

Final Projects

Start this week!

Wednesday lab will be on projects

Arrange with your mentor how you will meet (at your regular lab, their lab, some other time)

We will do demos during our final class.

Sensors

Apple iPhone 7 teardown



How many sensors?



Apple iPhone 7

How many sensors? At least 10

Dual 12MP wide-angle and telephoto cameras

Facetime camera

Infrared camera (for face recognition)

Microphones (2 at top, 2 at bottom)

Proximity sensor

Ambient light sensor

Accelerometer

Gyroscope

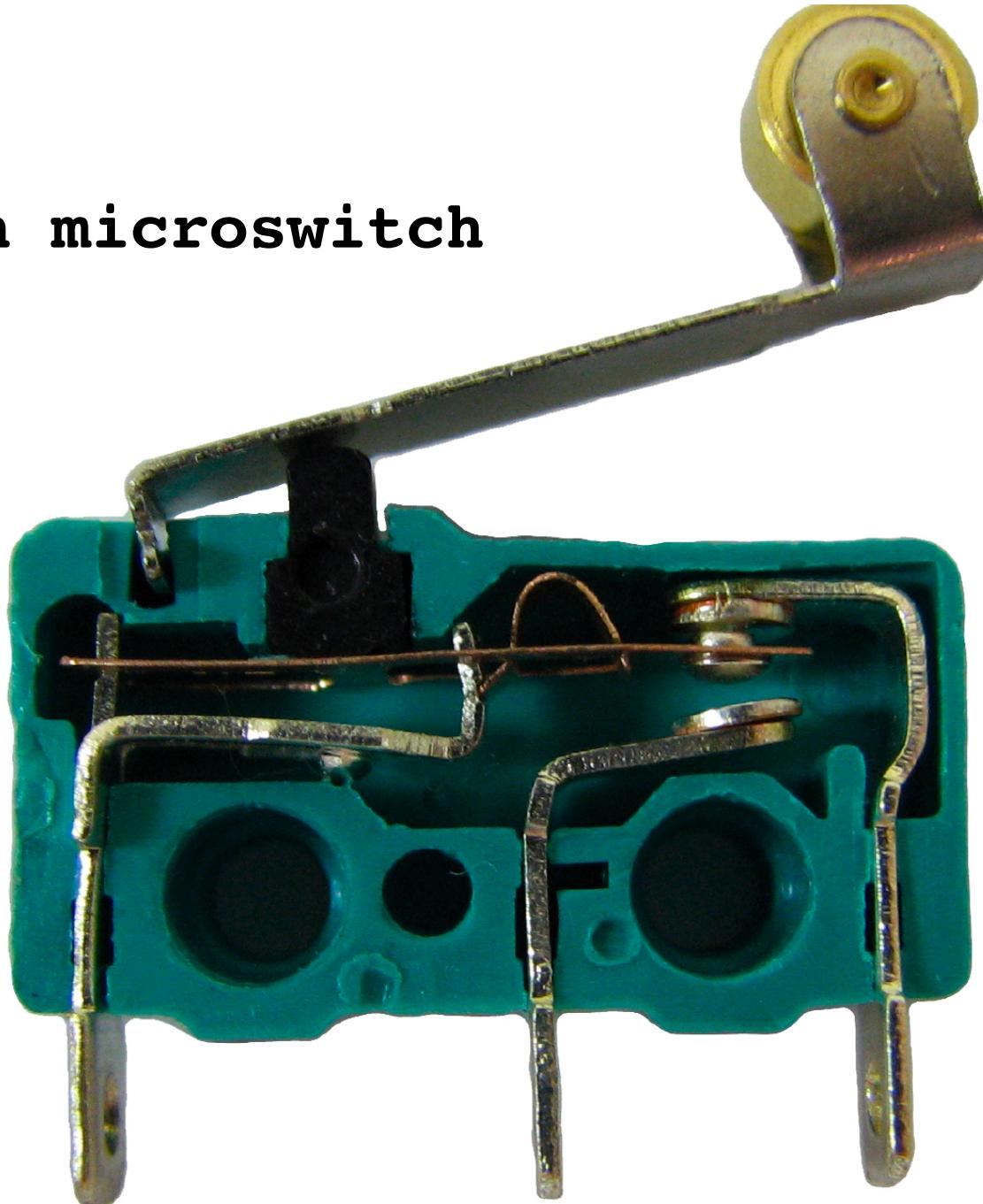
Compass (magnetometer)

Barometer (altimeter)

Touch ID fingerprint scanner

Pressure sensitive 3D multi-touch display

Snap-action microswitch

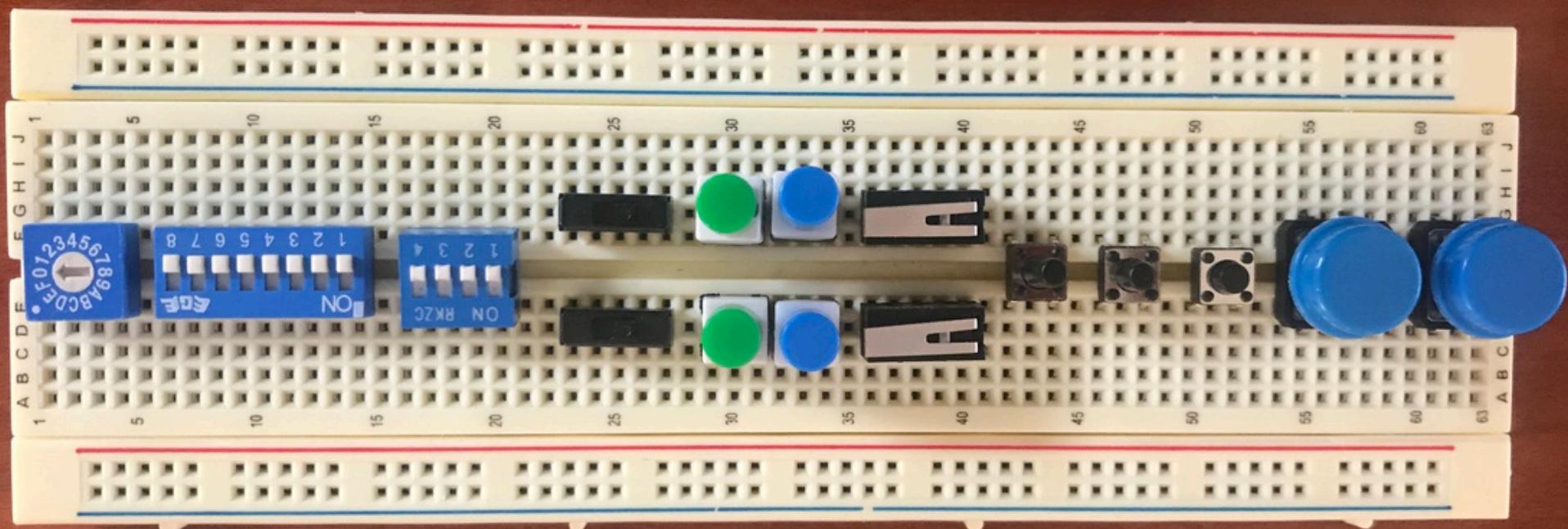


Common

NO

NC

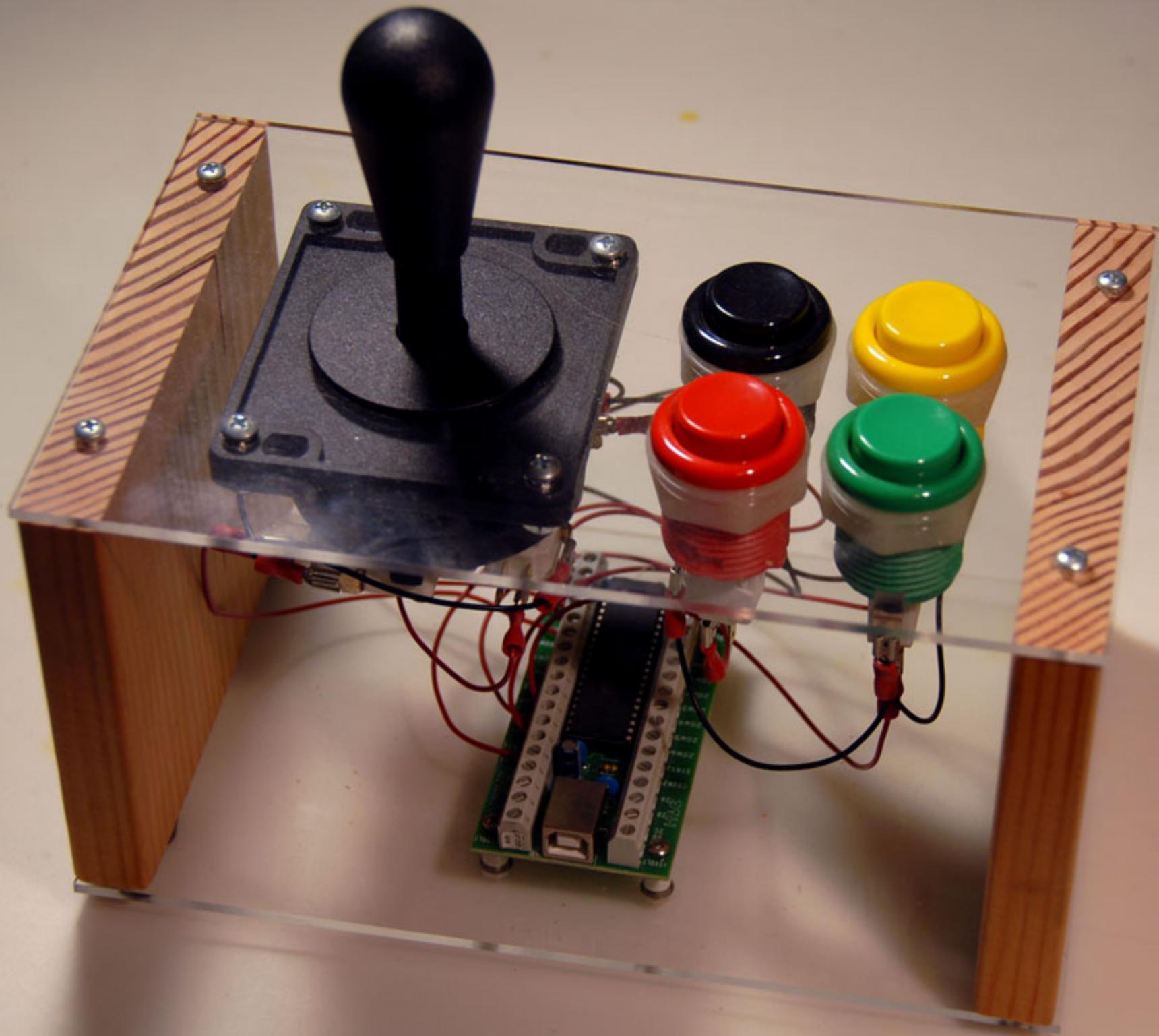
Buttons and Switches



Happ Pushbutton



Happ Joystick

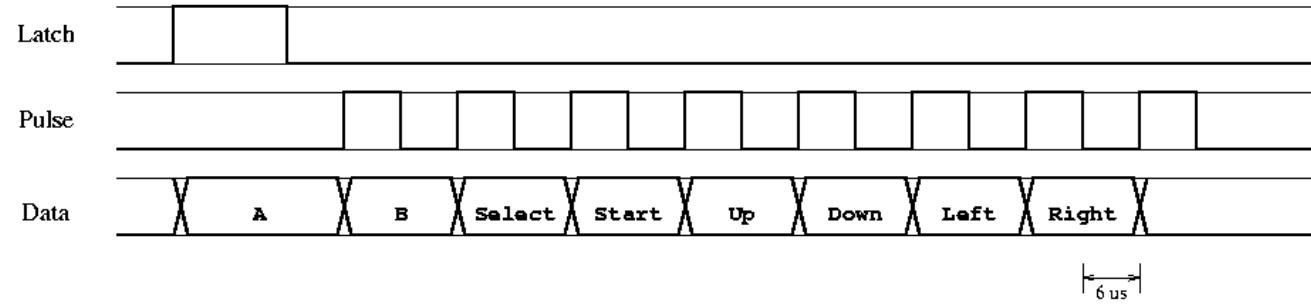
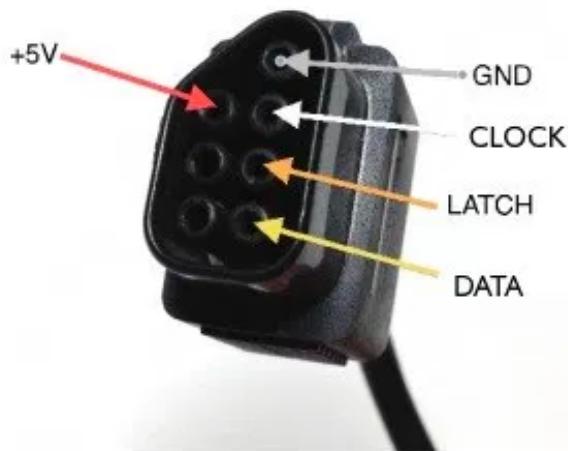




NES D-pad



Famicon D-pad



Analog to Digital (ADC)

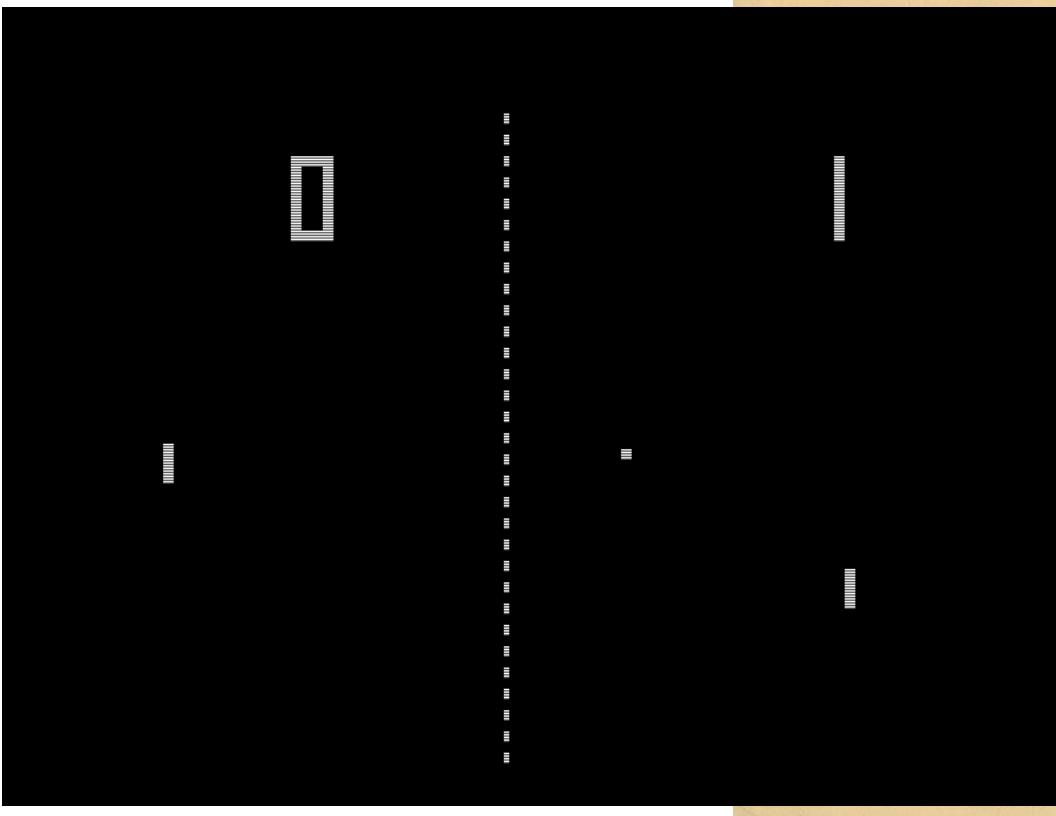


THE NEWEST 2 PLAYER
VIDEO SKILL GAME

PONG

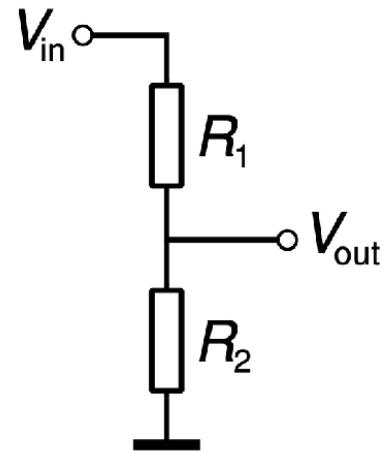
from ATARI CORPORATION
SYZYGY ENGINEERED

The Team That Pioneered Video Technology

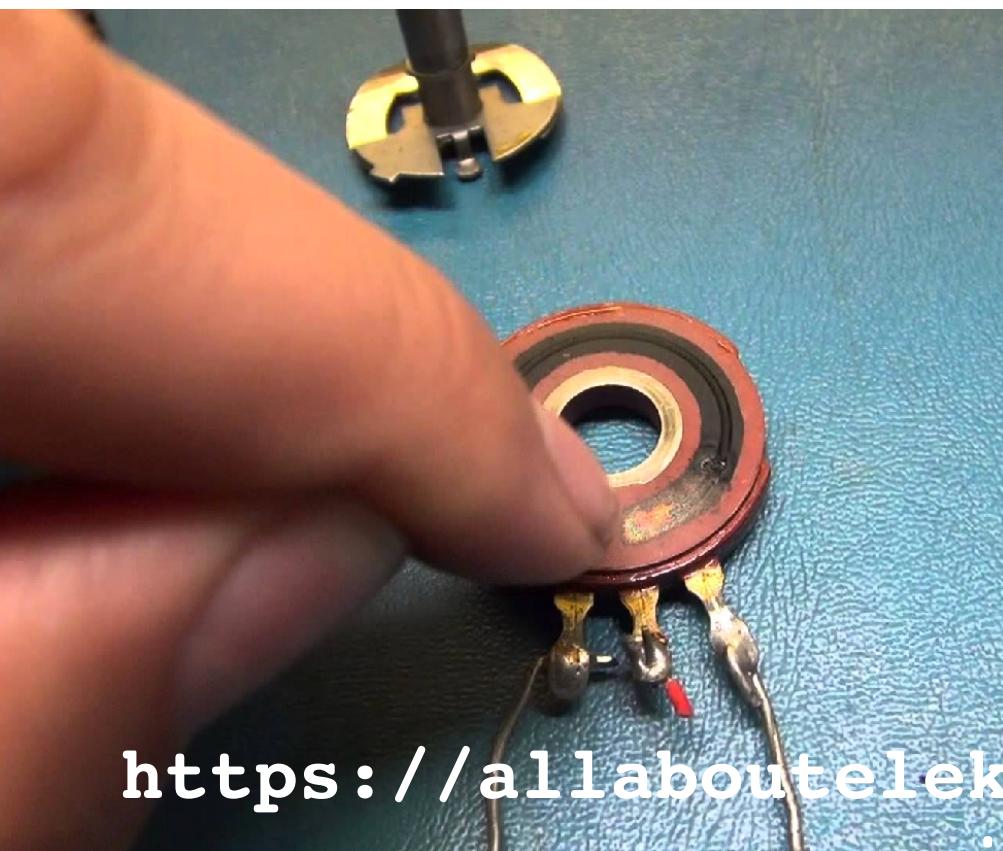
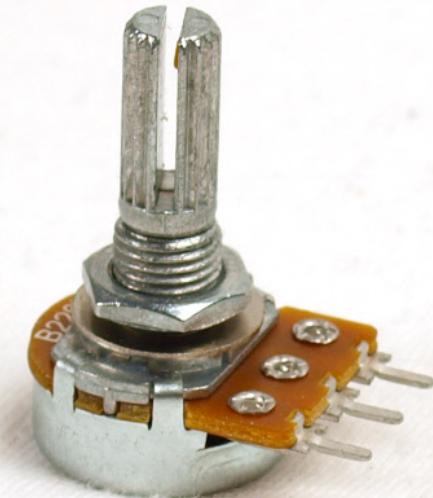


Maximum Dimensions:
WIDTH - 26"
HEIGHT - 50"
DEPTH - 24"
SHIPPING WEIGHT:
150 Lb.





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



<https://allaboutelectronics.wordpress.com/>

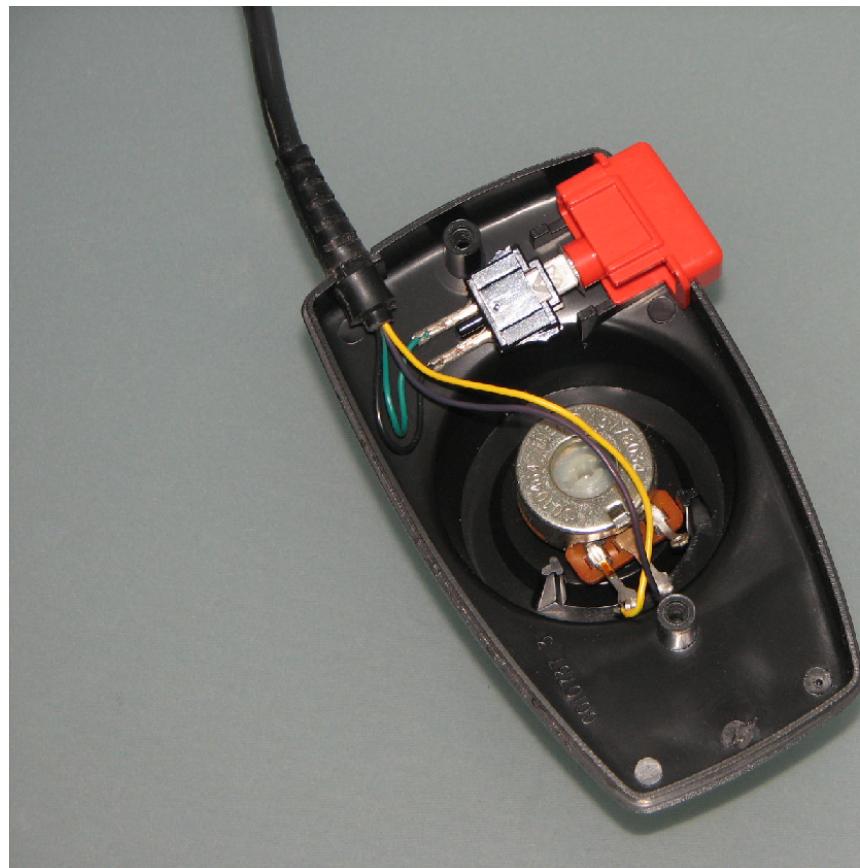
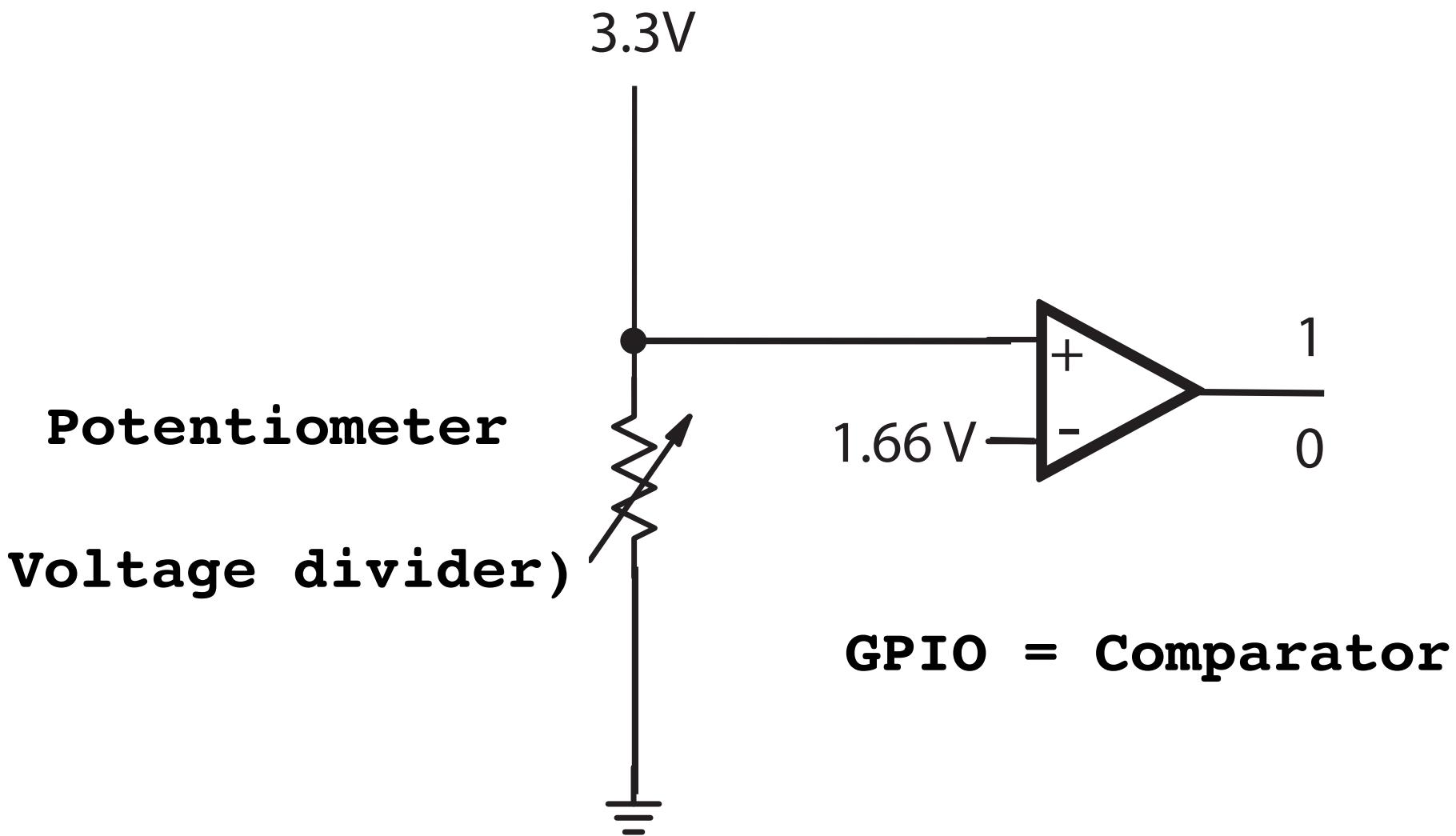


Image © Avon Fox
www.the-liberace.net
Image may be used and/or reproduced
with this watermark intact.

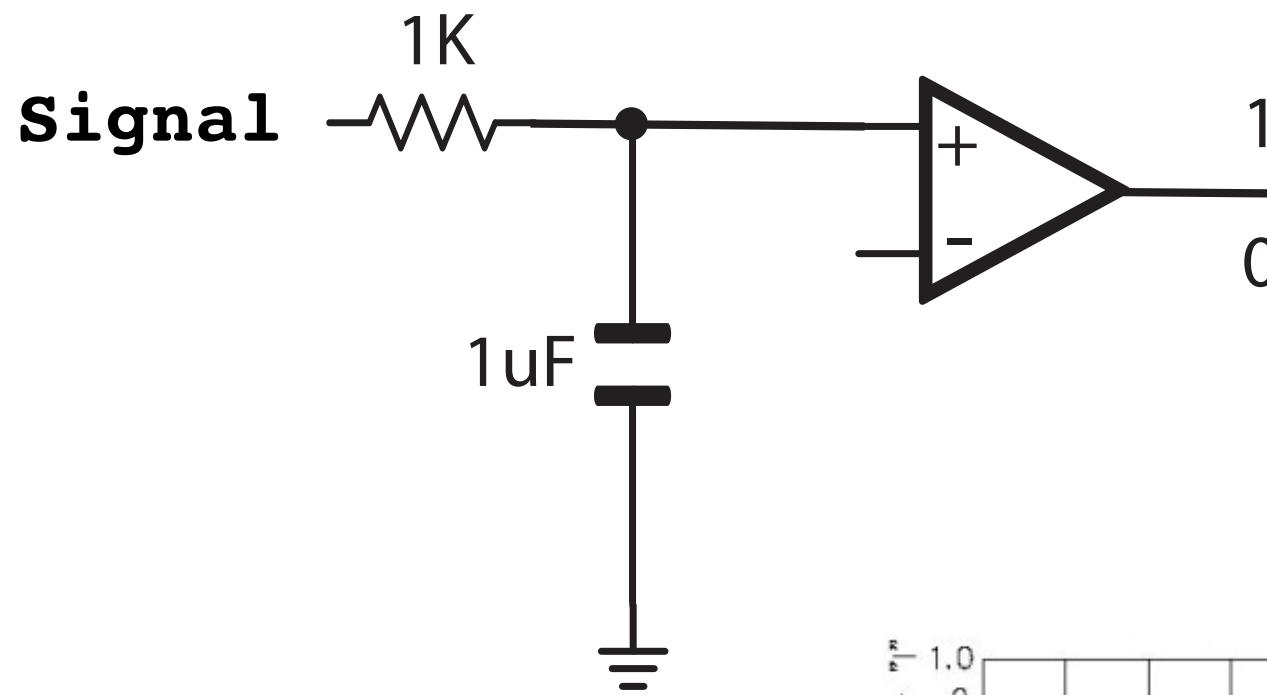
Atari 2600 Paddle

How would you measure the voltage?

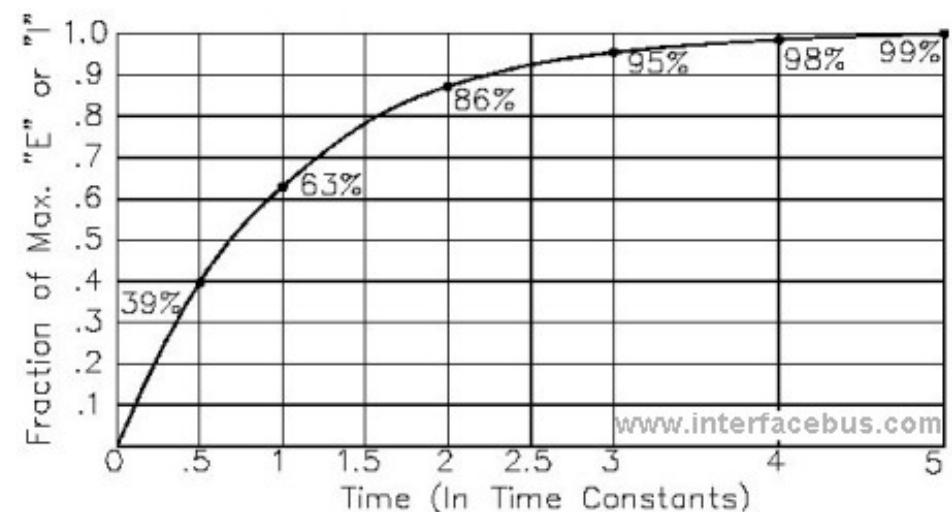


Charging Circuit

The time to fire depends on the input signal voltage

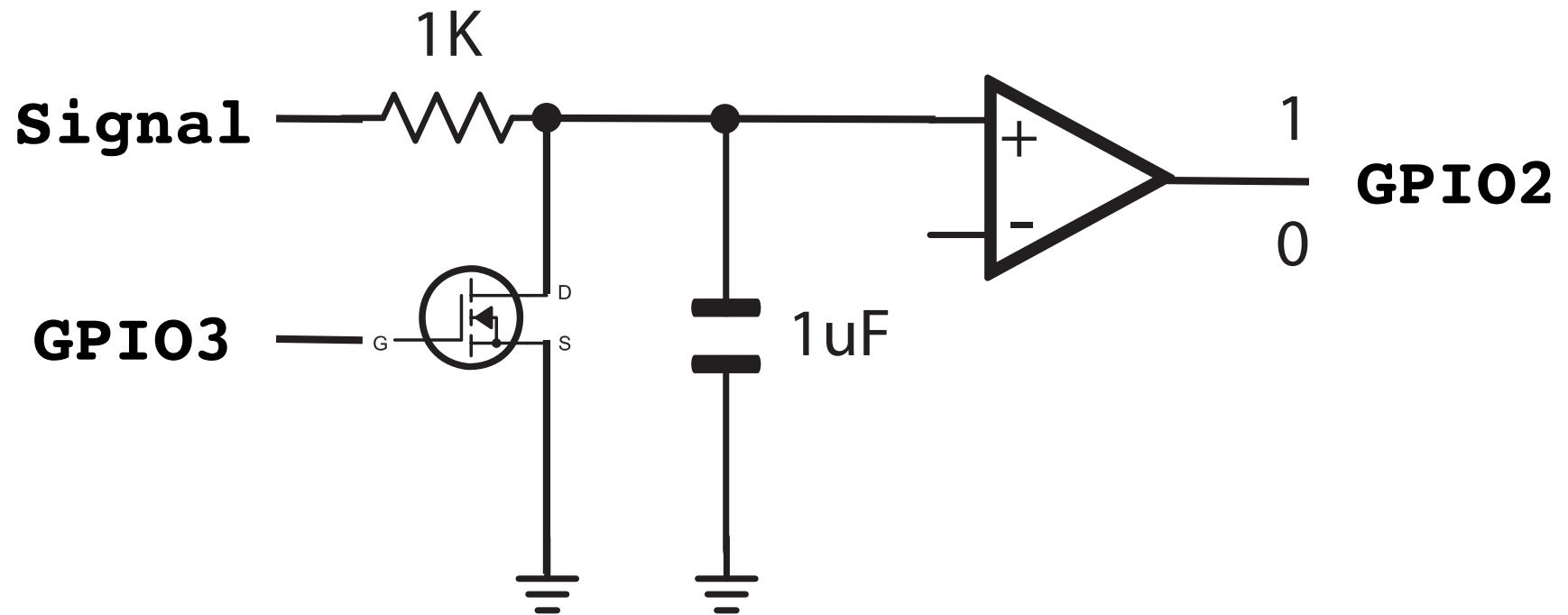


$$RC = 1000 \text{ usecs}$$



ADC

Time-to-charge then discharge, ... , ... , ...

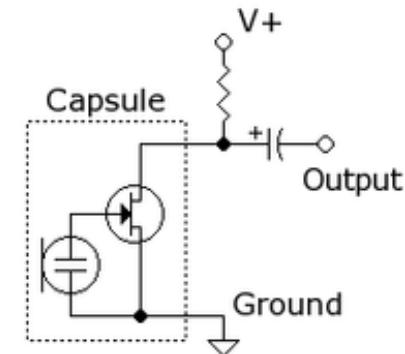


$$RC = 1000 \text{ usecs}$$

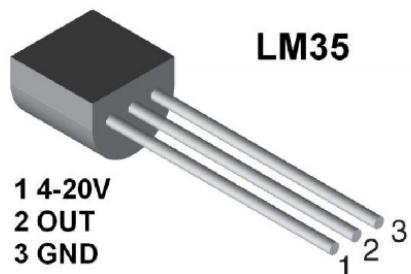
Analog Sensors



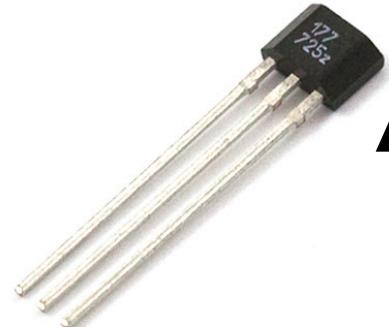
**Phototransistor
(light)**



**Electret Microphone
(pressure)**



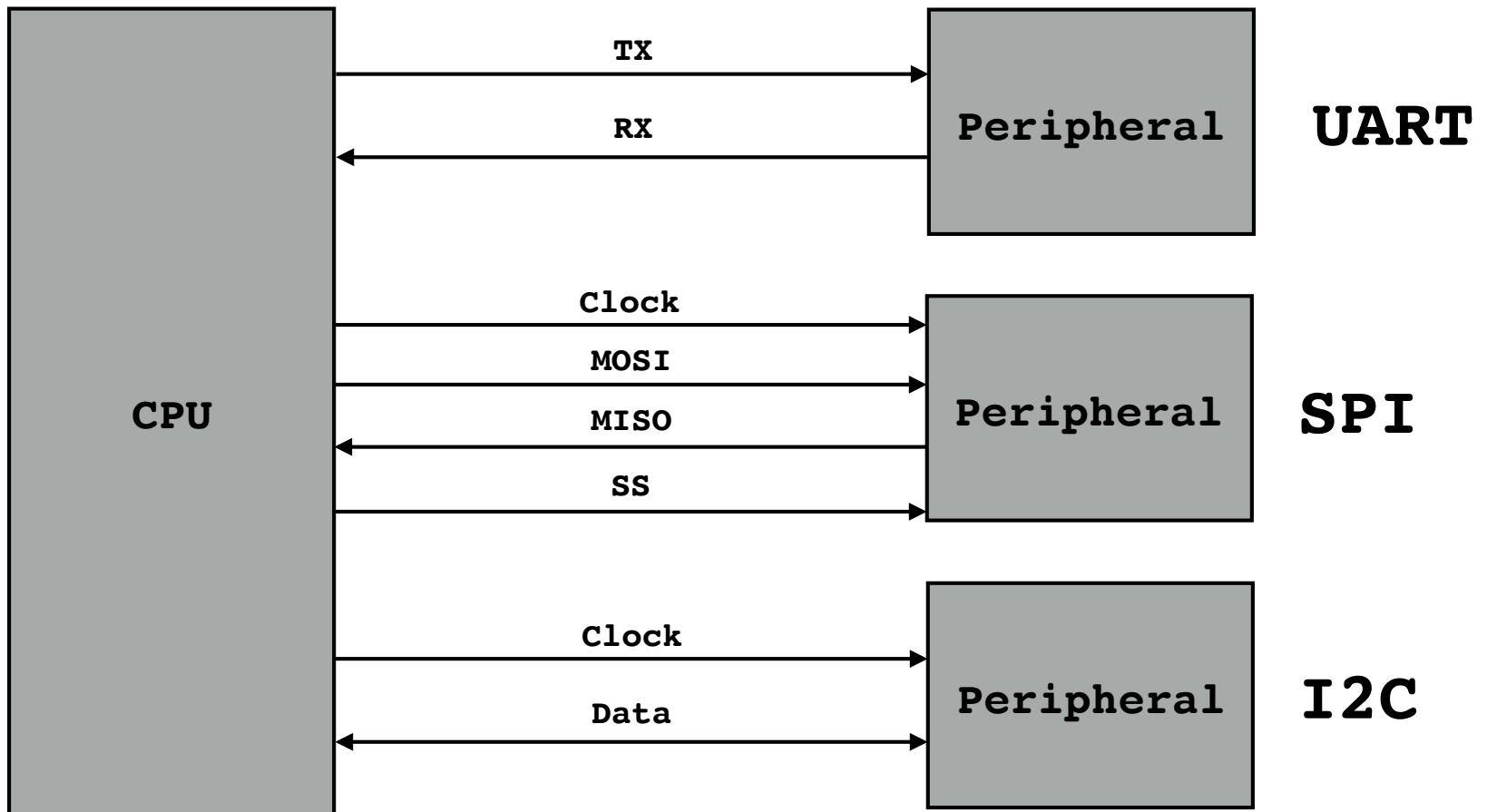
(temperature)



**Analog Hall Effect
(magnetic field)**

Digital Sensors

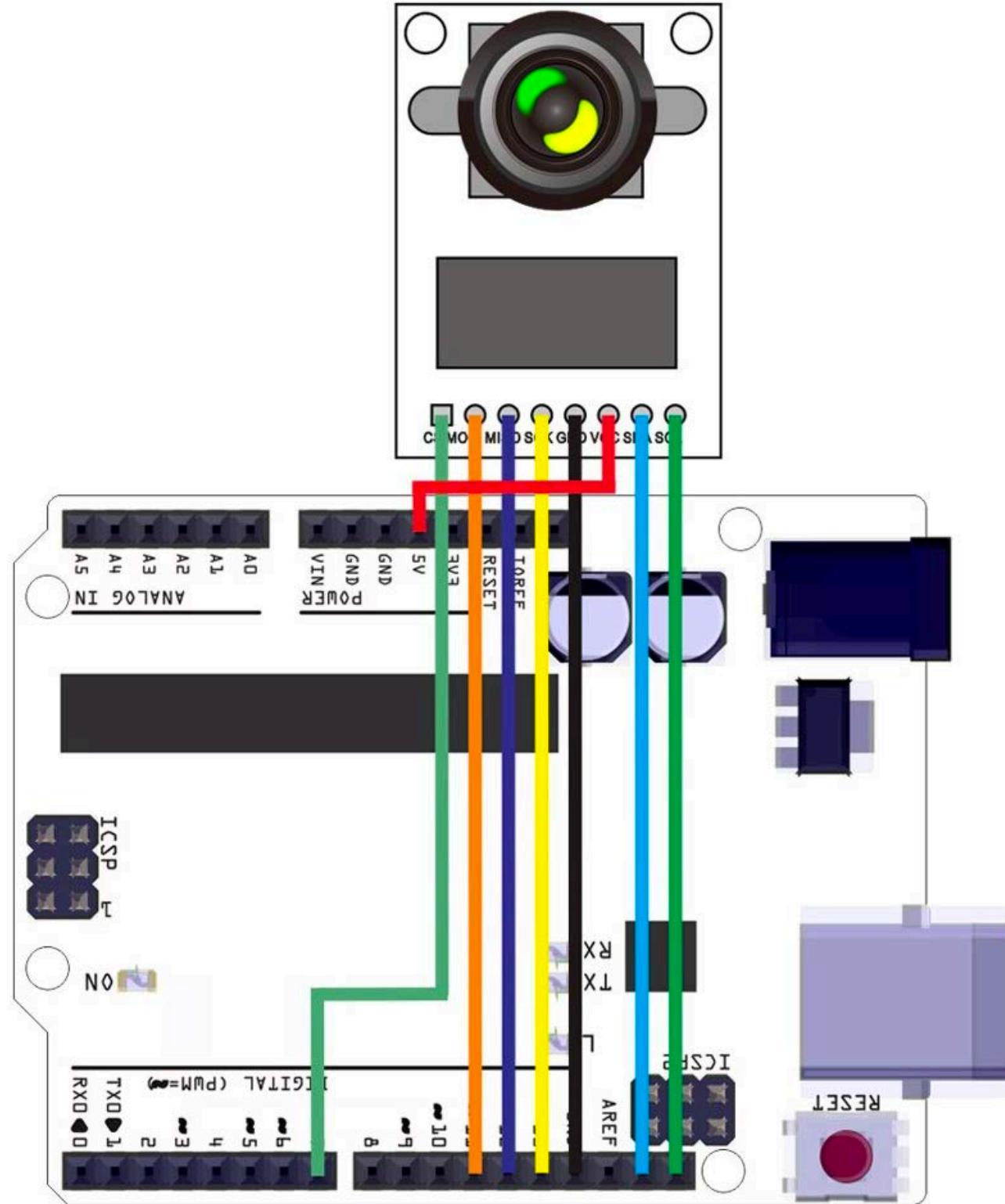
Bus Protocols





Arducam

SPI + I2C



Sensing the World

Resistance (conduction, capacitance)-

Convert energy to voltage/current

- Light (phototransistor)
- Sound/pressure/deformation (piezo, electret, strain gauge)
- Temperature (heat), humidity, pressure
- Electromagnetic fields (hall effect, compass, antenna)

Smart sensors (sensor with a digital interface)

- Acceleration/Orientation/Magnetic (force direction)
- Camera, IMU (inertial management unit), ...

Computer Arithmetic

What is the difference between
signed int and unsigned int?

Addition

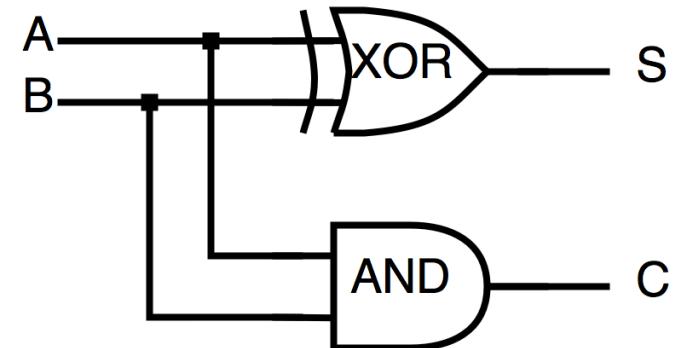
Adding 2 1-bit numbers: sum = a + b

a	b	sum
0	0	00
0	1	01
1	0	01
1	1	10

Adding 2 1-bit numbers (Half Adder)

a	b	sum
0	0	00
0	1	01
1	0	01
1	1	10

bit 0 of sum: $S = a \oplus b$
bit 1 of sum: $C = a \& b$



Have reduced addition to logical operations!

Adding 2 8-bit numbers

Carry

$$\begin{array}{r} 00000111 \text{ A} \\ +00001011 \text{ B} \\ \hline \end{array}$$

Sum

Adding 2 8-bit numbers

1 Carry

$$\begin{array}{r} 00000111 \text{ A} \\ +00001011 \text{ B} \\ \hline \end{array}$$

0 Sum

Adding 2 8-bit numbers

11 Carry	
00000111	A
+00001011	B

10 Sum	

Adding 2 8-bit numbers

00001111 Carry

00000111 A

+00001011 B

00010010 Sum

Adding 3 1-bit numbers

a b c = c s

0 0 0 0 0

0 1 0 0 1

1 0 0 0 1

1 1 0 1 0

0 0 1 0 1

0 1 1 1 0

1 0 1 1 0

1 1 1 1 1

Adding 3 1-bit numbers (Full Adder)

a b ci = co s

0 0 0 0 0

0 1 0 0 1

1 0 0 0 1

1 1 0 1 0

0 0 1 0 1

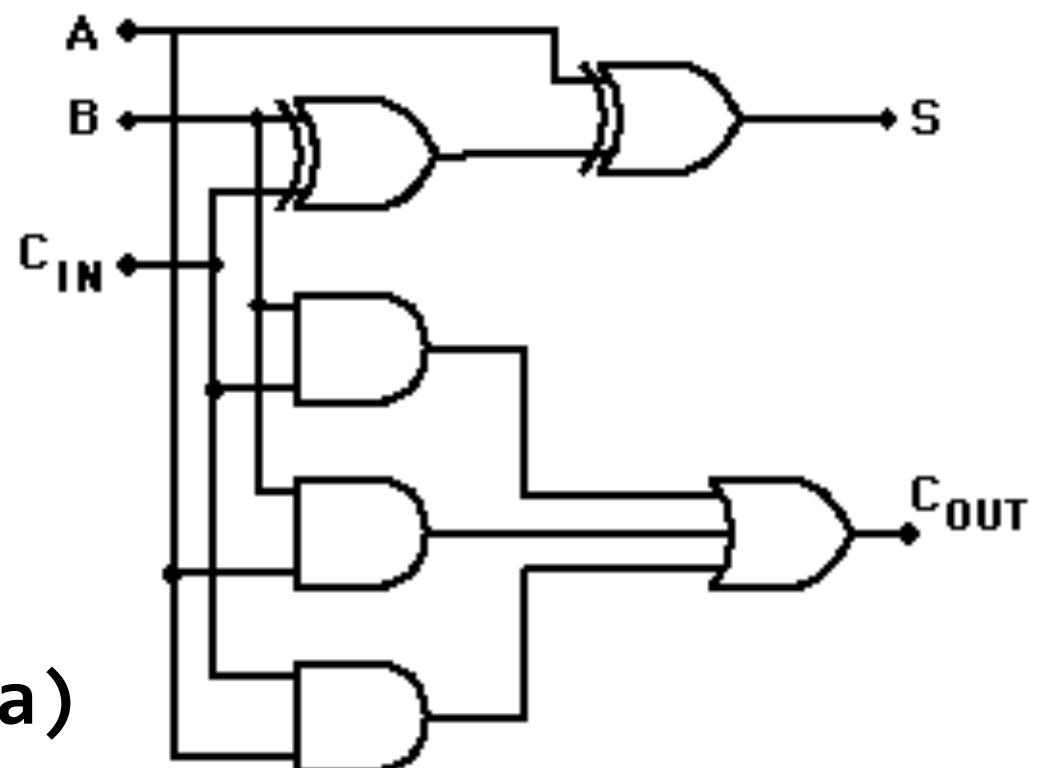
0 1 1 1 0

1 0 1 1 0

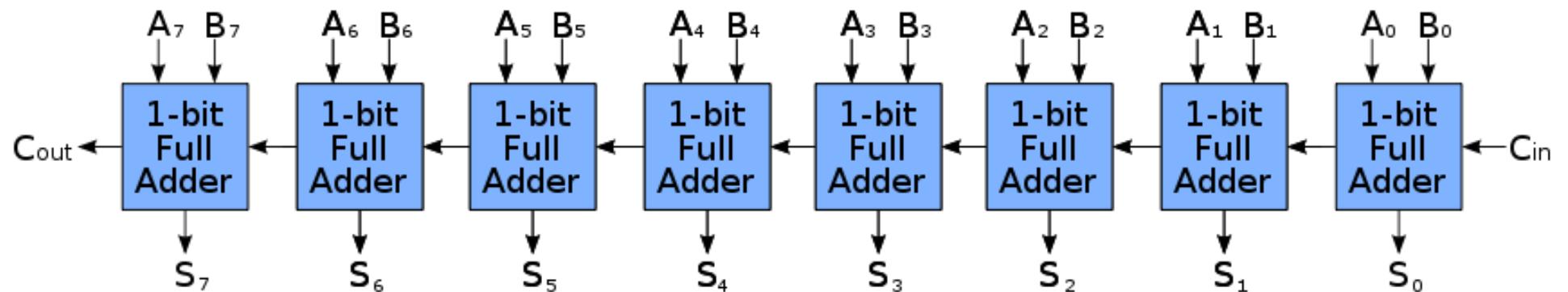
1 1 1 1 1

$$s = a \wedge b \wedge c_i$$

$$co = (a \& b) \mid (b \& c) \mid (c \& a)$$



8-bit Ripple Adder



Note Cin (carry in) and Cout (carry out)

```
// Multiple precision addition
// https://gcc.godbolt.org/z/6TRmY8
```

```
uint64_t add64(uint64_t a, uint64_t b)
{
    return a + b;
}
```

```
add64:
    adds r0, r0, r2
    adc  r1, r1, r3
    bx   lr
```

Binary Addition - Modular Arithmetic

11111111 Carry

11111111 A

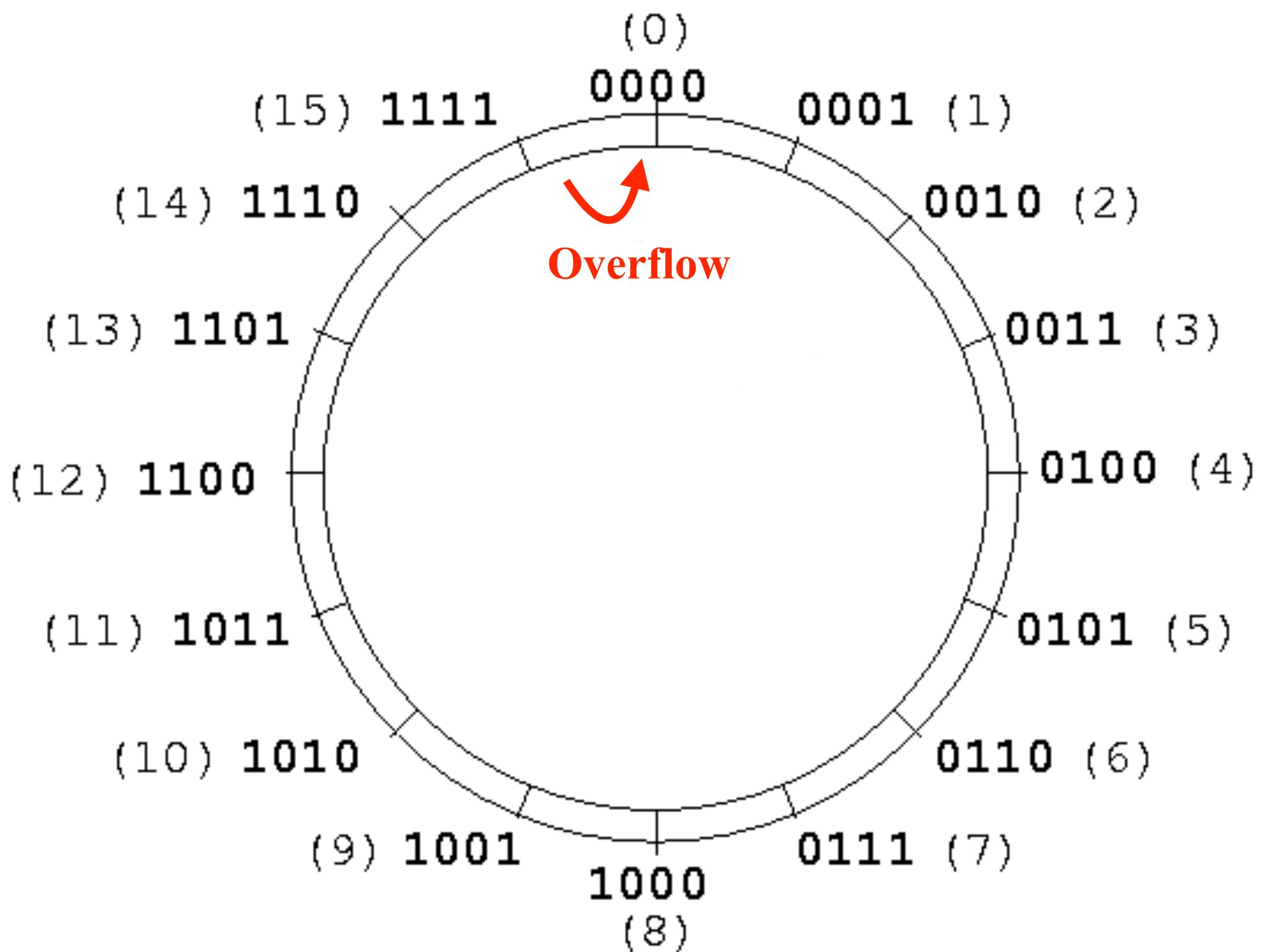
+00000001 B

10000000 Sum

To represent the result of adding two n-bit numbers to full precision requires n+1 bits

But we only have 8-bits!

sum = (A+B)%256 = 0b00000000



Gangnam Style overflows INT_MAX, forces YouTube to go 64-bit

Psy's hit song has been watched an awful lot of times.

PETER BRIGHT - 12/3/2014, 2:32 PM



- In two's complement, when you exceed the maximum value of int (2,147,483,647), you “wrap around” to negative numbers:

PSY - GANGNAM STYLE (강남스타일) M/V

officialpsy

Subscribe 7,600,830

-2142584554

Add to Share More

8,761,309 1,139,933

- Here is the link after Google upgraded to 64-bit integers:

PSY - GANGNAM STYLE (강남스타일) M/V

officialpsy

Subscribe 7,600,830

2,152,382,740

Add to Share More

8,761,309 1,139,933

Subtraction

BIG IDEA: Define subtraction using addition

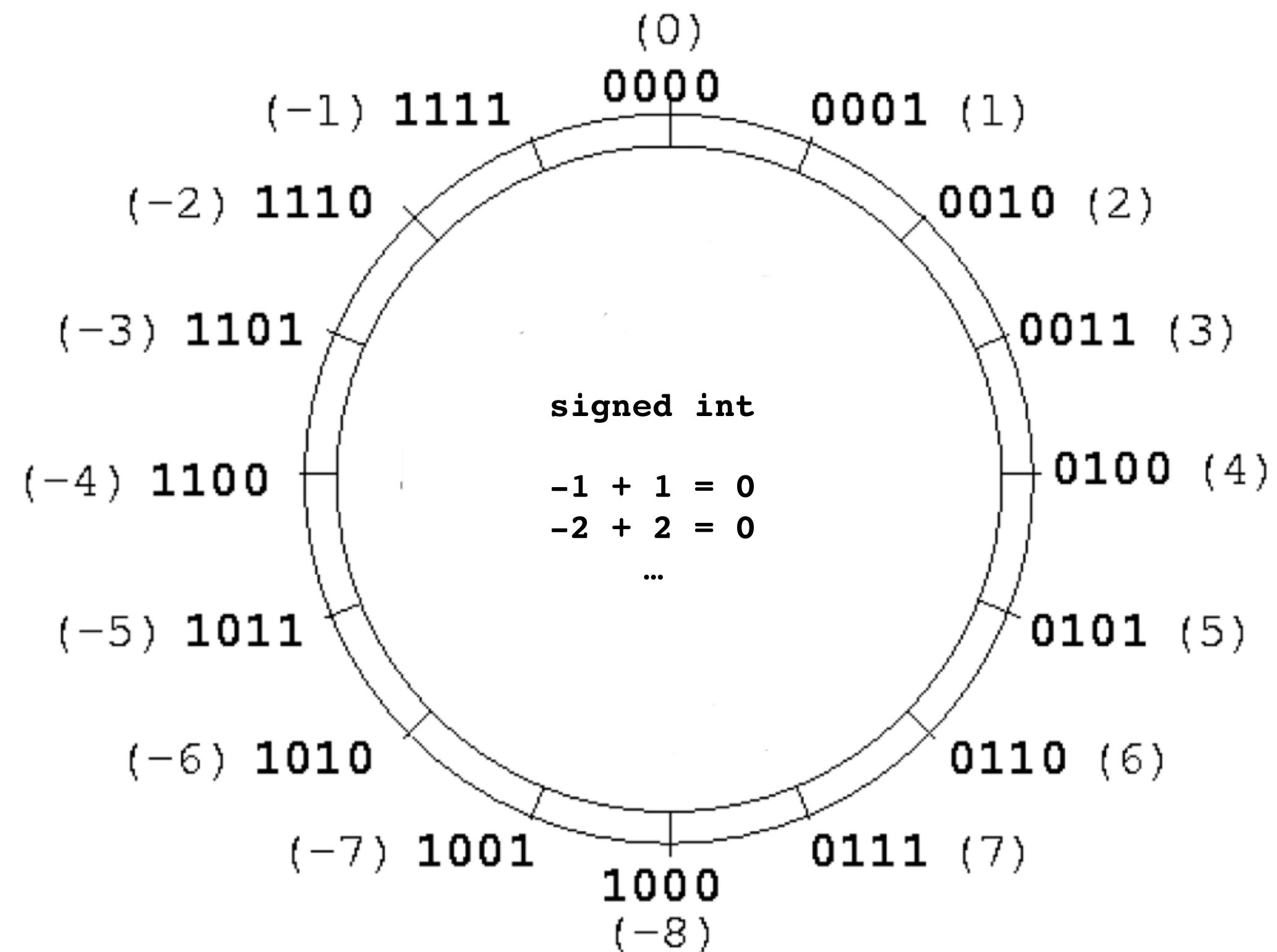
A clever way of defining subtraction by 1 is to find a number to add that yields the same result as the subtract by 1.

This number is the *negative* of the number.

More precisely, this number is the number that when added to 1, results in $0 \pmod{16}$

$$0x1 - 0x1 = 0x1 + 0xf = 0x10 \% 16 = 0x0$$

$0xf$ can be *interpreted* as -1



Signed 4-bit numbers,

0x0 = 0

0xf = -1

0xe = -2

...

0x8 = -8 (could be interpreted as 8)

0x7 = 7

...

0x1 = 1

0x0 = 0

if we choose to *interpret* 0x8 as -8,
then the most-significant bit of the
number indicates that it is negative (n)

signed int vs as unsigned int

**Are just *different interpretations* of
the bits comprising the number**

0xff vs -1

Negation

How do we negate an 8-bit number?

Find a number $-x$, s.t. $(x + (-x)) \% 256 = 0$

Subtract it from 256 = $2^8 = 100000000$

$$-x = 100000000 - x$$

Since then $(x + (-x)) \% 256 = 0$

11111111	Borrow	100000000	Carry
100000000		00000001	
-00000001		+11111111	
-----		-----	
11111111		00000000	

This method of representing negative numbers is called *two's complement*

Two's Complement

Almost all the code and computers you'll use rely on two's complement

But this didn't settle down until the 1970s

There are other ways to represent numbers and negative numbers

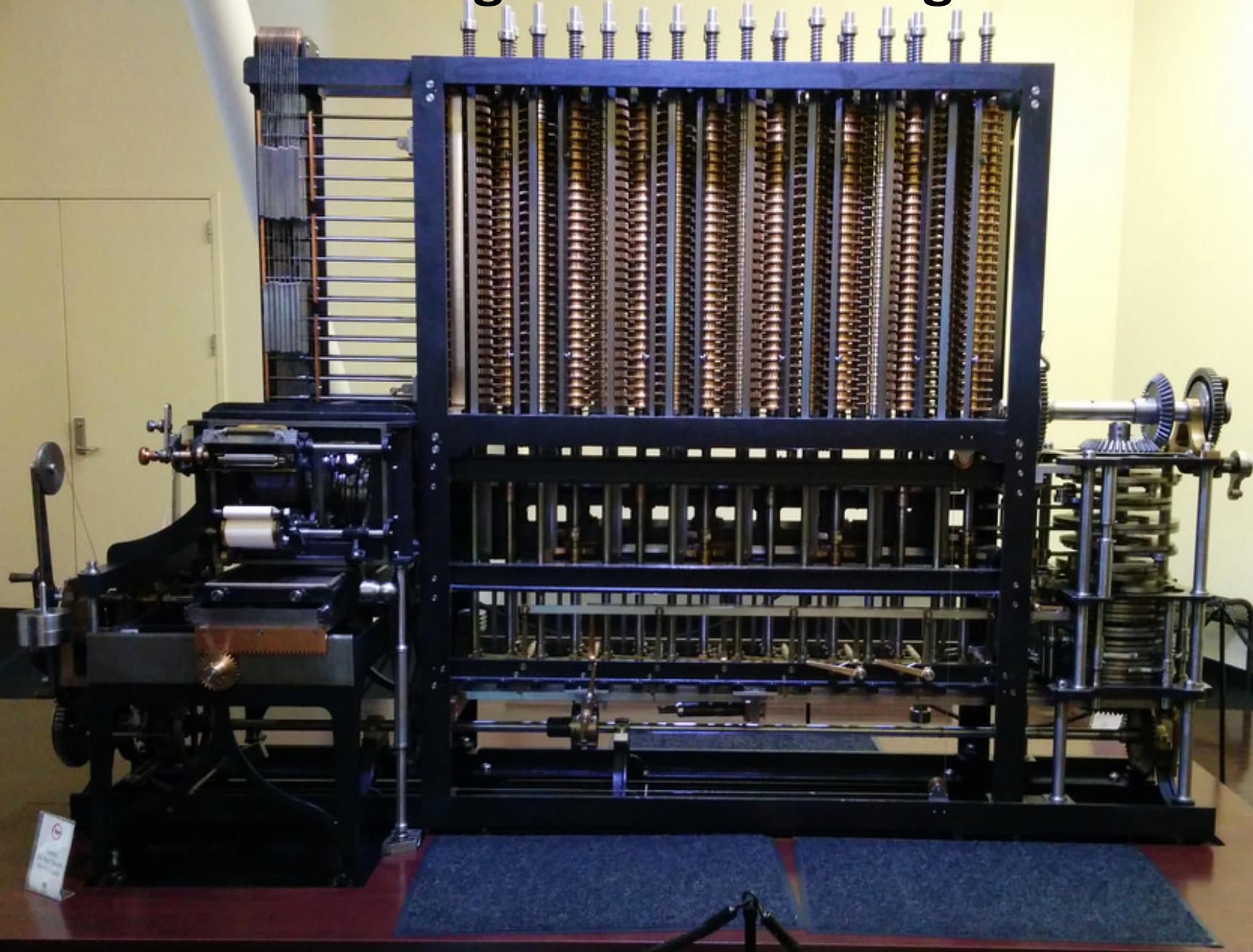
- One's complement: negative is just flip all the bits
 - There are two zeroes, $-0 + 1 \neq 0$ ($0\text{ff} + 0\text{01}$)
- Binary coded decimal: each digit is encoded by 4 bits (ignore 0xa - 0xf): supported on Intel except in 64 bit mode

IBM 3/60



By ArnoldReinhold - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=47096462>

Babbage Difference Engine



**Addition and Subtraction
of signed and unsigned numbers
are the same!**

**Methods used to *compare*
signed and unsigned numbers
are NOT the same!**

Types and Type Conversion

Type Conversion

Type conversion is a way of converting data from one type to another type

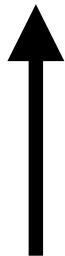
Explicit type conversion means that the programmer must specify type conversions. Often called *casting*.

Implicit type conversions means that the language will have rules for performing type conversion for you. Often called *coercion*

Casting sometimes refers to just a reinterpretation of the same bits.

Type Hierarchy

uint32 $\{0, \dots, 4294967295(0xffffffff)\}$



uint16 $\{0, \dots, 65535(0xffff)\}$

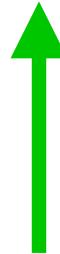


uint8 $\{0, \dots, 255(xff)\}$

Types are *sets* of allowed values

Arrow indicate *subsets*: $\text{uint16} \subset \text{uint32}$

uint32



uint16



uint8

**Type Promotion is Safe
(values preserved)**

```
#include <stdint.h>
```

```
uint16_t x = 0xffff;
```

```
uint32_t y = x;
```

```
// x = 0xffff
```

```
// y = ?
```

```
#include <stdint.h>
```

```
uint16_t x = 0xffff;
```

```
uint32_t y = x;
```

```
// x = 0xffff
```

```
// y = 0x0000ffff
```

```
int16_t x = -1;  
int32_t y = x;
```

```
// x = -1  
// y = ?
```

```
int16_t x = -1;  
int32_t y = x;
```

```
// x = -1  
// y = -1
```

int32 {-2,147,483,648,...,2,147,483,647}



int16 {-32768,...,32767}



int8 {-128,...,127}

Type Conversion is Safe
(values preserved)

```
int16_t x = -1;
```

```
int32_t y = x;
```

```
// x = -1 = 0xffff
```

```
// y = -1 = 0xffffffff
```

```
int16_t x = 1;
```

```
int32_t y = x;
```

```
// x = 1 = 0x0001
```

```
// y = 1 = 0x00000001
```

// To preserve signed values need *sign extension*

int8_t 0xfe -> **int32_t** 0xfffffffffe

int8_t 0x7e -> **int32_t** 0x0000007e

// Sign extend instructions:

//

// **sxtb** - sign extend byte to word

// **sxth** - sign extend half word to word

//

```
int32_t x = 0x80000;
```

```
int16_t y = x;
```

```
// x = 0x80000
```

```
// y = ?
```

```
int32_t x = 0x80000;
```

```
int16_t y = x;
```

```
// x = 0x80000
```

```
// y = 0x0000
```



value has changed

int32



int16



int8

Defined (remove most significant bits)

Dangerous (doesn't preserve all values)

```
int32_t x = -1;  
uint32_t y = x;
```

```
// x = -1  
// y = ?
```

```
int32_t x = -1;  
uint32_t y = x;
```

```
// x = -1  
// y = 0xffffffff = 4294967295
```



value has changed

x is negative, but y is positive!

uint32 ← **int32**

uint16 ← **int16**

uint8 ← **int8**

Defined (copies bits)

uint32 ← **int32**

uint16 ← **int16**

uint8 ← **int8**

Dangerous! (neg maps to pos)

uint32 → **int32**

uint16 → **int16**

uint8 → **int8**

**Technically Not Defined
(arm: copies bits)**

uint32 → **int32**

uint16 → **int16**

uint8 → **int8**

Dangerous !

(large positive numbers change)

**"Whenever you mix
signed and unsigned numbers
you get in trouble."**

Bjarne Stroustrup

Implicit Type Promotion

in

Binary Operators

Type promotions for binary operations

Note that the type of the result can be different than the type of the operands !

	u8	u16	u32	u64	i8	i16	i32	i64
u8	i32	i32	u32	u64	i32	i32	i32	i64
u16	i32	i32	u32	u64	i32	i32	i32	i64
u32	u32	u32	u32	u64	u32	u32	u32	i64
u64								
i8	i32	i32	u32	u64	i32	i32	i32	i64
i16	i32	i32	u32	u64	i32	i32	i32	i64
i32	i32	i32	u32	u64	i32	i32	i32	i64
i64	i64	i64	i64	u64	i64	i64	i64	i64

arm-none-eabi-gcc type promotions

uint32

int32

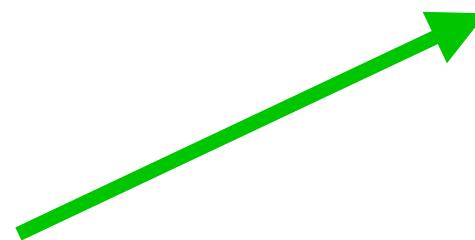
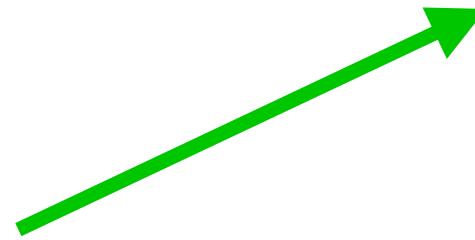
uint16

int16

uint8

int8

Safe?

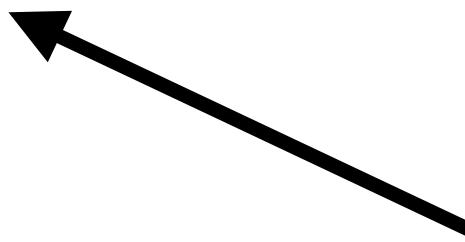
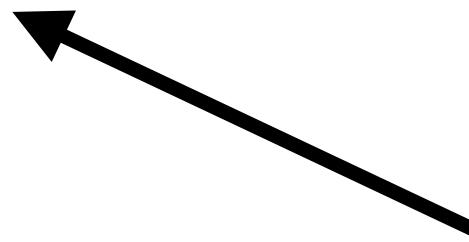


uint32 **int32**

uint16 **int16**

uint8 **int8**

Safe?





Bugs, Bugs, Bugs

A significant fraction of major Linux security vulnerabilities today are integer bugs.

```
#include <stdio.h>

int main(void)
{
    int a = -20;
    unsigned int b = 6;

    if( a < b )
        printf("-20<6 - all is well\n");
    else
        printf("-20>=6 - omg \n");
}
```

Be Wary of Implicit Type Conversion

**Modern languages like rust and go do
not perform implicit type conversion**

Summary

Negation is performed by forming the two's complement

- Signed numbers are represented in two's complement ($-x = 2^n - x = \sim x + 1$)

In 2's complement,

- Arithmetic between signed and unsigned numbers is identical
- Comparison between signed and unsigned numbers is different

Know the rules for type conversion, watch out for implicit type conversions and promotions

C Type Conversion and Promotion Rules

The semantics of numeric casts are:

Casting from a larger integer to a smaller integer (e.g. u32 -> u8) will truncate

Casting from a smaller integer to a larger integer (e.g. u8 -> u32) will zero-extend if the source is unsigned sign-extend if the source is signed

Casting between two integers of the same size (e.g. i32 -> u32) is a no-op

6.3.1.3 Signed and unsigned integers conversions

- 1 When a value with integer type is converted to another integer type, if the value can be represented by the new type, it is unchanged.
- 2 Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.
- 3 Otherwise, if the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

6.3.1.8 Usual arithmetic conversions

- 1 If both operands have the same type, then no further conversion is needed.
- 2 Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
- 3 Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
- 4 Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.
- 5 Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.