Handouts are in the back

The Big Ideas of CS 233

An Introduction to CS233

iclickers are for practice today

Late Add FAQ:
https://wiki.illinois.edu/wiki/display/cs233fa17/Registration+FAQ

Welcome!! :)

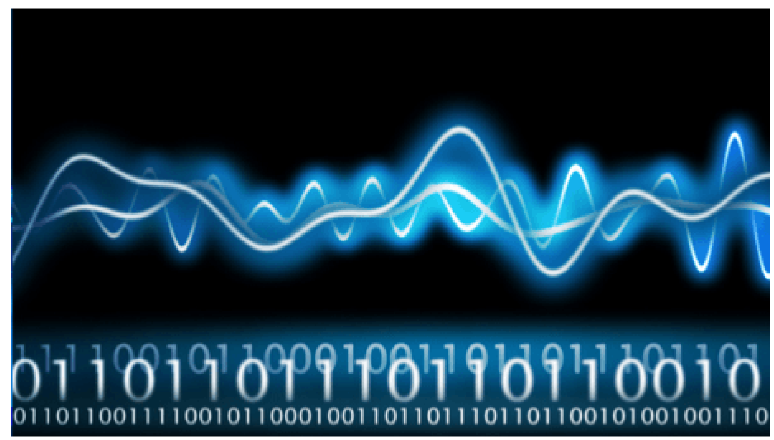
Class Mechanics on one Slide

- **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, follow the template)
- **Web Homeworks**: after every lecture in the beginning
  - Done individually
  - Numbers match the lecture number
- **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

Why I'm excited to be here

I will disappear for a couple weeks at some point this semester

iFoundry



Why take CS 233?

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*B

Which piece of code executes the matrix multiplication fastest?

Note: they all execute the algorithm correctly

a)
```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

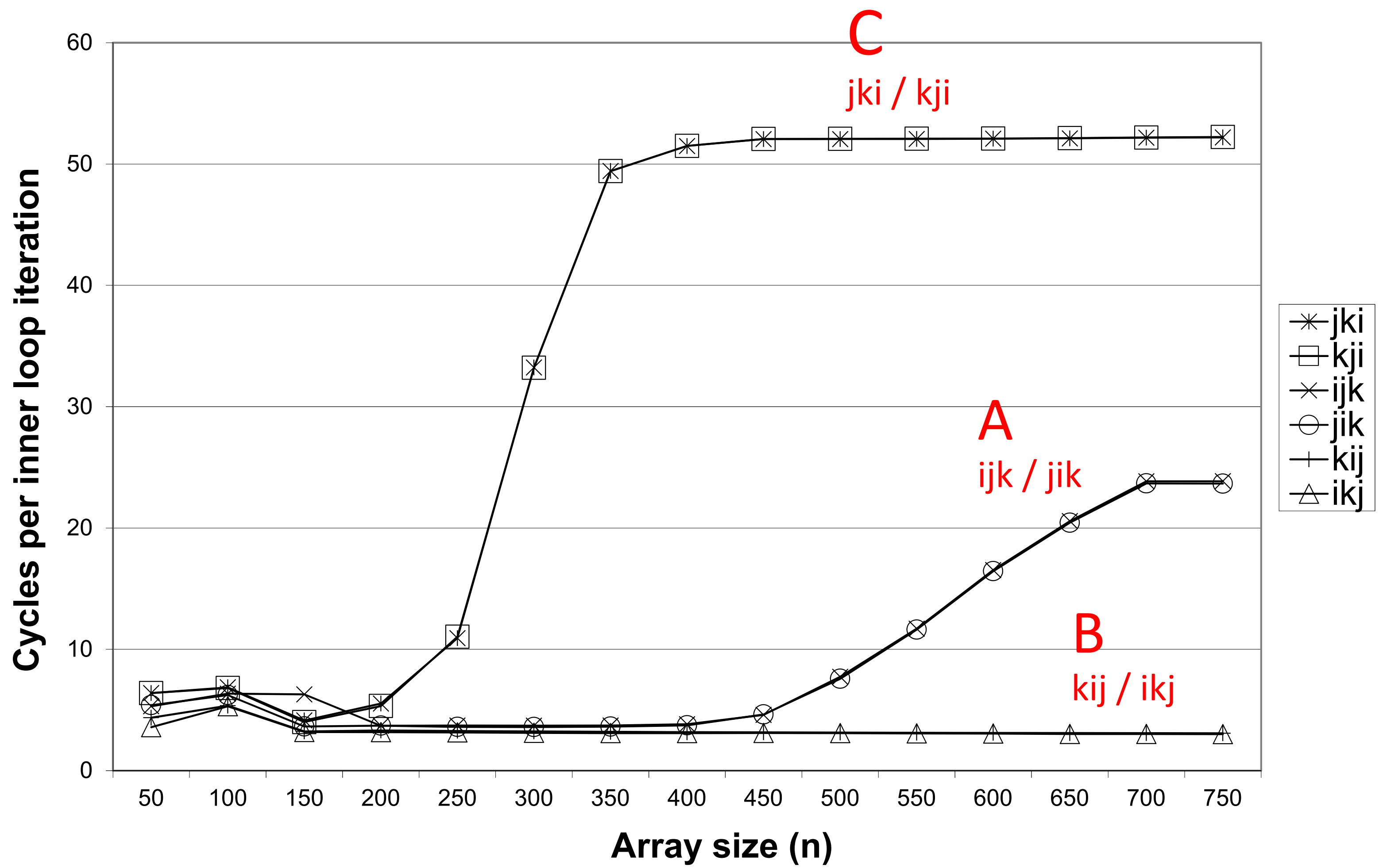
b)
```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

c)
```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

d) They are all approximately the same speed

e) (b) and (c) are faster than (a)

Core i7 Matrix Multiply Performance

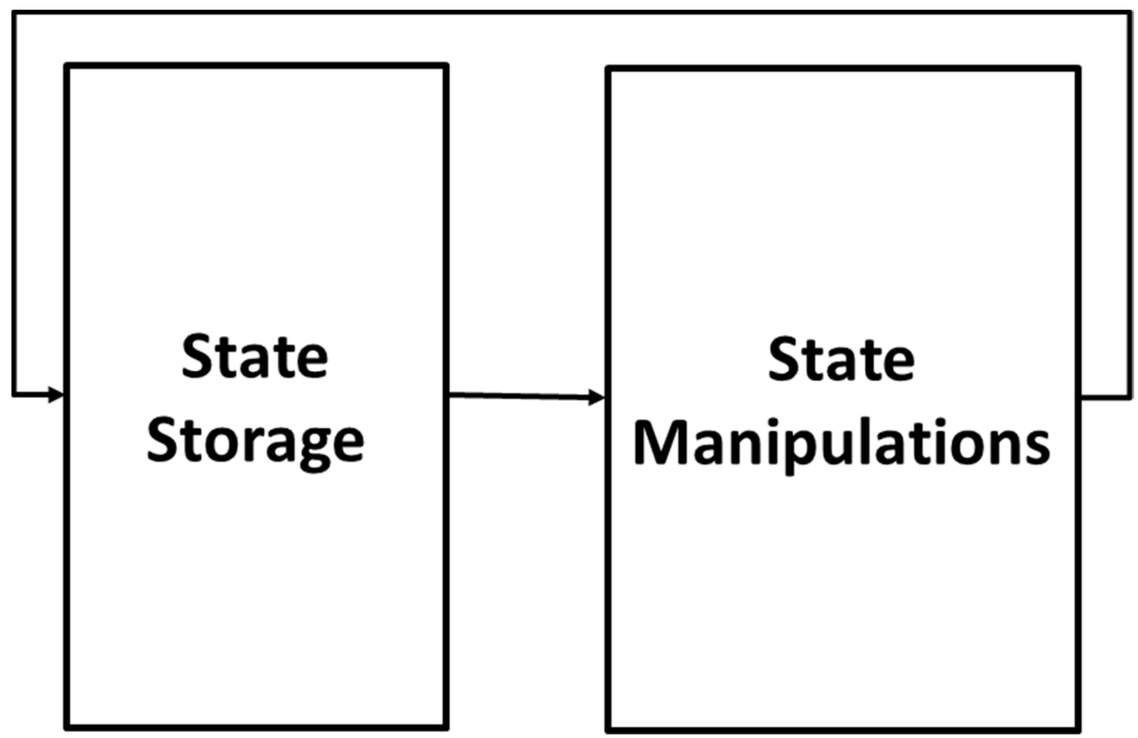


233 in one slide!

- 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

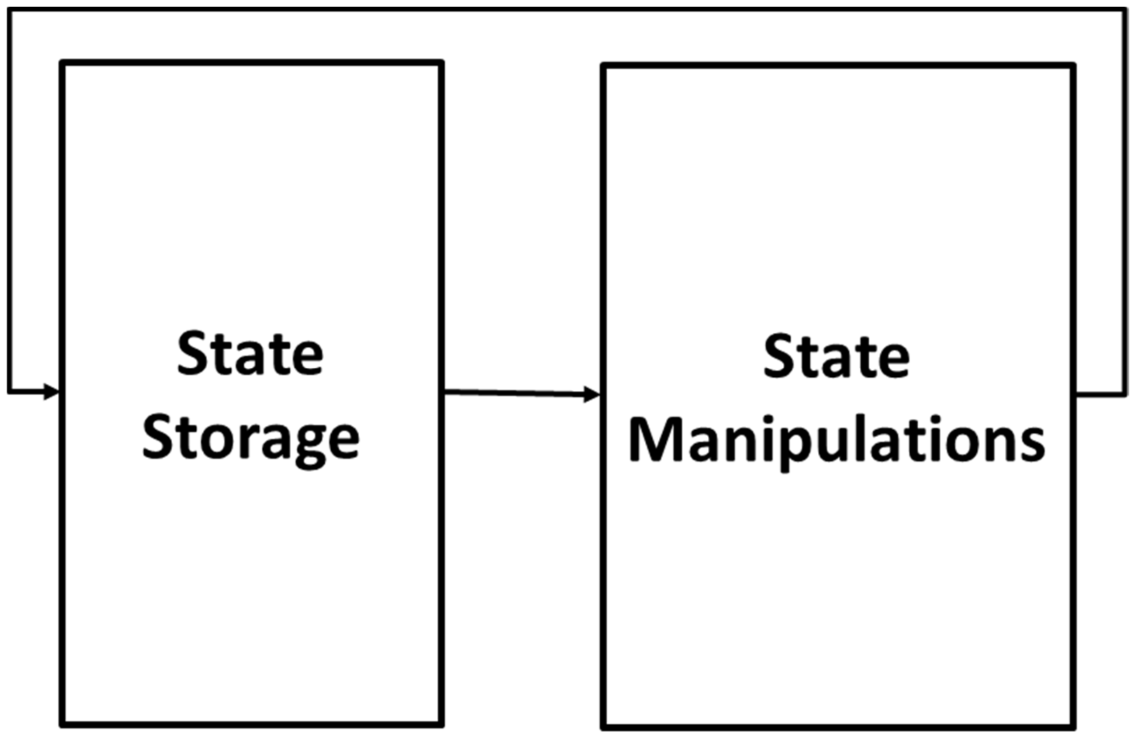
- We will have a SPIMbot contest!

A computer can do 2 things: Store state…



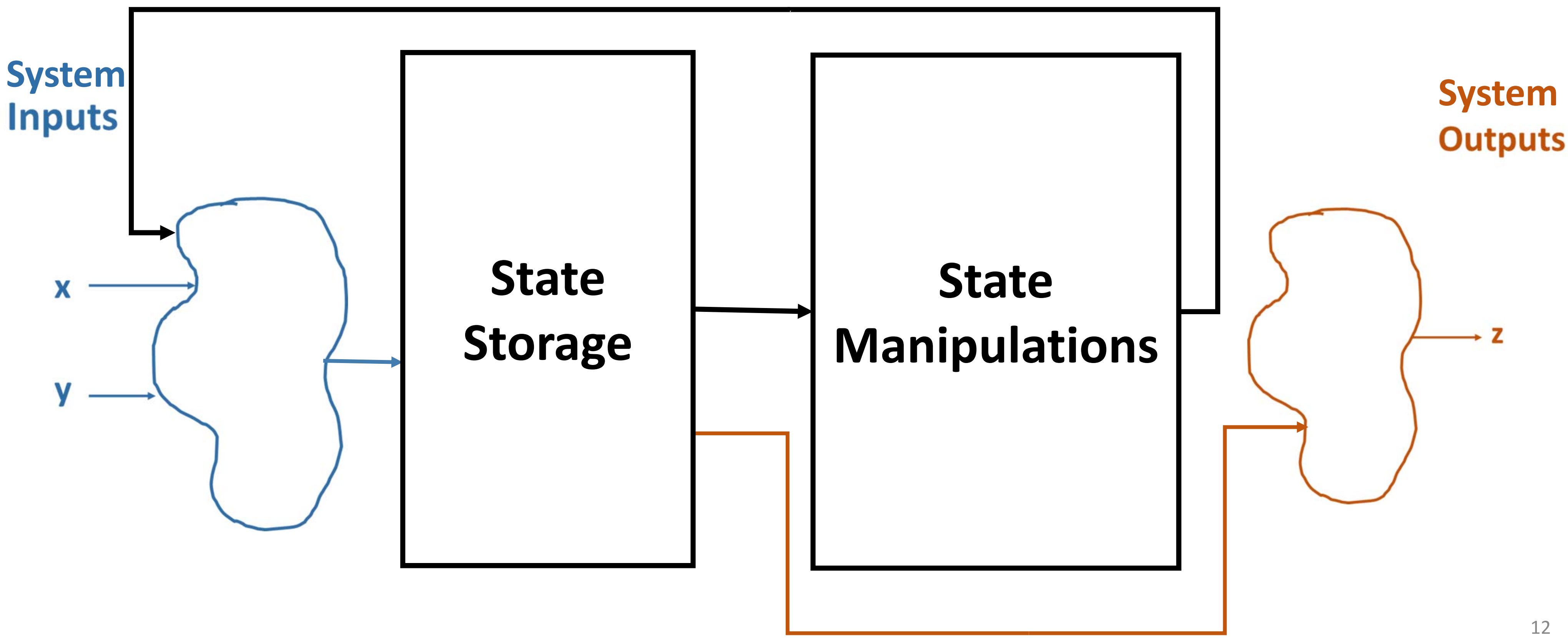
State is the relevant information about the progress of my system

A computer can do 2 things: …and manipulate state



Computation changes my state in a limited number of ways

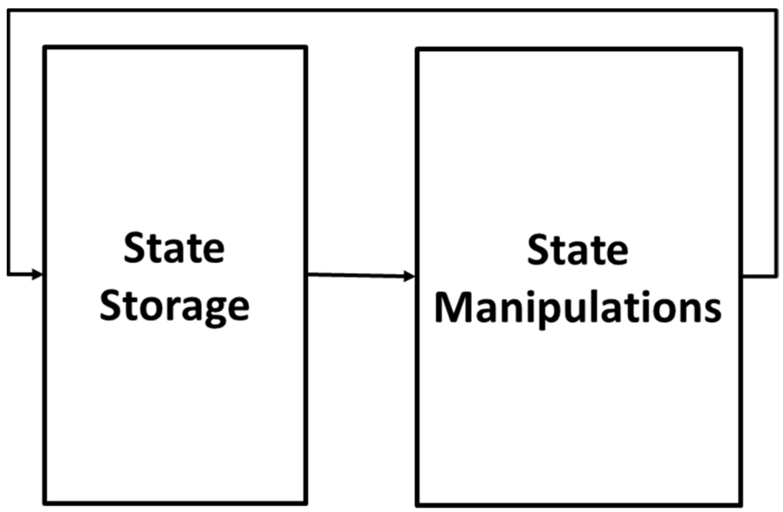
State changes can respond to user (system) inputs
State is used to compute a system output



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

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

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

Indirection(Address)

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

   y = *x;
   return y;
}
```

Control

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

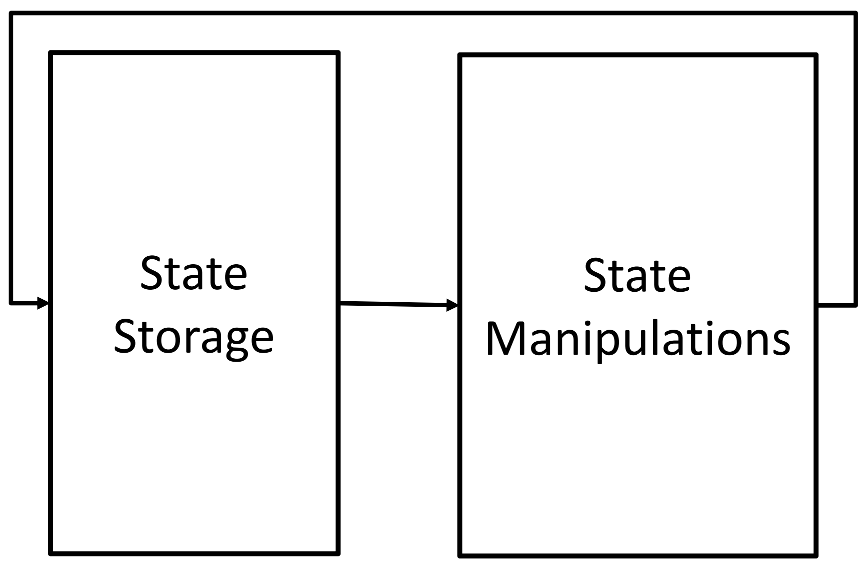
Boolean Algebra and Its Relation to Gates

Why you needed to take CS 173

We use Boolean algebra to manipulate the state of a system

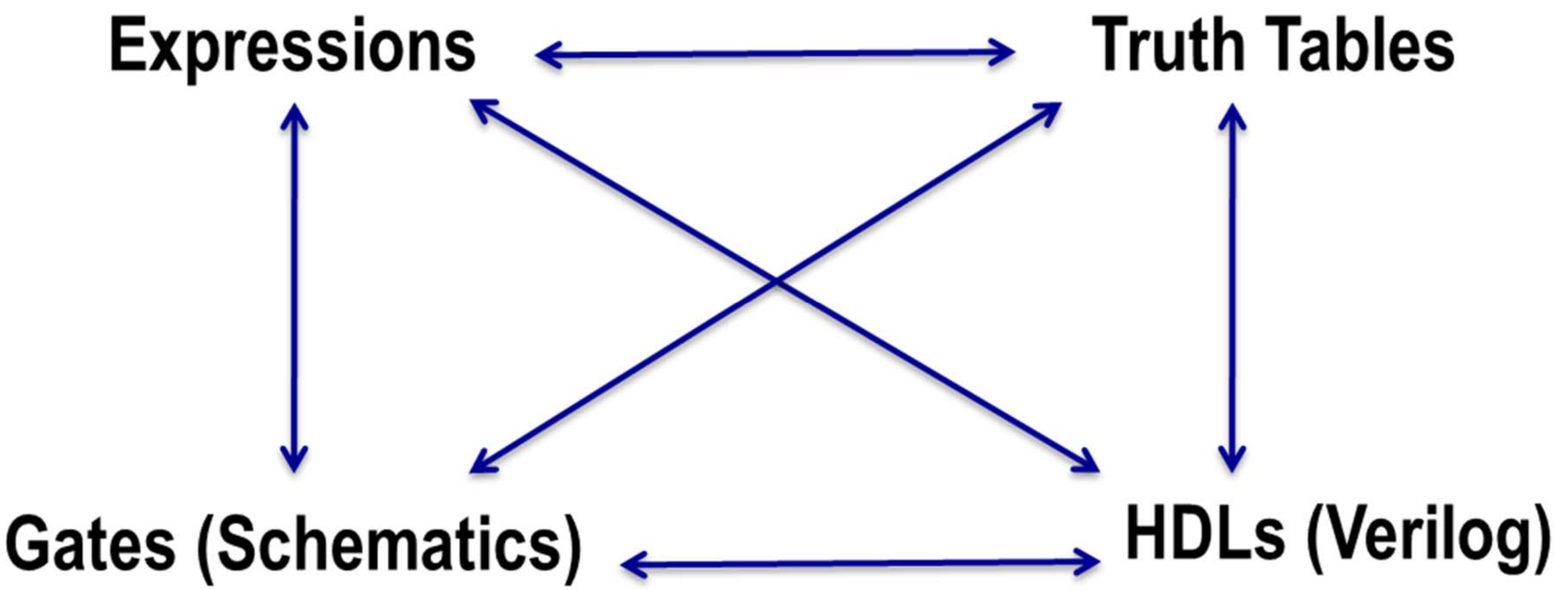
Computer can do 2 things

1) Store state
2) Manipulate state



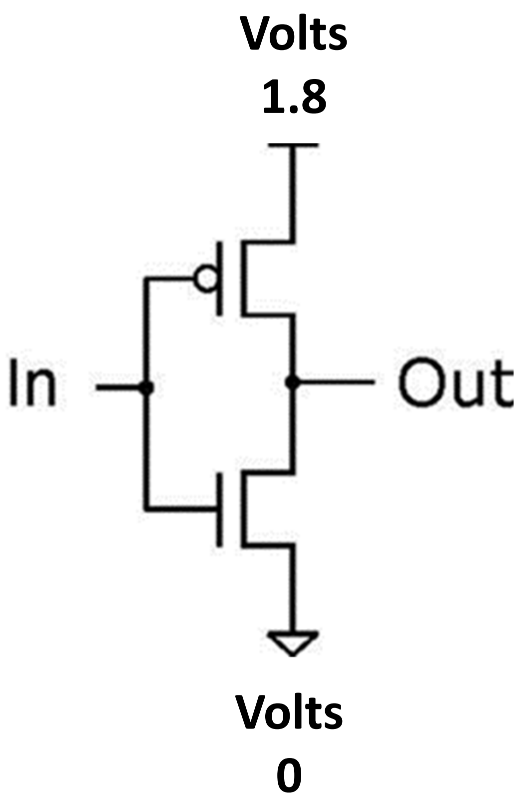
Today's lecture

- Basic Boolean expressions
  - Booleans
  - AND, OR and NOT

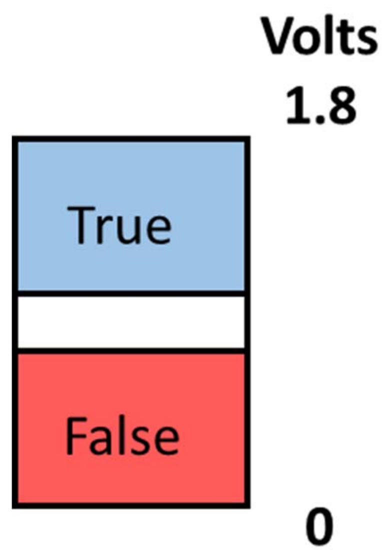


State information is encoded with 1s and 0s

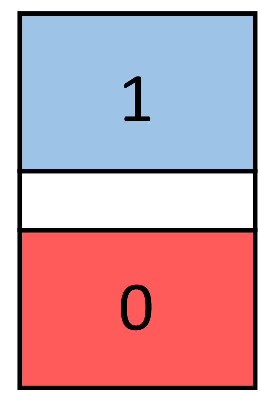
Circuit



Boolean



Binary



Boolean functions

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

0 or 1

$$y = f(x)$$

Output

Operations

Input

- 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

0 or 1

$$y = f(x)$$

- 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

x	f(x)
0	f(0)
1	f(1)

- The 4 possible 1-input Boolean functions

2^2

x	$f_0(x)$
0	0
1	0

x	$f_1(x)$
0	0
1	1

x	$f_2(x)$
0	1
1	0

x	$f_3(x)$
0	1
1	1

A 2-input Boolean function has 16 unique output functions

$z = f(x,y)$

- 4 possible input combinations, 16 possible functions:

n

$2^2 = 4$

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	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

$2^4 = 16$

2^{2^n}

- 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) $2 * 2 * n$

b) $2 * n^2$

c) 2^{n^2}

d) $2 * 2^n$

e) 2^{2^n}

We use three basic logical operations: AND, OR, and NOT

Operation:	AND (product) of two inputs	OR (sum) of two inputs	NOT (complement) on one input
Expression Notation:	xy, or x•y	x + y	x’ or $\bar{x}$

Truth table:

x	y	f(x)y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	f(x)y
0	0	0
0	1	1
1	0	1
1	1	1

x	f(~~x~~) x
0	1
1	0

These are sufficient to implement any Boolean function

Boolean expressions (formally)

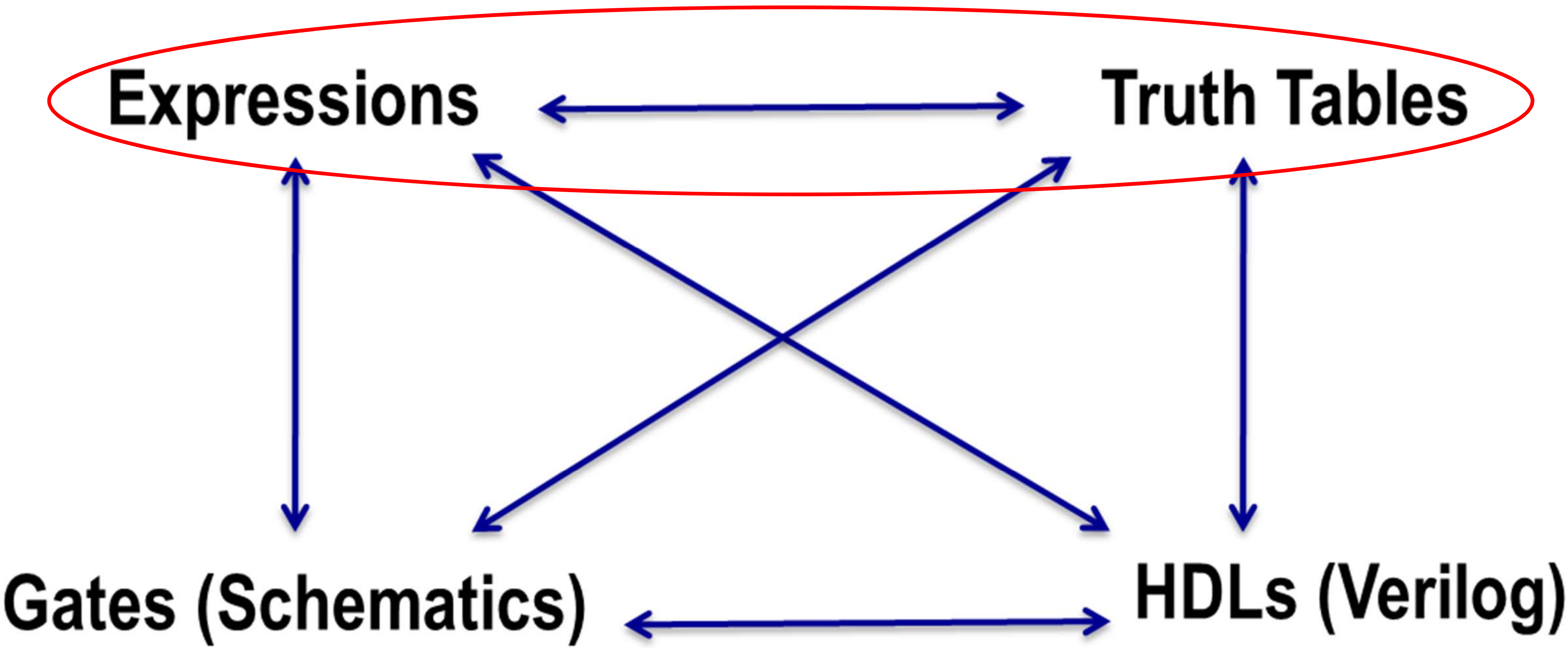
- Use these basic operations to form more complex expressions:

$$f(x,y,z) = (x + y')z + x'$$

- 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



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) = (0 + 1) 0 + 1 = 1

$= (1) \cdot 0 + 1$

$= 0 + 1$

$= 1$

f(1,0,1) = (____ + ____)____ + ____ = ____

x	y	z	f(x,y,z)
0	0	0	1
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	1
1	1	0	
1	1	1	

a)	0
b)	1

Boolean expressions $\longrightarrow$ Truth tables

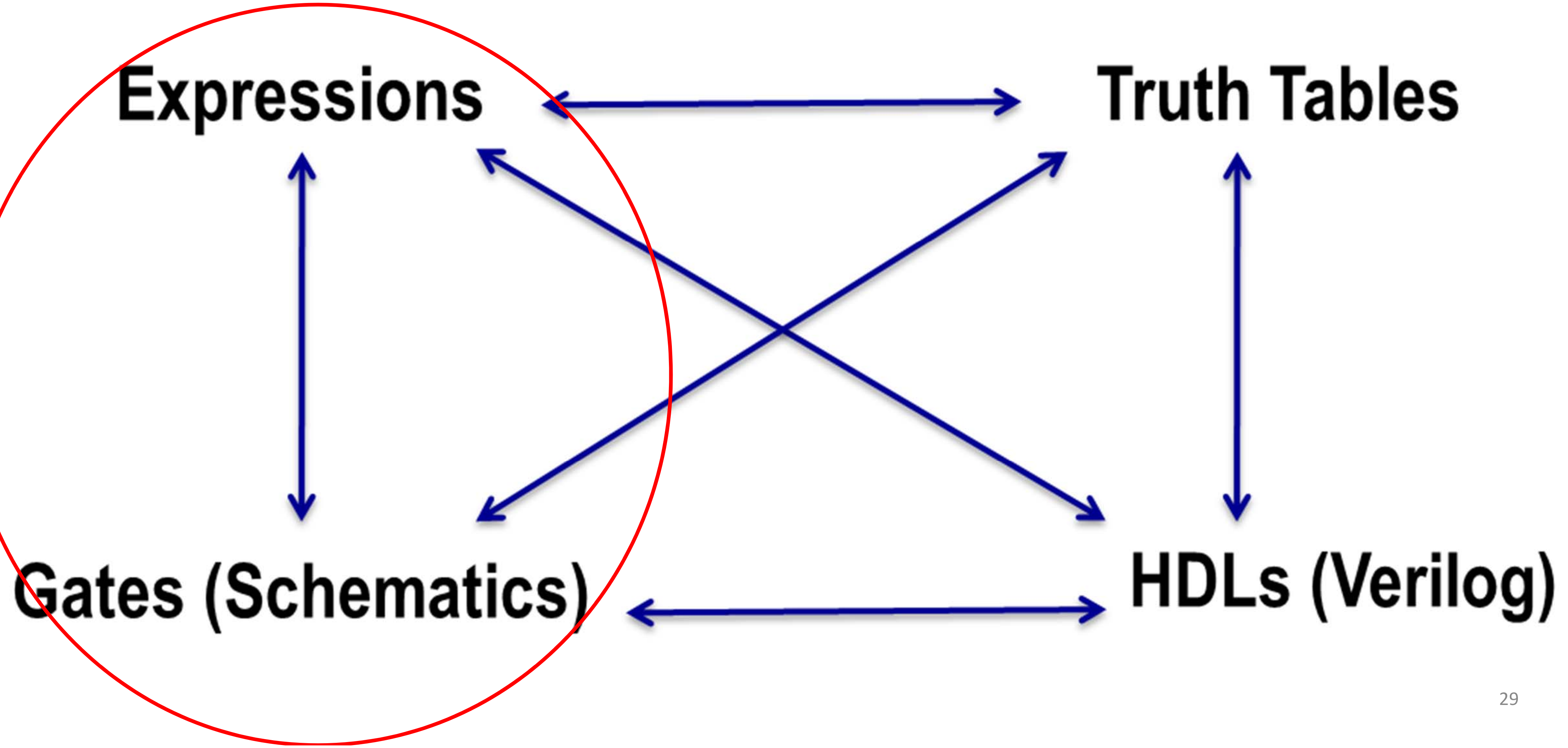
- To compute a truth table given a Boolean expression:
  - Evaluate the function for every combination of inputs.

$f(x,y,z) = (x + y')z + x'$

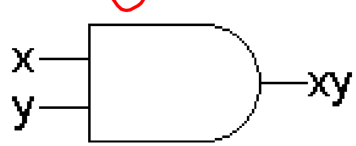
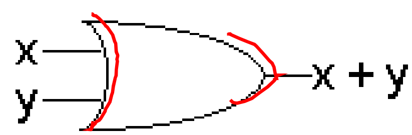
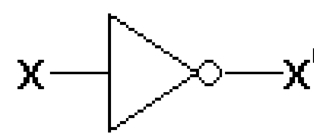
$f(0,0,0) = (0 + 1)0 + 1 = 1$
$f(0,0,1) = (0 + 1)1 + 1 = 1$
$f(0,1,0) = (0 + 0)0 + 1 = 1$
$f(0,1,1) = (0 + 0)1 + 1 = 1$
$f(1,0,0) = (1 + 1)0 + 0 = 0$
$f(1,0,1) = (1 + 1)1 + 0 = 1$
$f(1,1,0) = (1 + 0)0 + 0 = 0$
$f(1,1,1) = (1 + 0)1 + 0 = 1$

x	y	z	f(x,y,z)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

A quick reminder

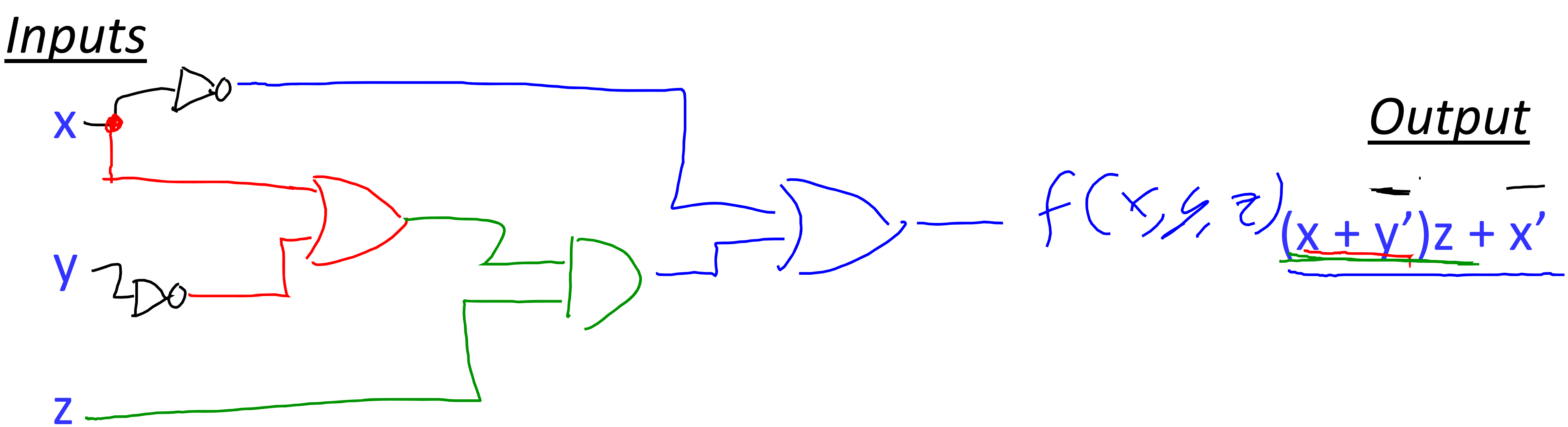


The Boolean operators map to three primitive logic gates

Operation:	AND (product) of two inputs	OR (sum) of two inputs	NOT (complement) on one input
Expression:	xy, or x•y	x + y	x’
Logic gate:			

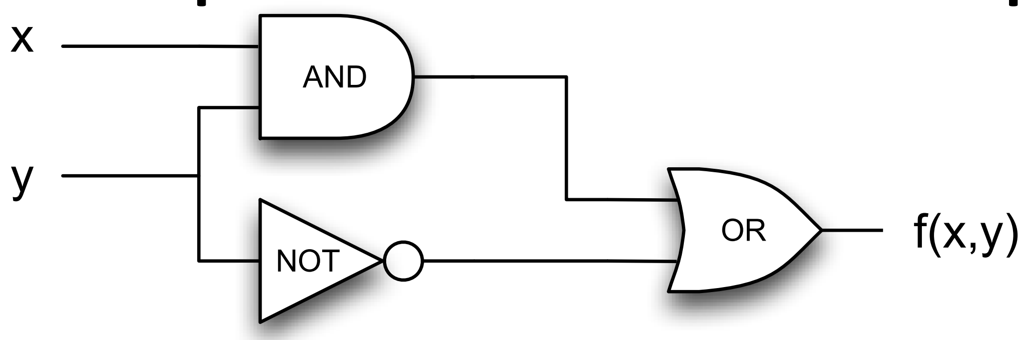
Boolean expressions → circuits

- *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.



Circuits → expressions

- **What Boolean expression does this circuit implement?**



a) (x + y)y’

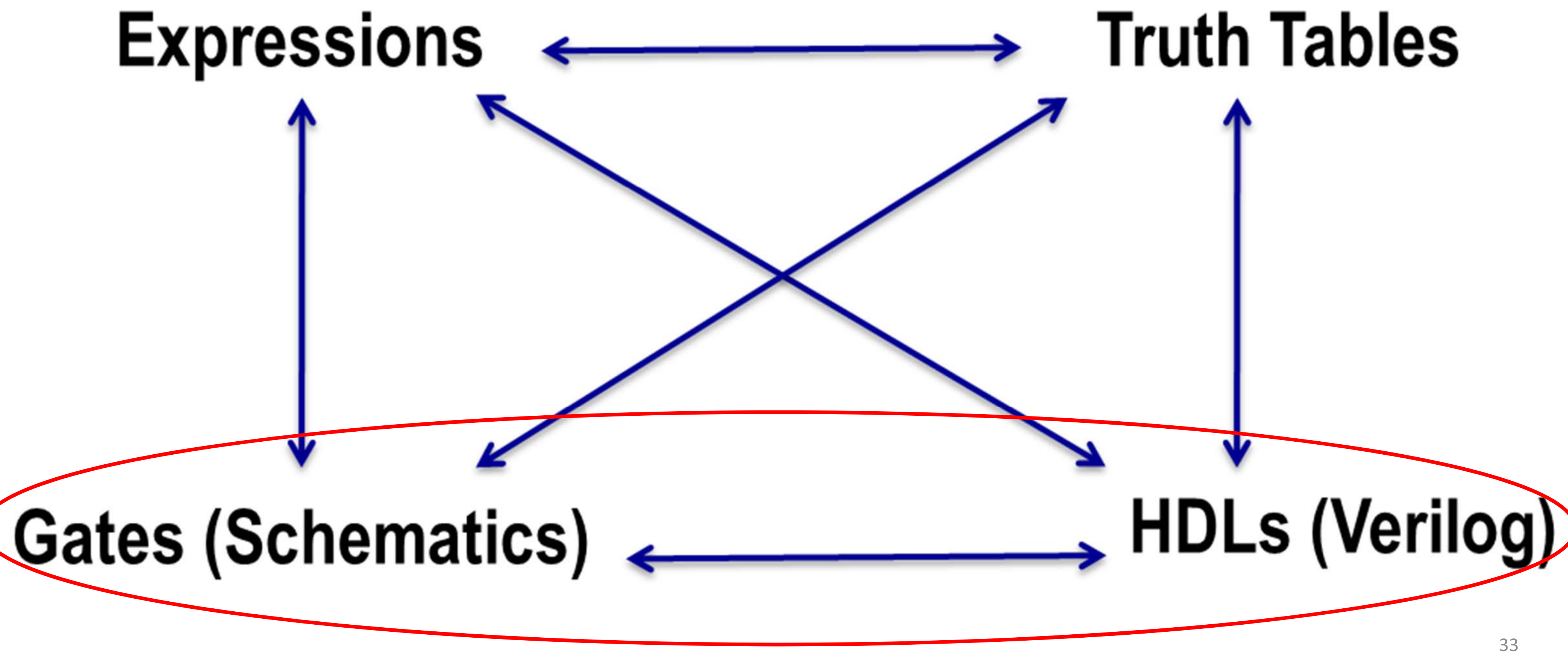
b) x + y + y’

c) xy’ + y

d) (xy) + y’

e) (x+y)(x+y’)

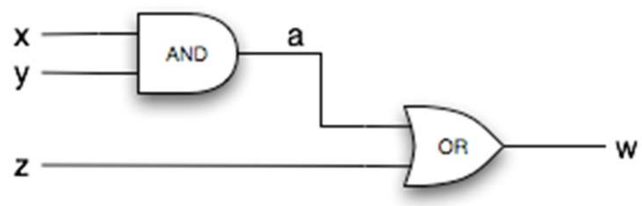
A quick reminder



Hardware Description Languages (HDL)

- Textual descriptions of circuits
  - (We're very good at manipulating text…)

A Circuit:



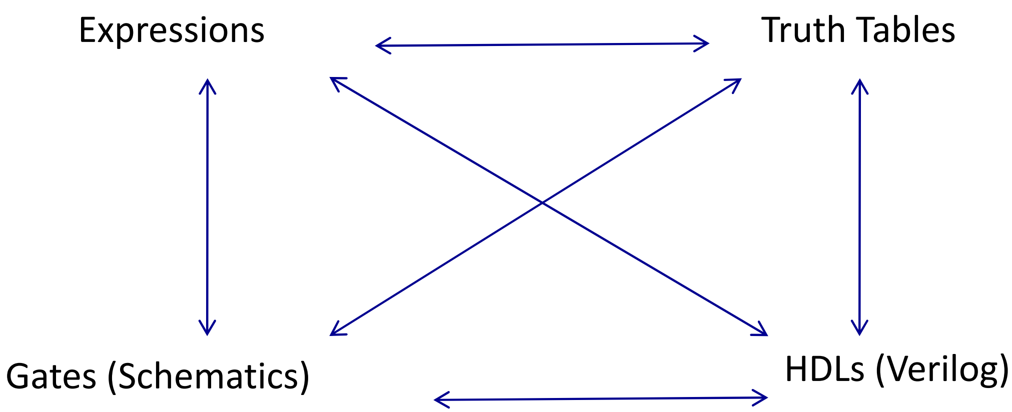
Verilog HDL Code:

```
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



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/ There is something due each night before a lecture.

Late Add FAQ:
https://wiki.illinois.edu/wiki/display/cs233fa17/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/cs233fa17/Registration+FAQ