The Big Ideas of CS 233
An Introduction to CS233
Class Mechanics on one Slide

- **Lectures**: bring pen/pencil + iclicker
  - See wiki for video lectures
- **Section/Lab**: bring pen/pencil, short quiz, start on Lab
- **Piazza**: how to ask questions (use good etiquette)
- **Web Homworks**: after every lecture in the beginning
  - Done individually
- **Labs**: due weekly on Sunday nights
  - Can be done in groups (up to 2). Don’t share code across groups.
- **Exams**: See the wiki
  - Second chance testing (read course policy)
- **Office hours**: normal deal
233 in one slide!

- The class consists roughly of 4 quarters: (Bolded words are the big ideas of the course, pay attention when you hear these words)
  1. You will build a simple computer processor
     Build and create **state** machines with **data**, **control**, and **indirection**
  2. You will learn how high-level language code executes on a processor
     Time limitations create **dependencies** in the **state** of the processor
  3. You will learn why computers perform the way they do
     Physical limitations require **locality** and **indirection** in how we access **state**
  4. You will learn about hardware mechanisms for parallelism
     **Locality**, **dependencies**, and **indirection** on performance enhancing drugs

- We will have a SPIMbot contest!
A computer can do 2 things: Store state...
State is the relevant information about the progress of my system.
A computer can do 2 things: …and manipulate state
Computation changes my state in a limited number of ways
State changes can respond to user (system) inputs
State is used to compute a system output
This game can be modeled with 3 system outputs: “game in progress,” “blue won,” “orange won”
The state abstraction informs how we think about code tracing

- The system clock constrains when each line of code executes
- Code executes in series

```
z = x + y;
x = 1;
if(x == z)
    y = 2;
```

Diagram:

- State Storage
- State Manipulations
You have seen state in three forms in your coding: Data, control, and indirection

Data

```c
int add_numbers(int x, int y){
    int z;

    z = x + y;
    return z;
}
```

Control

```c
int find_greater(int x, int y){
    if (x > y)
        return x;
    else
        return y;
}
```

Indirection (Address)

```c
int find_data(int* x){
    int y;

    y = *x;
    return y;
}
```
Boolean Algebra and Its Relation to Gates

Why you needed to take CS 173
Today’s lecture

- Basic Boolean expressions
  - Booleans
  - AND, OR and NOT
State information is encoded with 1s and 0s

Circuit

Volts
1.8

In

Out

Volts
0

Boolean

Volts
1.8

True

False

0

Binary

1

0
Boolean functions

- Just like in other mathematics, we can define functions:

\[ y = f(x) \]

- There are a finite number (2) of boolean values...
  - There are a finite number of boolean functions
  - Let’s discuss with an example
A 1-input Boolean function has 4 unique output functions

\[ y = f(x) \]

- A 1-input Boolean function has \(2^1 = 2\) possible input combinations:
- There are \(2^{(# \text{ of input combinations})}\) possible unique functions
  - For each input value, there are 2 possible output values (0 or 1)
  - The value of each output is independent from the value of each input
- The 4 possible 1-input Boolean functions

<table>
<thead>
<tr>
<th>x</th>
<th>f_0(x)</th>
<th>x</th>
<th>f_1(x)</th>
<th>x</th>
<th>f_2(x)</th>
<th>x</th>
<th>f_3(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
A 2-input Boolean function has 16 unique output functions

\[ z = f(x, y) \]

- 4 possible input combinations, 16 possible functions:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
<th>f8</th>
<th>f9</th>
<th>f10</th>
<th>f11</th>
<th>f12</th>
<th>f13</th>
<th>f14</th>
<th>f15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- We’ll focus on 2 functions for now
We use three basic logical operations: AND, OR, and NOT

<table>
<thead>
<tr>
<th>Operation:</th>
<th>AND (product) of two inputs</th>
<th>OR (sum) of two inputs</th>
<th>NOT (complement) on one input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>$xy$, or $x \cdot y$</td>
<td>$x + y$</td>
<td>$x'$ or $\overline{x}$</td>
</tr>
<tr>
<td>Notation:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Truth table: | | | |
|--------------|--------------|--------------|
| $x$ | $y$ | $f(x)$ | $x$ | $y$ | $f(x)$ | $x$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

These are sufficient to implement any Boolean function
Boolean expressions (formally)

- Use these basic operations to form more complex expressions:
  
  \[ f(x,y,z) = (x + y')z + x' \]

- Some terminology and notation:
  - \( f \) is the name of the function.
  - \((x,y,z)\) are the input variables, each representing 1 or 0. Listing the inputs is optional, but sometimes helpful.
  - A literal is any occurrence of an input variable or complement. The function above has four literals: \( x, y', z, \) and \( x' \).

- Precedences are similar to what you learned from algebra
  - NOT has the highest precedence, followed by AND, and then OR.
  - Fully parenthesized, the function above would be kind of messy:

  \[ f(x,y,z) = (((x+(y'))z) + x') \]
A quick reminder

Expressions → Truth Tables

Gates (Schematics) ↔ HDLs (Verilog)
To compute a truth table given a Boolean expression:
- Evaluate the function for every combination of inputs.

\[ f(x, y, z) = (x + y')z + x' \]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>f(x, y, z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ f(0, 0, 0) = (___ + ___)___ + ___ = ___ \]
A quick reminder
The Boolean operators map to three primitive logic gates

Operation:  | AND (product) of two inputs | OR (sum) of two inputs | NOT (complement) on one input
--- | --- | --- | ---
Expression:  | $xy$, or $x \cdot y$ | $x + y$ | $x'$
Logic gate:  | ![AND gate](image) | ![OR gate](image) | ![NOT gate](image)
Any Boolean expression can be converted into a circuit in a straightforward way.

- Write a gate for each operation in the expression in precedence order.
- We typically draw circuits with inputs on left and outputs on right.

**Inputs**

- x
- y
- z

**Output**

\[(x + y')z + x'\]
A quick reminder

Expressions

Truth Tables

Gates (Schematics)

HDLs (Verilog)
Hardware Description Languages (HDL)

- Textual descriptions of circuits
  - (We’re very good at manipulating text...)

A Circuit:

Verilog

wire x, y, z, a, w;
and a1(a, x, y);
// gatetype name(out, in1, in2);
or o1(w, a, z);

Not like a normal programming language

- Each statement describes one or more gates and/or wires.
Summary of what we discussed today

- We can interpret high and low voltages as true and false.
- A Boolean variable can be either 1 or 0.
- AND, OR, and NOT are the basic Boolean operations.
- We can express Boolean functions in many ways:
  - Expressions, truth tables, circuits, and HDL code
  - These are different representations for equivalent things