https://goo.gl/forms/bofSZIAZpPabYNs93
OTCR Consulting

Who We Are

20 years
75 students
300+ clients
100% job placement

What We Do

Business Strategy
Market entry, competitor analysis, acquisitions, product analysis and more

Tech Strategy
Analysis of new technologies and potential applications, and more

Tech Implementation
Development of full-stack websites and mobile applications

Who We Work With

Microsoft
KRAFT
Motorola
Teach for America
ST

How To Join

Info Nights
January 17th – 7 pm
Engineering Hall 106B1

January 22nd – 7 pm
Lincoln Hall 1002

Case Training
January 24th – 7 pm
David Kinley Hall 123

Applications Due
Due January 25th
By 11:59 pm

www.otcr.illinois.edu
OTCR Consulting
@otcr_consulting
Why take CS 233? A warm-up i>clicker

Consider the following pieces of code that implement matrix multiplication, where A, B, and C, are all n × 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

Cycles per inner loop iteration vs. Array size (n)
The Big Ideas of CS 233
An Introduction to CS233

Late Add FAQ:
https://wiki.illinois.edu/wiki/display/cs233fa17/Registration+FAQ
Why take CS 233?

The future of high performance computing will rely on collaboration between hardware and software

https://www.youtube.com/watch?v=3LVeEjsn8Ts&feature=youtu.be&t=4268

Software security is increasingly depending on an understanding of hardware

MELTDOWN  SPECTRE
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

- State Storage
- State Manipulations
The state abstraction informs how we think about code tracing

The system clock constrains when each line of code executes

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>9</td>
</tr>
<tr>
<td>y</td>
<td>4/2</td>
</tr>
<tr>
<td>z</td>
<td>5/13</td>
</tr>
</tbody>
</table>

Diagram: Time (clk) vs. Code Execution (Compute, STORE)
Suppose you are writing a program to control a robotic chef. Which of the following would **not** be part of the robot’s stored state?

A) List of ingredients in mixing bowl  
B) Temperature of the oven  
C) Location of the cake  
D) Decrementing the timer  
E) Time left to finish baking
i>clicker

Suppose you are writing a program to track animal health in a zoo. Which of the following would **not** be part of the stored program state?

A) Names of animals in the zoo  
B) Adding an animal to the zoo  
C) Location of each animal in the zoo  
D) Time each animal was last fed  
E) Results from last medical exam
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;
}
```
```c
for (i = 0; i < N; ++i) {
    sum += array[i];
}
```

Which form of state is embodied in each variable?

a) Data  
b) Control  
c) Indirection (Addressing)
Modular Design

Consider the following problem

- You are building a system to help avoid train collisions on subways.
- Each of the 28 segments of track:
  - Senses if there is a train on it (T = 1) or no train (T = 0)
  - Has a red/yellow/green stoplight, where exactly 1 light is on at a time
    - The red light is on (R = 1) if there is a train in the next segment
    - Otherwise, yellow is on (Y = 1) if a train is 2 segments away
    - Else, green is on (G = 1)

- We could implement this as one big circuit.
Divide-and-Conquer Design

- Instead build a module:

<table>
<thead>
<tr>
<th>1 segment ahead (Xin)</th>
<th>2 segment ahead (Yin)</th>
<th>F</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- And replicate:
What is $Y(Xin, Yin)$?

a) $Xin \oplus Yin$

b) $Xin + Yin'$

c) $Xin' + Yin$

d) $Xin \cdot Yin'$

e) $Xin' \cdot Yin$

<table>
<thead>
<tr>
<th>Xin</th>
<th>Yin</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
What is $G(\text{Xin}, \text{Yin})$?

- NOR = NOT OR

a) Xin OR Yin
b) Xin NOR Yin
c) Xin AND Yin
d) Xin NAND Yin
e) Xin XOR Yin

<table>
<thead>
<tr>
<th>Xin</th>
<th>Yin</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>0</td>
</tr>
</tbody>
</table>

NAND = NOT AND
Combinational design requires thinking about logical specifications

Suppose that I have a pack of cards. Each card has a shape drawn on one side and a number written on the other side. Suppose in addition that I claim the following rule is true:

**Rule:** If a card has a square on one side, then it has an odd number on the other side.
Shape on one side, number on the other

Which card or cards should you turn over in order to decide whether the rule holds true or false for these four cards?

**Rule:** If a card has a square on one side, then it has an odd number on the other side.

a) Square  

b) Square and 6  

c) Square and 3  

d) Square, 6, and 3  

e) All
Draw a truth table for the problem

S = 1, iff the shape side is a square
N = 1, iff the number side is odd

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>if square, then odd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</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>
Use your truth table

Which card or cards should you turn over in order to decide whether the rule holds true or false for these four cards?

Rule: *If a card has a square on one side, then it has an odd number on the other side.*

a) Square  b) Square and 6  c) Square and 3

d) Square, 6, and 3  e) All
Determining which function to create needs creative human intervention

- An English description/specification

Example: A sandwich shop has the following rules for making a good (meat) sandwich:

1. All sandwiches must have at least one type of meat.
2. Don’t put both roast beef and ham on the same sandwich.
3. Cheese only goes on sandwiches that include turkey.

Write a Boolean expression for the allowed combinations of sandwich ingredients using the following variables:

- \( c = 1 \), iff the sandwich has cheese
- \( h = 1 \), iff the sandwich has ham
- \( t = 1 \), iff the sandwich has turkey
- \( r = 1 \), iff the sandwich has roast beef
Truth table → Boolean → gates example

Step 1. Write a truth table

Rules:

(1) must have at least one meat.
(2) not both roast beef and ham.
(3) cheese only if turkey.

Ingredients:  
c = cheese  
h = ham  
t = turkey  
r = roast beef

Step 2. Every 1 becomes a term
### Truth table → Boolean → gates example

<table>
<thead>
<tr>
<th>c</th>
<th>h</th>
<th>t</th>
<th>r</th>
<th>f(c,h,t,r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</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>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<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>
<td>0</td>
</tr>
</tbody>
</table>

**Step 3. OR all the terms together**
Truth table → Boolean → gates example

c’h’t’r + c’h’t’r’ + c’h’t’r + c’ht’r’ + c’htr’ + ch’t’r’ + ch’t’r + chtr’

Step 4. Convert expression to 2 levels of gates

sandwich