The Big Ideas of CS 233
An Introduction to CS233

Late Add FAQ: https://wiki.illinois.edu/wiki/display/cs233sp18/Registration+FAQ
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 Homeworks: 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
CS 233 is being co-taught

Craig Zilles

Geoffrey Herman
Why take CS 233?
Why take CS 233? A warm-up clicker

Consider the following pieces of code that implement matrix multiplication, where A, B, and C, are all \( n \times n \) matrices and \( n \) is LARGE.

\[
C = A \times B
\]

Which piece of code executes the matrix multiplication fastest?

Note: they all execute the algorithm correctly

d) They are all approximately the same speed

e) (b) and (c) are faster than (a)
Core i7 Matrix Multiply Performance

![Graph showing cycles per inner loop iteration vs. array size (n) for different matrix multiplications. The graph compares jki / kji, ijk / jik, and kij / ikj.]
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 direction
  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 direction in how we access state
  4. You will learn about hardware mechanisms for parallelism
     Locality, dependencies, and direction 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”
You have seen state in three forms in your programming: data, control, and indirection.

```c
int add_numbers(int x, int y){
    int z;
    z = x + y;
    return z;
}
```

```c
int find_data(int* x){
    int y;
    y = *x;
    return y;
}
```

```c
int find_greater(int x, int y){
    if (x > y){
        return x;
    } else {
        return y;
    }
}
```
Boolean Algebra and Its Relation to Gates

Why you needed to take CS 173
We use Boolean algebra to manipulate the state of a system

Computer can do 2 things
1) Store state
2) Manipulate state
Today’s lecture

- Basic Boolean expressions
  - Booleans
  - AND, OR and NOT
  - Expressing Boolean functions:

```
Expressions <-> Truth Tables

Gates (Schematics) <-> HDLs (Verilog)
```
State information is encoded using voltages

Digital Abstraction: group ranges of voltages into two discrete (digital) values, for reliability and ease of design.
Boolean functions

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

  \[ y = f(x) \]

- Because 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>

- Repeter
- NOT
- Inverter
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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></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></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></td>
</tr>
</tbody>
</table>

- We’ll focus on 2 functions for now
i>clicker question

If there are $n$ inputs to a Boolean function, how many unique output functions could there be (i.e., how many unique columns would be created in the truth table)?

a) $2 \cdot 2 \cdot n$

b) $2 \cdot n \uparrow 2$

c) $2 \uparrow n \uparrow 2$

d) $2 \cdot 2 \uparrow n$

e) $2 \uparrow 2 \uparrow n$
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>
<tr>
<td>Truth table:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$x$</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>(\overline{1})</td>
</tr>
</tbody>
</table>

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
  - \( () \) have the highest **precedence**, followed by NOT, then 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)
Boolean expressions → Truth tables

- To compute a truth table given a Boolean expression:
  - Evaluate the function for every combination of inputs.

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

\[ f(0, 0, 0) = (\phi + 1)\phi + 1 = 1 \]

\[ f(1, 0, 1) = (\frac{1}{1})\frac{1}{\phi} = \frac{1}{\phi} \]

<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>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

a) 0
b) 1
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>1</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
A quick reminder

Expressions

Truth Tables

Gates (Schematics)

HDLs (Verilog)
The Boolean operators map to three primitive logic gates

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
<th>Logic gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND (product)</td>
<td>$xy, \text{ or } x \cdot y$</td>
<td><img src="image" alt="AND gate" /></td>
</tr>
<tr>
<td>OR (sum)</td>
<td>$x + y$</td>
<td><img src="image" alt="OR gate" /></td>
</tr>
<tr>
<td>NOT (complement)</td>
<td>$x'$</td>
<td><img src="image" alt="NOT gate" /></td>
</tr>
</tbody>
</table>
Boolean expressions → circuits

- 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.

\[(x + y')z + x'\]