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 $n \times n$ 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

Cycles per inner loop iteration vs. Array size (n)

- **C** (jki / kji)
- **A** (ijk / jik)
- **B** (kij / ikj)
Grab a handout

The Big Ideas of CS 233
An Introduction to CS233

Late Add FAQ:
https://wiki.illinois.edu/wiki/display/cs233fa18/Registration+FAQ
Why I’m excited to be here
Miscellany

- Class Transcribe
- Video lectures
- Extra study session: MWF 2-3pm in 1131 Siebel
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 Homeworcks**: after every lecture in the beginning
  - Done individually

- **Labs**: Part 1 due within 48 hours. Part 2 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
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
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;
}
...```
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
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

![Diagram showing the relationship between Expressions, Truth Tables, Gates (Schematics), and HDLs (Verilog) with bidirectional arrows between them.]
State information is encoded with 1s and 0s

Circuit

Boolean

Binary

Volts 1.8

Volts 0

True

False

Volts 1.8

0

1

0
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

\[
\begin{array}{c|c}
 x & f_0(x) \\
 0 & 0 \\
 1 & 0 \\
\end{array} \quad
\begin{array}{c|c}
 x & f_1(x) \\
 0 & 0 \\
 1 & 1 \\
\end{array} \quad
\begin{array}{c|c}
 x & f_2(x) \\
 0 & 1 \\
 1 & 0 \\
\end{array} \quad
\begin{array}{c|c}
 x & f_3(x) \\
 0 & 1 \\
 1 & 1 \\
\end{array}
\]
A 2-input Boolean function has 16 unique output functions

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

- 4 possible input combinations, 16 possible functions:

| x | y | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12 | f13 | f14 | f15 |
|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| 0 | 1 | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  |
| 1 | 0 | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  |
| 1 | 1 | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  |

- 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)** $2 \times 2 \times n$
- **b)** $2 \times n^2$
- **c)** $2^{n^2}$
- **d)** $2 \times 2^n$
- **e)** $2^{2^n}$
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 $\cdot y$</td>
<td>$x + y$</td>
<td>$\bar{x}$ or $\bar{x}$</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>
<tr>
<td>$x$</td>
<td>$y$</td>
<td>$f(x,y)$</td>
<td>$x$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>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
  - 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>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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) 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>1</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) of two inputs</td>
<td>$xy$ or $x \cdot y$</td>
<td>![Logic gate for AND]</td>
</tr>
<tr>
<td>OR (sum) of two inputs</td>
<td>$x + y$</td>
<td>![Logic gate for OR]</td>
</tr>
<tr>
<td>NOT (complement) on one input</td>
<td>$x'$</td>
<td>![Logic gate for NOT]</td>
</tr>
</tbody>
</table>
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.
What Boolean expression does this circuit implement?

- \( (x + y)y' \)
- \( x + y + y' \)
- \( xy' + y \)
- \( (xy) + y' \)
- \( (x+y)(x+y') \)

Circuits → expressions

\[ f(x,y) \]
What Boolean expression does this circuit implement?

a) \((x + y)y'\)
b) \(x + y + y'\)
c) \(xy' + y\)
d) \((xy) + y'\)
e) \((x+y)(x+y')\)
A quick reminder

Expressions → Truth Tables

Gates (Schematics) → HDLs (Verilog)

Expressions → Gates (Schematics)

Truth Tables → 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;

HDL Code:

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
Things to do before next lecture

- Get on Piazza for CS 233
- Watch the Introduction to Verilog video
  - We’ll send details by email and post on Piazza
- Do your Web Homework problems
  - [https://prairielearn.engr.illinois.edu/](https://prairielearn.engr.illinois.edu/) There is something due each night before a lecture.

Late Add FAQ:
[https://wiki.illinois.edu/wiki/display/cs233fa18/Registration+FAQ](https://wiki.illinois.edu/wiki/display/cs233fa18/Registration+FAQ)
Discussion Section starts this week!

- We’ll introduce you to the tools designing, testing, and debugging digital logic circuits
  - Verilog
  - Waveform Viewers

Late Add FAQ:
https://wiki.illinois.edu/wiki/display/cs233fa18/Registration+FAQ