

The Hardware/Software Interface

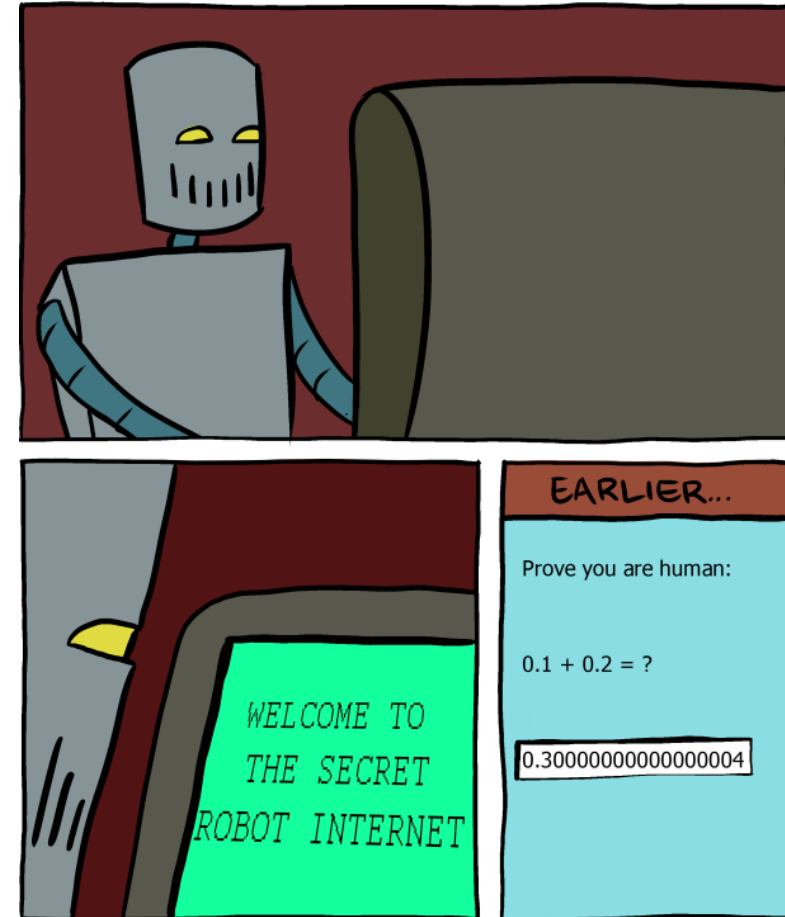
Floating Point

Instructors:

Justin Hsia, Amber Hu

Teaching Assistants:

Anthony Mangus	Divya Ramu
Grace Zhou	Jessie Sun
Jiuyang Lyu	Kanishka Singh
Kurt Gu	Liander Rainbolt
Mendel Carroll	Ming Yan
Naama Amiel	Pollux Chen
Rose Maresh	Soham Bhosale
Violet Monserate	



<http://www.smbc-comics.com/?id=2999>

Relevant Course Information

- ❖ Lecture polls are graded on *completion*
 - Don't change your answer afterward; misrepresents your understanding
- ❖ Early Course Reflection available on Canvas now, due Friday
- ❖ Lab 1a due tonight at 11:59 pm
 - Submit `pointer.c` and `lab1Asynthesis.txt`
 - Make sure there are no lingering `printf` statements in your code!
 - Make sure you submit *something* to Gradescope before the deadline and that the file names are correct
 - Can use late days to submit up until Wed 11:59 pm
- ❖ Lab 1b due next Monday (10/13)
 - Submit `aisle_manager.c`, `store_client.c`, and `lab1Bsynthesis.txt`

Lab 1b Aside: C Macros

❖ C macros basics:

- Basic syntax is of the form: `#define NAME expression`
- Allows you to use “NAME” instead of “expression” in code
 - Does naïve copy and replace *before* compilation – everywhere the characters “NAME” appear in the code, the characters “expression” will now appear instead
 - NOT the same as a Java constant
- Useful to help with readability/factoring in code

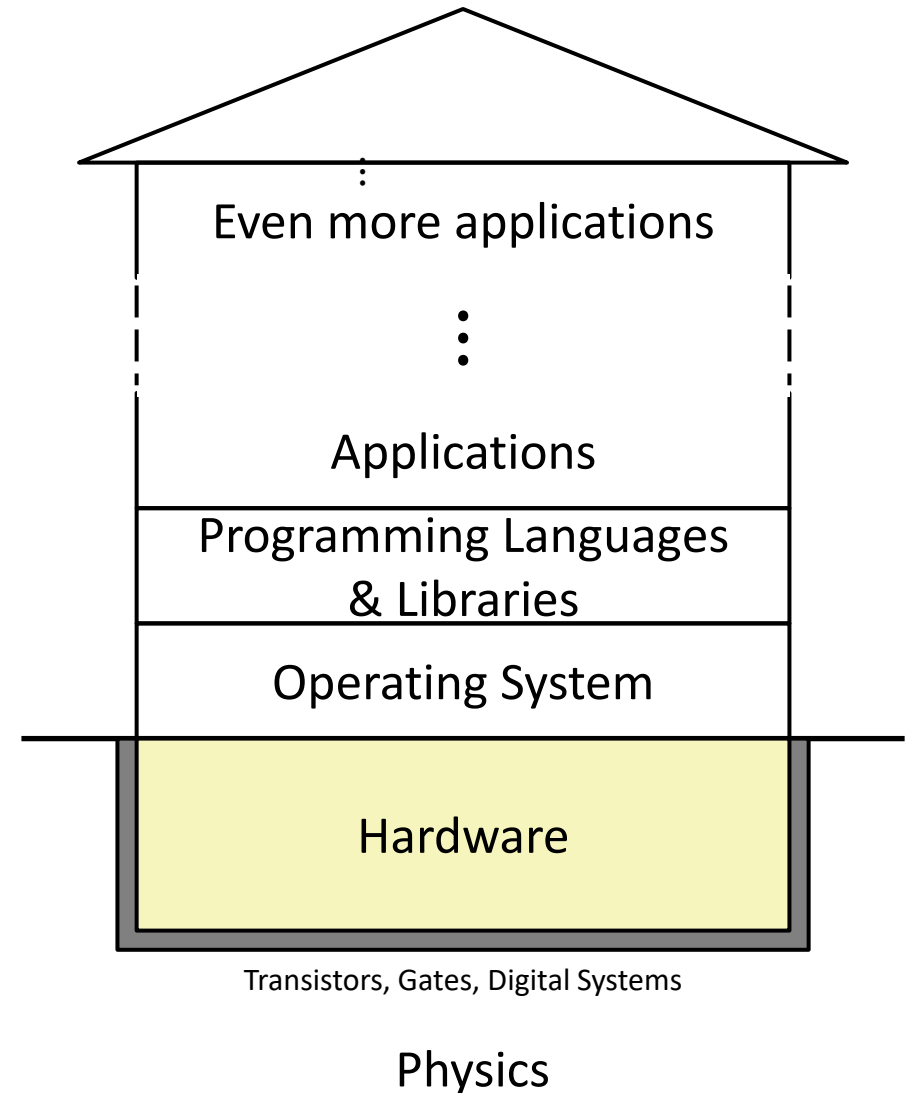
❖ You’ll use C macros in Lab 1b for defining bit masks

- See Lab 1b starter code and Lecture 04 (card operations) for examples

House of Computing Check-In

❖ Topic Group 1: **Data**

- Memory, Data, Integers, **Floating Point**, Arrays, Structs
- ❖ How do we store information for other parts of the house of computing to access?
 - How do we represent data and what limitations exist?
 - What design decisions and priorities went into these encodings?



Number Representation Revisited

❖ What can we represent in one word?

- Addresses
- Characters and Strings (ASCII)
- Signed and Unsigned Integers

❖ How do we encode the following:

- Real numbers (*e.g.*, 3.14159)
- Very large numbers (*e.g.*, 6.02×10^{23})
- Very small numbers (*e.g.*, 6.626×10^{-34})
- Special numbers (*e.g.*, ∞ , NaN)

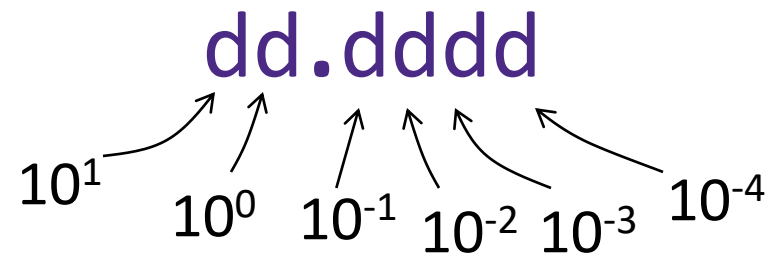
**Floating
Point**

Lecture Outline (1/5)

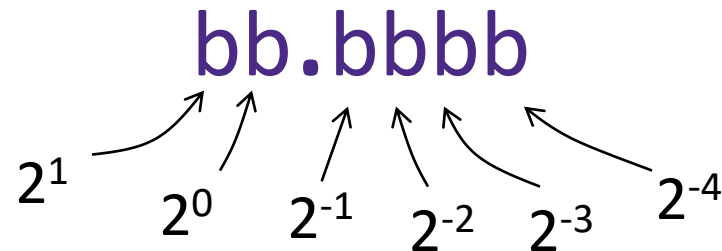
- ❖ **Scientific Notation**
- ❖ IEEE 754 Floating Point Encoding
- ❖ Floating Point Special Cases
- ❖ Floating Point Limitations and Dangers
- ❖ Floating Point in Real Life

Representation of Fractions (Review)

- ❖ In decimal, the *decimal point* signifies the boundary between integer and fractional parts:
 - Like leading zeros, can now have *trailing zeros* to the right of the point



- ❖ Same ideas apply in binary with the *binary point*:



Limits of Representation of Fractions

❖ Limitations:

- Given a *fixed* number of (consecutive) digits, you are limited in range, based on where you place the point
 - e.g., $bb.bb_{bb}$ ranges from 0 – 3.9375
- Even given an *arbitrary* number of digits, can only exactly represent numbers of the form $\sum_p (d_p \times b^p)$
 - b is the base, p is digit position (which can be negative), and d_p is the value of that digit's symbol
- Plenty of real and rational numbers **cannot** be exactly represented using digits:

Value	Decimal	Binary
$1/3$	$0.3333[3]..._{10}$	$0.010101[01]..._2$
$1/5$	0.2_{10}	$0.0011[0011]..._2$
π	$3.14159..._{10}$	$11.001001000011111..._2$

Scientific Notation (Review)

❖ General form:

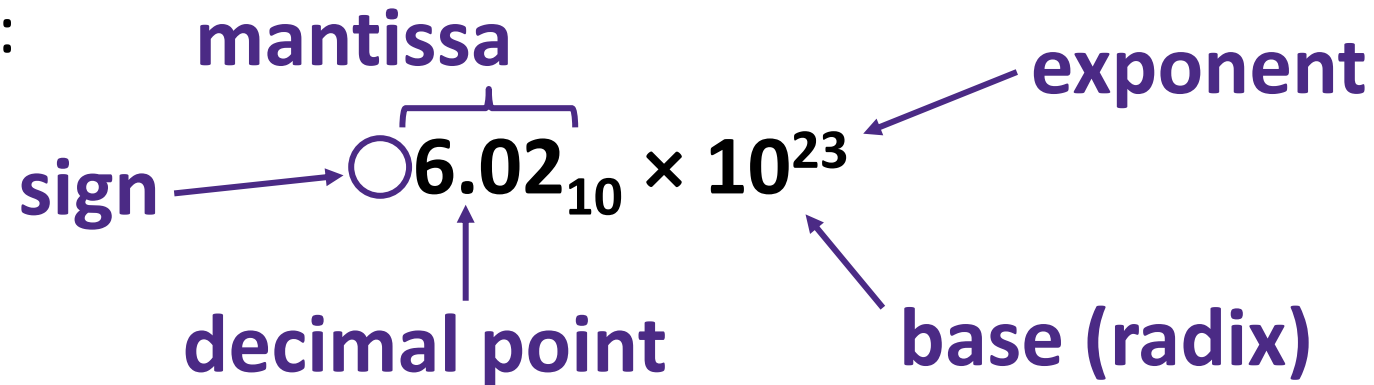
$$\text{numeral} \times \text{base}^{\text{power}}$$

- Changing power allows us to “shift” the point in the numeral

❖ *Normalized form*: exactly one digit (non-zero) to left of point

Decimal Scientific Notation

❖ Terminology:



- Changing power allows us to “shift” the point in the numeral

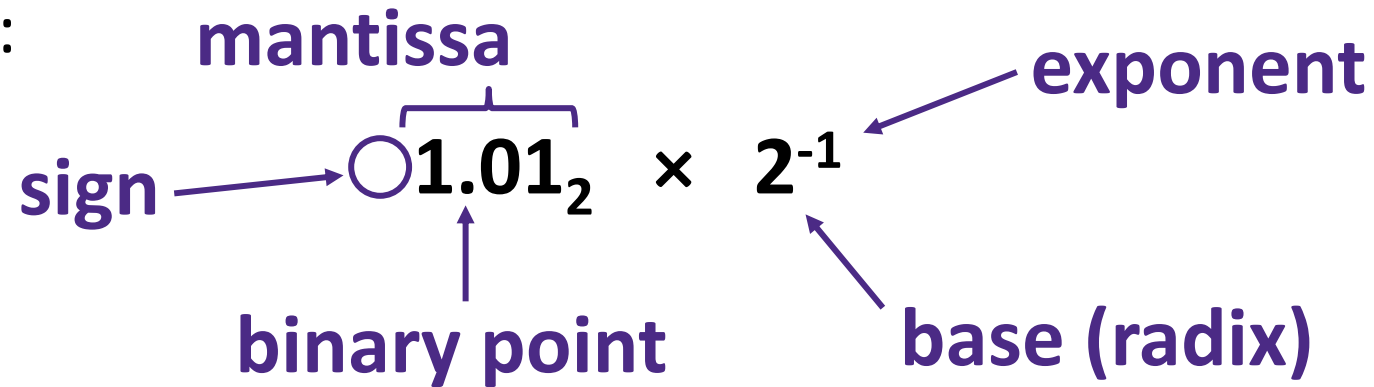
- Example: $3.51 \times 10^1 = 0.351 \times 10^2 = 35.1 \times 10^0$

not normalized

- ❖ Normalized form: exactly one digit (non-zero) to left of point

Binary Scientific Notation

❖ Terminology:



- Changing power allows us to “shift” the point in the numeral

- Example: $1.01_2 \times 2^{-1} = 0.101_2 \times 2^0 = 10.1_2 \times 2^{-2}$


not normalized

- ❖ Normalized form: exactly one digit (non-zero) to left of point

Lecture Outline (2/5)

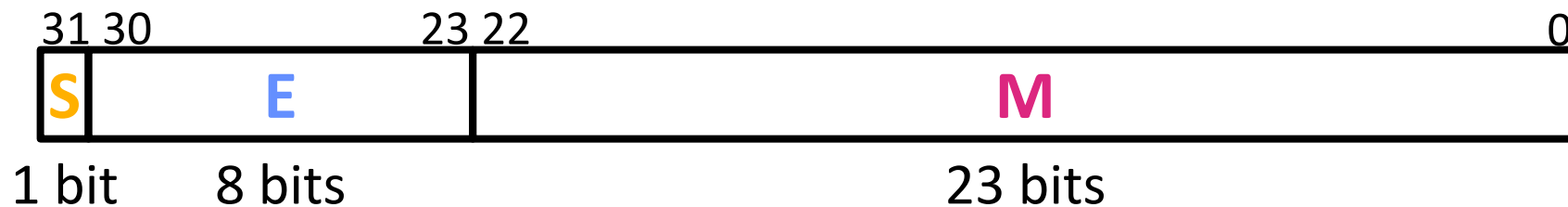
- ❖ Scientific Notation
- ❖ **IEEE 754 Floating Point Encoding**
- ❖ Floating Point Special Cases
- ❖ Floating Point Limitations and Dangers
- ❖ Floating Point in Real Life

IEEE Floating Point

- ❖ IEEE 754 (established in 1985)
 - Standard to make numerically-sensitive programs portable
 - Specifies two things: *representation scheme* and result of *floating point operations*
 - Supported by all major CPUs
- ❖ Driven by numerical concerns
 - Users (*e.g.*, scientists, numerical analysts) want them to be as **real** as possible
 - Builders want them to be **easy to implement** and **fast** 
 - Users mostly won out:
 - Nice standards for rounding, overflow, underflow, but... complex for hardware
 - Float operations can be an order of magnitude slower than integer ops → so slow that they are used as a performance gauge! (*e.g.*, FLOPS/s)

Floating Point Encoding (Review)

- ❖ C variable declared as `float`
- ❖ Use normalized, base 2 scientific notation:
 - Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$
 - Bit Fields: $(-1)^S \times 1.\text{M} \times 2^{(\text{E}-\text{bias})}$
- ❖ Representation Scheme:
 - **Sign** (bit **S** is 0 if positive, 1 if negative)
 - **Mantissa** is the fractional part of the normalized number; encoded in bit vector **M**
 - **Exponent** weights the value by a power of 2; encoded in the bit vector **E**



The Exponent Field (Review)

❖ Use **biased notation**

w=8, can encode $2^8=256$ exponents

- Read exponent as unsigned, but with **bias** of $2^{w-1}-1 = 127$
- Representable exponents roughly half positive and half negative
- $E = \text{Exp} + \text{bias} \leftrightarrow \text{Exp} = E - \text{bias}$

E: (unsigned)

Exp: (biased)



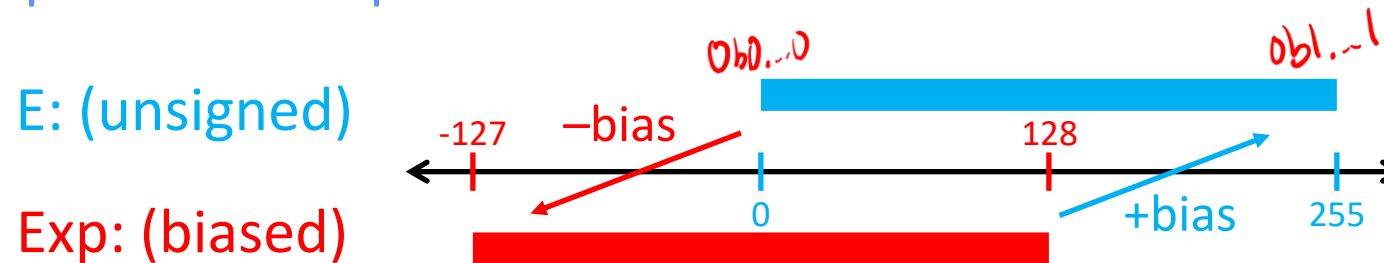
❖ Examples:

- If value has $\text{Exp} = 1$, then *encode* $1 + 127$ in unsigned, storing $E = 0b\ 1000\ 0000$
- If float has $E = 0b\ 0100\ 0000$, then we read out 64 as unsigned, shift this value to get $\text{Exp} = 64 - 127 = -63$

The Exponent Field – Why Biased?

❖ Use **biased notation**

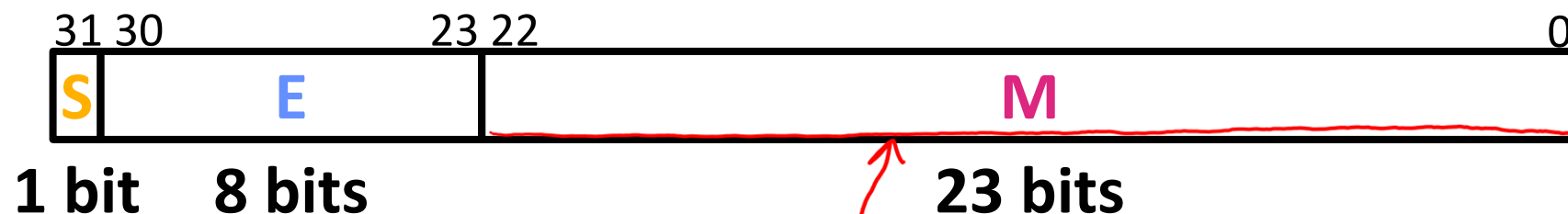
- Read exponent as unsigned, but with **bias of $2^{w-1}-1 = 127$**
- Representable exponents roughly half positive and half negative
- $E = \text{Exp} + \text{bias} \leftrightarrow \text{Exp} = E - \text{bias}$



❖ Why biased?

- Sign-and-magnitude: encodings for $\text{Exp} + \text{Man}$ are aligned with magnitude
- Makes floating point arithmetic easier (somewhat compatible with two's complement hardware)

The Mantissa/Fraction Field (Review)



$$(-1)^S \times (1 . M) \times 2^{(E - \text{bias})}$$

❖ Note the implicit leading 1 in front of the M bit vector

- Gives us an extra bit of *precision*

❖ Examples:

- Man of $1.\underline{10111}_2$ is encoded as $M = 0b \underline{101} \underline{1100} 0000 0000 0000 0000$
- $M = \underline{110} \underline{1000} 0000 0000 0000 0000$ is decoded as a Man = $1.\underline{1101}_2$

Normalized Floating Point Conversions (Review)

❖ FP \rightarrow Decimal

1. Append the bits of M to implicit leading 1 to form the mantissa.
2. Multiply the mantissa by $2^{E - \text{bias}}$.
3. Multiply the sign $(-1)^S$.
4. Multiply out the exponent by shifting the binary point.
5. Convert from binary to decimal.

❖ Decimal \rightarrow FP

1. Convert decimal to binary.
2. Convert binary to normalized scientific notation.
3. Encode sign as S (0/1).
4. Add the bias to exponent and encode E as unsigned.
5. The first bits after the leading 1 that fit are encoded into M.

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

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

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

$$2^{-4} = 0.0625$$

Polling Questions (1/2)

- ❖ What is the value encoded by the following floating point number?

0b ^S0 | ^E1000 0000 | ^M110 0000 0000 0000 0000 0000

- bias = $2^{8-1}-1 = 2^7-1 = 127$
- exponent = $E - \text{bias} = 2^7 - 127 = 128 - 127 = 1$
- mantissa = $1.M = 1.110...0_2$

$$(-1)^0 \times 1.11_2 \times 2^1 = 11.1_2 = \boxed{+3.5}$$

- ❖ Convert the decimal number $-7.375 = -1.11011 \times 2^2$ into floating point representation.

$$S = \underline{1}, E = 2 + 127 = 129 = 0b \underline{1000} \underline{0001}, M = 0b \underline{11011} \underline{0...0}$$

$$0b \underline{1100} \underline{0001} \underline{110110} \underline{0...0} = \boxed{0x \underline{C0EC} \underline{0000}}$$

Lecture Outline (3/5)

- ❖ Scientific Notation
- ❖ IEEE 754 Floating Point Encoding
- ❖ **Floating Point Special Cases**
- ❖ Floating Point Limitations and Dangers
- ❖ Floating Point in Real Life

Special Cases

❖ But wait... what happened to zero?

- *Special case:* **E** and **M** all zeros = 0
- Two zeros (sign and magnitude), but at least $0x00000000 = 0$ like integers
 $0x80000000 = -0$

❖ **E** = 0xFF, **M** = 0: $\pm \infty$

- all ones*
- e.g., division by 0
 - Still work in comparisons!

❖ **E** = 0xFF, **M** \neq 0: Not a Number (NaN)

- e.g., square root of negative number, $0/0$, $\infty - \infty$
- NaN propagates through computations
- Value of **M** can be useful in debugging (*tells you cause of NaN*)

New Representation Limits (Review)

❖ New largest value (besides ∞)?

- $E = 0xFF$ taken; next largest is $E = 0xFE$

- Largest will have $M = 0b1\dots1 \rightarrow 1.\overset{\text{23 ones}}{\color{red}1\dots1}_2 \times 2^{254-127} = 2^{128} - 2^{104}$

❖ New value closest to 0:

- $E = 0x00$ taken; next smallest is $E = 0x01$

- Smallest will have $M = 0 \rightarrow 1.\color{red}0\dots0_2 \times 2^{1-127} = 2^{-126}$

❖ Can we go smaller?

- Normalization and implicit 1 are to blame

Denorm Numbers

This is extra (non-testable) material

❖ *Special case:* $E = 0$, $M \neq 0$ are **denormalized numbers**

- No leading 1
- Uses implicit exponent of -126 even though $E = 0x00$

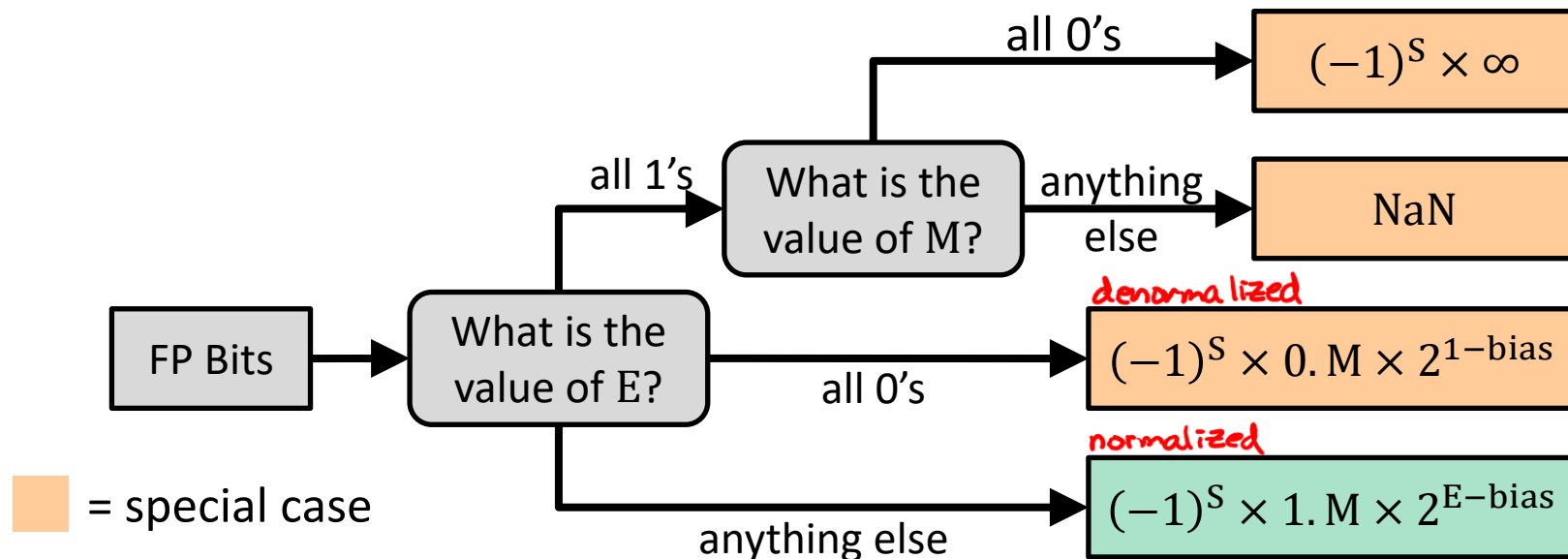
❖ Denormalized numbers close the gap between zero and the smallest normalized number

- Smallest norm: $\pm 1.0 \dots 0_{\text{two}} \times 2^{-126} = \pm 2^{-126}$
- Smallest denorm: $\pm 0.0 \dots 01_{\text{two}} \times 2^{-126} = \pm 2^{-149}$
 - There is still a gap between zero and the smallest denormalized number

So much
closer to 0

Floating Point Special Case Summary

E	M	Interpretation
0b0...0	0b0...0	± 0
0b0...0	non-zero	\pm denormalized num
everything else	anything	\pm normalized num
0b1...1	0b0...0	$\pm \infty$
0b1...1	non-zero	NaN



Lecture Outline (4/5)

- ❖ Scientific Notation
- ❖ IEEE 754 Floating Point Encoding
- ❖ Floating Point Special Cases
- ❖ **Floating Point Limitations and Dangers**
- ❖ Floating Point in Real Life

Distribution of Representable Values (Review)

❖ What ranges are NOT representable?

- Between largest norm and infinity
- Between zero and smallest denorm
- Between norm numbers?

Overflow (Exp too large)

Underflow (Exp too small)

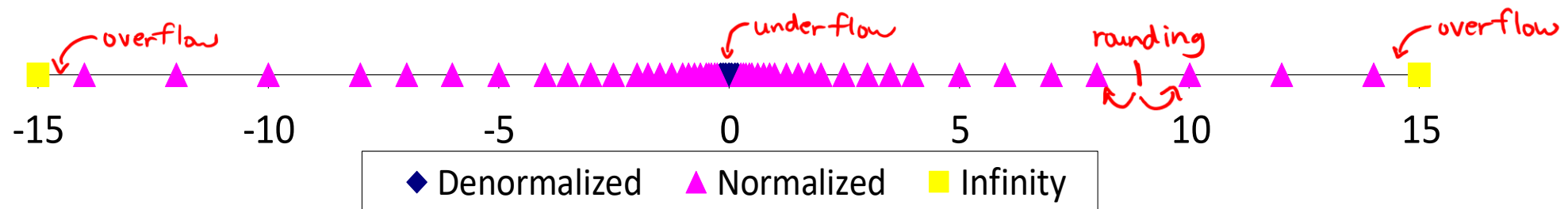
Rounding

❖ Given a FP number, what's the next largest representable number?

- What is this “step” when $\text{Exp} = 0$? 2^{-23}
- What is this “step” when $\text{Exp} = 100$? 2^{77}

if $M = 0b\ 0\dots 00$, then $2^{\text{Exp}} \times 1.0$
 if $M = 0b\ 0\dots 01$, then $2^{\text{Exp}} \times (1 + 2^{-23})$
 $\text{diff} = 2^{\text{Exp}-23}$

❖ Distribution of values is denser closer to zero:

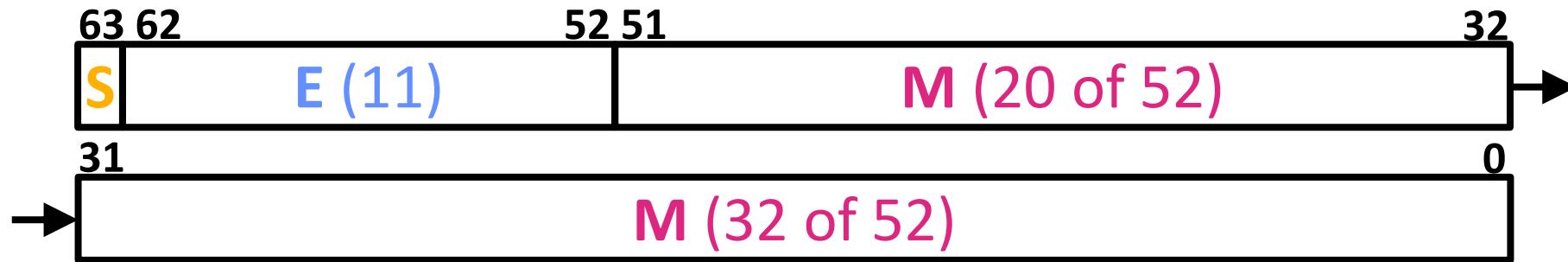


Precision and Accuracy

- ❖ **Accuracy** is a measure of the difference between the *actual value of a number* and its computer representation
- ❖ **Precision** is a count of the number of bits in a computer word used to represent a value
 - Capacity for accuracy
- ❖ *High precision permits high accuracy but doesn't guarantee it*
 - Example: **float** $\pi = 3.14$; will be represented using all 24 bits of the mantissa (highly precise), but is only an approximation (not accurate)

Need Greater Precision?

❖ **Double Precision** (vs. Single Precision) in 64 bits



- C variable declared as double
- Exponent bias is now $2^{10}-1 = 1023$, *bias = $2^{w-1}-1$*
- **Advantages:** greater precision (larger mantissa),
greater range (larger exponent)
- **Disadvantages:** more bits used,
slower to manipulate

Floating Point Arithmetic (Review)

$$\text{Value} = (-1)^S \times \text{Mantissa} \times 2^{\text{Exponent}}$$



- ❖ Basic theoretical idea for floating point operations like $+$ and \times :
 - 1) First, **compute the exact result**
 - 2) Then **encode** the result based on the specifics of your representation
 - If exponent is outside of range, then you will get *over/underflow*
 - If the exact result is not representable, then it will get *rounded* to fit the precision (width of M)

Properties of Floating Point Arithmetic (Review)

- ❖ Floats with value $\pm\infty$ and *NaN* can be used in operations
 - Result usually still $\pm\infty$ or NaN, but not always intuitive
- ❖ Floating point operations do not work like real math, due to *rounding*
 - Not associative: $(3.14 + 1e^{100}) - 1e^{100} \neq 3.14 + (1e^{100} - 1e^{100})$

0

3.14
 - Not distributive: $100 * (0.1 + 0.2) \neq 100 * 0.1 + 100 * 0.2$

30.00000000000000003553

30
 - Not cumulative: repeatedly adding a small number to a large one may do nothing
- ❖ Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!

instead use $\text{abs}(f1 - f2) < 2^{-20}$
 ↑ some arbitrary threshold

Floating Point in C

- ❖ Two common data types: `float`, `double`
- ❖ Floating point literals indicated by decimal point (`double` by default)
 - Examples: `1.0` (`double`), `1.0f` (`float`)
- ❖ Related libraries:
 - `math.h` for `INFINITY` and `NAN` constants, `float.h` for additional constants
- ❖ Casting between `int`, `float`, and `double` **changes the bit representation**
 - Tries to preserve the value, but not always reversible
 - Integral → floating point: may get rounded if not enough precision
 - Floating point → integral: fractional part will get lost/truncated

Polling Questions (2/2)

- ❖ For the following code, what is the smallest value of n that will encounter a limit of representation?

```
float f = 1.0; // 20
for (int i = 0; i < 13n; ++i)
    f *= 1024; // 1024 = 210
printf("f = %f\n", f);
```

$$E_{\max} = 0xFE, \quad \text{Exp}_{\max} = 254 - 127 = 127$$

for $n=13$, we hit 2^{130} , which causes overflow

n	f
1	2^{10}
2	2^{20}
3	2^{30}
4	2^{40}
\vdots	\vdots

always $M = 0$

Lecture Outline (5/5)

- ❖ Scientific Notation
- ❖ IEEE 754 Floating Point Encoding
- ❖ Floating Point Special Cases
- ❖ Floating Point Limitations and Dangers
- ❖ **Floating Point in Real Life**

Floating Point Issues in Real Life

❖ 1991: Patriot missile targeting error

- Time in system stored in integer (tenths of a second since boot)
- Converted to seconds by multiplying by $0.1 = 0.0 \overline{0011}_2$ leading to erroneous time (error grows the longer system has been on)



❖ 1996: V88 Ariane 501 rocket exploded 37 seconds after launch

- Reused code from Ariane 4 inertial reference platform
- Overflow when converting a 64-bit floating point number to a 16-bit integer (not protected by extra lines of code)



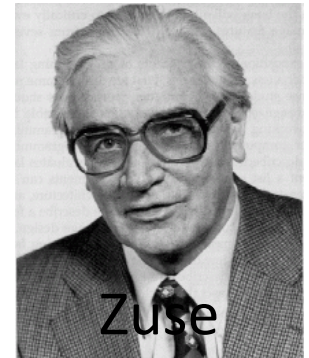
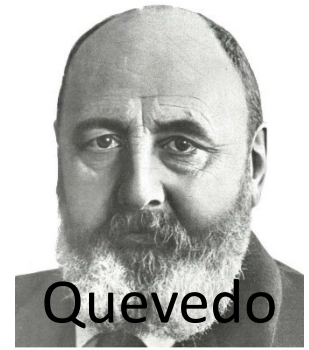
❖ Other related bugs:

- 1982: Vancouver Stock Exchange 50% error in less than 2 years due to truncation
- 1994: Intel Pentium FDIV (floating point division) hardware bug costs company \$475 million in recall

More on Floating Point History

❖ Early days

- First design with floating-point arithmetic in 1914 by Leonardo Torres y Quevedo
- Implementations started in 1940 by Konrad Zuse, but with differing field lengths (usually not summing to 32 bits) and different subsets of the special cases



❖ IEEE 754 standard created in 1985

- Primary architect was William Kahan, who won a Turing Award for this work
- Standardized bit encoding, well-defined behavior for *all* arithmetic operations



Floating Point in the “Wild”

- ❖ 3 formats from IEEE 754 standard widely used in computer hardware and languages
 - In C, called `float`, `double`, `long double`
- ❖ Common applications:
 - 3D graphics: textures, rendering, rotation, translation
 - “Big Data”: scientific computing at scale, machine learning
- ❖ Non-standard formats in domain-specific areas:
 - **Bfloat16**: training ML models; range more valuable than precision
 - **TensorFloat-32**: Nvidia-specific hardware for Tensor Core GPUs

Type	S bits	E bits	M bits	Total bits
Half-precision	1	5	10	16
Bfloat16	1	8	7	16
TensorFloat-32	1	8	10	19
Single-precision	1	8	23	32

Summary (1/2)

- ❖ Floating point approximates real numbers (large, small, & special):



- Normalized case: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}} = (-1)^S \times 1.M \times 2^{(E - \text{bias})}$

- **Mantissa** approximates fractional portion

- Size of mantissa field determines our representable **precision**
- Exceeding mantissa length causes **rounding**

- **Exponent** in biased notation (bias = $2^{w-1} - 1$)

- Size of exponent field determines our representable **range**
- Outside of representable exponents is **overflow** and **underflow**

- double (64 bits: [S (1) | E (11) | M (52)]) available if more precision needed

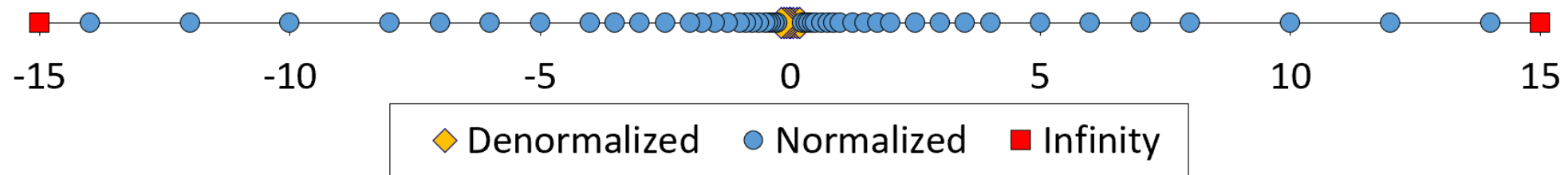
E	M	Meaning
0b0...0	anything	\pm denorm num (including 0)
anything else	anything	\pm norm num
0b1...1	0	$\pm \infty$
0b1...1	non-zero	NaN

Summary (2/2)

❖ Limitations of FP affect programmers all the time (!)

■ Overflow, underflow, rounding

- Rounding is a HUGE issue due to limited mantissa bits and gaps that are scaled by the value of the exponent



■ Floating point arithmetic is NOT associative or distributive

- ∞ and NaN are valid operands, but can produce unintuitive results

■ Do NOT use equality (==) with floating point numbers

■ Converting between integral and floating point data types *does* change the bits

- e.g., **int** i = 2; // stored as 0x00000002,
 float f = i; // stored as 0x40000000