

CS 61C:
Great Ideas in Computer Architecture
Performance and Floating Point Arithmetic

Instructors:

Bernhard Boser & Randy H. Katz

<http://inst.eecs.berkeley.edu/~cs61c/>

Outline

- Defining Performance
- Floating Point Standard
- And in Conclusion ...

What is Performance?

- *Latency* (or *response time* or *execution time*)
 - Time to complete one task
- *Bandwidth* (or *throughput*)
 - Tasks completed per unit time

Cloud Performance:

Why Application Latency Matters

Server Delay (ms)	Increased time to next click (ms)	Queries/ user	Any clicks/ user	User satisfaction	Revenue/ User
50	--	--	--	--	--
200	500	--	-0.3%	-0.4%	--
500	1200	--	-1.0%	-0.9%	-1.2%
1000	1900	-0.7%	-1.9%	-1.6%	-2.8%
2000	3100	-1.8%	-4.4%	-3.8%	-4.3%

Figure 6.10 Negative impact of delays at Bing search server on user behavior [Brutlag and Schurman 2009].

- Key figure of merit: application responsiveness
 - Longer the delay, the fewer the user clicks, the less the user happiness, and the lower the revenue per user

Clicker/Peer Instruction

System	Rate (Task 1)	Rate (Task 2)
A	10	20
B	20	10

Which system is faster?

A: System A

B: System B

C: Same performance

D: Unanswerable question!

... Depends Who's Selling ...

System	Rate (Task 1)	Rate (Task 2)	Average
A	10	20	15
B	20	10	15

Average throughput

System	Rate (Task 1)	Rate (Task 2)	Average
A	0.50	2.00	1.25
B	1.00	1.00	1.00

Throughput relative to B

System	Rate (Task 1)	Rate (Task 2)	Average
A	1.00	1.00	1.00
B	2.00	0.50	1.25

Throughput relative to A

CPU Performance Factors

- To distinguish between processor time and I/O, *CPU time* is time spent in processor
- $\text{CPU Time/Program} = \text{Clock Cycles/Program} \times \text{Clock Cycle Time}$
- Or
 $\text{CPU Time/Program} = \text{Clock Cycles/Program} \div \text{Clock Rate}$

CPU Performance Factors

- A program executes instructions
- CPU Time/Program
= Clock Cycles/Program x Clock Cycle Time
= Instructions/Program
x Average Clock Cycles/Instruction
x Clock Cycle Time
- 1st term called *Instruction Count*
- 2nd term abbreviated *CPI* for average ***Clock Cycles Per Instruction***
- 3rd term is 1 / Clock rate

Restating Performance Equation

- Time = $\frac{\text{Seconds}}{\text{Program}}$
 $= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$

Important Equation!

What Affects Each Component?

Instruction Count, CPI, Clock Rate

Hardware or software component?	Affects What?
Algorithm	Instruction Count, CPI
Programming Language	Instruction Count, CPI
Compiler	Instruction Count, CPI
Instruction Set Architecture	Instruction Count, Clock Rate, CPI

Clickers/Peer Instruction

Computer	Clock frequency	Clock cycles per instruction	#instructions per program
A	1GHz	2	1000
B	2GHz	5	800
C	500MHz	1.25	400
D	5GHz	10	2000

- Which computer has the highest performance for a given program?

Clicker/Peer Instruction

- Computer A clock cycle time 250 ps, $CPI_A = 2$
Computer B clock cycle time 500 ps, $CPI_B = 1.2$
Assume A and B have same instruction set
Which statement is true?

A: Computer A is ≈ 1.2 times faster than B

B: Computer A is ≈ 4.0 times faster than B

C: Computer B is ≈ 1.7 times faster than A

D: Computer B is ≈ 3.4 times faster than A

Workload and Benchmark

- *Workload*: Set of programs run on a computer
 - Actual collection of applications run or made from real programs to approximate such a mix
 - Specifies both programs and relative frequencies
- *Benchmark*: Program selected for use in comparing computer performance
 - Benchmarks form a workload
 - Usually standardized so that many use them

SPEC

(System Performance Evaluation Cooperative)

- Computer Vendor cooperative for benchmarks, started in 1989
- SPECCPU2006
 - 12 Integer Programs
 - 17 Floating-Point Programs
- Often turn into number where bigger is faster
- *SPECratio*: reference execution time on old reference computer divide by execution time on new computer to get an effective speed-up

SPECINT2006 on AMD Barcelona

Description	Instruction Count (B)	CPI	Clock cycle time (ps)	Execution Time (s)	Reference Time (s)	SPEC-ratio
Interpreted string processing	2,118	0.75	400	637	9,770	15.3
Block-sorting compression	2,389	0.85	400	817	9,650	11.8
GNU C compiler	1,050	1.72	400	724	8,050	11.1
Combinatorial optimization	336	10.0	400	1,345	9,120	6.8
Go game	1,658	1.09	400	721	10,490	14.6
Search gene sequence	2,783	0.80	400	890	9,330	10.5
Chess game	2,176	0.96	400	837	12,100	14.5
Quantum computer simulation	1,623	1.61	400	1,047	20,720	19.8
Video compression	3,102	0.80	400	993	22,130	22.3
Discrete event simulation library	587	2.94	400	690	6,250	9.1
Games/path finding	1,082	1.79	400	773	7,020	9.1
XML parsing	1,058	2.70	400	1,143	6,900	6.0

Summarizing SPEC Performance

- Varies from 6x to 22x faster than reference computer

- *Geometric mean* of ratios:
N-th root of product
of N ratios

$$\sqrt[n]{\prod_{i=1}^n \text{Execution time ratio}_i}$$

- Geometric Mean gives same relative answer no matter what computer is used as reference
- Geometric Mean for Barcelona is 11.7

Administrivia

- Midterm, November 1, in class, 3:40-5:00 PM
- See Piazza for breakout room assignments by class account
- DSP students: same times and places as Midterm I
- Proj 3-1 fair game, Proj 3-2 not on exam
- Exam Review Sunday, 1-3 PM, in 155 Dwinelle



Outline

- Defining Performance
- **Floating Point Standard**
- And in Conclusion ...

Quick Number Review

- Computers deal with numbers
- What can we represent in N bits?
 - 2^N things, and no more! They could be...
 - Unsigned integers:
0 to $2^N - 1$
(for $N=32$, $2^N - 1 = 4,294,967,295$)
 - Signed Integers (Two's Complement)
 $-2^{(N-1)}$ to $2^{(N-1)} - 1$
(for $N=32$, $2^{(N-1)} = 2,147,483,648$)

Other Numbers

1. Very large numbers? (seconds/millennium)

☞ $31,556,926,000_{10}$ ($3.1556926_{10} \times 10^{10}$)

2. Very small numbers? (Bohr radius)

☞ $0.0000000000529177_{10}\text{m}$ ($5.29177_{10} \times 10^{-11}$)

3. Numbers with both integer & fractional parts?

☞ 1.5

- *First consider #3*
- *...our solution will also help with #1 and #2*

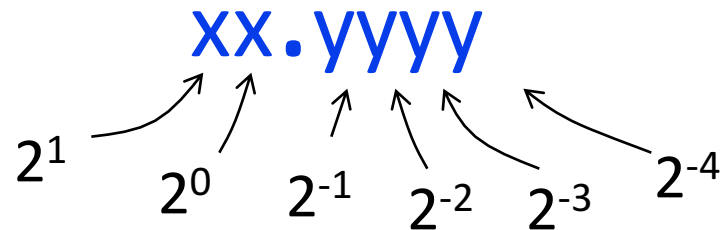
Goals for Floating Point

- Standard arithmetic for reals for all computers
 - Like two's complement
- Keep as much precision as possible in formats
- Help programmer with errors in real arithmetic
 - $+\infty$, $-\infty$, Not-A-Number (NaN), exponent overflow, exponent underflow
- Keep encoding that is somewhat compatible with two's complement
 - E.g., 0 in Fl. Pt. is 0 in two's complement
 - Make it possible to sort without needing to do floating point comparison

Representation of Fractions

- “Binary Point” like decimal point signifies boundary between integer and fractional parts:

Example 6-bit representation:



$$10.1010_{\text{two}} = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{\text{ten}}$$

If we assume “fixed binary point”, range of 6-bit representations with this format: 0 to 3.9375 (almost 4)

Fractional Powers of 2

i	2^{-i}	
0	1.0	1
1	0.5	1/2
2	0.25	1/4
3	0.125	1/8
4	0.0625	1/16
5	0.03125	1/32
6	0.015625	
7	0.0078125	
8	0.00390625	
9	0.001953125	
10	0.0009765625	
11	0.00048828125	
12	0.000244140625	
13	0.0001220703125	
14	0.00006103515625	
15	0.000030517578125	

Representation of Fractions with Fixed Pt.

What about addition and multiplication?

Addition is
straightforward:

$$\begin{array}{r}
 01.100 \quad 1.5_{\text{ten}} \\
 + 00.100 \quad 0.5_{\text{ten}} \\
 \hline
 10.000 \quad 2.0_{\text{ten}}
 \end{array}$$

Multiplication a bit more complex:

$$\begin{array}{r}
 01.100 \quad 1.5_{\text{ten}} \\
 00.100 \quad 0.5_{\text{ten}} \\
 \hline
 00 \quad 000 \\
 000 \quad 00 \\
 0110 \quad 0 \\
 00000 \\
 00000 \\
 \hline
 0000110000
 \end{array}$$

Where's the answer, **0.11**? (e.g., $0.5 + 0.25$;
Need to remember where point is!)

Representation of Fractions


Our examples 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_{ten} into binary. Represent with 5-bits choosing where to put the binary point.

... 000000.001010100000...

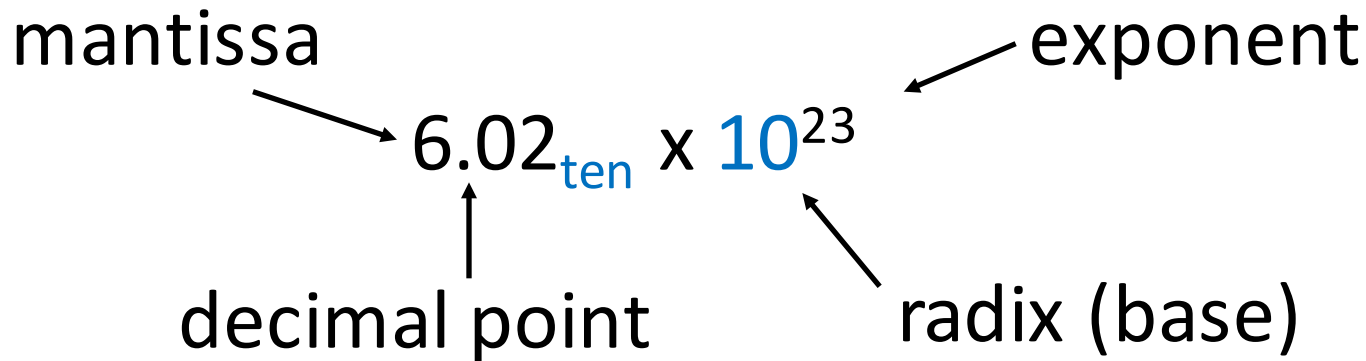


Store these bits and keep track of the binary
point 2 places to the left of the MSB

Any other solution would lose accuracy!

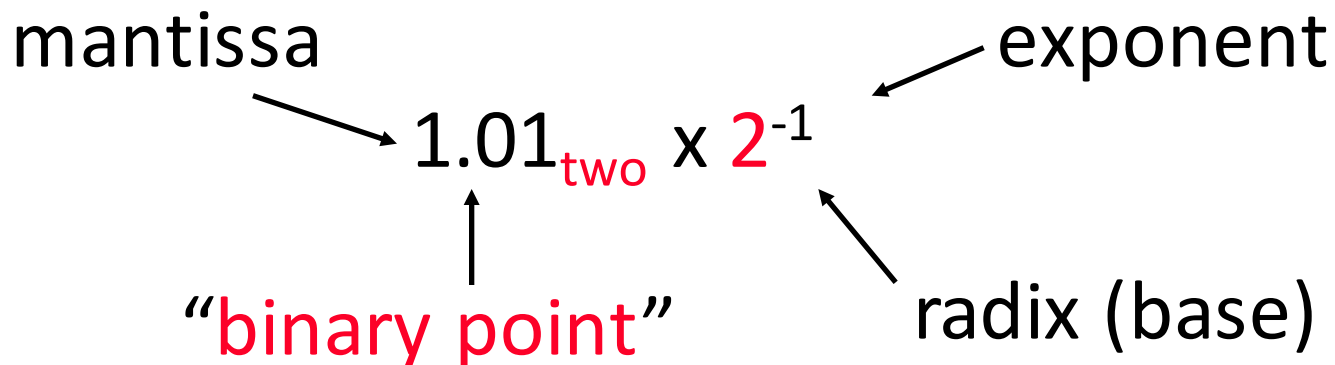
- With floating-point representation, each numeral carries an exponent field recording the whereabouts of its binary point
- Binary point **can be outside** the stored bits, so very large and small numbers can be represented

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 is 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
 - `double` for double precision

Floating Point Representation (1/4)

- 32-bit word has 2^{32} patterns, so must be approximation of real numbers ≥ 1.0 , < 2
- IEEE 754 Floating Point Standard:
 - 1 bit for *sign* (s) of floating point number
 - 8 bits for *exponent* (E)
 - 23 bits for *fraction* (F)
(get 1 extra bit of precision if leading 1 is implicit)

$$(-1)^s \times (1 + F) \times 2^E$$

- Can represent from 2.0×10^{-38} to 2.0×10^{38}

Floating Point Representation (2/4)

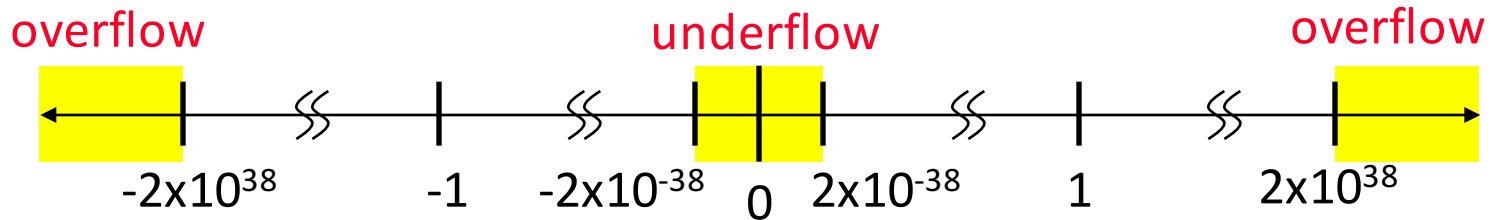
- Normal format: $+1.\text{xxx}\dots\text{x}_{\text{two}} * 2^{\text{yyy}\dots\text{y}_{\text{two}}}$
- Multiple of Word Size (32 bits)



- S represents Sign
Exponent represents y's
Significand represents x's
- Represent numbers as small as $2.0_{\text{ten}} \times 10^{-38}$ to as large as $2.0_{\text{ten}} \times 10^{38}$

Floating Point Representation (3/4)

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



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

Floating Point Representation (4/4)

- What about bigger or smaller numbers?
 - IEEE 754 Floating Point Standard:
Double Precision (64 bits)
 - 1 bit for *sign* (*s*) of floating point number
 - 11 bits for *exponent* (*E*)
 - 52 bits for *fraction* (*F*)
(get 1 extra bit of precision if leading 1 is implicit)
- $$(-1)^s \times (1 + F) \times 2^E$$
- Can represent from 2.0×10^{-308} to 2.0×10^{308}
 - 32 bit format called *Single Precision*

Which is Less? (i.e., closer to $-\infty$)

- **0** vs. 1×10^{-127} ?
- 1×10^{-126} vs. **1×10^{-127}** ?
- **-1×10^{-127}** vs. 0?
- **-1×10^{-126}** vs. -1×10^{-127} ?

Floating Point: Representing Very Small Numbers

- Zero: Bit pattern of all 0s is encoding for 0.000
 - \Rightarrow But 0 in exponent should mean most negative exponent (want 0 to be next to smallest real)
 - \Rightarrow Can't use two's complement ($1000\ 0000_{\text{two}}$)
- *Bias notation*: subtract bias from exponent
 - Single precision uses bias of 127; DP uses 1023
- 0 uses $0000\ 0000_{\text{two}} \Rightarrow 0 - 127 = -127$;
 ∞ , NaN uses $1111\ 1111_{\text{two}} \Rightarrow 255 - 127 = +128$
 - Smallest SP real can represent: $1.00\dots00 \times 2^{-126}$
 - Largest SP real can represent: $1.11\dots11 \times 2^{+127}$

IEEE 754 Floating-Point Standard (1/3)

Single Precision (Double Precision similar):



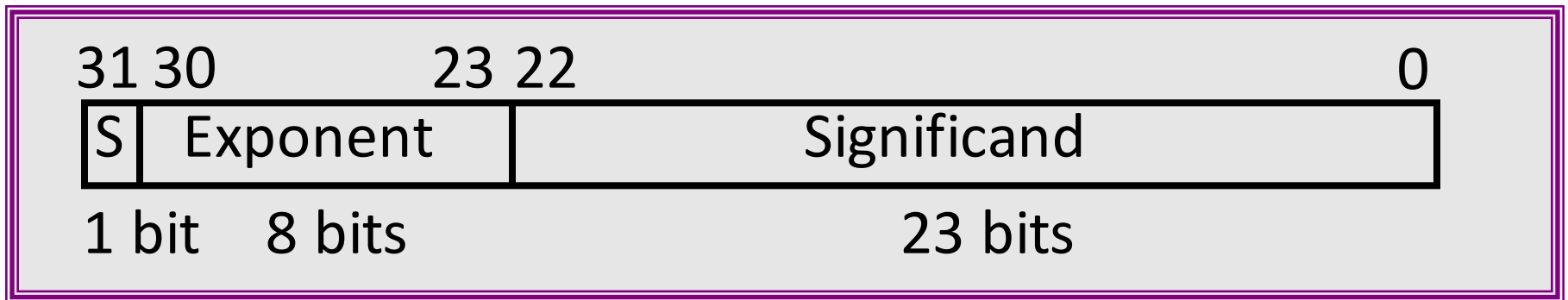
- Sign bit: 1 means negative
 0 means positive
- Significand in *sign-magnitude* format (not 2's complement)
 - To pack more bits, leading 1 implicit for normalized numbers
 - 1 + 23 bits single, 1 + 52 bits double
 - always true: $0 < \text{Significand} < 1$ (for normalized numbers)
- Note: 0 has no 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)
 - Use just magnitude and offset by half the range

IEEE 754 Floating Point Standard (3/3)

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

Bias Notation (+127)

How it is interpreted

How it is encoded

Decimal Exponent	signed 2's complement	Biased Notation	Decimal Value of Biased Notation
For infinities		11111111	255
127	01111111	11111110	254
...
2	00000010	10000001	129
1	00000001	10000000	128
0	00000000	01111111	127
-1	11111111	01111110	126
-2	11111110	01111101	125
...
-126	10000010	00000001	1
For Denorms	10000001	00000000	0

∞ , NaN

Getting closer to zero

Zero

Clickers/Peer Instruction

- Guess this Floating Point number:

1 1000 0000 1000 0000 0000 0000 0000 000

A: -1×2^{128}

B: $+1 \times 2^{-128}$

C: -1×2^1

D: $+1.5 \times 2^{-1}$

E: -1.5×2^1

More Floating Point

- What about 0?
 - Bit pattern all 0s means 0, so no implicit leading 1
- What if divide 1 by 0?
 - Can get infinity symbols $+\infty$, $-\infty$
 - Sign bit 0 or 1, largest exponent, 0 in fraction
- What if do something stupid? ($\infty - \infty$, $0 \div 0$)
 - Can get special symbols NaN for Not-a-Number
 - Sign bit 0 or 1, largest exponent, not zero in fraction
- What if result is too big? ($2 \times 10^{308} \times 2 \times 10^2$)
 - Get *overflow* in exponent, alert programmer!
- What if result is too small? ($2 \times 10^{-308} \div 2 \times 10^2$)
 - Get *underflow* in exponent, alert programmer!

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
- 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	<u>nonzero</u>	<u>???</u>
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	<u>nonzero</u>	<u>???</u>

Representation for Not-a-Number

- What do I get if I calculate `sqrt(-4.0)` or `0/0`?
 - If ∞ 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`
 - Can use the significand to identify which! (e.g., quiet NaNs and signaling NaNs)

Representation for Denorms (1/2)

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

$$a = 1.0..._2 * 2^{-126} = 2^{-126}$$
 - Second smallest representable positive number:

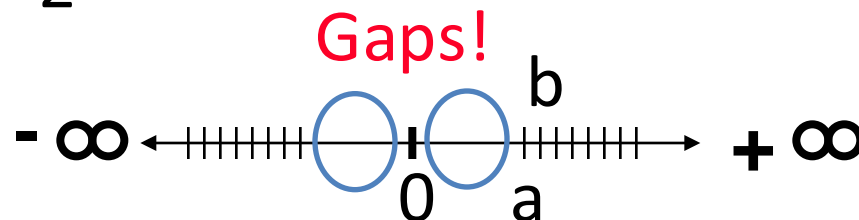
$$b = 1.000.....1_2 * 2^{-126}$$

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

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

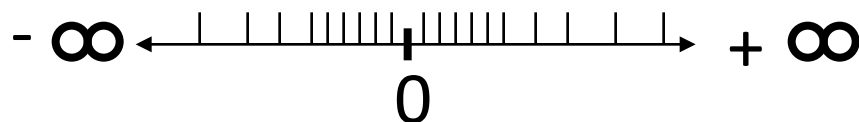
$$= 2^{-126} + 2^{-149}$$
- Normalization and implicit 1 is to blame!
- $a - 0 = 2^{-126}$
- $b - a = 2^{-149}$

Normalization
and implicit 1
is to blame!



Representation for Denorms (2/2)

- Solution:
 - We still haven't used Exponent = 0, Significand nonzero
 - DEnormalized number: no (implied) leading 1, **implicit exponent = -126**
 - Smallest representable positive number:
 $a = 2^{-149}$ (i.e., $2^{-126} \times 2^{-23}$)
 - Second smallest representable positive number:
 $b = 2^{-148}$ (i.e., $2^{-126} \times 2^{-22}$)



Special Numbers Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	<u>nonzero</u>	<u>Denorm</u>
1-254	anything	+/- fl. pt. #
255	<u>0</u>	<u>+/- ∞</u>
255	<u>Nonzero</u>	<u>NaN</u>

Outline

- Defining Performance
- Floating Point Standard
- And in Conclusion ...

And In Conclusion, ...

- Time (seconds/program) is measure of performance
$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$
- Floating-point representations hold approximations of real numbers in a finite number of bits
 - 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:

