

CS61C: Great Ideas in Computer Architecture (Machine Structure)

Instructors: Rosalie Fang, Charles Hong, Jero Wang

C Demo

Linked List Example

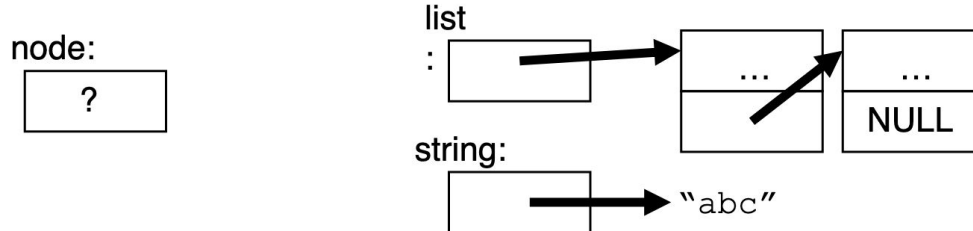
Linked List Example

- Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a linked list of strings.

```
struct Node {  
    char *value;  
    struct Node *next;  
};  
  
typedef struct Node *List;  
  
/* Create a new (empty) list */  
List ListNew(void)  
{ return NULL; }
```

Linked List Example

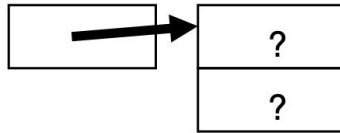
```
/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```



Linked List Example

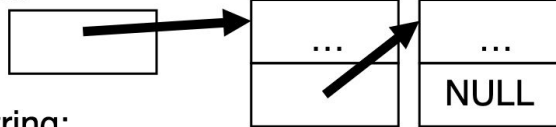
```
/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```

node:

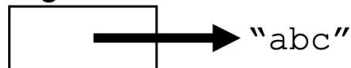


list

:

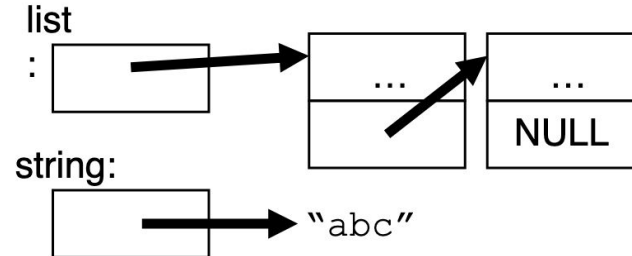
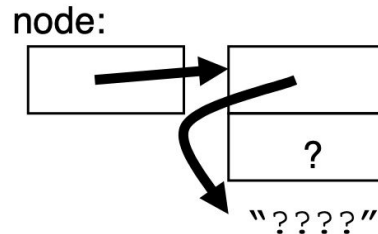


string:



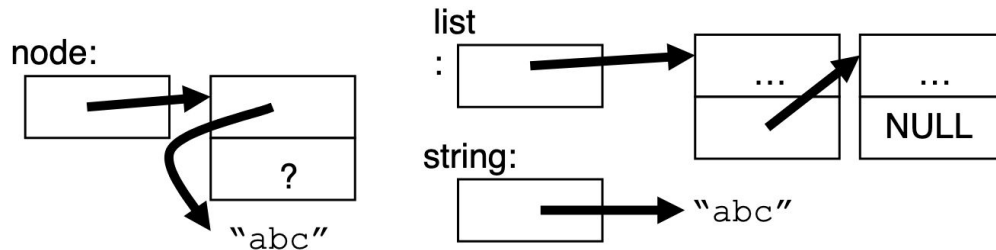
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```



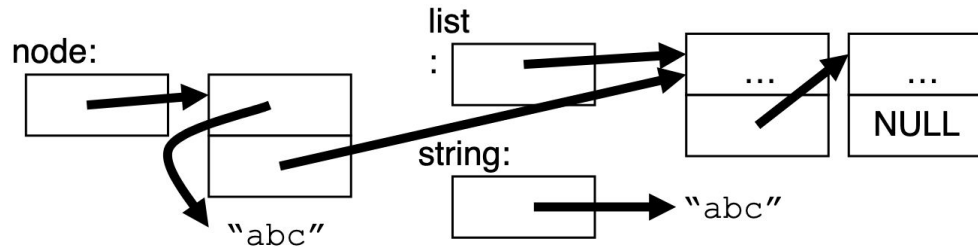
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```



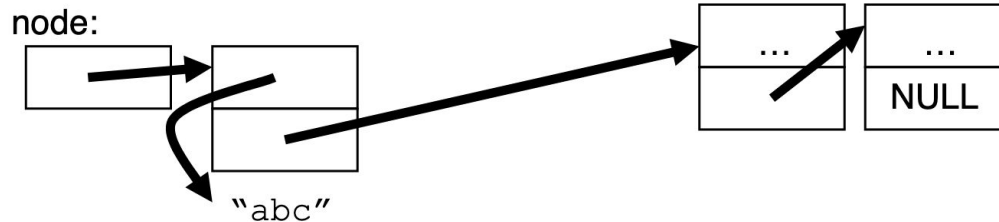
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```



Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```



Floating Point!!!

What have we learned so far?

- Computers are made to process bit-representation of information
- What can we represent in N bits?
 - At most 2^N unique things
 - Unsigned integers
 - Ranges from 0 to 2^N-1
 - $N=32$: max number is 4,294,967,295
 - Signed integers (two's complement)
 - Ranges from $-2^{(N-1)}$ to $2^{(N-1)}-1$
 - $N=32$: max number is 2,147,483,647

What about other numbers?

- Very large numbers (Avogadro's number)
 - 602,252,000,000,000,000,000,000 (6.02252×10^{23})

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”, Decimal

- We have a “decimal point” to signify the divide between integer and fraction parts
- Example: 153.294

$$\begin{array}{ccccccccc} \mathbf{\underline{1}} & \mathbf{\underline{5}} & \mathbf{\underline{3}.\underline{2}} & \mathbf{\underline{9}} & \mathbf{\underline{4}} & & & & \\ 10^2 & 10^1 & 10^0 & \bullet & 10^{-1} & 10^{-2} & 10^{-3} & & \\ = 1 \times 10^2 + 5 \times 10^1 + 3 \times 10^0 + 2 \times 10^{-1} + 9 \times 10^{-2} + & & & & & & & & \\ 4 \times 10^{-3} & & & & & & & & \\ & & & & & & & & \\ = 100 + 50 + 3 + 0.2 + 0.09 + 0.004 & & & & & & & & \\ & & & & & & & & \\ = 153.294!! & & & & & & & & \end{array}$$

“Fixed Point”, Binary

- We have a “binary point” to signify the divide between integer and fraction parts
- Example: 5 bit binary, positive numbers only

$$\begin{array}{ccccccc} \underline{\mathbf{1}} & \underline{\mathbf{0}} & \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{1}} & & \\ 2^1 & 2^0 & \bullet & 2^{-1} & 2^{-2} & 2^{-3} & \end{array}$$

$$= 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

$$= 2 + 0.25 + 0.125$$

$$= 2.375$$

With this 5-bit fixed point representation,
range is: 00.000 to 11.111 ($2^0 + 2^1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}$)
range is: 0 to 3.875

Arithmetic with Fixed Point

Addition is straightforward:



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

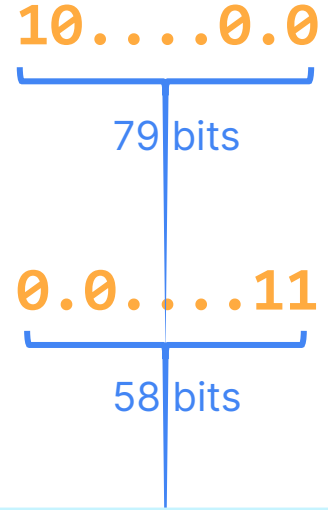
Multiplication is a bit more complex:

$$\begin{array}{r} 01.1000 \quad 1.5_{10} \\ \times 00.1000 \quad 0.5_{10} \\ \hline 00 \ 0000 \\ 000 \ 000 \\ 0000 \ 00 \\ 01100 \ 0 \\ 000000 \\ 000000 \\ \hline 0001100 \ 0000 \end{array}$$

Need to remember where point is...

What about other numbers?

- Very large numbers (Avogadro's number)
 - 602,252,000,000,000,000,000,000 (6.02252×10^{23})

- Very small numbers? (Bohr radius)
 - 0.000000000052917710 ($5.2917710 \times 10^{-11}$)
- #s with both integer & fractional parts?
 -  2.625



To store all these numbers, we'd need a fixed-point rep with at least 137 bits. There must be a better way!

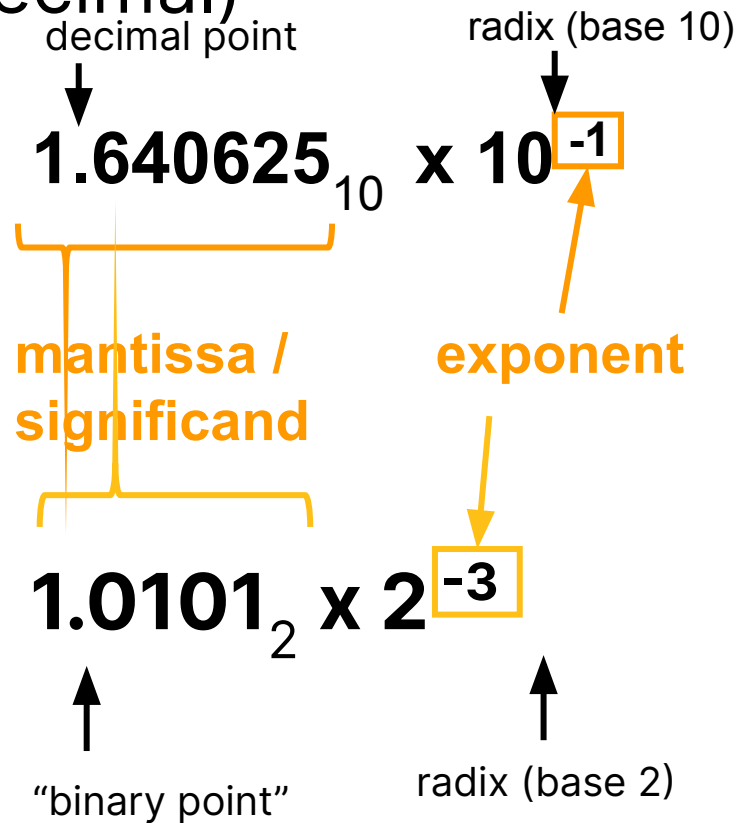
Floating Point

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_{10} .
 - 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.

Enter Scientific Notation (in Decimal)

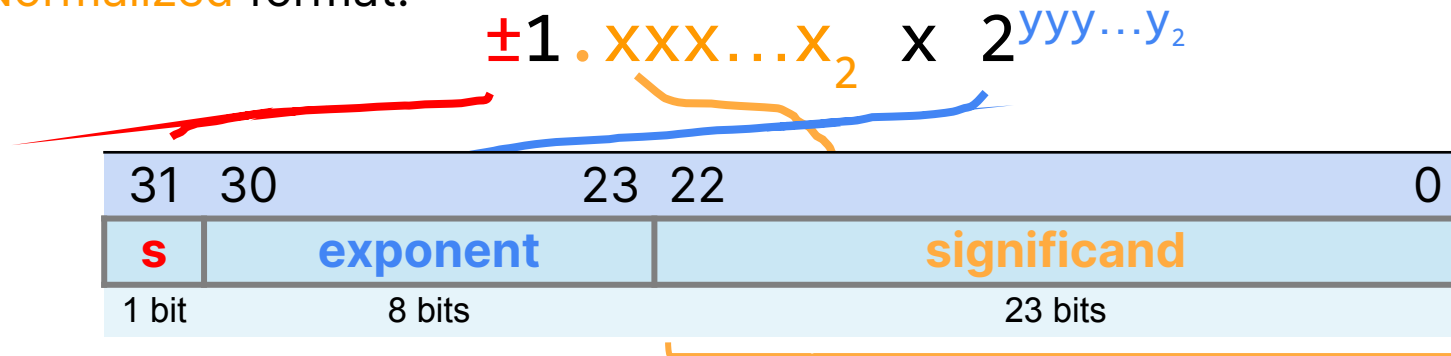
- “**Normalized form**”: no leading 0s (exactly one nonzero digit to left of point).
 - To represent 3/1,000,000,000:
 - **Normalized:** 3.0×10^{-9}
 - Not normalized: 0.3×10^{-8} , 30.0×10^{-10}
- “**Floating point**”: Computer arithmetic that supports this binary representation.
 - Represents numbers where the binary point is not fixed (as opposed to integers).
 - C variable types: **float**, **double**



Note: all normalized numbers start with a **1**, why?

IEEE 754 Floating Point Standard (1/3)

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



Sign Bit

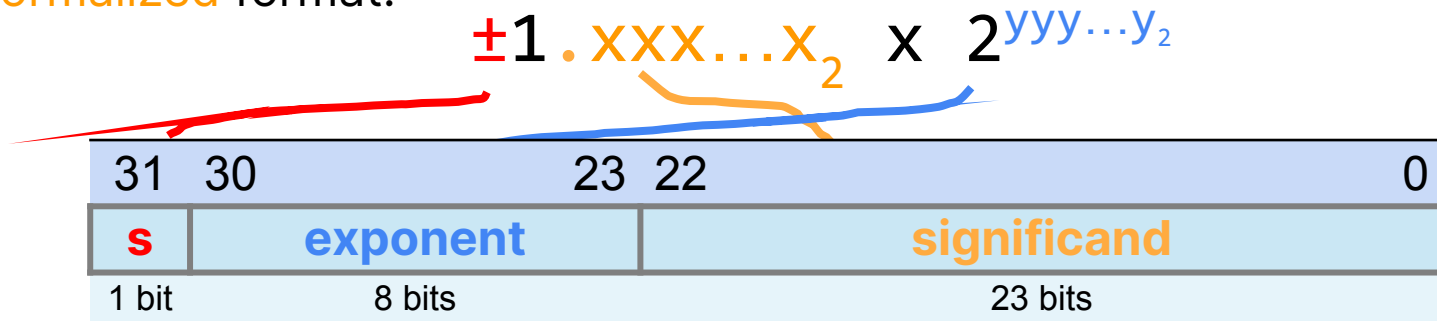
- 1 is negative
- 0 is positive

Significand field

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

IEEE 754 Floating Point Standard (2/3)

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

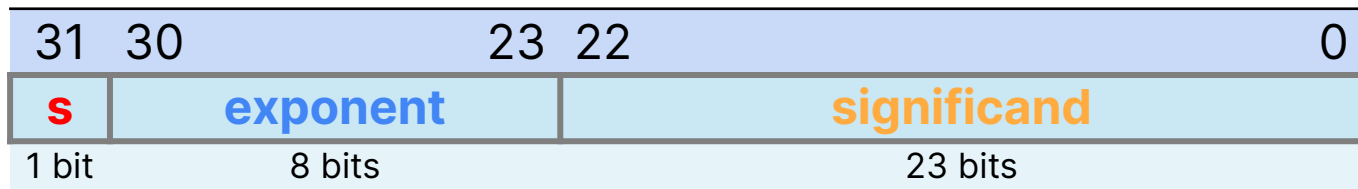


Exponent field uses “bias notation”:

- 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

IEEE 754 Floating Point Standard (3/3)

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



Sign Bit

- 1 is negative
- 0 is positive

Exponent field

- Bias = -127
- Add bias from (exponent field)₂ to (exponent value)₁₀

Significand field

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

$$(-1)^s \times (1.\text{significand}) \times 2^{(\text{exponent}-127)}$$

Normalized Example

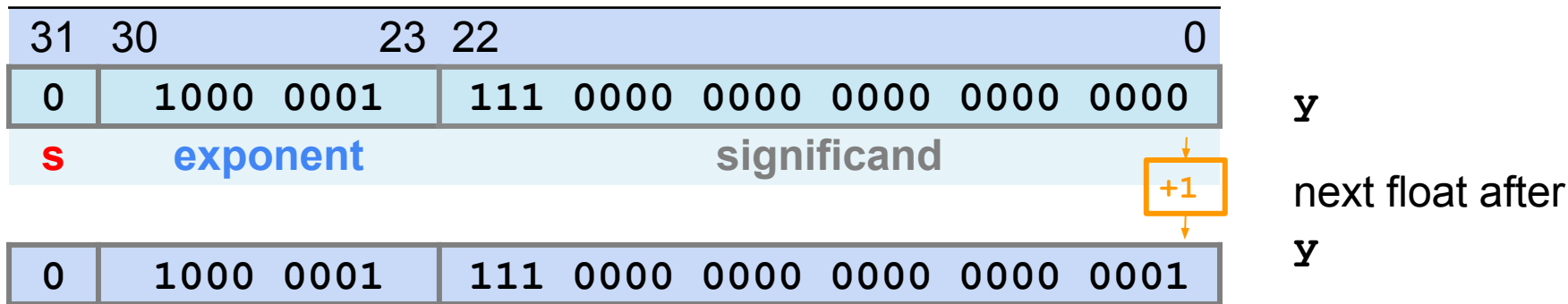
- What is the decimal equivalent of the following IEEE 754 single-precision binary floating point number?

31	30	23	22	0
1	1000 0001	111 0000 0000 0000 0000 0000		
s	exponent = 1 + 128 = 129			significand

$$\begin{aligned}
 & (-1)^s \times (1.\text{significand}) \times 2^{(\text{exponent}-127)} \\
 = & -1 \times 1.111 \times 2^{129-127} \\
 = & -1 \times 1.111 \times 2^2 \\
 = & -1 \times 111.1 \\
 = & -1 \times (4 + 2 + 1 + \frac{1}{2}) = -7.5
 \end{aligned}$$

Step Size

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



$$\mathbf{y} + ((.0\dots01)_2 \times 2^{(129-127)})$$

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

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

“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!

The image shows a digital clock and weather application interface. At the top center, the time "4:21 PM" is displayed in large white font. To its right are three vertical white dots. Below the time, the date "Wed, Jan 22" and battery status "94%" are shown. The main display area features a large white number "340,282,346,638,528,860,000,000,000,000,000,000,000°" arranged in four lines. On the left side, there is a yellow sun icon. Below the numbers, the word "F" is visible. At the bottom, the word "Sunny" is written in bold white font, followed by "San Diego, Wed 4:10 PM".

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

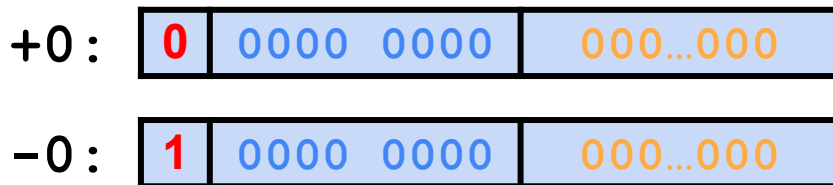
Biased exponents are 1 to 254 (2^{-126} to 2^{127}). Exponent fields 0, 255 are reserved!

Now we've been talking about
“normalized” floating point a lot...

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?



Special Numbers

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

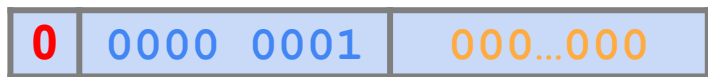
Biased Exponent	Significand field	Object
0	all zeros	± 0
0	nonzero	???
1 – 254	anything	Normalized floating point
255	all zeros	???
255	nonzero	???

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

Denorms: Gradual Underflow (1/2)

Problem:

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

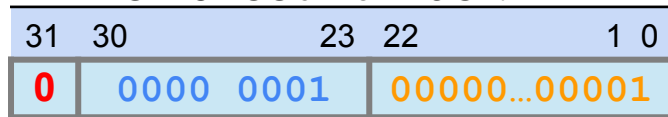


$$(1.0) \times 2^{(1-127)} = 2^{-126}$$

$$2^{-12}$$

$$2^{-14}$$

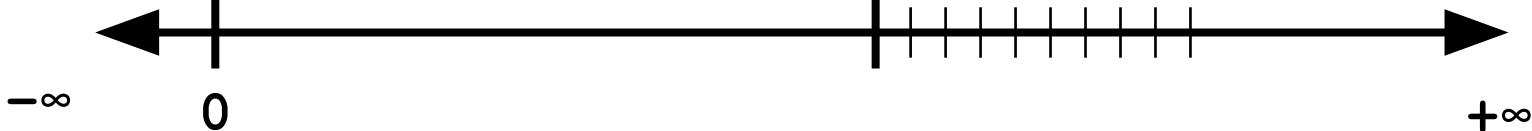
- 2nd smallest number:



$$(1.000\dots1_2) \times 2^{(1-127)}$$

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

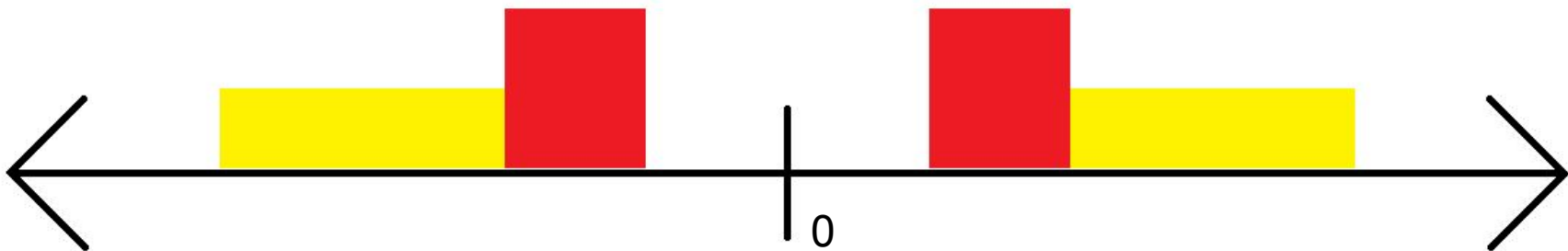
$$= 2^{-126} + 2^{-149}$$



Normalization and implicit 1 is to blame!

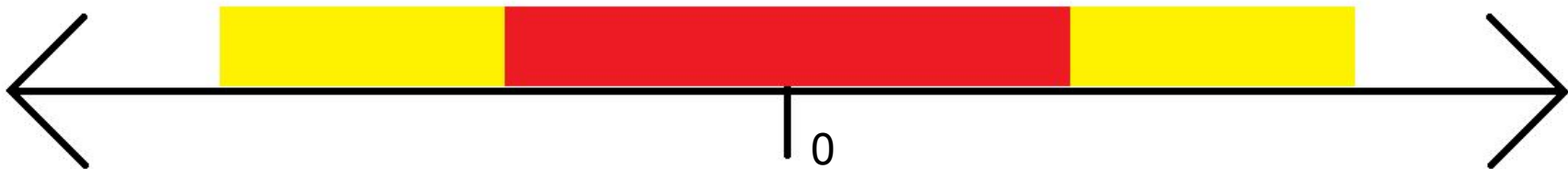
Problems with the implicit 1: Underflow

- The smallest number (in absolute value) we can represent is 2^{-127}
 - We can't represent 0
- The second smallest number is $2^{-127} + 2^{-150}$
 - There's a big gap in the set of representable numbers (10 million times bigger than the gaps between consecutive numbers)
- This is known as an underflow; the result of computation gets too small to be represented.
- In the below diagram, the red and yellow box represents the range (width of the box) and density (height of the box) of exponent 0b000...0 and 0b000...1, respectively. Note that each exponent “halves” the distance to 0.



Solution: Denormalized numbers (Gradual Underflow)

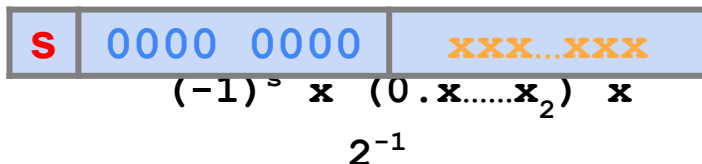
- Solution: Change the behavior of exponent 0b000...0 so that its range extends to 0
- How to do that?
 - When dealing with these numbers, use an implicit 0 instead of an implicit 1, so it doesn't overlap with exponent 0b000...1
- Ends up losing precision at small numbers (so there's still underflow), but at least it's not a sudden cliff drop.



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



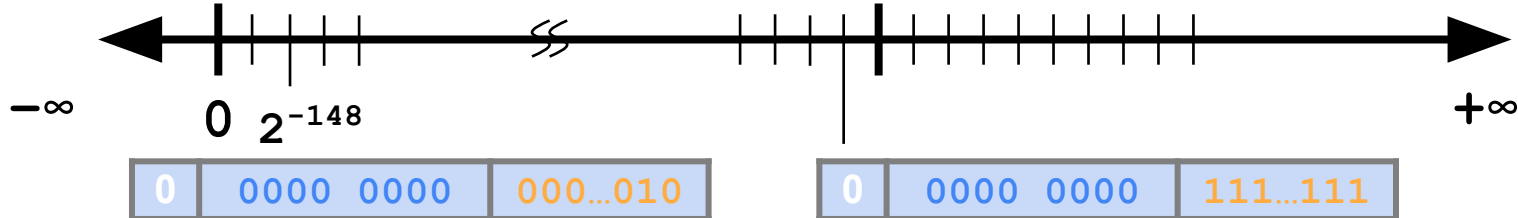
2^{-149}

0 2^{-148}

Smallest **normalized** number



2^{-126}



Representation for $\pm\infty$

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

$+\infty$:	0	1111 1111	000...000
$-\infty$:	1	1111 1111	000...000

Representation for Not a Number

- What do I get if I calculate $\text{sqrt}(-4.0)$ 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: $\text{op}(\text{NaN}, X) = \text{NaN}$
 - Hope NaNs help with debugging?
 - Can use the significand to encode/identify where errors occurred! (proprietary, not defined in standard)
- How many NaNs are there in the system?



Special Numbers, Summary

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

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

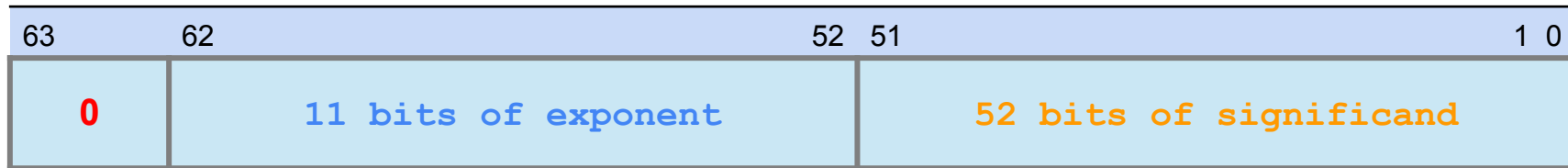
Don't confuse these two terms!

Precision and Accuracy

- “**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.

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}

Bias adjustment:

104 - 127 = -23

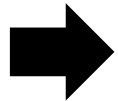
1.10101010100001101000010

= 1 + 1x2⁻¹ + 0x2⁻² + 1x2⁻³ + 0x2⁻⁴ + 1x2⁻⁵ +

...

= 1 + 2⁻¹ + 2⁻³ + 2⁻⁵ + 2⁻⁷ + 2⁻⁹ + 2⁻¹⁴ + 2⁻¹⁵ + 2⁻¹⁷ + 2⁻²²

= 1.0 + 0.666115



1.666115₁₀ * 2⁻²³ ≈ 1.986 * 10⁻⁷
(about 2/10,000,000)

Ex 2: Convert Decimal to Binary Floating Point

- -2.340625×10^1
- **Denormalize.** -23.40625
- **Convert integer part.**
 $23 = 16 + (7 = 4 + (3 = 2 + (1))) = 10111_2$
- **Convert fraction part.**
 $.40625 = .25 + (.15625 = .125 + (.03125))$
 $= .01101_2$
- **Put parts together + normalize.**
 $10111.01101 = 1.011101101 \times 2^4$
- **Convert exponent.**
 $127 + 4 = 10000011_2$

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

IEEE-754 Floating Point Converter

<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:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>	
Decimal representation	<input type="text" value="-3.5"/>		
Value actually stored in float:	<input type="text" value="-3.5"/>		
Error due to conversion:	<input type="text"/>		
Binary Representation	<input type="text" value="11000000011000000000000000000000"/>		
Hexadecimal Representation	<input type="text" value="0xc0600000"/>		

☒ +1
☐ -1

IEEE 754 Converter (JavaScript), V0.22

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

☒ +1
☐ -1

Recall: Goals for memory storage systems

- Any system/convention to store data ideally has the following properties:
- 1: The values that can be represented are relevant to the subject at hand
 - Different subject matters need different value sets, so we often have several different options with various trade-offs
- 2: The system is efficient, in terms of:
 - 2a: Memory use: As many possible bitstrings correspond to valid data values as possible
 - Ideal goal: Make it so that every bitstring corresponds to a different, valid data value
 - 2b: Operator cost: The operators we define are simple to make in circuitry, and require few transistors
 - Ideal goal: If we can reuse an existing circuit, then we don't need to add any additional circuitry!
 - Other aspects depending on the system; for now, we'll focus on these two
- 3: The system is intuitive for a human to understand
 - Often goes together with small operator cost, as well as adaptability to other systems
 - If a system's too complicated, no one will use it. Though if you manage to abstract away a lot of the complexity, you can get away with a less-intuitive system...