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

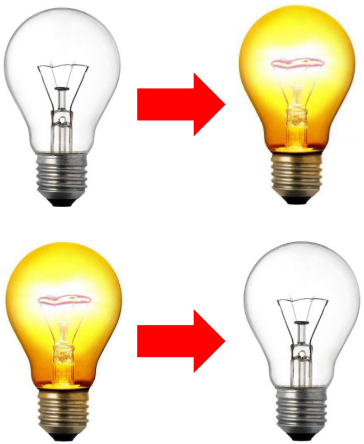
What are "bitwise" operations?

- 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 ~A, or ¬A, or $\overline{\mathbf{A}}$

- 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:

~	0	0	1	1	1	0	1	0
=	1	1	0	0	0	1	0	1

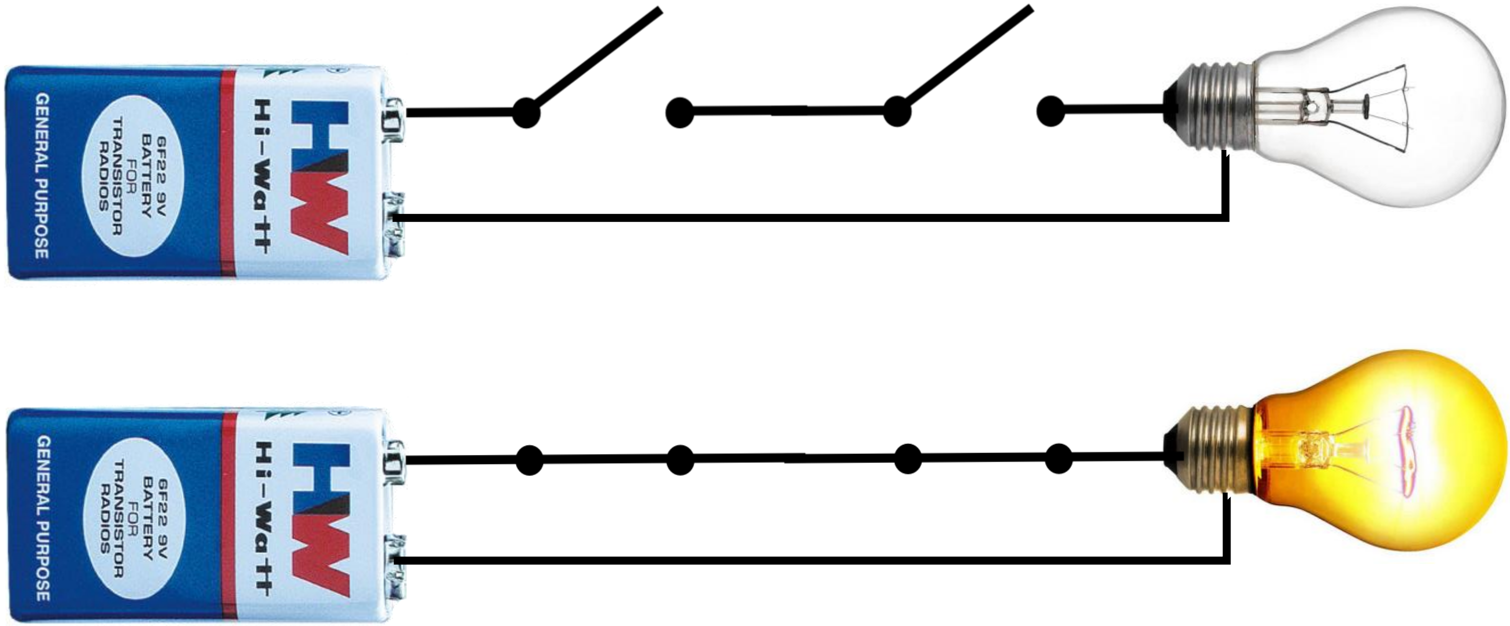
we did 8 **independent** NOT operations

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∧B   A·B   AB

- If we use the and instruction (& in C):

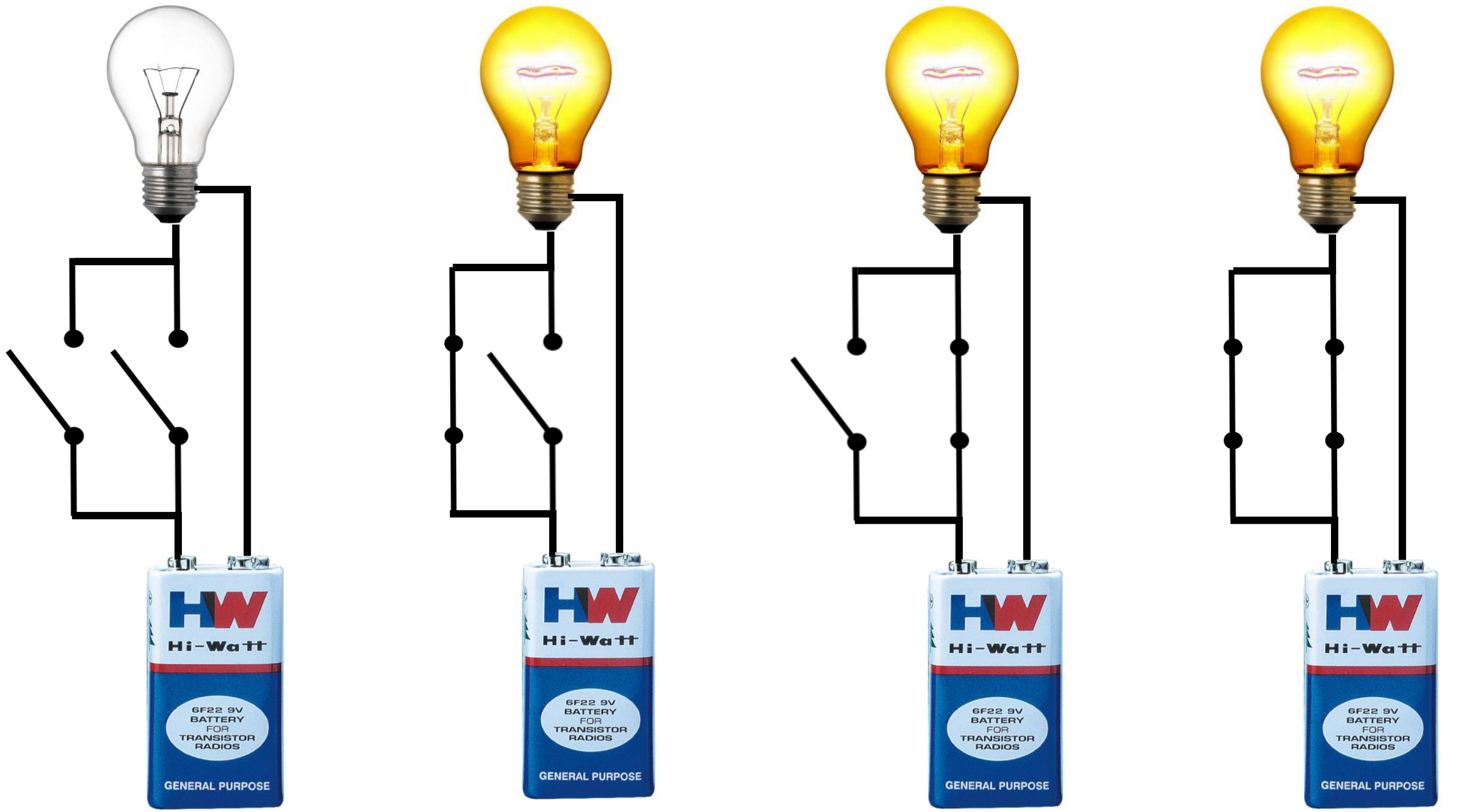
	1	1	1	1	0	0	0	0
&	0	0	1	1	1	0	1	0
=	0	0	1	1	0	0	0	0

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|B` `AVB` `A+B`

- If we use the or instruction (| in C):

	1	1	1	1	0	0	0	0
|	0	0	1	1	1	0	1	0
=	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.

`1 1 0 0 1 1 1 1` if we shift these bits **left by 1...**

`1 1 0 0 1 1 1 1 0` we stick a **0** at the bottom

`1 1 0 0 1 1 1 1 0 0` again!

`1 1 0 0 1 1 1 1 0 0 0` AGAIN!

`1 1 0 0 1 1 1 1 0 0 0 0` **AGAIN!!!!**

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 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
```

- 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?

Shift Right (Logical)

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

```
1 1 0 0 1 1 1 1
0 1 1 0 0 1 1 1 1
0 0 1 1 0 0 1 1 1
0 0 0 1 1 0 0 1 1
0 0 0 0 1 1 0 0 1
```

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

we stick a **0** at the top

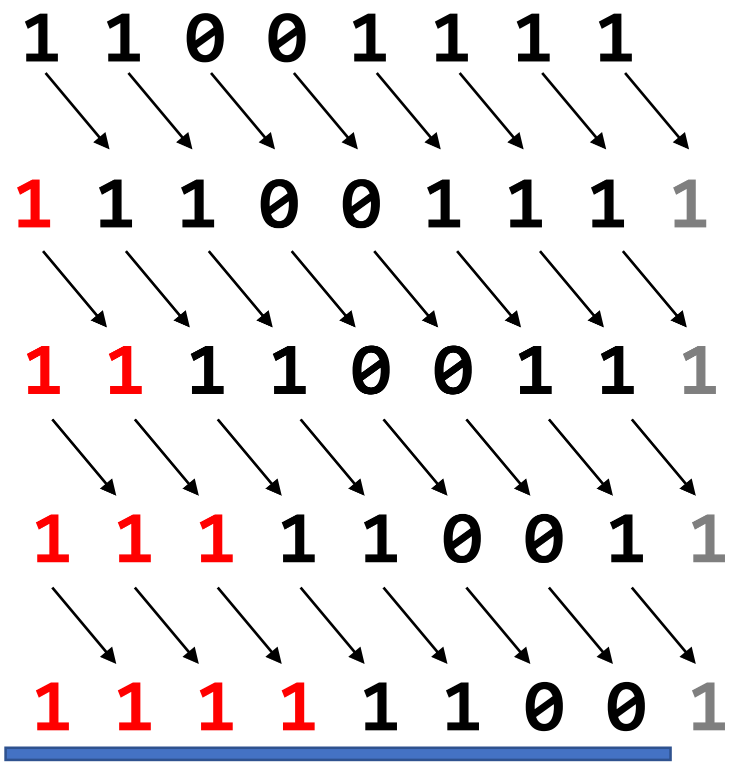
again!

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 << 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…**

$a >> n == a / 2^n$, 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
`x | y`	`or`	`or  x, x, y`
`x & y`	`and`	`and x, x, y`
`x ^ y`	`xor`	`xor x, x, y`
`!x`	`not`	`seq x, x, $0` ("seqz")
`~x`	`complement (negate)`	`nor x, x, $0` ("not")
`x << y`	`left-shift logical`	`sll x, x, y`
`x >> y`	`right-shift logical`	`srl x, x, y`

When x is signed (most of the time…):

`x >> y`	`right-shift arithmetic`	`sra x, x, y`

Fractional Encoding

Every Time I Teach Floats I Want Some Root Beer

Fractional numbers

- **Up to this point we have been working with integer numbers.**
  - Unsigned and signed!

2 0 1 9

- **However, Real world numbers are… Real numbers. Like so:**

2 0 1 9.320

- **That creates new challenges!**
  - Let’s start by taking a look at them.

Just a fraction of a number

- The numbers we use are written positionally: the position of a digit within the number has a meaning.
- What about when the numbers go over the decimal point?

2	0	1	9	.	3	2	0	???
1000s	100s	10s	1s		10ths	100ths	1000ths	
10^3	10^2	10^1	10^0		$10^{(-1)}$	$10^{(-2)}$	$10^{(-3)}$	

A fraction of a bit?

- Binary is the same!
- Just replace 10s with 2s.

0	1	1	0	.1	1	0	1
2^3	2^2	2^1	2^0	$2^{(-1)}$	$2^{(-2)}$	$2^{(-3)}$	$2^{(-4)}$
8s	4s	2s	1s	2ths?	4ths	8ths	16ths

To convert into decimal, just add stuff

$$0\ 1\ 1\ 0\ .1\ 1\ 0\ 1 =$$

2^3 2^2 2^1 2^0 $2^{(-1)}$ $2^{(-2)}$ $2^{(-3)}$ $2^{(-4)}$

$0 \times 8 +$
$1 \times 4 +$
$1 \times 2 +$
$0 \times 1 +$
$1 \times .5 +$
$1 \times .25 +$
$0 \times .125 +$
$1 \times .0625$

$= 6.8125_{10}$

From decimal to binary? Tricky?

$$\underline{6}.\underline{8125}_{10}$$

$6 \div 2_{10} = 3R0$

$3 \div 2_{10} = 1R1$

$$
\begin{array}{r}
0.8125_{10} \\
\underline{\text{x} \qquad 2} \\
1.6250 \\
\\
0.6250_{10} \\
\underline{\text{x} \qquad 2} \\
1.2500 \\
\\
0.2500_{10} \\
\underline{\text{x} \qquad 2} \\
0.5000 \\
\\
0.5000_{10} \\
\underline{\text{x} \qquad 2} \\
1.0000
\end{array}
$$

MSB

LSB

$$1\,1\,0.1101$$

So, it's easy right? Well...

What about: 0.1_{10}

$$
\begin{array}{l}
0.1_{10} \\
\underline{\times\ 2} \\
0.2 \\[6pt]
0.2_{10} \\
\underline{\times\ 2} \\
0.4 \\[6pt]
0.4_{10} \\
\underline{\times\ 2} \\
0.8 \\[6pt]
0.8_{10} \\
\underline{\times\ 2} \\
1.6
\end{array}
$$

0.0001

So, it's easy right? Well……

What about: 0.1_{10}

0.0001

1001

$$\begin{array}{r} 0.6_{10} \\ \underline{\times\ 2} \\ 1.2 \end{array} \qquad \begin{array}{r} 0.1_{10} \\ \underline{\times\ 2} \\ 0.2 \end{array}$$

$$\begin{array}{r} 0.2_{10} \\ \underline{\times\ 2} \\ 0.4 \end{array} \qquad \begin{array}{r} 0.2_{10} \\ \underline{\times\ 2} \\ 0.4 \end{array}$$

$$\begin{array}{r} 0.4_{10} \\ \underline{\times\ 2} \\ 0.8 \end{array} \qquad \begin{array}{r} 0.4_{10} \\ \underline{\times\ 2} \\ 0.8 \end{array}$$

$$\begin{array}{r} 0.8_{10} \\ \underline{\times\ 2} \\ 1.6 \end{array} \qquad \begin{array}{r} 0.8_{10} \\ \underline{\times\ 2} \\ 1.6 \end{array}$$

So, it's easy right? Well.........

What about: 0.1_{10}

0.0001
1001
1001
...

0.6_{10}	0.6_{10}	0.1_{10}
x 2	x 2	x 2
1.2	1.2	0.2
0.2_{10}	0.2_{10}	0.2_{10}
x 2	x 2	x 2
0.4	0.4	0.4
0.4_{10}	0.4_{10}	0.4_{10}
x 2	x 2	x 2
0.8	0.8	0.8
0.8_{10}	0.8_{10}	0.8_{10}
x 2	x 2	x 2
1.6	1.6	1.6

WELL...

$0.1_{10} =$ 0.000110011001100110011001100110011001100110011001100110
0110011001100110011001100110011001100110011001100110
0110011100110011001100110011001100110011001100110011
0100110011001100110011001100110011001100110011001100
1100110100110011001100110011001100110011001100110011
0011001100110100110011001100110011001100110011001100
110011001100110011001100110011001100110011001100110011001100110011001
100110011001100110011001100110011001100110011001100110011001100110011
001100110011001100110011001100110011001100110011001100110011001100110
011001100110011001100110011001100110011001100110011001100110011001100
110011001100110011001100110011001100110011001100110011001100110011001
100110011001100110011001100110011001100110011001100110011001100110011
001100110011001100110011001100110011001100110011001100110011001100110
011001100110011001100110011001100110011001100110011001100110011001100
110011001100110011001100110011001100110011001100110011001100110011001
100110011001100110011001100110011001100110011001100110011001100110011
001100110011001100110011001100110011001100110011001100110011001100110
011001100110011001100110011001100110011001100110011001100110011001100
110011001100110011001100110011001100110011001100110011001100110011001
100110011001100110011001100110011001100110011001100110011001100110011
001100110011001100110011001100110011001100110011001100110011001100110
011001100110011001100110011001100110011001100110011001100110011001100
110011001100110011001100110011001100110011001100110011001100110011001
100110011001100110011001100110011001100110011001100110011001100110011
001100110011001100110011001100110011001100110011001100110011001100110
011001100110011001100110011001100110011001100110011001100110011001100
110011001100110011001100110011001100110011001100110011001100110011001
10011001100110011001100110011001100110011001100110011001100110011001...

How much is it worth?

- Well, it depends on where you stop!

$$0.0001_2 = 0.0625$$

$$0.00011001_2 = 0.0976\ldots$$

$$0.000110011001_2 = 0.0998\ldots$$

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.

$$\begin{array}{r} \$12.34 \\ +\$10.81 \\ \hline \$23.15 \end{array} \qquad \begin{array}{r} 1234 \\ +1081 \\ \hline 2315 \end{array}$$

this is called ***fixed-point representation***

A rising tide

- Maybe half-and-half? 16.16 number looks like this:

0011 0000 0101 1010.1000 0000 1111 1111

binary point

the largest (signed) value we can represent is +32767.9999ish

the smallest fraction we can represent is 1/65536

What if we place the binary point to the left…

0011.0000 0101 1010 1000 0000 1111 1111

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

…but if we place the binary point to the right…

0011 0000 0101 1010 1000 0000.1111 1111

…then we trade off accuracy for ***range*** further away from 0.

Move the point

- What if we could float the point around?
  - Enter scientific notation: The number `-0.0039` can be represented:

$$-0.39 \times 10^{-2}$$

$$-3.9 \times 10^{-3}$$

- These both represent the same number, but we need to move the decimal point according to the power of ten represented.

- The bottom example is in normalized scientific notation.
  - There is only ***one non-zero*** digit to the left of the point.

- Because the decimal point can be moved, we call this representation:

Floating point

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)

$$+1001\ 0101 = +1.001\ 0101 \times 2^{+7}$$
$$-0.001\ 010 = -1.010 \times 2^{-3}$$
$$-1001\ 0000\ 0000\ 0000 = -1.001 \times 2^{+15}$$

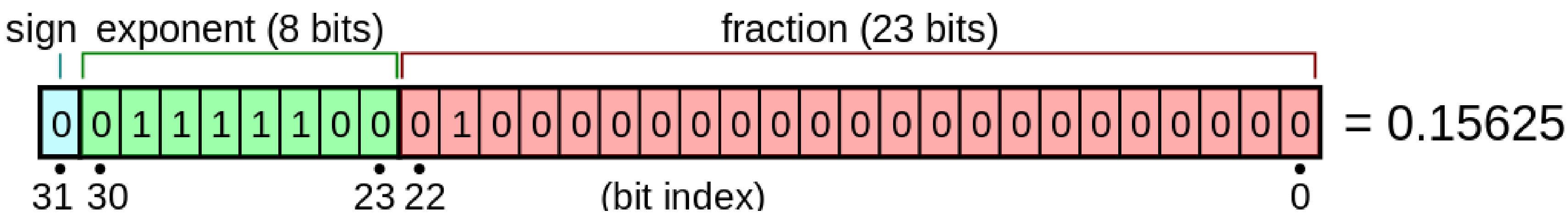
What do you notice
about the digit before
the **binary** point?

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

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***

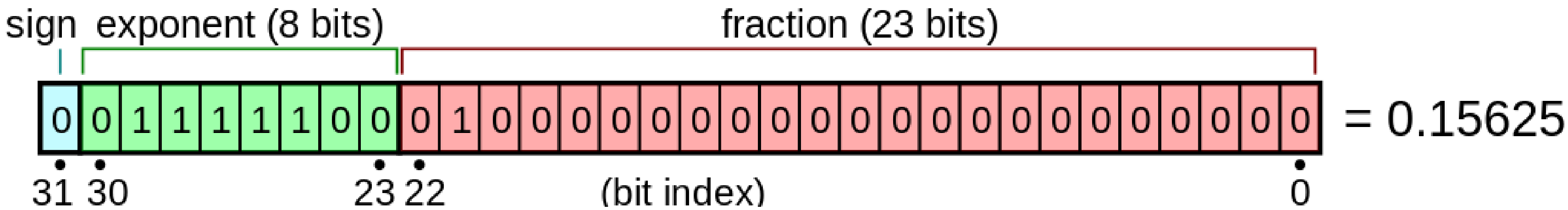


- **Tradeoff:**
  - More accuracy = More fraction bits
  - More range = More exponent bits
- **Every design has tradeoffs ¯_(ツ)_/¯**
  - Welcome to Systems!

illustration from user Stannered on Wikimedia Commons

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)?

illustration from user Stannered on Wikimedia Commons

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**

Signed		Biased
	-127 => 0	
	-10 => 117	
	34 => 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

$$+1.001\ 0101 \times 2^{+7} = (-1)^0 \times 1.001\ 0101 \times 2^{134-127}$$
$$-1.010 \times 2^{-3} = (-1)^1 \times 1.010 \times 2^{124-127}$$
$$-1.001 \times 2^{+15} = (-1)^1 \times 1.001 \times 2^{142-127}$$

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

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
   - $\mathtt{1.001\ 1010\ 0111}_2 \times 2^{+11}$
2. get the exponent field by adding the bias constant
   - $11 + 127 = 138 = \mathtt{10001010}_2$
3. copy the bits **after the binary point** into the fraction field

s	exponent	fraction
0	10001010	00110100111000000...000

positive

start at the **left** side!

Encoding a number as a float

You have a number, like -12.59375_{10}

1. **Convert to binary:** Integer part: 1100_2 Fractional part: 0.10011_2
2. **Write it in scientific notation:** $1100.10011_2 \times 2^0$
3. **Normalize it:** $1.10010011_2 \times 2^3$
4. **Calculate biased exponent** $+3 + 127 = 130_{10} = 10000010_2$

s	exponent	fraction
1	10000010	10010011000000000...000

0xC1498000

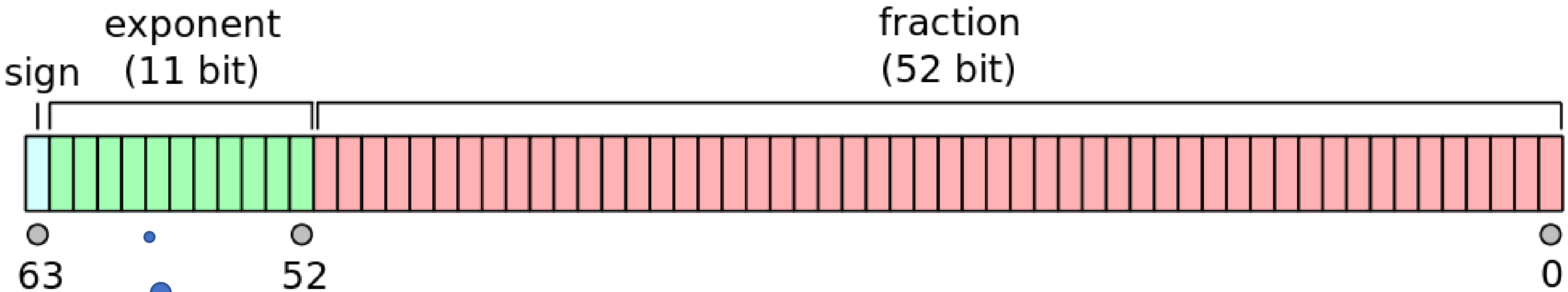
while (computers don't do real math) { … }

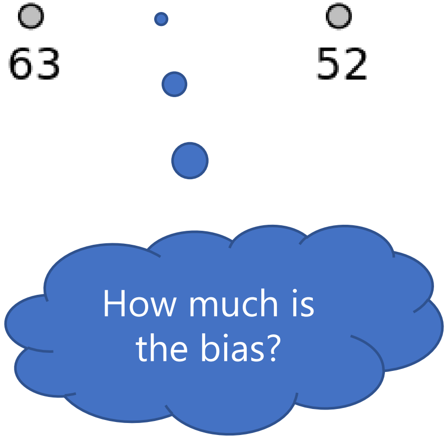
```java
public class FloatTest {
  public static void main(String[] args) {
    double var = Double.MAX_VALUE;

    while (var == Double.MAX_VALUE) {
      // Hang the program while the condition holds
      var = var + 0.1;
    }

    System.out.println("This never prints.");
  }
}
```

Q: Consider and/or review the IEEE 754 standard. What is happening here?

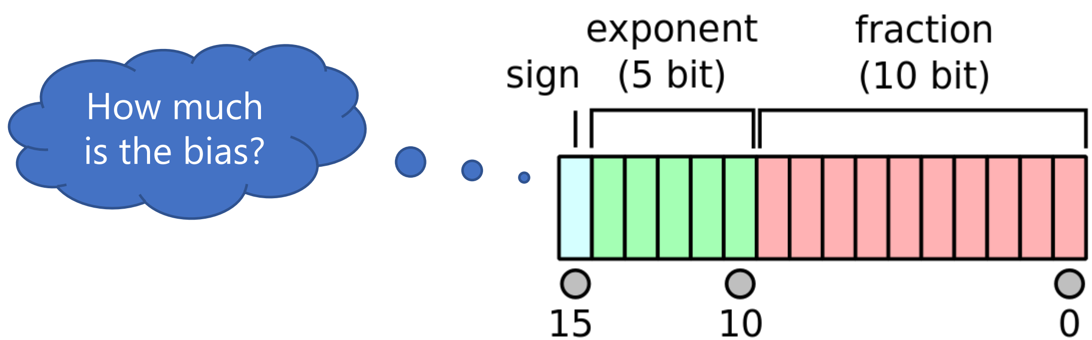
Other formats

- 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



both illustrations from user Codekaizen on Wikimedia Commons

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!!