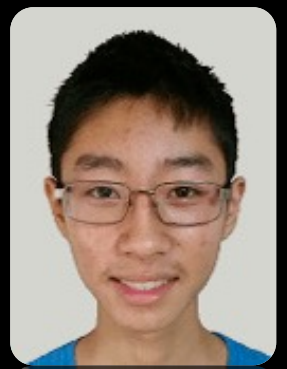
UC Berkeley
Teaching Professor
Lisa Yan

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)



UC Berkeley
Lecturer
Justin Yokota

Floating Point

Faulty Heap Management

[from last time]

How many memory management errors are in this code?

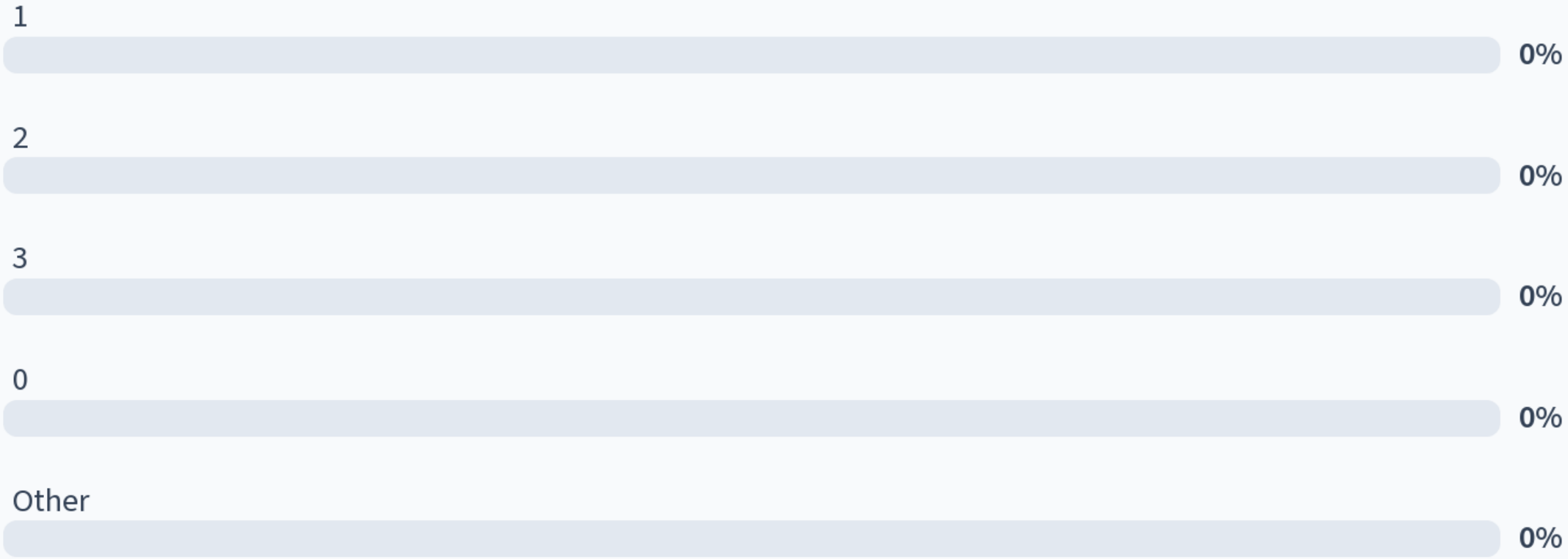
```
void free_mem_x() {
  int fnh[3];
  ...
  free(fnh);
}

void free_mem_y() {
  int *fum = malloc(4*sizeof(int));
  free(fum+1);
  ...
  free(fum);
  ...
  free(fum);
}
```

A. 1
B. 2
C. 3
D. 0
E. Other

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

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

How many memory management errors are in this code?



Faulty Heap Management

[Concept Check]

How many memory management errors are in this code?

```
void free_mem_x() {
  int fnh[3];
  ...
  free(fnh);
}

void free_mem_y() {
  int *fum = malloc(4*sizeof(int));
  free(fum+1);
  ...
  free(fum);
  ...
  free(fum);
  }
```

`free()` on stack-allocated memory

`free()` on memory that isn't the pointer from `malloc`

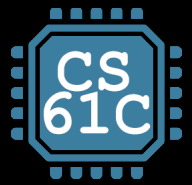
Double `free()`

A. 1
B. 2
C. 3
D. 0
E. Other

Great Idea #1: Abstraction (Levels of Representation/Interpretation)

Week 1	–	Welcome	C 1: Intro
Week 2	C 2: Pointers, Arrays, and Strings	Number 1: Bit ops, Integer num rep	C 3: Memory (Stack, Heap)
Week 3	**Number 2: Floating Point**	**C 4: Generics, Function Pointers**	RISC-V 1: Intro to Assembly

Learning C means learning the binary abstraction! Our first "unit" of CS61C interleaves bit representation with C.



Agenda

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

Yan, Yokota

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: $2^{(N-1)} - 1 = 2{,}147{,}483{,}647$)

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)
  - 0.000000000052917710 ($5.2917710 \times 10^{-11}$)

- #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:

- With the above 6-bit "fixed binary point" representation:
  - Range: 0 to 3.9375

$11.1111_{two} = 4 - 1x2^{-4}$

Arithmetic with Fixed Point

- Addition is straightforward:

```
  01.1000   1.5₁₀
+ 00.1000   0.5₁₀
  10.0000   2.0₁₀
```

- Multiplication is a bit more complex:

```
    01.1000   1.5₁₀
  x 00.1000   0.5₁₀
    00 0000
   000 000
  0000 00
 01100 0
000000
000000
0001100 0000 → 00.1100
```

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

- Very small numbers? (Bohr radius)
  - 0.000000000052917710 ($5.2917710 \times 10^{-11}$)

- #s with both integer & fractional parts?
  - ✅ 2.625

34 bits

58 bits

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

Agenda

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`…

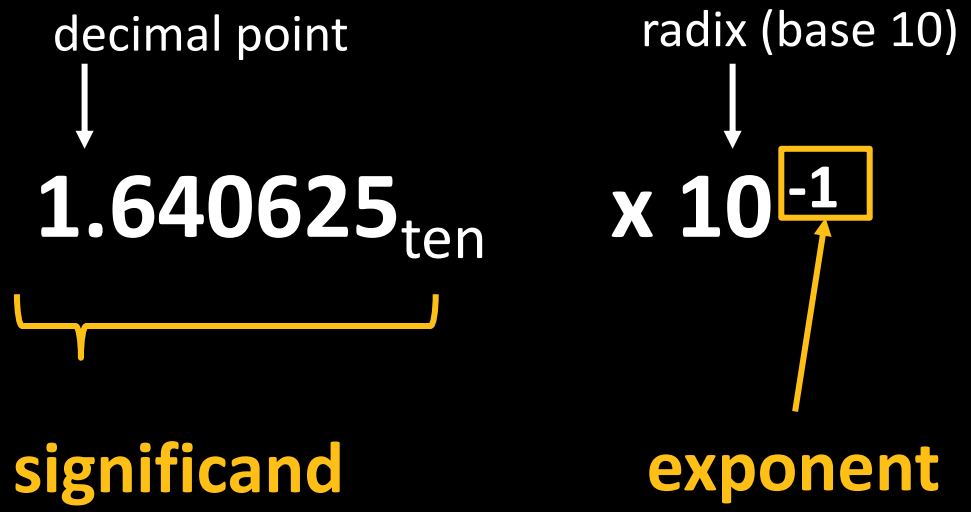
1. Store these "**significant bits**."

2. Keep track of the binary point 2 places to the left of the MSB ("**exponent field**").

- The binary point is stored separately from the significant bits, so very large and small numbers can be represented.

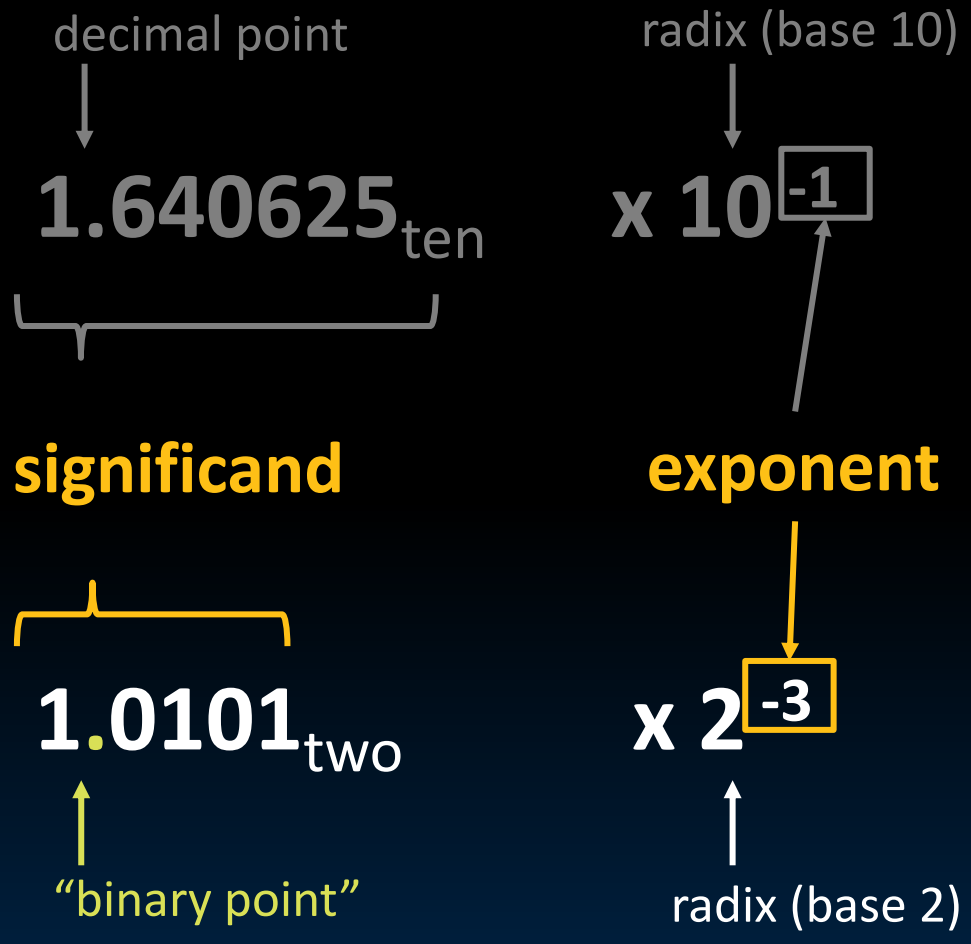
Enter Scientific Notation (in Decimal)

- "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: 0.3×10^{-8}, 30.0×10^{-10}



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: 0.3×10^{-8}, 30.0×10^{-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`

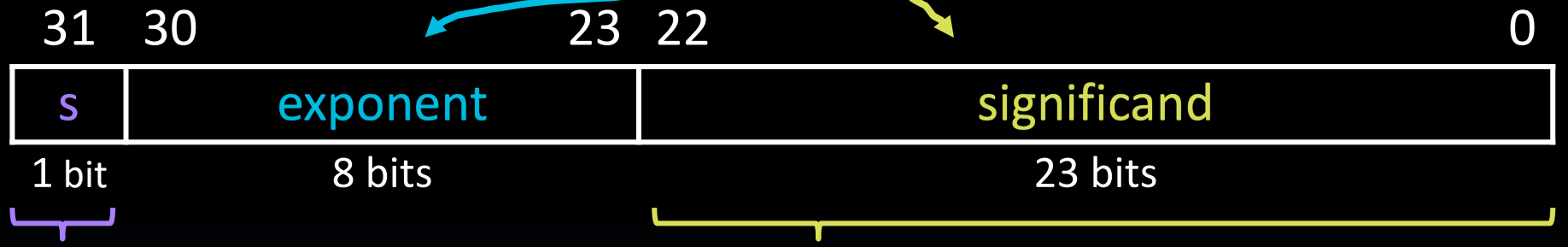
Note: **Normalized** binary representation always leads with a 1! (why?)

IEEE 754 Floating Point Standard (1/3)

- Single precision standard for 32-bit word. **In C,** `float`.
- **Normalized format:** $\pm 1.xxx\ldots x_{two} \times 2^{yyy\ldots y_{two}}$

IEEE 754 Floating Point Standard (1/3)

- Single precision standard for 32-bit word. **In C,** `float`.
- **Normalized format:** $\pm 1.xxx\ldots x_{two} \times 2^{yyy\ldots y_{two}}$



Sign bit:

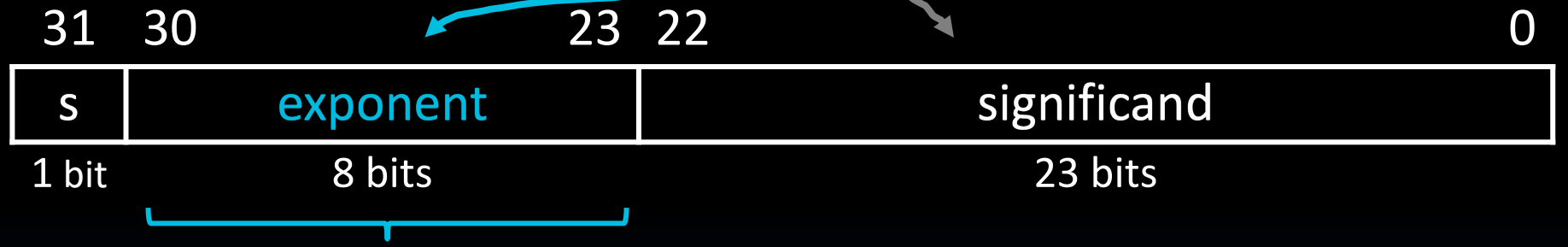
- 1 is negative
- 0 is positive

Significand field (i.e., mantissa):

- Normalized convention: significand leads w/1
- To pack more bits, treat leading 1 as **implicit**; represent other bits explicitly in significand.
  - **significand** = **1** + 23-bit **significand** field
  - 0 < significand field < 1

IEEE 754 Floating Point Standard (2/3)

- Single precision standard for 32-bit word. **In C,** `float`.
- **Normalized format:** $\pm 1.xxx\ldots x_{two} \times 2^{yyy\ldots y_{two}}$

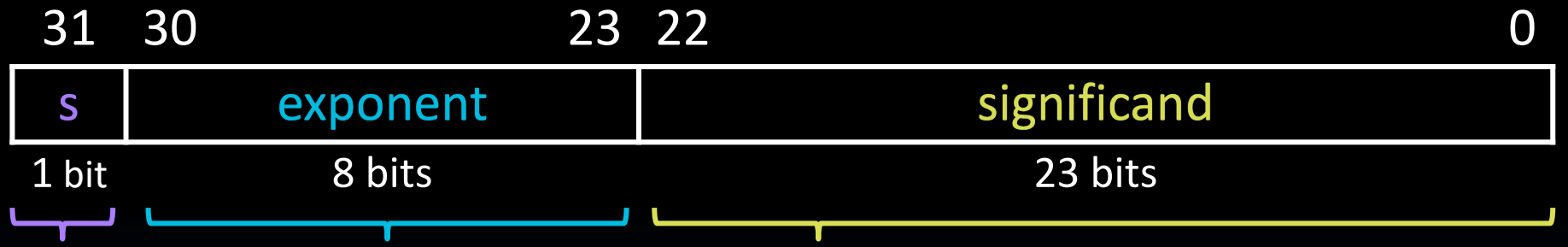


Exponent field uses "bias notation":

- 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…00 → 11…11
    0 to +MAX → -0 to -MAX

IEEE 754 Floating Point Standard (3/3) [Summary]

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



Sign bit

- 1 is negative
- 0 is positive

Exponent field

- Bias of **127**
- Subtract 127 from exponent field to get exponent value

Significand field (i.e., mantissa)

- To pack more bits, numerical significand leads w/implicit 1
- **significand** = **1** + 23-bit **significand** field

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

How to represent zero?

More later…

"Father" of the Floating Point Standard

- IEEE Standard 754 for Binary Floating-Point Arithmetic.

1989
ACM Turing
Award Winner!

Prof. William Kahan

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

Normalized Example

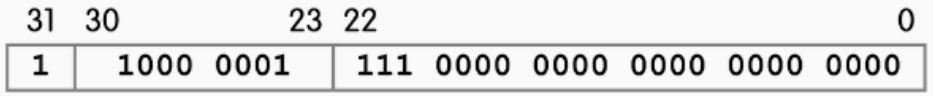
- What is the decimal equivalent of the following IEEE 754 single-precision binary floating point number?

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

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

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

0



-7 * 2^129 0%

-3.5 0%

-3.75 0%

-7 0%

-7.5 0%

Something else 0%

Normalized Example

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

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

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

$-1 \times (1.111)_{two} \times 2^{(2)}$

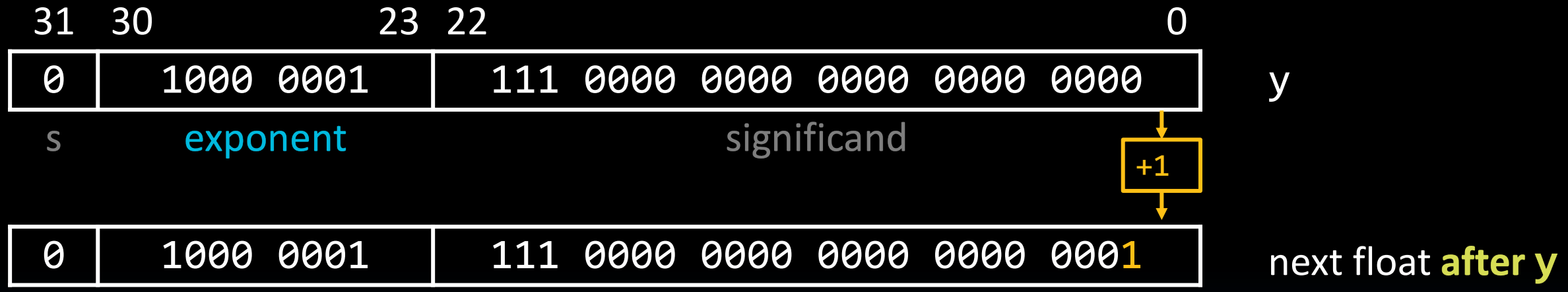
-111.1_{two}

-7.5_{ten}

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

Step Size

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



$$y + ((.0\ldots01)_{two} \times 2^{(129-127)})$$

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

$$y + 2^{-21}$$

"step size"

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

- Bigger exponents → bigger step size.
- Smaller exponents → smaller step size!

Computing in the News(?) Climate Change


```
INT_MAX:     2147483647
UINT_MAX:    4294967295
FLOAT_MAX: 3.40282e+38
```

s	1111 1110	1…1 (23 bits)

$(1.\mathbf{1\ldots1}) \times 2^{(\mathbf{254}-127)} \approx 3.4 \times 10^{38}$

This is the maximum normalized float rep in IEEE 754, where biased exponents are restricted to the range 1 to 254 (2^{-126} to 2^{127}).

Exponent fields 0, 255 are reserved for **special numbers**. More now!

Agenda

Special Numbers

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

Representing Zero

- Note: **Zero** has no normalized representation (i.e., 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?

Special Numbers

- Normalized numbers are only a fraction (heh) of floating point representations. For single-precision (32-bit):

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."

The 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 magnitude:

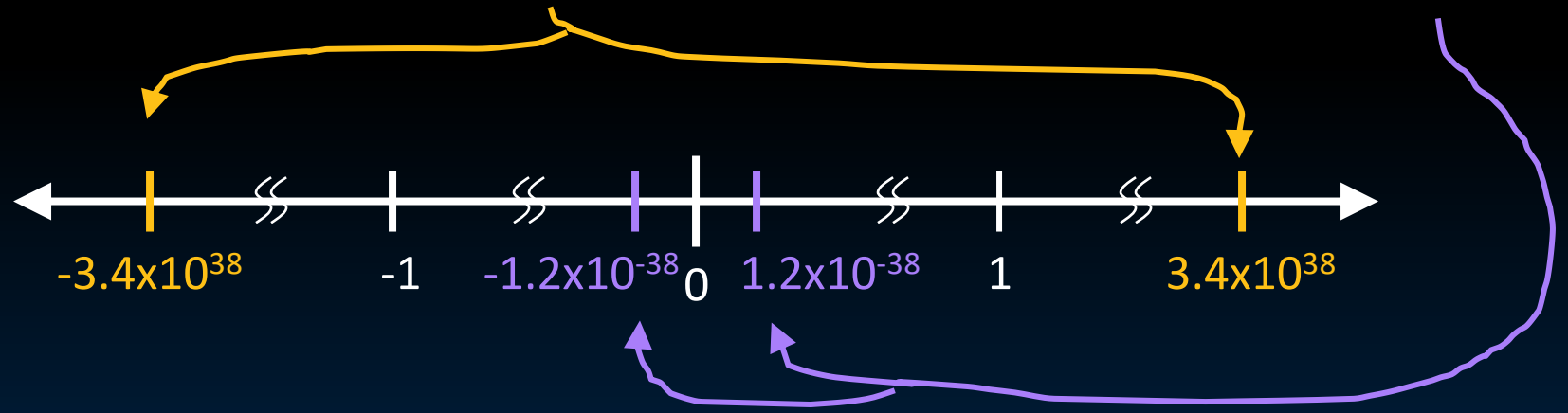
s	1111 1110	1…1 (23 bits)

$(1.1\ldots1) \times 2^{(254-127)} \approx 3.4 \times 10^{38}$

  - Smallest magnitude:

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 magnitude:

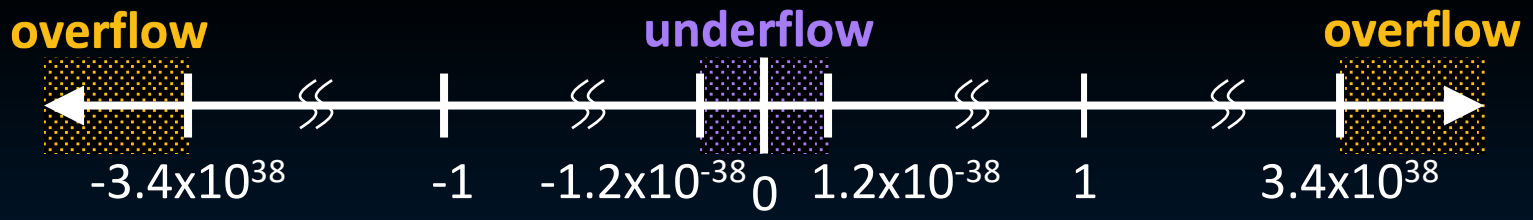
s	1111 1110	1…1 (23 bits)

$(1.1…1) \times 2^{(254-127)} \approx 3.4 \times 10^{38}$

  - Smallest magnitude:

s	0000 0001	0…0 (23 bits)

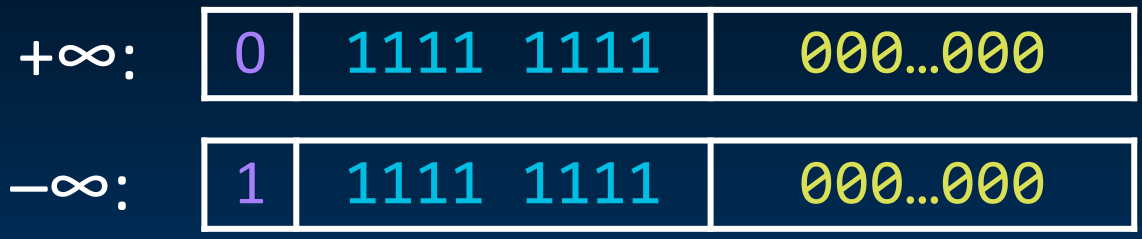
$(1.0) \times 2^{(1-127)} \approx 1.2 \times 10^{-38}$



- 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 ± ∞

- In FP, divide by ±0 should produce ±∞, 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 ±∞:
  - Reserve exponent value 1111 1111.
  - And keep significand all zeroes.



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(NaN, X) = NaN
  - Hope NaNs help with debugging?
  - Can use the significand to encode/identify where errors occurred! (proprietary; not defined in standard)

Denorms: Gradual Underflow (1/2)

[for next time]

- Problem:
  - There's a **gap** among representable FP numbers around zero!
  - Smallest normalized number:

0	0000 0001	000…000

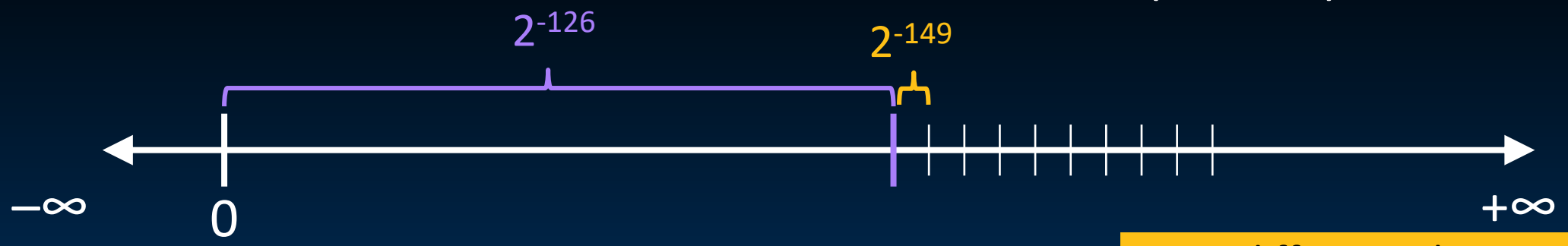
$(1.0) \times 2^{(1-127)} = 2^{-126}$

  - 2nd smallest normalized number:

31	30 … 23	22 … 1 0
0	0000 0001	00000…00001

$(1.000\ldots\ldots1_2) \times 2^{(1-127)}$

$= (1 + 2^{-23}) \times 2^{-126} = 2^{-126} + 2^{-149}$

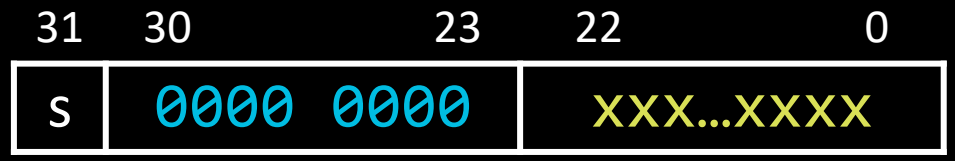


Huge difference between 0 and smallest vs. smallest and 2nd smallest! Occurs b/c of normalization and implicit leading 1.

Denorms: Gradual Underflow (2/2)

[for next time]

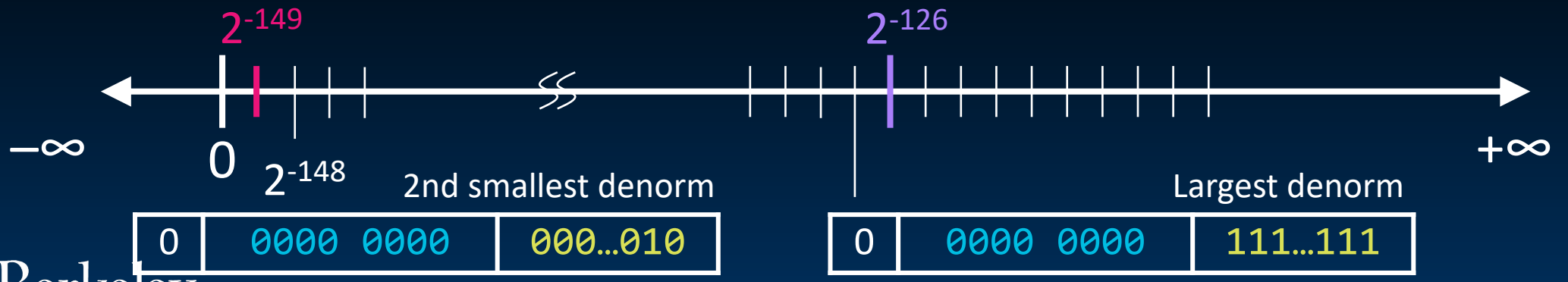
- Solution:
  - We still haven't used Exponent = 0, Significand nonzero.
  - **DEnormalized** number:
    - no (implied) leading 1
    - implicit exponent for all denorms = -126.



$(-1)^s \times (0.x\ldots\ldots x_2) \times 2^{-126}$

Smallest denormalized number

Smallest normalized number



Special Numbers, Summary

[for next time]

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

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

For you: IEEE-754 Floating Point Converter

[for next time]

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

IEEE 754 Converter (JavaScript), V0.22

	Sign	Exponent	Mantissa
Value:	-1	2^{1}	1.75
Encoded as:	1	128	6291456
Binary:	1	10000000	11000000000000000000000

Decimal representation	-3.5
Value actually stored in float:	-3.5
Error due to conversion:	
Binary Representation	11000000011000000000000000000000
Hexadecimal Representation	0xc0600000

+1 -1

IEEE 754 Converter (JavaScript), V0.22

	Sign	Exponent	Mantissa
Value:	+1	2^{23}	1.0000001192092896
Encoded as:	0	150	1
Binary:	0	10010110	00000000000000000000001

Decimal representation	8388609.0
Value actually stored in float:	8388609
Error due to conversion:	
Binary Representation	01001011000000000000000000000001
Hexadecimal Representation	0x4b000001

+1 -1

Agenda

[for next time]

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!

[for next time]

- **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)

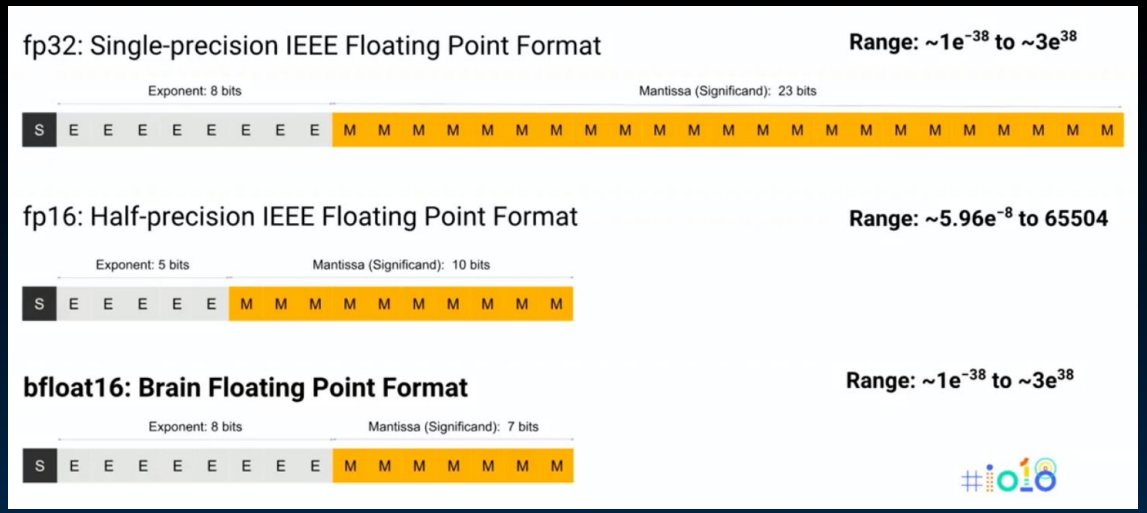
- Double Precision (vs. Single Precision)
  - C variable declared as `double`
  - Exponent bias now 1023.
  - Represent numbers from ~2.0 x 10^{-308} to ~2.0 x 10^{308}.
  - The primary advantage is greater **accuracy** due to larger significand.

Other Floating Point Representations

[extra]

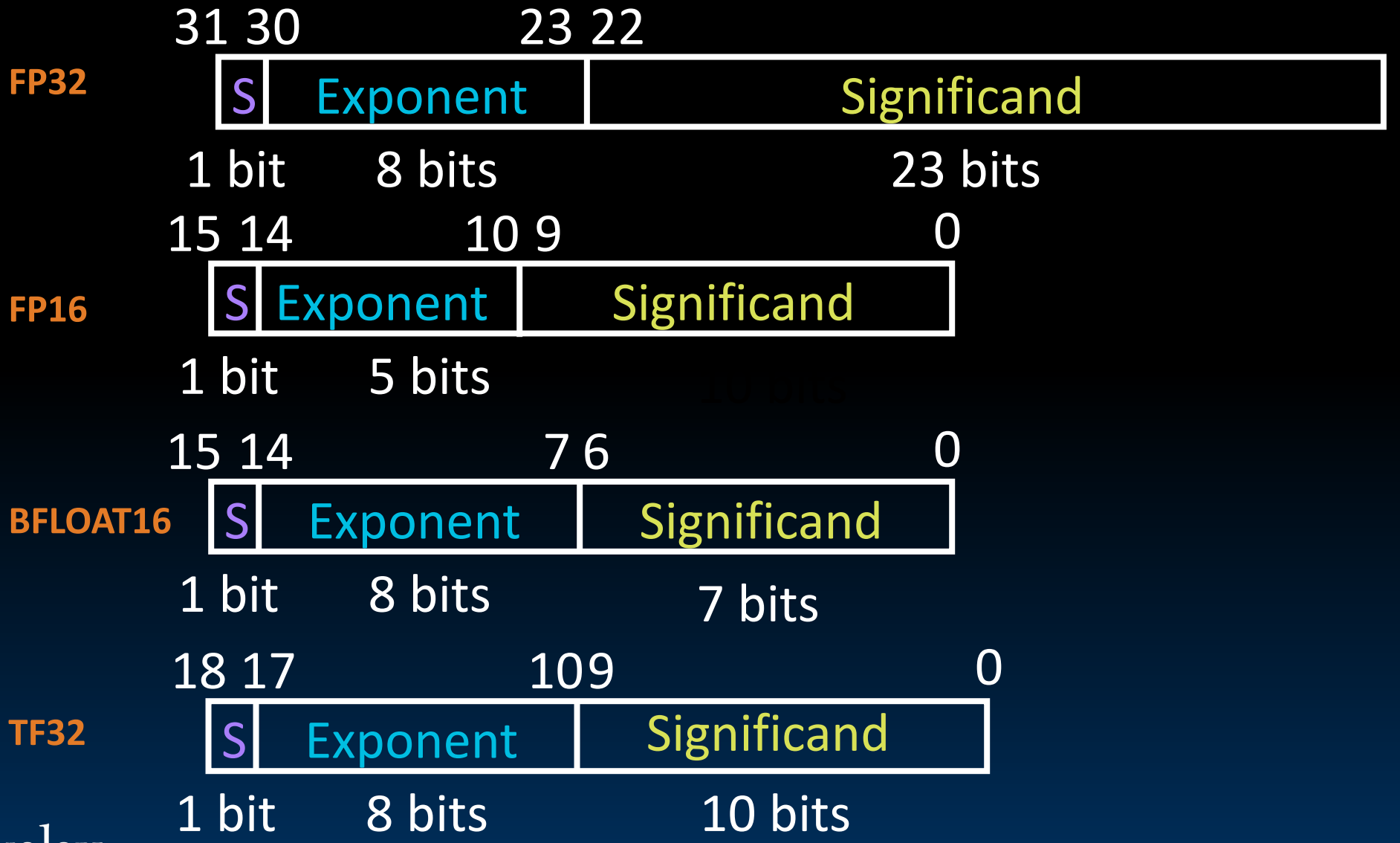
- Quad-Precision? Yep! (128 bits) "binary128"
  - Unbelievable range, precision (accuracy)
  - 15 exponent bits, 112 significand bits
- Oct-Precision? Yep! "binary256"
  - 19 exponent bits, 236 significant bits
- Half-Precision? Yep! "binary16" or "fp16"
  - 1/5/10 bits
- Half-Precision? Yep! "bfloat16"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

https://cloud.google.com/tpu/docs/bfloat16



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

Floating Point Soup



Who Uses What in Domain Accelerators?

Accelerator	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					

Unum

[extra]

- 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!

Dr. John Gustafson

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

Conclusion

www.h-schmidt.net/FloatConverter/IEEE754.html

- 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):

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

Can store NaN, ± ∞ ☑

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

Agenda

(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

0 → positive

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

Bias adjustment:

104 - 127 = -23

$1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$

$= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}$

$= 1.0 + 0.666115$

$1.666115_{ten} * 2^{-23} \approx 1.986 * 10^{-7}$

(about 2/10,000,000)

Ex 2: Convert Decimal to Binary Floating Point

-2.340625×10^1

1. Denormalize. -23.40625
2. Convert integer part.

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

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

  $= 0.33333\ldots_{10}$

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

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

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

  $= 0.0101010101\ldots_2 * 2^0$

  $= 1.0101010101\ldots_2 * 2^{-2}$

  - **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} $340_{10}/1000_{10}$
    $34_{10}/100_{10}$
  - In binary: 0.110_2 $110_2/1000_2 = 6_{10}/8_{10}$
    $11_2/100_2 = 3_{10}/4_{10}$
  - 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:

$1.6732 = (1x10^{0}) + (6x10^{-1}) + (7x10^{-2}) + (3x10^{-3}) + (2x10^{-4})$

  - In binary:

$1.1001 = (1x2^{0}) + (1x2^{-1}) + (0x2^{-2}) + (0x2^{-3}) + (1x2^{-4})$

  - 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