CS 105

"Tour of the Black Holes of Computing!"

Floating Point

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 it is not true, ideally with an example

```
int x = foo();
float f = bar();
double d = baz();
```

Assume neither `d` nor `f` is NaN

Assume a 32-bit machine

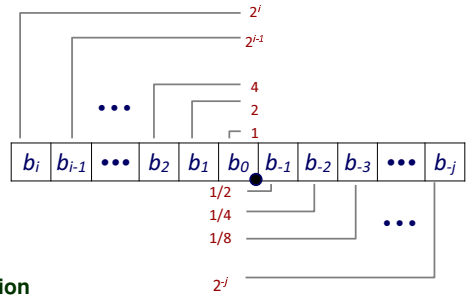
- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f)`
- `2/3 == 2/3.0`
- `d < 0.0` ⇒ `((d*2) < 0.0)`
- `d > f` ⇒ `-f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Fractional binary numbers

What is 1011.101_2?

- 101_2 = 5/8
- 1011_2 = 11

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: Examples

Value	Representation
5 3/4	101.11_2
2 7/8	10.111_2
1 7/16	1.0111_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
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - 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	Decimal Representation
1/3	$0.0101010101[01]\ldots_2$	0.333333333…
1/5	$0.001100110011[0011]\ldots_2$	0.200000000…
1/10	$0.0001100110011[0011]\ldots_2$	0.100000000…

Limitation #2

- Just one setting of binary point within the *w* bits
  - Limited range of numbers (very small values? very large?)

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

Floating-Point Representation

Numerical Form

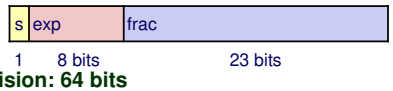
- $-1^s\ M\ 2^E$
  - Sign bit *s* determines whether number is negative or positive (negative zero representable)
  - Significand *M* normally a fractional value in range [1.0, 2.0).
  - Exponent *E* weights value by a power of two

Encoding

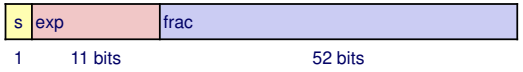
- MSB is sign bit
- `exp` field encodes *E* (emphasis on "encodes")
- `frac` field encodes *M* (likewise)

Precision Options (Not to Scale)

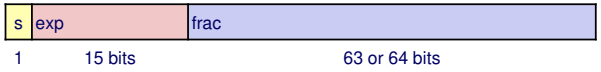
Single precision: 32 bits



Double precision: 64 bits



Extended precision: 80 bits (Intel only)



"Normalized" Values

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

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: 127 (Exp: 1…254, E: -126…127)
  - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)

Significand 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

Value

`float f = 15213.0;`

- $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$

Significand

M = 1.1101101101101_2

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

Exponent

E = 13

Bias = 127

Exp = 140 = 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:				1110	1101	1011	01	

Denormalized Values

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

Condition: exp = 000…0

Exponent value: E = 1 – Bias (instead of E = 0 – *Bias*)

Significand coded with implied leading 0: M = $0.xxx…x_2$

- `xxx…x`: bits of `frac`

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

Condition: `exp` = `111…1`

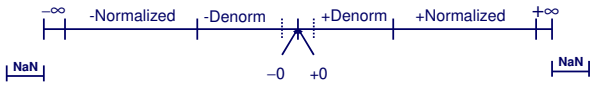
Case: `exp` = `111…1`, `frac` = `000…0`

- 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

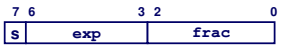


Tiny Floating-Point Example

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



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

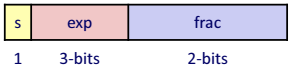
$v = (-1)^s M 2^E$
n: E = Exp – Bias
d: E = 1 – Bias

	s	exp	frac	E	Value	
	0	0000	000	−6	0	
	0	0000	001	−6	1/8*1/64 = 1/512	← closest to zero
Denormalized numbers	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	

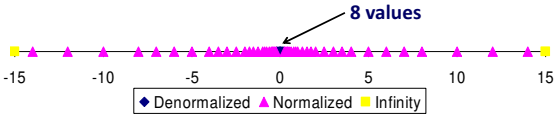
Distribution of Values

6-bit IEEE-like format

- e = 3 exponent bits
- f = 2 fraction bits
- Bias is $2^{3-1}-1 = 3$



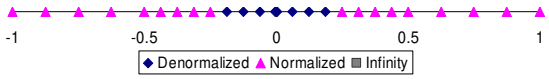
Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

6-bit IEEE-like format

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



Interesting Numbers

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 (float) ≈ 1.4×10^{-45}			
■ Double ≈ 4.9×10^{-324}			
Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
■ Single (float) ≈ 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 (float) ≈ 3.4×10^{38}			
■ Double ≈ 1.8×10^{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
  - Denormalized vs. normalized
  - Normalized vs. infinity

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 `frac`

Rounding

Rounding Modes (illustrated 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
  - Need randomness

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
- Halfway when bits to right of rounding position = $100\ldots_2$

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

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

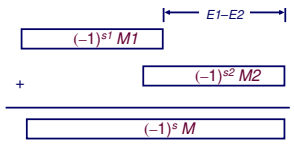
FP Addition

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
  - `(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 ≥ b ⇒ a+c ≥ b+c? *Almost*
  - Except for infinities & NaNs

Mathematical 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
  - Ex: `(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

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