# Data Representations and Assembly

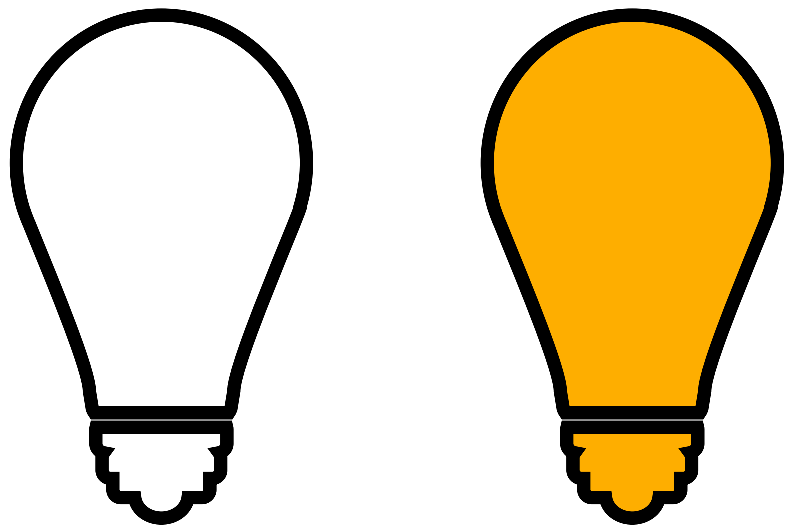
No class Monday, have a good Labor Day Weekend!


“Fortran program:
PIFRA=(A(JB,37)-A(JB,99)/A(JB,47)”

“You may scoff at the low-tech but this data has survived longer than most CD-Rs will last - they rot in a decade or so…”

# Binary for Data Representation

- “on/off” is the simplest way to convey state:
  - switch, punch card, electrical signal…
- Each bit (binary digit) doubles the information we can convey
- Some data can be *interpreted* multiple ways
  - int v float v char
  - signed v unsigned
  - data v control signal

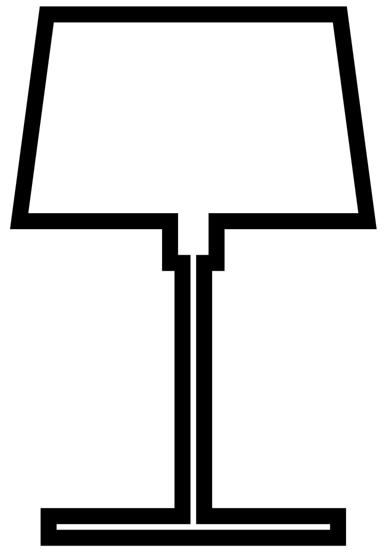
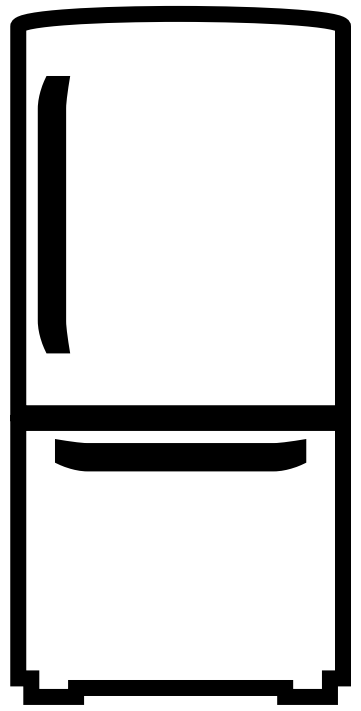
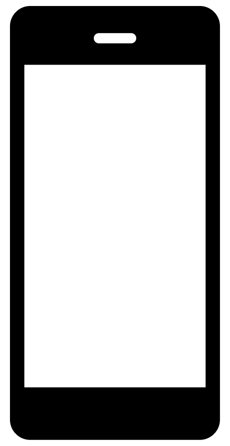
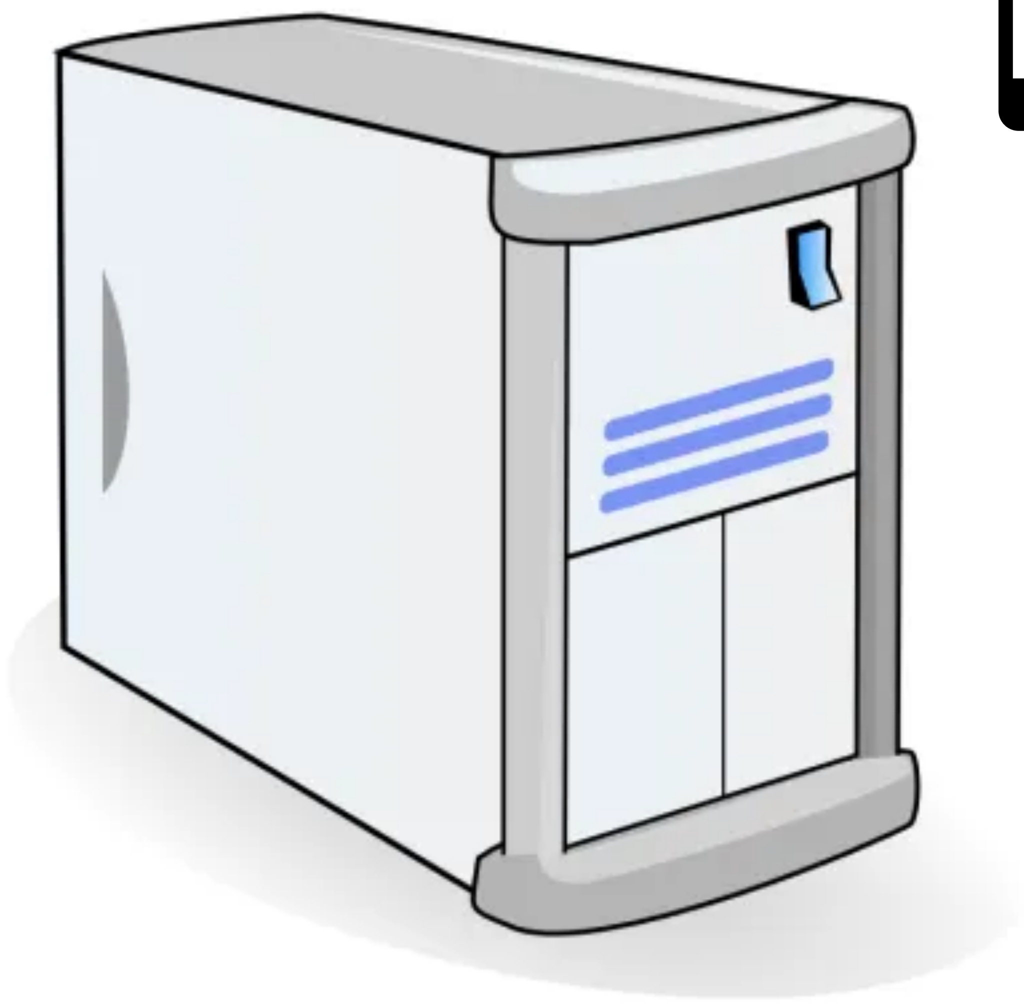
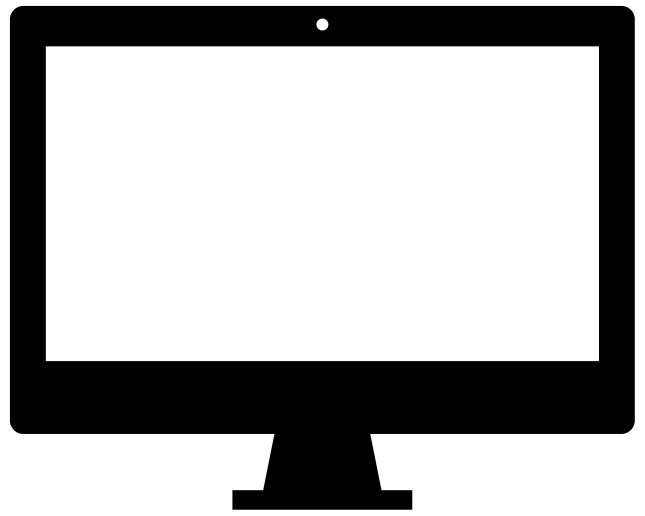
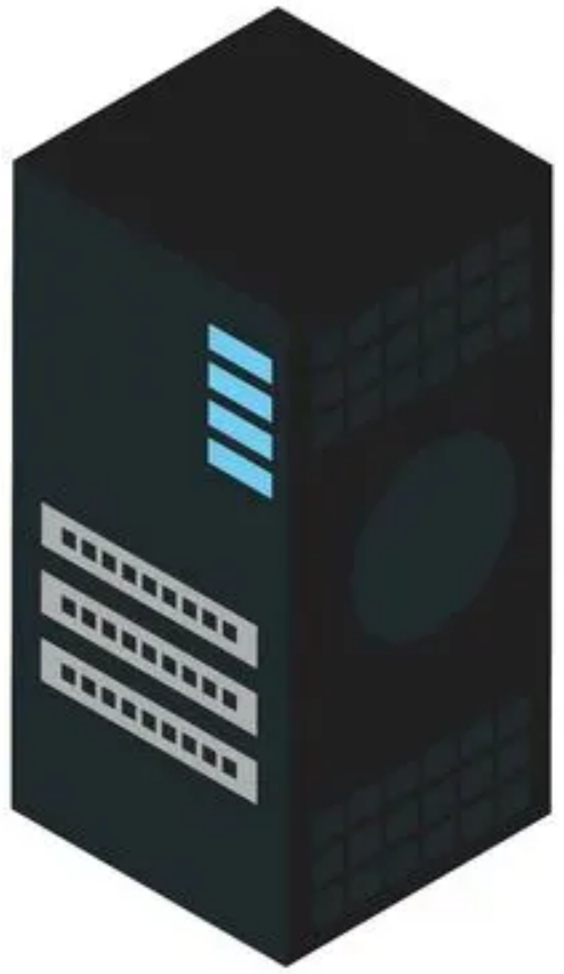


# Outline

- Very brief overview of hardware platforms (from Wednesday)

# Types of Hardware Platforms

- Embedded Devices/Internet of Things (IoT)
- Personal Mobile Devices (PMD)
- Desktop
- Server
- Cluster/Warehouse-Scale

# Types of Hardware Platforms

- Embedded Devices/Internet of Things (IoT): cost, energy, specialized application performance
- Personal Mobile Devices (PMD): cost, energy, media performance, responsiveness
- Desktop: combination of price and performance, energy, graphics performance
- Server: throughput, availability, energy, scalability
- Cluster/Warehouse-Scale: throughput, combination of price and performance, energy proportionality

RISC-V

ARMv8-32, x86_32

ARMv8-64, x86_64

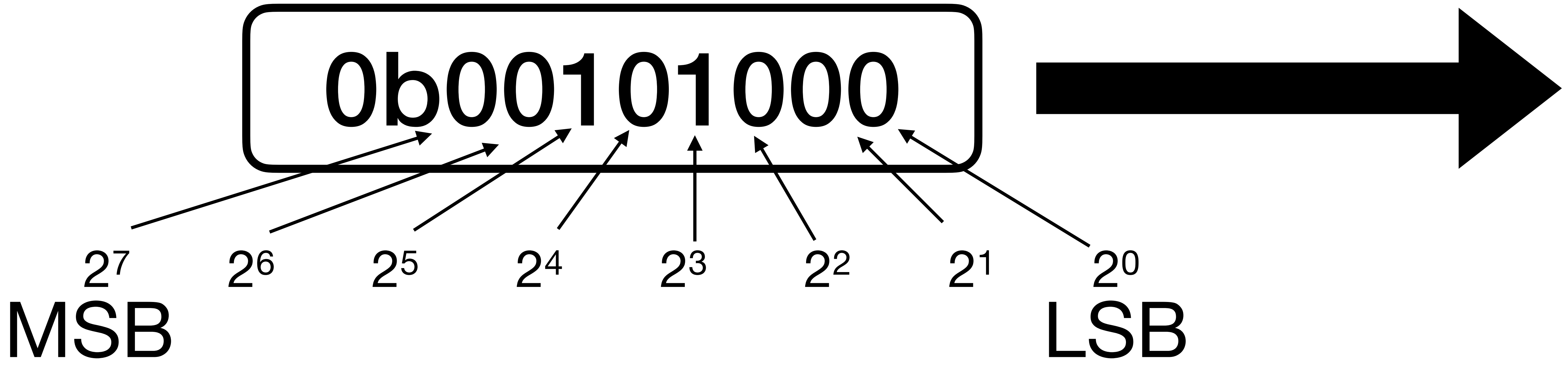
# Outline

- Review of binary and hexadecimal representations
- Review of data storage in memory
- Introducing storage of instructions in memory

Goal: you will seldom be asked to convert data representations by hand; understanding how computers “think” is a fundamental of architecture that will keep coming up

# Quick Review: Decimal (base 10) v Binary (base 2)

- Numerical base is a shorthand for counting
- Each place in a decimal number is an additional power of 10
- Each place in a binary number is an additional power of 2
- Computers “think” in base 2 but it’s helpful to know how to convert between the two to make sense of debugging output, etc.




$2^3 + 2^5 = 8 + 32 = 40$

# Hexadecimal (base 16)

- 16 digits: 0-9 and a-f (a = 10, b = 11, etc.)
- Often used by computer scientists because binary numbers get long
- Computers don't actually "think" in hexadecimal
- Trick for conversion: every four bits is a hex digit

0b00101000 → 0x28 → $8*16^0 + 2*16^1 = 8 + 32 = 40$

# Negative Binary Numbers

- In decimal: use “—“ to denote a negative number
- There is no “—“ in binary: what to do?
- 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)

```
      0b00101000
flip: 0b11010111
    +          1
----------------
      0b11011000
```

# Chat with your neighbor(s)!

## Flip the sign of the following numbers so that they are an 8 bit binary number…

| | | |
|---|---|---|
| 0b01111111 | $123_{10}$ | 0xCA |

# Bit Manipulation

- We still have addition, subtraction, etc (same principles, same mechanics)
- Also have: bitwise-and (&), or (|), xor (^), not (~)
  - Examples: use *bitmasks* to set (**or** w/ 1), clear (**and** w/ 0), or flip (**xor** x/ 1) certain bits
- Shifts: right (>>) and left (<<)
  - Left shift mathematically equivalent to multiplying by powers of 2
  - Right shift: logic (pad w/ 0s) or arithmetic (pad w/ sign-extend)

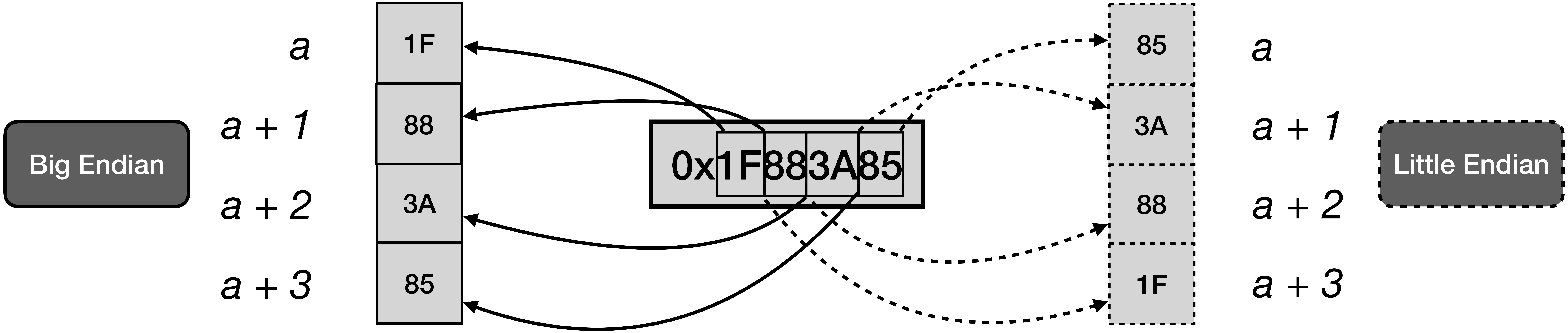
# Interpreting Data

- 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++, using `[u]int<SIZE>_t` e.g., `uint16_t` or `uint32_t`
- Just like how programs need to interpret bits as types, the processor interprets instruction bits as types!

# Storing Data in Memory

# 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 they stored?



# Chat with your neighbor(s)!

What is an advantage of a number stored in little endian format?

# Little Endian Arithmetic

```
uint32_t x = 0x1F883A85;
uint32_t y = 0x53E7A35;
uint32_t z = x + y;
```

# Storing Programs

**Not distinguishing between data and instructions is why buffer overflow attacks are possible!**

- Instructions are stored the same way as numbers… binary digits!
- Instructions live in memory
- The CPU needs to have a way of interpreting an instruction, just as any other data type stored in memory…
  - Consider the hexadecimal number: `0x00350513`
  - Could be interpreted as the decimal number `3474707` or as the RISC-V assembly instruction “increment register 10 by 3”!

# Storing Programs

**High-level language:** a portable language such as C, C++, Java that is composed of words and algebraic notation

**Assembly language:** a symbolic representation of machine instructions

**Machine language:** a binary representation of machine instructions

`x++;` —compiler→ `addi   a0, a0, 1` —assembler→ `0x00150513`

Definitions from textbook Chapter 2

# Takeaways

- Computers represent all data as binary values to easily interpret electrical signals
- We can get the value of a digit in any base to perform conversions
- Data is typically represented as little endian to easily fetch values from memory!

Exit ticket!

https://forms.cloud.microsoft/r/m2Jx8KNj9J