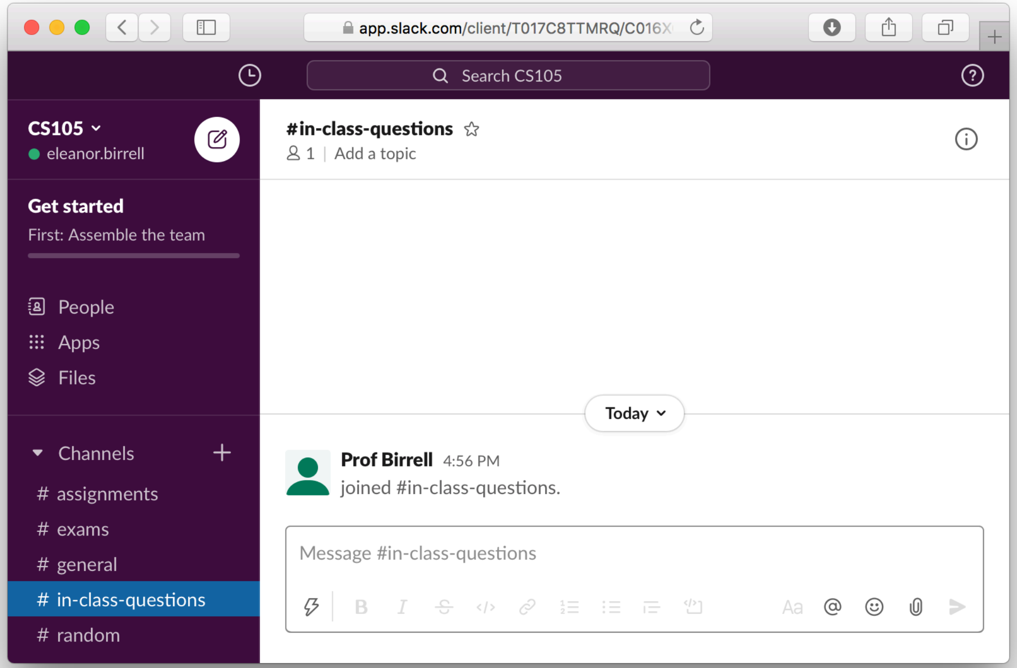
Lecture 1: Introduction to Computer Systems

CS 105 Fall 2020

Exercise 0: Introductions

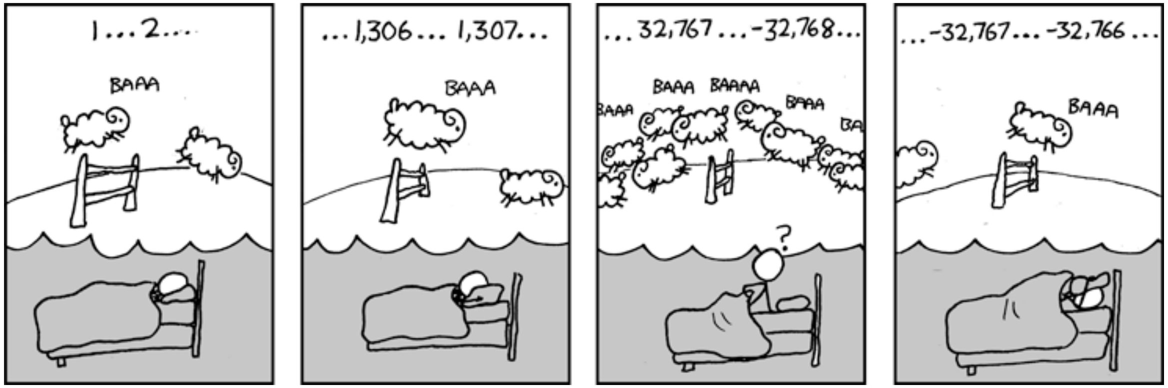
- Go onto the CS 105 slack (cs105-workspace.slack.com) and introduce yourself
- Notice that there is a channel called #in-class-questions. If you have questions while watching the lecture videos, post them there!



Abstraction

Correctness

- **Example 1: Is $x^2 \geq 0$?**
  - Floats: Yes!



  - Ints:
    - 40000 * 40000 → 1600000000
    - 50000 * 50000 → ??
- **Example 2: Is (x + y) + z = x + (y + z)?**
  - Ints: Yes!
  - Floats:
    - (2^30 + -2^30) + 3.14 → 3.14
    - 2^30 + (-2^30 + 3.14) → ??

Performance

```c
void copyij(int src[2048][2048],
            int dst[2048][2048]){
  int i,j;
  for (i = 0; i < 2048; i++){
    for (j = 0; j < 2048; j++){
      dst[i][j] = src[i][j];
    }
  }
}
```

4.3ms

```c
void copyji(int src[2048][2048],
            int dst[2048][2048]){
  int i,j;
  for (j = 0; j < 2048; j++){
    for (i = 0; i < 2048; i++){
      dst[i][j] = src[i][j];
    }
  }
}
```

81.8ms

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array

Security

```
void admin_stuff(int authenticated){
  if(authenticated){
    // do admin stuff
  }
}

int dontTryThisAtHome(char * user_input, int size) {
  char data[size];
  int ret = memcpy(*user_input, data);
  return ret;
}
```

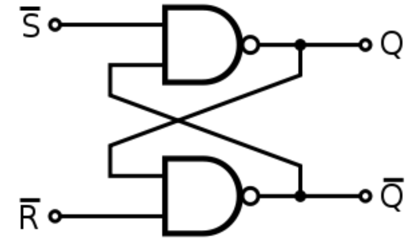
Bits

- a **bit** is a binary digit that can have two possible values

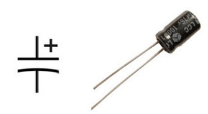
- can be physically represented with a two state device

Storing bits

- Static random access memory (SRAM): stores each bit of data in a flip-flop, a circuit with two stable states



- Dynamic Memory (DRAM): stores each bit of data in a capacitor, which stores energy in an electric field (or not)



- Magnetic Disk: regions of the platter are magnetized with either N-S polarity or S-N polarity

- Optical Disk: stores bits as tiny indentations (pits) or not (lands) that reflect light differently

- Flash Disk: electrons are stored in one of two gates separated by oxide layers

Boolean Algebra

- Developed by George Boole in 19th Century
- Algebraic representation of logic---encode "True" as 1 and "False" as 0

And

&	0	1
0	0	0
1	0	1

Or

|	0	1
0	0	1
1	1	1

Not

~	
0	1
1	0

Exclusive-Or (Xor)

^	0	1
0	0	1
1	1	0

Exercise 1: Boolean Operations

- Evaluate each of the following expressions
  1. `1 | (~1)`
  2. `~( 1 | 1)`
  3. `(~1) & 1`
  4. `~( 1 ^ 1)`

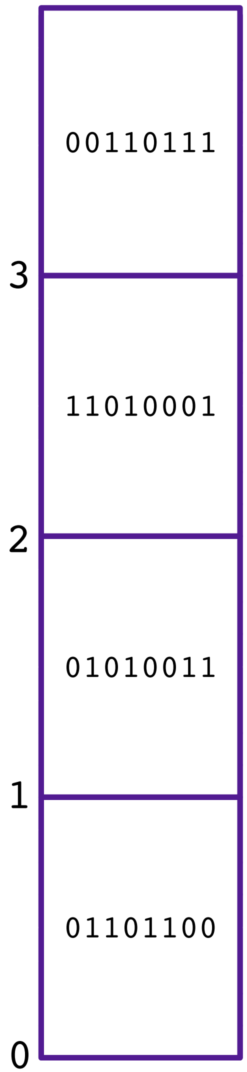
Exercise 1: Boolean Operations

- Evaluate each of the following expressions

```
1. 1 | (~1)      = 1 | 0 = 1
2. ~( 1 | 1)     = ~1    = 0
3. (~1) & 1      = 0 & 1 = 0
4. ~( 1 ^ 1)     = ~0    = 1
```

Bytes and Memory

- **Memory** is an array of ~~bits~~ bytes
- A **byte** is a unit of eight bits
- An index into the array is an **address**, **location**, or **pointer**
  - Often expressed in hexadecimal
- We speak of the *value* in memory at an address
  - The value may be a single byte …
  - … or a multi-byte quantity starting at that address



General Boolean algebras

- Bitwise operations on bytes

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
  01000001      01111101      00111100      10101010
```

- How does this map to set operations?

Exercise 2 : Bitwise Operations

- Assume: `a = 01101100, b = 10101010`

- What are the results of evaluating the following Boolean operations?

  - `~a`
  - `~b`
  - `a & b`
  - `a | b`
  - `a ^ b`

Exercise 2 : Bitwise Operations

- Assume: `a = 01101100, b = 10101010`

- What are the results of evaluating the following Boolean operations?

  - `~a      = ~01101100 = 10010011`
  - `~b      = ~10101010 = 01010101`
  - `a & b  =  01101100 & 10101010 = 00101000`
  - `a | b  =  01101100 | 10101010 = 11101110`
  - `a ^ b  =  01101100 ^ 10101010 = 11000110`

Bitwise vs Logical Operations in C

- Bitwise Operators &, |, ~, ^
  - View arguments as bit vectors
  - operations applied bit-wise in parallel
- Logical Operators &&, ||, !
  - View 0 as “False”
  - View anything nonzero as “True”
  - Always return 0 or 1
  - Early termination

Exercise 3: Bitwise vs Logical Operations

- `~01000001`
- `~00000000`
- `~~01000001`

- `!01000001`
- `!00000000`
- `!!01000001`

- `01101001 & 01010101`
- `01101001 | 01010101`

- `01101001 && 01010101`
- `01101001 || 01010101`

Exercise 3: Bitwise vs Logical Operations

- `~01000001` 10111110
- `~00000000` 11111111
- `~~01000001` 01000001

- `!01000001` 00000000
- `!00000000` 00000001
- `!!01000001` 00000000

- `01101001 & 01010101` 01000001
- `01101001 | 01010101` 01111101

- `01101001 && 01010101` 00000001
- `01101001 || 01010101` 00000001

Bit Shifting

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
  - Throw away extra bits on left
  - Fill with 0's on right

Undefined Behavior if you shift amount < 0 or ≥ word size

- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
  - Throw away extra bits on right
  - Logical shift: Fill with 0's on left
  - Arithmetic shift: Replicate most significant bit on left

Choice between logical and arithmetic depends on the type of data

Example: Bit Shifting

- 01101001 << 4 10010000
- 01101001 $>>_l$ 4 00000101
- 01101001 $>>_a$ 4 00000101

Exercise 4: Bit Shifting

- `10101010 << 4` `10100000`
- `10101010` $>>_l$ `4` `00001010`
- `10101010` $>>_a$ `4` `11111010`

Bits and Bytes Require Interpretation

00000000 00110101 00110000 00110001

might be interpreted as

- The integer $3{,}485{,}745_{10}$
- A floating point number close to 4.884569×10^{-39}
- The string “105”
- A portion of an image or video
- An address in memory

Information is Bits + Context

Exercise 5: Feedback

1. Rate how well you think this recorded lecture worked
    1. Better than an in-person class
    2. About as well as an in-person class
    3. Less well than an in-person class, but you still learned something
    4. Total waste of time, you didn't learn anything

2. How much time did you spend on this video lecture (including time spent on exercises)?

3. Do you have any comments or feedback?