

CSE 31

Computer Organization

Lecture 23 – Floating Point Numbers (wrap up)

Announcements

- Labs
 - Lab 9 grace period* ends this week
 - » Demo is REQUIRED to receive full credit
 - Lab 10 due this week (with **7 days grace period*** after due date)
 - » Demo is **NOT REQUIRED** to receive full credit
- Reading assignments
 - **All** Reading (01-08) and Homework (01-06) assignments **open for submission till 09-MAY, 11:59pm**
 - » Complete **Participation/Challenge** Activities in each section to receive grade
 - » IMPORTANT: Make sure to submit score to CatCourses by using the link in the assignment page
 - » You may re-do past Reading/Homework assignments to improve score.

* A 10% penalty will be applied for late *submissions* during the grace period.

Announcements

- Project 02
 - Due 05-MAY
 - Can work in teams of 2 students
 - » Each team member must identify teammate in “Comments...” text-box at the submission page
 - » If working in teams, each student must submit code (can be the same as teammate) and demo individually
 - » Grade can vary among teammates depending on demo
 - Demo required for project grade
 - » No partial credit for submission without demo
 - **No grace period**
 - » **Must complete submission and demo by due date.**

Representation of Fractions (review)

So far, in our examples we used a “fixed” binary point


What we really want is to “**float**” the binary point. Why?

Floating binary point most effective use of our limited bits (and thus more accuracy in our number

representation):

example: put 0.1640625 into binary. Represent in 5-bits choosing where to put the binary point.

... 000000.001010100000...



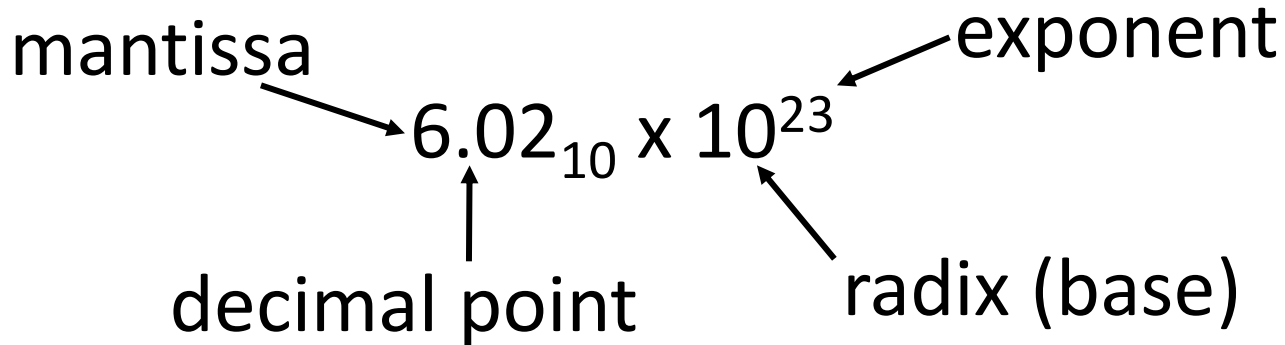
Store these bits and keep track of the binary point 2 places to the left of the MSB

Any other solution would lose accuracy!

With floating point rep., each numeral carries an exponent field recording the whereabouts of its binary point.

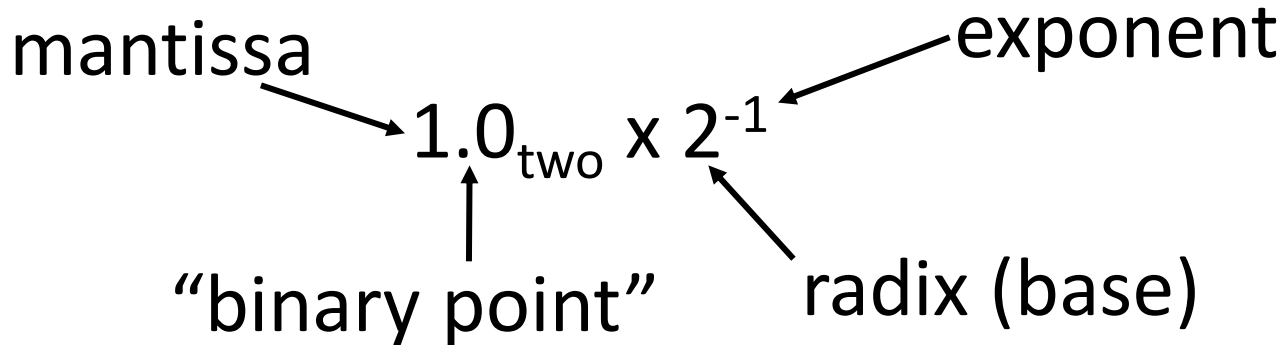
The binary point **can be outside** the stored bits, so very large and small numbers can be represented.

Scientific Notation (in Decimal)



- Normalized form: no leading 0s
(exactly one digit to left of decimal point)
- Alternatives to representing $1/1,000,000,000$
 - Normalized: 1.0×10^{-9}
 - Not normalized: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

Scientific Notation (in Binary)



- Computer arithmetic that supports it called floating point, because it represents numbers where the binary point is not fixed, as it is for integers
 - Declare such variable in C as `float`

Floating Point Representation (1/2)

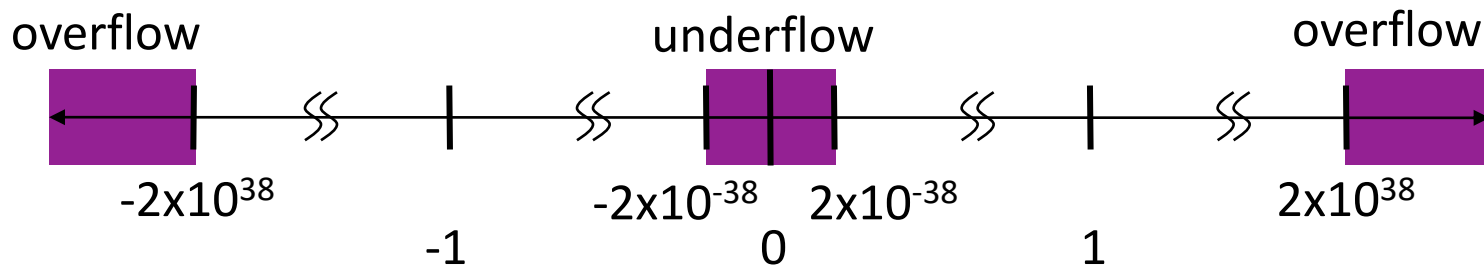
- Normal format: $+1.\text{xxx}\dots\text{x}_{\text{two}} * 2^{\text{yyy}\dots\text{y}_{\text{two}}}$
- Multiple of Word Size (32 bits)



- S represents Sign
- Exponent represents y's
- Significand represents x's
- Represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}

Floating Point Representation (2/2)

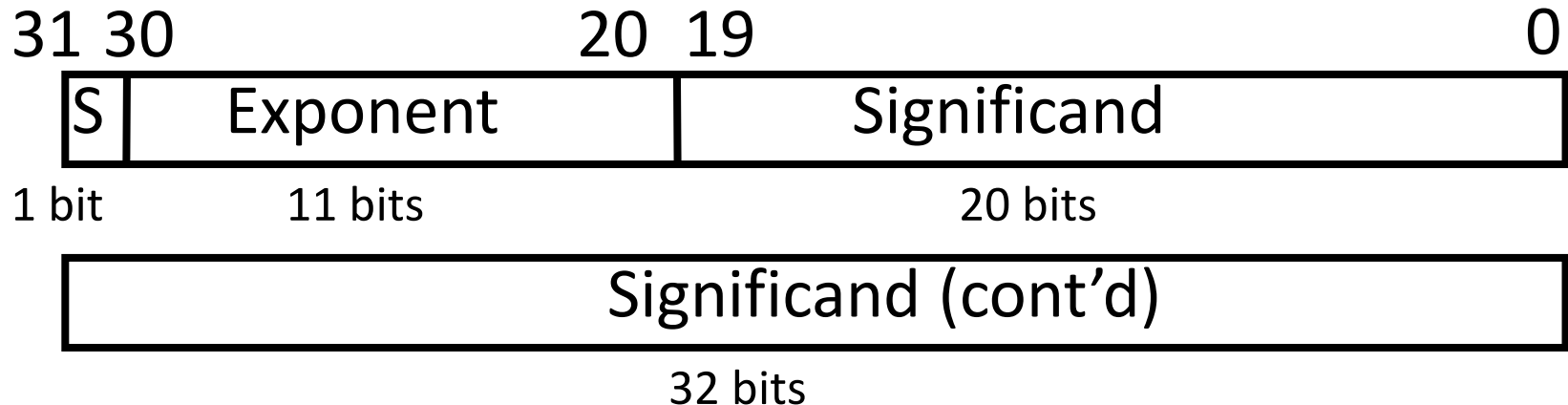
- What if result too large? ($> 2.0 \times 10^{38}$, $< -2.0 \times 10^{38}$)
 - Overflow! \Rightarrow Exponent larger than what can be represented in 8-bit Exponent field
- What if result too small? (> 0 & $< 2.0 \times 10^{-38}$, $> -2.0 \times 10^{-38}$ & < 0)
 - Underflow! \Rightarrow Negative exponent larger than what can be represented in 8-bit Exponent field



- What would help reduce chances of overflow and/or underflow?

Double Precision Fl. Pt. Representation

- Next Multiple of Word Size (64 bits)



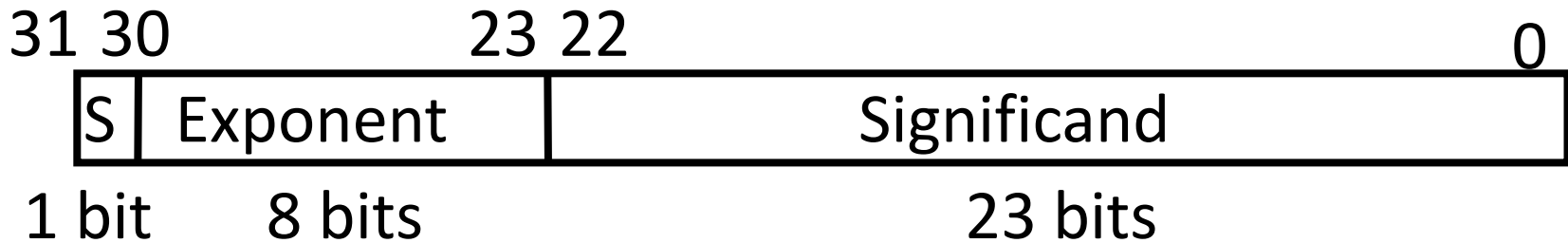
- Double Precision (vs. Single Precision)
 - C variable declared as double
 - Represent numbers almost as small as 2.0×10^{-308} to almost as large as 2.0×10^{308}
 - But primary advantage is greater accuracy due to larger significand

QUAD Precision Fl. Pt. Representation

- Next Multiple of Word Size (128 bits)
 - Unbelievable **range** of numbers
 - Unbelievable **precision** (accuracy)
- IEEE 754-2008 “binary128” standard
 - Has 15 exponent bits and 112 significand bits (can represent 113 precision bits)
- Oct-Precision?
 - Some have tried, no real traction so far
- Half-Precision?
 - Yep, “binary16”

IEEE 754 Floating Point Standard (1/3)

Single Precision (DP similar):

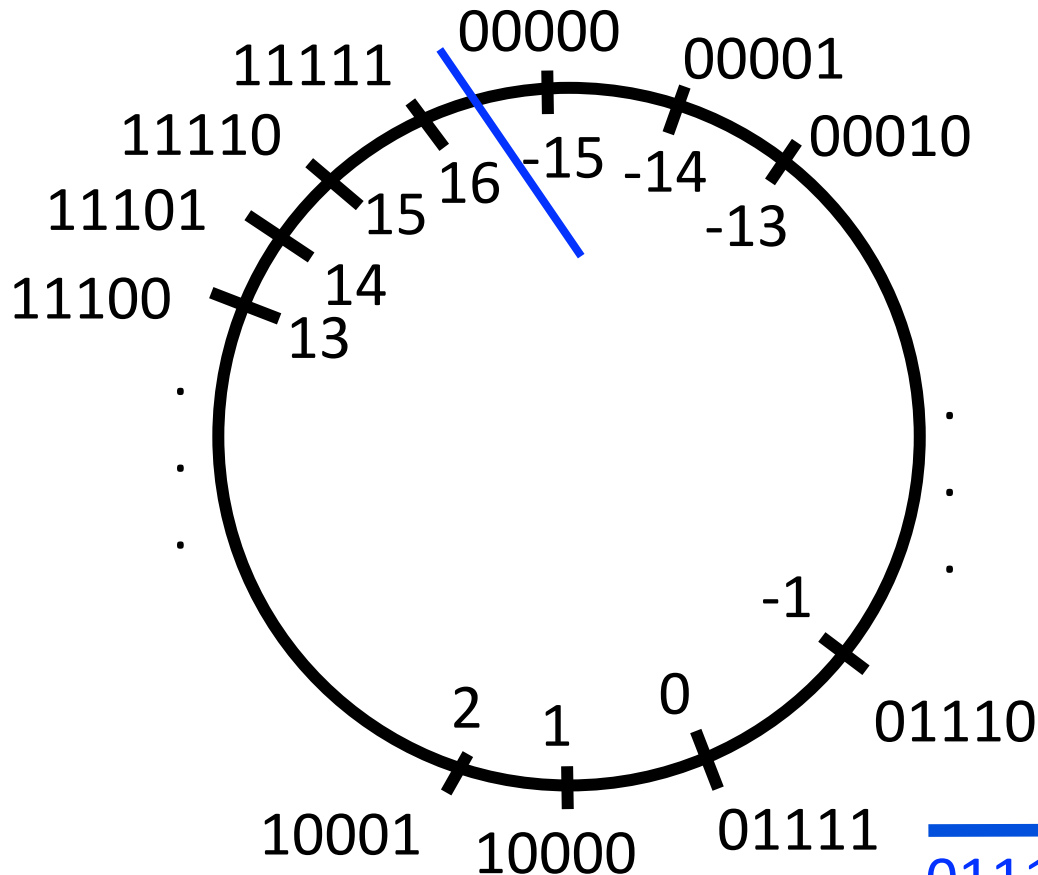


- Sign bit:
 - 1 means negative
 - 0 means positive
- Significand:
 - To pack more bits, leading 1 is implicit for normalized numbers
 - 1 + 23 bits single, 1 + 52 bits double
 - always true: $0 < \text{Significand} < 1$ (for normalized numbers)
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

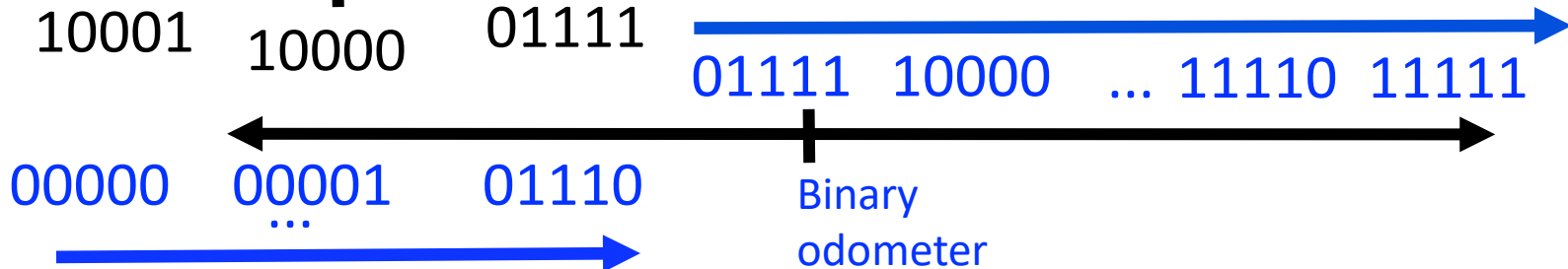
IEEE 754 Floating Point Standard (2/3)

- IEEE 754 uses “biased exponent” representation.
 - Designers wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
 - Wanted bigger (integer) exponent field to represent bigger numbers.
 - 2’s complement poses a problem (because negative numbers look bigger)
 - We’re going to see that the numbers are ordered EXACTLY as in sign-magnitude
 - » i.e., counting from binary 00...00 up to 11...11 goes from 0 to +MAX to -0 to -MAX to 0

Recall: Bias Encoding: $N = 5$ (bias = 15)



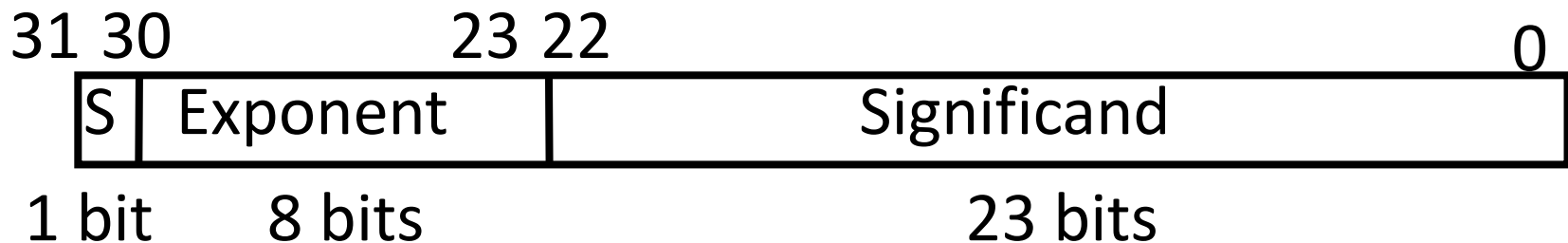
- ▶ Want 00... to represent the smallest number
- ▶ value = unsigned - bias
- ▶ Bias for N bits = $2^{N-1} - 1$
- ▶ one zero
- ▶ how many positives?
 - 2^{N-1}
 - (more than 2's complement)



IEEE 754 Floating Point Standard (3/3)

- Called Biased Notation, where bias is number subtracted to get real number
 - IEEE 754 uses bias of 127 for single precision
 - Subtract 127 from Exponent field to get actual value for exponent
 - 1023 is bias for double precision

- Summary (single precision):



- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
 - Double precision identical, except with exponent bias of 1023 (half, quad similar)

Understanding the Significand (1/2)

- Method 1 (Fractions):

- In decimal: 0.340_{10} $\Rightarrow 340_{10}/1000_{10}$
 $\Rightarrow 34_{10}/100_{10}$

- In binary: 0.110_2 $\Rightarrow 110_2/1000_2 = 6_{10}/8_{10}$
 $\Rightarrow 11_2/100_2 = 3_{10}/4_{10}$

- Advantage: less purely numerical, more thought oriented; this method usually helps people understand the meaning of the significand better

Understanding the Significand (2/2)

- Method 2 (Place Values):
 - Convert from scientific notation
 - In decimal:
 - » $1.6732 = (1 \times 10^0) + (6 \times 10^{-1}) + (7 \times 10^{-2}) + (3 \times 10^{-3}) + (2 \times 10^{-4})$
 - In binary:
 - » $1.1001 = (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$
 - Interpretation of value in each position extends beyond the decimal/binary point
 - Advantage: good for quickly calculating significand value; use this method for translating FP numbers

Converting Binary FP to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010
---	-----------	------------------------------

- Sign: 0 → positive
- Exponent:
 - 0110 1000_{two} = 104_{ten}
 - Bias adjustment: 104 - 127 = -23
- Significand:

$$\begin{aligned}
 &1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + \dots \\
 &= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22} \\
 &= 1.0 + 0.666115
 \end{aligned}$$
- Represents: $1.666115_{\text{ten}} * 2^{-23} \sim 1.986 * 10^{-7}$ (about 2/10,000,000)

Converting Decimal to FP

-2.340625×10^1

1. Denormalize: -23.40625
2. Convert integer part:
 $23 = 16 + (7 = 4 + (3 = 2 + (1))) = 10111_2$
3. Convert fractional part:
 $.40625 = .25 + (.15625 = .125 + (.03125)) = .01101_2$
4. Put parts together and normalize:
 $10111.01101 = 1.011101101 \times 2^4$
5. Convert exponent: $127 + 4 = 10000011_2$

1	1000 0011	011 1011 0100 0000 0000 0000
---	-----------	------------------------------

Quiz

1	1000 0001	111 0000 0000 0000 0000 0000
---	-----------	------------------------------

What is the decimal equivalent of the floating pt # above?

Note: $1000\ 0001_2 = 129_{10}$

$$2^{-1} = 0.5$$

$$2^{-2} = 0.25$$

$$2^{-3} = 0.125$$

- a) $-7 * 2^{129}$
- b) -3.5
- c) -3.75
- d) -7
- e) -7.5

Quiz Answer

What is the decimal equivalent of:

1	1000 0001	111 0000 0000 0000 0000 0000
S	Exponent	Significand

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

$$(-1)^1 \times (1 + .111) \times 2^{(129-127)}$$

$$-1 \times (1.111) \times 2^2$$

$$-111.1$$

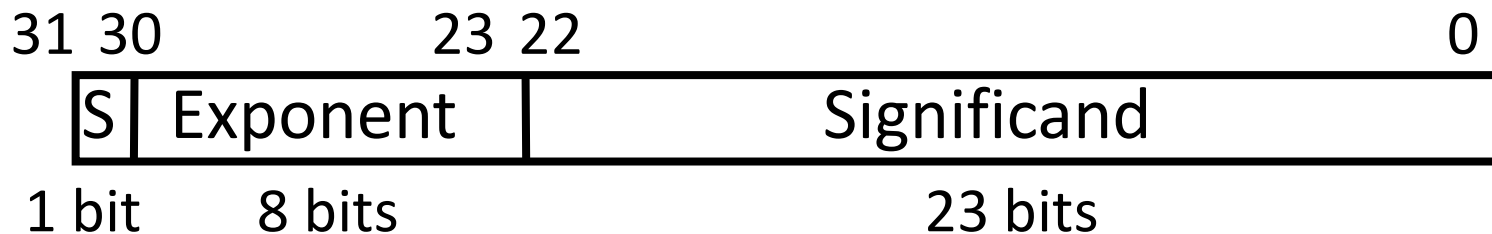
$$111.1 \Rightarrow 4 + 2 + 1 + .5 \Rightarrow 7.5$$

- a) $-7 * 2^{129}$
- b) -3.5
- c) -3.75
- d) -7
- e) **-7.5**

Summary

- Floating Point lets us:
 - Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
 - Store **approximate** values for very large and very small #s.
- **IEEE 754 Floating Point Standard** is most widely accepted attempt to standardize interpretation of such numbers
 - Every desktop or server computer sold since ~1997 follows these conventions

single precision:



- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
 - Double precision identical, except with exponent bias of 1023 (half, quad similar)

Example: Representing 1/3 in MIPS

- 1/3

$$= 0.33333..._{10}$$

$$= 0.25 + 0.0625 + 0.015625 + 0.00390625 + \dots$$

$$= 1/4 + 1/16 + 1/64 + 1/256 + \dots$$

$$= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \dots$$

$$= 0.0101010101..._2 * 2^0$$

$$= \boxed{1}.0101010101..._2 * 2^{-2}$$

– Sign: 0

– Exponent = $-2 + 127 = 125 = 01111101$

– Significand = 0101010101...

0	0111 1101	0101 0101 0101 0101 0101 010
---	-----------	------------------------------

Floating Point Fallacy

- FP add associative?

- $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$

- $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0$
 $= (0.0) + 1.0 = \underline{1.0}$

- $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0)$
 $= -1.5 \times 10^{38} + (1.5 \times 10^{38}) = \underline{0.0}$

- Therefore, Floating Point add is NOT associative!

- Why?

- » FP result approximates real result!

- » This example: 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}

Precision and Accuracy

Don't confuse these two terms!

Precision is a count of the number bits in a computer word used to represent a value.

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 significant (highly precise), but is only an approximation (not accurate).

Representation for $\pm \infty$

- In FP, divide by 0 should produce $\pm \infty$, not overflow.
- Why?
 - OK to do further computations with ∞
 - » E.g., $X/0 > Y$ may be a valid comparison
 - Ask math majors
- IEEE 754 represents $\pm \infty$
 - **Most positive** exponent reserved for ∞
 - Significands **all zeroes**



Representation for 0

- Represent 0?
 - exponent all zeroes
 - significand all zeroes
 - What about sign?
 - » Both cases valid.

+0 : 0 00000000 00000000000000000000000000000000

-0 : 1 00000000 00000000000000000000000000000000

Special Numbers

- What have we defined so far?

- (Single Precision)

Exponent	Significand	Object
0	0	0
0	Nonzero	???
1-254	Anything	+/- FP #
255	0	+/- ∞
255	Nonzero	???

- “Waste not, want not”
 - We’ll talk about Exp = 0 or 255, and Significand != 0 next

Representation for "Not a Number"

- What do you get if you calculate `sqrt(-4.0)` or `0/0`?
 - If ∞ is not an error, these shouldn't be either
 - Called **Not a Number (NaN)**
 - Exponent = 255, Significand nonzero
- Why is this useful?
 - Hope NaNs help with debugging
 - They contaminate: `op(NaN, X) = NaN`

Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0

– Smallest representable positive number:

$$a = 1.0..._2 * 2^{-126} = 2^{-126} \text{ (exp = 1, sig = 0)}$$

– Second smallest representable pos num:

$$b = 1.000.....1_2 * 2^{-126} \text{ (exp = 1, sig = 1)}$$

$$= (1 + 0.00...1_2) * 2^{-126}$$

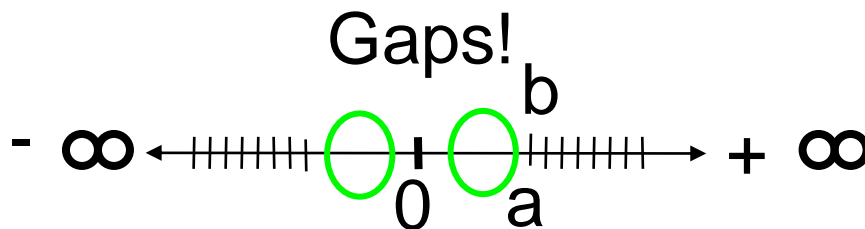
$$= (1 + 2^{-23}) * 2^{-126}$$

$$= 2^{-126} + 2^{-149}$$

$$a - 0 = 2^{-126}$$

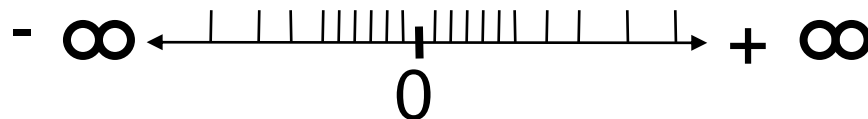
$$b - a = 2^{-149}$$

Normalization and implicit 1
is to blame!



Representation for Denorm (2/2)

- Solution:
 - We still haven't used Exponent = 0, Significand nonzero
 - Denormalized number: no (implied) leading 1, implicit exponent = -126 ($0 - 127 + 1$)
 - » $(-1)^s \times (\text{Significand}) \times 2^{(-126)}$
 - Smallest representable pos num:
 - » $a = 2^{-149}$ (sig = 1)
 - Second smallest representable pos num:
 - » $b = 2^{-148}$ (sig = 2)



Special Numbers Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	Nonzero	Denorm
1-254	Anything	+/- FP #
255	0	+/- ∞
255	Nonzero	NaN

Rounding

- When we perform math on real numbers, we have to worry about rounding to fit the result in the significant field.
- The FP hardware carries two extra bits of precision, and then round to get the proper value
- Rounding also occurs when converting:
 - double to a single precision value, or floating point number to an integer

IEEE FP Rounding Modes

- Halfway between two floating point values (rounding bits read 10)? Choose from the following:
 - Round towards $+\infty$
 - » Round “up”: $1.01 \underline{10} \rightarrow 1.10$, $-1.01 \underline{10} \rightarrow -1.01$
 - Round towards $-\infty$
 - » Round “down”: $1.01 \underline{10} \rightarrow 1.01$, $-1.01 \underline{10} \rightarrow -1.10$
- Truncate
 - Just drop the extra bits (round towards 0)
- **Unbiased (default mode)**. Round to nearest EVEN number
 - Half the time we round up on tie, the other half time we round down. Tends to balance out inaccuracies.
 - In binary, even means least significant bit is 0.
- Otherwise, not halfway (00, 01, 11)! Just round to the nearest float.

Casting floats to ints and vice versa

(int) floating_point_expression

Coerces and converts it to the nearest integer

(C uses truncation)

```
i = (int) (3.14159f);
```

(float) integer_expression

Converts integer to nearest floating point

```
f = f + (float) i;
```

int → float → int

```
if (i == (int) ((float) i)) {  
    printf("true");  
}
```

- **Will not** always print “true”
- Most large values of integers don’t have exact floating point representations!
- What about double?

float → int → float

```
if (f == (float) ((int) f)) {  
    printf("true");  
}
```

- Will not always print “true”
- Small floating point numbers (<1) don’t have integer representations
- For other numbers, rounding errors

Quiz 1:

1. Converting float -> int -> float produces same float number
2. Converting int -> float -> int produces same int number
3. FP add is associative:
 $(x+y)+z = x+(y+z)$

ABC

1: FFF

2: FFT

3: FTF

4: FTT

5: TFF

Quiz 1:

1. Converting float -> int -> float produces same float number
2. Converting int -> float -> int produces same int number
3. FP add is associative:
 $(x+y) + z = x + (y+z)$

1. 3.14 -> 3 -> 3
2. 32 bits for signed int,
but 24 for FP mantissa?
3. x = biggest pos #,
y = -x, z = 1 (x != inf)

ABC
1: FFF
2: FFT
3: FTF
4: FTT
5: TFF

Quiz 2:

- Let $f(1, 2)$ = # of floats between 1 and 2
- Let $f(2, 3)$ = # of floats between 2 and 3

1: $f(1,2) <$

$f(2,3)$

2: $f(1,2) =$

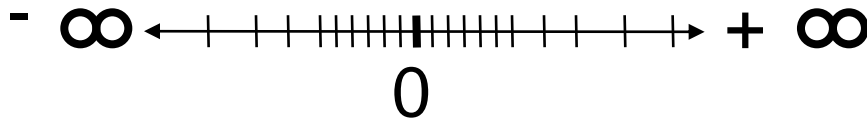
$f(2,3)$

3: $f(1,2) >$

$f(2,3)$

Quiz 2:

- Let $f(1, 2) = \#$ of floats between 1 and 2
- Let $f(2, 3) = \#$ of floats between 2 and 3



1: $f(1,2) <$

$f(2,3)$

2: $f(1,2) =$

$f(2,3)$

3: $f(1,2) >$

$f(2,3)$

Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	Nonzero	Denorm
1-254	Anything	+/- FP #
255	0	+/- ∞
255	Nonzero	NaN

- 4 Rounding modes (default: unbiased)
- MIPS Floating operations complicated, expensive