

# DATA REPRESENTATION II

3

CS/COE 0449  
Introduction to  
Systems Software

wilkie

(with content borrowed from Vinicius Petrucci  
and Jarrett Billingsley)

Spring 2019/2020

# BIT MANIPULATION

Flippin' Switches

# What are "bitwise" operations?

- The "numbers" we use on computers aren't *really* numbers right?
- It's often useful to treat them instead as a pattern of bits.
- Bitwise operations treat a value as a pattern of bits.



1



0



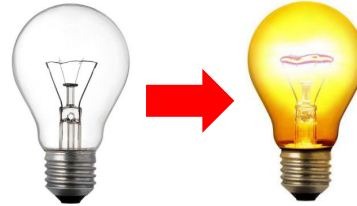
0



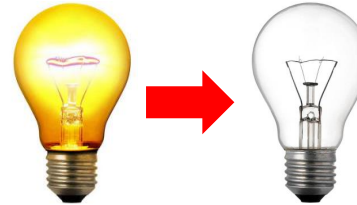
0

# The simplest operation: NOT (logical negation)

- If the light is off, turn it on.



- If the light is on, turn it off.

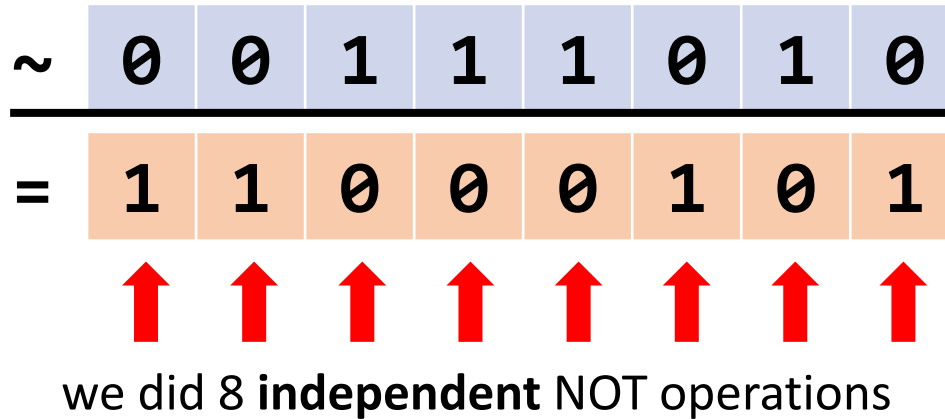


A	Q
0	1
1	0

- We can summarize this in a truth table.
- We write NOT as  $\sim A$ , or  $\neg A$ , or  $\bar{A}$
- In C, the NOT operation is the “!” operator

# Applying NOT to a whole bunch of bits

- If we use the not instruction (`~` in C), this is what happens:

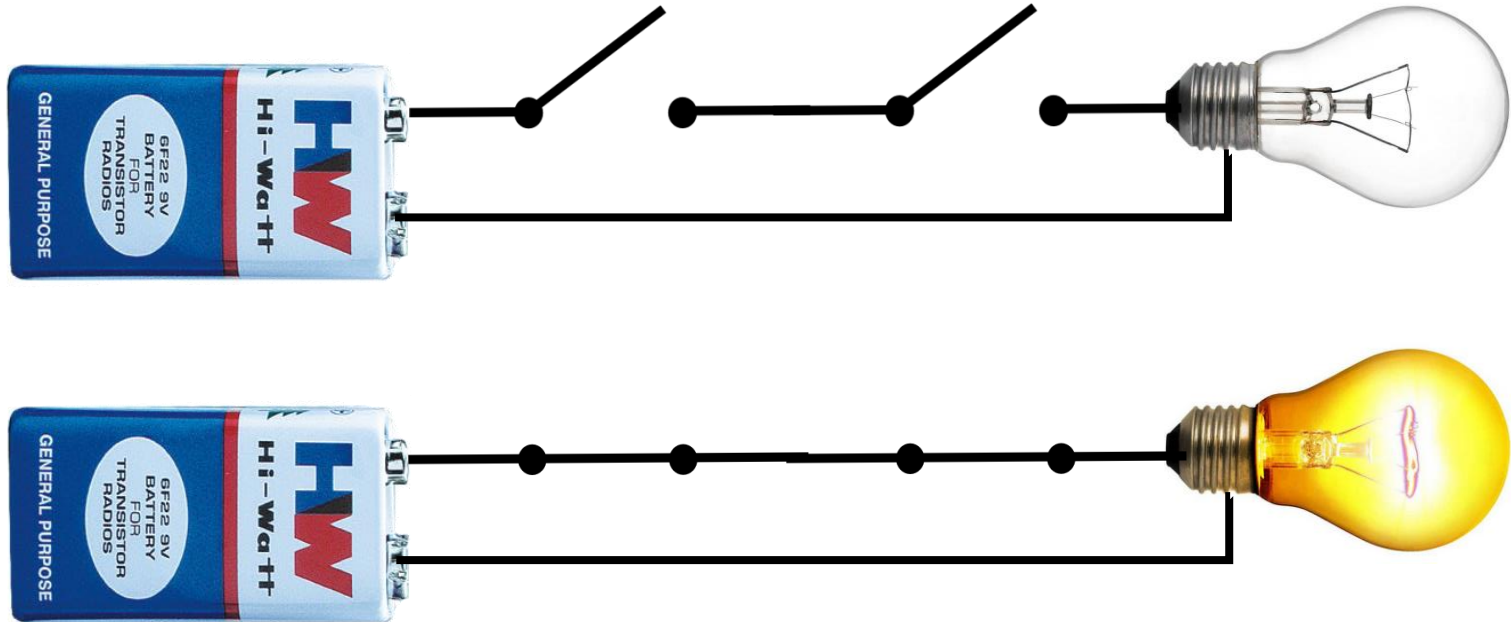


That's it.

only 8 bits shown cause 32 bits on a slide is too much

# Let's add some switches

- There are two switches in a row connecting the light to the battery.
- How do we make it light up?



# AND (Logical product)

- AND is a binary (two-operand) operation.
- It can be written a number of ways:  
 $A \& B$     $A \wedge B$     $A \cdot B$     $AB$
- If we use the and instruction (& in C):

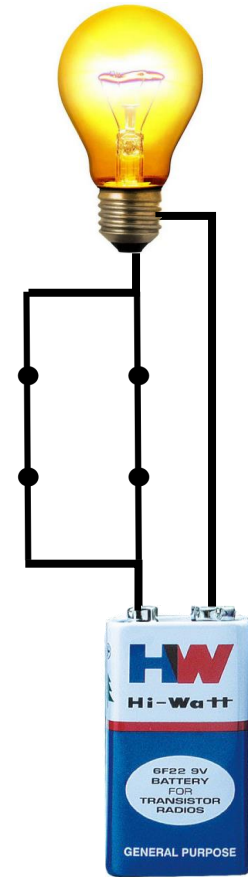
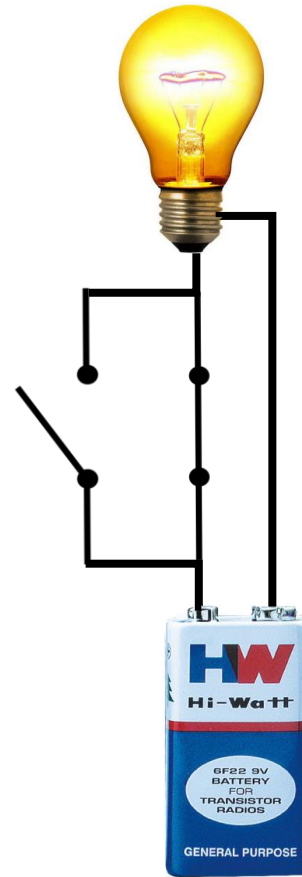
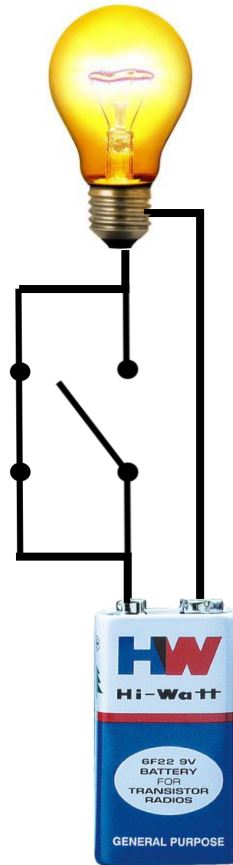
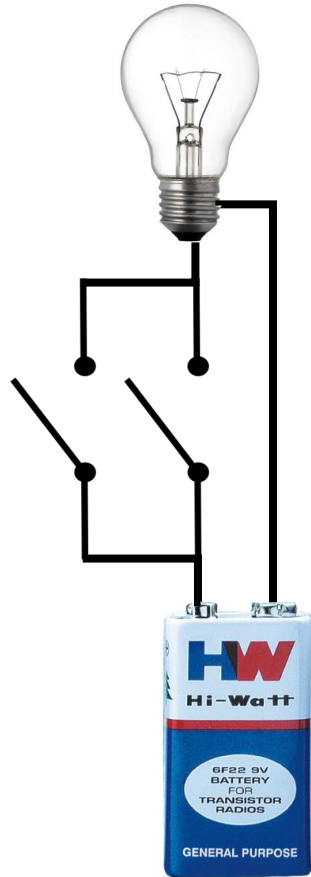
	1	1	1	1	0	0	0	0
&	0	0	1	1	1	0	1	0
<hr/>								
=	0	0	1	1	0	0	0	0
	↑	↑	↑	↑	↑	↑	↑	↑

we did 8 **independent** AND operations

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

# "Switching" things up ;))))))))))))))))))









- NOW how can we make it light up?





# OR ("Logical" sum...?)

- We might say "and/or" in English.
- It can be written a number of ways:  
 $A \mid B$     $A \vee B$     $A + B$
- If we use the or instruction (  $\mid$  in C):

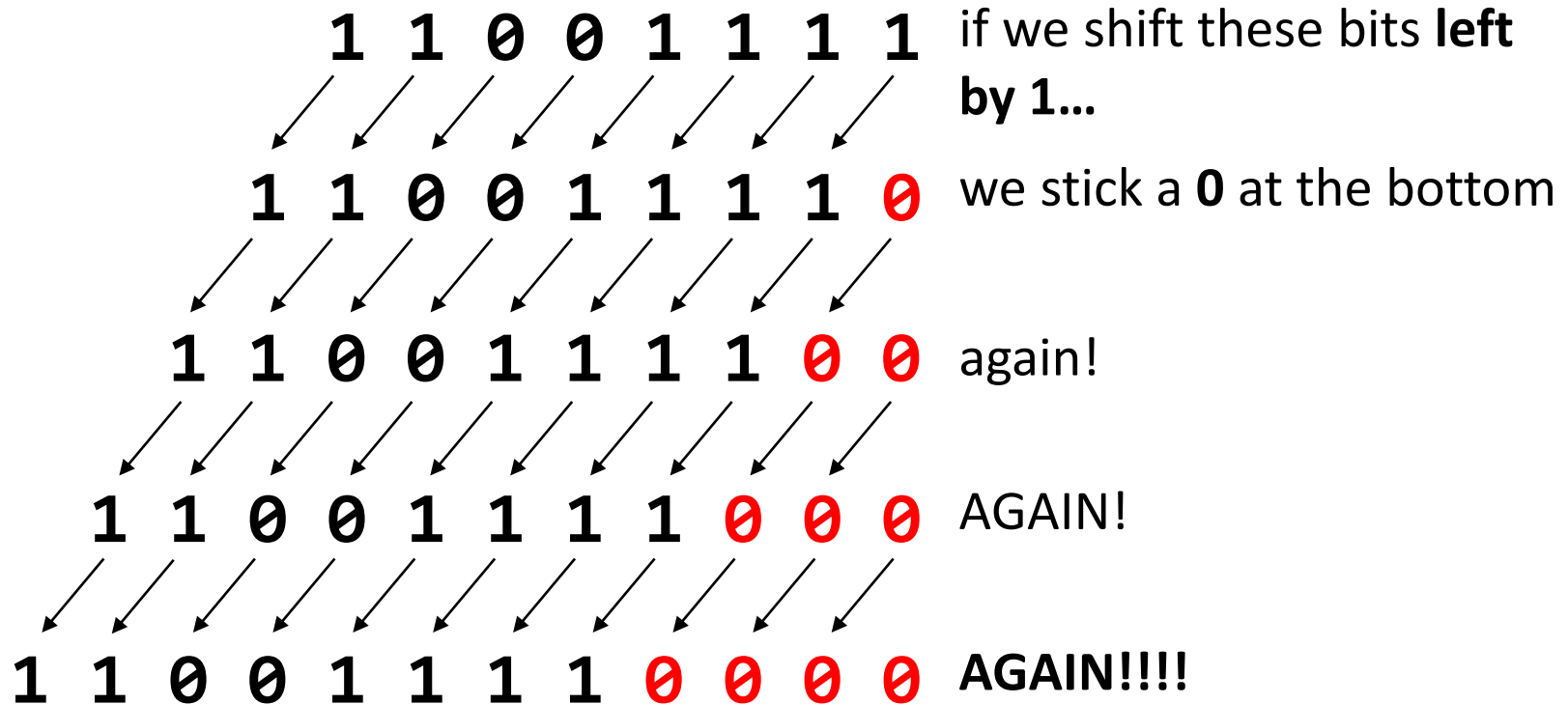
	1	1	1	1	0	0	0	0
$\mid$	0	0	1	1	1	0	1	0
<hr/>								
=	1	1	1	1	1	0	1	0
								

We did 8 **independent** OR operations.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

# Bit shifting

- Besides AND, OR, and NOT, we can move bits around, too.



# Left-shifting in C/Java

(animated)

- C (and Java) use the << operator for left shift

**B = A << 4;** // *B = A shifted left 4 bits*

If the bottom 4 bits of the result are now 0s...

- ...what happened to the *top* 4 bits?

**0011 0000 0000 1111 1100 1101 1100 1111**

the bit bucket is not a real place

it's a programmer joke ok

in the UK they might say the “Bit Bin”

bc that's their word for trash





- We can shift right, too

0	0	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	1	1	1	0	0	1	1	1	1
0	0	0	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	1	1	1	0	0	1	1	1
0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	1	1	1	0	0	1	1
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	1	1	1	0	0	1
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	1	1	1	0	0

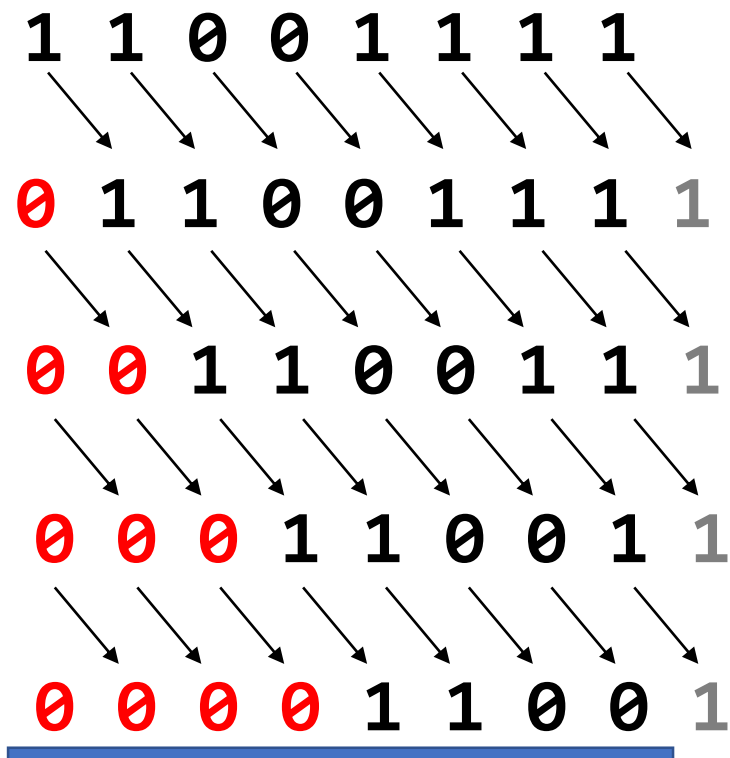
- C/Java use `>>`, this in MIPS is the **srl** (**S**hift **R**ight **L**ogical) instruction

see what I mean about 32 bits on a slide

**Q:** What happens when we shift a negative number to the right? 12

# Shift Right (Logical)

- We can shift right, too (srl in MIPS)



if we shift these bits **right**  
**by 1...**

we stick a **0** at the top

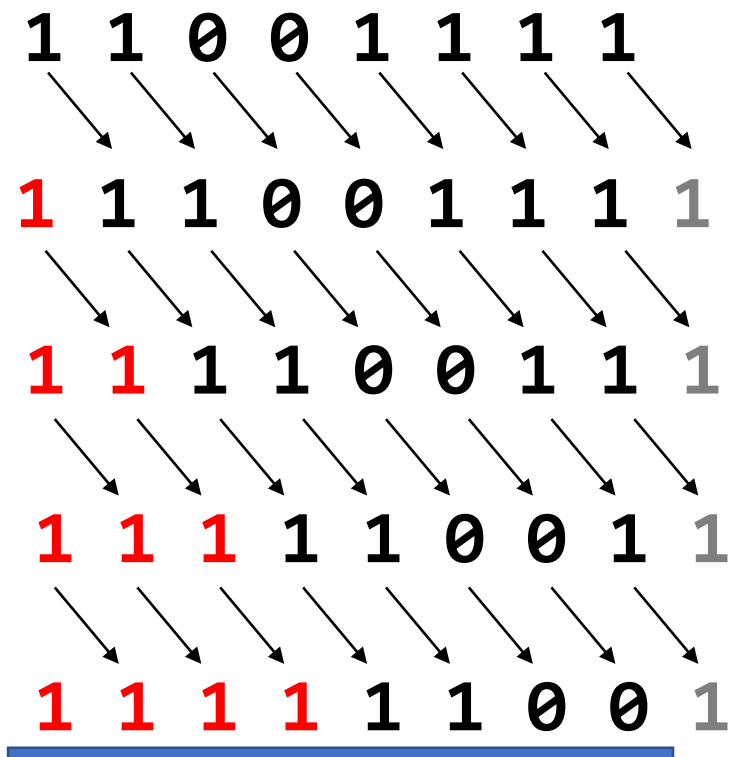
again!

AGAIN!

**Wait... what if this was a  
negative number?**

# Shift Right (Arithmetic)

- We can shift right with sign-extension, too (MIPS: sra)



if we shift these bits **right**  
**by 1...**

we copy the **1** at the top (or 0,  
if MSB was a 0)

again!

AGAIN!

**AGAIN!!!!!!** (It's still  
negative!)

# Huh... that's weird

- Let's start with a value like 5 and shift left and see what happens:

Binary	Decimal
101	5
1010	10
10100	20
101000	40
1010000	80

Why is this happening

Well uh... what if I gave you

**49018853**

How do you multiply that by 10?

by 100?

by 100000?

Something **very similar** is  
happening here

$$a \ll n == a * 2^n$$

- Shifting left by  $n$  is the same as multiplying by  $2^n$ 
  - You probably learned this as "moving the decimal point"
    - And moving the decimal point *right* is like shifting the digits *left*
- Shifting is fast and easy on most CPUs.
  - Way faster than multiplication in any case.
  - (It's not a great reason to do it when you're using C though)
- Hey... if shifting *left* is the same as multiplying...



# $a \gg n == a / 2^n$ , ish

- You got it
- Shifting right by  $n$  is like dividing by  $2^n$ 
  - *sort of.*
- What's  $101_2$  shifted right by 1?
  - $10_2$ , which is 2...
    - It's like doing **integer** (or **flooring**) division
- Generally, compilers are smart enough that you just multiply/divide
  - It's confusing to shift just to optimize performance.
  - It's good to not be clever until it is proven that you need to be.

# C Bitwise Operations: Summary

C code	Description	MIPS instruction
<code>x   y</code>	or	<code>or x, x, y</code>
<code>x &amp; y</code>	and	<code>and x, x, y</code>
<code>x ^ y</code>	xor	<code>xor x, x, y</code>
<code>!x</code>	not	<code>seq x, x, \$0 (“seqz”)</code>
<code>~x</code>	complement (negate)	<code>nor x, x, \$0 (“not”)</code>
<code>x &lt;&lt; y</code>	left-shift logical	<code>sll x, x, y</code>
<code>x &gt;&gt; y</code>	right-shift logical	<code>srl x, x, y</code>

**When x is signed (most of the time...):**

<code>x &gt;&gt; y</code>	right-shift arithmetic	<code>sra x, x, y</code>
---------------------------	------------------------	--------------------------

# FRACTIONAL ENCODING

Every Time I Teach Floats I Want Some Root Beer

# Fixing the point

- If we want to represent decimal places, one way of doing so is by assuming that the lowest  $n$  digits are the decimal places.

$$\begin{array}{r} \$12.34 \\ + \$10.81 \\ \hline \$23.15 \end{array}$$

$$\begin{array}{r} 1234 \\ + 1081 \\ \hline 2315 \end{array}$$

this is called **fixed-point representation**

# A rising tide

- A 16.16 fixed-point number looks like this:

0011 0000 0101 1010.1000 0000 1111 1111

*binary* point

the largest (signed) value we  
can represent is +32767.999

the smallest fraction we can  
represent is  $1/65536$

But if we let the binary point **float around**...

0011.0000 0101 1010 1000 0000 1111 1111

...we can get much higher **accuracy** near 0...

0011 0000 0101 1010 1000 0000.1111 1111

...and trade off accuracy for **range** further away from 0.

# IEEE 754

- Established in 1985, updated as recently as 2008.
- Standard for floating-point representation and arithmetic that virtually every CPU now uses.
- Floating-point representation is based around scientific notation:

$$\begin{aligned} 1348 &= +1.348 \times 10^{+3} \\ -0.0039 &= -3.9 \times 10^{-3} \\ -1440000 &= -1.44 \times 10^{+6} \end{aligned}$$



sign   significand   exponent

# Binary Scientific Notation

- scientific notation works equally well in any other base!
  - (below uses base-10 exponents for clarity)

$$+1001\ 0101 = +1.001\ 0101 \times 2^{+7}$$

$$-0.001\ 010 = -1.010 \times 2^{-3}$$

$$-1001\ 0000\ 0000\ 0000 = -1.001 \times 2^{+15}$$



what do you notice  
about the digit before  
the **binary** point?

$$(+/-)1.f \times 2^{\text{exp}}$$

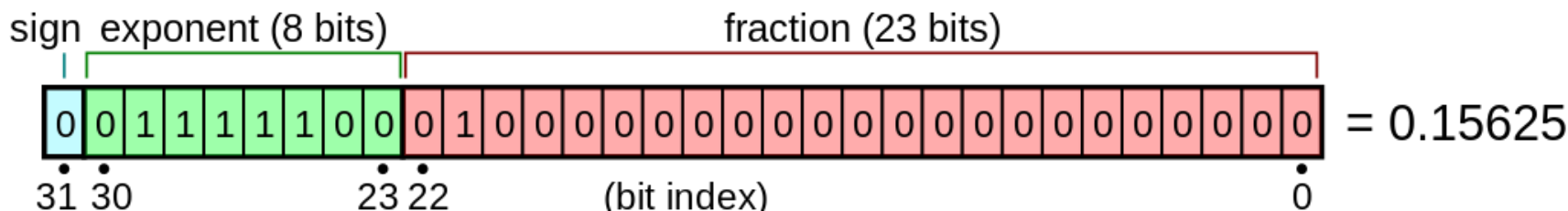
**f** – fraction

**1.f** – significand

**exp** – exponent

# IEEE 754 Single-precision

- Known as float in C/C++/Java etc., 32-bit float format
- 1 bit for sign, 8 bits for the exponent, 23 bits for the *fraction*

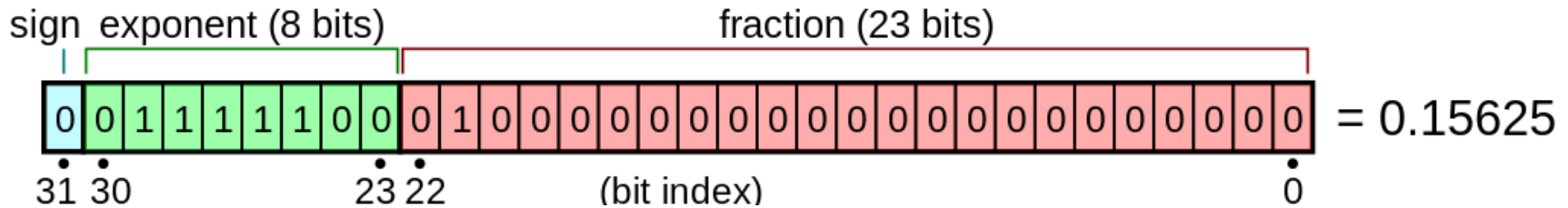


- Tradeoff:
  - More accuracy ? More fraction bits
  - More range ? More exponent bits
- Every design has tradeoffs ˘(ツ)˘/
  - Welcome to Systems!



# IEEE 754 Single-precision

- Known as float in C/C++/Java etc., 32-bit float format
- 1 bit for sign, 8 bits for the exponent, 23 bits for the *fraction*



- The fraction field only stores the digits after the binary point
- The 1 before the binary point is implicit!
  - This is called normalized representation
  - In effect this gives us a 24-bit significand
  - The only number with a 0 before the binary point is 0!
- The significand of floating-point numbers is in sign-magnitude!
  - Do you remember the downside(s)?

# The exponent field

- The exponent field is 8 bits, and can hold positive or negative exponents, but... it doesn't use S-M, 1's, or 2's complement.
- It uses something called biased notation.
  - biased representation = signed number + *bias constant*
  - single-precision floats use a bias constant of **127**

$$\mathbf{-127 \Rightarrow 0}$$

Signed	$\mathbf{-10 \Rightarrow 117}$	Biased
	$\mathbf{34 \Rightarrow 161}$	

- The exponent can range from -126 to +127 (1 to 254 biased)
  - 0 and 255 are reserved!
- Why'd they do this?
  - so you can sort floats with integer comparisons??

# Binary Scientific Notation (revisited)

- Our previous numbers are actually

$$\begin{array}{llll} +1.001\ 0101 & \times 2^{+7} & = (-1)^0 \times \textcolor{red}{1}.001\ 0101 & \times 2^{134-\textcolor{red}{127}} \\ -1.010 & \times 2^{-3} & = (-1)^1 \times \textcolor{red}{1}.010 & \times 2^{124-\textcolor{red}{127}} \\ -1.001 & \times 2^{+15} & = (-1)^1 \times \textcolor{red}{1}.001 & \times 2^{142-\textcolor{red}{127}} \end{array}$$

$$(-1)^s \times 1.f \times 2^{\text{exp}-\textcolor{red}{127}}$$

s – sign

f – fraction

exp – biased exponent

# Encoding an integer as a float

- You have an integer, like  $2471 = 0000\ 1001\ 1010\ 0111_2$ 
  1. put it in scientific notation
    - $1.001\ 1010\ 0111_2 \times 2^{+11}$
  2. get the exponent field by adding the bias constant
    - $11 + 127 = 138 = 10001010_2$
  3. copy the bits **after the binary point** into the fraction field

s	exponent	fraction
0	10001010	001101001110000000...000



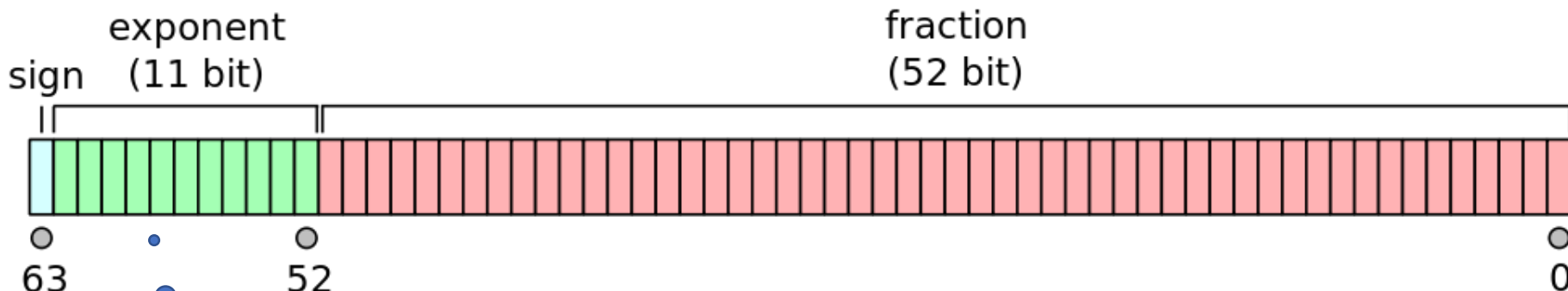
positive



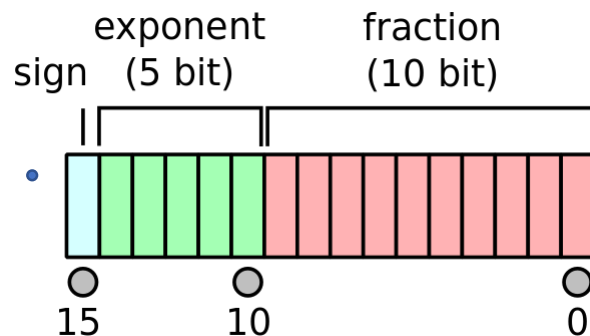
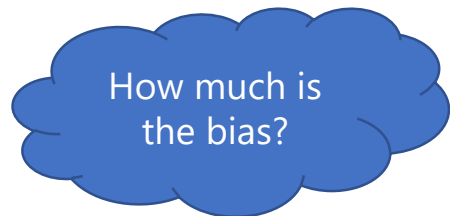
start at the **left** side!

# Other formats

- The most common other format is double-precision (C/C++/Java double), which uses an 11-bit exponent and 52-bit fraction



- GPUs have driven the creation of a half-precision 16-bit floating-point format. it's adorable



# This could be a whole unit itself...

- Floating-point arithmetic is COMPLEX STUFF.
- But it's not super useful to know unless you're either:
  - Doing lots of high-precision numerical programming, or
  - Implementing floating-point arithmetic yourself.
- However...
  - It's good to have an understanding of *why* limitations exist.
  - It's good to have an *appreciation* of how complex this is... and how much better things are now than they were in the 1970s and 1980s!
  - It's good to know things do not behave as expected when using float and double!!