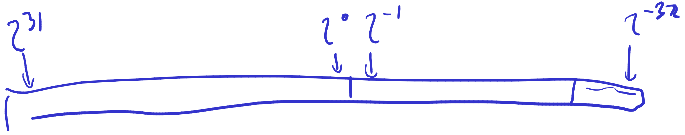
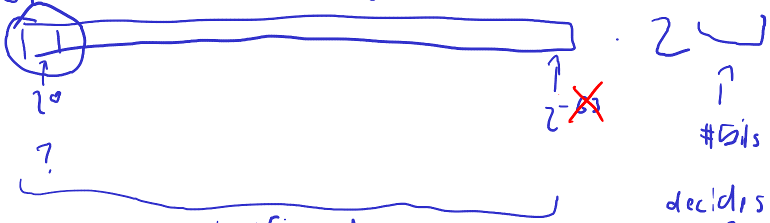
HW 2 out

Fixed point:



unevening of rounding error → scope

out of 64



bits in exponent decides magnitude of numbers we can store

significand $\in [0, 2)$

$$2^0 + 2^{-1} + 2^{-2} + \dots \to 2$$

exp range (w/ 8 bits):

$0 \dots (1111\ 1111)_2 = 255$

$2^0 \qquad 2^{255}$

→ makes numbers <1 hard

→ no

$-128 \dots 127$

$2^{-128} \dots 2^{127}$

In real life 64 bit FP:

$-1022 \dots 1023$ → 11 bits

1 sign bit

52 bits significand

$$23.625 = (\underbrace{1011}_{}1.101)_2$$

$$= (1.0111\ 101\quad)_2 \cdot 2^4$$

$$= (\underline{0}.10111\ 101) \cdot 2^5$$

← wasteful: if rounding, we would want all bits used for accuracy in the significand

"normalization": leading bit in the significand is always 1 normalization $\Rightarrow$

$$1 \leq \text{significand} < 2$$

Idea: Don't want to store leading 1 bit in sig. But: that makes 0 unrepresentable.

Floating Point Numbers

Convert $13 = (1101)_2$ into floating point representation.

What pieces do you need to store an FP number?

Floating Point: Implementation, Normalization

Previously: Consider *mathematical* view of FP. (via example: $(1101)_2$)

Next: Consider *implementation* of FP in hardware.

Do you notice a source of inefficiency in our number representation?

leading in in significand

sig: 1

won't store

stored

exp: $-1022 \ldots\ldots 1023$

$= (-1023)$ + stored bit pattern as nonneg. integer

implicit offset

Unrepresentable numbers?

Can you think of a somewhat central number that we cannot represent as

$$x = (1.\_\_\_\_\_\_\_\_\_\_\_\_)_2 \cdot 2^{-p}?$$

zero

↳ use $(000\ldots0)_2$ exponent
as special case to
turn off leading bit
in sig.

Demo: Picking apart a floating point number [cleared]

Subnormal Numbers

What is the smallest representable number in an FP system with 4 stored bits (5 total) in the significand and a stored exponent range of $[-7, 8]$?

when leading 1 is turned off.

Subnormal Numbers

What is the smallest representable number in an FP system with 4 stored bits (5 total) in the significand and a stored exponent range of $[-7, 8]$?

First attempt:

- Significand as small as possible $\rightarrow$ all zeros after the implicit leading one
- Exponent as small as possible: -7

So:

$$(1.0000)_2 \cdot 2^{-7}.$$

Unfortunately: wrong.

Subnormal Numbers, Attempt 2

What is the smallest representable number in an FP system with 4 stored bits in the significand and a (stored) exponent range of $[-7, 8]$?

Why learn about subnormals?

slow

Subnormal Numbers, Attempt 2

What is the smallest representable number in an FP system with 4 stored bits in the significand and a (stored) exponent range of $[-7, 8]$?

- Can go way smaller using the *special exponent* (turns off the leading one)
- Assume that the special exponent is -7.
- So: $(0.001)_2 \cdot 2^{-7}$ (with all four digits stored).

Numbers with the special epxonent are called *subnormal* (or *denormal*) FP numbers. Technically, zero is also a subnormal.

Why learn about subnormals?

- Subnormal FP is often slow: not implemented in hardware.
- Many compilers support options to 'flush subnormals to zero'.

Underflow

- FP systems without subnormals will *underflow* (return 0) as soon as the exponent range is exhausted.
- This smallest representable *normal* number is called the *underflow level*, or *UFL*.
- Beyond the underflow level, subnormals provide for *gradual underflow* by 'keeping going' as long as there are bits in the significand, but it is important to note that subnormals don't have as many accurate digits as normal numbers.
  Read a story on the epic battle about gradual underflow
- Analogously (but much more simply–no 'supernormals'): the overflow level, *OFL*.

Rounding Modes

How is rounding performed? (Imagine trying to represent π.)

$$(\underbrace{1.1101010}_{\text{representable}}11)_2$$

- chop off extra digits
- "round to nearest"

What is done in case of a tie? $0.5 = (0.1)_2$ ("Nearest"?)

If you round down always, might induce bias

"round to even": round down half the time.

Demo: Density of Floating Point Numbers [cleared]

Rounding Modes

How is rounding performed? (Imagine trying to represent π.)

$$\left(\underbrace{1.1101010}_{\text{representable}}11\right)_2$$

What is done in case of a tie? $0.5 = (0.1)_2$ ("Nearest"?)

Up or down? It turns out that picking the same direction every time introduces *bias*. Trick: *round-to-even*.

$$0.5 \to 0, \qquad 1.5 \to 2$$

Demo: Density of Floating Point Numbers [cleared]

Smallest Numbers Above...

- What is smallest FP number > 1? Assume 4 stored bits (5 total) in the significand.

What's the smallest FP number > 1024 in that same system?

Can we give that number a name?

Unit Roundoff

Unit roundoff or *machine precision* or *machine epsilon* or $\varepsilon_{\text{mach}}$ is...

the smallest number such that

$$fl(1+\varepsilon) > 1$$

- ε_{mach} depends on rounding rule
- If the rounding rule has tie breaking, then assume rounding up.

actually: $1 \oplus_{fl} 2^{-53} = 1$ ← IEEE 64

$\varepsilon_{mach} = 2^{-53}$

FP: Relative Rounding Error

What does this say about the relative error incurred in floating point calculations?

To get from x_{fl} to the next bigger representable number: $x_{fl}(1+\varepsilon_{mach})$

relative rounding error:

$$\left|\frac{\hat{x}-x}{x}\right| = \left|\frac{x(1+\varepsilon_{mach})-x}{x}\right| = \varepsilon_{mach}$$

ε_{mach} is an upper bound on relative rounding error.

FP: Machine Epsilon

What's machine epsilon for double-precision floating point with round-to-nearest? (52 stored bits in the significand, 53 total)

Demo: Floating Point and the Harmonic Series [cleared]

Implementing Arithmetic

How is floating point addition implemented?

Consider adding $a = (1.101)_2 \cdot 2^1$ and $b = (1.001)_2 \cdot 2^{-1}$ in a system with three stored bits (four total) in the significand.

$$
\begin{array}{rll}
 & 1.1010 & \cdot\ 2^5 \\
+ & 1.0001 & \cdot\ 2^3 \\
\hline
\end{array}
$$

$$
\begin{array}{rll}
 & 1.1010 & \cdot\ 2^5 \\
+ & 0.01001 & ? \ \cdot\ 2^3 \\
\hline
\end{array}
$$

Implementing Arithmetic

How is floating point addition implemented?
Consider adding $a = (1.101)_2 \cdot 2^1$ and $b = (1.001)_2 \cdot 2^{-1}$ in a system with three stored bits (four total) in the significand.

Rough algorithm:

1. Bring both numbers onto a common exponent
2. Do grade-school addition from the front, until you run out of digits in your system.
3. Round result.

$$
\begin{aligned}
a &= 1.\ 101 \cdot 2^1 \\
b &= 0.\ 01001 \cdot 2^1 \\
a + b &\approx 1.\ 111 \cdot 2^1
\end{aligned}
$$

Problems with FP Addition

What happens if you subtract two numbers of very similar magnitude?
As an example, consider $a = (1.1011)_2 \cdot 2^0$ and $b = (1.1010)_2 \cdot 2^0$.

Demo: Catastrophic Cancellation [cleared]