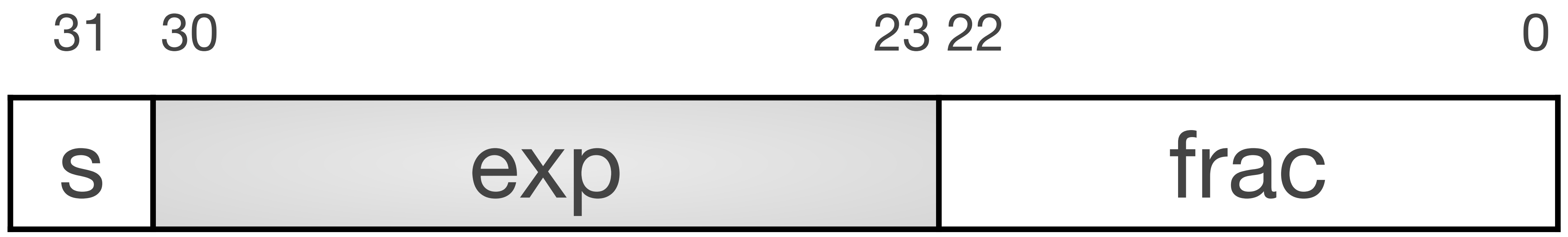
CS 107e
Sound / Floating Point Numbers

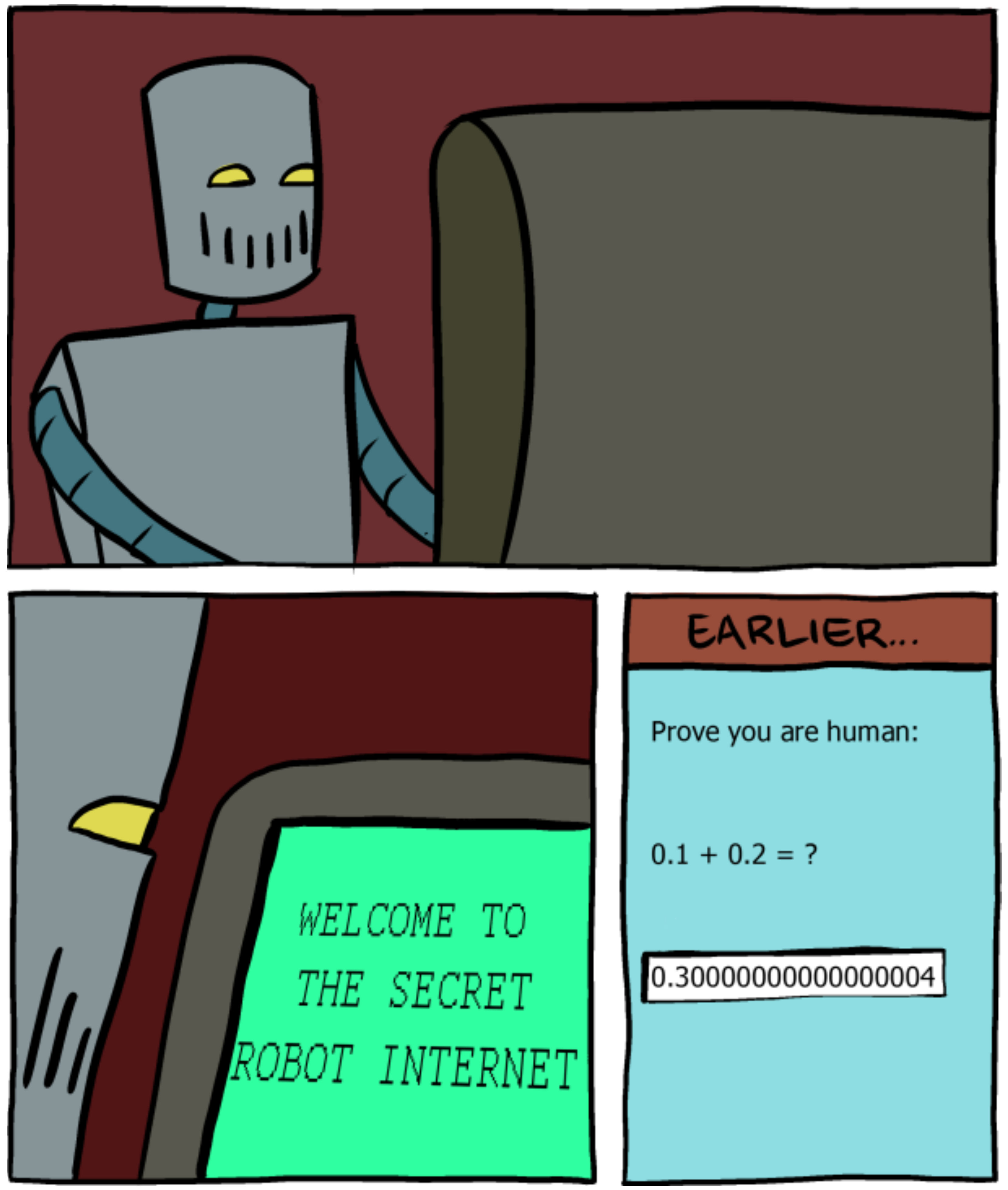
Monday, Mar 9, 2025

Computer Systems from the Ground Up
Winter 2025
Stanford University
Computer Science Department

$$V = (-1)^s \times M \times 2^E$$

Single precision (`float`)





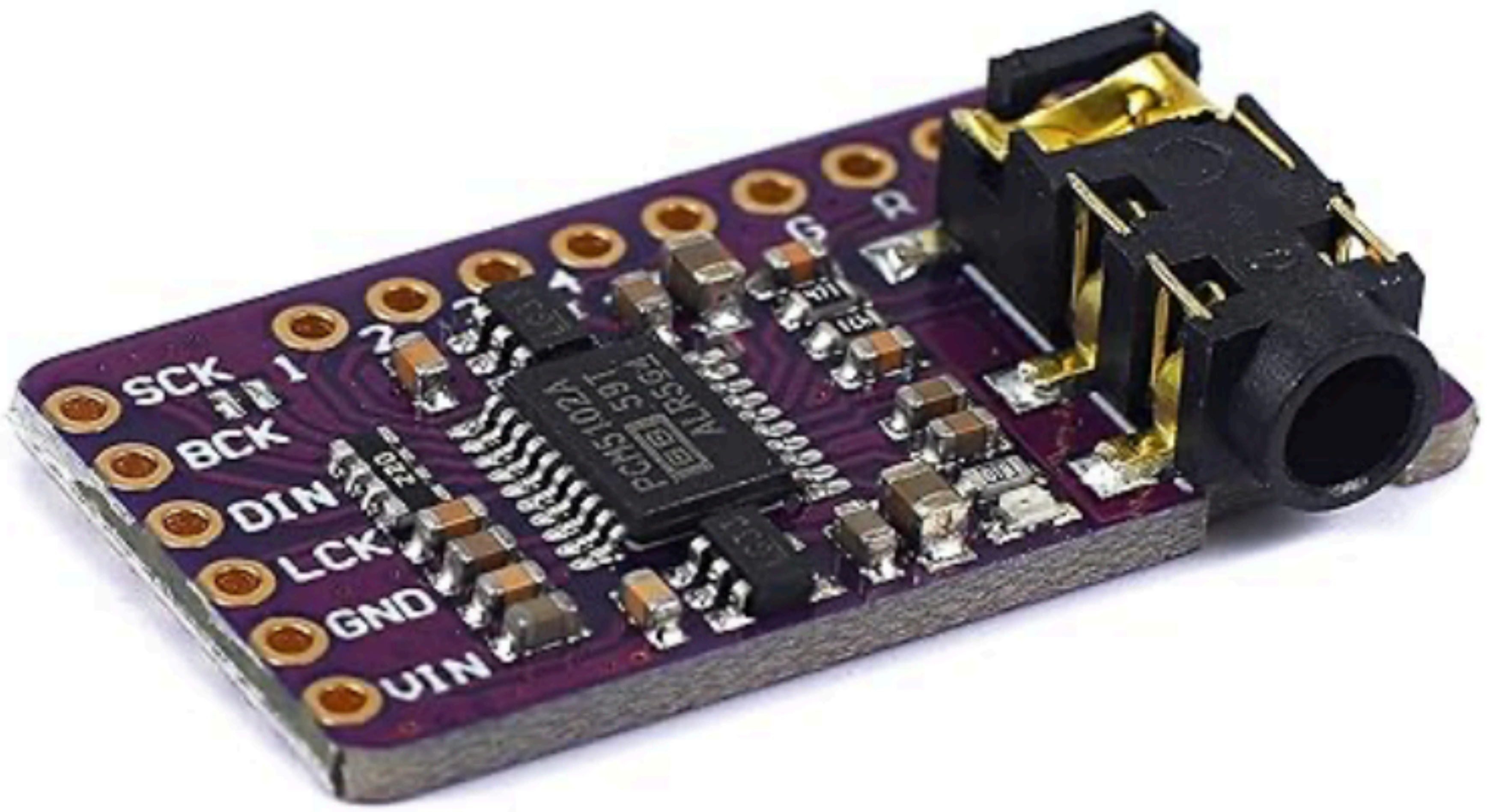
Today's Topics

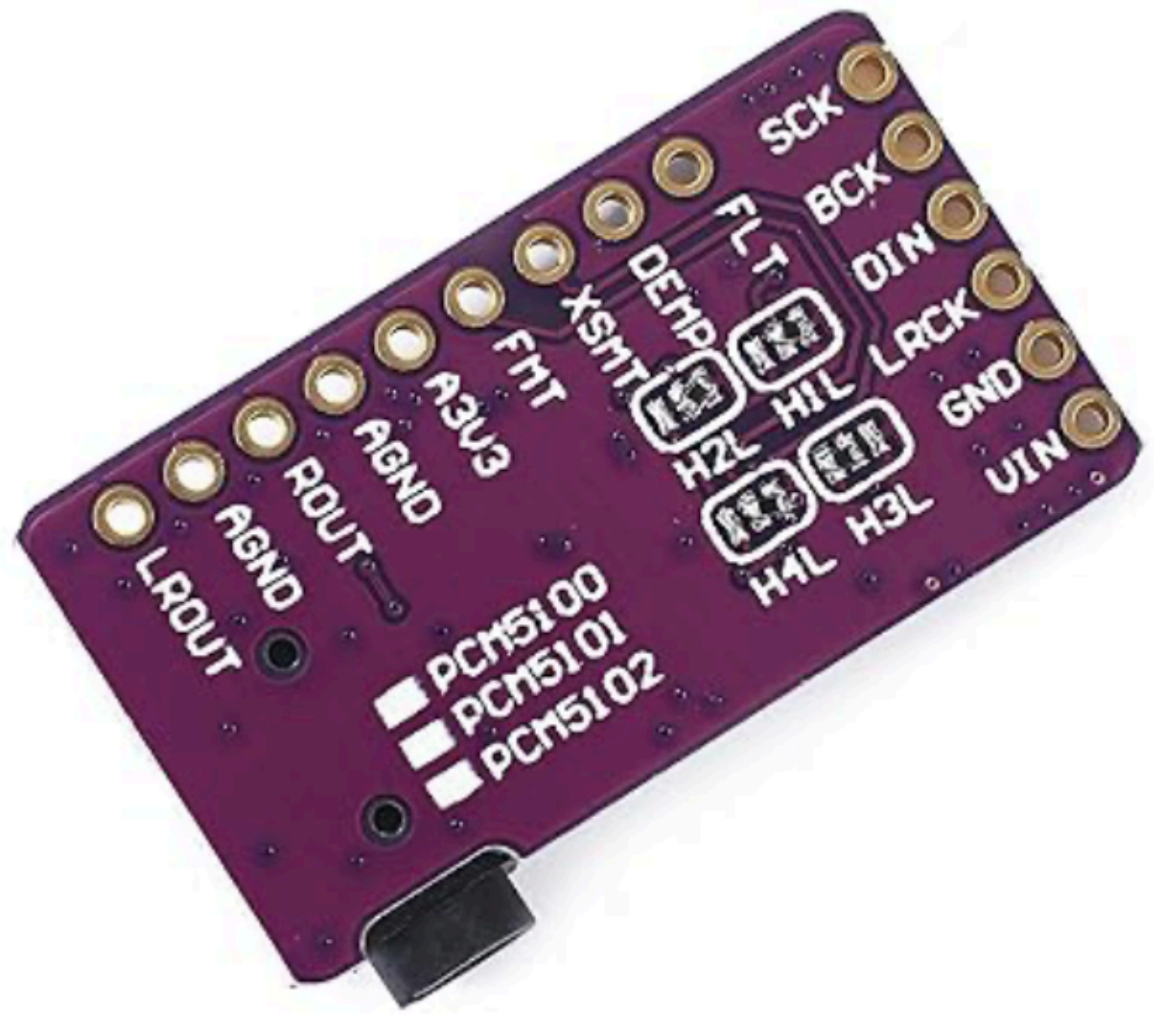
- Logistics
  - Next week: Final projects!
  - Project presentations: Friday, March 21st, 9am-12pm
- Today: (Fake) Sound / Floating Point Numbers
  - Demos: I2S Microphone / MIDI
  - Real Numbers
  - Fixed Point
  - IEEE 754 Floating Point
    - Normalized values
    - Denormalized values
    - Exceptional values
    - Arithmetic

I2S Microphone Demo

- Last week, Julie demonstrated I2S output (playing a .wav file converted into a C array). You can do this in two different ways:
  - Blocking: the sound plays, but your program is only playing sound. This is useful for short sounds that may indicate completion of a task, or an alert.
  - Non-blocking: Using DMA, the sounds plays while your program continues. This is useful for background sound (e.g., for a game).
- In addition to output I2S devices, you can also get input I2S microphones. These are inexpensive, and allow you to record sound.

The output device:

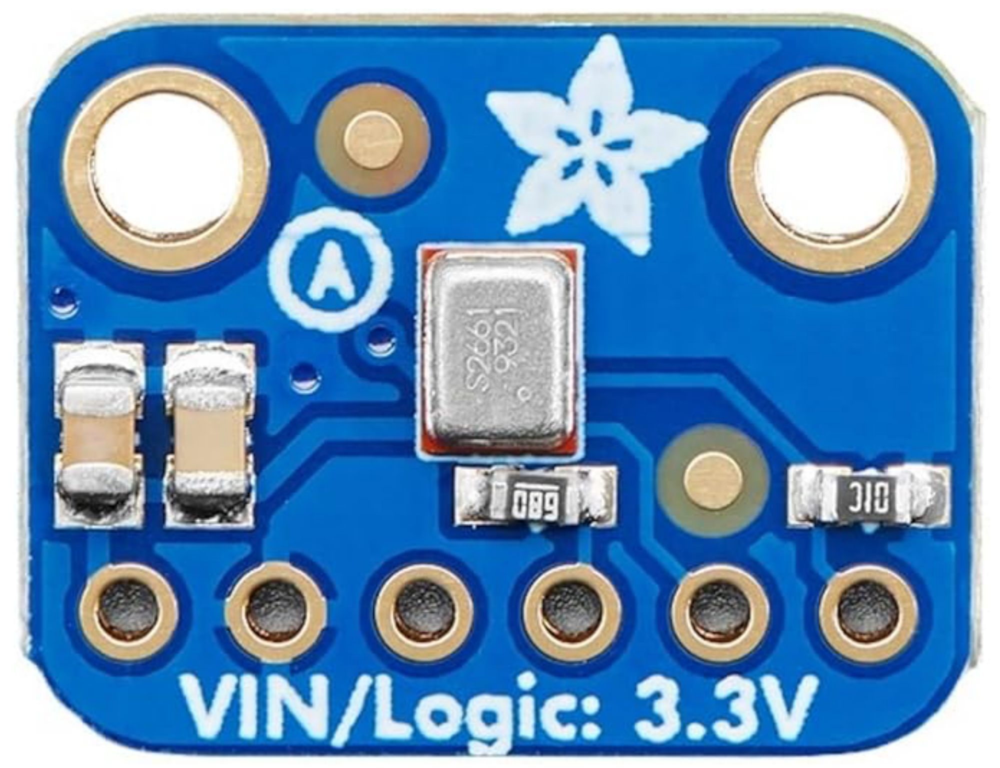


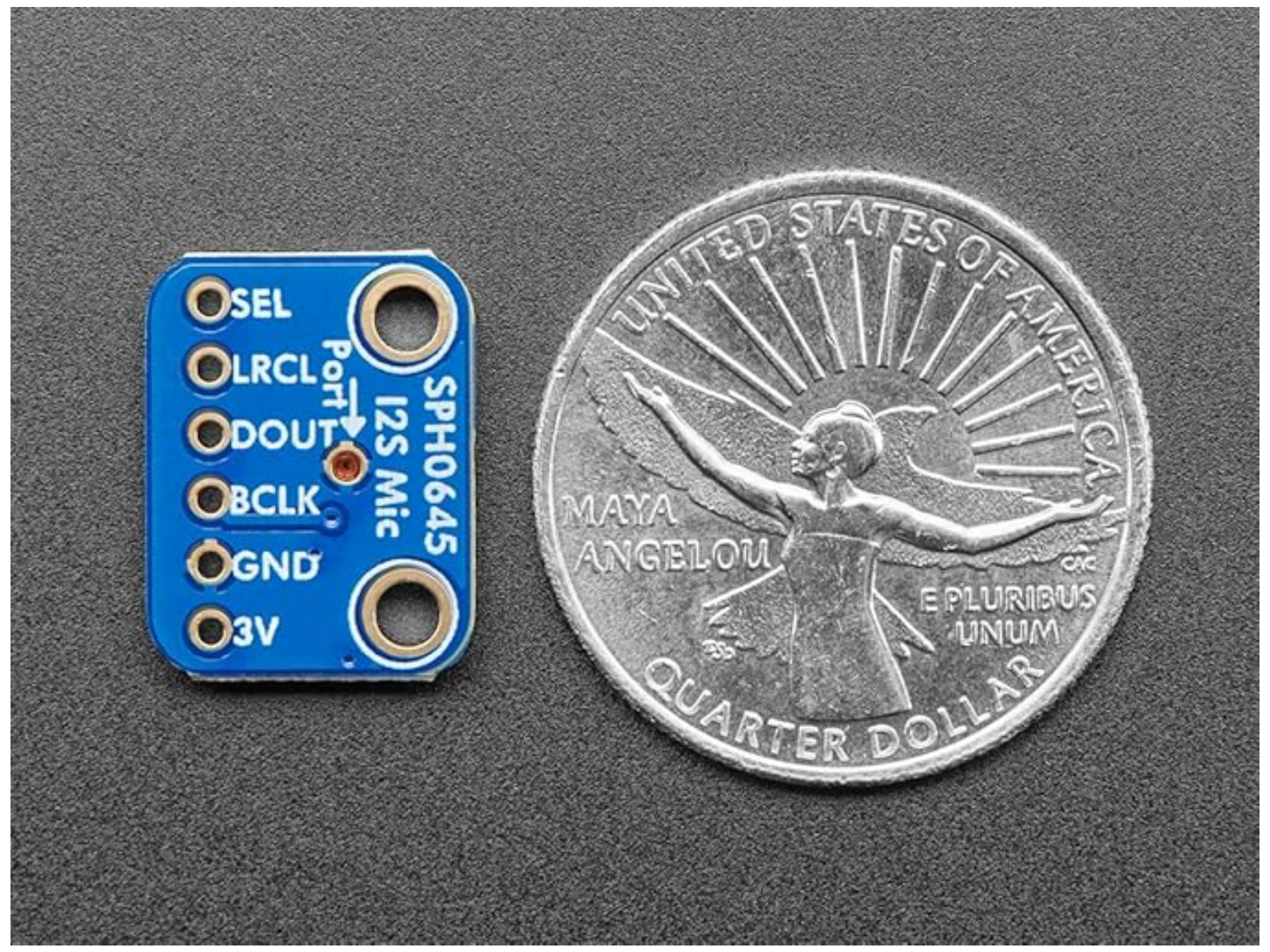


Teyleten Robot PCM5102 PCM5102A AUX Stereo Digital Audio DAC Decoder Board Module Voice Module Player Module Digital-to-Analog Converter IIS I2S

$6 on amazon.com (or by two for $9)

The microphone:





Adafruit I2S MEMS Microphone Breakout - SPH0645LM4H (<u>$12 on amazon.com</u>)

INMP441 Omnidirectional Microphone Module MEMS I2S Interface (<u>five for $13</u> on amazon.com)

MIDI

MIDI: Musical Instrument Digital Interface

- Simple interface to control musical instruments
- Emerged from electronic music and instruments in 1970s
- First version described in Keyboard magazine in 1982

MIDI

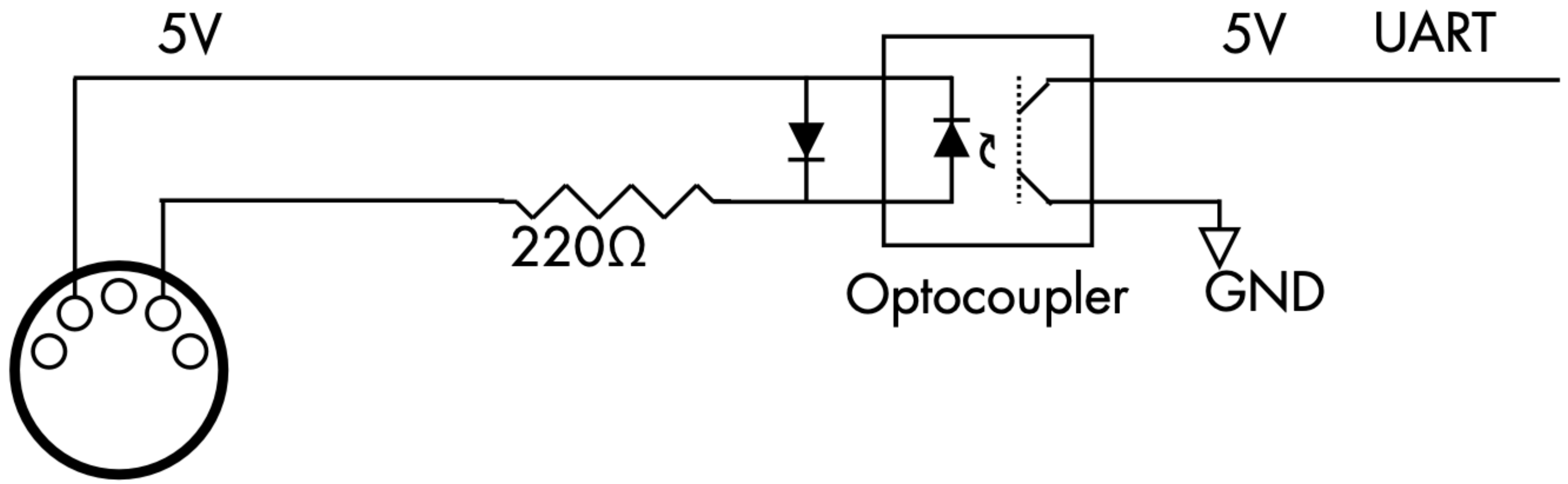
MIDI: Musical Instrument Digital Interface

- 31.25 kbps 8-N-1 serial protocol
- Commands are 1 byte, with variable parameters
- (c=channel, k=key, v=velocity, l=low bits, m=high bits)

Command	Code	Param	Param
Note on	`1001cccc`	`0kkkkkkk`	`0vvvvvvv`
Note off	`1000cccc`	`0kkkkkkk`	`0vvvvvvv`
Pitch bender	`1110cccc`	`0lllllll`	`0mmmmmmm`

MIDI Circuit

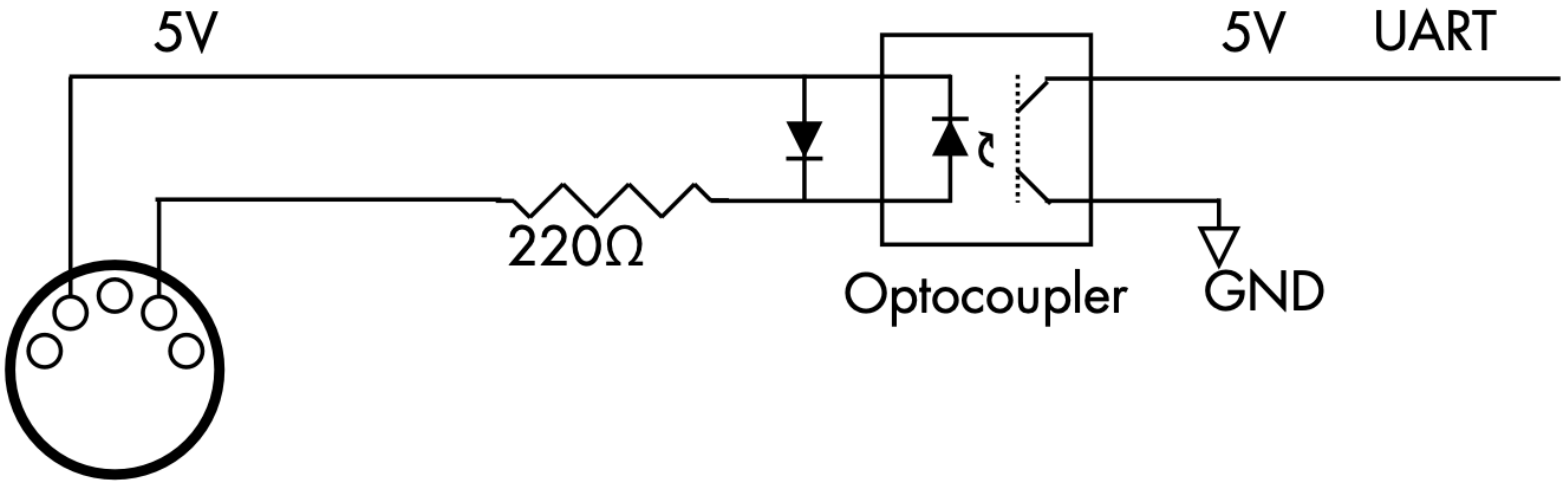
0 is high, 1 is low!



Optocoupler completely isolates circuits electrically: no noise in instrument

MIDI Hack

If we don't have an optocoupler, we can do okay with an additional 220Ω resistor:



Demos

Integers

```
0
```

Integers

1

Integers

2

Integers

3

(can we go forever?)

Integers

3

(can we go forever?)

Nope! (not in C, at least)

Integers

$$-1$$

(what about negative numbers?)

Integer Representations

The C language has two different ways to represent numbers, unsigned and signed:

unsigned: can only represent non-negative numbers

signed: can represent negative, zero, and positive numbers

Integers

In C, we have the following integer types, with their bit size for the Mango Pi (C does not explicitly define anything but `char`, which is always 1 byte):

`type`	`bits`	`bytes`	`min`	`max`
`char`	`8`	`1`	`-128`	`127`
`unsigned char`	`8`	`1`	`0`	`255`
`short`	`16`	`2`	`-32768`	`32767`
`unsigned short`	`16`	`2`	`0`	`65535`
`int`	`32`	`4`	`-2147483648`	`2147483647`
`unsigned int`	`32`	`4`	`0`	`4294967295`
`long`	`64`	`8`	`-9223372036854775808`	`9223372036854775807`
`unsigned long`	`64`	`8`	`0`	`18446744073709551615`
`long long`	`64`	`8`	`-9223372036854775808`	`9223372036854775807`
`unsigned long long`	`64`	`8`	`0`	`18446744073709551615`

Unsigned Integers

For positive (unsigned) integers, there is a 1-to-1 relationship between the decimal representation of a number and its binary representation. If you have a 4-bit number, there are 16 possible combinations, and the unsigned numbers go from 0 to 15:

```
0b0000 = 0     0b0001 = 1     0b0010 = 2     0b0011 = 3
0b0100 = 4     0b0101 = 5     0b0110 = 6     0b0111 = 7
0b1000 = 8     0b1001 = 9     0b1010 = 10    0b1011 = 11
0b1100 = 12    0b1101 = 13    0b1110 = 14    0b1111 = 15
```

The range of an unsigned number is $0 \rightarrow 2^w - 1$, where w is the number of bits in our integer. For example, a 32-bit `int` can represent numbers from 0 to $2^{32} - 1$, or 0 to 4,294,967,295.

Signed Integers: How do we represent them?

What if we want to represent negative numbers? We have choices!

One way we could encode a negative number is simply to designate some bit as a "sign" bit, and then interpret the rest of the number as a regular binary number and then apply the sign. For instance, for a four-bit number:

0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

This might be okay...but we've only represented 14 of our 16 available numbers...

Signed Integers: How do we represent them?

0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

What about 0 000 and 1 000? What should they represent?

Well...this is a bit tricky!

Signed Integers: How do we represent them?

0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

What about 0 000 and 1 000? What should they represent?

Well...this is a bit tricky!

Let's look at the bit patterns: 0 000 1 000

Should we make the 0 000 just represent decimal 0? What about 1 000? We could make it 0 as well, or maybe -8, or maybe even 8, but none of the choices are nice.

Signed Integers: How do we represent them?

0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

What about 0 000 and 1 000? What should they represent?

Well...this is a bit tricky!

Let's look at the bit patterns: 0 000 1 000

Should we make the 0 000 just represent decimal 0? What about 1 000? We could make it 0 as well, or maybe -8, or maybe even 8, but none of the choices are nice.

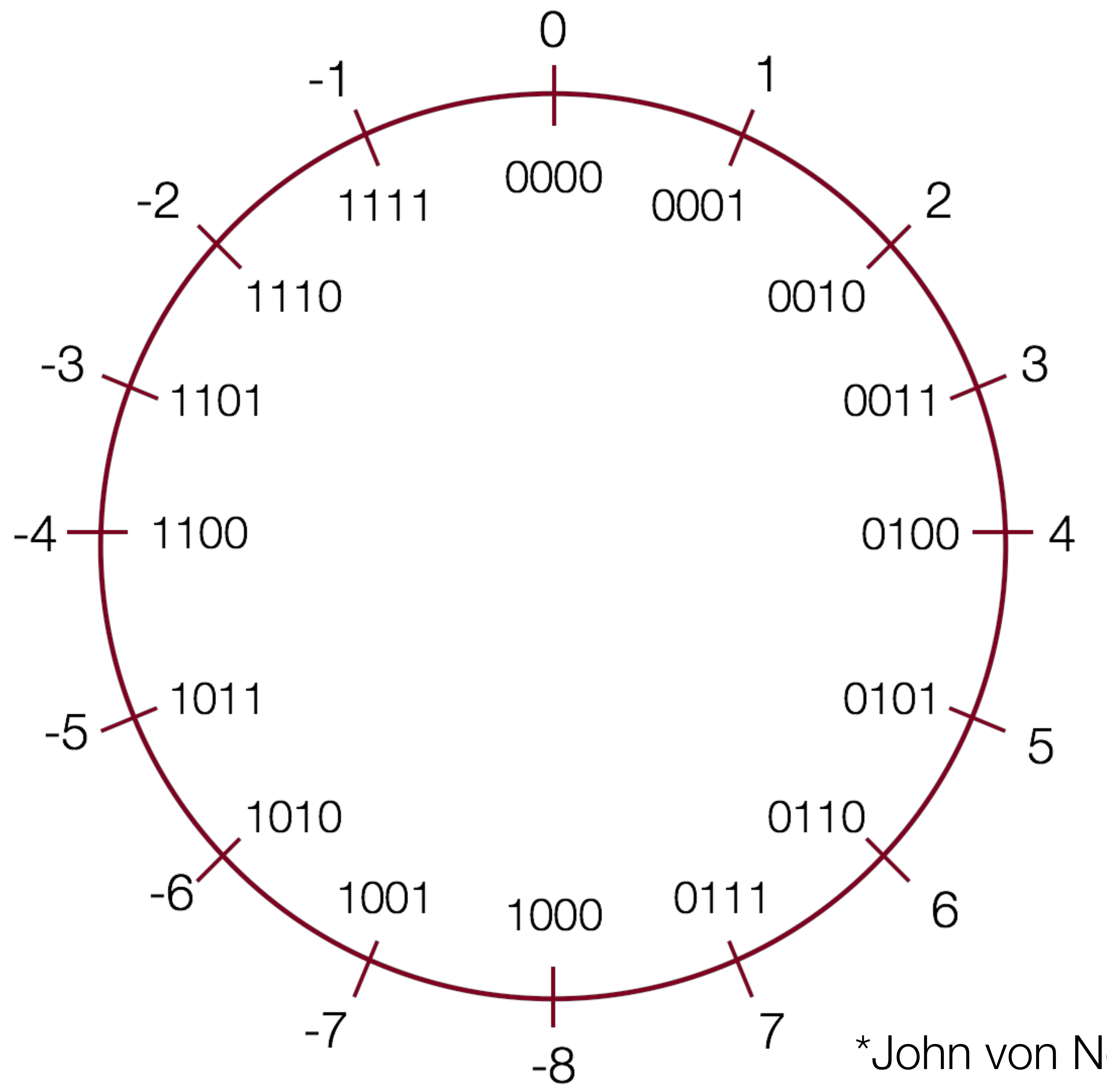
Fine. Let's just make 0 000 to be equal to decimal 0. How does arithmetic work? Well…to add two numbers, you need to know the sign, then you might have to subtract (borrow and carry, etc.), and the sign might change…this is going to get ugly!

Signed Integers: How do we represent them?

There is a better way!

Signed Integers: How do we represent them?

Behold: the "two's complement" circle:



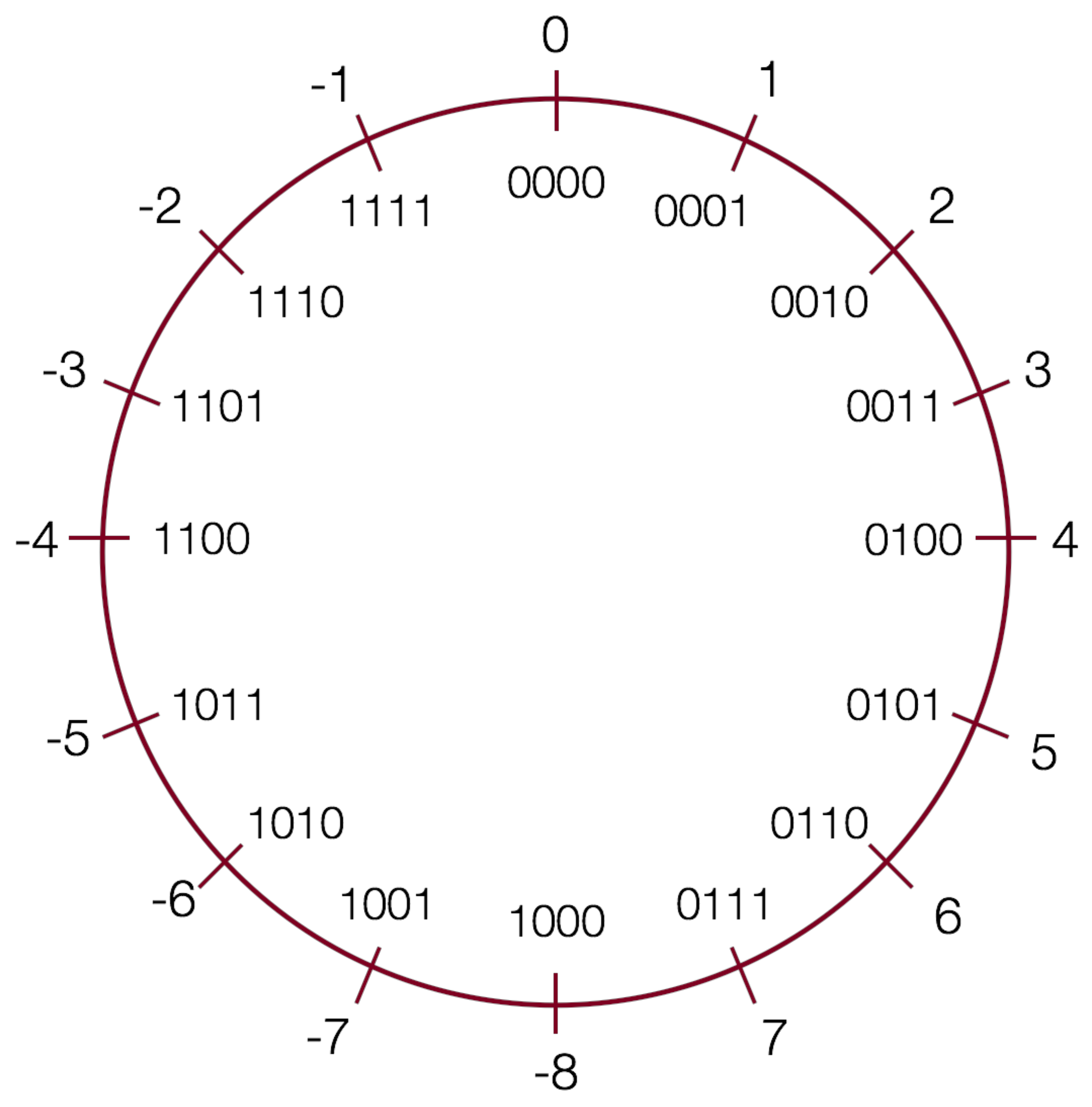
In the early days of computing*, two's complement was determined to be an excellent way to store binary numbers.

In two's complement notation, positive numbers are represented as themselves (phew), and negative numbers are represented as the two's complement of themselves (definition to follow).

This leads to some amazing arithmetic properties!

*John von Neumann suggested it in 1945, for the EDVAC computer.

Two's Complement



A two's-complement number system encodes positive and negative numbers in a binary number representation. The weight of each bit is a power of two, except for the most significant bit, whose weight is the negative of the corresponding power of two.

Definition: For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ of an w-bit integer $x_{w-1}x_{w-2}\ldots x_0$ is given by the following formula:

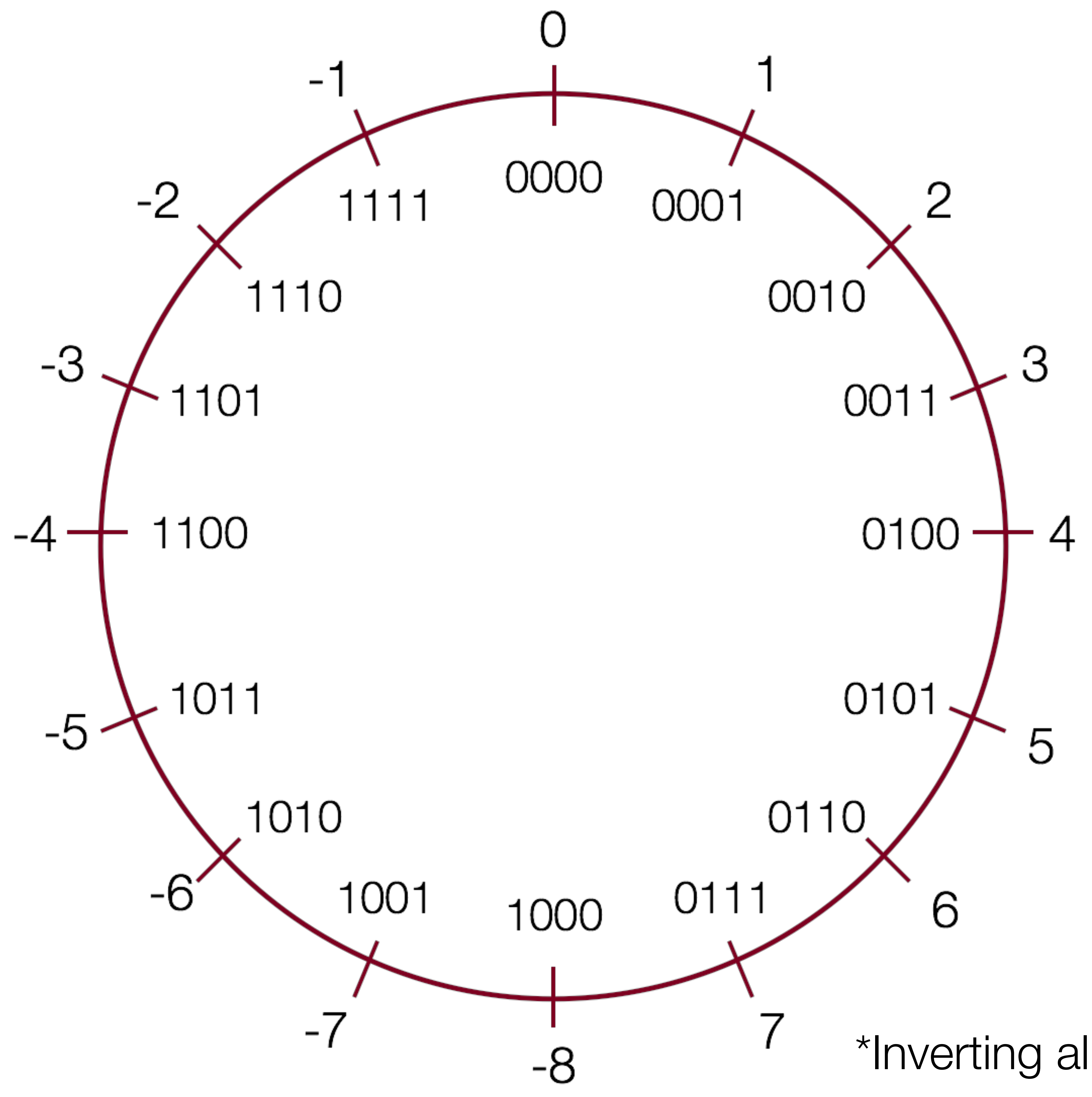
$$B2T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i.$$

$B2T_w$ means "Binary to Two's complement function"

In practice, a negative number in two's complement is obtained by inverting all the bits of its positive counterpart*, and then adding 1.

*Inverting all the bits of a number is its "one's complement"

Two's Complement



In practice, a negative number in two's complement is obtained by inverting all the bits of its positive counterpart*, and then adding 1, or: `x = ~x + 1`

Example: The number 2 is represented as normal in binary: 0010

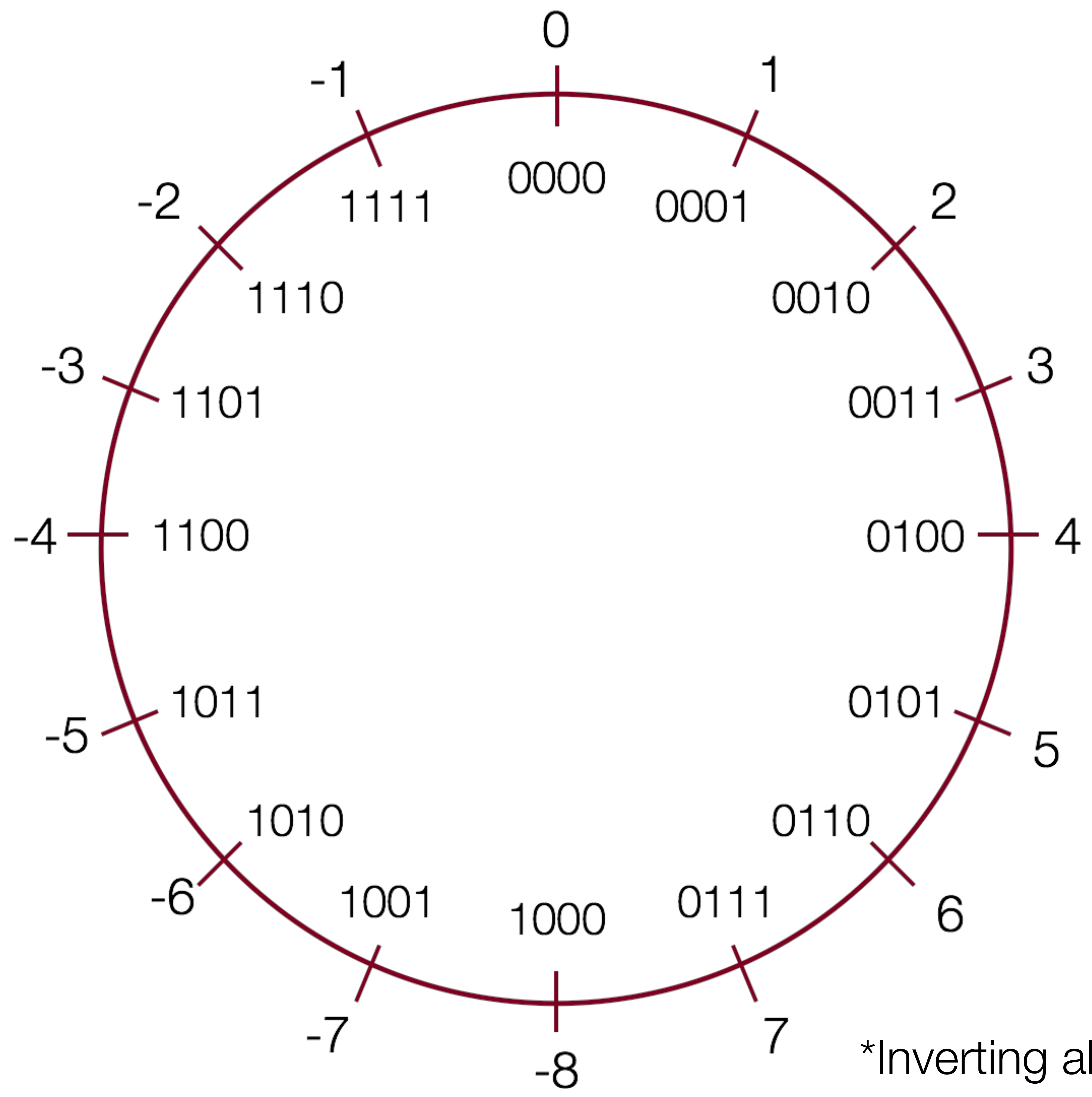
-2 is represented by inverting the bits, and adding 1:

0010 ☞ 1101

$$\begin{array}{r} 1101 \\ +\quad 1 \\ \hline 1110 \end{array}$$

*Inverting all the bits of a number is its "one's complement"

Two's Complement



Trick: to convert a positive number to its negative in two's complement, start from the right of the number, and write down all the digits until you get to a 1. Then invert the rest of the digits:

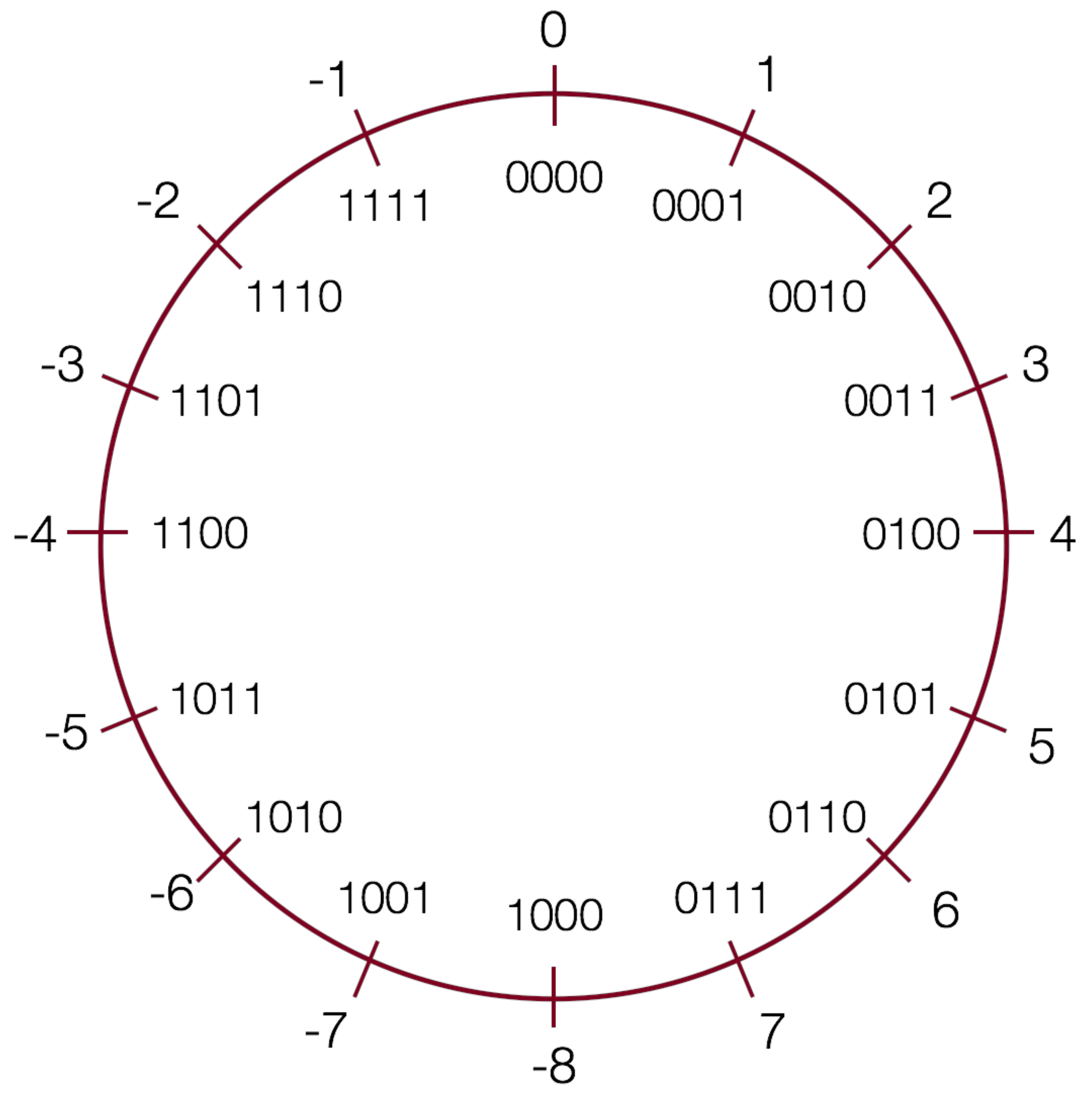
Example: The number 2 is represented as normal in binary: 0010

Going from the right, write down numbers until you get to a 1:
    10

Then invert the rest of the digits:
1110

*Inverting all the bits of a number is its "one's complement"

Two's Complement



To convert a negative number to a positive number, perform the same steps!

Example: The number -5 is represented in two's complements as: 1011

5 is represented by inverting the bits, and adding 1:

1011 ☞ 0100

```
 0100
+   1
 0101
```

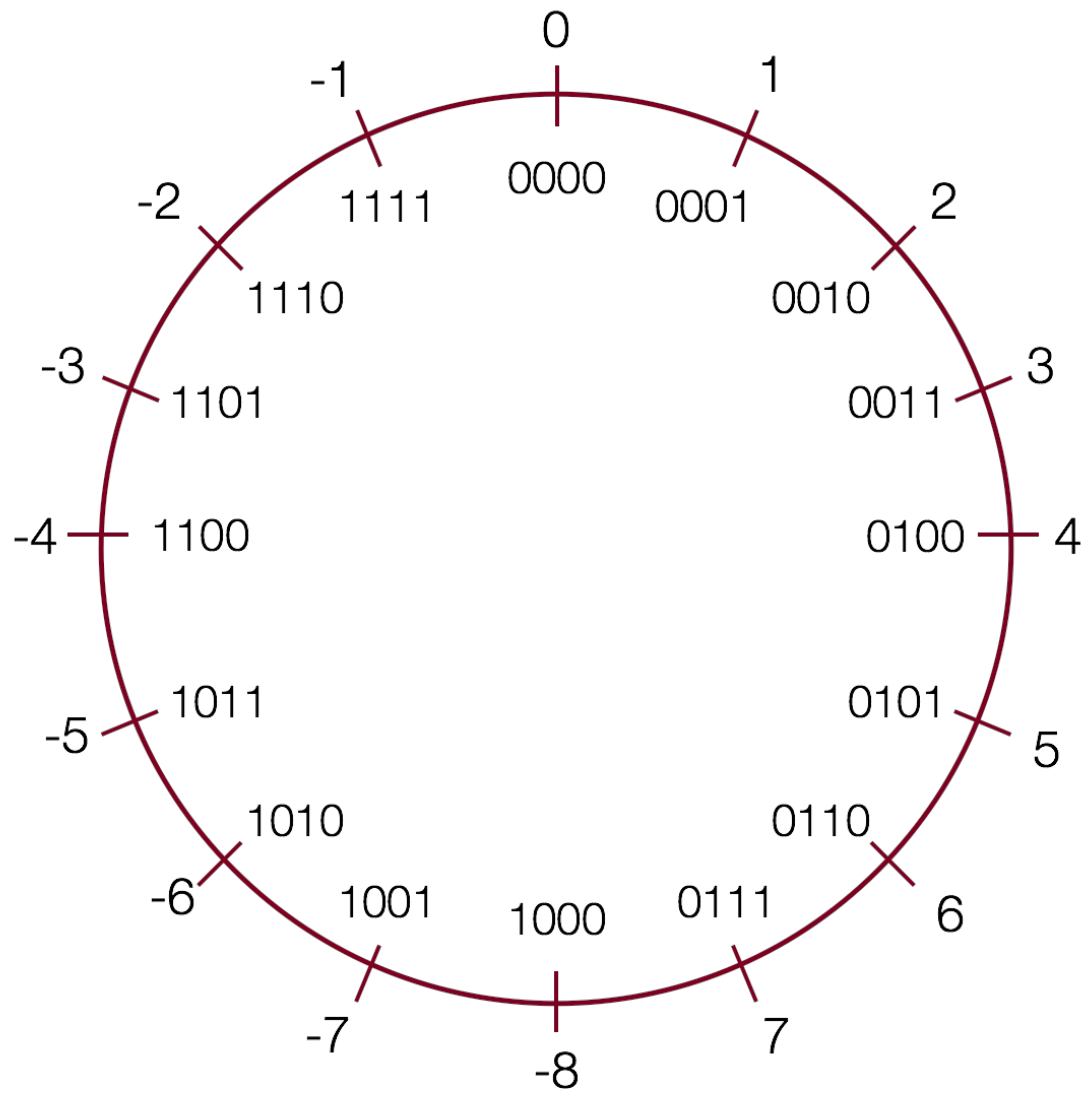
Shortcut: start from the right, and write down numbers until you get to a 1:

```
   1
```

Now invert all the rest of the digits:

0101

Two's Complement: Neat Properties

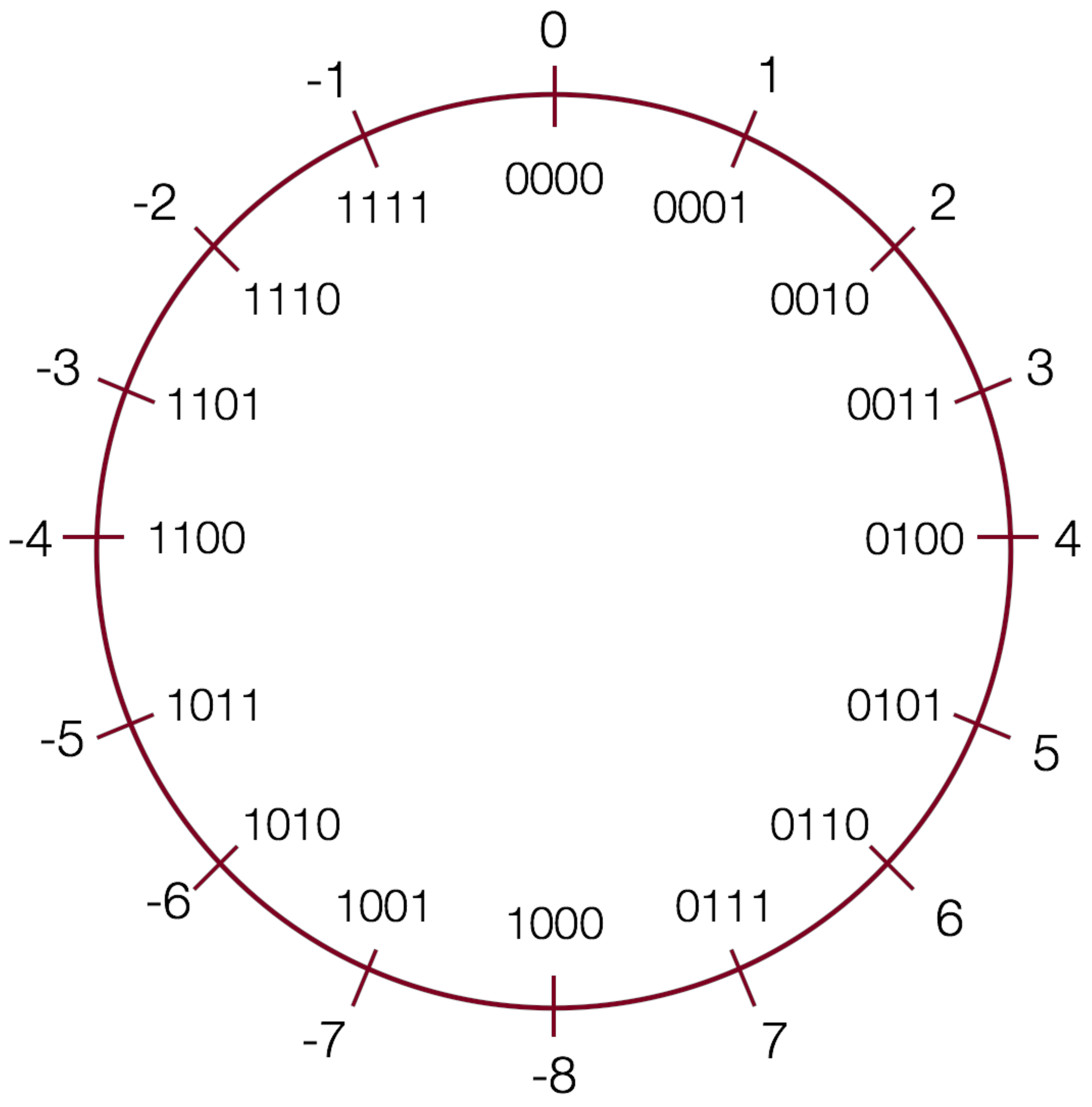


There are a number of useful properties associated with two's complement numbers:

1. There is only one zero (yay!)
2. The highest order bit (left-most) is 1 for negative, 0 for positive (so it is easy to tell if a number is negative)
3. Adding two numbers is just…adding! Example:
   2 + -5 = -3

$$\begin{array}{rl} 0010 & ☞\ 2 \\ \underline{+1011} & ☞\ -5 \\ 1101 & ☞\ -3 \text{ decimal (wow!)} \end{array}$$

Two's Complement: Neat Properties



More useful properties:

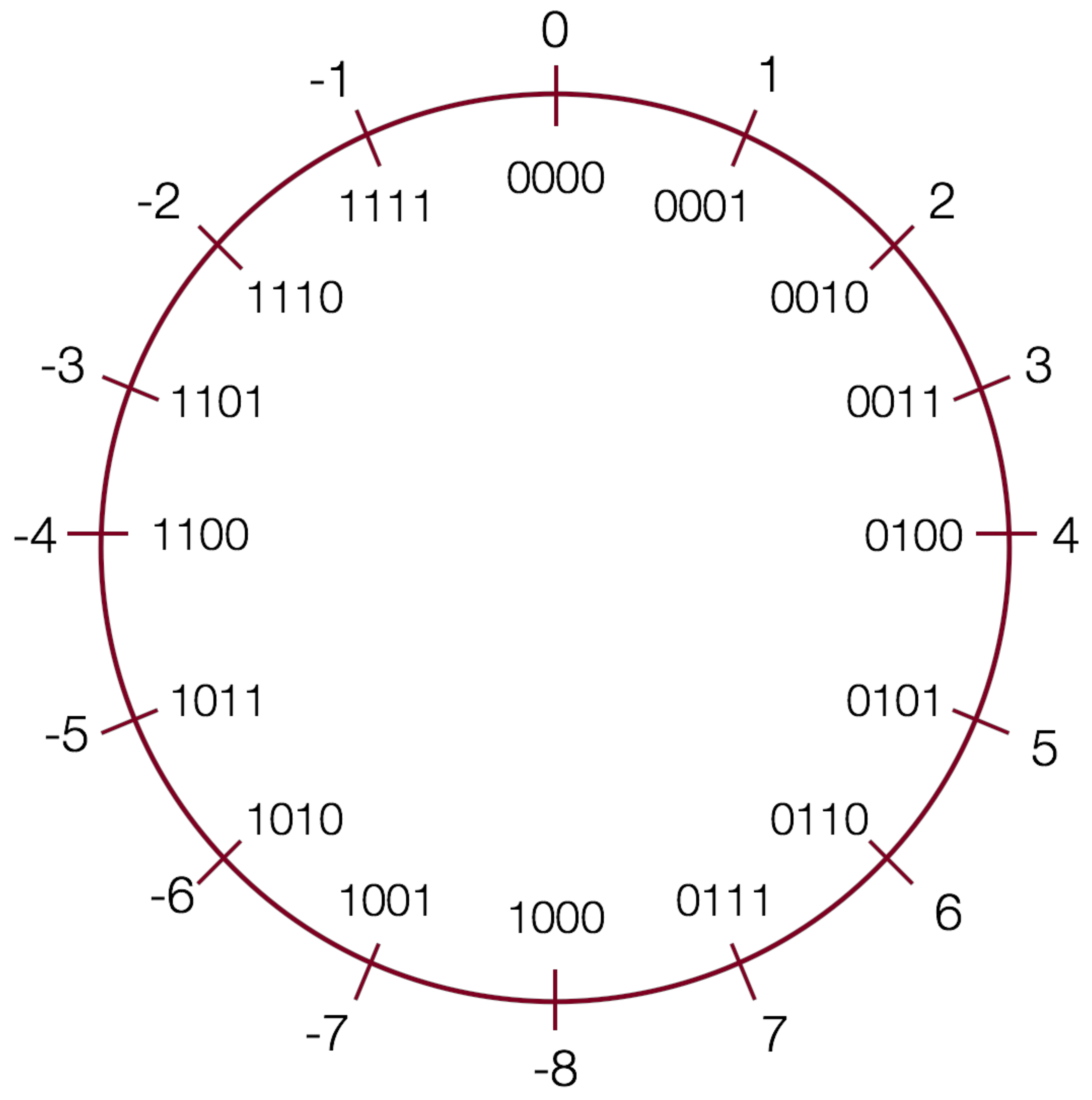
4. Subtracting two numbers is simply performing the two's complement on one of them and then adding.
   Example:
   4 - 5 = -1

0100 ☞ 4, 0101 ☞ 5

Find the two's complement of 5: 1011
add:

```
  0100 ☞ 4
+1011 ☞ -5
  1111 ☞ -1 decimal
```

Two's Complement: Neat Properties



More useful properties:

5. Multiplication of two's complement works just by multiplying (throw away overflow digits).

Example: -2 * -3 = 6

1110 ☞ -2

x1101 ☞ -3

1110

0000

1110

+1110

~~1011~~**0110** ☞ 6

Real Numbers

$$^{1}/_{5}$$

Real Numbers

$$\frac{1}{5} = 0.2$$

Real Numbers

$$\frac{1}{3}$$

Real Numbers

$$\frac{1}{3} = 0.33333\ldots$$

Real Numbers

$$\frac{1}{3} = 0.33333\ldots$$

When I was in 6th grade, this was a mind blowing concept.

Fractions

$$\frac{1}{3} = 0.33333\ldots$$

Especially this

Real Numbers

$$\pi = 3.14159\ldots$$

And don't get me started on this

Real Numbers

Once we leave the realm of integers, real numbers become … tricky.

- Some rational numbers, e.g., ⅕, can be represented exactly, by a fixed number of decimal digits (0.2 in this case)
- Some rational numbers, e.g., ⅓, can not be represented exactly, and we have this idea of "repeating indefinitely," which we represent with "…" (0.33333…)
- Irrational numbers can never be represented by a fixed number of decimal digits, and the meaning of "…" means "indefinitely" but loses the "repeating" part. Irrational numbers can never be represented exactly using a digit notation.

The big question: how do we represent real numbers in a computer?

Real Numbers in a Computer

The big question: how do we represent real numbers in a computer?

As always, we have choices. Here are some constraints:

1. We want to represent real numbers in a fixed number of bits. This means that we aren't going to be able to represent all real numbers exactly, nor even all rational numbers exactly. Furthermore, we can't even represent all rational numbers in a *range* exactly (there are infinitely many rational numbers in any fixed range).
2. We want to represent a large range of numbers.
3. We want to be able to perform calculations on the numbers.

Real Numbers in a Computer

One idea: Fixed Point

When we represented integers, we implicitly placed the decimal point (or binary point, in base 2) after the least significant digit, and we limited ourselves to positive powers of our base. E.g., 1234 is really 1234.0000…

We could just move the decimal place, and use that as our system for representing real numbers:

In decimal: $d_2 d_1 d_0 . d_{-1} d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$
e.g., 123.45

Fixed Point

$$d_2d_1d_0.d_{-1}d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$$

e.g., 123.45

What range of numbers can we represent now?

Fixed Point

$$d_2d_1d_0.d_{-1}d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$$

e.g., 123.45

What range of numbers can we represent now? 0 to 999.99

Fixed Point

$$d_2 d_1 d_0 . d_{-1} d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$$

e.g., 123.45

What range of numbers can we represent now? 0 to 999.99

What is the "precision" we can represent (i.e., how precise?)

Fixed Point

$$d_2 d_1 d_0 . d_{-1} d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$$

e.g., 123.45

What range of numbers can we represent now? 0 to 999.99

What is the "precision" we can represent (i.e., how precise?)

we can represent five decimal digits of precision, to the 100th place

Fixed Point

$$d_2 d_1 d_0 . d_{-1} d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$$

e.g., 123.45

We can't represent some rational numbers exactly:

123.456
123.333…
1000 (overflow? Also, 999.991, or 999.9901, or 999.99001, or …)
0.001 (underflow?)
We would have to round or truncate, or over/under-flow.

Fixed Point

$$d_2 d_1 d_0 . d_{-1} d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$$

e.g., 123.45

Fixed-point arithmetic is pretty easy:

```
 123.45
+678.90
-------
 802.35
```

Fixed Point

$$d_2d_1d_0.d_{-1}d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$$

e.g., 123.45

Fixed-point arithmetic is pretty easy:

```
 123.45
+678.90
-------
 802.35
```

```
  100.22
*   1.08
--------
   80176
  000000
 1002200
--------
 1082376 = 108.2376
         = 108.24
         (rounded)
```

Fixed Point

$$d_2d_1d_0.d_{-1}d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$$

Fixed point has its uses, but it is somewhat limiting. We can do regular arithmetic, and we know how many decimal places of precision we get.

However, the range is set by where we fix the decimal place, and we had hoped for a large range. If we set the decimal place to be to the left of the most significant digit for a five-digit number, our range would only be 0 to 0.99999.

Floating Point

A different idea is to represent numbers in the form $V = x \times 2^y$

In this form, we will break our number into two parts (actually, three, including a sign bit), with an exponent (y) and a fractional value (x).

Before we get into the details, let's investigate what fractional values in binary look like. Recall, in decimal:

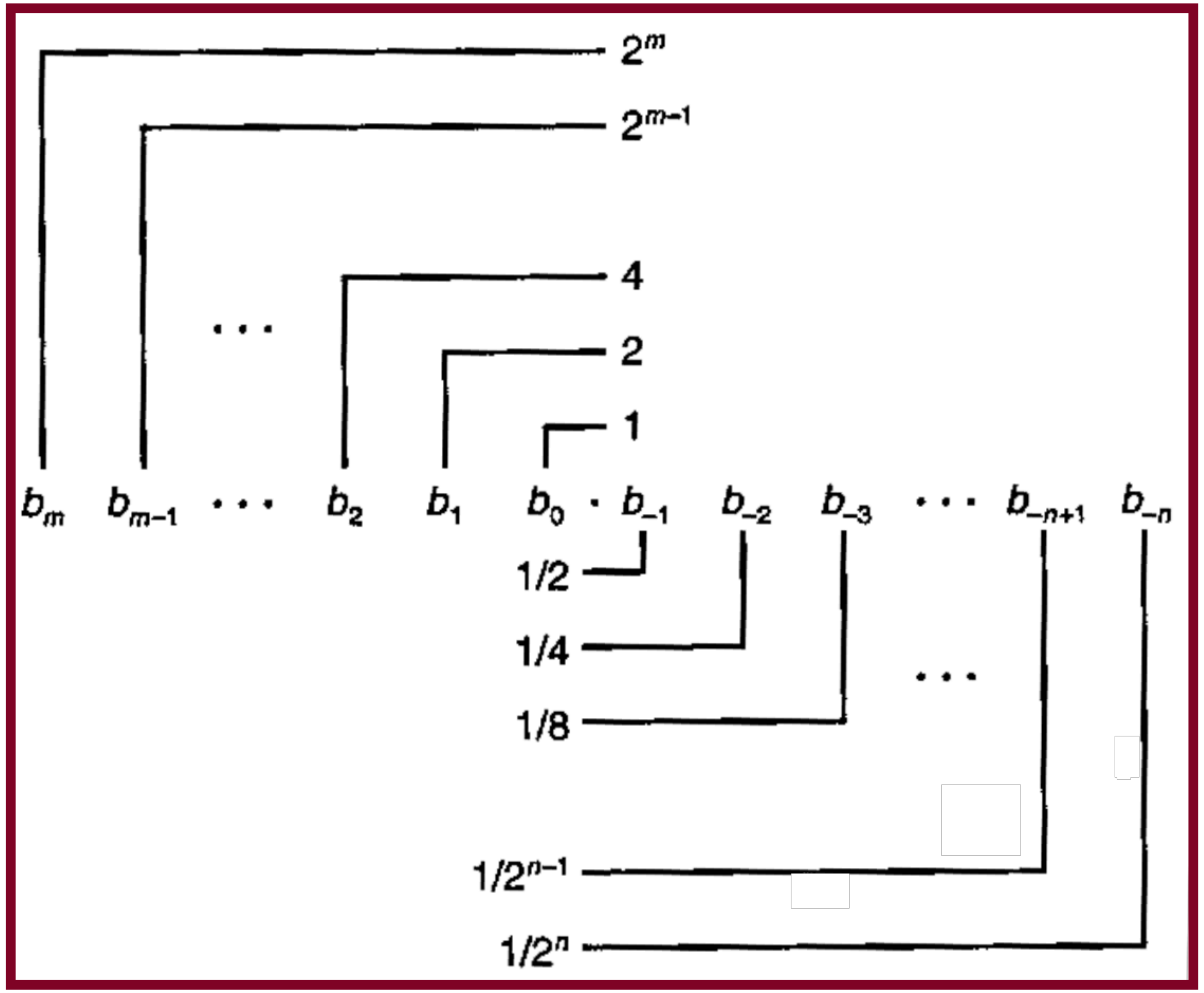
$$d_2 d_1 d_0 . d_{-1} d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$$

Digits after the decimal point are represented by negative powers of 10.

Fractional Values in Binary

In binary, digits after the *binary* point are represented by negative powers of *two*:

$$b_2b_1b_0.b_{-1}b_{-2} = b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2}$$



2^m
2^{m-1}
4
2
1
b_m b_{m-1} ··· b_2 b_1 b_0 . b_{-1} b_{-2} b_{-3} ··· b_{-n+1} b_{-n}
1/2
1/4
1/8
$1/2^{n-1}$
$1/2^n$

Online binary to decimal converter:

http://web.stanford.edu/class/cs107e/float/

Fractional Values in Binary

Example: 101.11 in binary:

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$$

What happens to your number if you shift the binary point to the left by one?

Fractional Values in Binary

Example: 101.11 in binary:

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \tfrac{1}{2} + \tfrac{1}{4} = 5\tfrac{3}{4}$$

What happens to your number if you shift the binary point to the left by one?

The number is divided by two.

Fractional Values in Binary

Example: 101.11 in binary:

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \tfrac{1}{2} + \tfrac{1}{4} = 5\tfrac{3}{4}$$

What happens to your number if you shift the binary point to the left by one?

The number is divided by two.

What happens to your number if you shift the binary point to the right by one?

Fractional Values in Binary

Example: 101.11 in binary:

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \tfrac{1}{2} + \tfrac{1}{4} = 5\tfrac{3}{4}$$

What happens to your number if you shift the binary point to the left by one?

The number is divided by two.

What happens to your number if you shift the binary point to the right by one?

The number is multiplied by two.

Fractional Values in Binary

Example: 101.11 in binary:

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$$

What happens to your number if you shift the binary point to the left by one?

The number is divided by two.

What happens to your number if you shift the binary point to the right by one?

The number is multiplied by two.

What is represented by 0.111111…1?

Fractional Values in Binary

Just like decimal with numbers like ⅓ and ⅙, binary cannot represent exactly any numbers like ⅓ and ⅕, nor even ⅒:

```c
#include "printf.h"
#include "ftoa.h"

void main() {
    char buf[256];
    float f = 0.1;
    // convert to a string,
    // with 20 decimal
    // places of precision
    ftoa(f,buf,20);
    printf("f: %s\n",buf);
}
```

```
$ ./testTenth
0.10000000149011611938
```

Fractional binary notation can only exactly represent numbers that can be written in the form:

$$x \times 2^y$$

IEEE Floating Point

When designing a number format, choices need to be made about the format specification. In the late 1970s, Intel sponsored William Kahan (from Berkeley…) to design a floating point standard, which formed the basis for the "IEEE Standard 754," or *IEEE Floating Point*, which almost all computers use today. The standard defines the bit pattern (32-bit, 64-bit, etc.) as a number in the form:

$$V = (-1)^s \times M \times 2^E$$

Where:

- The *sign s* is negative (s == 1) or positive (s == 0), with the sign for numerical value 0 as a special case.
- The *significand M* (sometimes called the *Mantissa*), is a fractional binary number that ranges *either* between 1 and 2-ϵ or between 0 and 1-ϵ.
- The *exponent E* weights the value by a (possibly negative) power of 2.

IEEE Floating Point Examples

$$V = (-1)^s \times M \times 2^E$$

Example: For s=0, M=1.5, E=9: $V = (-1)^0 \times 1.5 \times 2^9 = 768$

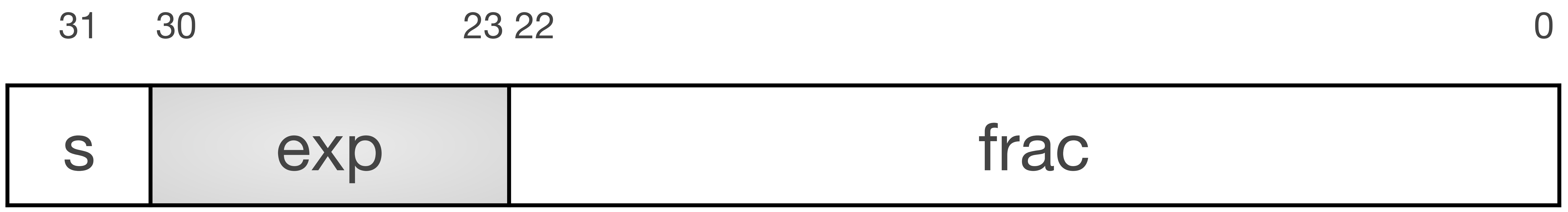
IEEE Floating Point

$$V = (-1)^s \times M \times 2^E$$

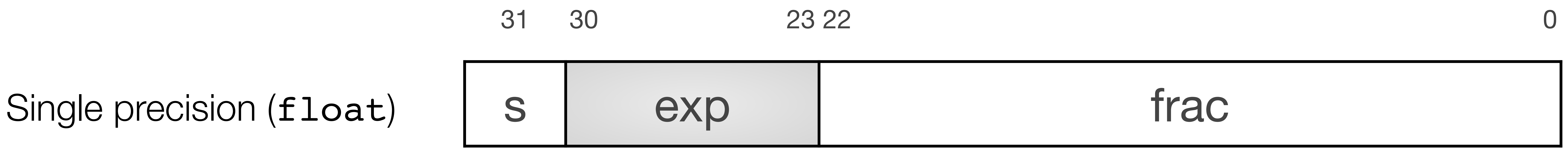
The bit representation of a floating point number is divided into three fields to encode these values:

- The single sign bit *s* directly encodes the sign *s*.
- The *k*-bit exponent field, $\mathrm{exp} = e_{k-1} \ldots e_1 e_0$ encodes the exponent *E*.
- The *n*-bit fraction field $\mathrm{frac} = f_{n-1} \ldots f_1 f_0$ encodes the significand *M*, but the value encoded also depends on whether or not the exponent field equals 0.

Single precision (`float`)

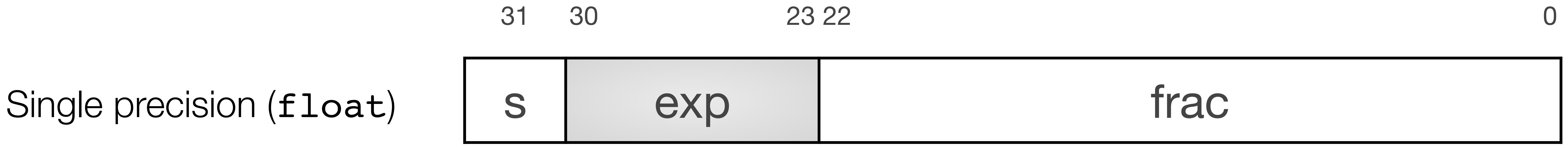


Before We Continue



Right now, you're saying to yourself, "Uhh…this is going to be complicated."

Before We Continue

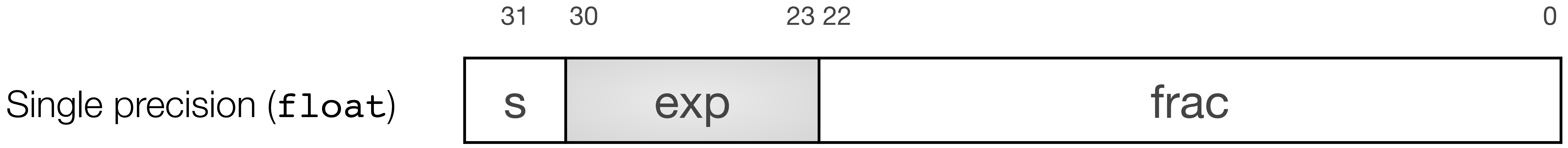


Right now, you're saying to yourself, "Uhh…this is going to be complicated."

Yes, it does take some time to learn. We want you to appreciate a few things about the IEEE floating point format:

1. It is based on decisions and choices that were made, with good reason (we will discuss those reasons).
2. It is efficient, and attempts to eek out as much as it can from those 32 or 64 bits — computing is often about efficiency, and the people who came up with the standard really thought hard about it.
3. We don't want you to think "I could never come up with that!" — rather, we want you to appreciate it for what it is.

Normalized Floats



- A float is considered to have a "normalized" value if the exponent is not all 0s and it is not all 1s, and is the most common case (e.g., bits 23-30 are not the value 0 or the value 255).
- The *exponent* is a *signed integer* in ***biased*** form. The exponent has a value exp - *bias*, where the "bias" is 2^{k-1} - 1, and where *k* is the number of bits in the exponent (k=8 for floats, meaning that the bias is 2^7 - 1 = 127). For floats, the exponent range is -126 to +127.
- The *fraction* is interpreted as having a fractional value *f*, where $0 \leq f < 1$, and having a binary representation of $0.f_{n-1} \cdots f_1 f_0$, with the binary point to the left of the most significant bit.
- The significand is defined to be M = 1 + *f*. This is **an implied leading 1 representation**, and a trick for getting an additional digit for free!

An extra bit of precision for free?

Single precision (`float`)

31	30 … 23	22 … 0
s	exponent (8 bits)	fraction (23 bits)

- Yes! We can always adjust the exponent so that the significand is in the range $1 \leq M < 2$ (assuming no overflow), so we don't need to explicitly represent the leading bit, because it is always 1 (very cool!)
- Remember, the designers of IEEE Floating Point wanted the best system, and this is a cool idea.

Example:

31	30 … 23	22 … 0
0	01111110	00000000000000000000000

- Sign: 0 (positive)
- Exponent: 01111110 = 126 (biased), so exponent of 2 will be 126 - 127 = -1
- Fraction: 0, which is assumed to be 1.0 (binary), which is 1.0 decimal
- Therefore, this number represents $+1.0 \times 2^{-1} = 0.5$ **<u>(to the converter!)</u>**

Let's try some more...

Single precision (`float`)

31	30 — 23	22 — 0
s	exponent (8 bits)	fraction (23 bits)

- Example:

31	30 — 23	22 — 0
0	10000100	01010000000000000000000

- Sign: 0 (positive)
- Exponent: 10000100 = 132 (biased), so exponent of 2 will be 132 - 127 = 5
- Fraction: 0101, which is assumed to be 1.0101 (binary), which is:

$$1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 1.3125$$

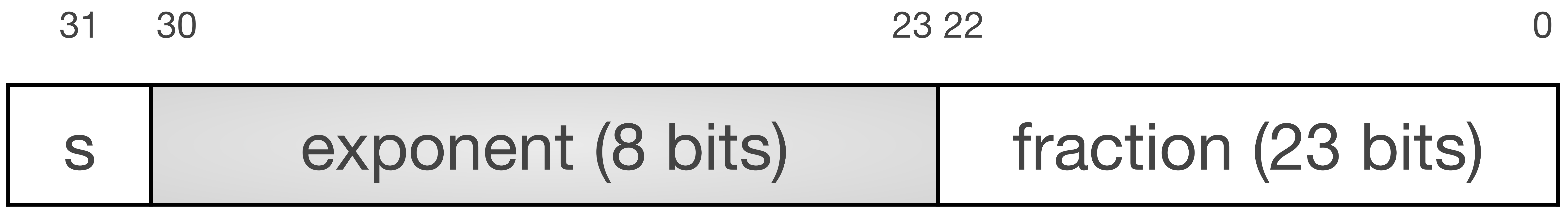
- Therefore, this number represents +1.3125 x 2^5 = 42.0 **<u>(to the converter!)</u>**

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

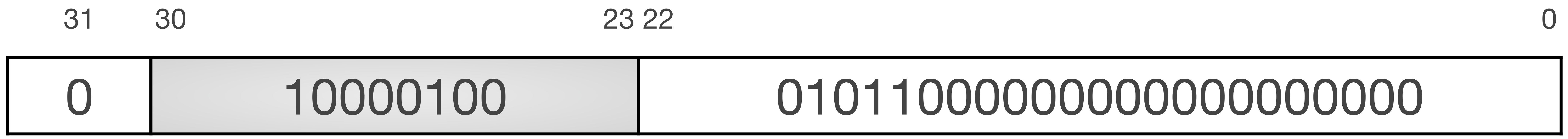
<u>http://web.stanford.edu/class/cs107e/float/</u>

Let's try some more...

Single precision (`float`)



- Example:



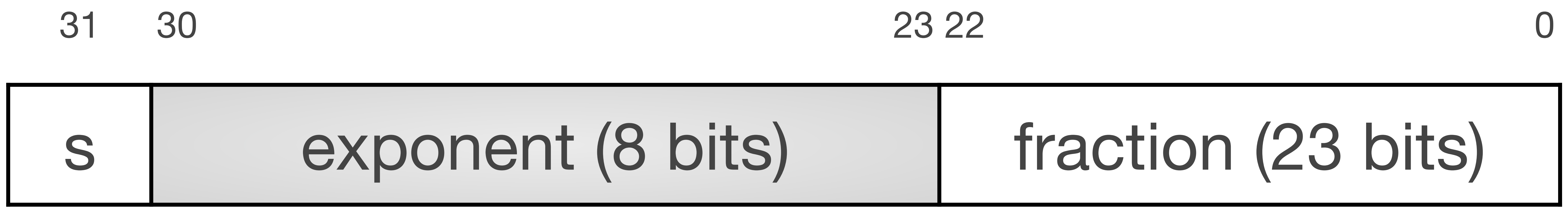
- Sign: 0 (positive)
- Exponent: 10000100 = 132 (biased), so exponent of 2 will be 132 - 127 = 5
- Fraction: 01011, which is assumed to be 1.01011 (binary), which is:

$$1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 1.34375$$

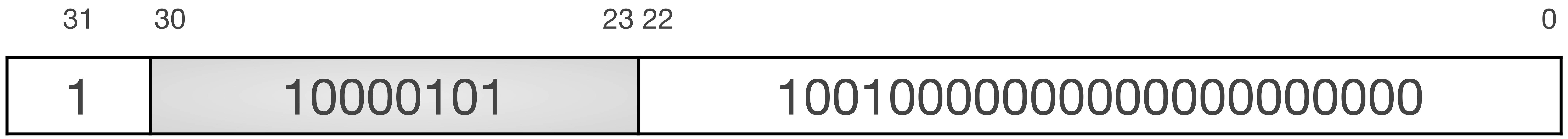
- Therefore, this number represents +1.34375 x 2^5 = 43.0 **(to the converter!)**

Let's try some more...

Single precision (`float`)



- Example:



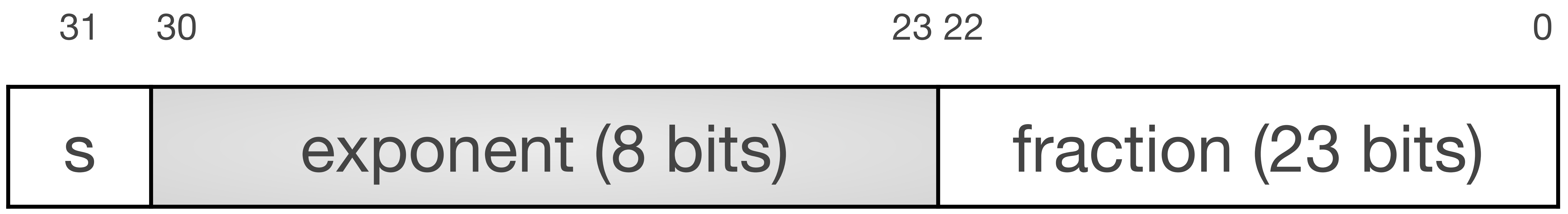
- Sign: 1 (negative)
- Exponent: 10000101 = 133 (biased), so exponent of 2 will be 133 - 127 = 6
- Fraction: 1001, which is assumed to be 1.1001 (binary), which is:

$$1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 1.5625$$

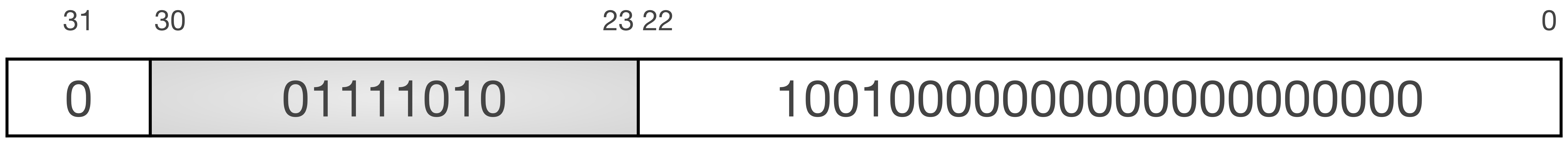
- Therefore, this number represents -1.5625 x 2^6 = -100.0 **<u>(to the converter!)</u>**

Let's try some more...

Single precision (`float`)



- Example:



- Sign: 0 (positive)
- Exponent: 01111010 = 122 (biased), so exponent of 2 will be 122 - 127 = -5
- Fraction: 1001, which is assumed to be 1.1001 (binary), which is:

$$1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 1.5625$$

- Therefore, this number represents +1.5625 x 2^{-5} = 0.048828125

(to the converter!)

Arithmetic with Floating Point Numbers

Take a look at this example:

```
void main()
{
    double a = 0.1;
    double b = 0.2;
    double c = 0.3;
    double d = a + b;
    printf("0.1 + 0.2 == 0.3 ? %s\n", a + b == c ? "true" : “false");
}
```

```
0.1 + 0.2 == 0.3 ? false
```

The rounding that happens during the calculation of 0.1 + 0.2 produces a different number than 0.3!

Arithmetic with Floating Point Numbers

```c
#include "printf.h"
#include "ftoa.h"

void main() {
    double a = 0.1;
    double b = 0.2;
    double c = 0.3;
    double d = a + b;

    char bufa[30],bufb[30],bufc[30],bufd[30];
    ftoa(a,bufa,20);
    ftoa(b,bufb,20);
    ftoa(c,bufc,20);
    ftoa(d,bufd,20);

    printf("0.1 + 0.2 == 0.3 ? %s\n", a + b == c ? "true" : "false");
    printf("0.1:\t%s\n",bufa);
    printf("0.2:\t%s\n",bufb);
    printf("0.3:\t%s\n",bufc);
    printf("a + b:\t%s\n",bufd);
}
```

```
0.1 + 0.2 == 0.3 ? false
0.1:   0.10000000000000000000
0.2:   0.20000000000000000000
0.3:   0.30000000000000000000
a + b: 0.30000000000000004440
```

Arithmetic with Floating Point Numbers

Here is another example:

```
#include “printf.h”
void main()
{
    printf("16777224.0f == 16777225.0f ? %s\n",
            16777224.0f == 16777225.0f ? "true" : "false");
}
```

Arithmetic with Floating Point Numbers

Here is another example:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    printf("16777224.0f == 16777225.0f ? %s\n",
           16777224.0f == 16777225.0f ? "true" : "false");
    return 0;
}
```

```
16777224.0f == 16777225.0f ? true
```

It turns out that `16777225` is an integer that you cannot represent as a 32-bit float.

- The IEEE Floating Point Standard was a carefully thought-out way to get the most out of a discrete set of bits. It may not be simple, but it is a great study in good engineering design.
- Floating point numbers represent a very large range, in a limited number of bits. A 32-bit float can only hold a bit over 4 billion numbers and has a range of **-3.4E-38 to +3.4E+38**. Not only is this literally an infinite number of reals that the format must try and represent, but that is a phenomenal range of numbers. The 64-bit double range is **-1.7E-308 to +1.7E+308** (!)
- Most numbers are, therefore, only represented approximately in float format, including many integers. Example:
  - 1 trillion = 1,000,000,000,000, and in 32-bit floating point, it is actually represented by the value 999,999,995,904, off by 4096!
  - You almost certainly *don't* want to use floats for currency!

Floating Point Takeaways

- You have to be very careful with your arithmetic when you are dealing with floats:
  - Associativity does not hold for numbers far apart in the range.
  - Many numbers are not exact (e.g., 0.1, 0.4, etc.)
  - Equality comparison operations are often unwise.

References and Advanced Reading

- References:
  - IEEE 754: https://en.wikipedia.org/wiki/IEEE_754
  - IEEE Floating point: http://steve.hollasch.net/cgindex/coding/ieeefloat.html
  - Floating point arithmetic: https://en.wikipedia.org/wiki/Floating-point_arithmetic#Dealing_with_exceptional_cases
- Advanced Reading:
  - Comparing floats using equality: https://stackoverflow.com/questions/1088216/whats-wrong-with-using-to-compare-floats-in-java
  - Floating point converter: https://www.h-schmidt.net/FloatConverter/IEEE754.html
  - What Every Computer Scientist Should Know About Floating-Point Arithmetic
  - Floating point rounding errors: https://softwareengineering.stackexchange.com/questions/101163/what-causes-floating-point-rounding-errors
  - Why do we have a bias in floating point exponents?