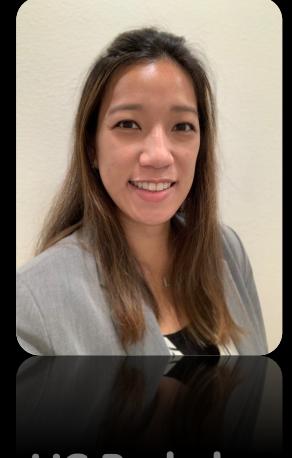




UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture (a.k.a. Machine Structures)



UC Berkeley
Teaching Professor
Lisa Yan

Apple + EECS Faculty Office Hours

Information Session | September 8 | 12-1 p.m. | Soda Hall, Wozniak Lounge

Sponsor: Electrical Engineering and Computer Sciences (EECS).

You're invited to a special event with EECS and Apple

Learn about Berkeley and Apple's joint initiative in hardware education

Interested in VLSI or chip design? Circuit design or computer architecture? Want to know what's next?

Join Apple engineers and Berkeley faculty to help guide you in these exciting fields!

Floating Point



Today's EECS Colloquium: David Patterson

A Decade of Machine Learning Accelerators: Lessons Learned and Carbon Footprint

Abstract: The success of deep neural networks (DNNs) from Machine Learning (ML) has inspired domain specific architectures (DSAs) for them. Google's first generation DSA offered 50x improvement over conventional architectures for ML inference in 2015. Google next built the first production DSA supercomputer for the much harder problem of training. Subsequent generations greatly improved performance of both phases. We start with ten lessons learned from such efforts.

The rapid growth of DNNs rightfully raised concerns about their carbon footprint. The second part of the talk identifies the "4Ms" (Model, Machine, Mechanization, Map) that, if optimized, can reduce ML training energy by up to 100x and carbon emissions up to 1000x. By improving the 4Ms, ML held steady at <15% of Google's total energy use despite it consuming ~75% of its floating point operations. With continuing focus on the 4Ms, we can realize the amazing potential of ML to positively impact many fields in a sustainable way.

Wednesday, September 07, 2022

4:00 - 5:00 pm

(3:30– 4:00pm: Light Refreshments)

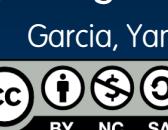
306 Soda Hall (HP Auditorium)

Webinar:

<https://eecs.berkeley.edu/research/colloquium>



David Patterson
EECS Pardee Emeritus, UC Berkeley
Distinguished Engineer, Google



Basics, Fixed Point

- Basics, Fixed Point
- Floating Point
- Special Numbers
- Other Floating Point Representations
- (Bonus) More Examples

Quote of the Day

- “95% of the folks out there are completely **clueless** about floating-point.”

– James Gosling, 1998-02-28

- Sun Fellow
- Java Inventor



Garcia, Yan

Review of Integer Number Representations

- Computers made to process numbers, represented as bits.
- What can we represent in N bits?
 - 2^N things, and no more! We've covered a few conventions...
 - Unsigned integers:
 - Range: 0 to $2^N - 1$
 - N=32: $2^N - 1 = 4,294,967,295$
 - Signed Integers (Two's Complement)
 - Range: $-2^{(N-1)}$ to $2^{(N-1)} - 1$
 - N=32: $2^{(N-1)} - 1 = 2,147,483,647$

= ~~無符号整数~~
有符号整数

What about other numbers?

- **Very large numbers (sec/millennium)**

- 31,556,926,010 ($3.155692610 \times 10^{10}$)

Scientific Notation

- **Very small numbers? (Bohr radius)**

- 0.00000000052917710 ($5.2917710 \times 10^{-11}$)

- **#s with both integer & fractional parts?**

- 2.625



Let's start with representing this number...

“Fixed Point” Binary Representation

- “**Binary Point**,” like decimal point, signifies the boundary between integer and fractional parts.
- Example 6-bit representation, positive numbers only:

101010 → 1 0 . 1 0 1 0
2¹ 2⁰ 2⁻¹ 2⁻² 2⁻³ 2⁻⁴

$$\begin{aligned} & 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} \\ & = 2 + 0.5 + 0.125 \\ & = 2.625_{\text{ten}} \end{aligned}$$

- With the above 6-bit “fixed binary point” representation:

- Range: 0 to

3.9375

$$11.1111_{\text{two}} = 4 - 1 \times 2^{-4}$$

乘2取整

Arithmetic with Fixed Point

- Addition is straightforward:

$$\begin{array}{r} 01.1000 \\ + 00.1000 \\ \hline 10.0000 \end{array} \quad \begin{array}{l} 1.5_{10} \\ 0.5_{10} \\ \hline 2.0_{10} \end{array}$$

- Multiplication is a bit more complex:

$$\begin{array}{r} 01.1000 \\ \times 00.1000 \\ \hline 000000 \\ 000000 \\ 011000 \\ 000000 \\ \hline 0001100000 \end{array} \rightarrow 00.1100$$

A white arrow points from the right side of the first column of the multiplication result towards the decimal point in the final answer.

Need to remember where point is...

What about other numbers?

- **Very large numbers (sec/millennium)**

- 31,556,926,010 ($3.155692610 \times 10^{10}$)

Scientific Notation

科学计数法

11...0.0

34 bits

- **Very small numbers? (Bohr radius)**

- 0.00000000052917710 ($5.2917710 \times 10^{-11}$)

0.0...11

58 bits

✓ #s with both integer & fractional pa

- 2.625

To store all these numbers, we'd need a fixed-point rep with at least 92 bits. There must be a better way!

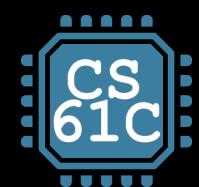
Floating Point

- Basics, Fixed Point
- Floating Point
- Special Numbers
- Other Floating Point Representations
- (Bonus) More Examples

Floating Point

- A “floating binary point” most effectively uses of our limited bits (and thus more accuracy) in our number representation).
- Example:
 - Suppose we wanted to represent 0.1640625_{ten} .
 - Binary representation: ... 000000 . 001010100000...
 - 2. Keep track of the binary point 2 places to the left of the MSB (“exponent field”).
 - 1. Store these “significant bits.”
- The binary point is stored separately from the significant bits, so very large and small numbers can be represented.

最高有效位



Enter Scientific Notation (in Decimal)

- “Normalized form”: no leadings 0s
(exactly one nonzero digit to left of point).

- To represent $3/1,000,000,000$:

- **Normalized:**

 3.0×10^{-9}

- Not normalized:

 $0.3 \times 10^{-8}, 30.0 \times 10^{-10}$

decimal point



1.640625_{ten}



mantissa

radix (base 10)



$\times 10^{-1}$

exponent

Enter Scientific Notation (in Binary)

- “Normalized form”: no leadings 0s (exactly one nonzero digit to left of point).

- To represent 3/1,000,000,000:
 - Normalized: 3.0×10^{-9}
 - Not normalized: $0.3 \times 10^{-8}, 30.0 \times 10^{-10}$

- “Floating point”: Computer arithmetic that supports this binary representation.

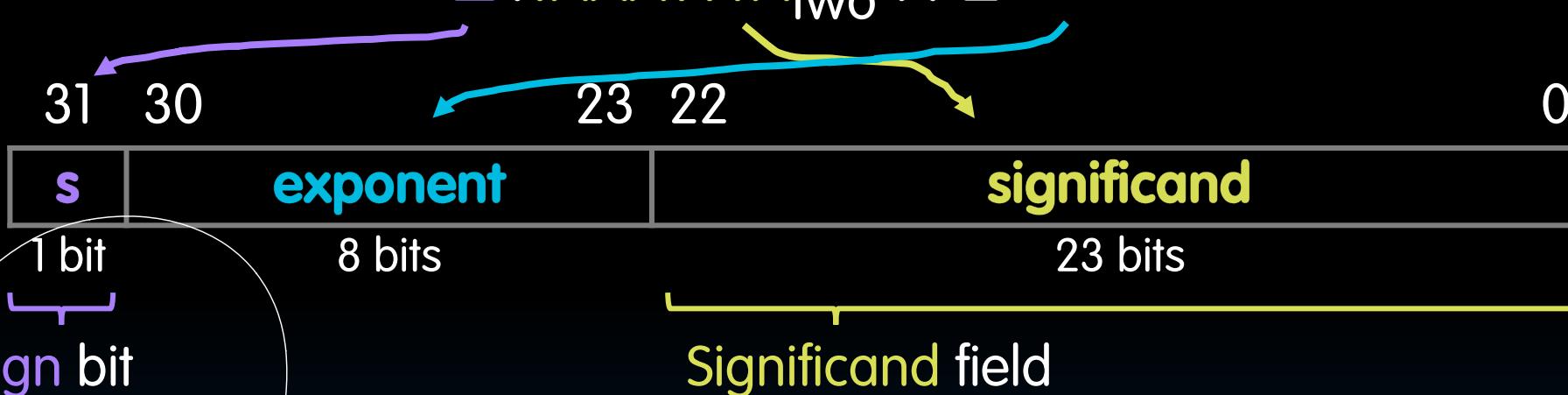
- Represents numbers where the binary point is not fixed (as opposed to integers).
- C variable types: **float**, **double**

Note: Normalized binary representation always leads with a 1! (why?)



IEEE 754 Floating Point Standard (1/3)

- Single precision standard for 32-bit word. In C, `float`.
- Normalized format: $\pm 1.\text{XXX}\dots\text{X}_{\text{two}} \times 2^{\text{yyy}\dots\text{y}_{\text{two}}}$



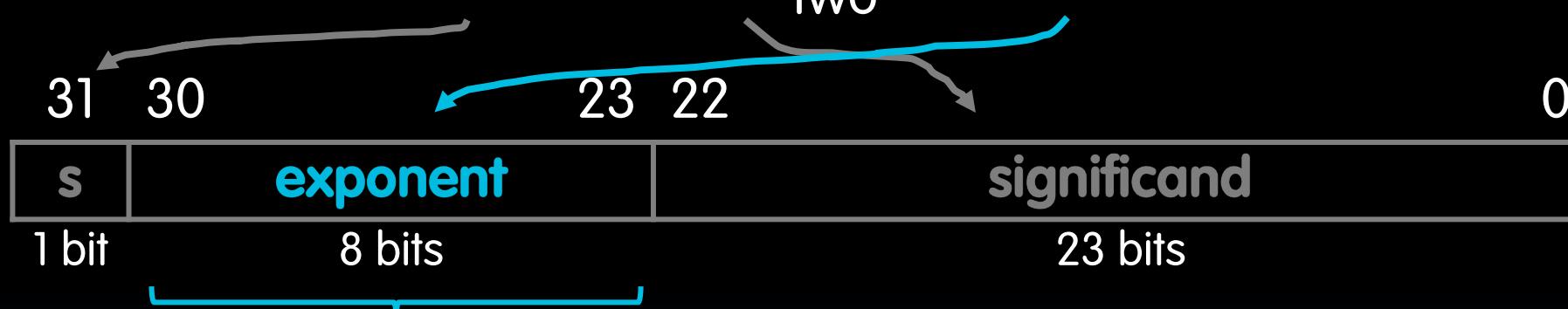
Significand field

- To pack more bits, mantissa leads with **implicit 1**
 - Mantissa = 1 + 23-bit significand field
 - $0 < \text{significand field} < 1$

Sign bit

IEEE 754 Floating Point Standard (2/3)

- Single precision standard for 32-bit word. In C, `float`.
- Normalized format: $\pm 1.\text{XXX}\dots\text{X}_{\text{two}} \times 2^{\text{yyy}\dots\text{y}_{\text{two}}}$



Exponent field uses “bias notation”:

- Bias of 127: Subtract 127 from exponent field to get exponent value.
- Designers wanted FP numbers to be used even without specialized FP hardware, e.g., sort records with FP numbers using integer compares.
- Idea: for the same sign, bigger exponent field should represent bigger numbers.
 - 2's complement poses a problem (because negative numbers look bigger)
 - We'll see numbers ordered EXACTLY as in sign-magnitude $00\dots 00 \rightarrow 11\dots 11$

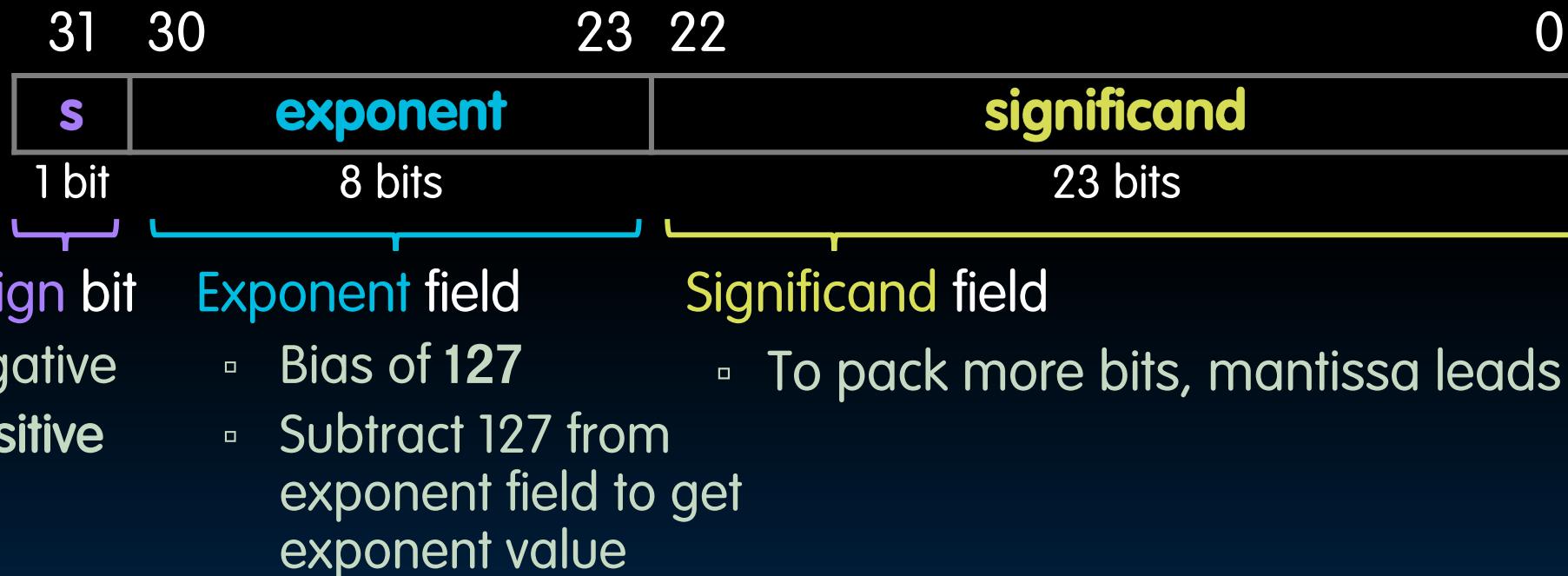
0 to +MAX \rightarrow -0 to -MAX

Garcia, Yan



IEEE 754 Floating Point Standard (3/3)

- Single precision standard for 32-bit word. In C, `float`.
- Normalized format:



$$(-1)^S \times (1 + \text{significand}) \times 2^{(\text{exponent}-127)}$$

How to represent zero?

More later...

“Father” of the Floating Point Standard

- IEEE Standard 754 for Binary Floating-Point Arithmetic.



Prof. William Kahan

www.cs.berkeley.edu/~wkahan/ieee754status/754story.html



Garcia, Yan

Normalized Example

- What is the decimal equivalent of the following IEEE 754 single-precision binary floating point number?

31	30	23	22	0
1	1000 0001	111 0000 0000 0000 0000 0000		

$(-1)^S \times (1 + \text{significand}) \times 2^{(\text{exponent}-127)}$

- A. $-7 * 2^{129}$
- B. -3.5
- C. -3.75
- D. -7
- E. -7.5
- F. Something else

When poll is active, respond at **pollev.com/yanl**

Text **YANL** to **22333** once to join

What is the decimal equivalent of the following IEEE 754 single-precision binary floating point number?

31	30	23	22	0
1	1000 0001	111 0000 0000 0000 0000 0000		

$-7 * 2^{129}$

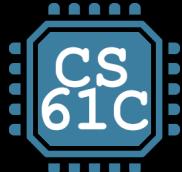
-3.5

-3.75

-7

-7.5

Something else



Normalized Example

IEEE 754

- What is the decimal equivalent of the following IEEE 754 single-precision binary floating point number?

A diagram illustrating the IEEE 754 single precision floating-point format. It shows a 32-bit binary number divided into three fields: sign (s), exponent, and significand. The sign bit is at index 31, the exponent bits are from index 30 to 23, and the significand bits are from index 22 to 0. The value 1 is shown in the sign field, 1000 0001 in the exponent field, and 111 0000 0000 0000 0000 0000 in the significand field.

31	30	23	22	0
1	1000 0001	111 0000 0000 0000 0000 0000		
s	exponent	significand		

$(-1)^S \times (1 + \text{significand}) \times 2^{(\text{exponent}-127)}$

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

$$-1 \times (1.111)_{\text{two}}$$

-111.1_{two}

-7.5_{ten}

- A. $-7 * 2^{129}$
 - B. -3.5
 - C. -3.75
 - D. -7
 - E. -7.5
 - F. Something else

小故上偏承

Step Size

- What is the next representable number after y ? Before y ?



`y + ((.0...01) two x 2^(129-127))`

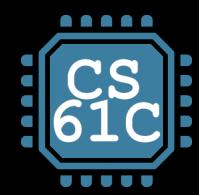
$y + (2^{-23} \times 2^{(2)})$

y + 2⁻²¹

“step size”

Because we have a fixed # of bits, we cannot represent all numbers.
Step size is the spacing between consecutive floats with a given exponent.

- Bigger exponents \rightarrow bigger step size.
 - Smaller exponents \rightarrow smaller step size!



Climate Change

4:21 PM

⋮

Wed, Jan 22



340,282,346,638,5

28,860,000,000,00

0,000,000,000,000°



F

Sunny

San Diego, Wed 4:10 PM

INT_MAX: 2147483647

UINT_MAX: 4294967295

FLOAT_MAX: 3.40282e+38

s	1111 1110	1...1 (23 bits)
---	-----------	-----------------

(1.1...1) × 2⁽²⁵⁴⁻¹²⁷⁾ ≈ 3.4 × 10³⁸

Biased exponents are 1 to 254 (2⁻¹²⁶ to 2¹²⁷). Exponent fields 0, 255 are reserved!

Special Numbers

- Basics, Fixed Point
- Floating Point
- Special Numbers
- Other Floating Point Representations
- (Bonus) More Examples

Representing Zero

- Note: **Zero has no normalized representation (no leading 1).**
- IEEE 754 represents ± 0 :
 - Reserve exponent value **0000 0000**; signals to hardware to not implicitly add 1.
 - Keep significand all zeroes.
 - What about sign? Both cases valid! Why?

+0:	0	0000 0000	000...000
-0:	1	0000 0000	000...000

Garcia, Yan

Special Numbers

- Normalized numbers are only a fraction (heh) of floating point representations. For single-precision:

Biased Exponent	Significand field	Object
0	all zeros	± 0
0	nonzero	???
1 – 254	anything	Normalized floating point
255	all zeros	???
255	nonzero	???

- Professor Kahan had clever ideas;
“Waste not, want not.”

Biased exponent fields 0 and 255 accommodate **overflow**, **underflow**, and arithmetic errors.

Overflow and Underflow

- Because 0 and 255 are reserved exponent fields, the range of normalized single-precision floating point is:
 - Largest number:
 - Smallest number:

s	1111 1110	1...1 (23 bits)
---	-----------	-----------------

$$(1.1\dots1) \times 2^{(254-127)} \approx 3.4 \times 10^{38}$$

s	0000 0001	0...0 (23 bits)
---	-----------	-----------------

$$(1.0) \times 2^{(1-127)} \approx 1.2 \times 10^{-38}$$



Overflow and Underflow

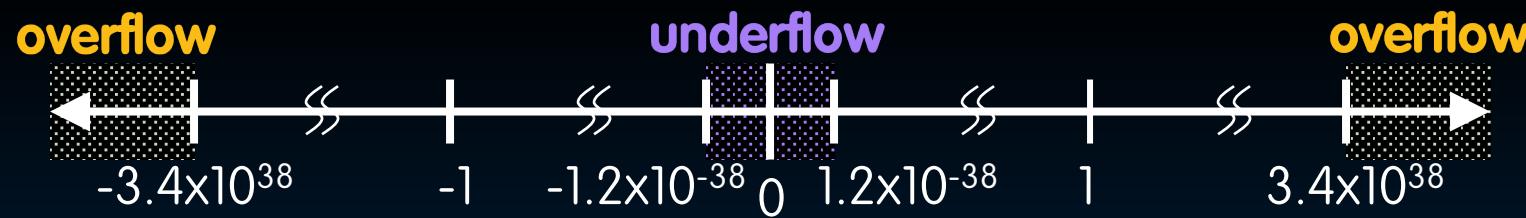
- Because 0 and 255 are reserved exponent fields, the range of normalized single-precision floating point is:
 - Largest number:
 - Smallest number:

s	1111 1110	1...1 (23 bits)
---	-----------	-----------------

$$(1.1\dots1) \times 2^{(254-127)} \approx 3.4 \times 10^{38}$$

s	0000 0001	0...0 (23 bits)
---	-----------	-----------------

$$(1.0) \times 2^{(1-127)} \approx 1.2 \times 10^{-38}$$



- What if a number falls outside this range?
 - Too large: *overflow*. Magnitude of value is too large to represent!
 - Too small: *underflow*. Magnitude of value is too small to represent!

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 $\pm\infty$:
 - Reserve exponent value 1111 1111.
 - And keep significand all zeroes.

+∞:	0	1111 1111	000...000
-∞:	1	1111 1111	000...000

Representation for Not a Number

- **What do I get if I 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?**

- They contaminate: $op(\text{NaN}, X) = \text{NaN}$
 - Hope NaNs help with debugging?
 - Can use the significand to encode/identify where errors occurred!
(proprietary, not defined in standard)



Garcia, Yan

Denorms: Gradual Underflow (1/2)

- Problem:

- There's a **gap** among representable FP numbers around zero!

- Smallest number:

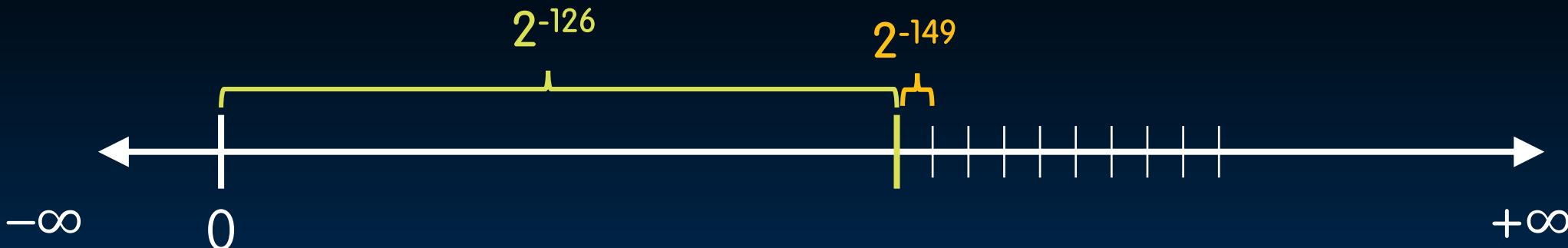
0	0000 0001	000...000
---	-----------	-----------

$$(1.0) \times 2^{(1-127)} = 2^{-126}$$

- 2nd smallest number:

31	30	23	22	1	0
0	0000 0001	00000...0000	1		

$$\begin{aligned}(1.000\dots1_2) \times 2^{(1-127)} \\= (1 + 2^{-23}) \times 2^{-126} = 2^{-126} + 2^{-149}\end{aligned}$$

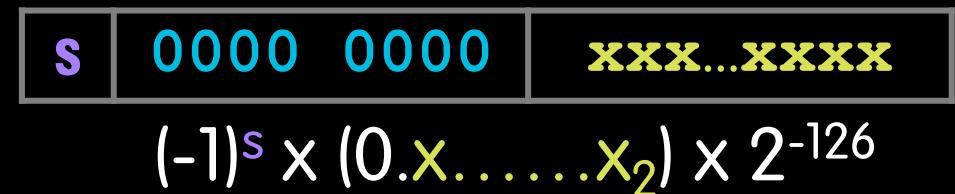


Normalization and implicit 1 is to blame!

Denorms: Gradual Underflow (2/2)

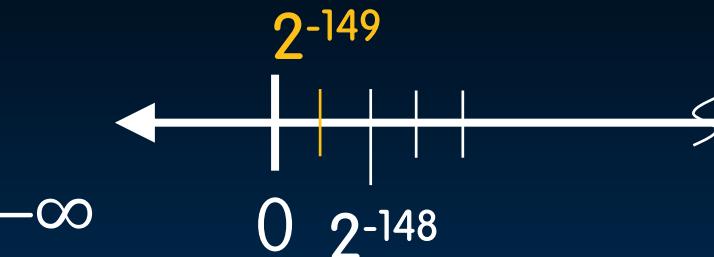
- Solution:

- We still haven't used Exponent = 0, Significand nonzero.
- DEnormalized** number:
 - no (implied) leading 1
 - implicit exponent = -126.



Smallest denormalized number

0	0000 0000	000...001
---	-----------	-----------



0	0000 0000	000...010
---	-----------	-----------

Smallest normalized number

0	0000 0001	000...000
---	-----------	-----------



0	0000 0000	111...111
---	-----------	-----------



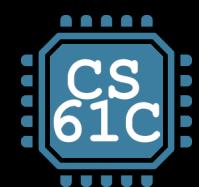
Special Numbers, Summary

Biased Exponent	Significand field	Object
0	all zeros	± 0
0	nonzero	Denormalized numbers
1 – 254	anything	Normalized floating point
255	all zeros	$\pm \infty$
255	nonzero	NaNs

Reserved exponent fields 0 and 255 accommodate **overflow**, **underflow**, and arithmetic errors.

Other Floating Point Representations

- Basics, Fixed Point
- Floating Point
- Special Numbers
- Other Floating Point Representations
- (Bonus) More Examples



Precision and Accuracy

Don't confuse these two terms!

- “Precision” is a count of the number of bits used to represent a value.
- “Accuracy” is 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 23 bits of the significand (highly precise), but it is only an approximation (not accurate).

Double Precision Floating Point

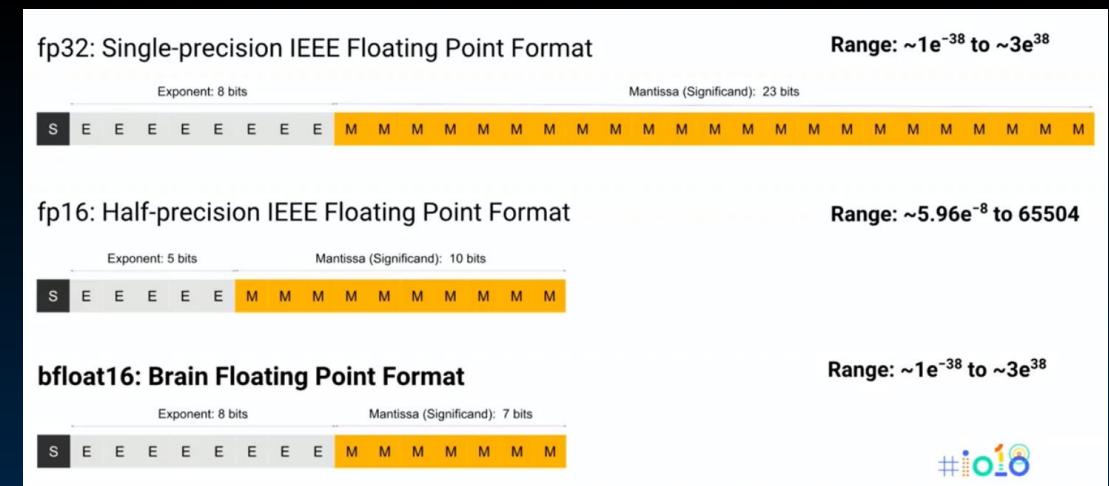
- **binary64: Next Multiple of Word Size (64 bits)**



- **Double Precision (vs. Single Precision)**
 - C variable declared as **double**
 - Exponent bias now 1023.
 - Represent numbers from $\sim 2.0 \times 10^{-308}$ to $\sim 2.0 \times 10^{308}$.
 - The primary advantage is greater **accuracy** due to larger significand.

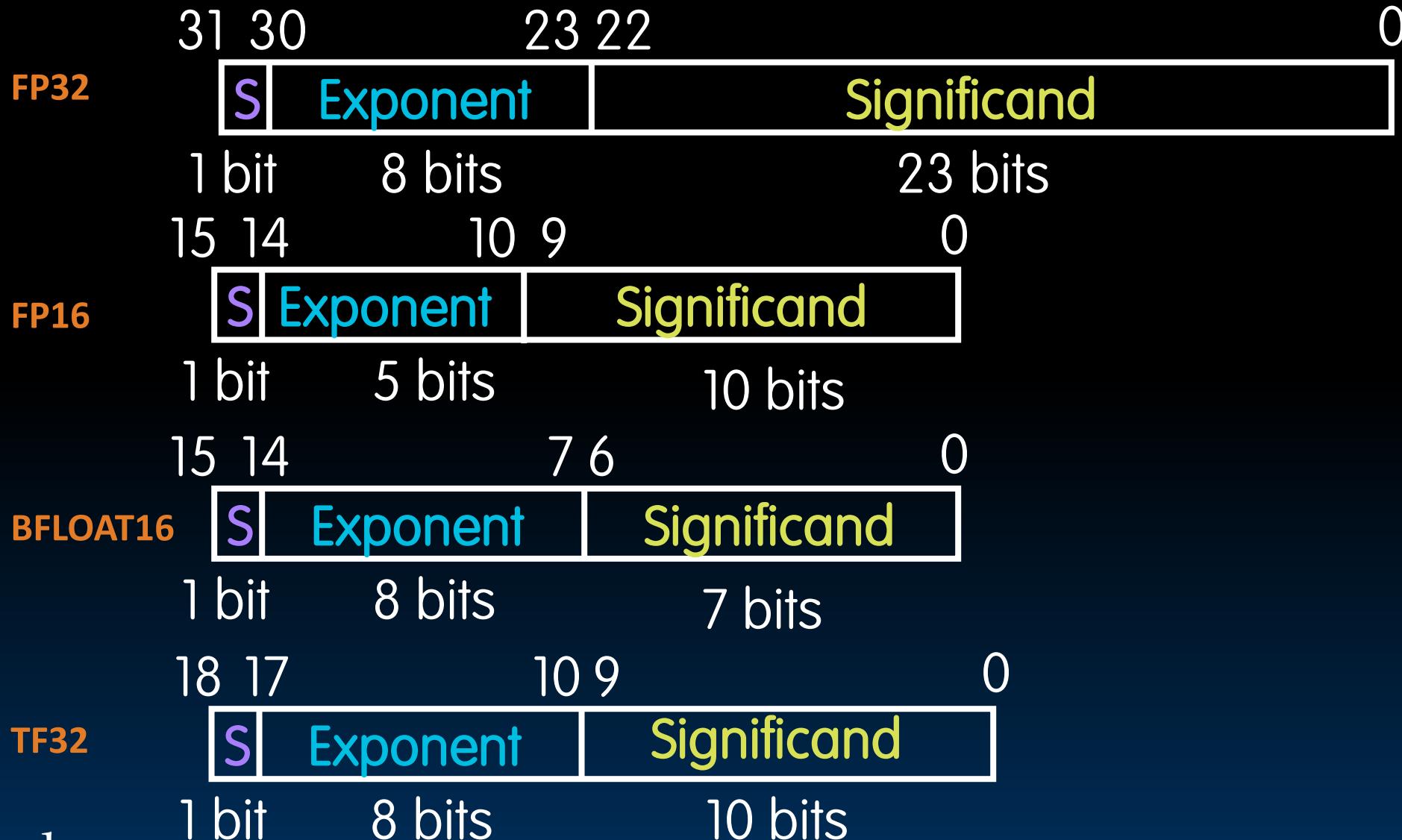
Other Floating Point Representations

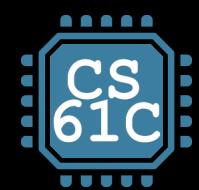
- **Quad-Precision? Yep! (128 bits) “binary128”**
 - Unbelievable range, precision (accuracy)
 - 15 exponent bits, 112 significand bits
- **Oct-Precision? Yep! “binary256”**
 - 19 exponent bits, 236 significant bits
- **Half-Precision? Yep! “binary16” or “fp16”**
 - 1/5/10 bits
- **Half-Precision? Yep! “bfloat16”**
 - Competing with fp16
 - Same range as fp32!
 - Used for faster ML



https://en.wikipedia.org/wiki/Floating-point_arithmetic

Floating Point Soup





Who Uses What in Domain Accelerators?

	int4	int8	int16	fp16	bf16	fp32	tf32
Google TPU v1			x				
Google TPU v2						x	
Google TPU v3						x	
Nvidia Volta TensorCore	x	x			x		
Nvidia Ampere TensorCore	x	x	x	x	x	x	x
Nvidia DLA		x	x	x			
Intel AMX	x					x	
Amazon AWS Inferentia		x		x		x	
Qualcomm Hexagon	x						
Huawei Da Vinci		x		x			
MediaTek APU 3.0	x	x		x			
Samsung NPU	x						
Tesla NPU		x					

- **Everything so far has had a fixed set of bits for Exponent and Significand.**
 - What if the field widths were variable?
 - Also, add a “**u-bit**” to tell whether number is exact or in-between unums.
 - “Promises to be to floating point what floating point is to fixed point.”
- **Claims to save power!**



[https://en.wikipedia.org/wiki/Unum_\(number_format\)](https://en.wikipedia.org/wiki/Unum_(number_format))

Dr. John Gustafson

Garcia, Yan

Conclusion

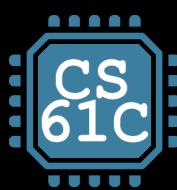
- **Floating Point lets us...**
 - Represent numbers with both integer and fractional parts making efficient use of available bits.
 - Store approximate values for very large and very small #s.
- **IEEE 754 Floating Point Standard is the most widely accepted attempt to standardize interpretation of such numbers.**
 - Every computer since ~1997 follows these conventions!
- **Summary (single precision, or fp32):**



Can store
NaN, $\pm \infty$



Exponent tells Significand how much (2^i) to count by (... , $1/4$, $1/2$, 1 , 2 , ...)



Next time: IEEE-754 Floating Point Converter

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>

IEEE 754 Converter (JavaScript), V0.22

Sign	Exponent	Mantissa
Value: Encoded as: Binary:	-1 1 <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/>	2^1 128 1.75 6291456
Decimal representation: -3.5 Value actually stored in float: -3.5 Error due to conversion: Binary Representation: 11000000011000000000000000000000 Hexadecimal Representation: 0xc0600000		

Sign	Exponent	Mantissa
Value: Encoded as: Binary:	+1 0 <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>	2^{23} 150 1.0000001192092896 1
Decimal representation: 8388609.0 Value actually stored in float: 8388609 Error due to conversion: Binary Representation: 01001011000000000000000000000001 Hexadecimal Representation: 0x4b000001		

(Bonus) More Examples

- Basics, Fixed Point
- Floating Point
- Special Numbers
- Other Floating Point Representations
- (Bonus) More Examples

Ex 1: Convert Binary Floating Point to Decimal

31 30 23 22 0

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

s exponent

significand

0 → positive

0110 1000_{two} = 104_{ten}

$$\begin{aligned} & 1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + \dots \\ & = 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22} \\ & = 1.0 + 0.666115 \end{aligned}$$

Bias adjustment:

$$104 - 127 = -23$$

→ $1.666115_{\text{ten}} \times 2^{-23} \approx 1.986 \times 10^{-7}$
(about 2/10,000,000)

Ex 2: Convert Decimal to Binary Floating Point

-2.340625 $\times 10^1$

1. Denormalize.

-23.40625

2. Convert integer part.

$$23 = 16 + (7 = 4 + (3 = 2 + (1))) = 10111_2$$

3. Convert fraction part.

$$.40625 = .25 + (.15625 = .125 + (.03125)) = .01101_2$$

4. Put parts together + normalize.

$$10111.01101 = 1.011101101 \times 2^4$$

5. Convert exponent.

$$127 + 4 = 10000011_2$$

0	1000 0011	011 1011 0100 0000 0000 0000
---	-----------	------------------------------

Ex 3: Represent 1/3

- $1/3$

$$= 0.33333\dots_{10}$$

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

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

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

$$= 0.0101010101\dots_2 * 2^0$$

$$= 1.0101010101\dots_2 * 2^{-2}$$

- **Sign:** 0
- **Exponent** = $-2 + 127 = 125 = 01111101$
- **Significand** = 0101010101...

0	0111 1101	010 1010 1010 1010 1010 1010
---	-----------	------------------------------

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

