Special values

- If the mantissa is always (1 + f), then how is 0 represented?
  - The fraction field f should be 0000...0000.
  - The exponent field e contains the value 00000000.
  - With signed magnitude, there are *two* zeroes: +0.0 and -0.0.
- There are representations of positive and negative infinity, which might sometimes help with instances of overflow.
  - The fraction f is 0000...0000.
  - The exponent field e is set to 11111111.
- Finally, there is a special "not a number" value, which can handle some cases of errors or invalid operations such as 0.0/0.0.
  - The fraction field f is set to any non-zero value.
  - The exponent e will contain 11111111.
- The smallest and largest possible exponents e=00000000 and e=11111111 (and their double precision counterparts) are reserved for special values.

©1998 Morgan Kaufmann Publishers

In short

E	F	meaning
00000000	0…0	0
00000000	X…X	Valid number
11111111	0…0	Infinity
11111111	X…X	Not a Number

Unnormalized

©1998 Morgan Kaufmann Publishers

Specifically

- If E=255 and F is nonzero, then V=NaN ("Not a number")
- If E=255 and F is zero and S is 1, then V=-Infinity
- If E=255 and F is zero and S is 0, then V=Infinity
- If 0<E<255 then $V=(-1)^S \times 2^{E-127} \times (1.F)$ where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If E=0 and F is nonzero, then $V=(-1)^S \times 2^{-126} \times (0.F)$. These are "unnormalized" values.
- If E=0 and F is zero and S is 1, then V=-0
- If E=0 and F is zero and S is 0, then V=0

©1998 Morgan Kaufmann Publishers

Range of *normalized* single-precision numbers

$$(1 - 2s) * (1 + f) * 2^{e-127}.$$

- Normalized FP: the exponent > 0
- And the smallest *positive* non-zero number is $1 * 2^{-126} = 2^{-126}$.
  - The smallest e is 00000001 (1).
  - The smallest f is 00000000000000000000000 (0).
- The largest possible “normal” number is $(2 - 2^{-23}) * 2^{127} = 2^{128} - 2^{104}$.
  - The largest possible e is 11111110 (254).
  - The largest possible f is 11111111111111111111111$(1 - 2^{-23})$.
- In comparison, the smallest and largest possible 32-bit integers in two’s complement are only -2^{32} and $2^{31} - 1$
- How can we represent so many more values in the IEEE 754 format, even though we use the same number of bits as regular integers?

©1998 Morgan Kaufmann Publishers

If we take the unnormalized values

	Unnormalized	Normalized
Value range	$\pm 2^{-149}$ to $(1-2^{-23})\times 2^{-126}$	$\pm 2^{-126}$ to $(2-2^{-23})\times 2^{127}$

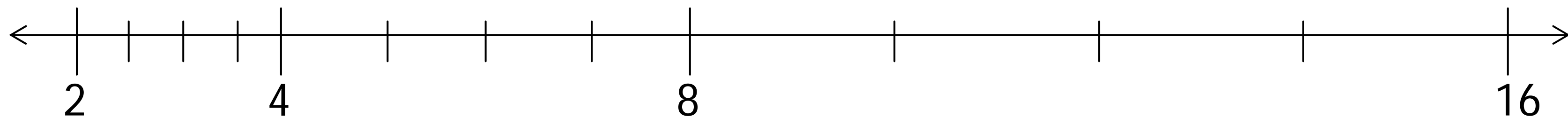
Not representable numbers

- Negative numbers less than $-(2-2^{-23}) \times 2^{127}$ (*negative overflow*)
- Negative numbers greater than -2^{-149} (*negative underflow*)
- Zero
- Positive numbers less than 2^{-149} (*positive underflow*)
- Positive numbers greater than $(2-2^{-23}) \times 2^{127}$ (*positive overflow*)

©1998 Morgan Kaufmann Publishers

Finiteness

- There *aren't* more IEEE numbers.
- With 32 bits, there are $2^{32}-1$, or about 4 billion, different bit patterns.
  - These can represent 4 billion integers *or* 4 billion reals.
  - But there are an infinite number of reals, and the IEEE format can only represent *some* of the ones from about -2^{128} to $+2^{128}$.
  - Represent same number of values between 2^n and 2^{n+1} as 2^{n+1} and 2^{n+2}



- Thus, floating-point arithmetic has "issues"
  - Small roundoff errors can accumulate with multiplications or exponentiations, resulting in big errors.
  - Rounding errors can invalidate many basic arithmetic principles such as the associative law, $(x + y) + z = x + (y + z)$.
- The IEEE 754 standard guarantees that all machines will produce the same results—but those results may not be mathematically correct!

©1998 Morgan Kaufmann Publishers

Limits of the IEEE representation

- Even some integers cannot be represented in the IEEE format.

```
int x   = 33554431;
float y = 33554431;
printf( "%d\n", x );
printf( "%f\n", y );
```

```
33554431
33554432.000000
```

- Some simple decimal numbers cannot be represented exactly in binary to begin with.

$$0.10_{10} = 0.0001100110011..._{2}$$

©1998 Morgan Kaufmann Publishers

0.10

- During the Gulf War in 1991, a U.S. Patriot missile failed to intercept an Iraqi Scud missile, and 28 Americans were killed.
- A later study determined that the problem was caused by the inaccuracy of the binary representation of 0.10.
  - The Patriot incremented a counter once every 0.10 seconds.
  - It multiplied the counter value by 0.10 to compute the actual time.
- However, the (24-bit) binary representation of 0.10 actually corresponds to 0.099999904632568359375, which is off by 0.000000095367431640625.
- This doesn't seem like much, but after 100 hours the time ends up being off by 0.34 seconds–enough time for a Scud to travel 500 meters!
- Professor Skeel wrote a short article about this.

Roundoff Error and the Patriot Missile. SIAM News, 25(4):11, July 1992.

©1998 Morgan Kaufmann Publishers

Floating-point addition example

- To get a feel for floating-point operations, we'll do an addition example.
  - To keep it simple, we'll use base 10 scientific notation.
  - Assume the mantissa has four digits, and the exponent has one digit.
- An example for the addition:

$$99.99 + 0.161 = 100.151$$

- As normalized numbers, the operands would be written as:

$$9.999 * 10^{1} \qquad 1.610 * 10^{-1}$$

©1998 Morgan Kaufmann Publishers

Steps 1-2: the actual addition

1. **Equalize the exponents.**

   The operand with the smaller exponent should be rewritten by increasing its exponent and shifting the point leftwards.

$$1.610 * 10^{-1} = 0.01610 * 10^{1}$$

   With four significant digits, this gets rounded to:

   This can result in a loss of least significant digits—the rightmost 1 in this case. But rewriting the number with the larger exponent could result in loss of the *most* significant digits, which is much worse.

2. **Add the mantissas.**

$$\begin{array}{rcl} 9.999 & * & 10^{1} \\ +\ 0.016 & * & 10^{1} \\ \hline 10.015 & * & 10^{1} \end{array}$$

©1998 Morgan Kaufmann Publishers

Steps 3-5: representing the result

3. **Normalize the result if necessary.**

$$10.015 * 10^1 = 1.0015 * 10^2$$

This step may cause the point to shift either left or right, and the exponent to either increase or decrease.

4. **Round the number if needed.**

$1.0015 * 10^2$ gets rounded to $1.002 * 10^2$

5. **Repeat Step 3 if the result is no longer normalized.**

We don't need this in our example, but it's possible for rounding to add digits—for example, rounding 9.9995 yields 10.000.

Our result is $1.002*10^2$, or 100.2 . The correct answer is 100.151, so we have the right answer to four significant digits, but there's a small error already.

©1998 Morgan Kaufmann Publishers

Multiplication

- To multiply two floating-point values, first multiply their magnitudes and add their exponents.

$$
\begin{array}{rrcl}
 & 9.999 & * & 10^{1} \\
* & 1.610 & * & 10^{-1} \\
\hline
 & 16.098 & * & 10^{0}
\end{array}
$$

- You can then round and normalize the result, yielding $1.610 * 10^{1}$.
- The sign of the product is the exclusive-or of the signs of the operands.
  – If two numbers have the same sign, their product is positive.
  – If two numbers have different signs, the product is negative.

$0 \oplus 0 = 0 \qquad 0 \oplus 1 = 1 \qquad 1 \oplus 0 = 1 \qquad 1 \oplus 1 = 0$

- This is one of the main advantages of using signed magnitude.

©1998 Morgan Kaufmann Publishers

The history of floating-point computation

- In the past, each machine had its own implementation of floating-point arithmetic hardware and/or software.
  - It was impossible to write portable programs that would produce the same results on different systems.
- It wasn't until 1985 that the IEEE 754 standard was adopted.
  - Having a standard at least ensures that all compliant machines will produce the same outputs for the same program.

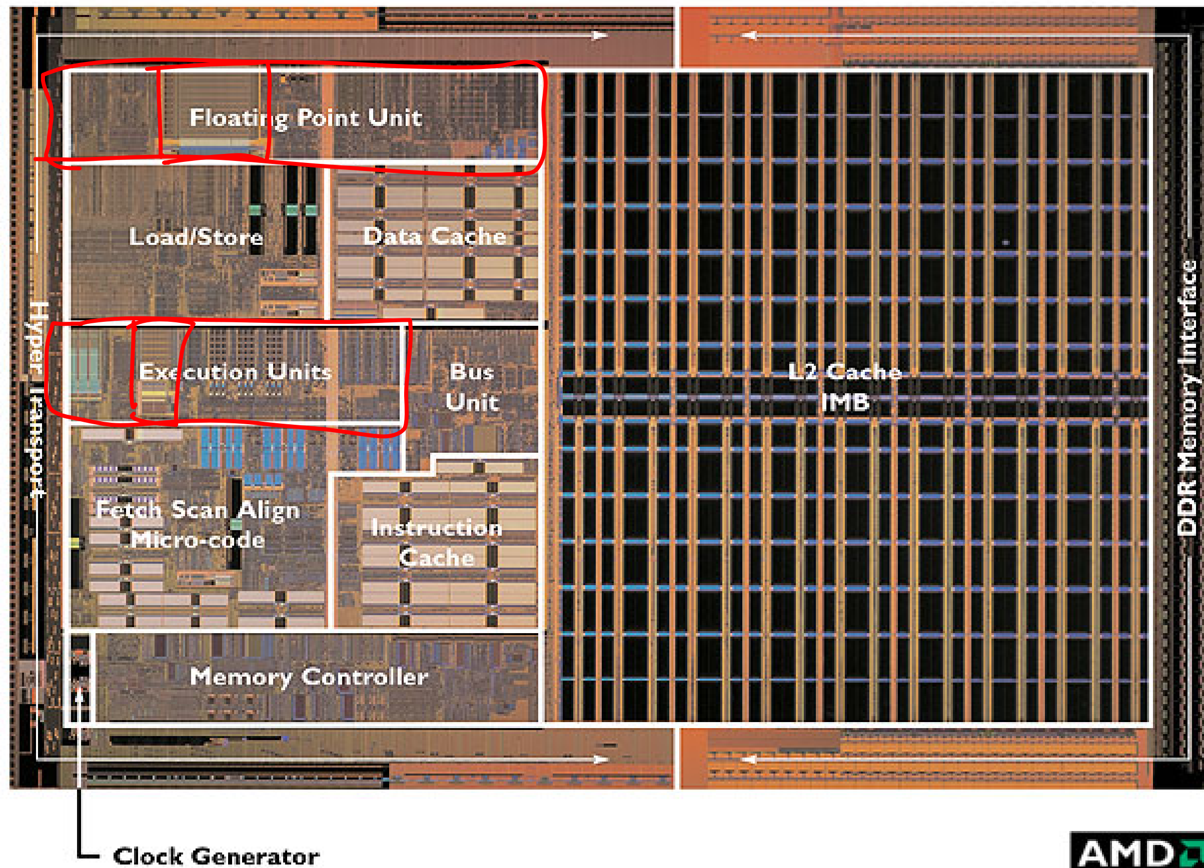
©1998 Morgan Kaufmann Publishers

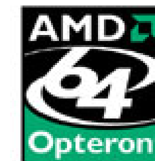
Floating-point hardware

- When floating point was introduced in microprocessors, there wasn't enough transistors on chip to implement it.
  – You had to buy a floating point co-processor (e.g., the Intel 8087)
- As a result, many ISA's use separate registers for floating point.
- Modern transistor budgets enable floating point to be on chip.
  – Intel's 486 was the first x86 with built-in floating point (1989)
- Even the newest ISA's have separate register files for floating point.
  – Makes sense from a floor-planning perspective.

©1998 Morgan Kaufmann Publishers

FPU like co-processor on chip





©1998 Morgan Kaufmann Publishers

Summary

- The IEEE 754 standard defines number representations and operations for floating-point arithmetic.
- Having a finite number of bits means we can't represent all possible real numbers, and errors will occur from approximations.

©1998 Morgan Kaufmann Publishers