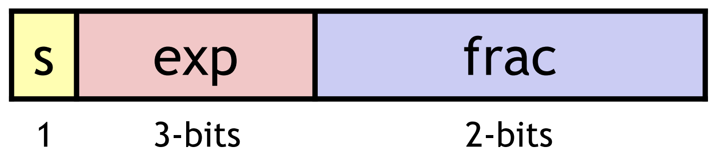
IEEE 754 Rules & Properties

Analysis of IEEE 754

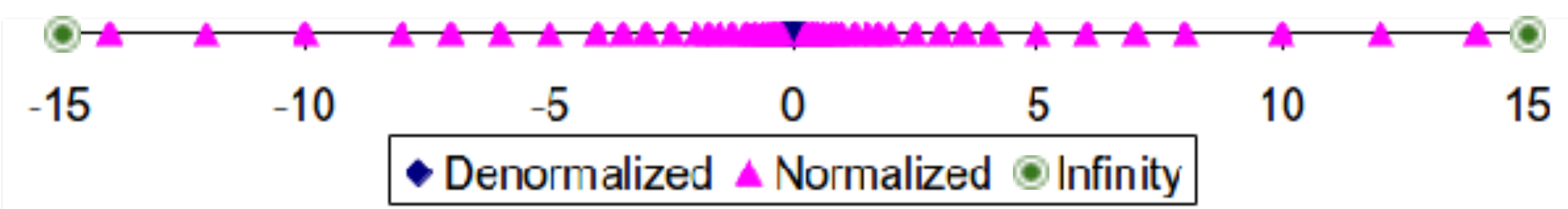
- As we saw last time, IEEE 754..
  - can represent numbers at wildly different magnitudes (limited by the length of the exponent)
  - provides the same relative accuracy at all magnitudes (limited by the length of the mantissa)
- There are some other nice properties as well related to rounding and arithmetic operations as we'll see today.
- Turns out there are some drawbacks as well.

Distribution of values

- Remember our 6-bit version of IEEE 754?

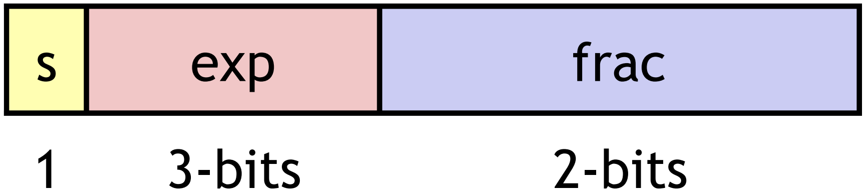


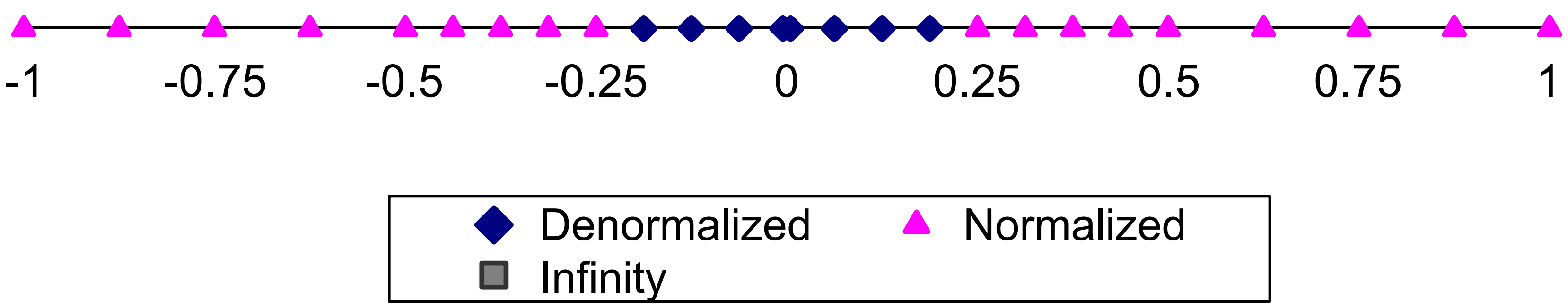
- The below graph plots values along a number line between negative and positive infinity.
- Notice how we lose precision as the whole numbers get larger.
- Why is that?



Distribution of values (close-up view)

- 6-bit IEEE-like format
  - e = 3 **exp** bits
  - f = 2 **frac** bits
  - Bias is 3





+ Special properties of IEEE encoding

- Floating point zero is all zeroes at the bit level.
  - This means zero is all 0’s.
- Can use unsigned integer comparison at the bit level, with a couple notable exceptions…
  - Must consider sign bit
  - Must consider positive and negative 0
  - NaN’s
    - Using unsigned comparison a Nan cannot be greater than any other value.
    - Bit-identical NaN values must not be considered equal.
- Otherwise proper ordering, even across types (ex. norm vs denorm)

Interpreting as unsigned bit patterns

- Lets convince ourselves. Pick two and test.
- Denorm **000011** and Norm **000101**
  - What are their decimal values with unsigned int interpretation?
- This is not an accident!
- Special case, the sign bit.
  - Can be overcome without too much trouble (out of scope for this course)

s	exp	frac
1	3-bits	2-bits

000000	010000	**100000**	110000
000001	010001	**100001**	110001
000010	010010	**100010**	110010
000011	010011	**100011**	110011
000100	010100	100100	110100
000101	010101	100101	110101
000110	010110	100110	110110
000111	010111	100111	110111
001000	011000	101000	111000
001001	011001	101001	111001
001010	011010	101010	111010
001011	011011	101011	111011
001100	**011100**	101100	**111100**
001101	**011101**	101101	**111101**
001110	**011110**	101110	**111110**
001111	**011111**	101111	**111111**

◉ Normalized ◉ Denormalized ◉ Special

Rounding

- When you do an operation on two floating point numbers such as multiplication or addition, no assurance that there are enough bits to hold result.
- We need a rounding strategy
  - $x +_f y = \text{Round}(x + y)$
  - $x *_f y = \text{Round}(x * y)$
- Basic idea
  - Compute exact result, make it fit into desired precision
  - Possibly overflow if exponent too large
  - Possibly round to fit into ***frac***

Rounding modes

- IEEE 754 rounding modes

	\$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
▪ *Round Nearest* (default)	\$1	\$2	\$2	\$2	–\$2

- IEEE 754 does *Rounding Nearest (Even)* rounding by default,
  - Special case: round to the 'nearest even' when you are exactly half-way between two possible rounded values.
  - All others rounding modes are statistically biased.
  - You can change mode, but you have to drop to assembly to do so.

+ ‘Round to nearest’ in decimal

- Applying to other decimal places / bit positions
- When exactly half-way between two possible values, *round so that least significant digit is even*
- E.g., round to nearest hundredth (2 digits right of decimal point)

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half-way - round up so that the LSD is even)
7.8850000	7.88	(Half-way - round down so that the LSD is even)

‘Round to nearest’ in binary

- Binary fractional numbers
  - “Half-way” when bits to right of rounding position = $100\ldots0_2$
  - “Even” when least significant bit is 0
- E.g., round to nearest 1/4 (2 bits right of binary point)

Value_{10}	**Value_2**	**Rounded_2**	**Action**	**Rounded_{10}**
2 3/32	10.00011	10.00	(less than 1/2)	2
2 3/16	10.00110	10.01	(greater than 1/2)	2 1/4
2 7/8	10.11100	11.0**0**	(1/2 round-up)	3
2 5/8	10.10100	10.1**0**	(1/2 round-down)	2 1/2

Properties of floating point addition

- Closed under addition? **Yes**
  - But may generate infinity or NaN
- Commutative? **Yes**
- Associative? **No**
  - Due to overflow and inexactness of rounding
  - (-1e20 + 1e20) + 3.14 == 3.14
  - -1e20 + (1e20 + 3.14) == 0.0
- 0 is additive identity? **Yes**
- Every element has additive inverse? **Almost**
  - Yes, except for infinities & NaNs
- Monotonicity **Almost**
  - $a \geq b \Rightarrow a + c \geq b + c$?
  - Except for infinities & NaNs

+ Properties of floating point multiplication

- Closed under multiplication? **Yes**
  - But may generate infinity or NaN
- Commutative? **Yes**
- Associative? **No**
  - Due to overflow and inexactness of rounding
  - (1e20 * 1e20) * 1e-20 = *inf*, 1e20 * (1e20 * 1e-20) = 1e20
- 1 is multiplicative identity? **Yes**
- Multiplication distributes over addition? **No**
  - Due to overflow and inexactness of rounding
  - 1e20 * (1e20 - 1e20) = 0.0, 1e20 * 1e20 - 1e20 * 1e20 = *NaN*
- Monotonicity **Almost**
  - $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$?
  - Except for infinities & NaNs

Remember this?

- We saw this on day one…
- **Example**: Is (x + y) + z = x + (y + z)?
  - for integral types? yes.
  - for floating point types?
    - (-1e20 + 1e20) + 3.14 == 3.14
    - -1e20 + (1e20 + 3.14) == 0.0
- Do you have any intuition as to why yet?

Floating point in C

- Coercion and casting
  - Coercion 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 → double**
    - Exact conversion, as int has <= 53 bits
  - **int → float**
    - Will round according to rounding mode, as int has >= 23 bits
- See *coercion_casting.c*

+ 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 == (double)(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`

- See *float_puzzles.c*

Summary

- IEEE Floating Point has clear mathematical properties
- 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
- Has improved the state of computing with floating point numbers tremendously and has received a number of impactful improvements since its introduction in the 80's!