Floating Point Numbers

CMPU 224 – Computer Organization
Jason Waterman

Fractional decimal numbers

- What is the representation for 123.456?

Fractional decimal numbers

- What is the representation for 123.456?

	1	2	3	.	4	5	6

Fractional decimal numbers

- What is the representation for 123.456?

	1	**2**	**3**	**.**	**4**	**5**	**6**
Weight	100	10	1	.			
Weight	10^2	10^1	10^0	.			
Value	1 * 100	2 * 10	3 * 1	.			

Fractional decimal numbers

- What is the representation for 123.456?

	1	**2**	**3**	**.**	**4**	**5**	**6**
Weight	100	10	1	.	1/10	1/100	1/1000
Weight	10^2	10^1	10^0	.			
Value	1 * 100	2 * 10	3 * 1	.			

Fractional decimal numbers

- What is the representation for 123.456?

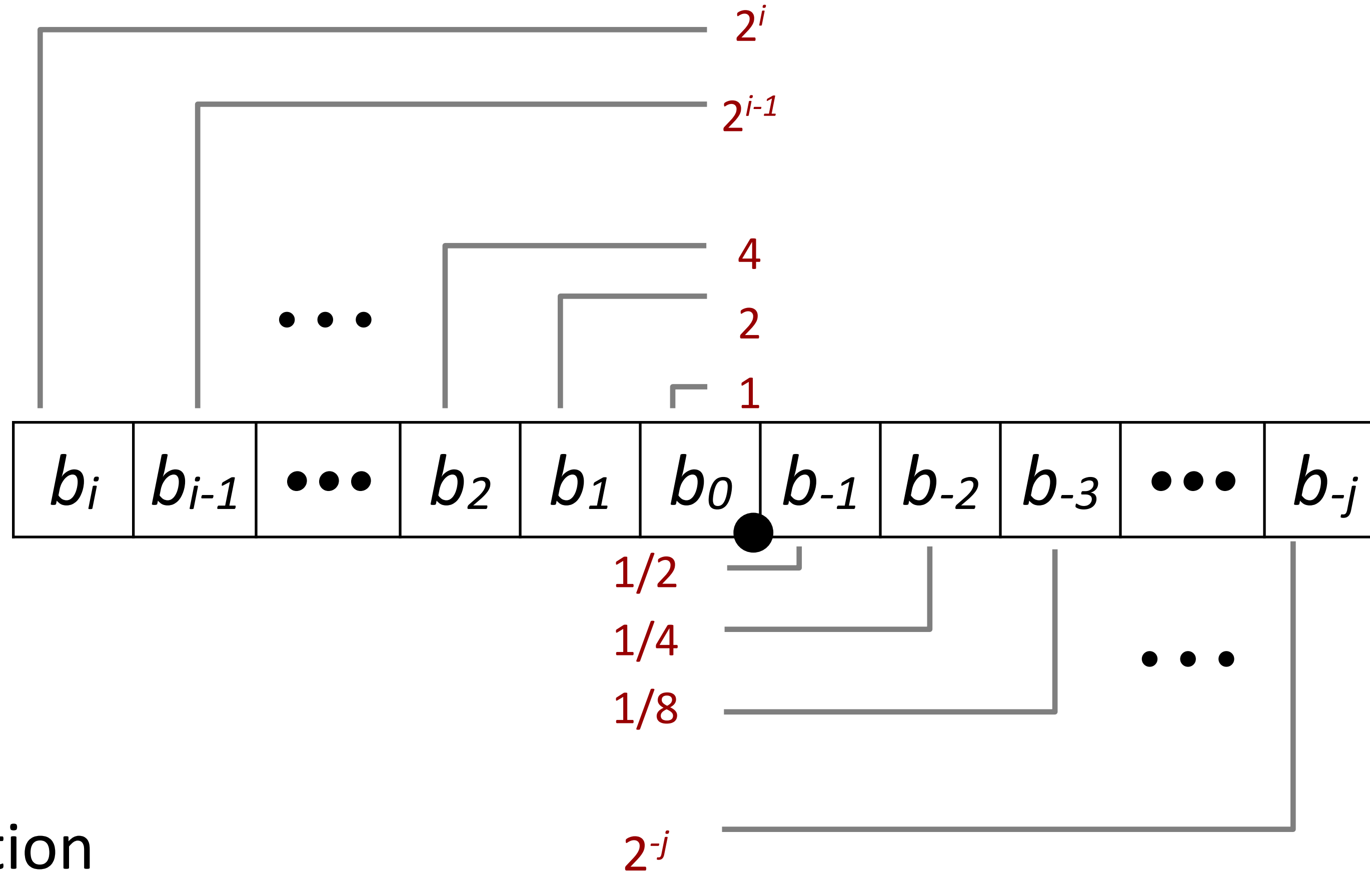
	1	**2**	**3**	**.**	**4**	**5**	**6**
Weight	100	10	1	.	1/10	1/100	1/1000
Weight	10^2	10^1	10^0	.	10^{-1}	10^{-2}	10^{-3}
Value	1 * 100	2 * 10	3 * 1	.			

Fractional decimal numbers

- What is the representation for 123.456?

	1	2	3	.	4	5	6
Weight	100	10	1	.	1/10	1/100	1/1000
Weight	10^2	10^1	10^0	.	10^{-1}	10^{-2}	10^{-3}
Value	1 * 100	2 * 10	3 * 1	.	4/10	5/100	6/1000

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 Number Examples

- Value — Representation

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
  - Multiply by 2 by shifting left
  - Numbers of the 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
    - 1/3 `0.0101010101[01]`$\ldots_2$
    - 1/5 `0.001100110011[0011]`$\ldots_2$
    - 1/10 `0.0001100110011[0011]`$\ldots_2$

- 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 fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

Scientific Notation

- Allows us to specify a number and where the decimal point goes
- Useful notation for very small and very large numbers
- $\pm m \times 10^n$
  - n is the order of magnitude
  - m is called the significand (also called the mantissa)
- Example
  - 123.456e-2 = 123.456×10^{-2} = 1.23456
  - 123.456e2 = 123.456×10^{2} = 12345.6
  - 1.23456e4 = 1.23456×10^{4} = 12345.6
- Normalized notation
  - Exponent is chosen so the m is at least one but less than 10
  - 12345.6 would be written as 1.23456e4 in normalized form

Floating-Point Representation

- Numerical Form: $v = (-1)^s \times M \times 2^E$
  - **Sign bit *s*** determines whether number is negative (1) or positive (0)
  - **Significand *M*** is the binary fractional value of the number, usually normalized
  - **Exponent *E*** weights the significand by a (possibly negative) power of two

- Example: floating-point representation of 15213.0
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$ (normalized form)
  - Significand
    - $M = 1.1101101101101_2$
  - Exponent
    - E = 13
  - Sign bit
    - S = 0 (positive number)

Floating Point Representation

- Numerical Form:

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

  - **Sign bit *s*** determines whether number is negative (1) or positive (0)
  - **Significand *M*** is the binary fractional value of the number, usually normalized
  - **Exponent *E*** weights value by a (possibly negative) power of two

- Encoding
  - MSB s is sign bit ***s*** (0 for +, 1 for -)
  - exp field encodes ***E*** (but is not equal to E)
  - frac field encodes ***M*** (but is not equal to M)

Precision options (diagram not to scale)

- Single precision: 32 bits (`float` in C)

- Double precision: 64 bits (`double` in C)

- Extended precision: 80 bits (Intel only)

Normalized Values (common case)

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

- Used to represent most numbers
  - Any number that can be written in normalized form
  - Everything except some numbers very close to zero
- Significand ($M = 1.xxx…x_2$) is encoded in the frac field
  - xxx…x: bits are stored in frac
  - The leading ‘1.’ is not encoded, it is implied
    - Gives us an extra bit of precision for “free”
  - Minimum value when frac=000…0 (M = 1.0)
  - Maximum value when frac=111…1 (M = 2.0 – ε)
- Exponent (E) encoded as a ***biased*** value: ***E* = *Exp* – *Bias***
  - *Exp* is a **unsigned** binary value
  - *Bias* = $2^{k-1} - 1$, where *k* is number of exponent bits
    - Single precision (8-bit exp): 127 (exp: 1…254, E: -126…127)
    - Double precision (11-bit exp): 1023 (exp: 1…2046, E: -1022…1023)
  - *Exp* is encoded as E + Bias

Normalized Encoding Example

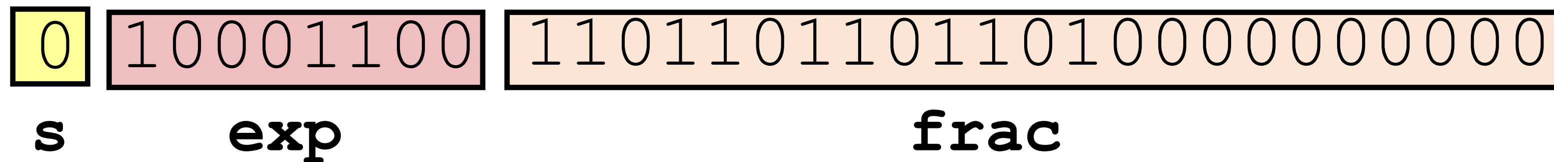
$v = (-1)^s \, M \, 2^E$
$E = Exp - Bias$

- Value: float F = 15213.0;
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- Significand
  - M = 1.1101101101101_2
  - frac = $11011011011010000000000_2$

- Exponent
  - E = 13
  - Bias = 127
  - Exp = 140 = 10001100_2

- Result:



Denormalized Values

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

- Goal: To represent 0 and have good precision for very small numbers
  - Can’t do this with normalized values having an implied leading 1.xxxx…xxx
- Condition: exp = 000…0 (all zeros for exp)
- Significand coded with implied leading 0: $M = 0.xxx…x_2$
  - xxx…x: are the bits encoding frac
- Exponent value: E = 1 – Bias (instead of E = 0 – Bias)
  - This allows for a smooth transition between normalized and denormalized numbers
- 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

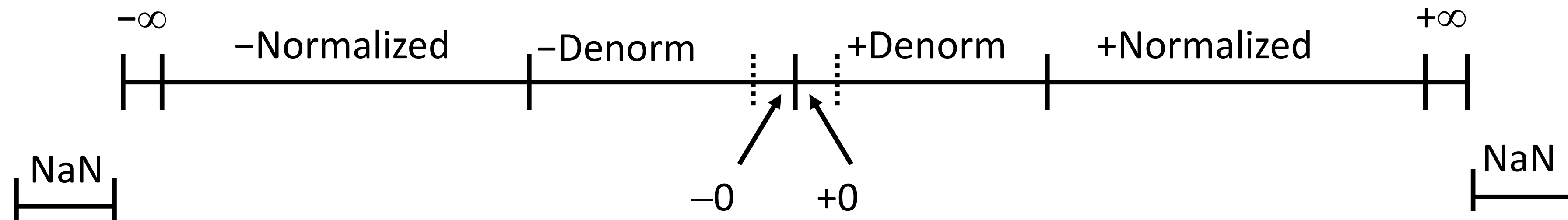
Infinity and NaN

- The other special condition: exp = 111…1 (all ones)

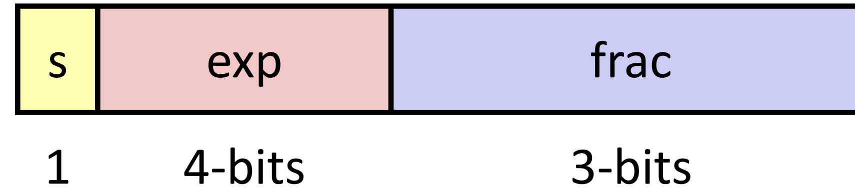
- Case: exp = 111…1, frac = 000…0
  - Represents value ∞ (infinity)
  - Both positive and negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

- Case: exp = 111…1, frac ≠ 000…0
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), $\infty - \infty$, $\infty \times 0$

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

Dynamic Range (Positive Only)

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

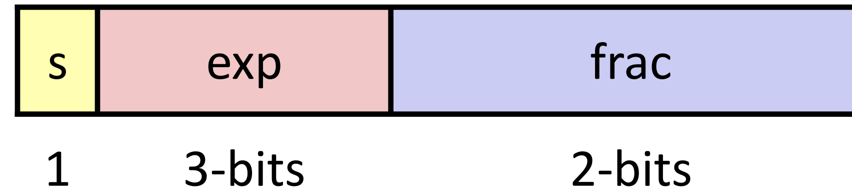
***d: E* = 1 – *Bias* (7)**

***n: E* = *Exp* – *Bias* (7)**

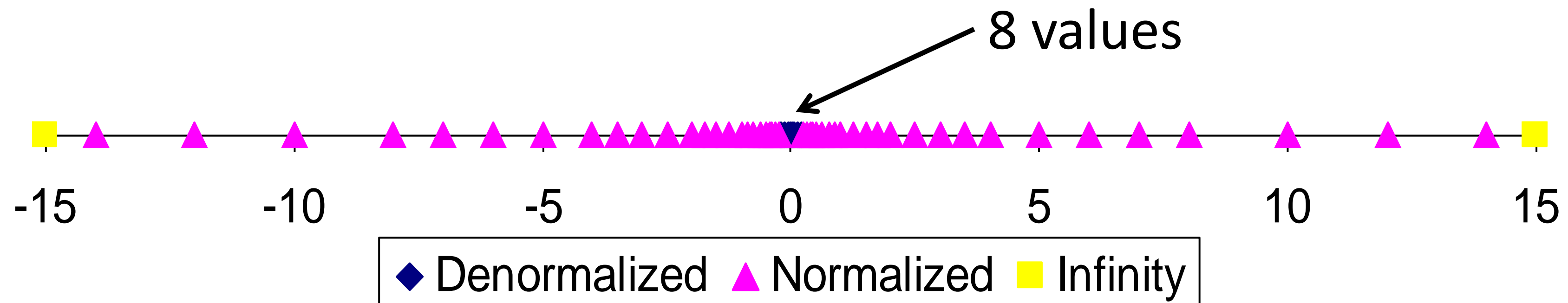
	s exp frac	E	Value	
Denormalized numbers	0 0000 000	-6	0	
	0 0000 001	-6	1/8*1/64 = 1/512	closest to zero
	0 0000 010	-6	2/8*1/64 = 2/512	
	…			
	0 0000 110	-6	6/8*1/64 = 6/512	
	0 0000 111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0 0001 000	-6	8/8*1/64 = 8/512	smallest norm
	0 0001 001	-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
  - exp = 3 exponent bits
  - frac = 2 fraction bits
  - Bias is $2^{(3-1)}-1 = 3$

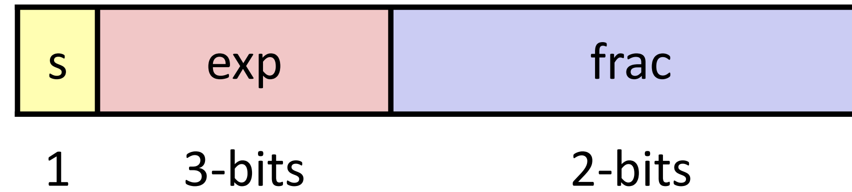


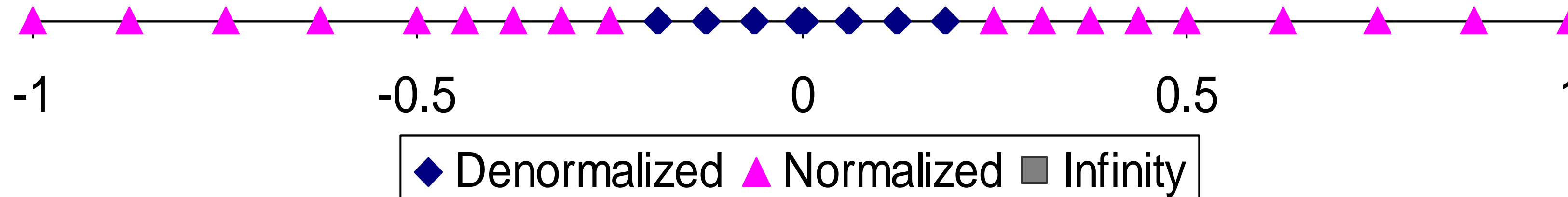
- Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3





Special Properties of the IEEE 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: 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 (illustrate with rounding to the nearest dollar)

	\$1.40	\$1.60	\$1.50	\$2.50	–\$1.50
Nearest Even (default)	\$1	\$2	\$2	\$2	–\$2
Towards zero	\$1	\$1	\$1	\$2	–\$1
Round down (−∞)	\$1	\$1	\$1	\$2	–\$2
Round up (+∞)	\$2	\$2	\$2	\$3	–\$1

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

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)

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 TMin
  - int → double
    - Exact conversion, as long as int has ≤ 53 bit word size
  - int → float
    - Will round according to rounding mode

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form M x 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 in some corner cases
    - Overflow and inexactness of rounding
    - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`
  - Makes life difficult for compilers & serious numerical applications programmers