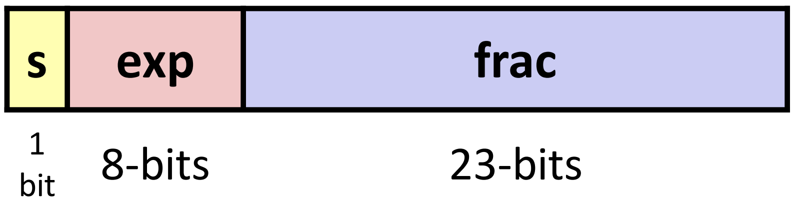
$$(-1)^s \times M \times 2^E$$

  - **Sign bit** ***s*** determines whether number is negative or positive
  - **Significand** (mantissa) ***M*** normally a fractional value in range
    - [1.0, 2.0), or
    - [0.0, 1.0)
  - **Exponent** ***E*** weights value by power of two
- **Encoding**
  - MSB s is sign bit ***s***
  - exponent field encodes ***E*** (but is not equal to E)
  - fraction field encodes ***M*** (but is not equal to M)

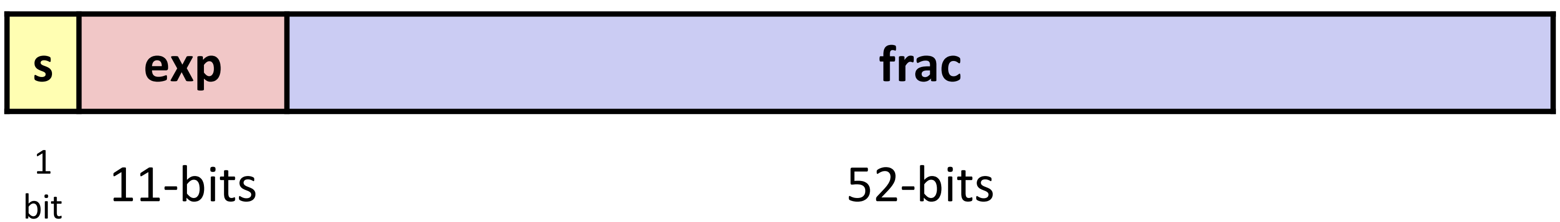
s	exponent	fraction

Precision options

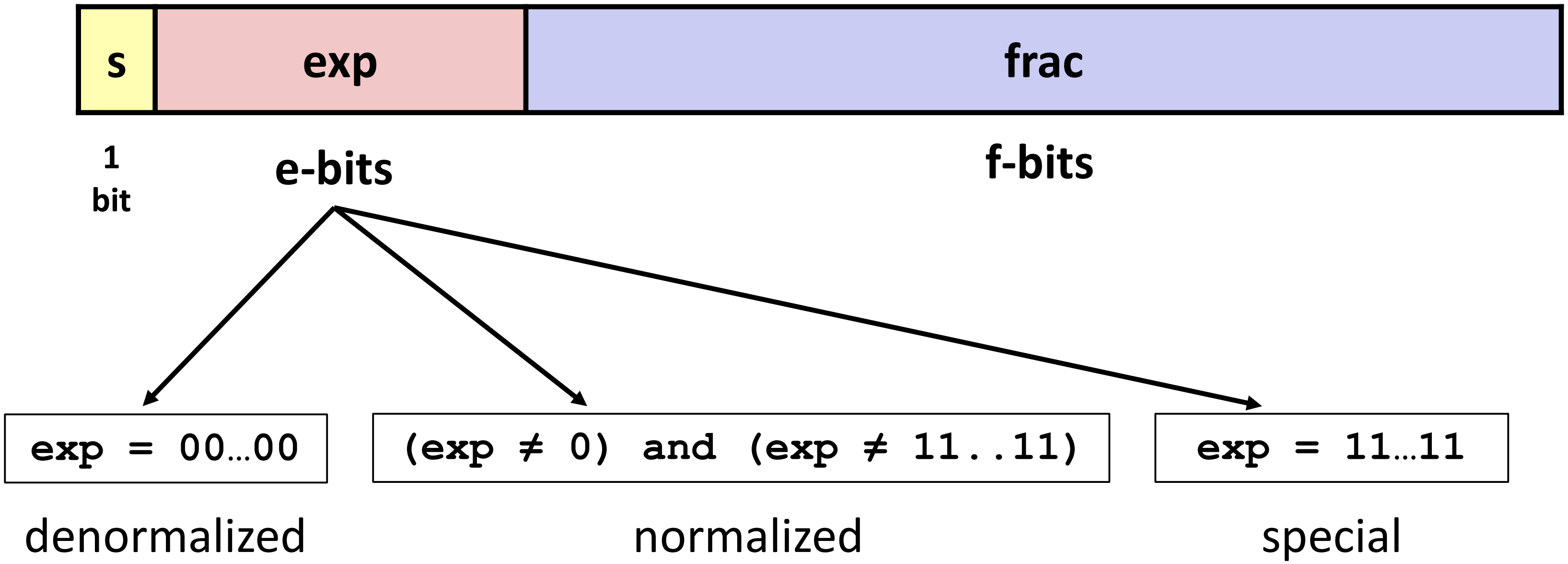
- **Single precision: 32 bits**
  ≈ 7 decimal digits, $10^{\pm 38}$



- **Double precision: 64 bits**
  ≈ 16 decimal digits, $10^{\pm 308}$



Three “kinds” of Floating Point Numbers



“Normalized” Values

`value = (–1)`^{s} `M 2`^{E}
`E = e – Bias`

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

- **Exponent coded as a *biased* value: *E = e – Bias***
  - *e:* unsigned value of **exp** field
  - *Bias* = 2^{k-1} - 1, where *k* is number of exponent bits
    - **Single precision:** *Bias* = 127 (e: 1…254, E: -126…127)
    - **Double precision:** *Bias* = 1023 (e: 1…2046, E: -1022…1023)

- **Significand (mantissa) coded with implied leading 1:** *M* = **1.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”

Normalized Encoding Example (single prec.)

value = $(-1)^s$ M 2^E
E = e – Bias

- **Value:** `float F = 12345.0;`
  - $12345_{10} = 11000000111001_2$
  - $12345.0_{10} = \mathbf{1.1000000111001_2 * 2^{13}}$

- **Significand**
  - $M = 1.\underline{1000000111001}_2$
  - $\mathrm{frac} = \mathbf{10000001110010000000000_2}$ (23 bits)

- **Exponent**
  - $E = 13$
  - $Bias = 127$
  - $e = 140 = \mathbf{10001100_2}$ (8 bits)

- **Result:**

s	exp	frac
0	10001100	10000001110010000000000

"Denormalized" Values

$\texttt{value} = (-1)^{s}\, M\, 2^{E}$
$E = 1 - \text{Bias}$

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

“Special” Values

- **When:** **`exp = 111…1`**

- **Cases**
  - **`exp`** = **`111…1`**, **`frac`** = **`000…0`**
    - Represents value ∞ (infinity)
    - Operation that overflows
    - Both positive and negative
    - e.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
  - **`exp`** = **`111…1`**, **`frac`** $\neq$ **`000…0`**
    - Not-a-Number (NaN)
    - Represents case when no numeric value can be determined
    - e.g., sqrt(–1), $\infty - \infty$, $\infty \times 0$

C float Decoding Example

$$value = (-1)^s\ M\ 2^E$$
$$E = e - Bias$$
$$Bias = 2^{k-1} - 1 = 127$$

`float f = 0xC0A00000`

binary: `1100 0000 1010 0000 0000 0000 0000 0000`

1	1000 0001	010 0000 0000 0000 0000 0000
1-bit	8-bits	23-bits

S = 1 → negative number

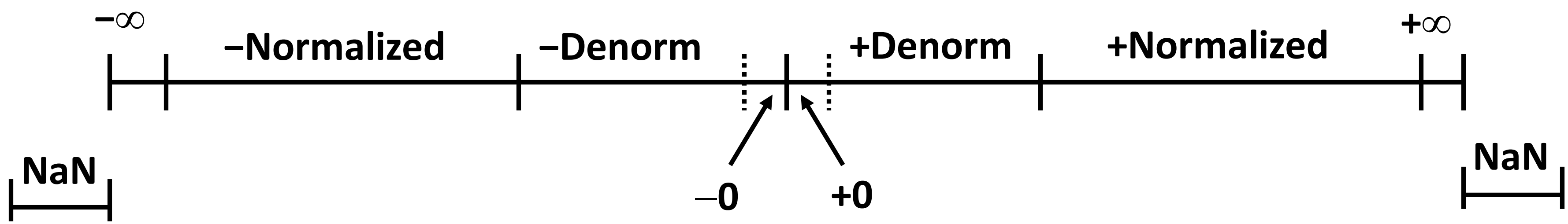
E = 129-127 = 2

$M = 1.010\ 0000\ 0000\ 0000\ 0000_2$

$= 1 + \frac{1}{4} = 1.25_{10}$

$value = (-1)^s\ M\ 2^E = (-1)^1 * 1.25 * 2^2 = -5.0$

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

s	exp	frac
1-bit	4-bits	3-bits

- **8-bit Floating Point Representation**
  - the sign bit is in the most significant bit
  - the next 4 bits are the **`exp`**, with a bias of 7
  - the last 3 bits are the **`frac`**

- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

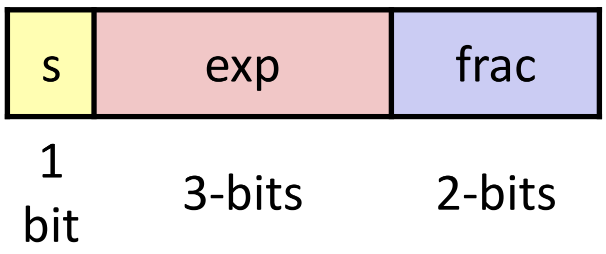
Dynamic Range (8-bit FP numbers)

value = $(-1)^s$ M 2^E
Norm: E = e − Bias
Denorm: E = 1 − Bias

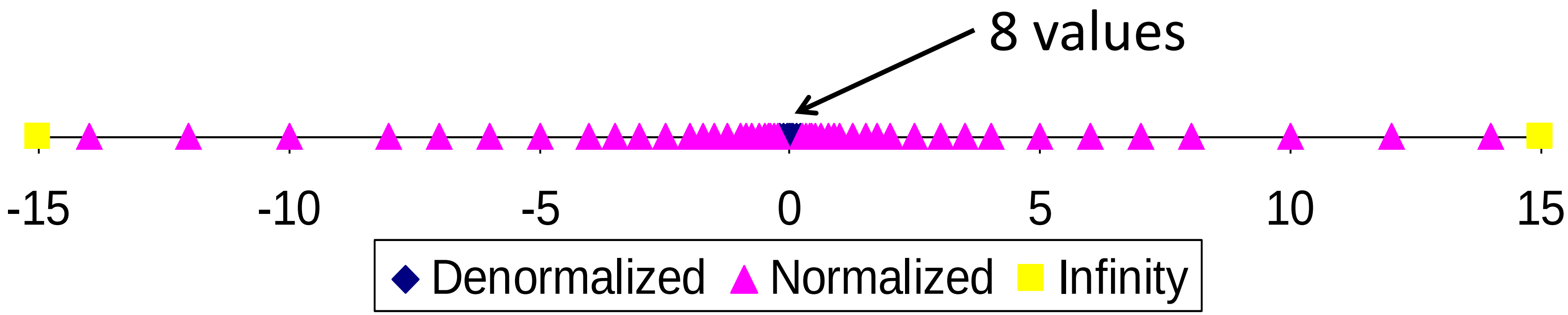
Bias=7	S	exp	frac	E	Value	Explanation
Denormalized Numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8 * 1/64 = **1/512**	Closest to zero
	0	0000	010	-6	2/8 * 1/64 = **2/512**	$(-1)^0 * (0+\frac{1}{4}) * 2^{-6}$
					....	
	0	0000	110	-6	6/8 * 1/64 = **6/512**	$(-1)^0 * (0+\frac{3}{4}) * 2^{-6}$
	0	0000	111	-6	7/8 * 1/64 = **7/512**	Largest denormalized
Normalized Numbers	0	0001	000	-6	1 * 1/64 = **8/512**	Smallest normalized
	0	0001	001	-6	9/8 * 1/64 = **9/512**	$(-1)^0 * (1+\frac{1}{8}) * 2^{-6}$
					....	
	0	0110	110	-1	14/8 * 1/2 = **14/16**	$(-1)^0 * (1+\frac{3}{4}) * 2^{-1}$
	0	0110	111	-1	15/8 * 1/2 = **15/16**	Closest to one below
	0	0111	000	0	1 * 1 = **1**	$(-1)^0 * (1+0) * 2^0$
	0	0111	001	0	9/8 * 1 = **9/8**	Closest to one above
	0	0111	010	0	10/8 * 1 = **10/8**	$(-1)^0 * (1+\frac{1}{4}) * 2^0$
					....	
	0	1110	110	7	14/8 * 128 = **224**	$(-1)^0 * (1+\frac{3}{4}) * 2^7$
	0	1110	111	7	15/8 * 128 = **240**	Largest normalized
	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

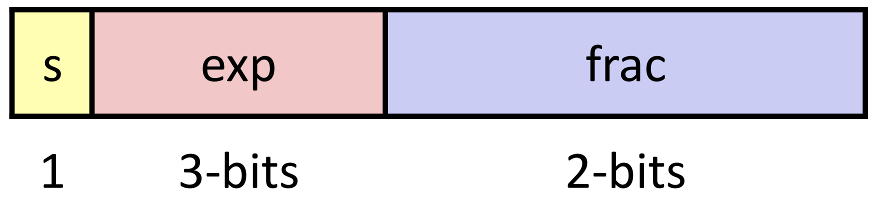


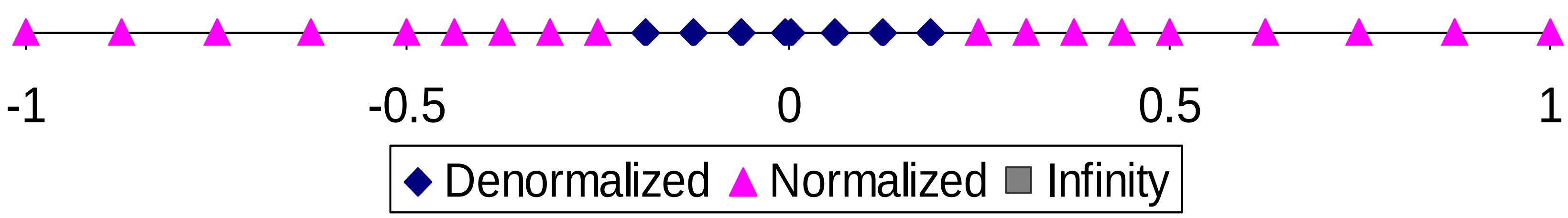
- **Notice how the distribution gets denser toward zero.**



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 the IEEE 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

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

- $x *^f y = \mathrm{Round}(x * y)$

- **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 Modes

VALUE	Towards Zero	Round down (-∞)	Round up (+∞)	Nearest Even (default)
`1.40`	1	1	2	1
`1.60`	1	1	2	2
`1.50`	1	1	2	2
`2.50`	2	2	3	2
`-1.5`	-1	-2	-1	-2

Closer Look at Round-To-Nearest-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
- Examples: round to nearest hundredth (2 digits after the decimal point)

Value	Action	Rounded Value
`7.8949999`	Less than half way	`7.89`
`7.8950001`	Greater than half way	`7.90`
`7.8950000`	Half way—round up	`7.90`
`7.8850000`	Half way—round down	`7.88`

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 after the binary point)

Value	Binary	Rounded	Action	Rounded Value
$2\frac{3}{32}$	10.00011_2	10.00_2	$<\frac{1}{2} \rightarrow$ down	2
$2\frac{3}{16}$	10.0011_2	10.01_2	$>\frac{1}{2} \rightarrow$ up	$2\frac{1}{4}$
$2\frac{7}{8}$	10.111_2	11.00_2	$=\frac{1}{2} \rightarrow$ up	3
$2\frac{5}{8}$	10.101_2	10.10_2	$=\frac{1}{2} \rightarrow$ down	$2\frac{1}{2}$

Floating Point Addition

$$(-1)^s\, M\ 2^E = (-1)^{s1}\, M1\ 2^{E1} + (-1)^{s2}\, M2\ 2^{E2}$$

1. **Compare exponents (*E1* and *E2*)**
   - shift the binary point of the number with smaller exponent to the left until the exponents are equal.
2. **Add the significands (*M1* and *M2*) with binary points aligned**
3. **Fix the result**
   a) If $M \geq 2$, shift M right and increment E
   b) If $M < 1$, shift M left k positions, decrement E by k
   c) Overflow if E out of range
   d) Round M to fit `frac` precision

Example: $1.010_2 * 2^2 + 1.110_2 * 2^3 = (0.1010_2 + 1.1100_2) * 2^3$
$= 10.0110_2 * 2^3 = 1.00110_2 * 2^4 = 1.010_2 * 2^4$

E1(2) < E2(3) → 0.1010 $\boxed{1}$

```
  0.1010
+ 1.1100
 10.0110
```

$\boxed{2}$

M(10.0110) ≥ 2 $\boxed{3.a}$ → M=round(1.00110) $\boxed{3.d}$ → M=1.010, E=4

Mathematical Properties of FP Addition

- **Closed under addition?** *Yes*
    - But may generate infinity or NaN
- **Commutative?** *Yes*
    - `3.14 + 1e10 = 1e10 + 3.14`
- **Associative?** *No*
    - Overflow and inexactness of rounding
    - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`
- **0 is additive identity?** *Yes*
- **Every element has additive inverse?** *Almost*
    - Yes, except for infinities & NaNs
- **Monotonicity**
    - $a \geq b \Rightarrow a+c \geq b+c$? *Almost*
        - Except for infinities & NaNs

Floating Point Multiplication

$$(-1)^s\ M\ 2^E = (-1)^{s1}\ M1\ 2^{E1} * (-1)^{s2}\ M2\ 2^{E2}$$

1. Exact Result

- Sign *s*: $s1$ ^ $s2$
- Significand *M*: $M1 * M2$
- Exponent *E*: $E1 + E2$

2. Fix the result

- If $M \geq 2$, shift *M* right and increment *E*
- Overflow if *E* out of range
- Round *M* to fit `frac` precision

Example: $1.010_2 * 2^2 * 1.110_2 * 2^3 = 10.001100_2 * 2^5$

$= 1.0001100_2 * 2^6 = 1.001_2 * 2^6$

Mathematical Properties of FP Mult

- **Closed under multiplication?** ***Yes***
  - But may generate infinity or NaN
- **Commutative?** ***Yes***
- **Associative?** ***No***
  - Possibility of overflow, inexactness of rounding
  - `(1e20*1e20)*1e-20=inf`, `1e20*(1e20*1e-20)=1e20`
- **1 is multiplicative identity?** ***Yes***
- **Multiplication distributes over addition?** ***No***
  - Possibility of overflow, inexactness of rounding
  - `1e20*(1e20-1e20)=0.0`, `(1e20*1e20)-(1e20*1e20)=NaN`
- **Monotonicity**
  - $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$? ***Almost***
    - 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`/`float`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to rounding mode

Floating Point C Puzzles

Initialization

```
int x = …;
float f = …;
double d = …;
```

Assume neither **`d`** nor **`f`** is NaN

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

Floating Point C Puzzles

Initialization

```
int x = …;
float f = …;
double d = …;
```

Assume neither **d** nor **f** is NaN

`x == (int)(float) x`	False
`x == (int)(double) x`	

Floating Point C Puzzles

Initialization

```
int x = …;
float f = …;
double d = …;
```

`x == (int)(float) x`	**False**
`x == (int)(double) x`	**True**
`f == (float)(double) f`	

Assume neither **`d`** nor **`f`** is NaN

Floating Point C Puzzles

Initialization

```
int x = …;
float f = …;
double d = …;
```

Assume neither **`d`** nor **`f`** is NaN

`x == (int)(float) x`	False
`x == (int)(double) x`	True
`f == (float)(double) f`	True
`d == (double)(float) d`	

Floating Point C Puzzles

Initialization

```
int x = …;

float f = …;

double d = …;
```

Assume neither **d** nor **f** is NaN

`x == (int)(float) x`	False
`x == (int)(double) x`	True
`f == (float)(double) f`	True
`d == (double)(float) d`	False
`f == -(-f);`	

Floating Point C Puzzles

Initialization

```
int x = …;

float f = …;

double d = …;
```

Assume neither **d** nor **f** is NaN

`x == (int)(float) x`	False
`x == (int)(double) x`	True
`f == (float)(double) f`	True
`d == (double)(float) d`	False
`f == -(-f);`	True
`2/3 == 2/3.0`	

Floating Point C Puzzles

Initialization

```
int x = …;
float f = …;
double d = …;
```

Assume neither **d** nor **f** is NaN

`x == (int)(float) x`	False
`x == (int)(double) x`	True
`f == (float)(double) f`	True
`d == (double)(float) d`	False
`f == -(-f);`	True
`2/3 == 2/3.0`	False
`d < 0.0 ⇒ ((d*2) < 0.0)`	

Floating Point C Puzzles

Initialization

```
int x = …;

float f = …;

double d = …;
```

Assume neither **d** nor **f** is NaN

`x == (int)(float) x`	False
`x == (int)(double) x`	True
`f == (float)(double) f`	True
`d == (double)(float) d`	False
`f == -(-f);`	True
`2/3 == 2/3.0`	False
`d < 0.0 ⇒ ((d*2) < 0.0)`	

Floating Point C Puzzles

Initialization

```
int x = …;
float f = …;
double d = …;
```

Assume neither **d** nor **f** is NaN

`x == (int)(float) x`	False
`x == (int)(double) x`	True
`f == (float)(double) f`	True
`d == (double)(float) d`	False
`f == -(-f);`	True
`2/3 == 2/3.0`	False
`d < 0.0 ⇒ ((d*2) < 0.0)`	True
`d > f ⇒ -f > -d`	

Floating Point C Puzzles

Initialization

```
int x = …;
float f = …;
double d = …;
```

`x == (int)(float) x`	False
`x == (int)(double) x`	True
`f == (float)(double) f`	True
`d == (double)(float) d`	False
`f == -(-f);`	True
`2/3 == 2/3.0`	False
`d < 0.0 ⇒ ((d*2) < 0.0)`	True
`d > f   ⇒ -f > -d`	True
`d*d >= 0.0`	

Assume neither `d` nor `f` is NaN

Floating Point C Puzzles

Initialization

```
int x = …;
float f = …;
double d = …;
```

Assume neither **d** nor **f** is NaN

`x == (int)(float) x`	False
`x == (int)(double) x`	True
`f == (float)(double) f`	True
`d == (double)(float) d`	False
`f == -(-f);`	True
`2/3 == 2/3.0`	False
`d < 0.0 ⇒ ((d*2) < 0.0)`	True
`d > f ⇒ -f > -d`	True
`d*d >= 0.0`	True
`(d+f)-d == f`	

Floating Point C Puzzles

Initialization

```
int x = …;
float f = …;
double d = …;
```

Assume neither **d** nor **f** is NaN

`x == (int)(float) x`	False
`x == (int)(double) x`	True
`f == (float)(double) f`	True
`d == (double)(float) d`	False
`f == -(-f);`	True
`2/3 == 2/3.0`	False
`d < 0.0 ⇒ ((d*2) < 0.0)`	True
`d > f ⇒ -f > -d`	True
`d*d >= 0.0`	True
`(d+f)-d == f`	False

Creating Floating Point Number

s	exp	frac
1 bit	4-bits	3-bits

- **Steps**
  1. Normalize to have leading 1
  2. Round to fit within fraction
  3. Postnormalize to deal with effects of rounding

- **Case Study: Convert numbers to tiny floating point format**

Value	Binary
128	10000000_2
15	00001111_2
33	00100001_2
35	00100011_2
63	00111111_2

Step 1: Normalize to have leading 1

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

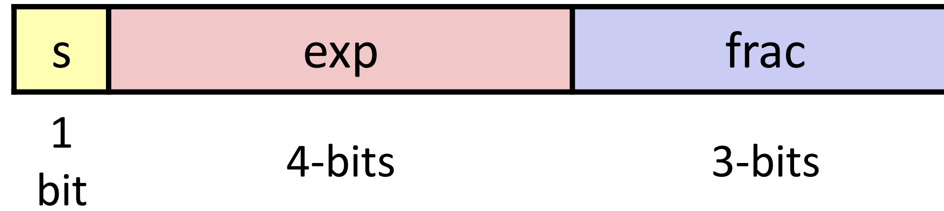
s	exp	frac
1 bit	4-bits	3-bits

Value	Binary	Fraction	Exponent
128	10000000_2	1.0000000_2	7
15	00001111_2	1.1110000_2	3
33	00100001_2	1.0000100_2	5
35	00100011_2	1.0001100_2	5
63	00111111_2	1.1111100_2	5

Step 2: Round to fit within fraction

Round-to-nearest-even

frac = 3 bits



Value	Fraction	Rounded	Exponent
128	1.0000000_2	1.000_2	7
15	1.1110000_2	1.111_2	3
33	1.0000100_2	1.000_2	5
35	1.0001100_2	1.001_2	5
63	1.1111100_2	10.000_2	5

Step 3: Postnormalize to deal with effects of rounding

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

Value	Rounded	Exponent	Adjusted	Adjusted Exponent	Result
128	1.000_2	7		7	$1.000_2 * 2^7 = 128$
15	1.111_2	3		3	$1.111_2 * 2^3 = \frac{15}{8} * 2^3 = 15$
33	1.000_2	5		5	$1.000_2 * 2^5 = 32$
35	1.001_2	5		5	$1.001_2 * 2^5 = \frac{9}{8} * 2^5 = 36$
63	10.000_2	5	1.000_2	6	$1.000_2 * 2^6 = 64$

Binary Representations

$$\texttt{value} = (-1)^s \, M \, 2^E$$

$$E = e - Bias$$

$$Bias = 2^{k-1} - 1 = 7$$

Unsigned Value	Mantissa	Exponent	Floating Point Value	Binary representation
128	1.000_2	7	128	0 1110 000
15	1.111_2	3	15	0 1010 111
33	1.000_2	5	32	0 1100 000
35	1.001_2	5	36	0 1100 001
63	1.000_2	6	64	0 1101 000

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

Interesting Numbers {single,double}

Description	*exp*	*frac*	*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}			