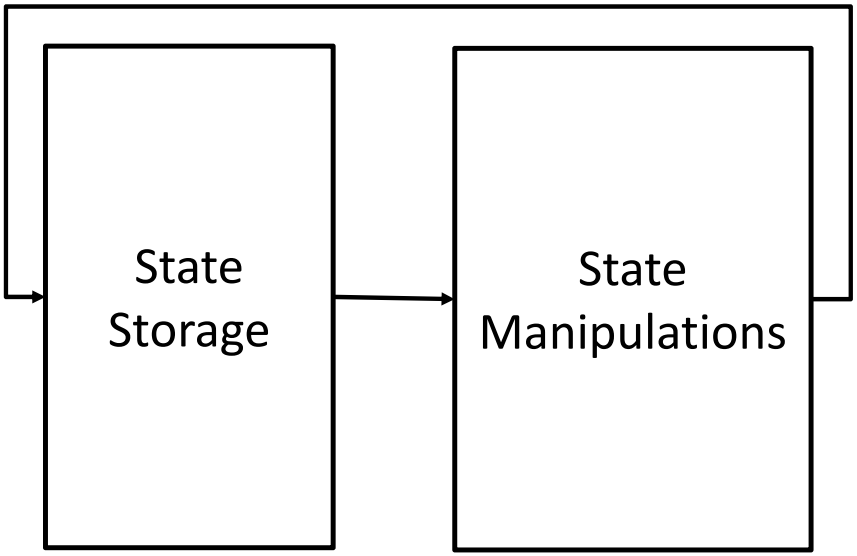
Combinational Logic Design

How do we create the specification?

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

- Combinational Logic
  - Different Representations of Boolean Functions (review)
- How to design any circuit
  - Write a truth table
  - Sum-of-Products implementation
  - Example
- Other gates you should know about (XOR, NAND, NOR)
- Divide-and-Conquer design

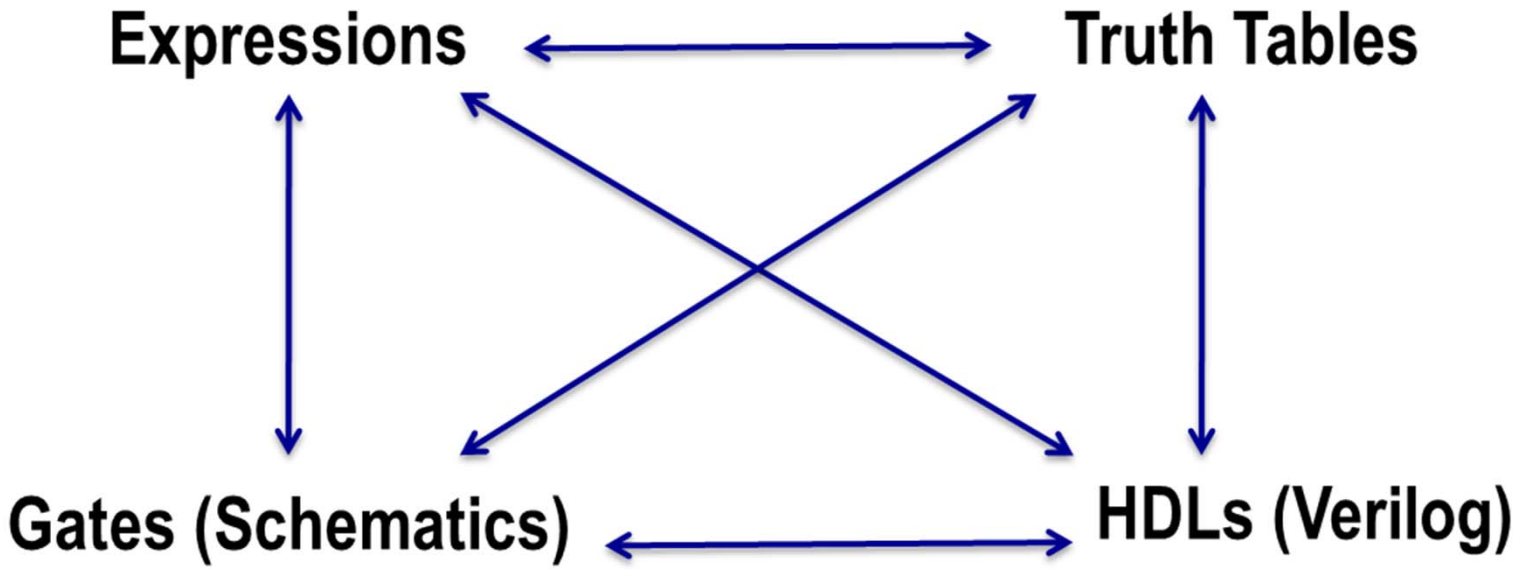
Combinational Logic

- Definition: Boolean circuits where the output is a pure function of the present input only.

- Circuits made up of gates, that don't have any feedback
  - No feedback: **outputs** are not connected to **inputs**
  - If you change the inputs, *and wait for a while*, the correct outputs show up.
    - Real circuits have delays (more on this later)

- Can be represented by Boolean Algebra

These four representations of Boolean functions are informationally equivalent



Relatively mechanical to translate between these formats

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

English → Truth Table example

- Most reliable method
  1. Write a truth table
  2. Every row evaluating to 1 becomes a term
  3. OR all the terms together

- This will give an un-optimized expression
  - (we can write computer programs to optimize expressions)
    - (or better yet, use the ones that other people wrote…)
  - (we can't write programs to design circuits for us.)

An easy, useful form of Boolean expressions is called Sum of Products (SOP)

- A sum of products (SOP) expression contains:
  - only OR (sum) operations at the "outermost" level
  - Each term that is summed must be an AND (product) of literals

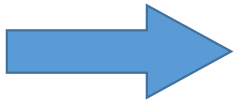
$$f(x,y,z) = y' + x'yz' + xz$$

- SOP expressions are advantageous because they can be implemented using a two-level circuit
  - literals (includes complements) at "0th" level
  - AND gates at the first level
  - a single OR gate at the second level

Truth tables → Boolean expressions

1. For each row in truth table where output is 1
   - Write a product term that is true for that set of inputs
     - And only for that set of inputs

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



xy’z

   - This product will include each variable exactly once
2. OR all the product terms together

   product-term1 + product-term2 + product-term3 + …

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.

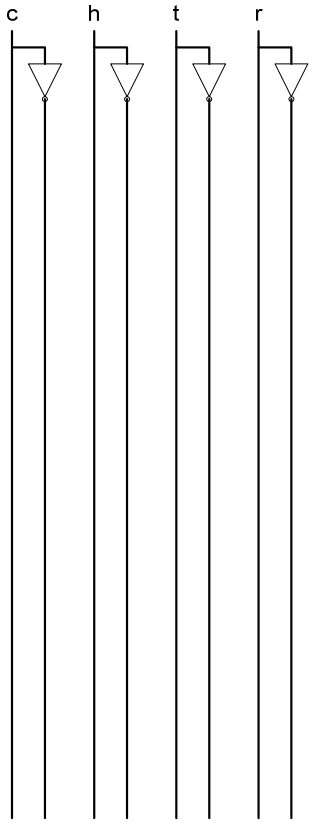
Ingredients: c = cheese
h = ham
t = turkey
r = roast beef

c	h	t	r	f(c,h,t,r)
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Step 2. Every 1 becomes a term

Truth table → Boolean → gates example

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

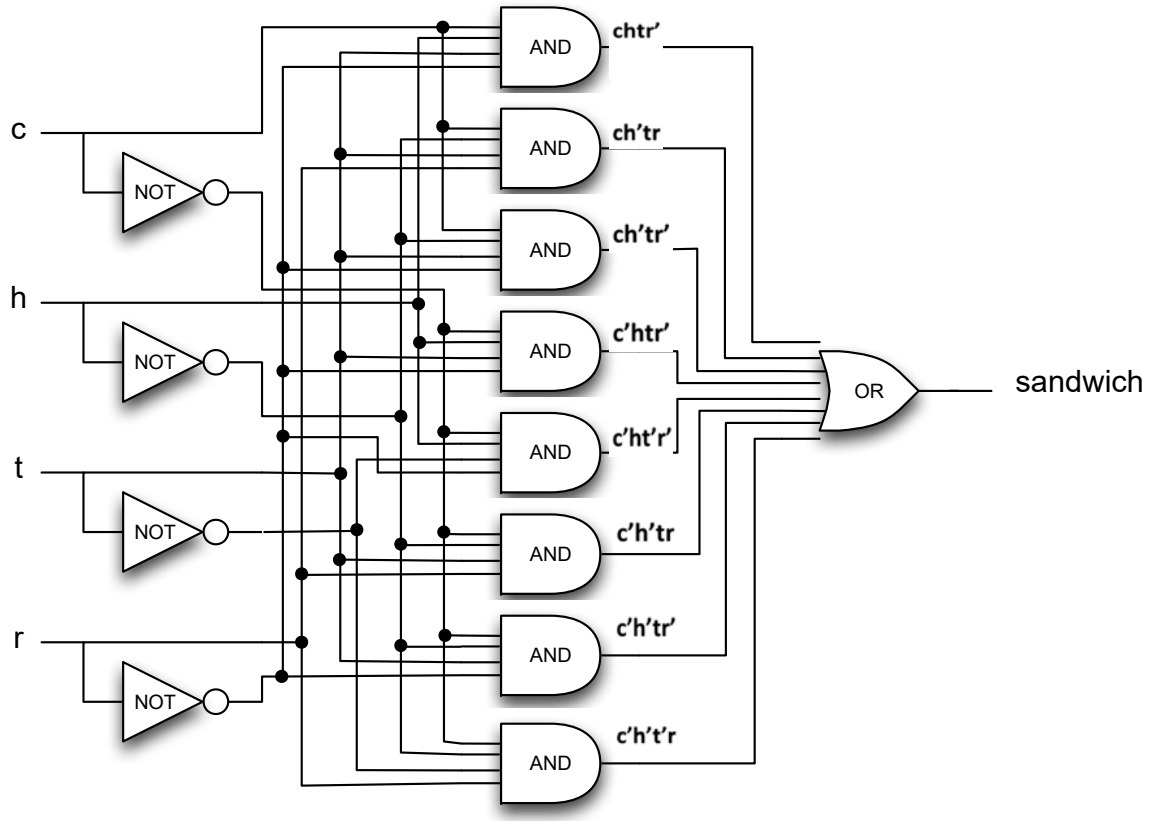


___ sandwich

Step 4. Convert expression to 2 levels of gates

Truth table → Boolean → gates example

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



Three other notable 2-input functions

- Remember this table?

AND ↓ (f1) OR ↓ (f7)

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

Three additional handy logical operations: NAND, NOR, and XOR

Operation:	NAND (NOT-AND)	NOR (NOT-OR)	XOR (eXclusive OR)
Expression Notation:	$(xy)' = x' + y'$	$(x + y)' = x'y'$	$x \oplus y = x'y + xy'$

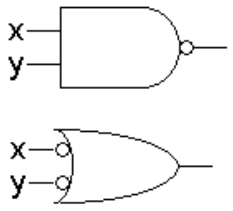
Truth table:

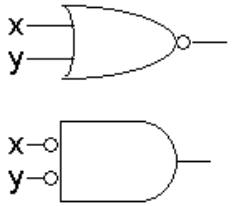
x	y	(xy)'
0	0	1
0	1	1
1	0	1
1	1	0

x	y	(x+y)'
0	0	1
0	1	0
1	0	0
1	1	0

x	y	x⊕y
0	0	0
0	1	1
1	0	1
1	1	0

Logic gates:





A two-input XOR gate's output is true when exactly one of its inputs is true

x	y	x⊕y
0	0	0
0	1	1
1	0	1
1	1	0

$$x \oplus y = x'y + xy'$$

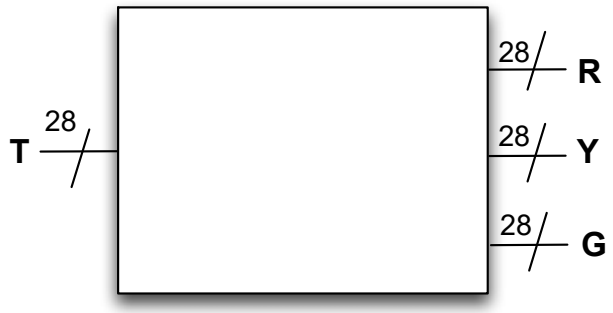
- XOR generalizes as "an odd number of input variables are 1"

- Several fascinating properties of the XOR operation:

$x \oplus 0 = x$	$x \oplus 1 = x'$
$x \oplus x = 0$	$x \oplus x' = 1$
$x \oplus (y \oplus z) = (x \oplus y) \oplus z$	[Associative]
$x \oplus y = y \oplus x$	[Commutative]
$m \oplus x \oplus x = m$	[Encryption with one-time pad]

Divide-and-Conquer Design

- Consider the following problem
  - You are building 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.

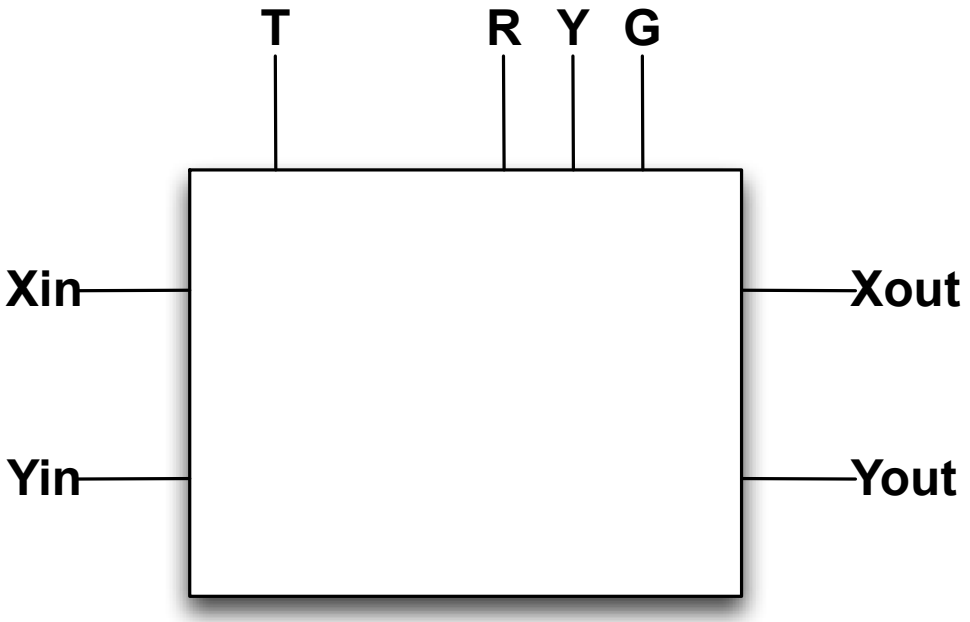



Proximity
sensor

Why would that be a bad idea?

Divide-and-Conquer Design

- Instead build a module:



- And replicate: