Floating Point

Spring, 2019

Hyoung-Kee Choi

This Powerpoint slides are modified from its original version available at http://www.cs.cmu.edu/afs/cs/academic/class/15213-s09/www/lectures/ppt-sources/

Last Time: Integers

- Representation: **unsigned** and **signed**
- Conversion, casting
  - Bit representation maintained but reinterpreted
- Expanding, truncating
  - Truncating is $\mathbf{mod}\ \mathbf{2}^{w}$
- Addition, negation, multiplication, shifting
  - Operations are $\mathbf{mod}\ \mathbf{2}^{w}$

- "Ring" properties hold
  - Associative, commutative, distributive, additive 0 and inverse
- Ordering properties do **not** hold
  - $u > 0 \Rightarrow u + v > v$
  - $u > 0, v > 0 \Rightarrow u \cdot v > 0$

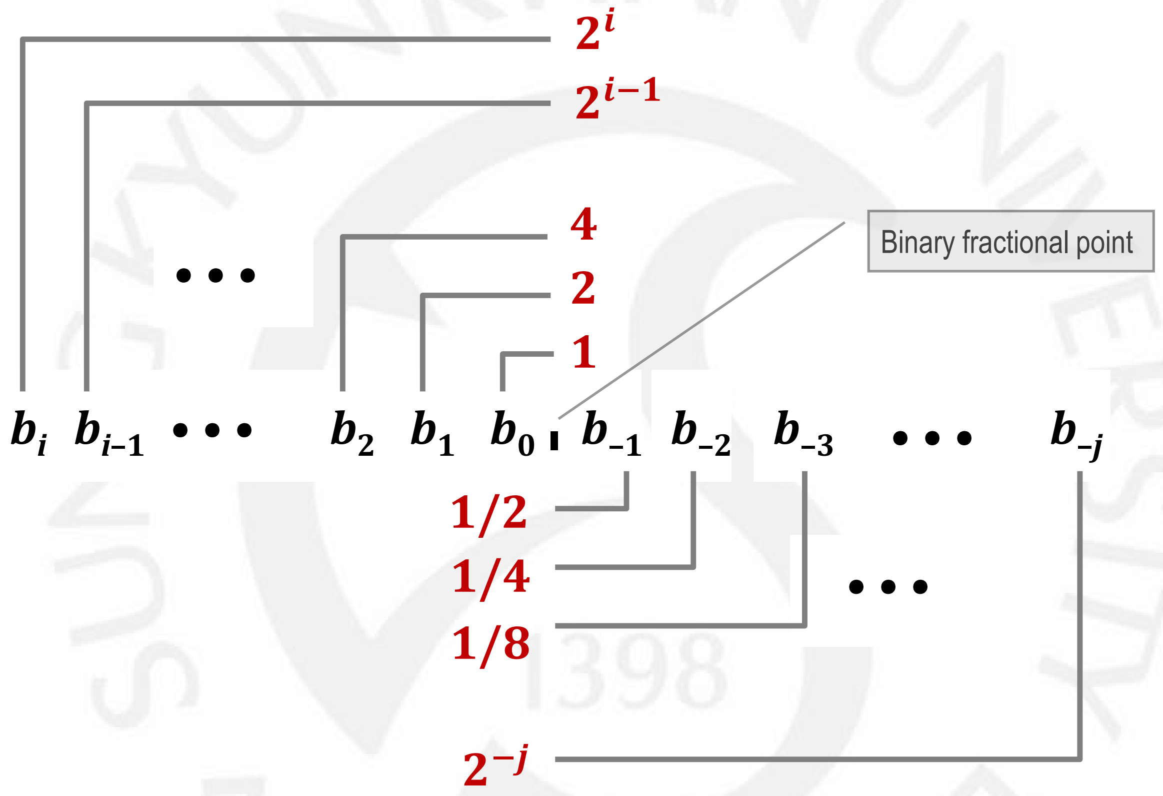
Today: Floating Point

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

Fractional Binary Number

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 $0.111111\ldots_2$ just below 1.0
    - $\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{2^i}+\cdots\rightarrow 1.0$
    - Use notation $1.0-\varepsilon$

Representable Numbers

- Limitation #1
  - Can only exactly represent numbers of the form $\frac{x}{2^k}$
  - Other rational numbers have repeating bit representations

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 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 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:

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

  - 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 encodes **E** (but is NOT equal to **E**)
  - **frac** field encodes **M** (but is NOT equal to **M**)

s	exp	frac

Precisions

- Single precision: 32 bits

s	exp	frac
1	8	23

- Double precision: 64 bits

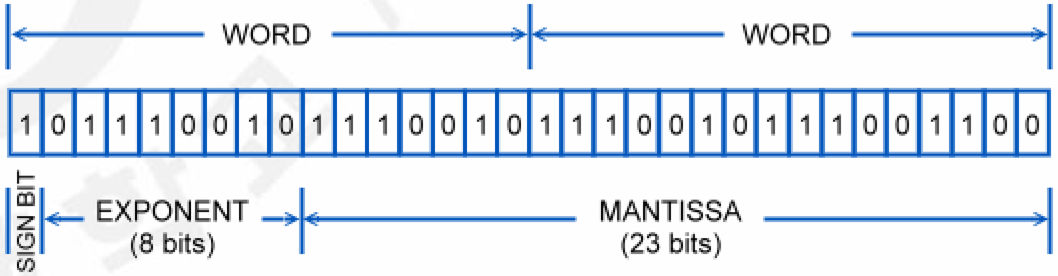
s	exp	frac
1	11	52

- Extended precision: 80 bits (Intel only)

s	exp	frac
1	15	64

Normalized Values

- Condition: **exp** ≠ **000…0** and **exp** ≠ **111…1**
- Exponent coded as biased value: **E = Exp – Bias**
  - **Exp**: unsigned value of **exp**
  - **Bias** = $2^{e-1} - 1$, where **e** 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: $\mathbf{M = 1.xxx{\ldots}x_2}$
  - **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**
  - $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \text{ x } 2^{13}$
- Significand
  - **M** $= 1.1101101101101_2$
  - **frac** $= 11011011011010000000000_2$
- Exponent
  - **E = 13**
  - **Bias = 127**
  - **Exp = 140 =** 10001100_2
- Result

s	exp	frac
0	10001100	11011011011010000000000

Denormalized Values

- Condition: **exp = 000…0**
- Exponent value: **E = –Bias + 1**
  - Instead of **E = 0 – Bias**
  - -126 (32) or -1022 (64)
- Significand coded with implied leading 0: **M = 0.xxx…x**$_2$
  - **xxx…x**: bits of **frac**

- $(-1)^s \times M \times 2^E$
- Cases
  - **exp = 000…0, frac = 000…0**
    - Represents value 0
    - Note distinct values: **+0** and **–0** (why?)
  - **exp = 000…0, frac ≠ 000…0**
    - Numbers very close to **0.0**
    - Lose precision as get smaller
    - Equi-spaced

Special Values

- Condition: $\mathbf{exp = 111...1}$
- Case: $\mathbf{exp = 111...1}$, $\mathbf{frac = 000...0}$ **[부정]**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., $\mathbf{1.0/0.0 = -1.0/-0.0 = +\infty}$, $\mathbf{1.0/-0.0 = -\infty}$
- Case: $\mathbf{exp = 111...1}$, $\mathbf{frac \neq 000...0}$ **[불능]**
  - Not-a-Number (**NaN**)
  - Represents case when no numeric value can be determined
  - E.g., $\mathbf{sqrt(-1)}$, $\infty - \infty$, $\mathbf{+\infty \times 0}$

Visualization

	$-\infty$	-normalized	-denorm			+denorm	+normalized	$+\infty$	
NaN				-0	$+0$				NaN

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	4	3

- 8-bit Floating Point Representation
  - Sign bit is in the most significant bit
  - Next four bits are the exponent, with a bias of 7.
  - Last three bits are **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	denom
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 or NaN	

Dynamic Range (Positive Only)

	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

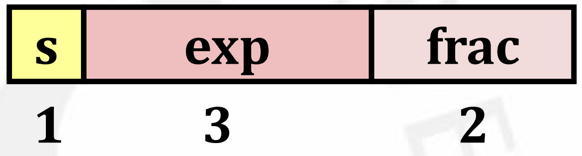
- 6-bit IEEE-like format
  - $e = 3$ exponent bits
  - $f = 2$ fraction bits
  - Bias is $2^{3-1}-1 = 3$

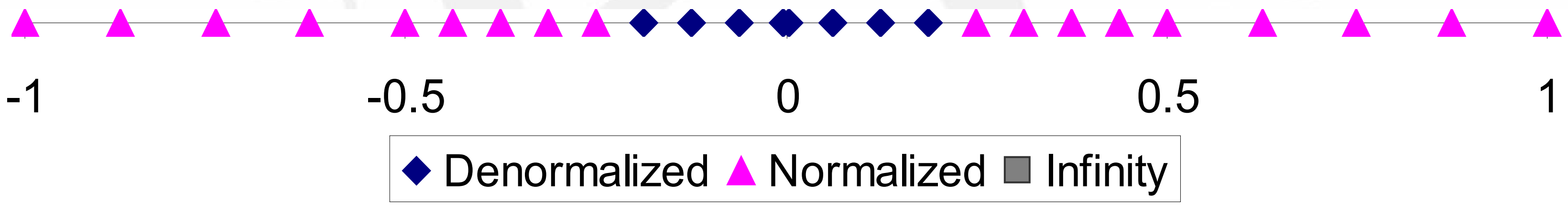
s	exp	frac
1	3	2

- 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 $2^{3-1} - 1 = 3$





Do It Yourself

- Convert $\mathbf{10.4_{10}}$ to single precision floating point
- Recall that:

$$\mathbf{10.4_{10}\ is\ 1010.[0110]_2}$$

Solution to DIY

1. Normalize
   - $\mathbf{1010.0110_2 \times 2^0 = 1.0100110 \times 2^3}$
2. Determine sign bit
   - Positive, so $\mathbf{S = 0}$
3. Determine exponent
   - $\mathbf{2^3}$ so $\mathbf{3}$ + bias $\mathbf{(= 127) = 130 = 10000010_2}$
4. Determine Significand
   - Drop leading $\mathbf{1}$ of mantissa, expand to $\mathbf{23}$ bits = $\mathbf{01001100110011001100110}$

0	10000010	01001100110011001100110

Interesting Numbers

{single,double}

Description	exp	frac	Numerical	Approx. Value
Zero	00…00	00…00	0.0	
Smallest Positive Denormalized	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$	Single $\approx 1.4 \times 10^{-45}$ Double $\approx 4.9 \times 10^{-324}$
Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$	Single $\approx 1.18 \times 10^{-38}$ Double $\approx 2.2 \times 10^{-308}$
Smallest Positive 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 $\approx 3.4 \times 10^{38}$ Double $\approx 1.8 \times 10^{308}$

Special Properties of Encoding

- FP (Floating Point) zero same as integer zero
  - All bits are zero
- Can (Almost) use unsigned integer **comparison**
  - Must first compare sign bits
  - Must consider **-0 = 0**
  - **NaN**s problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denormalized 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

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

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

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

Four Modes of 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

- Round down
  - Rounded result is close to but no greater than true result
- Round up
  - Rounded result is close to but no less than true result
- What are the advantages of the modes?

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**
  - Example: Round to nearest hundredth

Exercise

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

- Round to nearest hundredth

Rounding Binary Numbers

- Binary Fractional Numbers
  - "Even" when least significant bit is 0
  - "Half way" when bits to right of rounding position = $\mathbf{100..._2}$
- 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	down	2
2 3/16	10.00110_2	10.01_2	up	2 1/4
2 7/8	10.11100_2	11.00_2	up	3
2 5/8	10.10100_2	10.10_2	down	2 1/2

Floating Point Multiplication

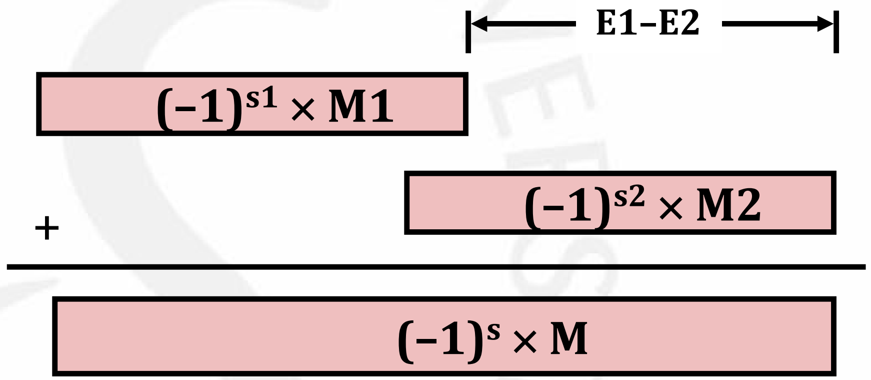
$$(-1)^{s1} \times M1 \times 2^{E1} \times \ (-1)^{s2} \times M2 \times \ 2^{E2}$$

- Exact Result: $(-1)^s \times M \times 2^E$
  - Sign s: $s1$ ^ $s2$
  - Significand M: $M1 \times M2$
  - Exponent E: $E1 + E2$
- Fixing
  - If $M \geq 2$, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit $\mathbf{frac}$ precision
- Implementation
  - Biggest chore is **MULTIPLYING SIGNIFICANDS**

Floating Point Addition

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

- Assume $\mathbf{E1} > \mathbf{E2}$
- Exact Result: $(-1)^s \times \mathbf{M} \times 2^{\mathbf{E}}$
  - Sign $\mathbf{s}$, significand $\mathbf{M}$:
    - Result of signed align & add
  - Exponent $\mathbf{E}$: $\mathbf{E1}$
- Fixing
  - If $\mathbf{M} \geq 2$, shift $\mathbf{M}$ right, increment $\mathbf{E}$
  - if $\mathbf{M} < 1$, shift $\mathbf{M}$ left $\mathbf{k}$ positions, decrement $\mathbf{E}$ by $\mathbf{k}$
  - Overflow if $\mathbf{E}$ out of range
  - Round $\mathbf{M}$ to fit $\mathbf{frac}$ 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**
    - Except for infinities & **NaN**s
- Monotonicity
  - $a \geq b \Rightarrow a+c \geq b+c$? **Almost**
    - Except for infinities & **NaN**s

Mathematical Properties of FP Multiplication

- 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
    - `(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 $\mathbf{T_{min}}$
- **int ⇒ double**
  - **Exact** conversion, as long as **int** has ≤ **53** bit word size
- **int ⇒ float**
  - Will round according to rounding 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

```
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`

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $\mathbf{M} \times \mathbf{2^E}$
- 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