

Lecture 01

Introduction

CS213 – Intro to Computer Systems
Branden Ghena – Spring 2021

Slides adapted from:

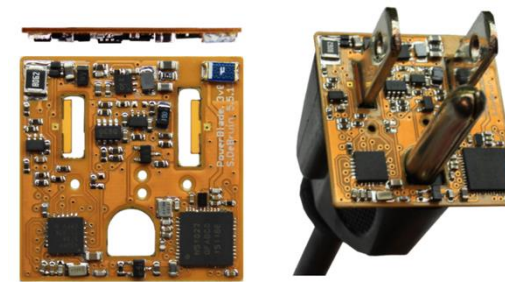
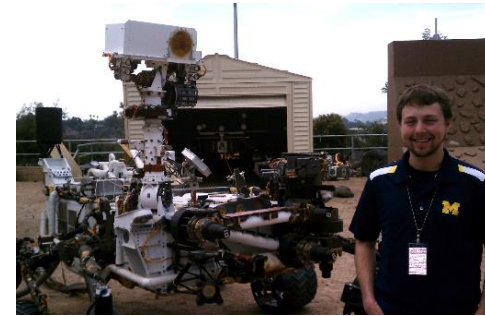
St-Amour, Hardavellas, Bustamante (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Welcome to CS213!

- In brief: How *does* a computer work anyway?
- We will explore that question across four major sections:
 - **Representations** of information on a computer
 - How the **machine** executes software
 - How **memory** is organized
 - How the **operating system** manages this all for efficiency and security

Branden Ghena (he/him)

- Assistant Faculty of Instruction
- Education
 - Undergrad: Michigan Tech
 - Master's: University of Michigan
 - PhD: University of California, Berkeley
- Research
 - Resource-constrained sensing systems
 - Low-energy wireless networks
 - Embedded operating systems
- Teaching
 - Computer Systems
 - Fundamentals of Computer Programming II
 - Operating Systems
 - Microprocessor System Design
 - Wireless Protocols for the IoT



Things I love



Today's Goals

- Introduce the theme and goals of the course
- Describe how this class is going to function
- Discuss how a computer system works at a high level
- Begin exploring how computers represent information with bits and bytes

Outline

- **Course Themes**
- Logistics
- Running a program
- Representing numbers with binary

Convenient computing

- Computers operate on integers, reals, structs, arrays, etc.
- Computers operate on variables and functions
- Computers execute conditionals, loops, etc.
- Memory is an infinite bag of objects my program can allocate
- Memory doesn't have to be shared with any other program
- Memory is always equivalently fast to access
- Etc.

Convenient **illusions** in computing

- Computers operate on integers, reals, structs, arrays, etc.
- Computers operate on variables and functions
- Computers execute conditionals, loops, etc.
- Memory is an infinite bag of objects my program can allocate
- Memory doesn't have to be shared with any other program
- Memory is always equivalently fast to access
- Etc.

- None of these are actually true!
 - But we usually program as if they were, and we get away with it!
 - What's going on?

The power of abstraction

- These illusions are really ***abstractions***
- They approximate reality, but leave out details
 - Instead, they provide an *interface* that we can work and think with
- We can forget about those details, and be more productive
- Abstractions we love
 - Abstract data types
 - Asymptotic analysis
 - High-level programming languages
 - Operating systems
 - Etc.

The Limits of Abstraction

- Sometimes, abstractions break down
 - Their implementation is buggy
 - Mismatch between expected interface and implementation
 - Their performance is inadequate
 - We need control over the details they hide
 - Security concerns make these details important
- At that point, details come rushing back
 - Can't pretend they don't exist anymore
 - We must know how to deal with them
- This class is about being ready when that happens

Expectation/Implementation Mismatch

- Ariane 5 explosion (1996)
 - Inertial reference system converted a 64-bit float to a 16-bit integer
 - Had worked in the past in Ariane 4, but Ariane 5 was faster
 - Speed too large to fit in a 16-bit integer -> software fault
 - Expectation: inertial reference system could handle any rocket
 - Reality: guidance system faults when traveling at supersonic speeds



Inadequate performance

- Abstracted lower-level details can affect performance a lot!
- **Web accelerators:** Squid vs Varnish
 - Varnish is designed to take advantage of virtual memory, Squid is not
 - Squid needed 12 servers running at 100% CPU usage
 - Varnish needed 3 servers running at 10% CPU usage for the same load
 - <http://queue.acm.org/detail.cfm?id=1814327>
- **Cache friendliness:** latter is 10-32 times slower on Intel systems

```
void copyij(int src[4096][4096],
            int dst[4096][4096])
{
    int i,j;
    for (i = 0; i < 4096; i++)
        for (j = 0; j < 4096; j++)
            dst[i][j] = src[i][j];
}
```



```
void copyji(int src[4096][4096],
            int dst[4096][4096])
{
    int i,j;
    for (j = 0; j < 4096; j++)
        for (i = 0; i < 4096; i++)
            dst[i][j] = src[i][j];
}
```

Security concerns

- Recent example: Meltdown and Spectre (2018)
 - <https://meltdownattack.com/>
 - Speculative execution on processors allows code to run before checking if it *should* run
 - Cache timing attacks can tell if some value was recently loaded from memory
 - **Combination:** attacker can read memory that should be protected
 - Huge vulnerability that was actually easy to understand
 - But no one had realized it existed!



CS213 goals

1. Break through abstractions to understand how computer processors and memories affect software design and performance.
2. Introduce concepts of “computer systems” areas:
 - Architecture, Compilers, Security, Embedded, Operating Systems, etc.

Course design goal

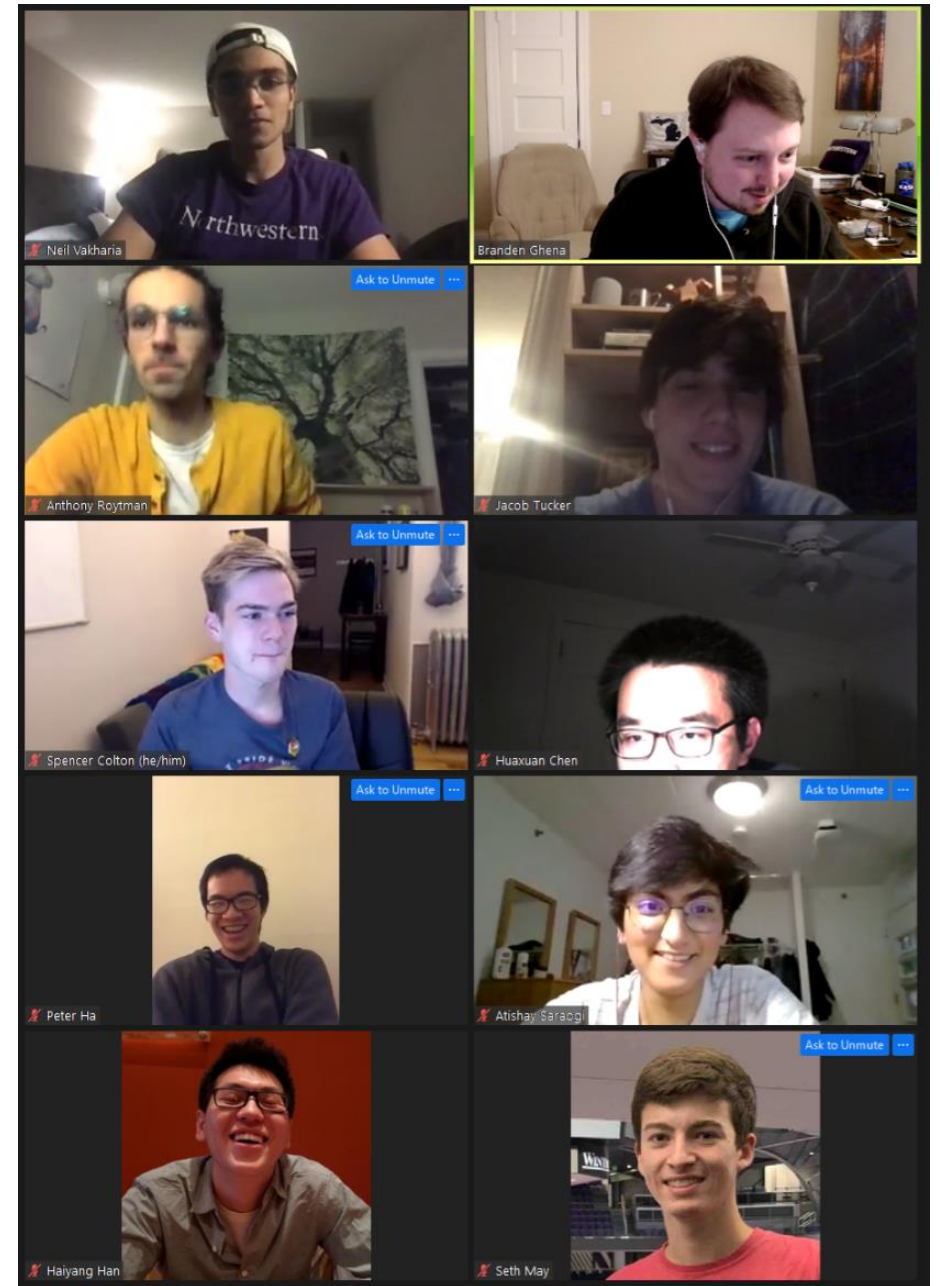
- Most systems courses are builder-centric
 - **Computer Architecture:** design a pipelined processor in Verilog
 - **Operating Systems:** implement portions of an operating system
 - **Compilers:** write a compiler for a simple language
 - **Networking:** Implement and simulate network protocols
 - Fun, for sure
 - But ultimately, many more of you will ***build on*** systems
 - Rather than ***build systems*** directly
- This course is programmer-centric
 - Purpose is to show that by knowing more about the underlying system, one can be more effective as a programmer
 - Not just a course for dedicated hackers
 - **We bring out the hacker in everyone!**

Outline

- Course Themes
- **Logistics**
- Running a program
- Representing numbers with binary

Course Staff

- TA: Drake Han
 - PhD student in Computer Engineering
- PMs:
 - Huaxuan Chen, Seth May,
Neil Vakharia, Spencer Colton,
Jacob Tucker, Anthony Roytman,
Atishay Saraogi, Peter Ha



Course details

- Lectures: synchronous, recorded via Zoom
 - Please attend and ask questions!
 - Panopto tab on Canvas will have recordings (a few hours later)
- Office hours: (start next week)
 - Via gather.town
 - More info will be posted to Campuswire when schedule is ready
 - Scheduling a wide range of hours to work for everyone
 - Can reach out on Campuswire to schedule a meeting too
- Textbook:
 - Computer Systems: A Programmer's Perspective **3rd Edition**
 - A *very* useful reference

Asking questions

- Class and office hours are always an option!
- Campuswire: (similar to piazza)
 - Post questions
 - Answer each other's questions
 - Find lab partners
 - Find posts from the course staff
 - Post private info just to course staff
- Please do not email me! Post to Campuswire instead!
 - I'll be updating roster again a few times

Programming Labs

- Four labs
 - Data Lab – manipulate bits and bytes
 - Bomb Lab – deconstruct software to understand it
 - Attack Lab – exploit security vulnerabilities in software
 - SETI Lab – make software faster with concurrency
- Work on these preferably as a group of two
 - Work together and don't split up assignments (otherwise you won't learn)
 - Individual is acceptable but less good
- Very different from CS211 style projects
 - Emphasis on the thinking rather than the programming

Grades

- Grade breakdown
 - 50% Programming Labs (4 labs at 12.5% each)
 - 20% Homeworks (4 homeworks at 5% each)
 - 15% Midterm Exam 1
 - 15% Midterm Exam 2
- Exact number to letter mapping is flexible
 - But this course is not curved
- Exams will be synchronous during class time
 - With an alternate time for students in opposite timezones

Academic Integrity

- This is something I take very seriously
- Collaboration good; plagiarism bad
 - You should know where that line is, and be nowhere near it
 - When in doubt, ask the instructor *before* you do something you're not sure about
- At no point should you see someone else's solutions
 - Not your colleagues', not your friends', not your cousin's, not something you found online
- I report everything suspicious to the dean

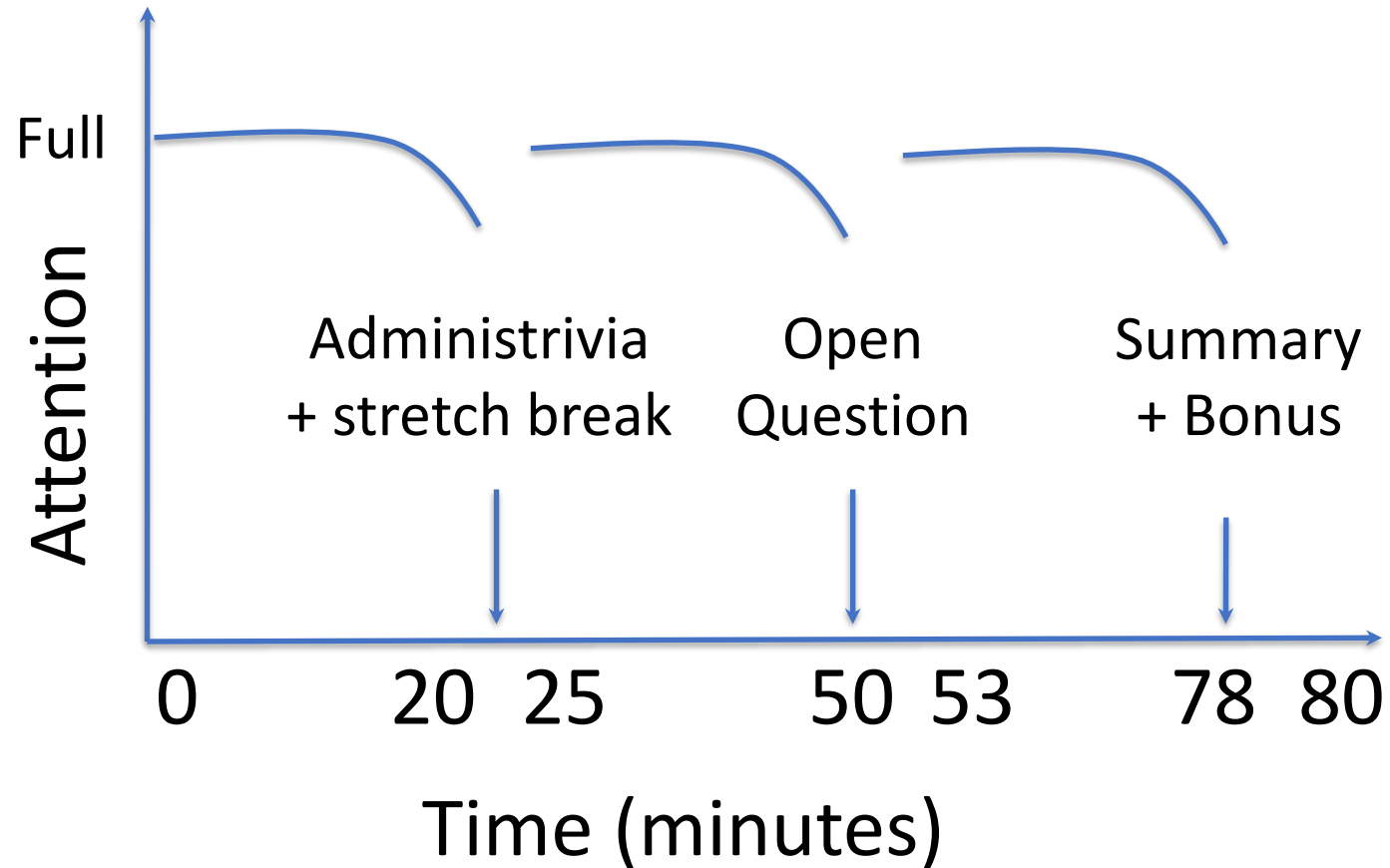
Expectations

- This class is **hard**
 - And it's hard in a different way. So much new material that interacts
 - Opportunity to learn a lot from it
- I'm confident that you can all succeed
 - Every CS student has to pass through this course
 - Labs, Homeworks, Lecture, Office Hours all designed to support you
- You'll gain a much deeper understanding of how computers operate
 - Maybe it's not for you, maybe you'll love it

How to succeed in this class

- Do the readings
- Come to lecture
- Ask questions
- Solve practice problems in the textbook
- Start assignments early
- Stay on top of the material

Architecture of a lecture



Outline

- Course Themes
- Logistics
- **Running a program**
- Representing numbers with binary

Hello World

- What happens when you run "hello" on your system?
 - And *why* does it happen?

```
/*  
 * hello world  
 */  
#include <stdio.h>  
  
int main()  
{  
    printf("hello, world\n");  
}
```

- **Goal:** introduce key concepts, terminology, and components

Compiling `hello`

- Compiling `hello`

```
unix> gcc -o hello hello.c
```

- GCC is our compiler
- It takes our source code (`hello.c`)
 - A text file containing characters
 - Text file = readable by humans
- And translates (compiles) it into **assembly code**
 - A text representation of x86 instructions
 - Here, not explicitly stored in a file
 - We'll be working with assembly a lot this quarter
- Then translates (assembles) that into an executable (`hello`)
 - A binary file containing x86 machine code
 - Binary file = not meant to be read by humans (but sometimes we have to)

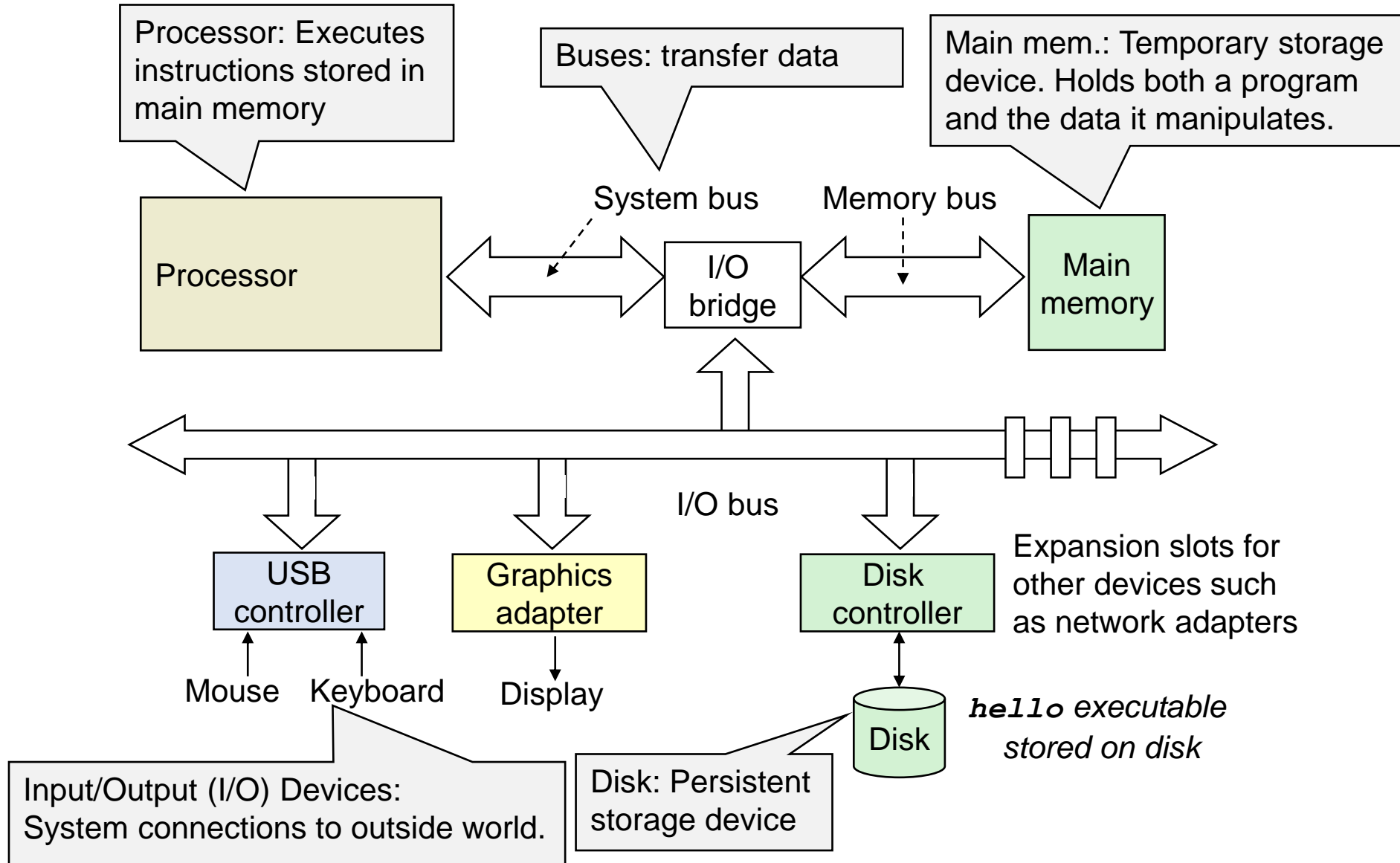
Running `hello`

- Running `hello`

```
unix> ./hello  
hello, world  
unix>
```

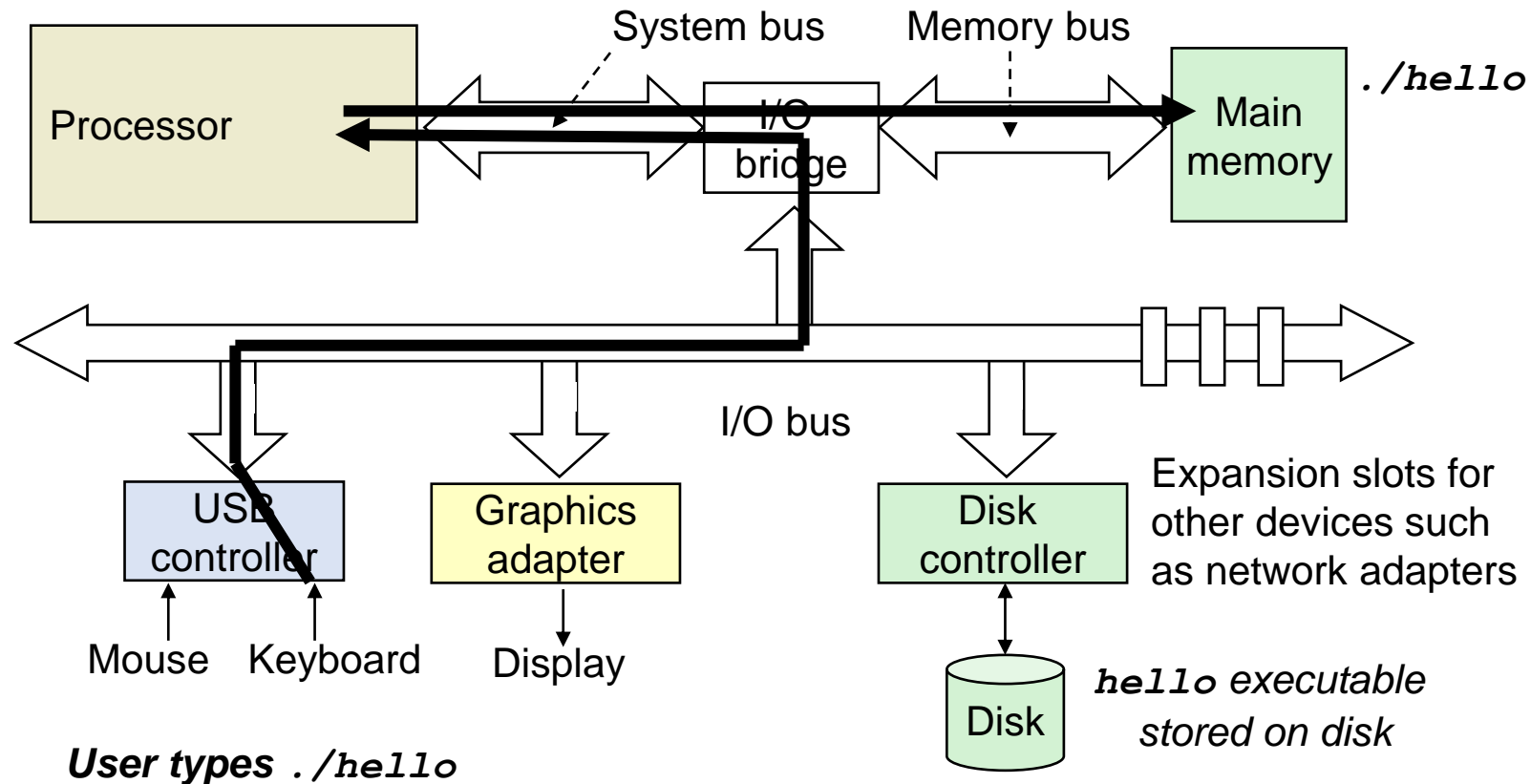
- What does the shell do?
 - Prints a prompt
 - Waits for you to type a command
 - Then loads and runs the `hello` program
- What happens at the hardware level?

Hardware organization



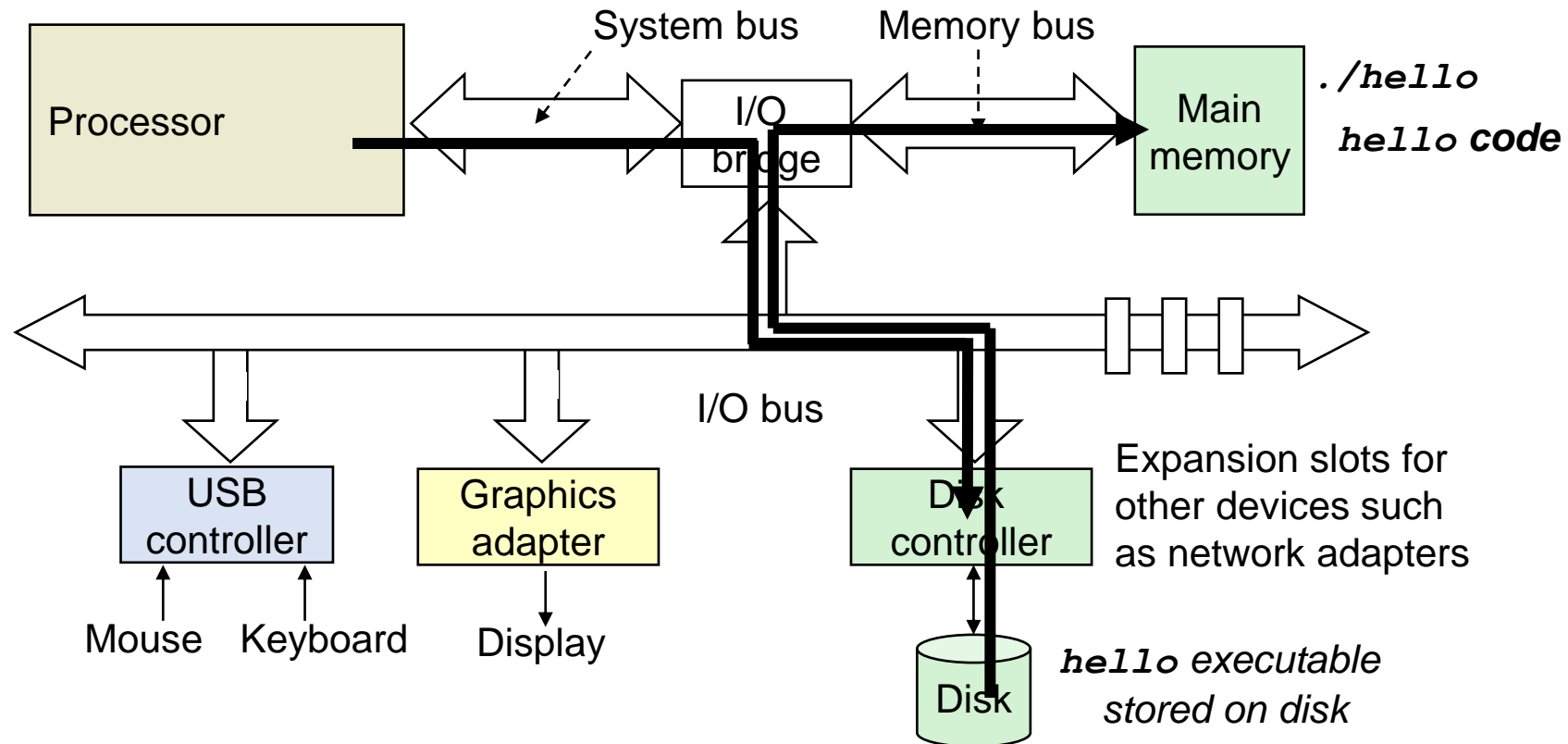
Running `hello`

Reading the `./hello` command from the keyboard



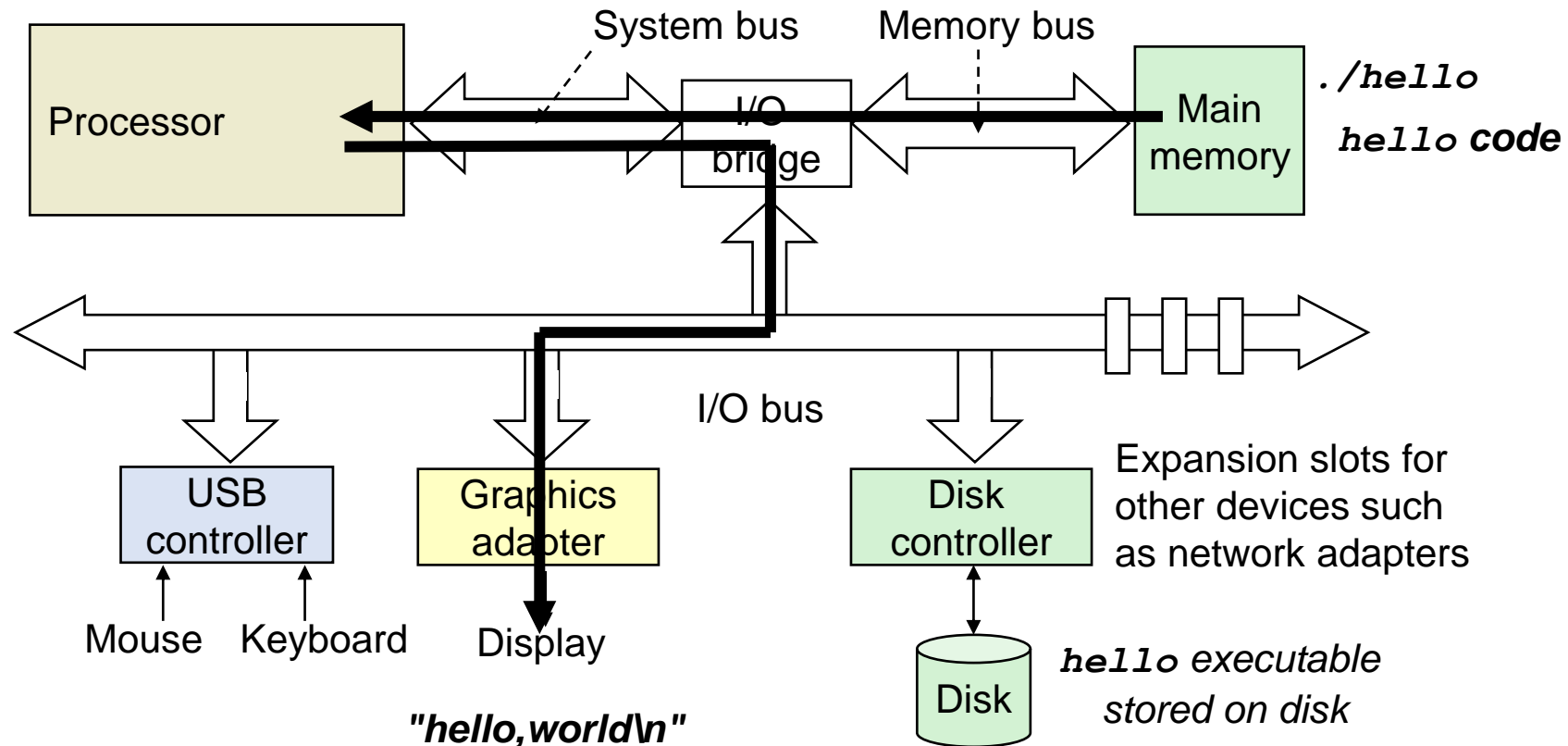
Running `hello`

Shell program loads the `hello` executable into main memory



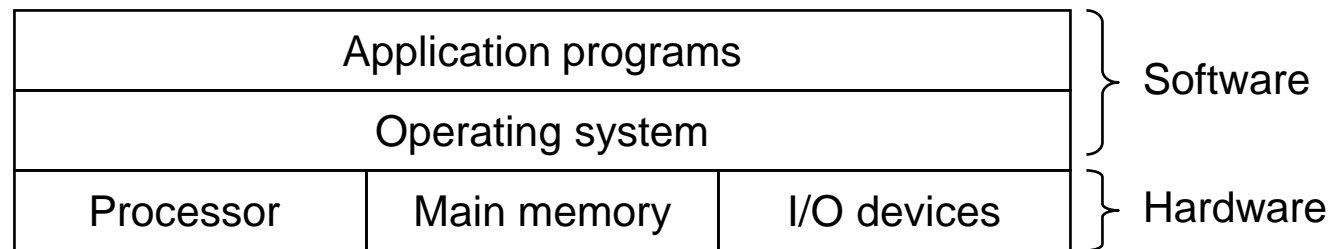
Running `hello`

The processor reads the `hello` code, executes instructions, and displays “hello...”



Operating system

- Neither `hello` nor our shell interfaced with the hardware directly
 - All interactions were mediated by the ***operating system***
- ***Operating system***: a layer of software interposed between the application program and the hardware



- Primary goals
 - Protect resources from misuse by applications
 - Provide simple and uniform mechanisms for manipulating hardware devices
 - Manage sharing of resources between applications

A computer system is more than just HW

- A collection of intertwined hardware and software that must cooperate to achieve the end goal – running applications
 - **Hardware:** expensive, fast, immutable
 - **Software:** cheap (comparatively), flexible, easily changed
 - Different tradeoffs
 - So we'll use them for different roles!
- The rest of the course will expand on this

Open Question + Break

- **What part of the `hello` example takes the longest to run on a computer?**

Open Question + Break

- **What part of the hello example takes the longest to run on a computer?**
 - The user typing (seconds)
 - Maybe that's cheating and we should start after they hit enter

Open Question + Break

- **What part of the hello example takes the longest to run on a computer?**
 - The user typing (seconds)
 - Maybe that's cheating and we should start after they hit enter
 - Almost certainly loading the program from disk (milliseconds)
 - Possibly sending text to graphics (microseconds – milliseconds)
 - Definitely not executing the code (nanoseconds – microseconds)

Outline

- Course Themes
- Logistics
- Running a program
- **Representing numbers with binary**

Positional Numbering Systems

- The position of a *numeral* (e.g., digit) determines its contribution to the overall number
 - Makes arithmetic simple (compared to, say, roman numerals)
 - Any number has one canonical representation
- Example: base 10
 - $10456_{10} = 1*10^4 + 0*10^3 + 4*10^2 + 5*10^1 + 6*10^0$
- Other bases are also possible
 - Base 2: $10010010_2 = 1*2^7 + 1*2^4 + 1*2^1 = 146_{10}$
 - Base 60, used by the Babylonians
 - The source of 60 seconds in a minute, 60 minutes in an hour
 - And 360 degrees in a circle
 - Base 20, used by the Maya and Gauls (bits remain in French today)

Base 2 Example

- We'll use base 2 a ***LOT***
- Let's convert 134_{10} to base 2
- We need to decompose 134_{10} into a sum of powers of 2
 - Start with the largest power of 2 that is smaller or equal to 134_{10}
 - Subtract it, then repeat the process

$$134_{10} - 128_{10} = 6_{10}$$

$$6_{10} - 4_{10} = 2_{10}$$

$$2_{10} - 2_{10} = 0_{10}$$

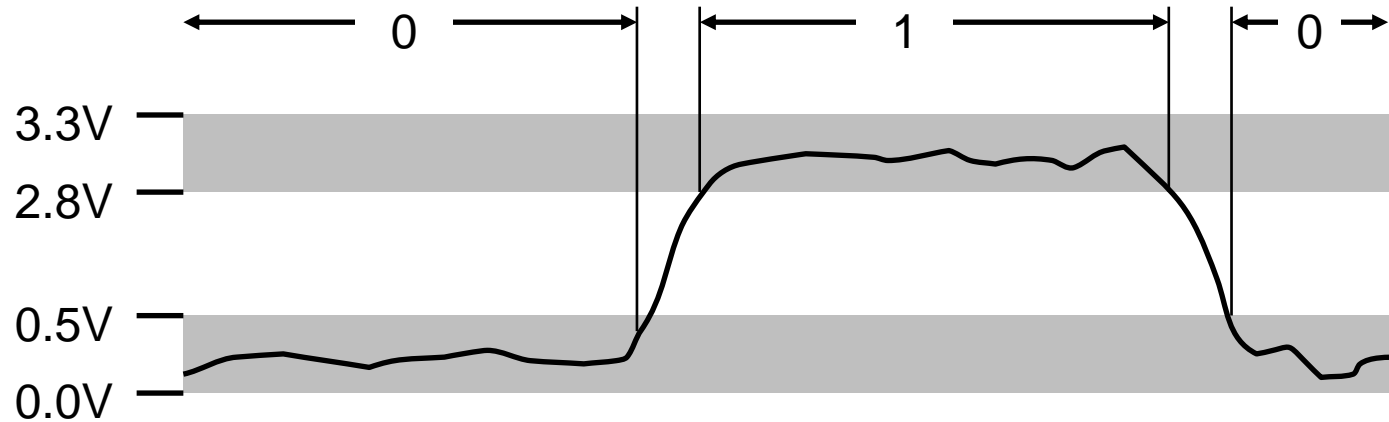
$$134_{10} = \underline{1} \times 128 + 0 \times 64 + 0 \times 32 + 0 \times 16 + 0 \times 8 + \underline{1} \times 4 + \underline{1} \times 2 + 0 \times 1$$

$$134_{10} = \underline{1} \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + \underline{1} \times 2^2 + \underline{1} \times 2^1 + 0 \times 2^0$$

$$134_{10} = 10000110_2$$

Why computers use Base 2

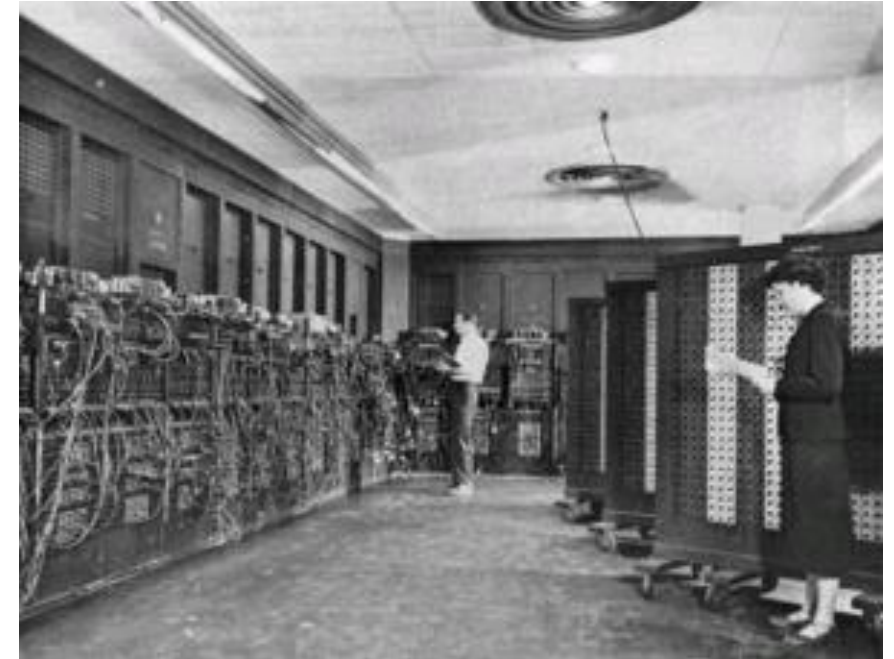
- Simple electronic implementation
 - Easy to store with bi-stable elements
 - Reliably transmitted on noisy and inaccurate wires



- Straightforward implementation of arithmetic functions
- (Pretty much) all computers use base 2

Why don't computers use Base 10?

- Because implementing it electronically is a pain
 - Hard to store
 - ENIAC (first general-purpose electronic computer) used 10 vacuum tubes / digit
 - Hard to transmit
 - Need high precision to encode 10 signal levels on single wire
 - Messy to implement digital logic functions
 - Addition, multiplication, etc.



Base 16: Hexadecimal

- Writing long sequences of 0s and 1s is tedious and error-prone
 - And takes up a lot of space on a page!
- So we'll often use base 16 (also called *hexadecimal*)
- $16 = 2^4$, so every group of 4 bits becomes a hexadecimal digit (or *hexit*)
 - If we have a number of bits not divisible by 4, add 0s on the left (always ok, just like base 10)
- Base 2 = 2 symbols (0, 1)
Base 10 = 10 symbols (0-9)
Base 16, need 16 symbols
 - Use letters A-F once we run out of decimal digits

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

0 0 1 0 1 0 0 1 0 1 1 1 1 0 1 1 \rightarrow 0x297B

“0x” prefix = it’s in hex

Bytes

- A single bit doesn't hold much information
 - Only two possible values: 0 and 1
 - So we'll typically work with larger groups of bits
- For convenience, we'll refer to groups of 8 bits as ***bytes***
 - And usually work with multiples of 8 bits at a time
 - Conveniently, 8 bits = 2 hexits
- Some examples
 - 1 byte: $0b01100111 = 0x67$
 - 2 bytes: $11000100\ 00101111_2 = 0xC42F$

"0b" prefix = it's in binary

Practice problem

- **Convert 0x42 to decimal**

- Steps

- Convert 0x42 to binary:

- Convert binary to decimal:

Practice problem

- **Convert 0x42 to decimal**

- Steps

- Convert 0x42 to binary:

- 0x4 -> 0b0100 0x2 -> 0b0010 0x42 -> 0b 0100 0010

- Convert binary to decimal:

- $1*2^6 + 1*2^1 = 64 + 2 = 66$

Practice problem

- **Convert 0x42 to decimal**
- Critical thinking:
 - What are the maximum and minimum values?
 - Minimum 0
 - Maximum 255
 - How big is 0x42 out of 0xFF?
 - ~25% (0x40, 0x80, 0xC0, 0x100)
 - So $255/4 \approx 240/4 \approx 60$

Big idea: bits can be used to represent anything

- Depending on the context, the bits `11000011` could mean
 - The number 195
 - The number -61
 - The number -1.1875
 - The value True
 - The character `'|'`
 - The `ret` x86 instruction
- You have to know the **context** to make sense of any bits you have!
 - People and software they write determine what the bits actually mean

Outline

- Course Themes
- Logistics
- Running a program
- Representing numbers with binary

Outline

- Backup: Boolean Algebra

Boolean Algebra

- You've programmed with **and** and **or** in earlier classes
 - Written **&&** and **||** in C and C++
- **Boolean algebra is a generalization of that**
 - A mathematical system to represent (propositional) logic
 - 2 truth values: true = **1**, false = **0**
 - 3 operations: and = **&**, or = **|**, not (or complement) = **~**
- **Follow the rules for each operation to compute results**
 - Rules are the like those you know from programming
$$(1 \mid 0) \& 0 \longrightarrow 1 \& 0 \longrightarrow 0$$
$$(1 \& 1) \& \sim(0 \mid 0) \longrightarrow 1 \& \sim(0) \longrightarrow 1 \& 1 \longrightarrow 1$$

Truth Tables for Boolean Algebra

- For each possible value of each input, what is the output
 - Axes are the inputs
 - Inside of the table are the outputs

not: $\sim A$

\sim	
0	1
1	0

and: $A \& B$

$\&$	0	1
0	0	0
1	0	1

or: $A \mid B$

\mid	0	1
0	0	1
1	1	1

De Morgan's Laws, Exclusive Or

- Can express boolean operators in terms of the others
- De Morgan's laws: & using | and ~, | using & and ~
 - $A \& B = \sim(\sim A \mid \sim B)$
 - A and B are true if and only if neither A nor B is false
 - $A \mid B = \sim(\sim A \& \sim B)$
 - A or B are true if and only if A and B are not both false
- Can define new operators in terms of existing ones:
 - Exclusive or (xor, ^) in terms of inclusive or (|)
 - $A \wedge B = (\sim A \& B) \mid (A \& \sim B)$
 - Exactly one of A and B is true
 - $A \wedge B = (A \mid B) \& \sim(A \& B)$
 - Either A is true, or B is true, but not both
 - The two definitions are equivalent

xor: $A \wedge B$

\wedge	0	1
0	0	1
1	1	0

Generalized Boolean Algebra

- Binary bits can represent truth values: 0 = false, 1 = true
- Boolean operations can be extended to work on vectors of bits
 - Operations applied one bit at a time: ***bitwise***

01101001	01101001	01101001	01101001
& 01010101	01010101	^ 01010101	~ 01010101
<hr/>	<hr/>	<hr/>	<hr/>
01000001	01111101	00111100	10101010

- All of the properties of Boolean algebra apply
 - Relationships between operations, etc.

Bit-level operations in C

- Operations `&`, `|`, `~`, `^` available in C
 - Apply to any “integral” data type
 - `long`, `int`, `short`, `char`, `unsigned`
 - View arguments as bit vectors
 - Arguments applied bit-wise
- Examples (char data type, single byte)
 - **`~0x00 → 0xFF`**
 $\sim 00000000_2 \rightarrow 11111111_2$
 - **`~0x41 → 0xBE`**
 $\sim 01000001_2 \rightarrow 10111110_2$
 - **`0x69 | 0x55 → 0x7D`**
 $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Logic operations in C – not the same!

- Logical operations `||`, `&&` and `!` (Logic OR, AND & Not)
 - Contrast to bit-wise operators
 - View 0 as “False”
 - *View anything nonzero as “True”*
 - Always return 0 or 1 (i.e., false or true) rather than a sequence of bits
 - Early termination (*if you can answer by just looking at the first argument, you are done*)
- Examples (char data type)
 - `!0x41` \rightarrow `0x00`
 - `!0x00` \rightarrow `0x01`
 - `!!0x41` \rightarrow `0x01`
 - `0x59 && 0x35` \rightarrow `0x01`
 - `0x59 || 0x35` \rightarrow `0x01`
 - `p && *p` (avoids null pointer dereference)

Watch out for `&&` vs. `&`
(and `||` vs. `|`) ...
one of the more
common slip-ups in
C programming