15-213

"The course that gives CMU its Zip!"

Floating Point
Sept 5, 2002

Topics

- IEEE Floating Point Standard
- Rounding
- Floating Point Operations
- Mathematical properties

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)(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`

IEEE Floating Point

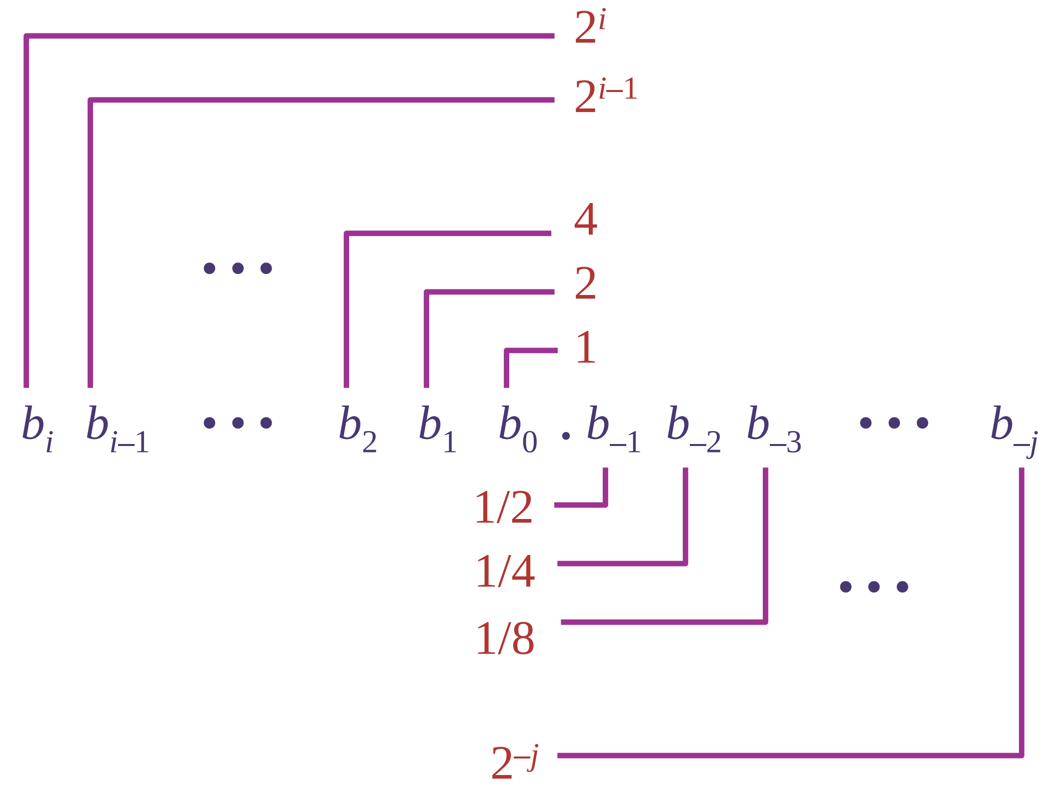
IEEE Standard 754

- 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 go fast
  - Numerical analysts predominated over hardware types in defining standard

Fractional Binary Numbers



Representation

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

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

Frac. Binary Number Examples

Value	Representation
5-3/4	`101.11`$_2$
2-7/8	`10.111`$_2$
63/64	`0.111111`$_2$

Observations

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form `0.111111…`$_2$ 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

- **Can only exactly represent numbers of the form $x/2^k$**
- **Other numbers have repeating bit representations**

Value	**Representation**
1/3	$\mathtt{0.0101010101[01]}\ldots_2$
1/5	$\mathtt{0.001100110011[0011]}\ldots_2$
1/10	$\mathtt{0.0001100110011[0011]}\ldots_2$

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

s	exp	frac

- MSB is sign bit
- `exp` field encodes *E*
- `frac` field encodes *M*

Floating Point Precisions

Encoding

`s`	`exp`	`frac`

- MSB is sign bit
- exp field encodes *E*
- `frac` field encodes *M*

Sizes

- Single precision: 8 exp bits, 23 `frac` bits
  - 32 bits total
- Double precision: 11 exp bits, 52 `frac` bits
  - 64 bits total
- Extended precision: 15 exp bits, 63 `frac` bits
  - Only found in Intel-compatible machines
  - Stored in 80 bits
    - » 1 bit wasted

"Normalized" Numeric Values

Condition

- `exp` ≠ `000…0` and `exp` ≠ `111…1`

Exponent coded as *biased* value

E = *Exp* – *Bias*

- *Exp* : unsigned value denoted by `exp`
- *Bias* : Bias value
  - » Single precision: 127 (*Exp*: 1…254, *E*: -126…127)
  - » Double precision: 1023 (*Exp*: 1…2046, *E*: -1022…1023)
  - » in general: *Bias* = 2^{e-1} - 1, where e is number of exponent bits

Significand coded with implied leading 1

M = $\mathtt{1.xxx…x}_2$

- `xxx…x`: bits of `frac`
- Minimum when `000…0` (*M* = 1.0)
- Maximum when `111…1` (*M* = 2.0 – ε)
- 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 = $1.\underline{1101101101101}_2$

`frac` = $\underline{\mathtt{1101101101101}}\mathtt{0000000000}_2$

Exponent

E = **13**

$Bias$ = **127**

Exp = **140** = $\mathbf{10001100}_2$

Floating Point Representation (Class 02):

Hex:	**4**	**6**	**6**	**D**	**B**	**4**	**0**	**0**
Binary:	**0100**	**0110**	**0110**	**1101**	**1011**	**0100**	**0000**	**0000**
140:	**100**	**0110**	**0**					
15213:			***1*110**	**1101**	**1011**	**01**		

Denormalized Values

Condition

- `exp` = `000…0`

Value

- Exponent value $E = -Bias + 1$
- Significand value $M =$ `0.xxx…x`$_2$
  - `xxx…x`: bits of `frac`

Cases

- `exp` = `000…0`, `frac` = `000…0`
  - Represents value 0
  - Note that have distinct values +0 and –0
- `exp` = `000…0`, `frac` ≠ `000…0`
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - "Gradual underflow"

Special Values

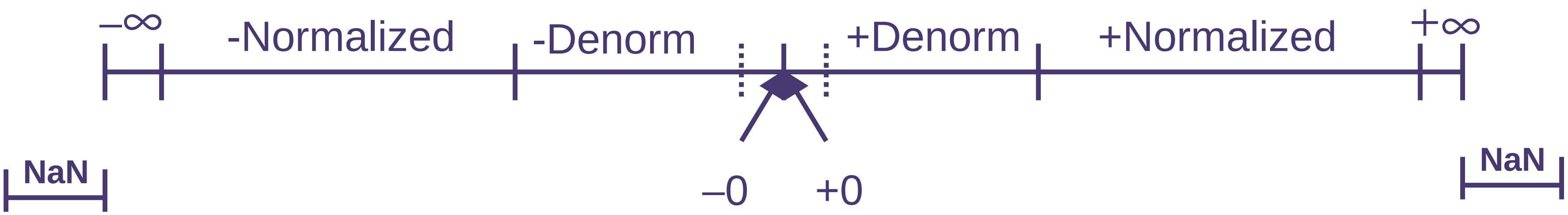
Condition

- `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 = +∞, 1.0/–0.0 = –∞
- `exp` = `111…1`, `frac` ≠ `000…0`
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ – ∞

Summary of Floating Point Real Number Encodings



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

7	6 3	2 0
`s`	`exp`	`frac`

Values Related to the Exponent

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	
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	
15	1111	n/a		(inf, NaN)

Dynamic Range

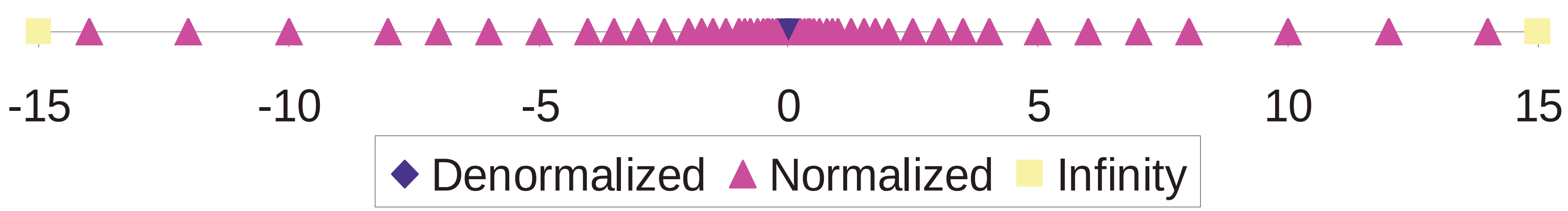
	s	exp	frac	*E*	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	← closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	← largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	← smallest norm
	0	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	← 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 3

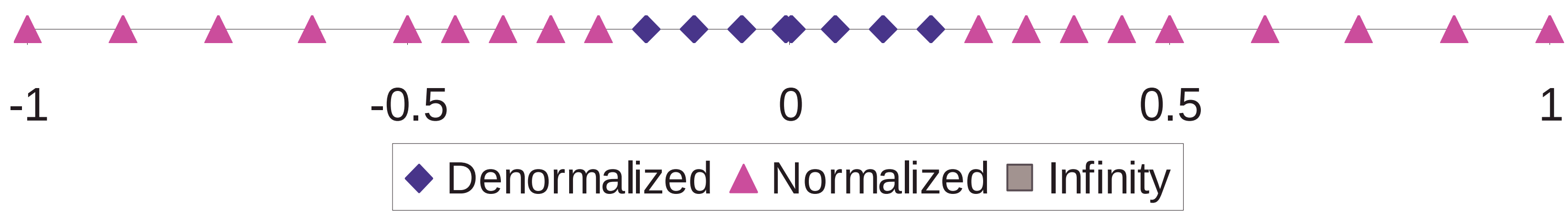
Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

6-bit IEEE-like format

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



Interesting Numbers

Description	exp	frac	Numeric Value
Zero	00…00	00…00	0.0
Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}} \text{ X } 2^{-\{126,1022\}}$
■ Single ≈ $1.4 \text{ X } 10^{-45}$			
■ Double ≈ $4.9 \text{ X } 10^{-324}$			
Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) \text{ X } 2^{-\{126,1022\}}$
■ Single ≈ $1.18 \text{ X } 10^{-38}$			
■ Double ≈ $2.2 \text{ X } 10^{-308}$			
Smallest Pos. Normalized	00…01	00…00	$1.0 \text{ X } 2^{-\{126,1022\}}$
■ Just larger than largest denormalized			
One	01…11	00…00	1.0
Largest Normalized	11…10	11…11	$(2.0 - \varepsilon) \text{ X } 2^{\{127,1023\}}$
■ Single ≈ $3.4 \text{ X } 10^{38}$			
■ Double ≈ $1.8 \text{ X } 10^{308}$			

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

Floating Point Operations

Conceptual View

- First compute exact result
- Make it fit into desired precision
  - Possibly overflow if exponent too large
  - Possibly round to fit into `frac`

Rounding Modes (illustrate with $ rounding)

	$1.40	$1.60	$1.50	$2.50	–$1.50
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

Note:

1. Round down: rounded result is close to but no greater than true result.
2. Round up: rounded result is close to but no less than true result.

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

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 = $\texttt{100...}_2$

Examples

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

Value	Binary	Rounded	Action	Rounded Value
2 3/32	$\texttt{10.00011}_2$	$\texttt{10.00}_2$	(<1/2—down)	2
2 3/16	$\texttt{10.00110}_2$	$\texttt{10.01}_2$	(>1/2—up)	2 1/4
2 7/8	$\texttt{10.11100}_2$	$\texttt{11.00}_2$	(1/2—up)	3
2 5/8	$\texttt{10.10100}_2$	$\texttt{10.10}_2$	(1/2—down)	2 1/2

FP Multiplication

Operands

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

Exact Result

$(-1)^{s}\ M\ \ 2^{E}$

- **Sign** *s***:** *s1* ^ *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**

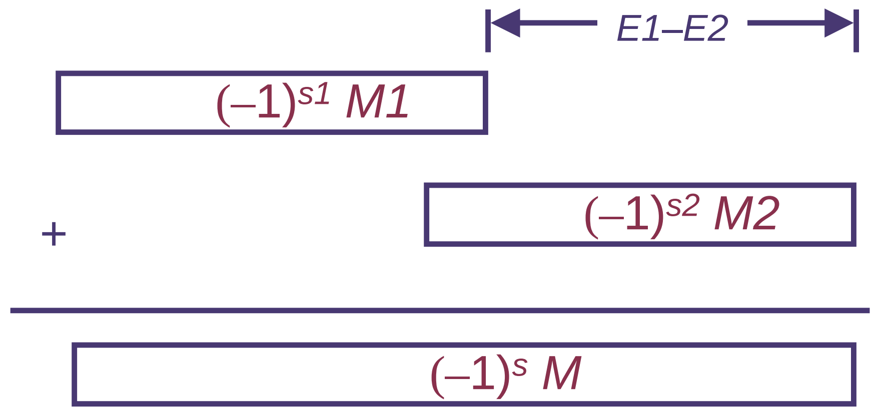
FP Addition

Operands

$(-1)^{s1}\ M1\ \ 2^{E1}$

$(-1)^{s2}\ M2\ \ 2^{E2}$

- **Assume** $E1 > E2$



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**

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
- **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 infinities & NaNs

Math. Properties of FP Mult

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 infinities & NaNs

Floating Point in C

C Guarantees Two Levels

`float`	single precision
`double`	double precision

Conversions

- Casting between `int`, `float`, and `double` changes numeric values
- `Double` or `float` to `int`
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range
    - » Generally saturates to TMin or TMax
- `int` to `double`
  - Exact conversion, as long as int has ≤ 53 bit word size
- `int` to `float`
  - Will round according to rounding mode

Answers to Floating Point Puzzles

```
int x = …;

float f = …;

double d = …;
```

Assume neither
d nor f is NAN

• `x == (int)(float) x`	No: 24 bit significand
• `x == (int)(double) x`	Yes: 53 bit significand
• `f == (float)(double) f`	Yes: increases precision
• `d == (float) d`	No: loses precision
• `f == -(-f);`	Yes: Just change sign bit
• `2/3 == 2/3.0`	No: 2/3 == 0
• `d < 0.0` ⇒ `((d*2) < 0.0)`	Yes!
• `d > f` ⇒ `-f > -d`	Yes!
• `d * d >= 0.0`	Yes!
• `(d+f)-d == f`	No: Not associative

Ariane 5

- Exploded 37 seconds after liftoff
- Cargo worth $500 million

Why

- Computed horizontal velocity as floating point number
- Converted to 16-bit integer
- Worked OK for Ariane 4
- Overflowed for Ariane 5
  - Used same software

Summary

IEEE Floating Point Has Clear Mathematical Properties

- Represents numbers of form $M \times 2^{E}$
- 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