Floating Point

CSE 351 Summer 2018

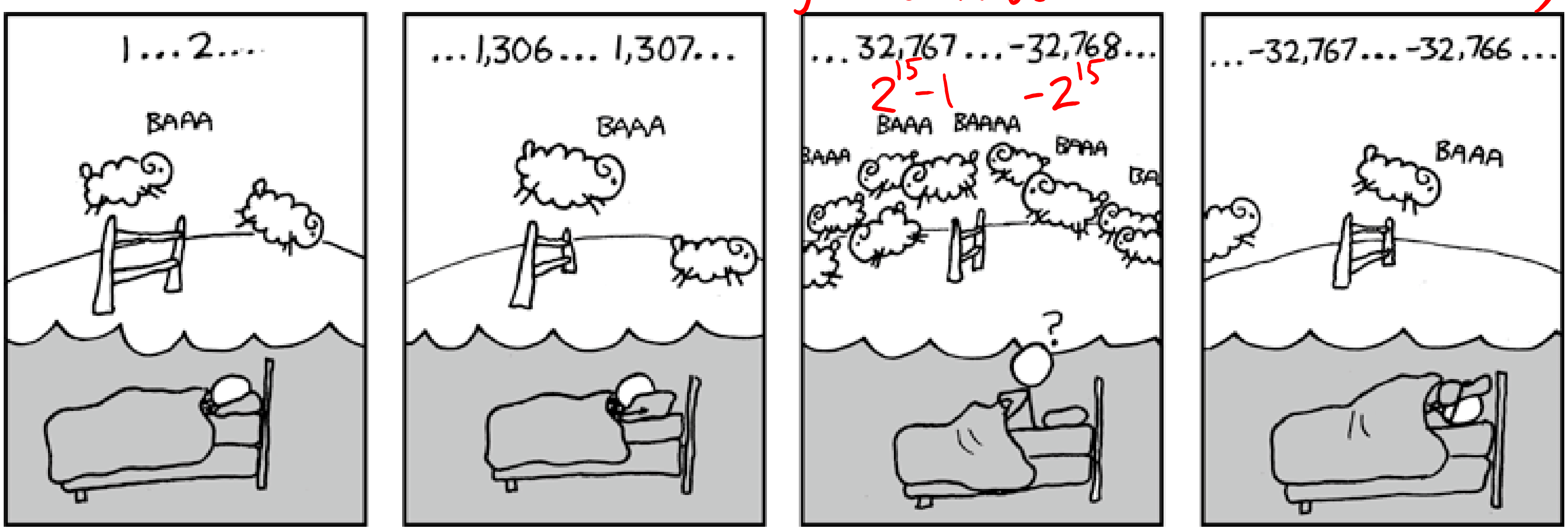
Instructor:

Justin Hsia

Teaching Assistants:

Josie Lee Natalie Andreeva Teagan Horkan

signed overflow in 16 bits → short (in C)



http://xkcd.com/571/

Administrivia

- Lab 1a due tonight at 11:59 pm
  - Only submit `pointer.c`
  - Make sure to use `dlc.py`
- Lab 1b due Thursday (7/5)
  - Submit `bits.c`, `lab1reflect.txt`
  - Start *early* and come to office hours!
- Homework 2 due 7/11
  - On Integers, Floating Point, and x86-64

Floating Point Topics

- Fractional binary numbers
- **IEEE floating-point standard**
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won't cover
  - It's a 58-page standard…

IEEE Floating Point

- IEEE 754
  - Established in 1985 as uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and result of floating operations
  - Now supported by all major CPUs

- Driven by numerical concerns
  - **Scientists**/numerical analysts want them to be as **real** as possible
  - **Engineers** want them to be **easy to implement** and **fast**

    competing goals
  - In the end:
    - Scientists mostly won out
    - Nice standards for rounding, overflow, underflow, but...
    - Hard to make fast in hardware
    - **Float operations can be an order of magnitude slower than integer operations**

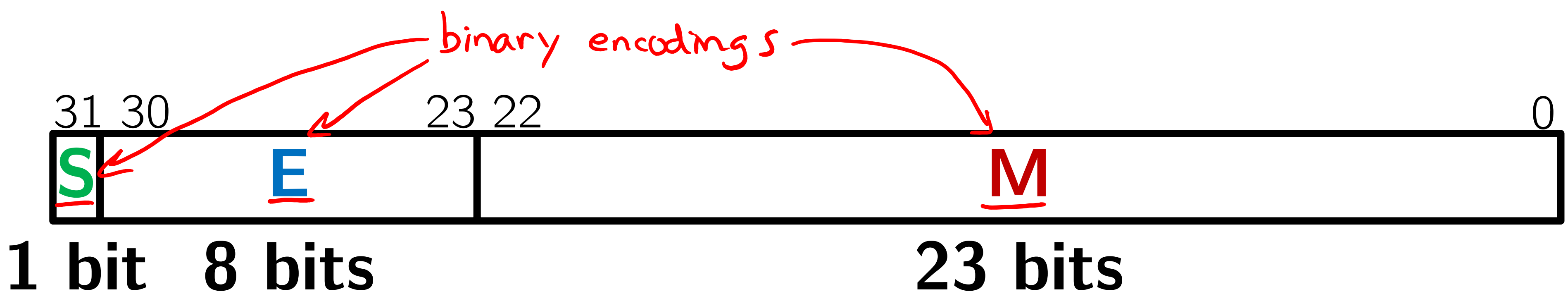
      FLOPs

      used in computer benchmarks

Floating Point Encoding

- Use normalized, base 2 scientific notation:
  - Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$
  - Bit Fields: $(-1)^{S} \times 1.M \times 2^{(E-\text{bias})}$
- Representation Scheme: (3 separate fields within 32 bits)
  - Sign bit (0 is positive, 1 is negative)
  - Mantissa (a.k.a. significand) is the fractional part of the number in normalized form and encoded in bit vector **M**
  - Exponent weights the value by a (possibly negative) power of 2 and encoded in the bit vector **E**

values



The Exponent Field

- Use biased notation
  - Read E as unsigned, but with *bias* of $2^{w-1}-1 = 127$ w=8
  - Representable exponents roughly ½ positive and ½ negative
  - Exponent 0 (Exp = 0) is represented as E = 0b 0111 1111 $= 2^8 - 1$
    E - bias = 0 = Exp ☑
- Why biased?
  - Makes floating point arithmetic easier
  - Makes somewhat compatible with two's complement

- **Practice:** To encode in biased notation, add the bias then encode in unsigned:
  - Exp = 1 $\xrightarrow{+\text{bias}}$ 128 $\xrightarrow{\text{encode}}$ E = 0b 1000 0000
  - Exp = 127 → 254 → E = 0b 1111 1110 $(254 = 255 - 1 = (2^8-1)-1)$ 8 ones in a row
  - Exp = -63 → 64 → E = 0b 0100 0000

The Mantissa (Fraction) Field

31	30 … 23	22 … 0
S	E	M
1 bit	8 bits	23 bits

$$(-1)^{S} \times (1\,.\,M) \times 2^{(E-\text{bias})}$$

- Note the implicit 1 in front of the M bit vector
  - (+) $Exp = 0$ $Man = 1.1\,0\ldots0$
  - Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000 is read as $1.1_2 = 1.5_{10}$, *not* $0.1_2 = 0.5_{10}$
  - Gives us an extra bit of *precision*
- Mantissa "limits"
  - $2^{Exp} \times 1.0..0 = 2^{Exp}$
  - Low values near M = 0b0…0 are close to 2^{Exp}
  - High values near M = 0b1…1 are close to 2^{Exp+1}
  - $2^{Exp} \times 1.1\ldots1 = 2^{Exp}(2-2^{-23}) = 2^{Exp+1} - 2^{Exp-23}$

Peer Instruction Question

- What is the correct value encoded by the following floating point number?
  - 0b $\overset{S}{0}$ $\overset{E}{10000000}$ $\overset{M}{11000000000000000000000}$

    + $\quad$ Exp = 1 ($128-127$ ← bias) $\quad$ Man = 1.110…0 (↑ implicit)
  - Vote at http://PollEv.com/justinh

A. **+ 0.75**

B. **+ 1.5**

C. **+ 2.75**

$\boxed{\textbf{D. + 3.5}}$

E. **We're lost…**

$+1.11_2 \times 2^1$

$11.1_2 = 2^1 + 2^0 + 2^{-1} = 3.5$

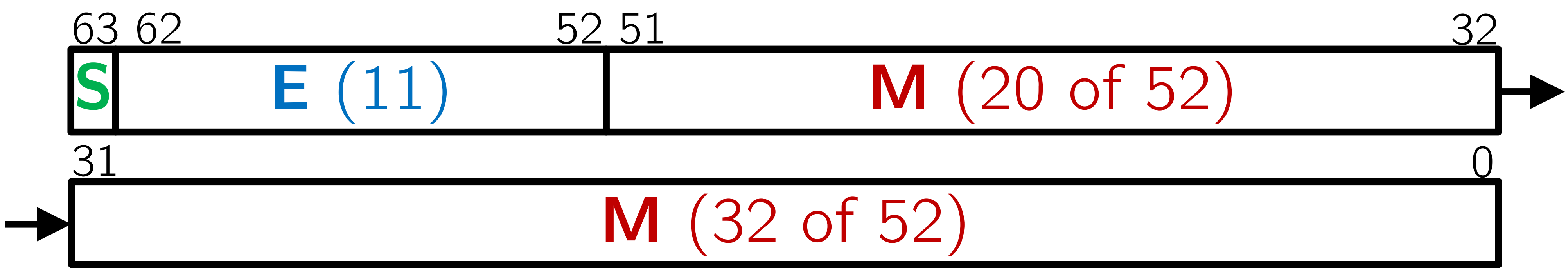
Precision and Accuracy

- Precision is a count of the number of bits in a computer word used to represent a value
  - Capacity for accuracy
- Accuracy is a measure of the difference between the *actual value of a number* and its computer representation

  - *High precision permits high accuracy but doesn't guarantee it. It is possible to have high precision but low accuracy.*
  - **Example:** `float pi = 3.14;`
    - `pi` will be represented using all 24 bits of the mantissa (highly precise), but is only an approximation (not accurate)

Need Greater Precision?

- Double Precision (vs. Single Precision) in 64 bits



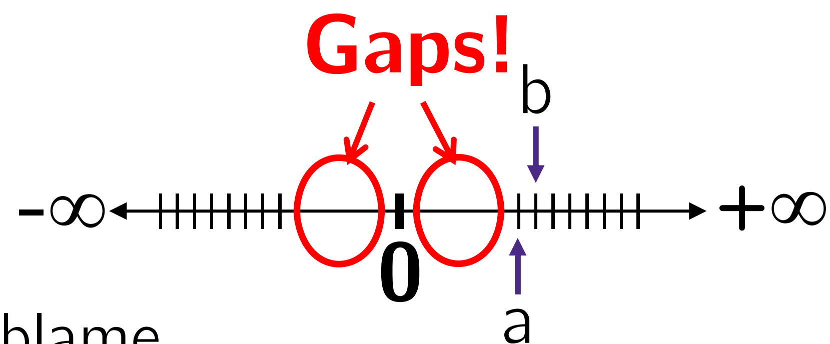
  - C variable declared as `double`
  - Exponent bias is now $2^{10}-1 = 1023$, bias $= 2^{w-1}-1$

  - **Advantages:** greater precision (larger mantissa), greater range (larger exponent)
  - **Disadvantages:** more bits used, slower to manipulate

Representing Very Small Numbers

- But wait… what happened to zero?
  $S=0, E=0, M=0 \Rightarrow Exp=-127,\ Man=1.0\ldots0$
  - Using standard encoding 0x00000000 = $1.0\times 2^{-127} \neq 0$
  - *Special case:* E and M all zeros = 0
    - Two zeros! But at least 0x00000000 = 0 like integers
      0x80000000 = -0

- New numbers closest to 0:
  (E = 0x01, Exp = -126)
  - a = $1.0\ldots0_2\times 2^{-126} = 2^{-126}$
  - b = $1.0\ldots01_2\times 2^{-126} = 2^{-126} + 2^{-149}$ (23)
  - Normalization and implicit 1 are to blame
  - *Special case:* E = 0, M ≠ 0 are denormalized numbers



Denorm Numbers

This is extra (non-testable) material

- Denormalized numbers
  - No leading 1
  - Uses implicit exponent of -126 even though $E = \text{0x00}$

- Denormalized numbers close the gap between zero and the smallest normalized number
  - Smallest norm: $\pm\ 1.0\ldots0_{two}\times 2^{-126} = \pm\ 2^{-126}$
  - Smallest denorm: $\pm\ 0.0\ldots01_{two}\times 2^{-126} = \pm\ 2^{-149}$
    - There is still a gap between zero and the smallest denormalized number

So much closer to 0

Other Special Cases

all ones

- E = 0xFF, M = 0: $\pm\infty$
  - *e.g.* division by 0
  - Still work in comparisons!
- E = 0xFF, M $\neq$ 0: Not a Number (NaN)
  - *e.g.* square root of negative number, 0/0, $\infty - \infty$
  - NaN propagates through computations
  - Value of M can be useful in debugging (tells you cause of NaN)
- New largest value (besides ∞)?
  - E = 0xFF has now been taken!
  - E = 0xFE has largest: $1.1\ldots1_2 \times 2^{127} = 2^{128} - 2^{104}$

Floating Point Encoding Summary

	E	M	Meaning
smallest E (all 0's)	0x00	0	± 0
	0x00	non-zero	± denorm num
everything else	0x01 – 0xFE	anything	± norm num
largest E (all 1's)	0xFF	0	± ∞
	0xFF	non-zero	NaN

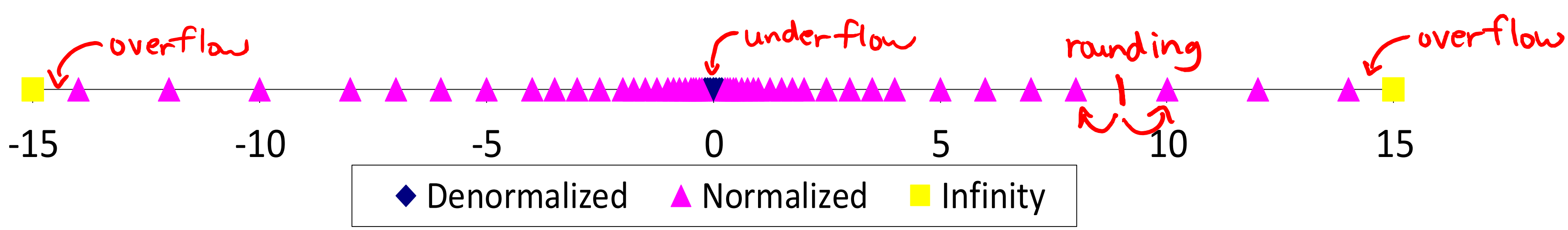
Distribution of Values

- What ranges are NOT representable?
  - Between largest norm and infinity — Overflow (Exp too large)
  - Between zero and smallest denorm — Underflow (Exp too small)
  - Between norm numbers? — Rounding
- Given a FP number, what's the bit pattern of the next largest representable number?

  if M=0b0...00, then $2^{Exp} \times 1.0$

  if M=0b0...01, then $2^{Exp} \times (1+2^{-23})$

  diff $= 2^{Exp-23}$
  - What is this "step" when Exp = 0? 2^{-23}
  - What is this "step" when Exp = 100? 2^{77}
- Distribution of values is denser toward zero



Floating Point Operations: Basic Idea

$$\text{Value} = (-1)^{S} \times \text{Mantissa} \times 2^{\text{Exponent}}$$

S	E	M

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

- Basic idea for floating point operations:
  - First, compute the exact result
  - Then *round* the result to make it fit into desired precision:
    - Possibly over/underflow if exponent outside of range
    - Possibly drop least-significant bits of mantissa to fit into M bit vector

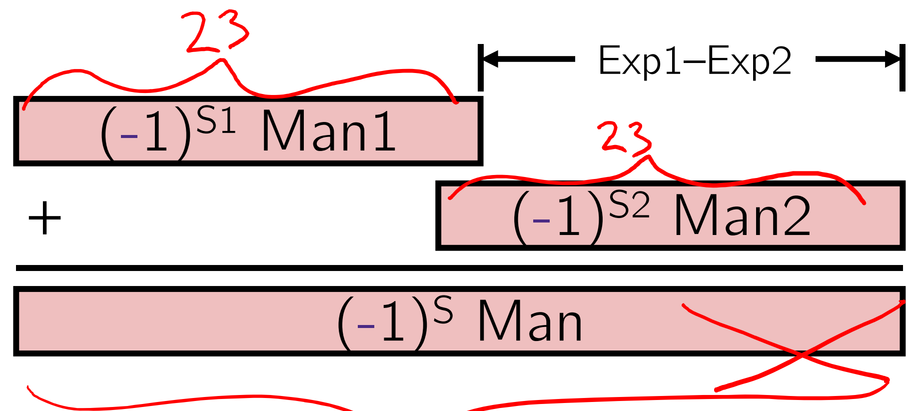
Floating Point Addition

Line up the binary points!

- $(-1)^{S1}\times \text{Man1}\times 2^{\text{Exp1}} + (-1)^{S2}\times \text{Man2}\times 2^{\text{Exp2}}$
  - Assume Exp1 > Exp2

$$\begin{array}{r} 1.010*2^{2} \\ +\ 1.000*2^{-1} \\ \hline ??? \end{array} \qquad \begin{array}{r} 1.0100*2^{2} \\ +\ 0.0010*2^{2} \\ \hline 1.0110*2^{2} \end{array}$$

- Exact Result: $(-1)^{S}\times \text{Man}\times 2^{\text{Exp}}$
  - Sign S, mantissa Man:
    - Result of signed align & add
  - Exponent E: E1



truncated if > 23 bits wide!

- Adjustments:
  - If Man ≥ 2, shift Man right, increment Exp
  - If Man < 1, shift Man left k positions, decrement Exp by k
  - Over/underflow if Exp out of range
  - Round Man to fit mantissa precision

Floating Point Multiplication

- $(-1)^{S1} \times Man1 \times 2^{Exp1} \quad \times \quad (-1)^{S2} \times Man2 \times 2^{Exp2}$

- Exact Result: $(-1)^{S} \times M \times 2^{E}$
  - Sign S: S1 ^ S2
  - Mantissa Man: Man1 × Man2
  - Exponent Exp: Exp1 + Exp2

- Adjustments:
  - If Man ≥ 2, shift Man right, increment Exp
  - Over/underflow if Exp out of range
  - Round Man to fit mantissa precision

Mathematical Properties of FP Operations

- Exponent overflow yields +∞ or -∞
- Floats with value +∞, -∞, and NaN can be used in operations
  - Result usually still +∞, -∞, or NaN; but not always intuitive
- Floating point operations do not work like real math, due to rounding

  10^{100} 0

  - Not associative: `(3.14+1e100)–1e100 != 3.14+(1e100–1e100)`

    0 **3.14**

  - Not distributive: `100*(0.1+0.2) != 100*0.1+100*0.2`

    30.000000000000003553 **30**

  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing

Floating point topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- **Floating point in C**

- There are many more details that we won't cover
  - It's a 58-page standard…

Floating Point in C

!!!

- C offers two (well, 3) levels of precision

`float`	`1.0f`	single precision (32-bit)
`double`	`1.0`	double precision (64-bit)
`long double`	`1.0L`	(*"double double" or quadruple*) precision (64-128 bits)

- `#include <math.h>` to get `INFINITY` and `NAN` constants
- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!

instead use abs(f1 - f2) < 2^{-20}

some arbitrary threshold

Floating Point Conversions in C

!!!

- Casting between `int`, `float`, and `double` **changes** the bit representation
  - `int` → `float`
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - `int` or `float` → `double`
    - Exact conversion (all 32-bit `int`s representable)
  - `long` → `double`
    - Depends on word size (32-bit is exact, 64-bit may be rounded)
  - `double` or `float` → `int`
    - Truncates fractional part (rounded toward zero)
    - "Not defined" when out of range or NaN: generally sets to `Tmin` (even if the value is a very big positive)

Floating Point and the Programmer

$1.0 \times 2^0 \to S=0,\ E=0111\ 1111,\ M=0\ldots0$

f1 = 0b 0/011 1111 1/000 0000 0000 0000 0000 0000 = 0x3F800000

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for (i = 0; i < 10; i++)
    f2 += 1.0/10.0;
  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf("f1 = %10.9f\n", f1);
  printf("f2 = %10.9f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

specify float constant

f2 should $== 10 \times \frac{1}{10} = 1$

10^{30}

10^{-30}

$10^{30} == 10^{30} + 10^{-30}$

```
$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

f2

Summary

- Floating point approximates real numbers:

31	30 … 23	22 … 0
S	**E** (8)	**M** (23)

  - Handles large numbers, small numbers, special numbers
  - Exponent in biased notation (bias = $2^{w-1}-1$)
    - Outside of representable exponents is *overflow* and *underflow*
  - Mantissa approximates fractional portion of binary point
    - Implicit leading 1 (normalized) except in special cases
    - Exceeding length causes *rounding*

E	M	Meaning
0x00	0	± 0
0x00	non-zero	± denorm num
0x01 – 0xFE	anything	± norm num
0xFF	0	± ∞
0xFF	non-zero	NaN

BONUS SLIDES

An example that applies the IEEE Floating Point concepts to a smaller (8-bit) representation scheme.

These slides expand on material covered today, so while you don't need to read these, the information is "fair game."

Tiny Floating Point Example

S	E	M
1	4	3

- 8-bit Floating Point Representation
  - The sign bit is in the most significant bit (MSB)
  - The next four bits are the exponent, with a bias of $2^{4-1}-1 = 7$
  - The last three bits are the mantissa

- Same general form as IEEE Format
  - Normalized binary scientific point notation
  - Similar special cases for 0, denormalized numbers, NaN, ∞

Dynamic Range (Positive Only)

	S	E	M	Exp	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	

Special Properties of Encoding

- Floating point zero (0^+) exactly the same bits as integer zero
  - All bits = 0

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