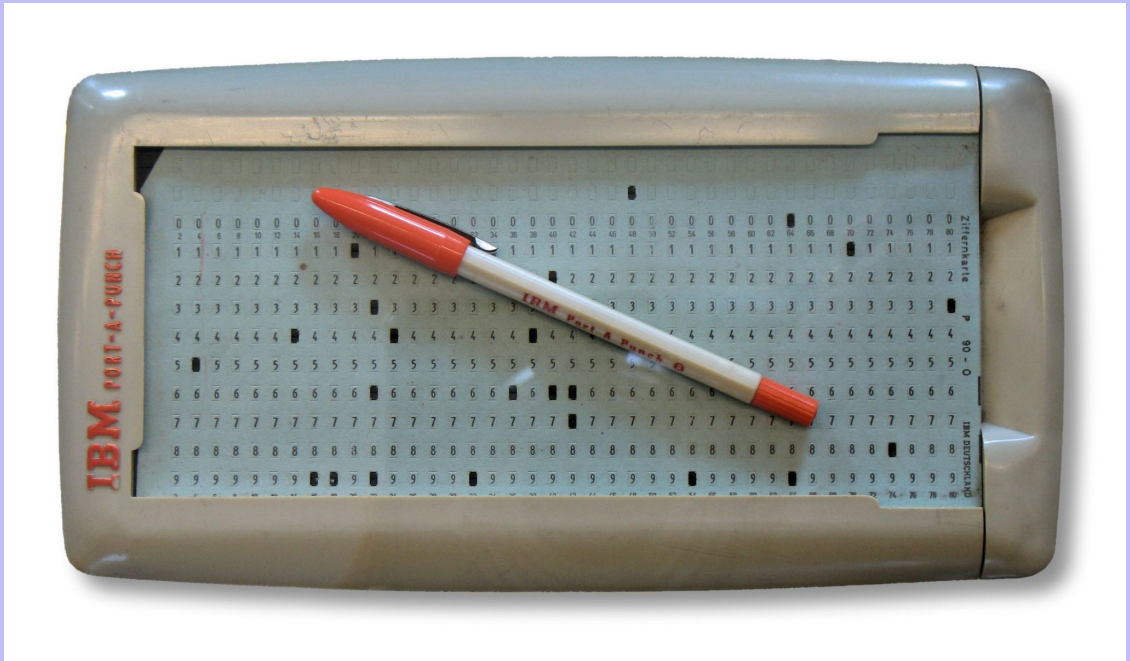
Data representations

Remember to fill out HW0!
HW1a (setup) is out – post issues on Ed

? ? ?



By Journey234 - Own work, Public Domain, link

Binary for data representation

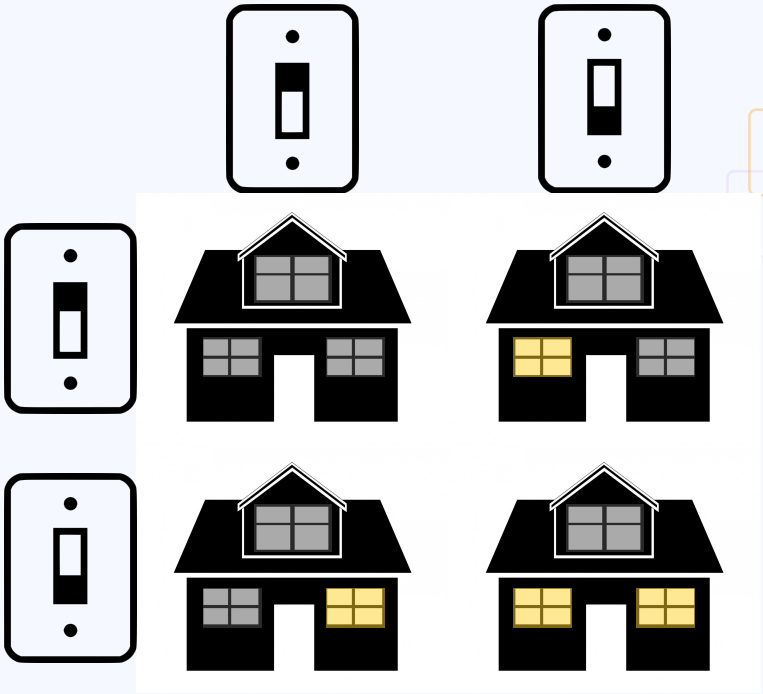
"on/off" is simplest way to convey state

- switch, punch card, electrical signal…

each bit (binary digit) doubles the information we can convey

same data can be *interpreted* multiple ways

- int vs. float vs. char
- signed vs. unsigned
- data vs control signal



What to take away from today

Except for HW1 (where you practice these concepts), you won't be asked to convert between representations by hand

For quick debugging and general understanding, it's helpful to know the conversions

Understanding how computers "think" about data is one of the fundamentals of architecture – concepts like signed/unsigned, size, endianness, semantics will keep coming up

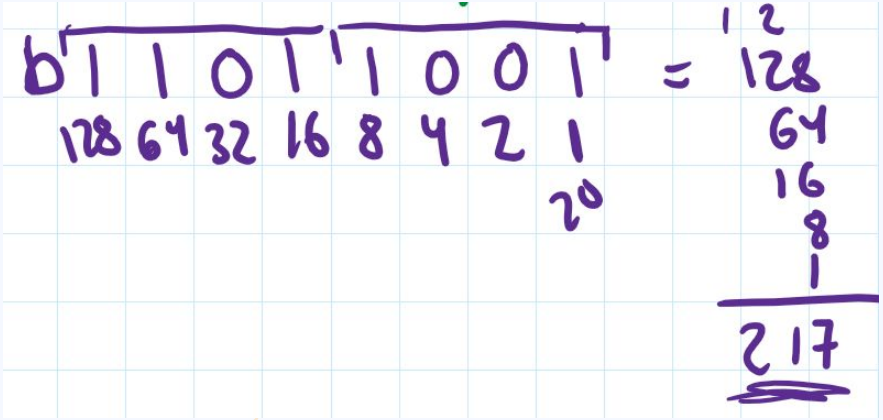
Decimal (base10) vs. binary (base2)

Numerical base is a shorthand for counting

Each place in a base10 number is an additional power of 10

Each place in a base2 number is an additional power of 2

Computers “think” in base2 but it’s helpful for us to know how to convert between the two



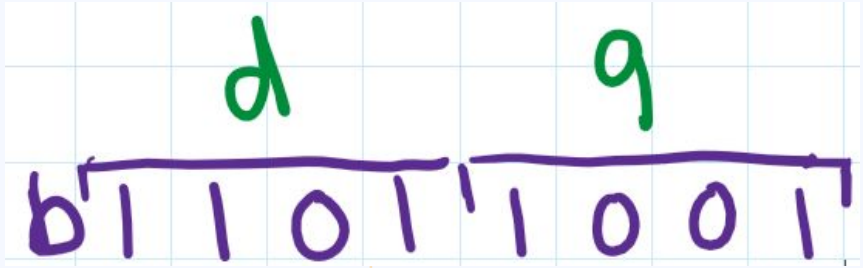
Hexadecimal (base16)

16 digits: 0-9 and a-f (a = 10, b = 11, etc)

Often used by CS because binary numbers get long

Computers don't actually "think" in hexadecimal

Trick for converting between hex and binary: 4 bits = hex digit



Interpreting data

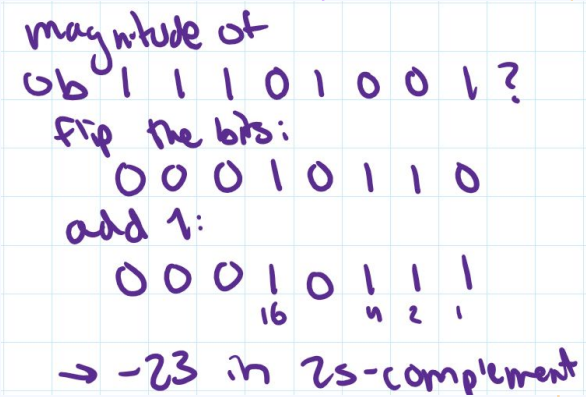
Example

Same bits in memory, different operations/interpretations

Type specifiers define *semantics* of what kind of operations are expected for a piece of data

When programming in C++, use [u]int<SIZE>_t, e.g. int16_t or uint32_t

Negative binary numbers



In decimal: use “-” to denote a negative number

There is no “-” in binary: what to do?

Enter: two’s complement

- Negate a number by flipping the bits and adding 1
- Turns out, math just works
- Easy to check if number is negative (1 in MSB = negative)
- Easy to cast to larger number (“sign-extend” by copying MSB)

Bit manipulation

We still have addition, subtraction, etc (same principles, same mechanics)

Also have: bitwise- and (&), or (|), xor (^), not (~)

- Use *bitmasks* to set (**`or`** w/ 1), clear (**`and`** w/ 0), or flip (**`xor`** w/ 1) certain bits (examples)

Shifts: right (>>) and left (<<)

- Left shift mathematically equivalent to multiplying by powers of 2
- Right shift: logical (pad w/ 0s) or arithmetic (pad w/ sign-extend)
- examples

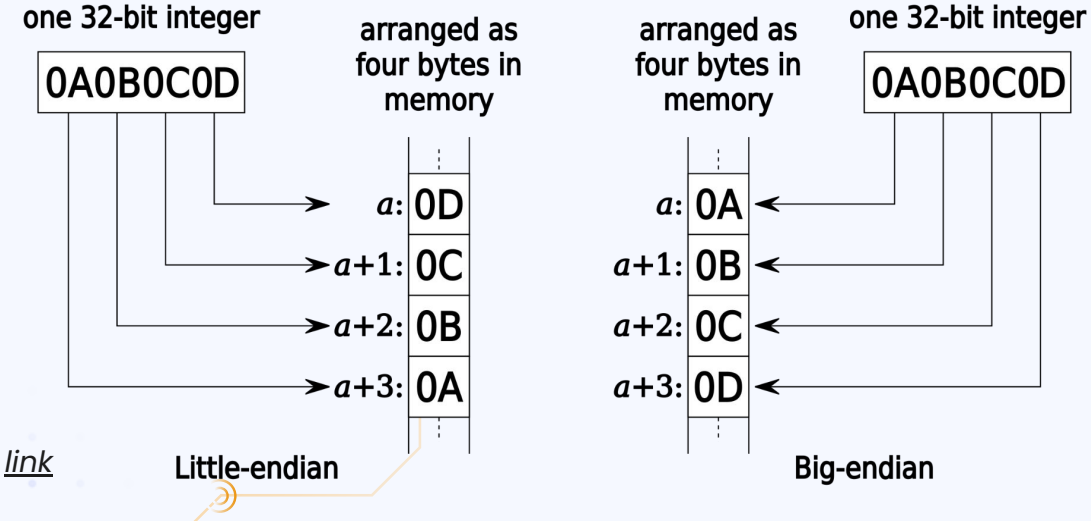
>> has different behavior on ints vs uints
do you think the decision to use logical vs. arithmetic shift is made by the compiler or the CPU?

Numbers in memory

Memory stores information for a computer

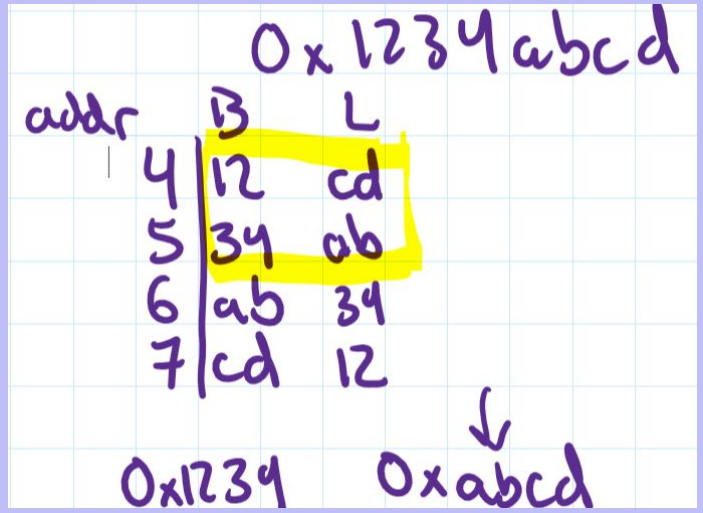
Each *byte* (8 bits) of data has a location (address) in memory

We often compute on 32-bit or 64-bit numbers (4 or 8 bytes) – how are these stored?



? ? ?

What can we conclude about the endianness of the machine running this code?



Aligned addressing

Each memory address refers to a byte

32-bit integers *typically* have addresses that are multiples of 4 (why?)

Mostly up to the compiler; constrained by ISA

Address	Value
0x0000	0x4d
0x0001	0xf0
0x0002	0x00
0x0003	0x18

Stored-program computers

Modern computers hinge on two principles:

- Instructions are represented in memory the same way as numbers
- Memory can be altered by programs

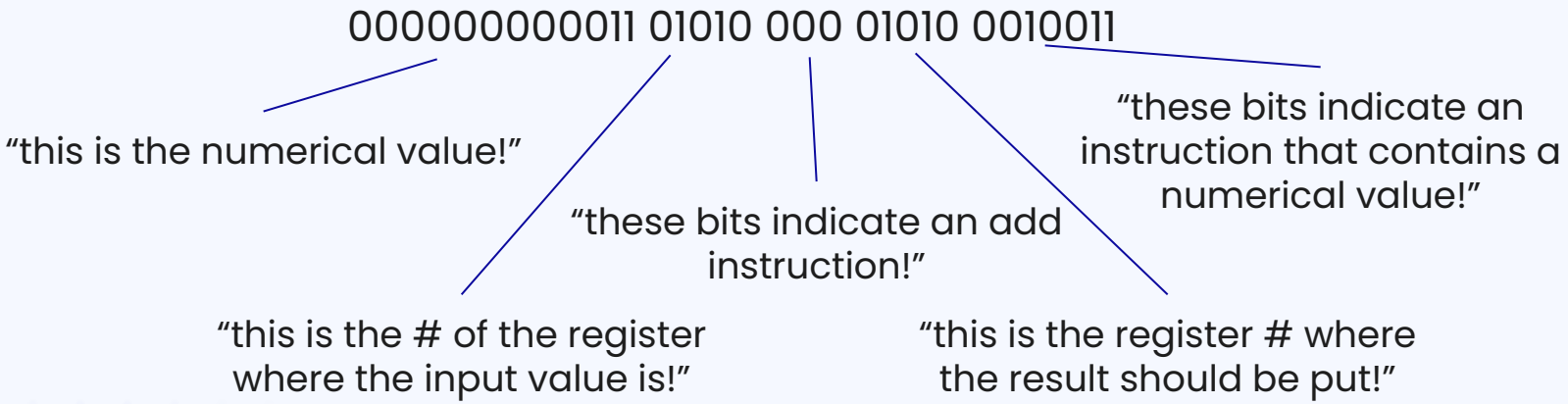
First principle means that:

- Instructions live in memory
- The CPU needs to have a way of interpreting an instruction, just as it would any other data in memory

Interpreting instructions

We could interpret ‘0xe7` as 233, -23, or ← based on context

Similarly, we can interpret `0x00350513’ as 3474707 or the instruction “increment register 10 by 3”



We’ll keep coming back to this next week as we learn more about the instructions that are available!