Digital I/O

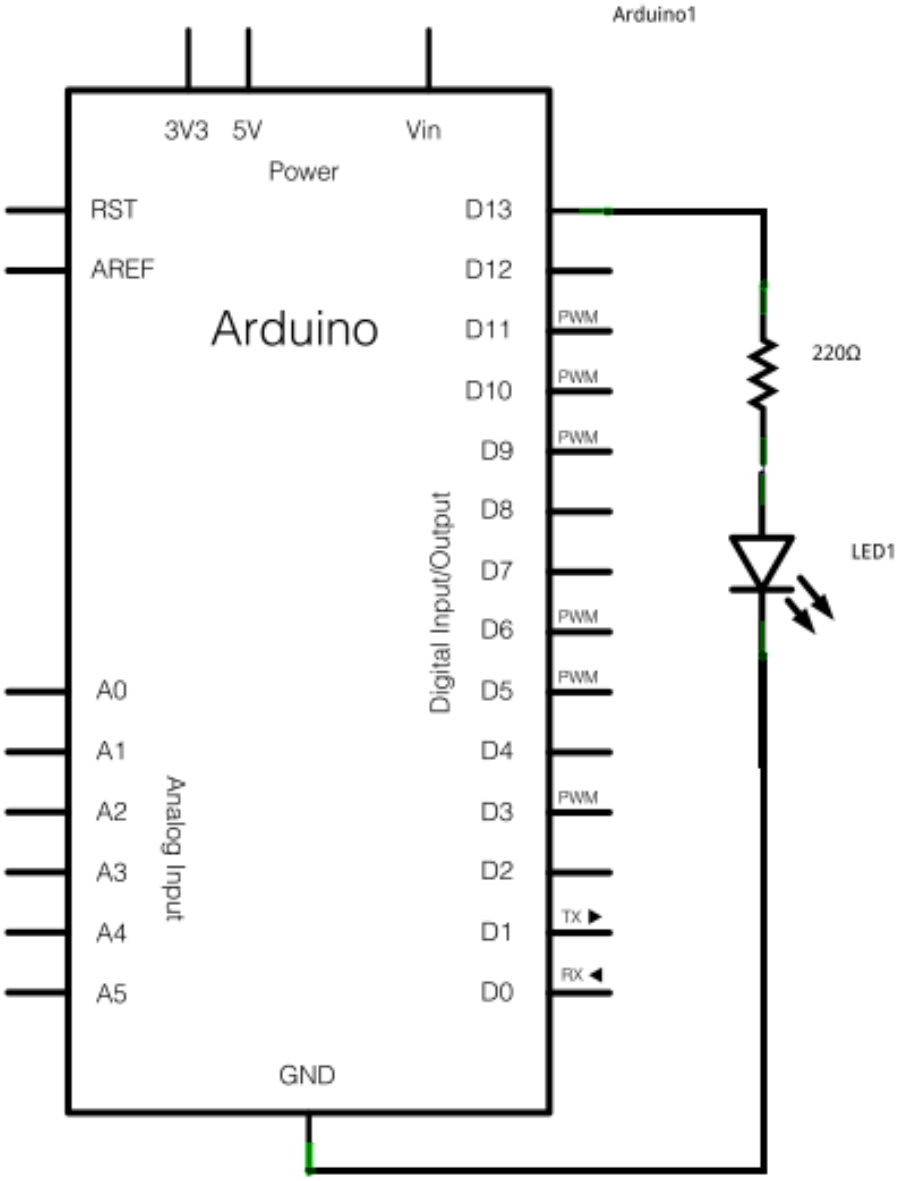
CSE 132

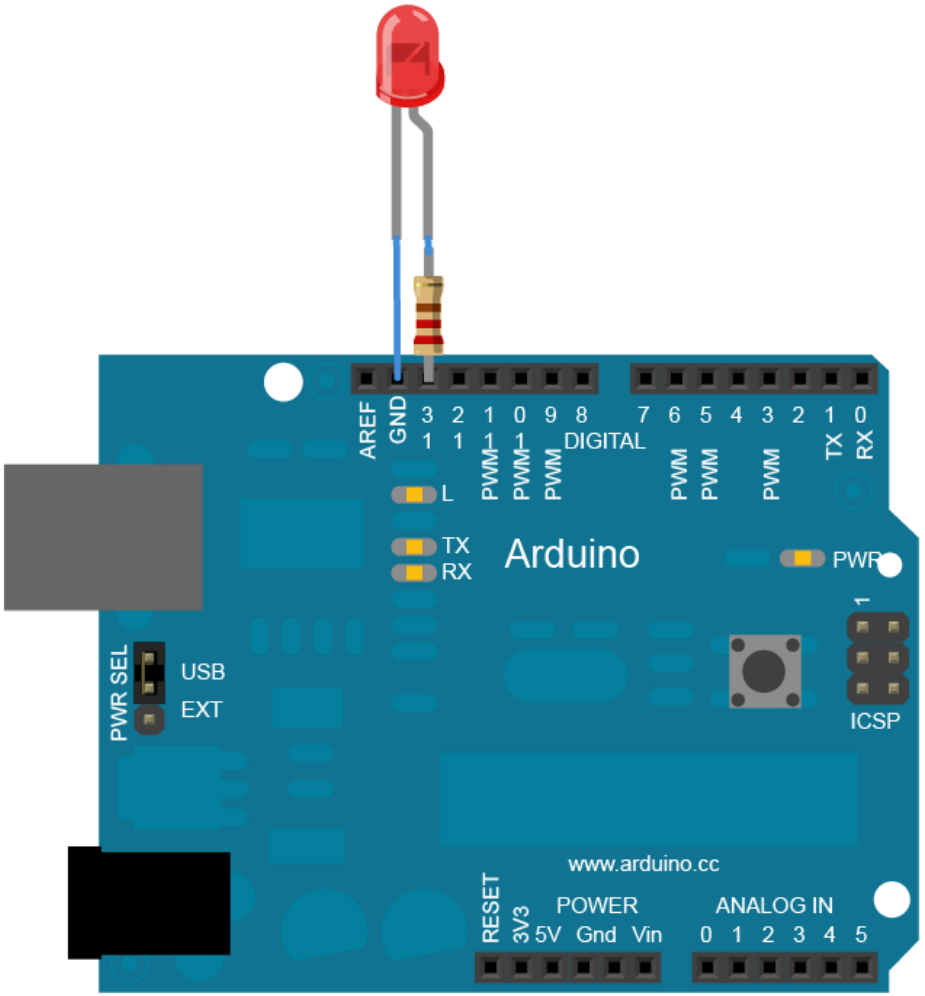
Arduino Input/Output

- 20 pins on physical chip can be configured to do digital input, digital output, analog input, analog output (not all pins can do each function)
- We first configure pins at startup, then use them

```
const int myPin = 13;
void setup() {
    pinMode(myPin, OUTPUT);
}
void loop() {                          //generates square wave
    digitalWrite(myPin, LOW);
    digitalWrite(myPin, HIGH);
}
```

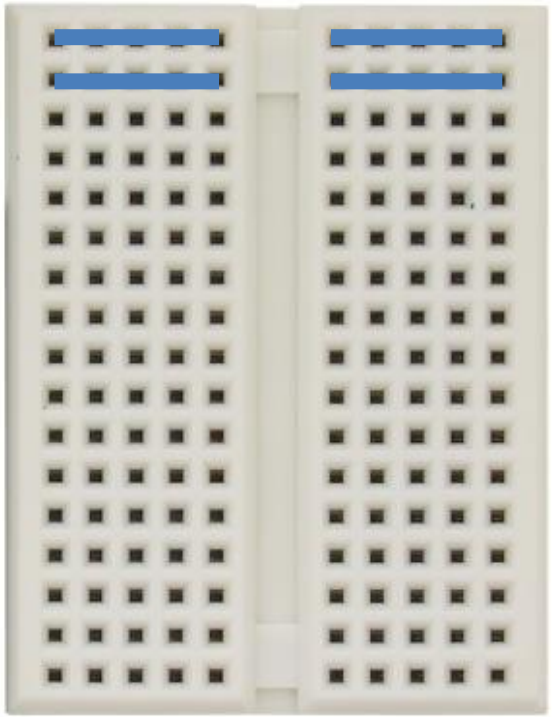
Digital Output LED (light-emitting diode)





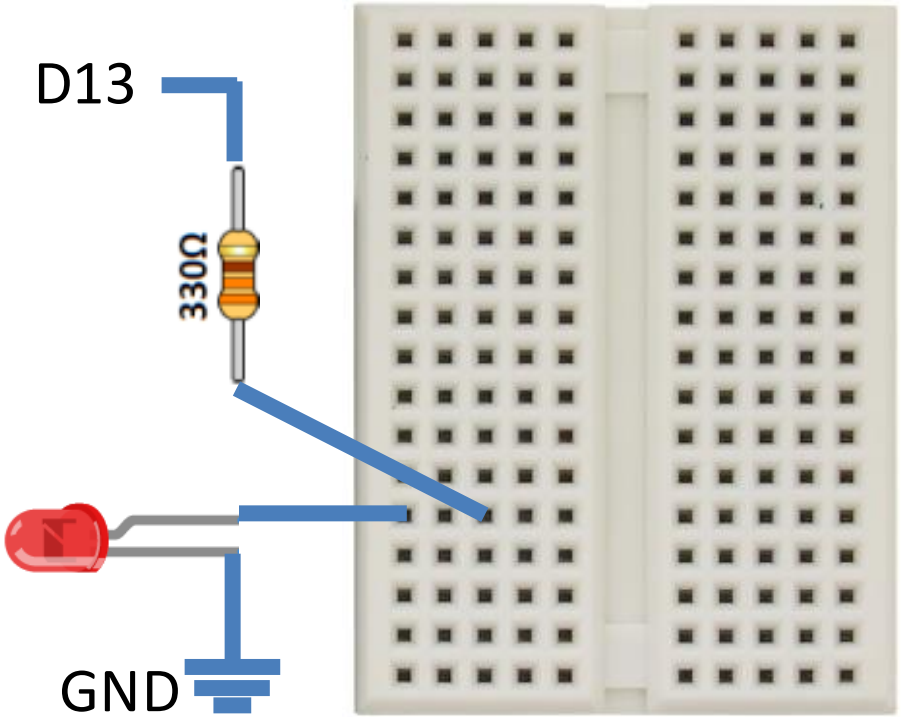
Building Circuits

- 5 horizontal holes are connected:



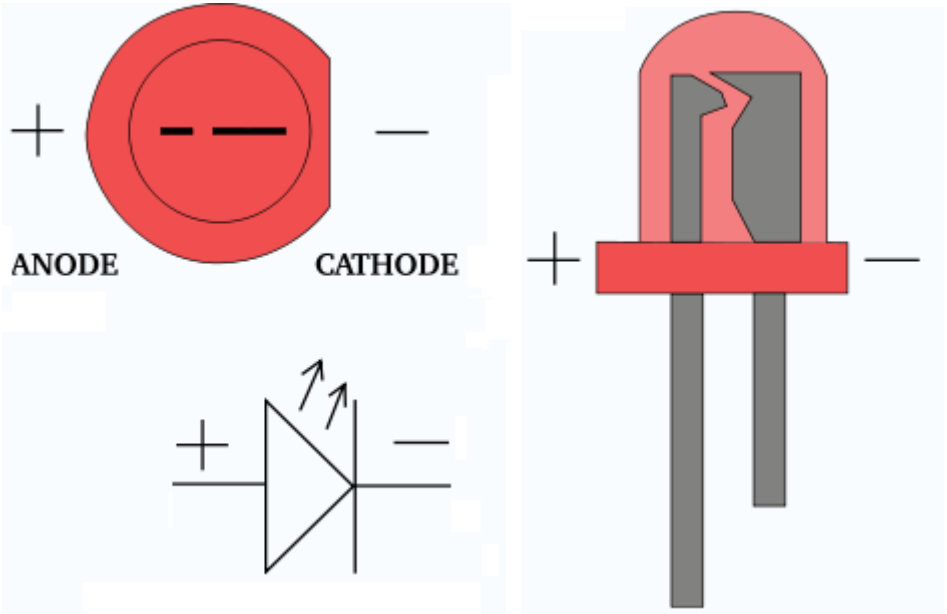
Building Circuits

- Connect components using breadboard:



LEDs

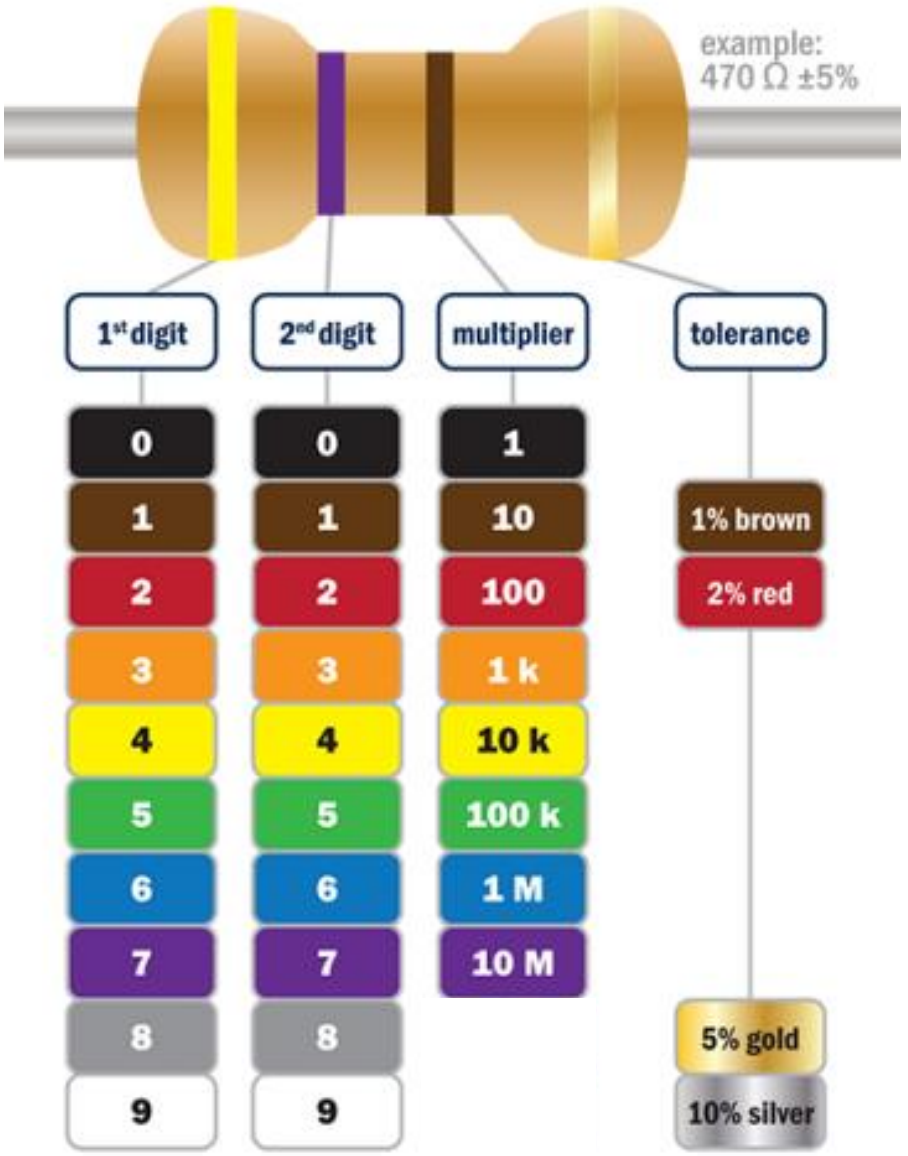
- Anode is “+” side, cathode is “–” side
- Anode has longer lead (assuming not clipped)
- Cathode is the flat side on LED body



Resistor Color Codes

- 1st two digits are values
- 3rd digit is multiplier
- 4th digit is tolerance
- 200 to 500 Ω gives good light out of LED
- We will use 330 Ω, or **orange-orange-brown**

33 x 10^{1} Ω



What Inputs Can You Think Of?

- Keyboard
- Mouse
- Webcam
- Temperature
- Microphone
- USB
- Internet
- External memory
- Power
- Emergency stop button
- Proximity detector
- Motion detector

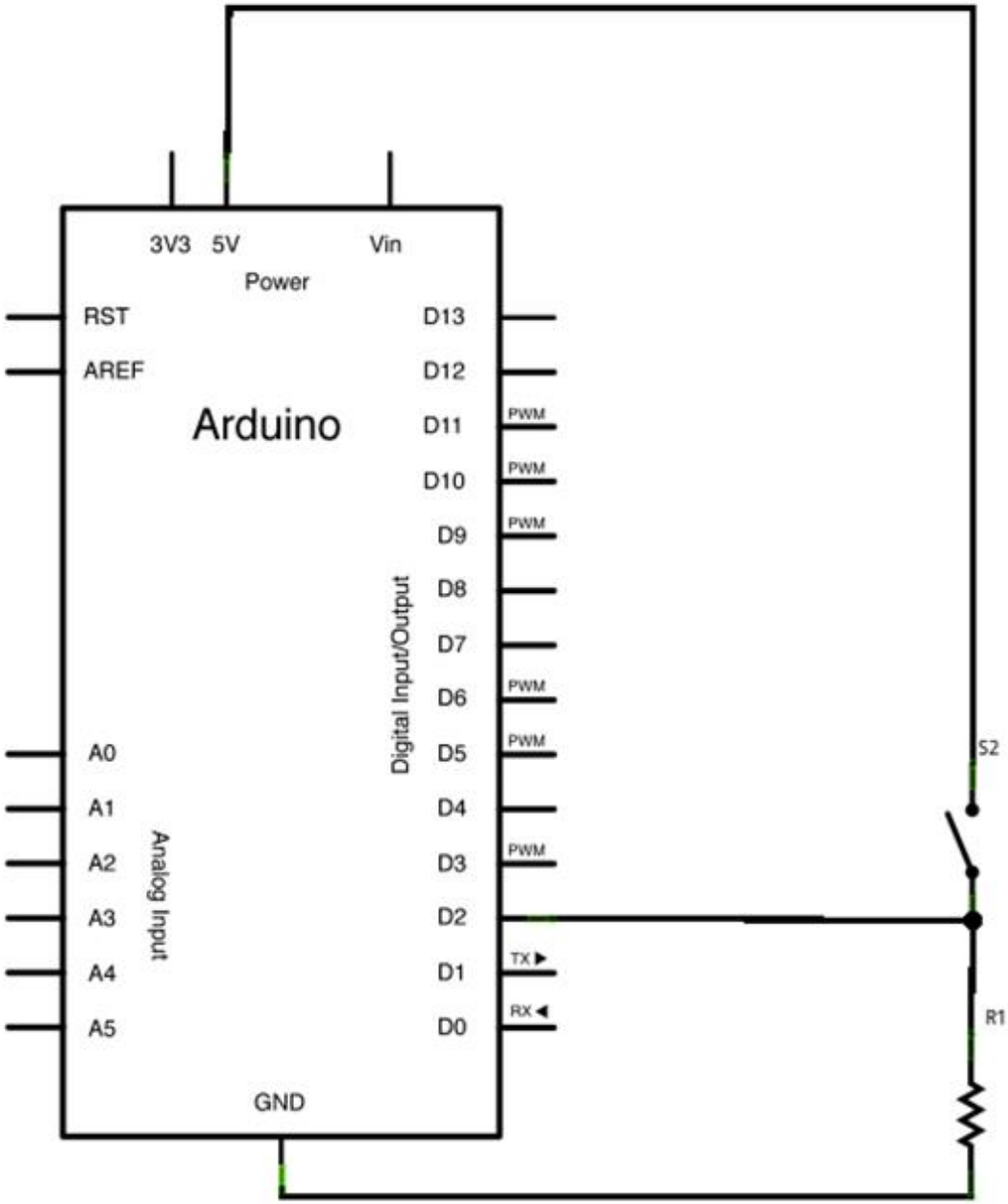
Digital Inputs from Physical World

- Example use case
  - Proximity detector
  - Garage door safety beam
- Sensing technologies
  - Capacitive
  - Inductive
  - Optical
  - Radar
  - Hall-effect (magnetic)

Simplest Digital Input

- Push-button switch
  - Pressed = 1, HIGH, TRUE
  - Not-pressed = 0, LOW, FALSE
- Electro-mechanical device
  - When button is pressed, electrical contacts conduct
  - When make/break contact, the contacts can bounce
  - This bouncing can happen over milliseconds
  - But software operates at microsecond scales
  - Even a simple push-button isn't so simple!

Push-button Schematic



← Push here

← Voltage here goes high

Software

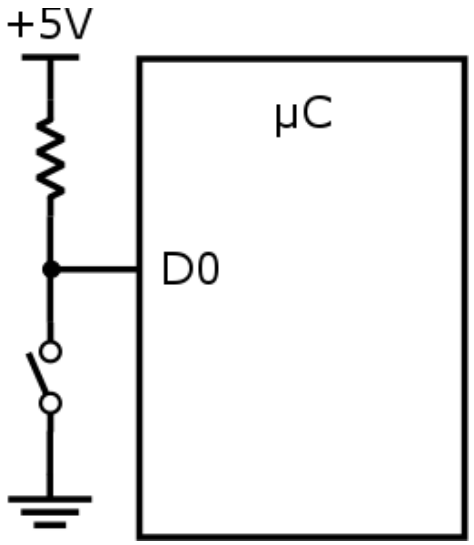
- Switch between input pin and +5V
  - Input goes HIGH when switch is pressed
  - Input goes LOW when switch is not pressed

```
setup() {
   pinMode(pin, INPUT);
}

loop() {
   inputVal = digitalRead(pin);
}
```

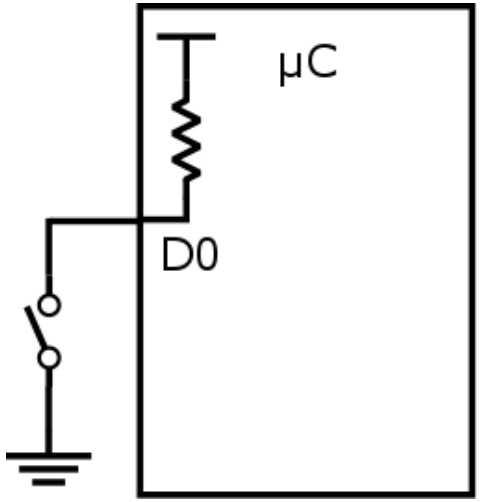
Watch out for signaling convention!

- Switch between input pin and GND
  - Input LOW when switch is pressed
- Why would one do this?



- Because the resistor is available, built into the processor

```
setup() {
   pinMode(pin, INPUT_PULLUP);
}
```



Switch “Debouncing”

Read switch state

Wait enough time for switch to quit bouncing

Read switch state again

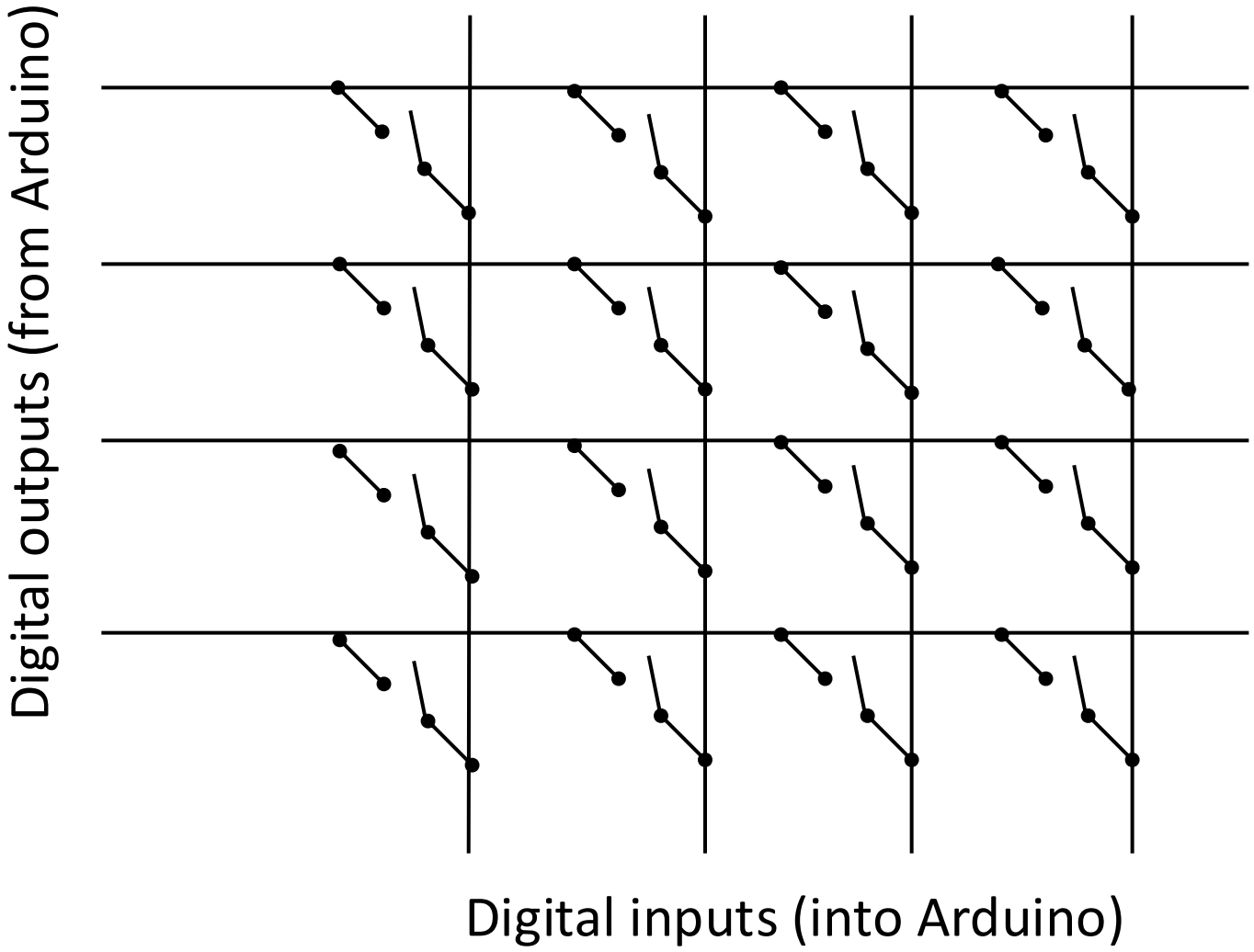
if two switch states agree

    Done

else

    Start over

What about many switches?



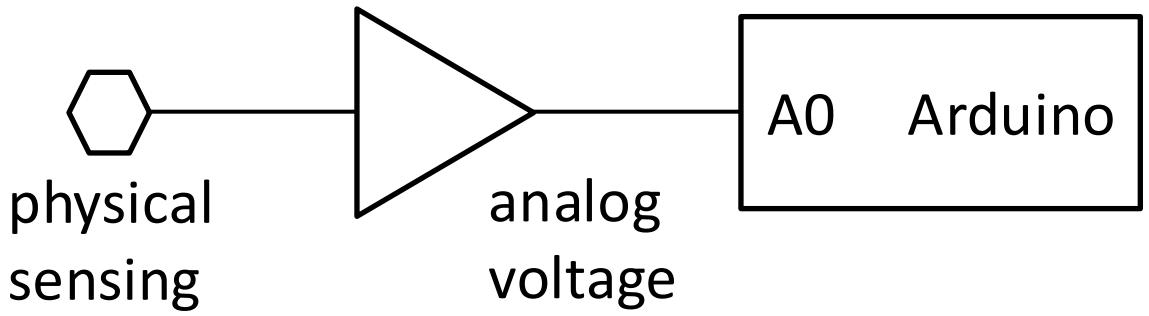
Quiz Time

- Go to Canvas and answer the single question for Quiz 2A

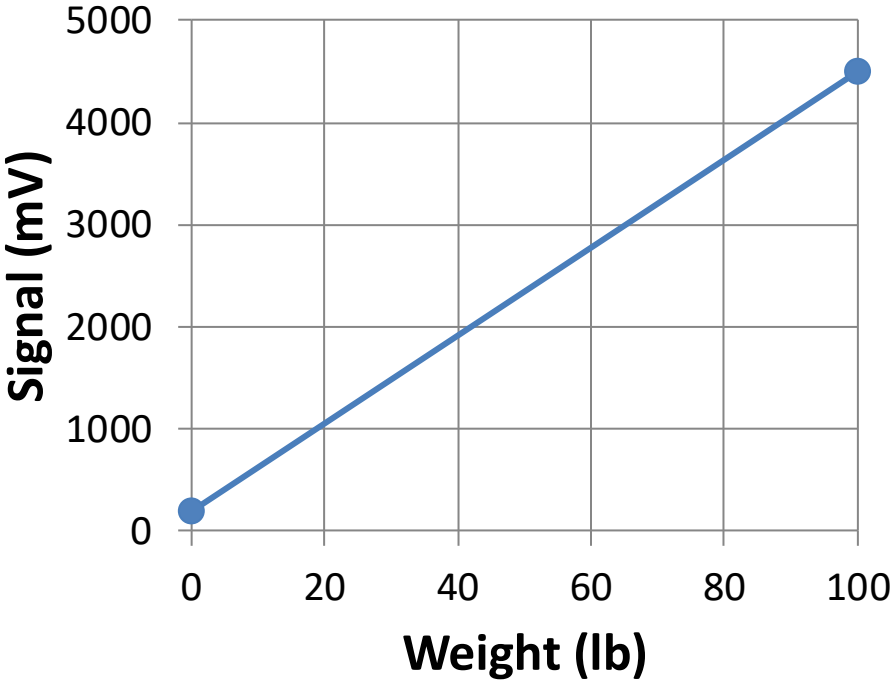
- What does LED stand for?
  - Light Emitting Diode
  - Light End Device
  - Long Electricity Digit
  - Last Ever Device

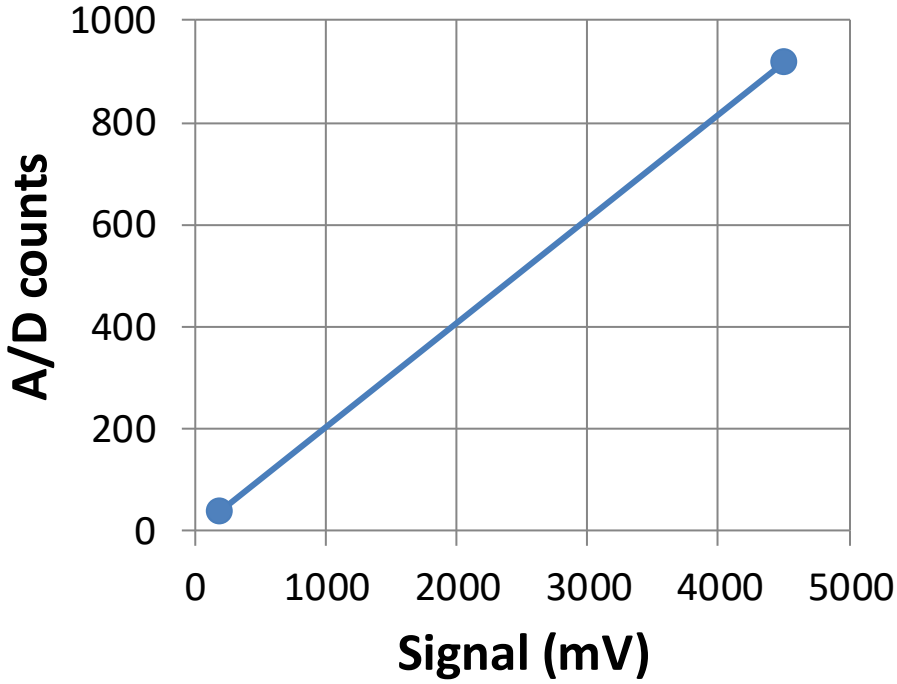
Analog to Digital Conversion

- Convert physical property to voltage signal
- A/D converter on Arduino converts voltage signal to digital representation
  - 10-bit A/D converter has range 0 to $2^{10} - 1$ (0 to 1023) for voltage range 0 to V_{REF}



Understanding Ranges





$$signal = m \times weight + b$$

$$signal = 43\frac{mV}{lb} \times weight + 200mV$$

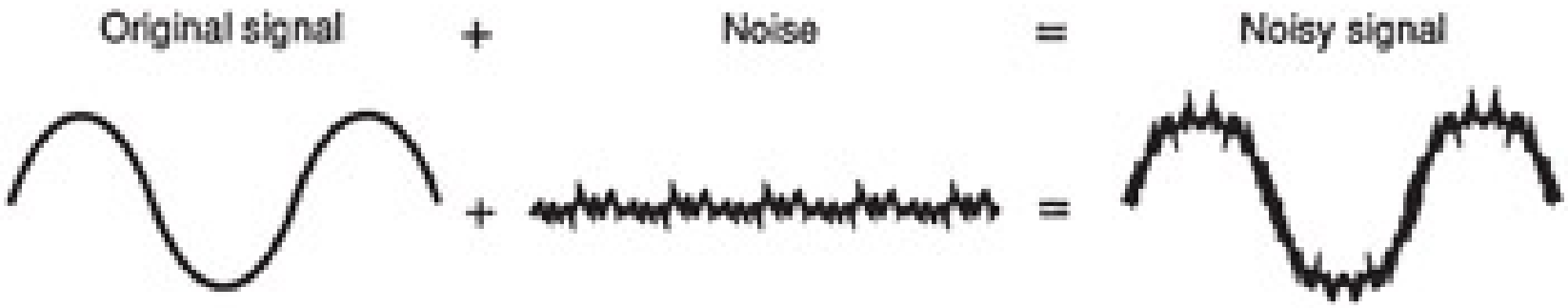
$$counts = m \times signal + b$$

$$counts = 0.2\frac{cnt}{mV} \times signal + 0$$

$$counts = 8.6\frac{cnt}{lb} \times weight + 40$$

$$weight = 0.116\frac{lb}{cnt} \times counts - 4.65$$

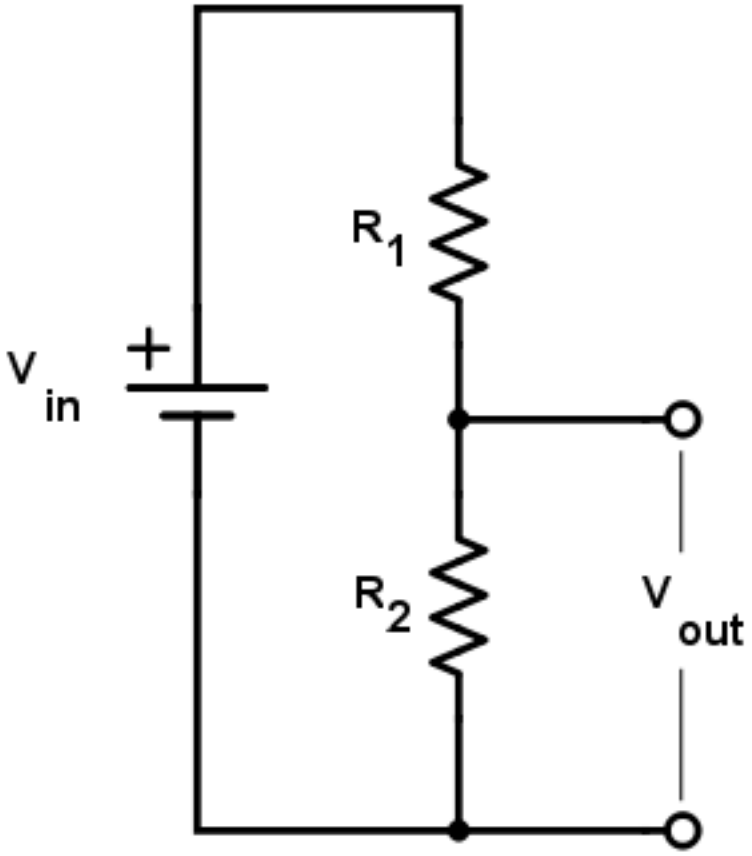
Noisy Analog Signals



- Noise is ever present in analog signals
- For stable signal, quick fix is to average several readings

$$avg = \frac{1}{N}\sum_{1}^{N} A/D\ input_i$$

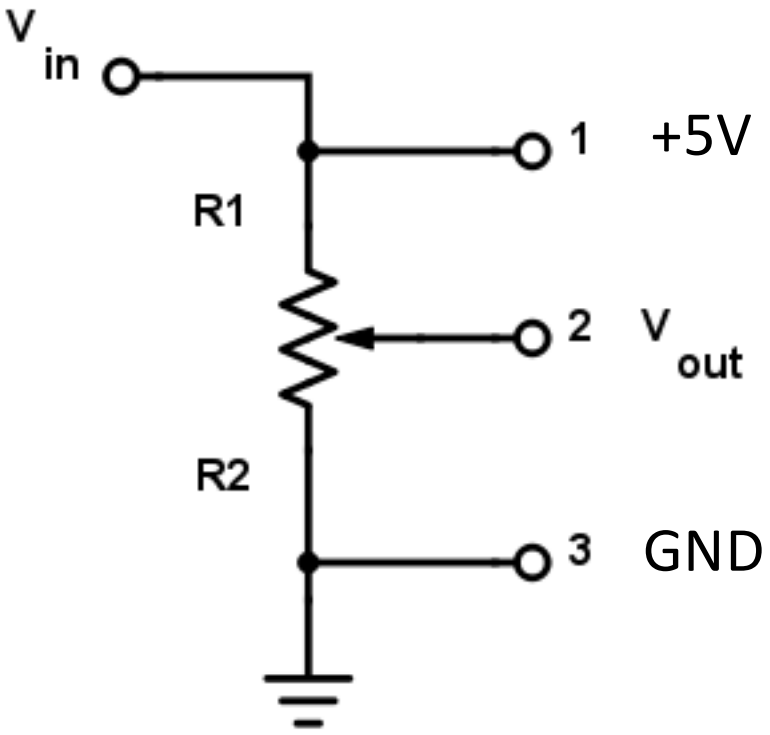
Simple Voltage Divider

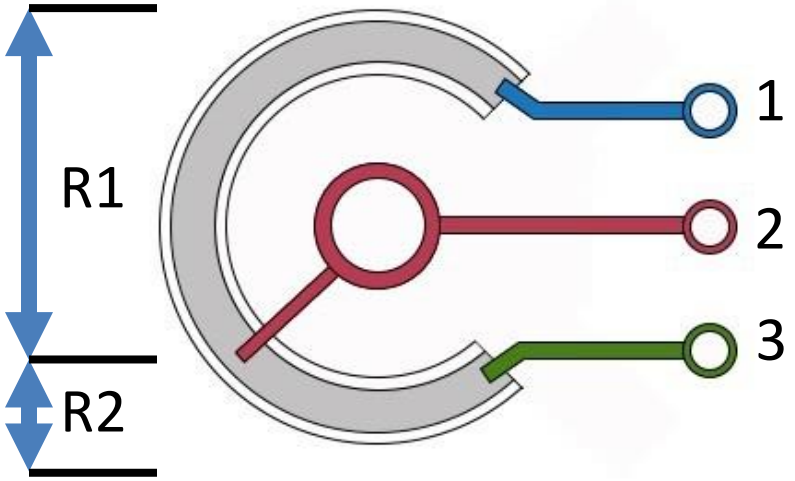


$$V_{out} = \frac{R_2}{R_1 + R_2} V_{in}$$

https://www.allaboutcircuits.com/tools/voltage-divider-calculator

Simple Voltage Divider





$$V_{out} = \frac{R_2}{R_1 + R_2} 5\text{V}$$

https://www.allaboutcircuits.com/tools/voltage-divider-calculator

https://ez.analog.com/ez-blogs/b/engineering-mind/posts/designing-a-kelvin-varley-potentiometer-part-1-of-3-resistors-in-voltage-divider

What about fractions?

- Positional number systems work on both sides of the decimal point (radix point).

- If radix is r (n integer digits, m fractional digits):

$$val = a_{n-1} \cdot r^{n-1} + a_{n-2} \cdot r^{n-2} + \ldots + a_0 \cdot r^0 + a_{-1} \cdot r^{-1} + a_{-2} \cdot r^{-2} + a_{-m} \cdot r^{-m}$$

- e.g., $wx.yz_{16} = w \cdot 16 + x + y \cdot 16^{-1} + z \cdot 16^{-2}$
  or $wx.yz_2 = w \cdot 2 + x + y \cdot 2^{-1} + z \cdot 2^{-2}$

Two kinds of numbers

- Integers – radix point is assumed to be at the far right end of the digits:
  - E.g. 01001110.
- Fixed point – radix point is at a given, fixed location:
  - E.g. 0100.1110
  - 0.1001110 is a common representation on digital signal processors

Q notation

- Qn.m means a number with n+m bits (digits), n integer and m fractional. Sign bit is often in addition to this.
- E.g., Q3.4 for 0100.1100, with value 4.75
- Qm means a number with m+1 bits, m are fractional
- E.g., Q3 notation would have 4 bits and the following values
  - wxyz = w.xyz = $w \cdot (-1) + x \cdot (1/2) + y \cdot (1/4) + z \cdot (1/8)$
  - range is now -1 to +7/8, with resolution 1/8

Floating point representation

What about the reals? Use scientific notation.

In base 10: $x \cdot 10^y$ $0.32 \times 10^{-3} = 0.00032$

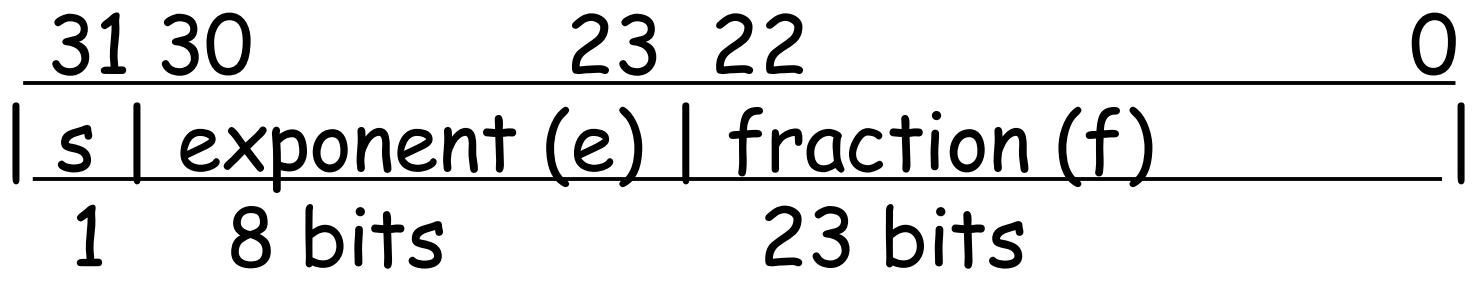
In base 2: $x \cdot 2^y$ called floating point

IEEE Floating Point

- Limited range of x and y (fixed # of bits) means we cannot represent every real number exactly

- IEEE std. 754 describes a standard form for floating point number representations
  - Single precision is 32 bits in size
  - Double precision is 64 bits in size

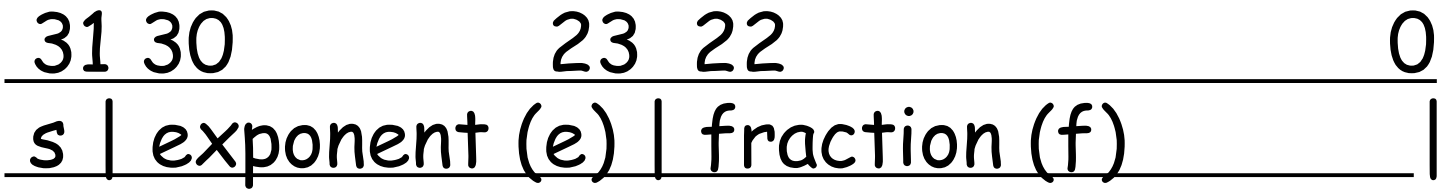
Single precision (32 bits)



value = $(-1)^s \times 2^{e-127} \times \underline{1}.f$

↑ hidden "1"

range = $\pm\ 2 \times 10^{\pm 38}$



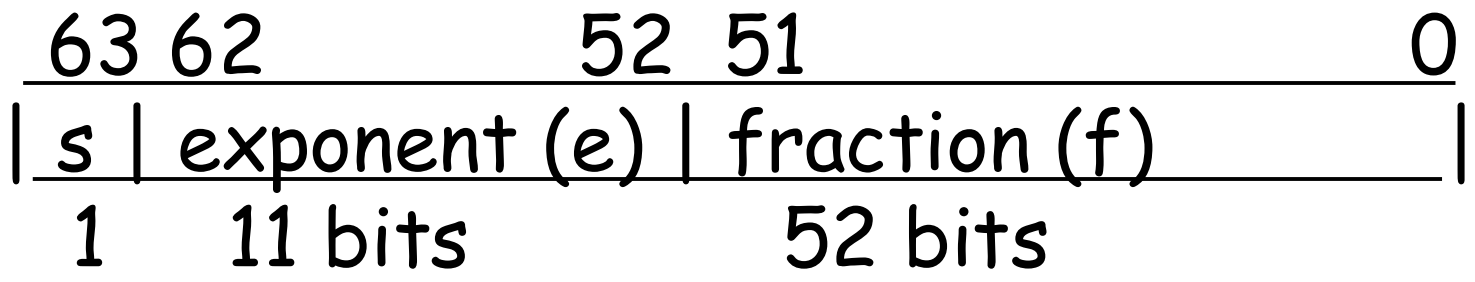
- s = 0, e = 0, f = 0 $\Rightarrow$ value = zero
- e = 255, f = 0 $\Rightarrow$ value = $(-1)^s \times$ infinity
- e = 255, f ≠ 0 $\Rightarrow$ value = "not a number" triggers exception
- e = 0, f ≠ 0 $\Rightarrow$ denormalized

$$\text{value} = (-1)^s \times 2^{-126} \times \underline{0}.f$$

↑ hidden "0"

- Note use of sign-magnitude for entire number, and excess notation (excess 127) for exponent

Double precision (64 bits)



value = $(-1)^s \times 2^{e-1023} \times \underline{1}.f$

↑ hidden "1"

range = $\pm\ 2 \times 10^{\pm 308}$

$e = 0,\ f \neq 0 \Rightarrow$ denormalized

value = $(-1)^s \times 2^{-1022} \times \underline{0}.f$

Text – Characters and Strings

- ASCII – American Standard Code for Information Interchange
  - 7-bit codes representing basic Latin characters and numbers [A-Z, a-z, 0-9], some common punctuation, and control characters
  - There are a number of extensions to 8 bits, but only the 7-bit codes really standard.
- Unicode – 8- or 16-bit codes extending to a much wider set of languages
  - The first 128 codes are equivalent to the 7-bit ASCII standard

C Strings

- Strings are sequences of ASCII characters, stored one byte per character (8 bits), terminated by a NULL (zero) character
- E.g., “Hello!”

01001000	‘H’	0x48
01100101	‘e’	0x65
01101100	‘l’	0x6c
01101100	‘l’	0x6c
01101111	‘o’	0x6f
00100001	‘!’	0x21
00000000	NULL	0x00

ASCII Facts

- Numerical digits are assigned in order of increasing value

  i.e., ‘0’ = 0x30

  ‘1’ = 0x31

  ‘2’ = 0x32

  ‘9’ = 0x39

- For single character, value conversion is simply a difference of 0x30

More ASCII Facts

- Letters are also assigned in lexicographical order:

  ‘A’ = 0x41
  ‘B’ = 0x42

  ‘Z’ = 0x5a

  ‘a’ = 0x61
  ‘b’ = 0x62

  ‘z’ = 0x7a

- Upper/lower case conversion is simply a difference of 0x20

Still More ASCII Facts

- First 32 characters (0-0x1f) are control codes:

0x00	^@	null (C string terminator)
0x07	^G	bell
0x0a	^J	line feed
0x0c	^L	form feed
0x0d	^M	carriage return

Line breaks are not standardized

- End of line conventions differ by operating system:
  - In MS Windows: 0x0a, 0x0d is end of line
  - In Unix/Linux: 0x0a is end of line
  - 0x0a, linefeed, is sometimes called ‘newline’
- In C, ‘\n’ is mapped to OS end of line termination convention

Java Strings

- Strings are represented via the class “String”
- String objects are immutable
- The character encoding is system specific, e.g., either UTF-8 or UTF-16 (typical).
- The length is an instance variable in the object (in most implementations)
- The characters are stored in a char[] array (again, in most implementations)

Unicode

- Standard for character representation
  - Supports wide variety of languages, symbols
- UTF-8
  - Variable length code with 8-bit code units
  - U+0000 to U+007F are the same as ASCII
- UTF-16
  - Uses 16-bit code units, also variable length
  - Latin + Greek + Cyrillic + Coptic + Armenian + Hebrew + Arabic + Syrian + Tāna + N'Ko fit in 16 bits
- UTF-32
  - Uses 32-bit code units, fixed length

Images

- Consider the following bits:

0x00240008188142 3c

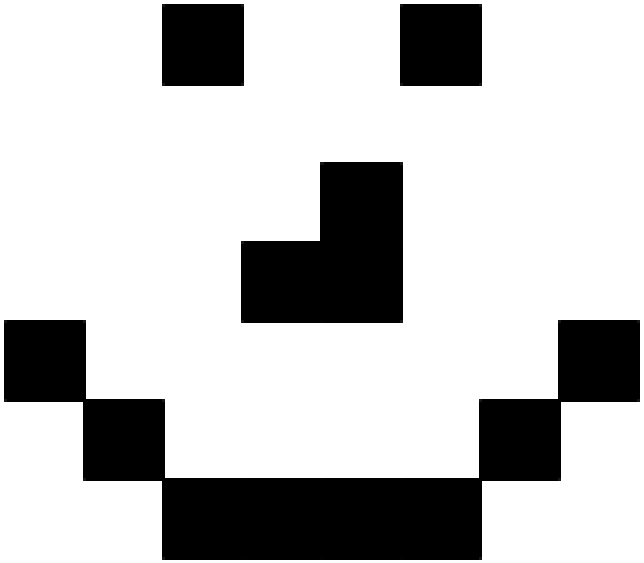
0000 0000 0010 0100 0000 0000 0000 1000

0001 1000 1000 0001 0100 0010 0011 1100

- Make 1 dark and 0 light:

Images

- Arrange in rows, one byte per row:

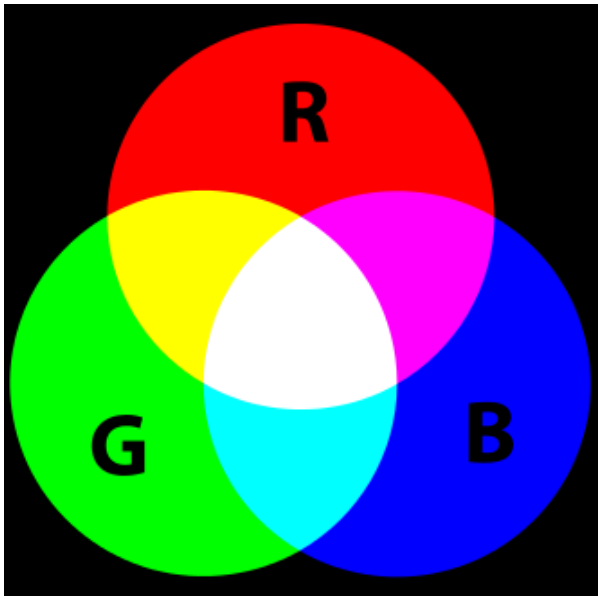


- Each bit is a “pixel” in the image

Add color and more pixels

Color

- Additive color – primaries Red, Green, Blue



- Position close together and put diffuser above
  - This builds one pixel

Logistics

- Assignment 1 is due Jan 26
  - Demos *only* available in first 15 minutes of lab
  - Don't assume lab time available to complete it!!!!
  - Quiz 1B is also due Jan 26 (in the evening is OK)
  - Yes, the logic puzzles are puzzles! Minor points off for not getting them all right
- Module 2 is up now
  - Studio 2 is Monday Jan 26
    - Prep material is posted on webpage
    - Get signed out by a TA in studio
  - Assignment 2 is due Monday Jan 2