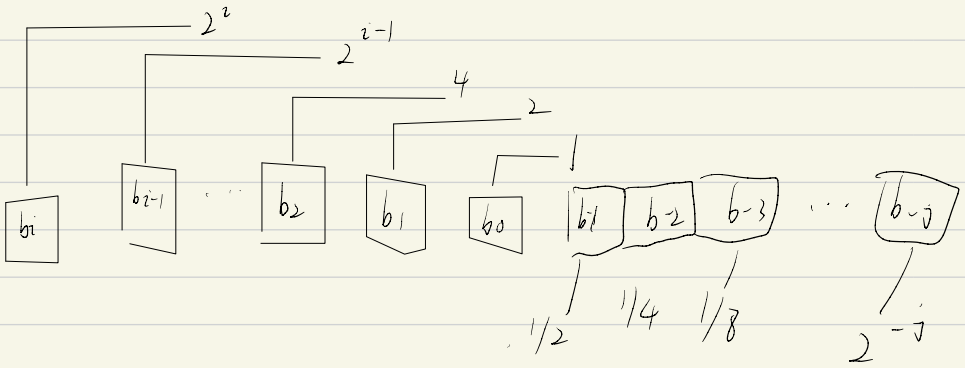
Floating Point:

Fractional binary numbers

(I) what is 1011.101_2 ?

Fractional Binary Numbers:



(i) Representation:

(ii) Bits to right of "binary point" represent fractional powers of 2,

(iii) Represents rational number:

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

e.g. Value	Representation
$5\ 3/4$	101.11_2
$2\ 7/8$	10.111_2
$1\ 7/16$	1.0111_2

Observations:

(i) Divide by 2 by shifting right (unsigned)

(ii) Multiply by 2 by shifting left.

(iii) (*) Numbers of form $0.111111\ldots_2$ are just below 1.0.

(*) $1/2 + 1/4 + 1/8 + \ldots + 1/2 \rightarrow 1.0$

(*) Use notation $1.0 - \varepsilon$

Limitation #1

Can only exactly represent numbers of the form $x/2^k$

Other rational numbers have repeating bit representations

Value	Representations
1/3	$0.0101\,0101\,0101\,[0101]\ldots_2$
1/5	$0.0011\,0011\,0011\,[0011]\ldots_2$
1/10	$0.0001\,1001\,1001\,[1001]\ldots_2$

Limitation #2.

Just one setting of binary point within the w bits.

Limited range of numbers (very small values? very large?)

floating point ← trade-off

- a large range of values
- much precision given the number of bits.

IEEE floating Point

IEE standard 754

Established in 1985 as uniform std. for floating point arithmetic.
. Before that, many idiosyncratic formats.
(peculiar or individual)

floating point Representation:

(I) Numerical Form:

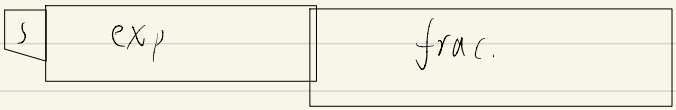
$$(-1)^s M \cdot 2^E$$

(Mantissa)

(i) Sign bit s determines whether number is negative or positive.
(ii) Significand M normally a fractional value in range [1.0, 2.0).
(iii) Exponent E weights value by power of two.

(II) Encoding

(i) MSB is sign bit s.
(ii) exp field encodes E (but is not equal to E)
(iii) frac field encodes M (but is not equal to M)



Single Precision (32 bits)	1-bit	8-bits	23-bits
Double Precision (64 bits)	1-bit	11-bits	52-bits,
Extended Precision 80 bits	1-bit	15-bits	63 or 64-bits,

"Normalized" Values $V = (-1)^S M 2^E$

(I) When : $exp \neq 000\ldots0$ and $exp \neq 111\ldots1$

(II) Exponent coded as a biased value: $E = Exp - Bias$.

(i) Exp: unsigned value of exp. field.

(ii) 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)

(III) Significand coded with implied leading 1: $M = 1.xxx\ldots x_2$

(i) xxx…x: bits of frac field. ($M = 1.0$)

(ii) Maximum when frac = 111…1 ($M = 2.0 - \varepsilon$)

Get extra leading bit for "free"

Value: float F = 15213.0

$15213_{10} = 111\,011\,011\,011\,01_2$

$= 1.11011\,011\,011\,01_2 \times 2^{13}$

Significant:

$M = 1.11\,011\,011\,011\,01_2$

frac = $11\,011\,011\,011\,0100\,0000\,0000_2$

throw this away

Exponent:

$E = 13$ Bias $= 127 \Rightarrow Exp = 140 = 10001100_2$

0	1000 1100	1101 1010 0000 0000 0000
s	exp	frac

$$0 \le Exp \le 2^8 - 1$$

$$0 \le Exp \le 255$$ shift

$$-127 \le E \le (255 - 127 = 128)$$

Denormalized Values

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

$E = 1 - Bias$

(I) Condition: exp = 000...0

(II) Exponent value: $E = 1 - Bias$ (instead of $E = 0 - Bias$)

(III) Significant coded with implied leading 0: $M = 0.XXX\ldots X_2$

(IV) Cases:

(i) exp = 000...0, frac = 000...0

- ★: Represents zero value
- ★: Note distinct values: +0 and −0 (why?)

(ii) exp = 000...0, frac ≠ 000...0

- ★: Numbers closet to 0.0
- ★: Equispaced.

Special Values

(I) Condition: exp = 111...1

(II) Case: exp = 111...1, frac = 000...0

(i) Represents value ∞ (infinity)

(iii) Operation that overflows

(iii) Both positive and negative

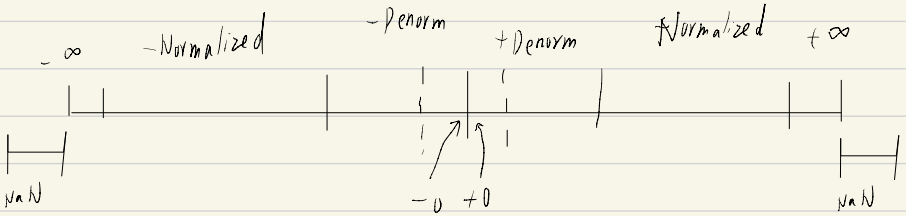
(iv) E.g. 1.0/0.0 = −1.0/0.0 = +∞, 1.0/−0.0 = −∞

(III) Case: exp = 111...1, frac ≠ 000...0

(i) Not-a-Number (NaN)

(ii) Represents case when no numeric value can be determined

(iii) E.g. sqrt(−1), ∞ − ∞, ∞ × 0

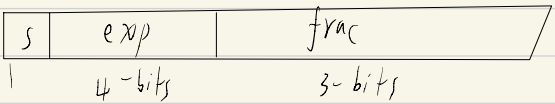


Today: Floating Point:

(I) BG: Fractional binary numbers
(II) IEE floating point std.: Definition.
(III) Example and properties
(IV) Rounding, addition, multiplication
(V) Floating point in C
(VI) Summary

e.g.

Tiny Floating Point Example



(I) 8 bits Floating Point Representation.

(i) the sign bit is in the most significant bit

(ii) the next four bits are the exponent, with a bias of 7

(iii) the last three bits are the frac

(II) Same general form as IEEE Format.

(i) normalized, denormalized

(ii) representation of 0, NaN, infinity

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.	closet 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	closest to 1 below
	0	0110	111	-1	15/8 * 1/2 = 15/16	
	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	

(+1)

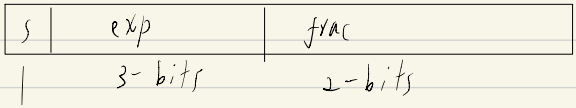
Distribution of Values.

(I) 6-bit IEEE-like format

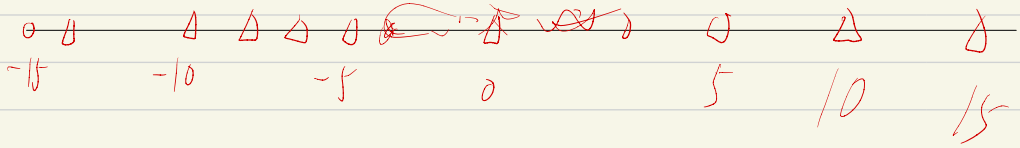
e = 3 exponent bits

f = 2 fraction bits

Bias: $2^{3-1}-1 = 3$.



(II) Notice how the distribution gets denser toward zero.



Special Properties of the IEEE Encoding:

(I) FP Zero Same as Integer Zero
  (i) All bits = 0

(II) Can (Almost) Use Unsigned Integer Comparison.
  (i) Must first compare sign bits
  (ii) Must consider $-0 = 0$.
  (iii) NaNs problematic
  Will be greater than any other values
  What should comparison yield.
  (iv) Otherwise OK
    Denorm vs. normalized.
    Normalized vs. infinity.

Floating Point Operations: Basic Idea.

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

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

[I] Basic idea:

(i) First compute exact result

(ii) Make it fit into desired precision.

* possibly overflow if exponent too large
* Possibly round to fit into frac

Rounding

Rounding modes (illustrate w/ $ 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

nearest even number

if equi. near...

Closer Look at Round-To-Even.

(I) Default Rounding Mode.

(i) Hard to get any other kind without dropping into assembly.

(ii) All others are statistically biased.

Sum of set of positive numbers will consistently be over- or under-estimated.

(II) Applying to other Decimal Places / Bit Operations

When exactly halfway between two possible values.

Round so that least significant digit is even.

e.g. round to the 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)
7.8950000	7.88	(Half way - round down)

Rounding Binary Numbers.

(2) Binary Fractional Numbers

(i) "Even" when least significant bit is 0

(ii) "Half way" when bits to right of rounding position = 100_2

(II) Examples

(i) Round to nearest $\frac{1}{4}$ (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	11.00_2	(1/2 – up)	3
2 5/8	10.10100_2	10.10_2	(1/2 – down)	2 1/2

FP Multiplication.

(I) $(-1)^{S1} M1 2^{E1} \times (-1)^{S2} M2 2^{E2}$

(II) Exact Result: $(-1)^{S} M 2^{E}$

Sign S: $S1 \wedge S2$

Significand M: $M1 \times M2$

Exponent E: $E1 + E2$.

(III) Fixing:

(i) If $M \geq 2$, shift M right, increment E

(ii) If E out of range, overflow

(iii) Round M to fit frac precision.

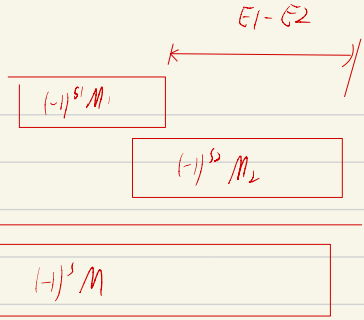
(IV) Implementation.

(i) Biggest chore is multiplying significands

Floating Point Addition:

$(-1)^{S1} M1 \; 2^{E1} + (-1)^{S2} M_2 \; 2^{E2}$

Assume $E1 > E2$



(I) Exact Result: $(-1)^S M \cdot 2^E$

(i) Sign S, significand M:

Result of signed align & add

Exponent E: $E1$

(II) Fixing:

(i) If $M \geq 2$, shift M right, increment E.

(ii) If $M < 1$, shift M left k positions, decrement E by k.

(iii) Overflow if E out of range

(iv) Round M to fit frac precision.

Mathematic Properties of FP Add

(I) Compare to those of Abelian Group.

(i) Closed under addition — Yes

But may generate infinity or NaN

(ii) Commutative? — Yes

(iv) Associative? — NO

- Overflow and inexactness of rounding.
- $3.14 + 1e10 - 1e10 = 0$, $3.14 + (1e10 - 1e10) = 3.14$

(v) 0 is additive identity? — Yes

(vi) Every element has additive inverse

Yes, but except for infinities & NaNs

(II) Monotonicity

(i) $a \geq b \Rightarrow a + c \geq b + c$ — Almost

Except for infinities & NaNs

Mathematical Properties of FP Mult?

extra properties?

Multiplication distributes over addition? No.

Possibility of overflow, inexactness of rounding.

✗ $1e20 * (1e20 - 1e20) = 0.0$, $1e20 * 1e20 - 1e20 * 1e20 = 0$

Monotonicity,

$a \geq b$ & $c > 0$ $\quad a * c \geq b * c$ Almost

Except for NaNs & infinities

Floating Point in C:

(I) C Guarantees two levels:

float — single precision

double — double precision

(II) Conversion / casting:

(i) Casting between int, float, and double changes bit representation.

(ii) double / float → int

truncate fractional part

like rounding toward zero.

Not defined when out of range or NaN; generally sets to TMin.

(iii) int → double

Exact conversion, as long as int has ≤ 53 bit word size.

(iv) int → float

will round according to rounding mode.

---

int x = ... float f = ... double d = ...

Assume neither d or f is NaN:

x == (int)(float) x	TRUE
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 (Monotonicity)
d*d >= 0.0 (TRUE)	(d+f) - d == f (FALSE)

Summary:

(i) IEEE Floating Point has clear mathematical properties

(ii) Represents numbers of form $M \times 2^E$

(iii) One can reason about operations independent of implementation

- ☆ As if computed with perfect precision and then rounded.

(iv) Not the same as real arithmetic

- (i) Violates assoc / distr.
- (ii) Makes life difficult for compilers & serious numerical applications programmers