CS 217 Data Management and Information Processing

02 - Data Storage and Representation

Overview

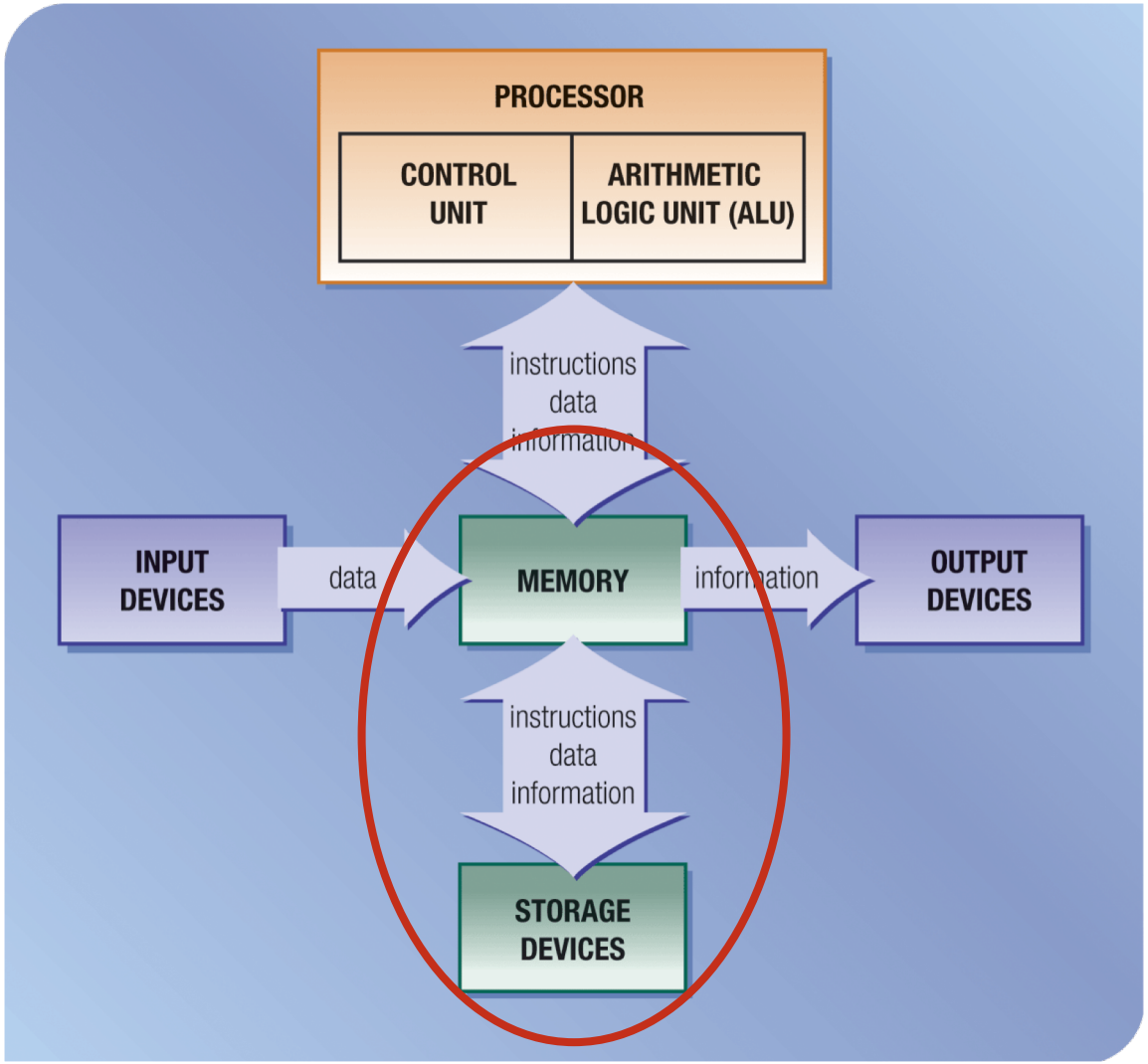
- Data storage
  - All data are stored in bits
  - Different storage device
  - RAID for reliability
- Data representation
  - Binary format for integers
  - 2’s complement
  - Additions of integers
  - Floating point

Data Storage

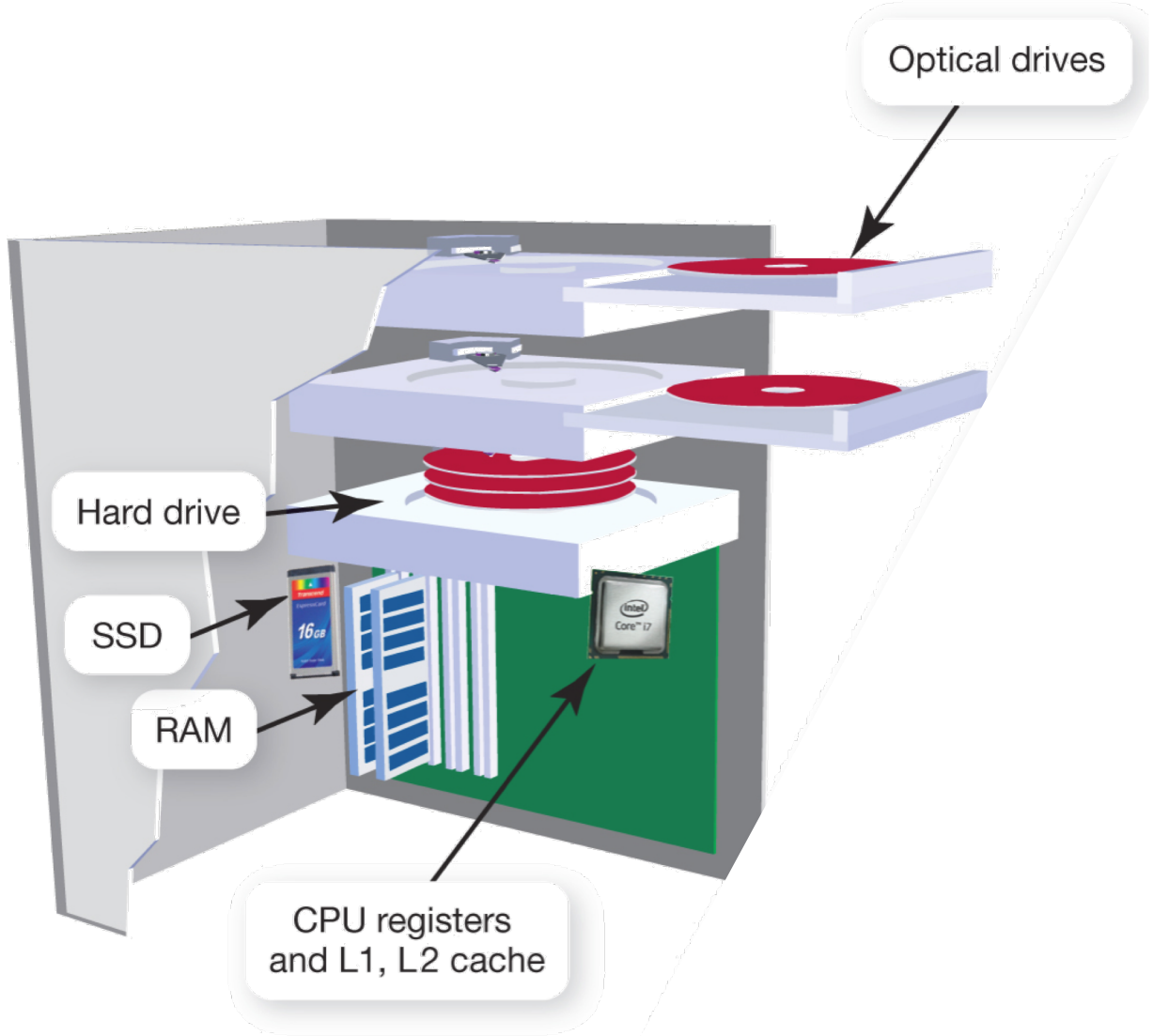
Basic Unit of Data Storage

- Bit (binary digit): computer works with binary digits. These digits are in the form of 0's and 1's. A binary digit is called bit.
  - One bit takes one storage location in memory. It is the smallest unit for data storage.
- Byte (B): A sequence of eight bits is called a byte. The capacity of the memory or the storage is expressed in terms of bytes.
- Kilobyte (KB): 1KB = 1024 B
- Megabyte (MB): 1MB = 1024 KB
- Gigabyte (GB): 1GB = 1024 MB
- Terabyte (TB): 1TB = 1024 GB

What are the Components of a Computer?



Different Types of Computer Storage



Data Storage

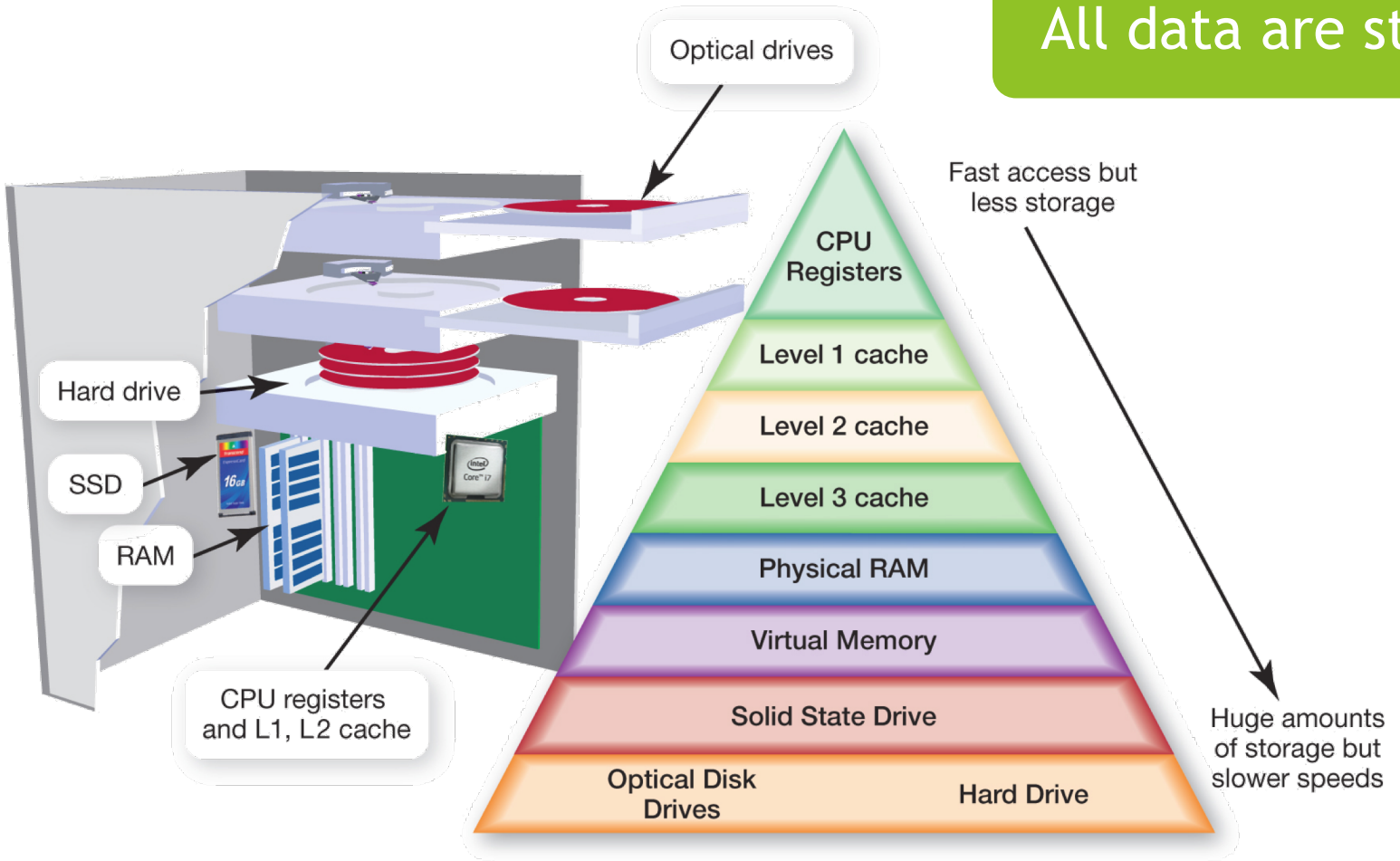
- Computer data storage, often called storage or memory, is a technology consisting of computer components and recording media used to retain digital data.
- Types:
  - Volatile storage: data will be lost after power-off
  - Non-Volatile storage: data will not be lost after power-off
- Volatile storage tends to be more expensive, but faster.

Storage has limited bandwidth

- **Bandwidth**: the amount of data to transfer at once
- All types of computer storage are limited to reading/writing just a small fraction at once.
- **Magnetic disks**:
  - The read/write head can read the electronical charges on a tiny portion of the magnetic disk.
- **RAM (memory):**
  - Memory and flash chips store lots of data, but only a few bytes can be transferred at once, because there are only a couple hundred electrical connections at the edge.
  - SSDs (flash) is similar, with even fewer electrical connections.

The Memory Hierarchy

All data are stored in bits!



Computers have a hierarchy of storage

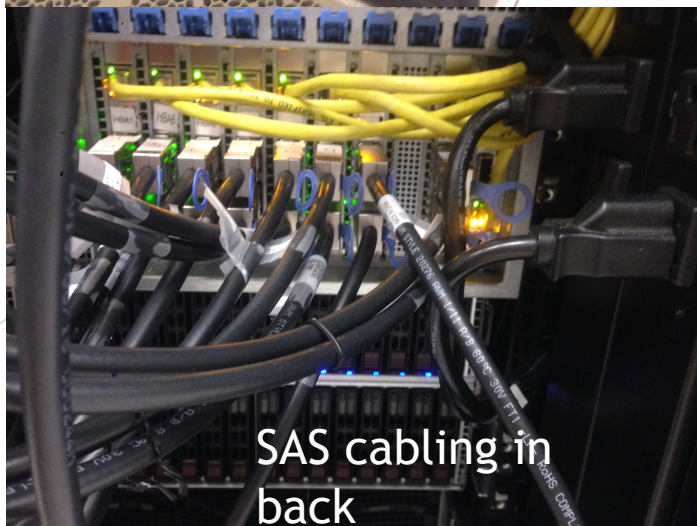
Larger, but slower

delay		*capacity*
0.3ns	CPU Registers	1 kB (kilobyte)
5ns	CPU Caches (L2)	16 MB
50ns	Random Access Memory (RAM)	16 GB
100µs	Flash Storage (SSD)	1 TB
5ms	Magnetic Disk	8 TB

- Delay (latency): the time between the request is made and the data is received
- Disk is about *ten billion* times larger than registers, but has about *ten million* times larger delay.
- Goal is to work as much as possible in the top levels.

A Database Server @ NU

- 264 fast (10k RPM) magnetic disks (for production)
- 56 slow (7200 RPM) magnetic disks (for backup)
- ~150 TB storage capacity
- Comprised of 6 physical chassis (boxes) in one big cabinet, about the size of a coat closet.



Large Volume Storage

- Can store a lot of data, for a long time
- A very small chance to lose data, due to hardware failure.
- Read https://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers
- Solution?

Redundant Array of Independent Disks (RAID)

- Distribute a storage system intelligently across multiple disks to
  - Maintain high reliability *and* availability
  - Enable fast recovery from failure
  - Increase performance

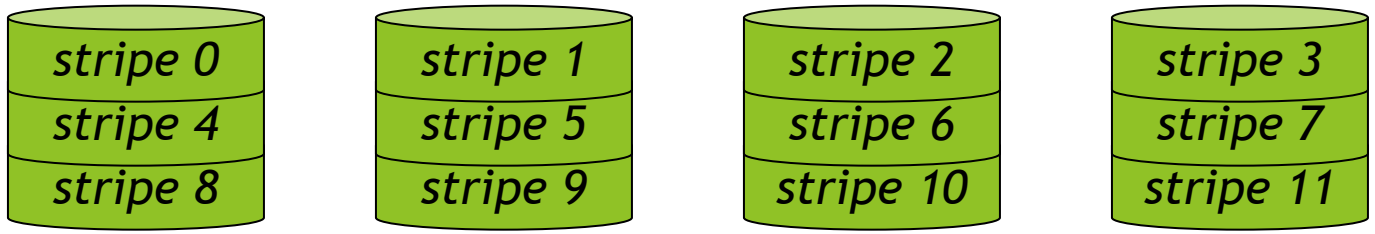
“Levels” of RAID

- Level 0 – non-redundant striping of blocks across disk
- Level 1 – simple mirroring
- ~~Level 2 – striping of bytes or bits with ECC~~
- ~~Level 3 – Level 2 with parity, not ECC~~
- Level 4 – Level 0 with parity block
- Level 5 – Level 4 with distributed parity blocks
  - We will not talk about Level 4 and 5, read Wikipedia if you are interested.

Never seriously used

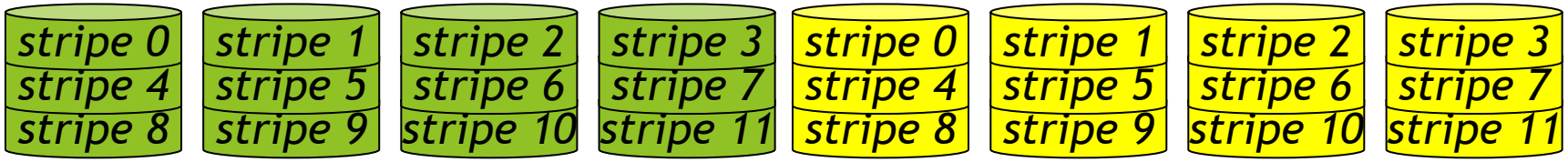
RAID Level 0 – Simple Striping

- Partition our database into 12 stripes



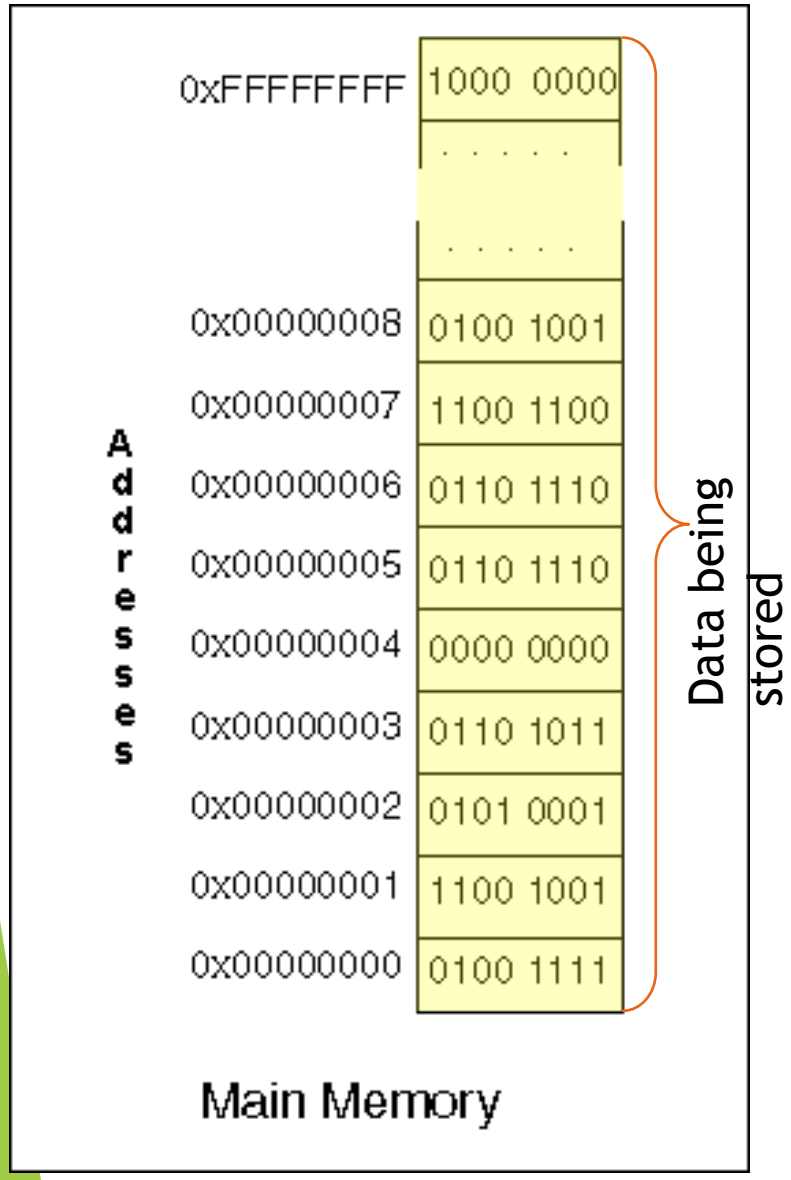
- Each stripe is a subset of contiguous data
- Stripe *i* is on disk (*i* **mod** *n*), where n is the number of disks
- Advantage
  - Read/write *n* blocks in parallel; *n* times bandwidth
- Disadvantage
  - No redundancy at all. System can still fail!

RAID Level 1– Striping and Mirroring



- Each stripe is written twice
  - Two separate, identical disks
- Stripe *i* is on disks (*i* **mod** 2n) & (*i*+*n* **mod** 2*n*)
- Advantages
  - Read/write *n* blocks in parallel; *n* times bandwidth
  - Redundancy: Data lose only if two disks fail
  - Failed disk can be replaced by copying
- Disadvantage
  - A lot of extra disks for much more reliability than we need

Abstract view of computer storage



- Storage is a big array of bytes.
- Computer can read or write from one location (address) at a time.
- The picture at left is misleading because a human observer can *see all the data* at once. The computer cannot!

Data Representation

Computers store information in **binary**

- Ones and Zeros
- …0001001000010010011100110110101010111100…

- Why?
  - Simplicity
  - Noise robustness
  - By convention

- But how do we get meaning from a sequence of ones and zeros?

Interpretation

- An encoding defines what the zeros and ones represent
- “01000100011000010111010001100001” can represent:
  - The number 1,147,237,473 as an integer
  - The number 901.8184 as a float
  - The four letters “Data” in the ASCII or UTF-8 character encoding
  - This color (at 37% transparency) in RGBA
  - 32 separate True or False values
- Any crazy encoding is possible, but there are some standards.

Integers

- Integers are the simplest of all data encodings
- Whole numbers only (no fractions)
- Numbers are represented directly in the “base two” positional notation
- The familiar “base ten” representation of numbers is just a convention due to the fact that humans have ten fingers.

Integers in detail

Decimal 137_{ten}

1	3	7
x100	x10	x1

← powers of 10

100 + 30 + 7 = 137

Binary $10001001_{two} = 137_{ten}$

1	0	0	0	1	0	0	1
x128	x64	x32	x16	x8	x4	x2	x1

← powers of 2

128 + 0 + 0 + 0 + 8 + 0 + 0 + 1 = 137

Simple binary integers

$1_{ten} = 1_{two}$
$2_{ten} = 10_{two}$
$4_{ten} = 100_{two}$
$8_{ten} = 1000_{two}$
$16_{ten} = 10000_{two}$
$32_{ten} = 100000_{two}$
$64_{ten} = 1000000_{two}$
$128_{ten} = 10000000_{two}$

$3_{ten} = 11_{two}$
$7_{ten} = 111_{two}$
$15_{ten} = 1111_{two}$
$31_{ten} = 11111_{two}$
$63_{ten} = 111111_{two}$
$127_{ten} = 1111111_{two}$
$255_{ten} = 11111111_{two}$

There are only 10 types of people in this world… those who understand binary and those who don’t.

Binary tricks

- Remember the first eight powers of two:
  - 2, 4, 8, 16, 32, 64, 128, 256
- Remember that **$2^{10} = 1024$** ≈ 1000
  - Lets you estimate the number of binary digits in a decimal integer:
    Every three decimal digits gives about ten binary digits
- Remember the important large powers of two:
  - $2^8 = 256$
  - 2^{16} ≈ 64 thousand
  - **2^{32} ≈ 4 billion**
  - 2^{64} ≈ really big

Addition in binary

4 + 7 = 11

1 ← carry

```
  4
+ 7
-----
1 1
```

100 + 111 = 1011

1 ← carry

```
  1 0 0
+ 1 1 1
-------
1 0 1 1
```

More binary addition

63 + 98 = 161

```
1  1  ← carry
   6  3
+  9  8
-------
1  6  1
```

11111 + 110010 = 1010001

```
1  1  1  1  1  ← carry
      1  1  1  1  1
+  1  1  0  0  1  0
-------------------
1  0  1  0  0  0  1
```

What about negative integers?

- Signed integers can represent both positive and negative integers
- We need an extra bit to represent the sign of the number
- But we don't just use a simple sign bit
- We use two's complement to represent negative numbers, because it
  - Simplifies the computer's addition and subtraction circuitry, and
  - And it has just one representation of zero
- Negative numbers "roll over" from the top of the binary range.

Two's complement for three-bit numbers

```
 3: 011
 2: 010
 1: 001
 0: 000
-1: 111 ↩ rollover
-2: 110
-3: 101
-4: 100
```

-2 + 1 = -1

```
110 + 001 = 111
```

- Subtraction is done in the exact same way as addition!

Subtraction works just like addition!

Just negate the second number and add.

3 - 2 = 3 + (-2) =

```
1  1  ← carry
   0  1  1
+  1  1  0
-------------
   0  0  1  ← our answer!
```

We ignore the final carry because it falls outside of the 3-bits we are working with. That's how we roll-over between negative and positive.

```
 3:  011
 2:  010
 1:  001
 0:  000
-1:  111
-2:  110
-3:  101
-4:  100
```

Two's complement negation

To negate a number:

- Flip all the bits. Ones become zeros and zeros become ones.
- Add one

For example -3

- Start with the bits for three: `011`
- Flip the bits: `100`
- Add one: `101`

```
 3: 011
 2: 010
 1: 001
 0: 000
-1: 111
-2: 110
-3: 101
-4: 100
```

Overflow: when numbers don't fit

For example, 2 + 2 = 4

4 cannot be represented in a three-bit signed integer. What happens when we try this addition?

```
  1   ← carry
  0 1 0
+ 0 1 0
-------
  1 0 0   ← answer looks like -4!
```

```
 3:  011
 2:  010
 1:  001
 0:  000
-1:  111
-2:  110
-3:  101
-4:  100
```

- Remember that the left-most bit indicates the sign.

Examples with 4 and 8 bits

4-bit is between -8 and 7

8-bit is between -128 and 127

Two’s complement for 4 bits

Two's complement	Decimal
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	−1
1110	−2
1101	−3
1100	−4
1011	−5
1010	−6
1001	−7
1000	−8

A few more things about integers

- Multiplication: two's complement works magically here too
- Positive division works as expected
- *Sign extension*: when increasing the "bit size" of a negative number, add leading ones.
  - Eg., -2 is `1110` as a 4-bit signed integer and `11111110` in 8 bits.
- Computers typically use 32 or 64 bit integers.

Just for fun

- This video shows how addition is actually implemented in hardware: https://www.youtube.com/watch?v=1I5ZMmrOfnA
  Search YouTube for “PBS ALU”

- If you’re interested in learning more, take COMP_ENG-203 Intro to Computer Engineering

Limitations of Integers

Integers are great for **counting**, but sometimes we need to **measure** fractional quantities.

Binary numbers can have "decimal" places, too

- 0.1111111111_{two} is slightly smaller than 1
- 0.0000000001_{two} is slightly larger than 0
- 0.1_{two} is one half

- $10.101_{two} = 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$
  $= 2 + 0 + 1/2 + 0 + 1/8 = 2\frac{5}{8}$

How shall we represent fractional number in the computer?

Floating point

- Based on **scientific notation**:
  - $10{,}340 = 1.034 \times 10^{4}$
  - $0.00424 = 4.24 \times 10^{-3}$
- Scientific notation gives a compact representation of extreme values:
  - $1{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000 = 1.0 \times 10^{24}$
  - $0.000\ 000\ 000\ 000\ 000\ 000\ 000\ 001 = 1.0 \times 10^{-24}$
- In binary:
  - $100010_{two} = 1.0001_{two} \times 2^{5}{}_{ten} = 1.0001 \times 10^{101}{}_{two}$
  - $0.00101_{two} = 1.01_{two} \times 2^{-3}{}_{ten} = 1.01 \times 10^{-11}{}_{two}$

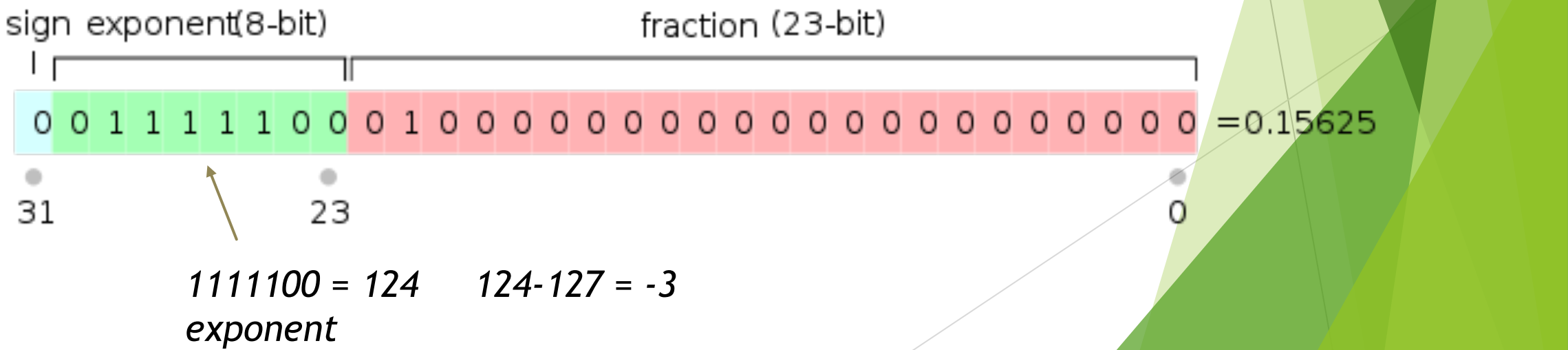
How do computers work with floats?

- It’s complicated and slow!
- Have to manipulate both the fraction and the exponent.
- Addition is no longer simple, as it was for integers.

Representing floating point in bits

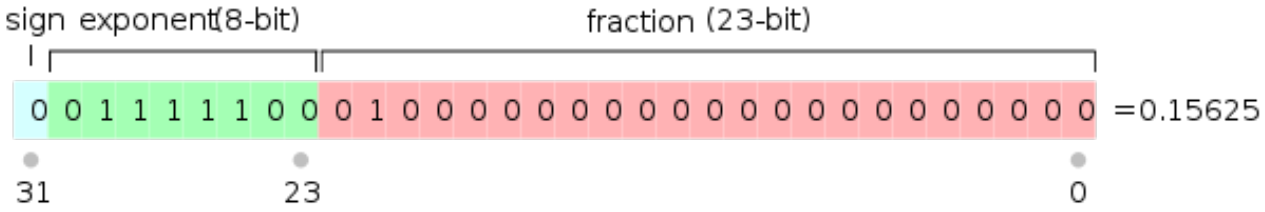
$$0.15625_{ten} = 0.00101_{two} = 1.\underline{01} \times 10^{\underline{-11}}{}_{two}$$

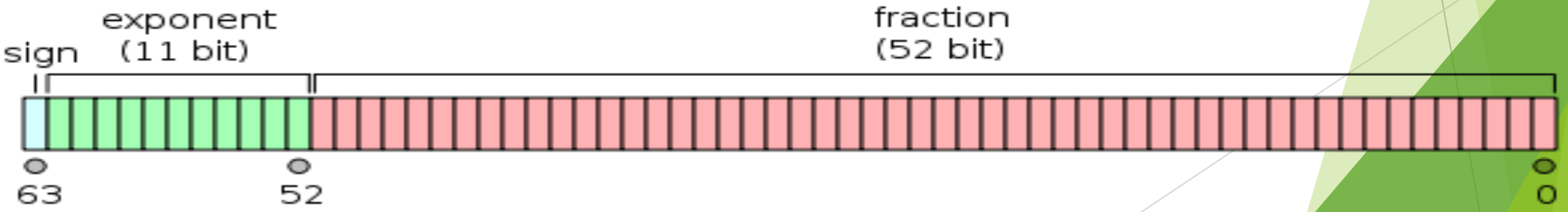
- Three essential parts are the **sign**, **fraction**, & **exponent**
  - Notice that the first significant figure is always “1” so we don’t have to store it
- In the mid 1980s, the IEEE-754 standardized the floating point representation of 32 and 64 bit numbers:
  - The exponent has a sign too, but the standard says to add a “bias” of 127



64-bit floating point

- Similar to 32-bit, but we have more precision in the fraction and larger exponents are possible.
- 32-bit is called **single precision** and 64-bit is called **double precision**.
- Double precision can represent larger, smaller, and more precise numbers.





The Flexibility and Flaws of Floats

- A 32-bit signed integer can represent all the whole numbers between
  -2,147,483,648 and 2,147,483,647
- A 32-bit floating point number can be as large as
  $\pm 3.402823 \times 10^{38}$
  = 340,282,300,000,000,000,000,000,000,000,000,000
- or as tiny as $5.8774718 \times 10^{-39}$
  = 0.000 000 000 000 000 000 000 000 000 000 000 005 877 471 8
- But, single-precision floats have only 24 bits of precision:
  - Can only precisely store **integers** up to 2^{24} = **16,777,216**
- Floats can store larger numbers than integers of the same bit-length, but with less precision because 8 bits are set aside for the exponent.

When to use the various number representations

- When **counting** or labelling things, always use integers
- When **measuring** things, usually use floating point
  - May use fixed point if speed/simplicity is more important than accuracy
- Use 64-bit integers when you need values > 2 billion
- Floating point rules of thumb:
  - Single precision gives ~7 decimal digits of precision, max of $\sim 10^{38}$
  - Double precision gives ~16 decimal digits of precision, max of $\sim 10^{308}$

Summary

- Data storage
  - All data are stored in bits
  - Different storage device
  - RAID for reliability – only need to understand the concept
- Data representation
  - Binary format for integers
  - 2’s complement
  - Additions of integers
  - Floating point – do not require conversion or calculation