

Digital I/O

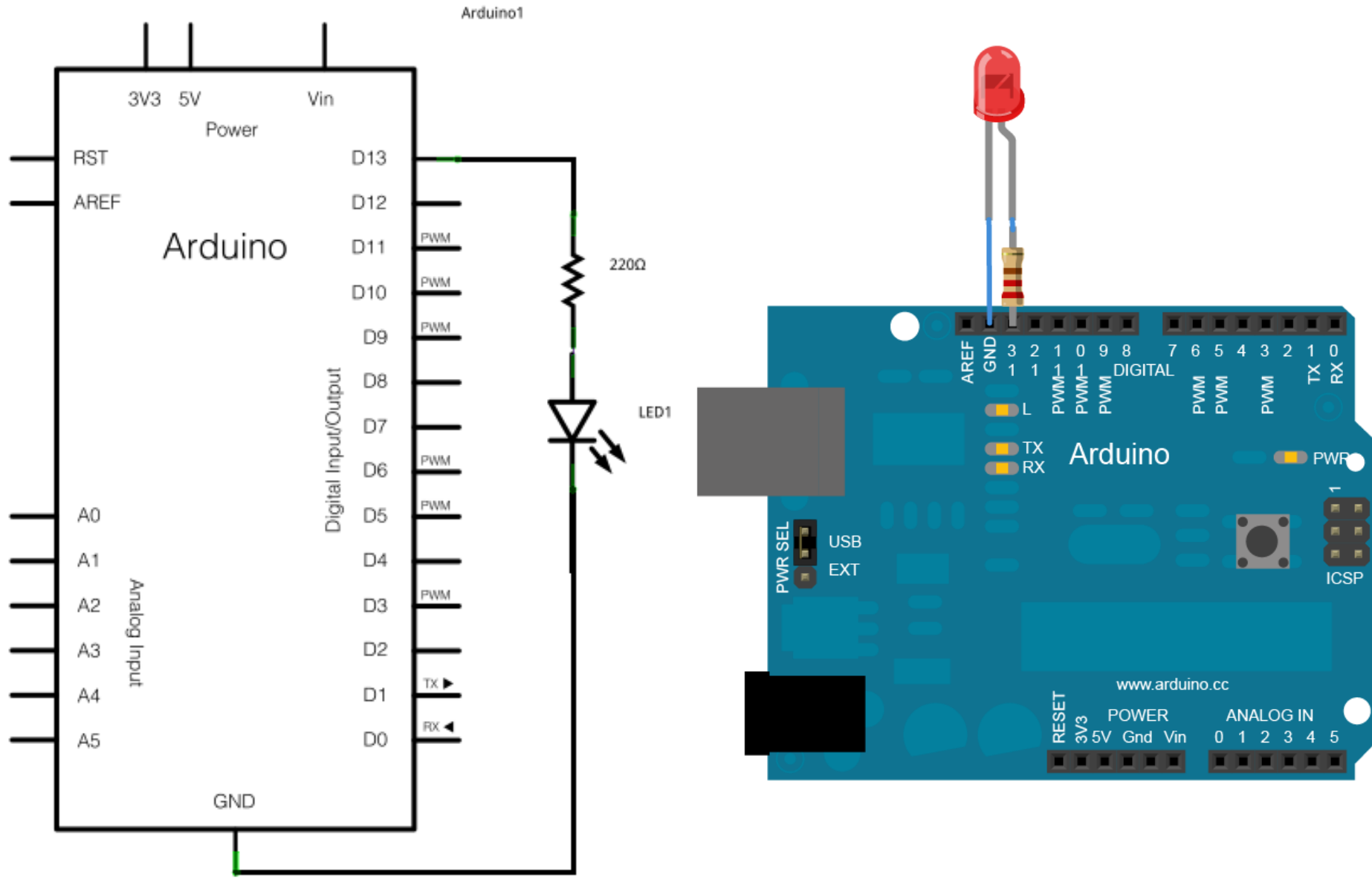
CSE 132

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)



Voltage, Current, Resistance

- Voltage (V, volts) – electrical “pressure”
 - A difference in potential between two points
 - Always measured across a component
- Current (I, amps) – the rate charge flows
 - Measured through a component
- Resistance (R, ohms Ω) – opposition to that flow
- Ground (GND) is the 0 V reference
 - Current needs a complete loop back to GND
 - No loop $\Rightarrow$ no current $\Rightarrow$ nothing happens

Ohm's Law

$$V = I \times R$$

$$I = V / R$$

$$R = V / I$$

Water analogy

Voltage = water pressure

Current = gallons per minute

Resistance = narrowness of the pipe

Squeeze the pipe (more R) and less water flows for the same pressure.

On the Arduino, digital pins are a 0 V / 5 V world.

LOW = 0 V (the pin is tied to GND). HIGH = 5 V. `digitalRead()` reports which side of ~2.5 V the pin sits on.

Reading Circuit Diagrams

A schematic shows what is connected to what — not where you put it on the breadboard.



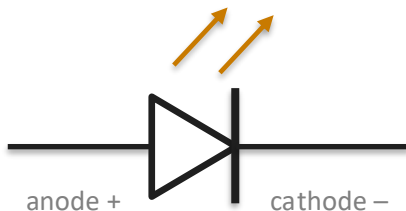
Voltage source
the 5 V pin and GND



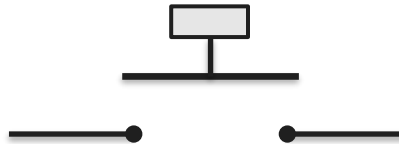
Ground (GND)
the 0 V reference



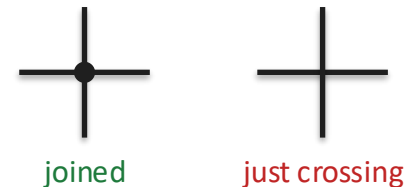
Resistor
limits current (Ω)



LED
bar = flat side = toward GND



Push button
open until pressed



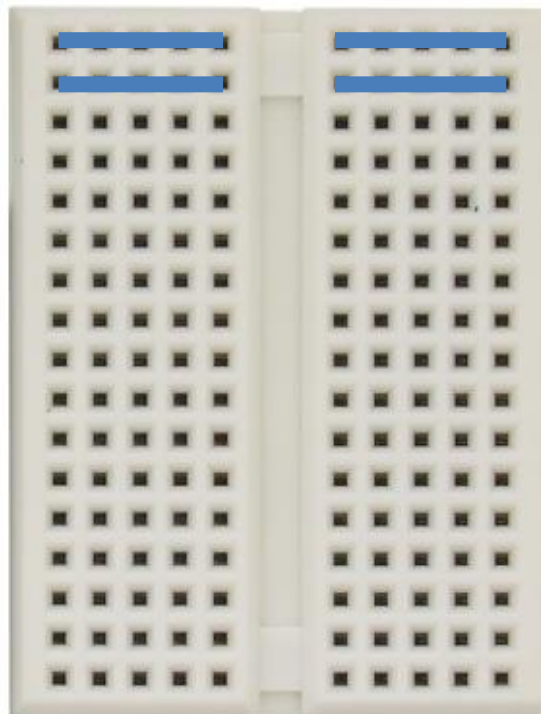
Wires
a dot means connected

Trace the loop.

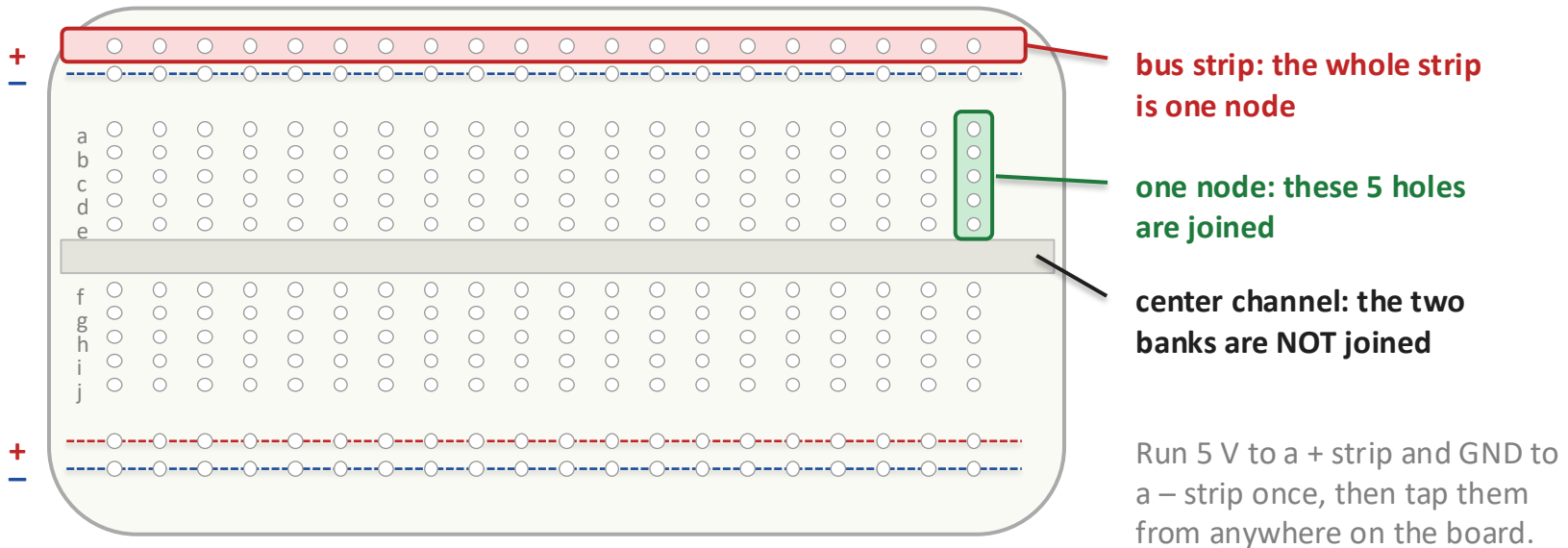
Start at +5 V, follow the wire through every component in the path, and end at GND. If you cannot trace a complete loop, no current flows and nothing happens — no matter what the code says.

Building Circuits

- 5 horizontal holes are connected:



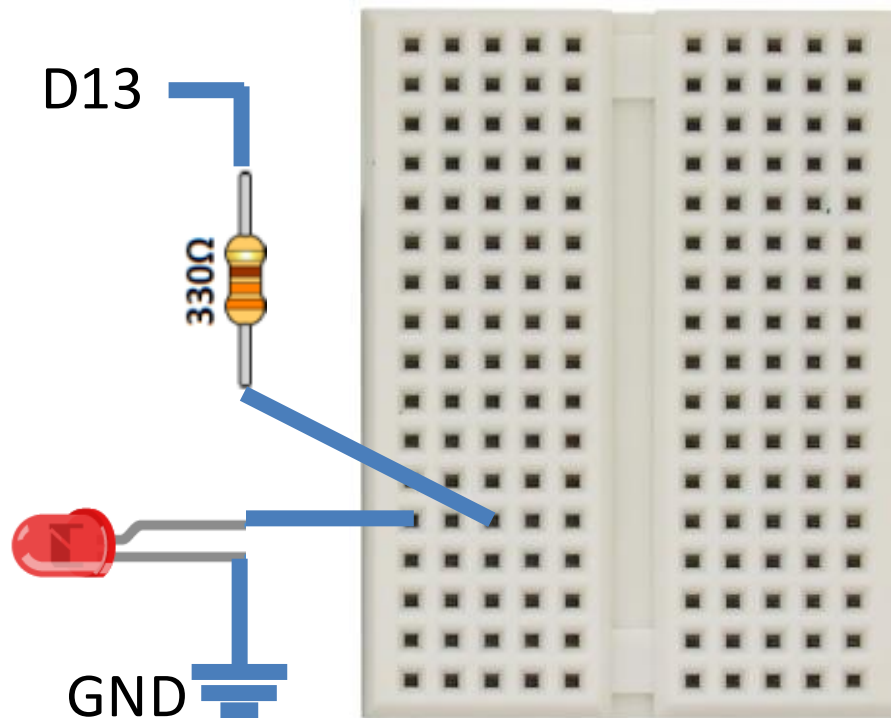
Breadboard Anatomy



- Nothing is soldered — the board is just a grid of pre-wired nodes.
- A group of 5 always runs perpendicular to the center channel, and never crosses it.
- Two component leads in the same group of 5 are connected. Leads in different groups are not.
- The + and – bus strips are separate nodes: top + and bottom + are not joined on most boards.

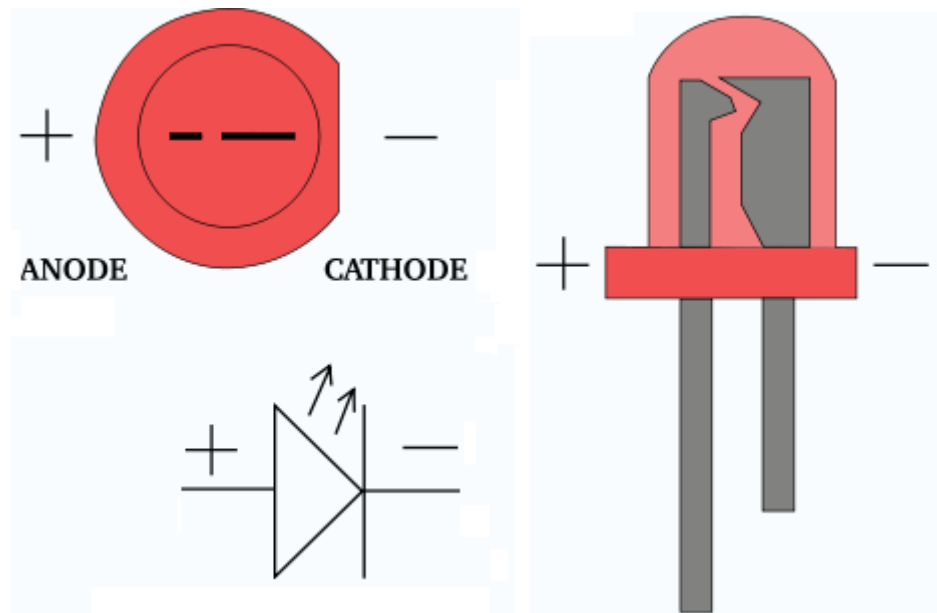
Building Circuits

- Connect components using breadboard:



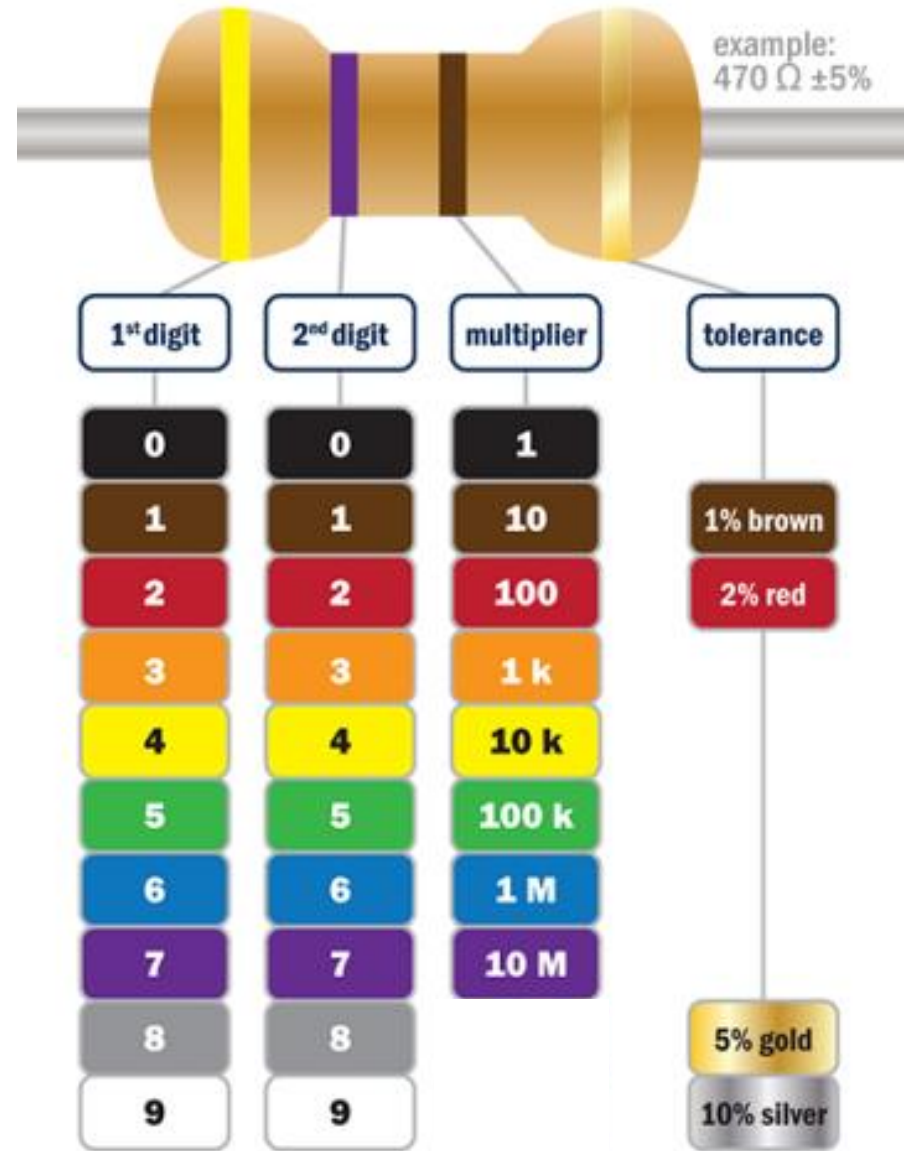
LEDs

- 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 10¹ Ω



Two Ways to Ruin an LED

- 1. Put it in backwards
 - An LED is a diode: current only flows anode → cathode
 - Backwards it simply stays dark — and you will blame your code
 - Long lead = anode = toward the pin. Flat spot on the rim = cathode = toward GND
- 2. Leave out the series resistor
 - An LED is not a resistor. Above its forward voltage (~2 V red) it barely limits current at all
 - 5 V straight through an LED ⇒ hundreds of mA ⇒ dead LED
 - You also blow past the ATmega328's 40 mA per-pin limit — you can damage the pin too

Do the math (Ohm's Law)

$$I = (5\text{ V} - 2\text{ V}) / 330\ \Omega \approx 9\text{ mA}$$

Bright, and comfortably under 40 mA.

Before you apply power

- ☐ Arduino unplugged while you wire
- ☐ One resistor per LED — not one shared
- ☐ Long lead to the pin, flat side to GND

What Inputs Can You Think Of?

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

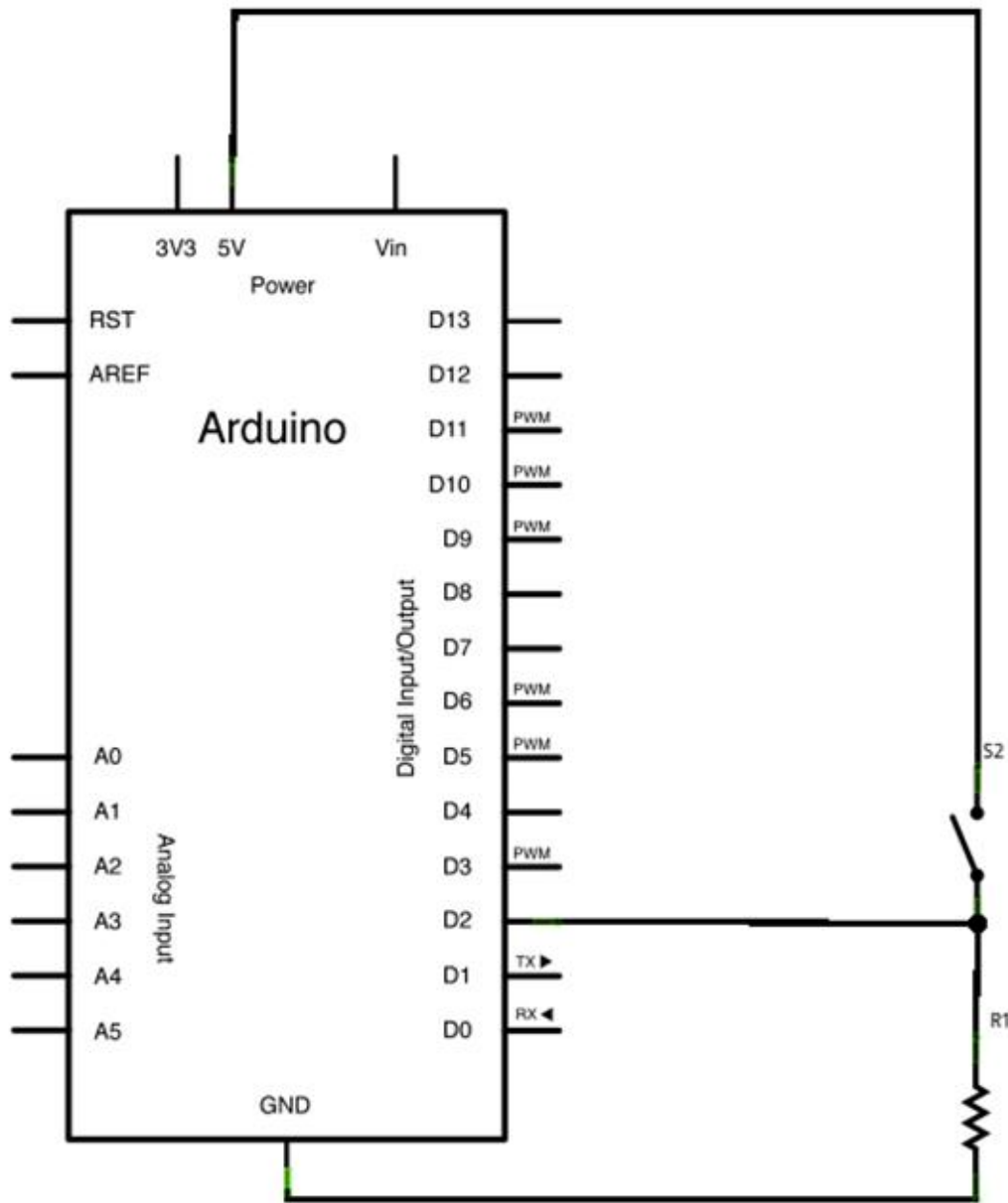
Digital Inputs from Physical World

- 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

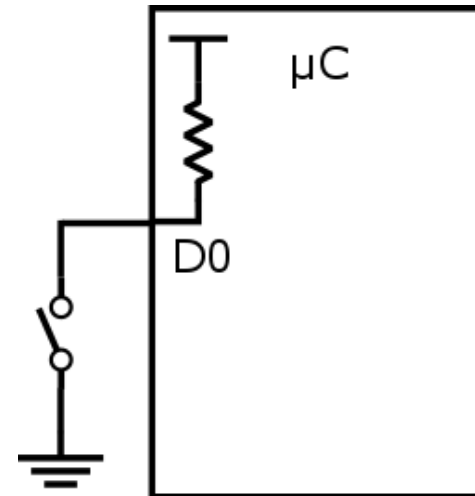
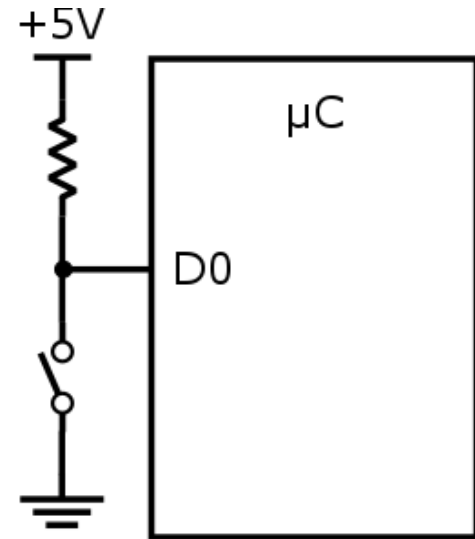
```
setup() {  
    pinMode(pin, INPUT);  
}
```

```
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

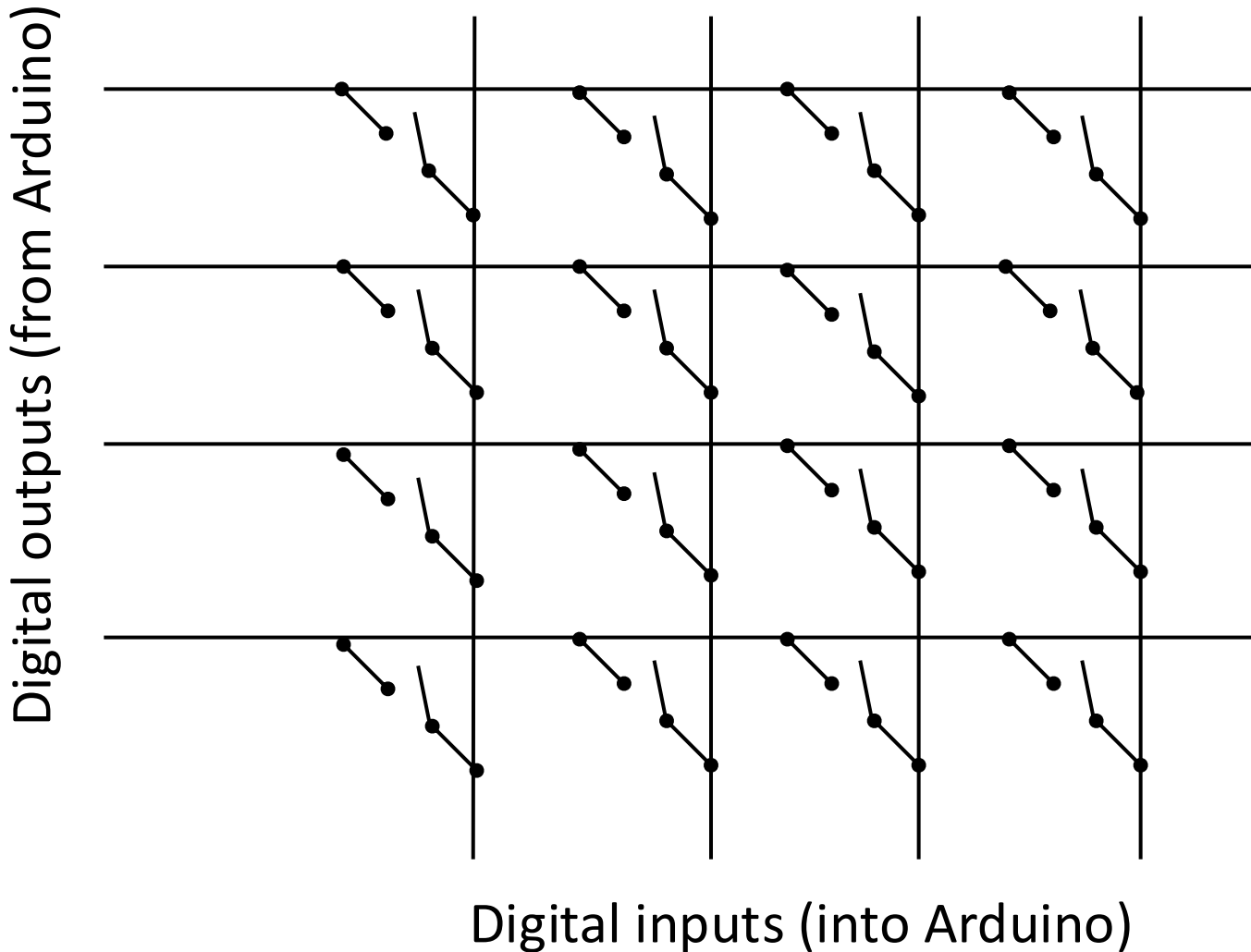
if two switch states agree

 Done

else

 Start over

What about many switches?

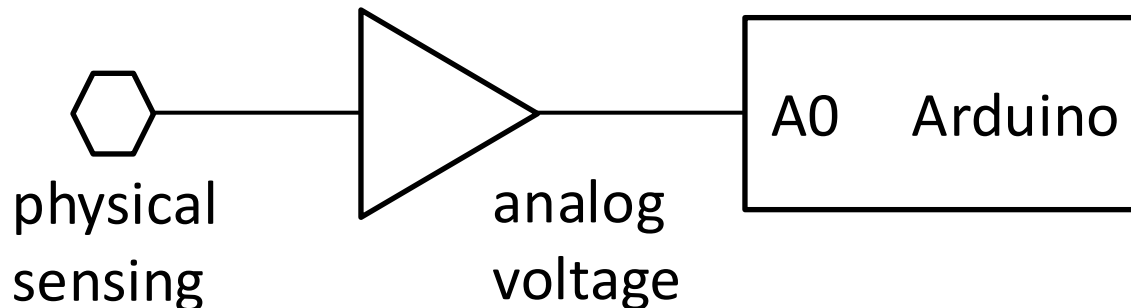


Quiz Time

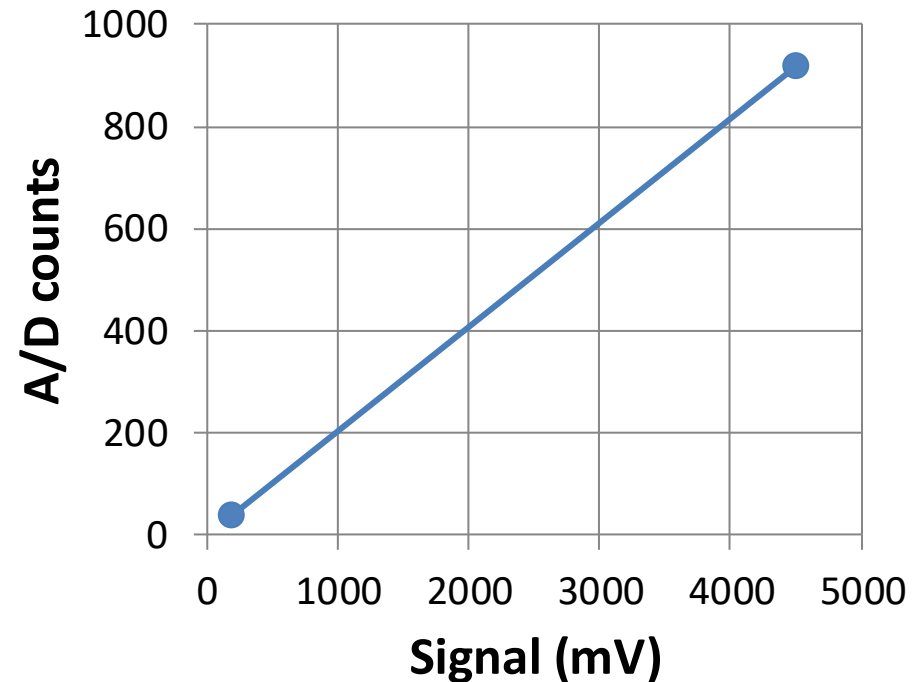
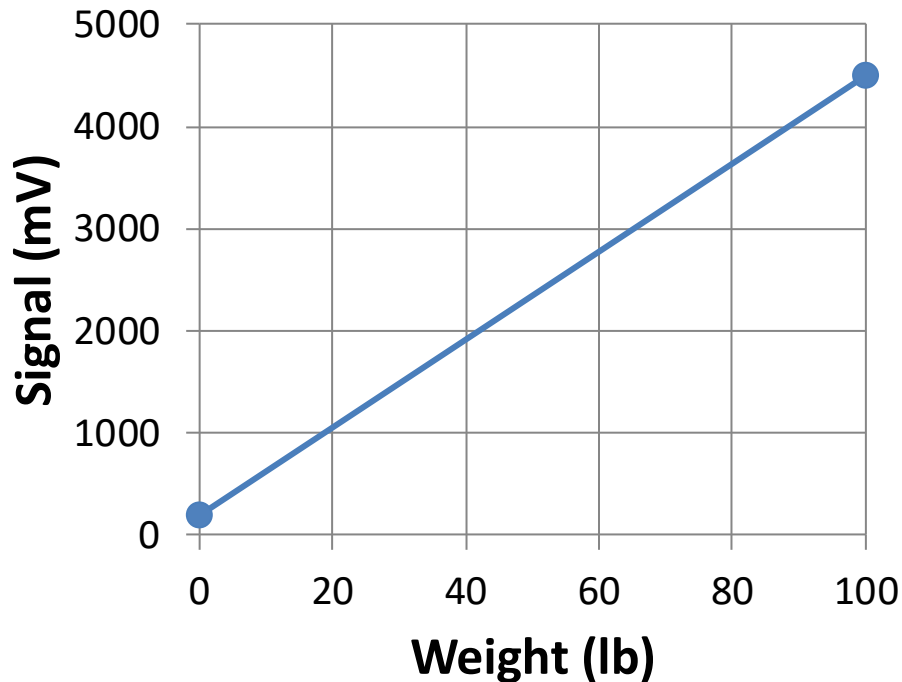
- Go to Canvas and answer the single question for Quiz 2A
- What does LED stand for?
 - Light Emitting Diode
 - Light End Device
 - Long Electricity Digit
 - Last Ever Device

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



$$\text{signal} = m \times \text{weight} + b$$

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

$$\text{counts} = m \times \text{signal} + b$$

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

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

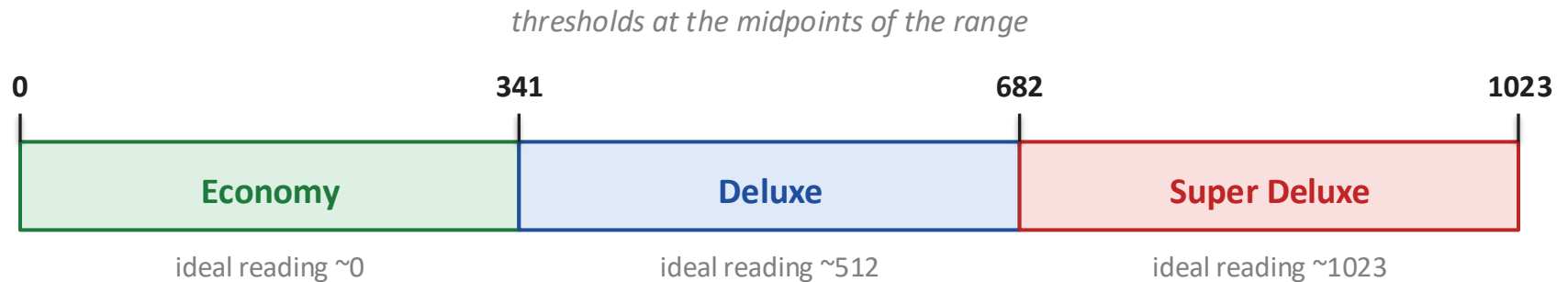
$$\text{weight} = 0.116 \frac{\text{lb}}{\text{cnt}} \times \text{counts} - 4.65$$

Understanding Ranges: The Cycle Knob

The potentiometer is a voltage divider whose wiper sweeps 0 V → 5 V.

`analogRead()` returns 0 ... 1023 (10 bits). Not 0–5. Not 0–255. volts = reading × 5.0 / 1023

A three-position knob does not give you three values — it gives you three neighborhoods.



```
int v = analogRead(A0);

if      (v < 341)  cycle = ECONOMY;
else if (v < 682) cycle = DELUXE;
else                cycle = SUPER_DELUXE;
```

Measure first, then pick numbers.

Write a throwaway sketch that just prints `analogRead()` and turn the knob end to end.

Noise and part tolerance mean your “middle” may not be 512.

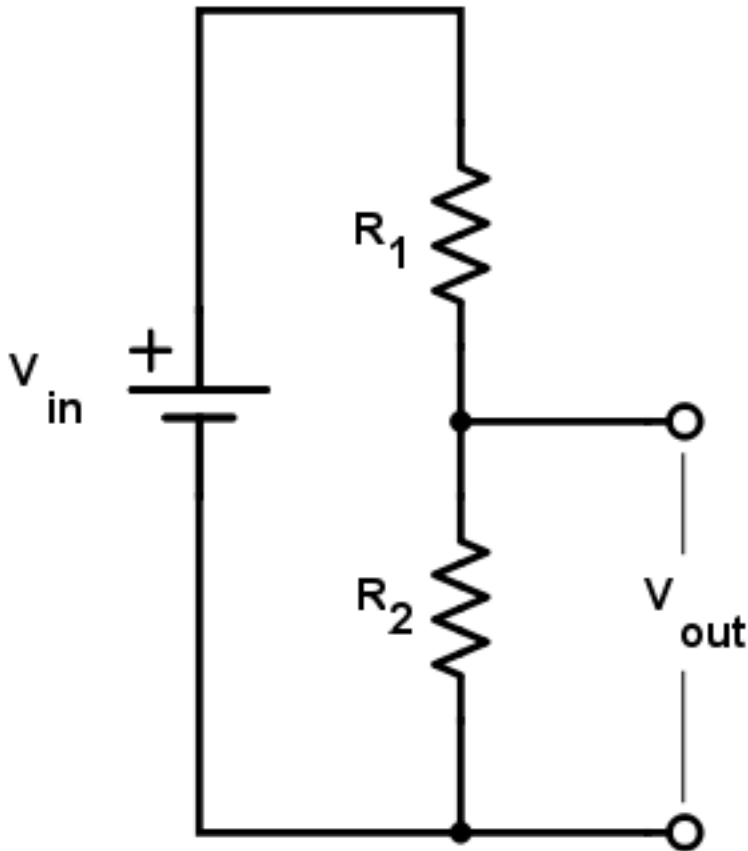
Noisy Analog Signals



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

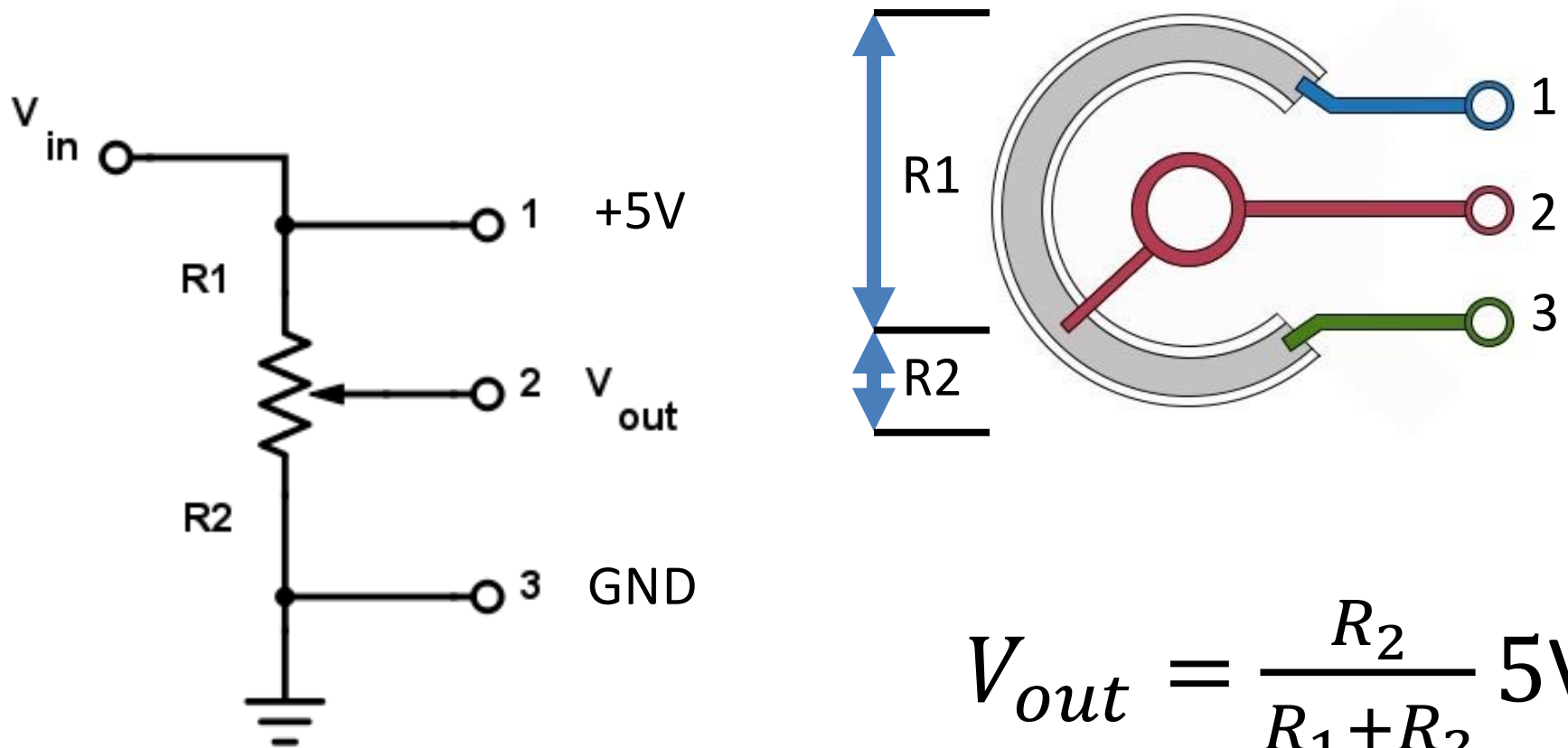
$$avg = \frac{1}{N} \sum_{1}^N A/D \text{ input}_i$$

Simple Voltage Divider



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

Simple Voltage Divider



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

From FSM Diagram to Sketch

In Module 1 you drew state machines. Here is how a drawing becomes code.

- Name the states — use an enum
- One variable holds the current state
- In loop(), switch on that variable
- Each case: set outputs, test conditions, pick the next state

On the drawing

a circle

text in the circle

an arrow

the label on the arrow

the start arrow

In the sketch

→ case LABEL:

→ the outputs

→ if (...) state = ...

→ the if condition

→ state's first value

```
enum State { IDLE, RUNNING, DONE };
State state = IDLE;          // start arrow

void loop() {
  switch (state) {
    case IDLE:
      setOutputs(LOW, LOW);
      // INPUT_PULLUP: pressed == LOW
      if (digitalRead(BUTTON) == LOW)
        state = RUNNING;
      break;

    case RUNNING:
      setOutputs(HIGH, LOW);
      delay(5000);           // lasts 5 s
      state = DONE;
      break;

    case DONE:
      setOutputs(LOW, HIGH);
      delay(2000);
      state = IDLE;
      break;
  }
}
```

This is the shape, not the answer — the Washer needs its own states.
delay() is fine for now; Module 3 shows why blocking delays break down.

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):

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

- 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

- $Q_n.m$ means a number with $n+m$ bits (digits), n integer and m fractional. Sign bit is often in addition to this.
- E.g., $Q_{3.4}$ for 0100.1100, with value 4.75
- Q_m means a number with $m+1$ bits, m are fractional
- E.g., Q_3 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: $x \cdot 10^y$ $0.32 \times 10^{-3} = 0.00032$

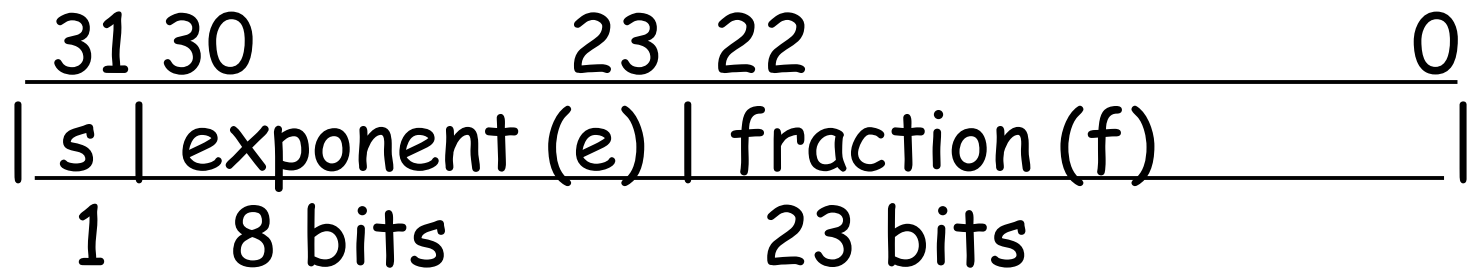
In base 2: $x \cdot 2^y$ called floating point

↑ ↑
| |
| | exponent
| |
| | mantissa

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

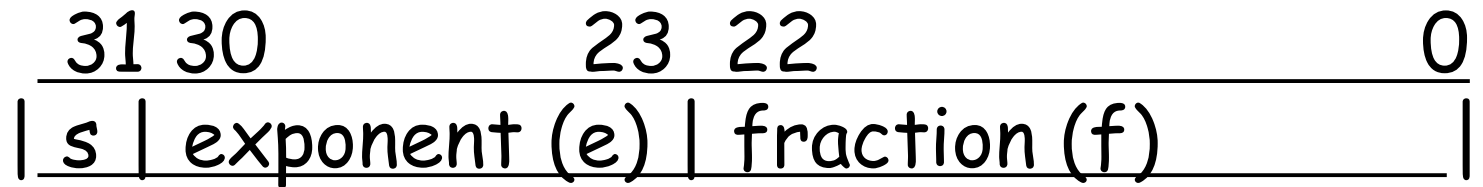
Single precision (32 bits)



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

↑ hidden "1"

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



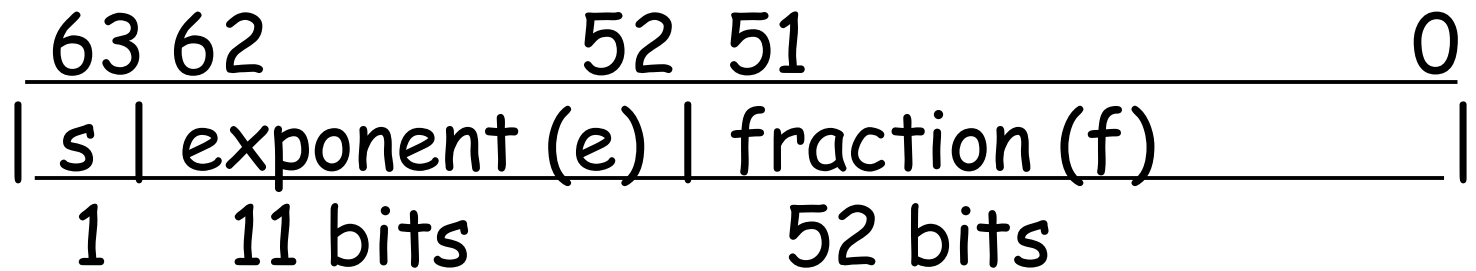
- $s = 0, e = 0, f = 0 \Rightarrow \text{value} = \text{zero}$
- $e = 255, f = 0 \Rightarrow \text{value} = (-1)^s \times \text{infinity}$
- $e = 255, f \neq 0 \Rightarrow \text{value} = \text{"not a number"}$ triggers exception
- $e = 0, f \neq 0 \Rightarrow \text{denormalized}$

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

$\uparrow$ hidden "0"

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

Double precision (64 bits)



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

↑ hidden "1"

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

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

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

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:

0x002400081881423c

0000 0000 0010 0100 0000 0000 0000 1000

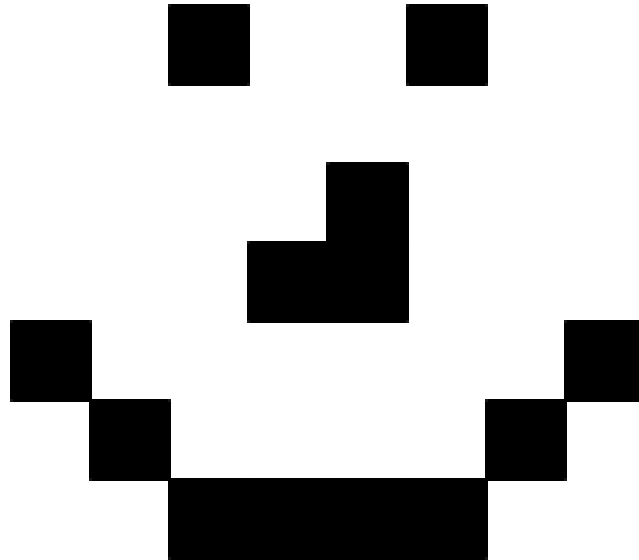
0001 1000 1000 0001 0100 0010 0011 1100

- Make 1 dark and 0 light:



Images

- Arrange in rows, one byte per row:



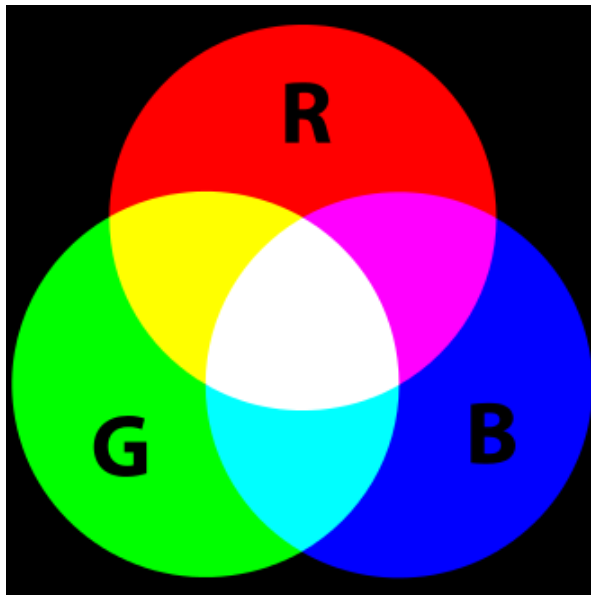
- Each bit is a “pixel” in the image

Add color and more pixels



Color

- Additive color – primaries Red, Green, Blue



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