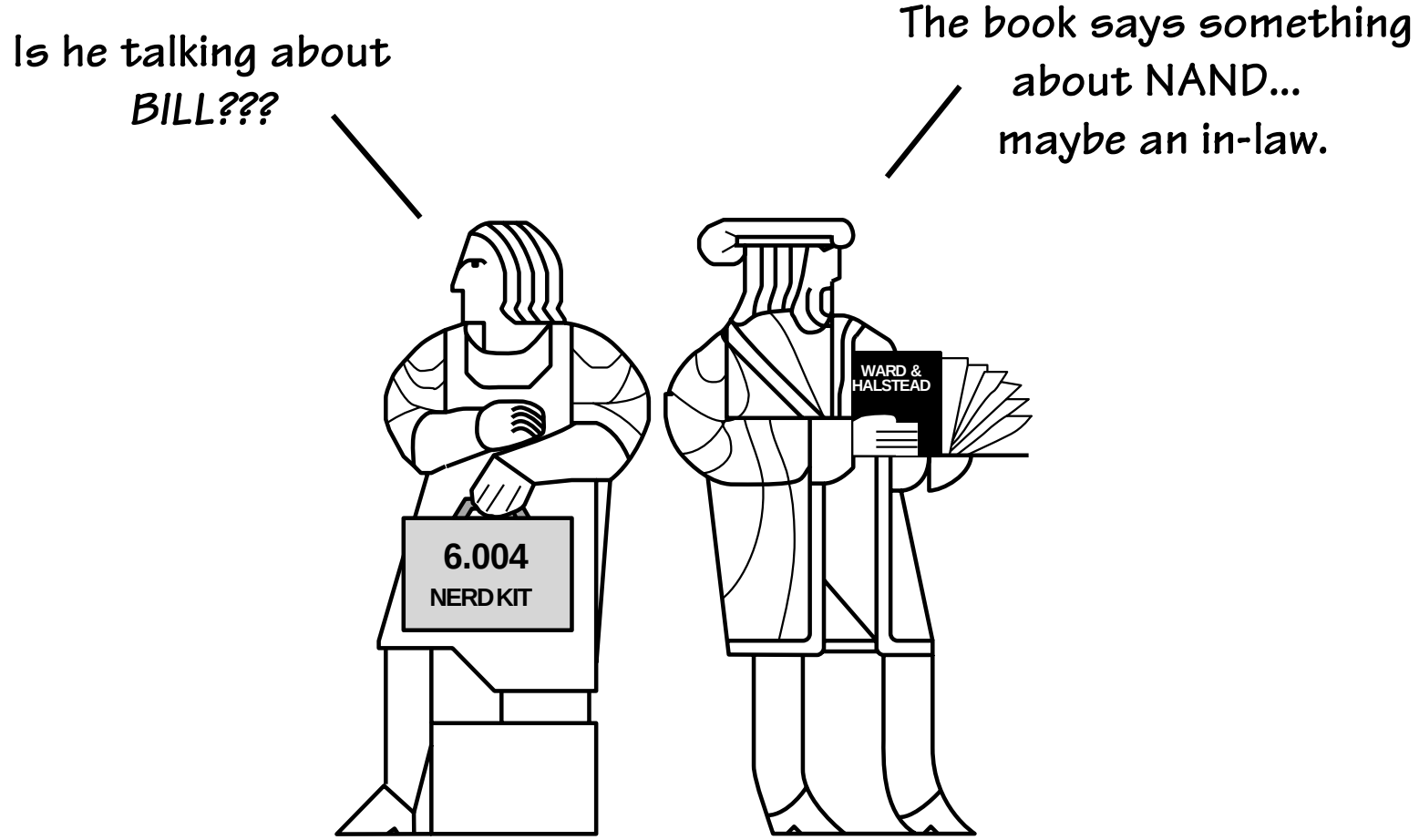
# Gates

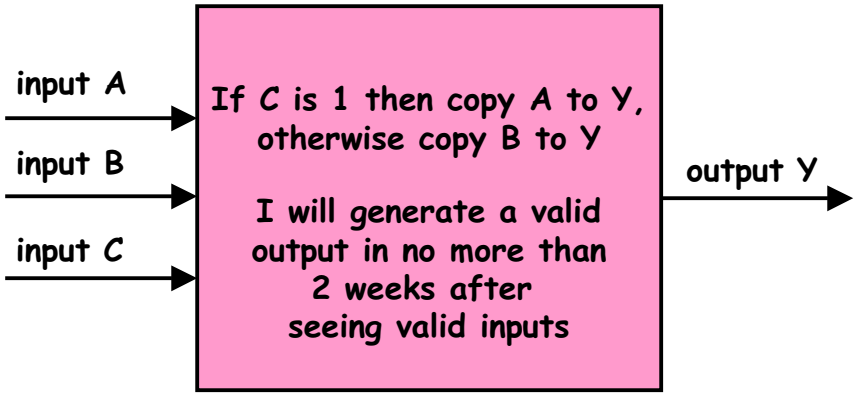



## Handouts: Lecture Slides, PS2, Lab1

# A Quick Review

- A *combinational device* is a circuit element that has
  - one or more digital *inputs*
  - one or more digital *outputs*
  - a *functional specification* that details the value of each output for every possible combination of valid input values
  - a *timing specification* consisting (at minimum) of an upper bound $t_{PD}$ on the required time for the device to compute the specified output values from an arbitrary set of stable, valid input values

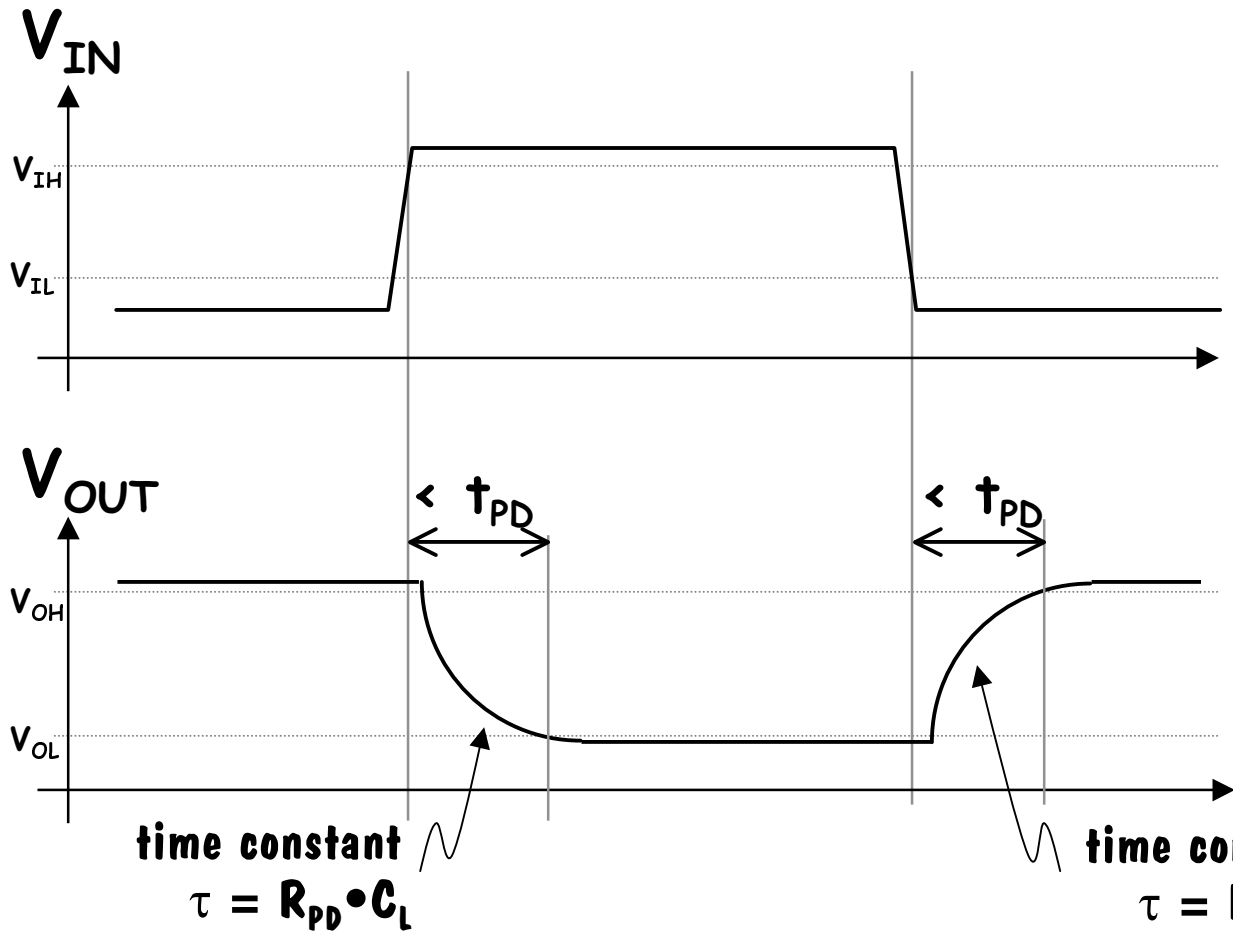
**Static discipline**



# Delays, they're a fact of ~~life~~ light

Propagation delay ($t_{PD}$):

An UPPER BOUND on the delay from valid inputs to valid outputs.




**GOAL:**

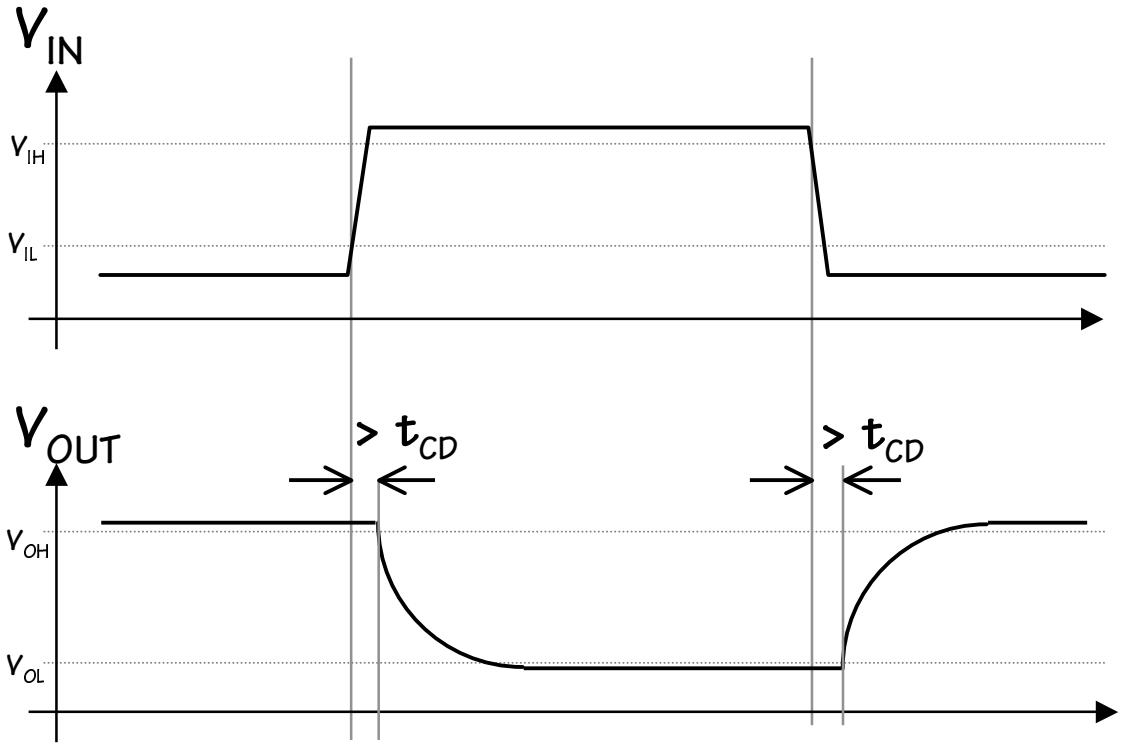
*minimize* propagation delay!

**ISSUE:**

keep Capacitances low and transistors fast

# Contamination Delay

INVALID inputs take time to propagate, too...




Do we really need $t_{CD}$?

Usually not... it'll be important when we design circuits with registers (coming soon!)

If $t_{CD}$ is not specified, safe to assume it's *0*.

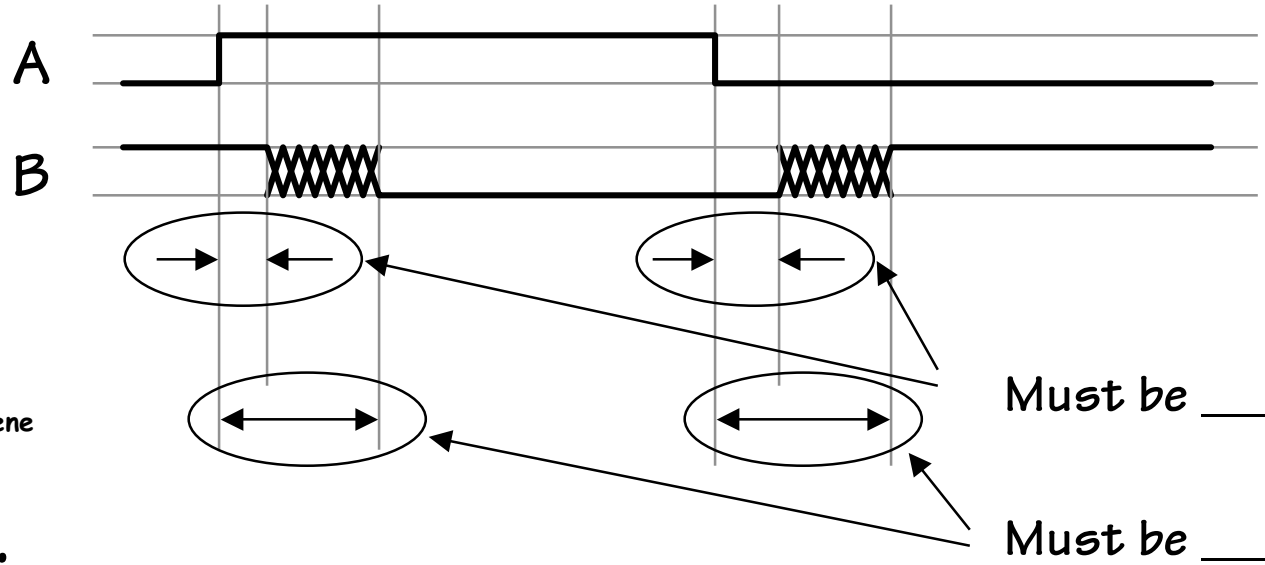
CONTAMINATION DELAY, $t_{CD}$

A LOWER BOUND on the delay from any invalid input to an invalid output

# The Combinational Contract


| A | B |
|---|---|
| 0 | 1 |
| 1 | 0 |

$t_{PD}$ propagation delay
$t_{CD}$ contamination delay




Must be $> t_{CD}$

Must be $< t_{PD}$


N.B:

1. *No Promises during* 
2. Default (conservative) spec: $t_{CD} = 0$

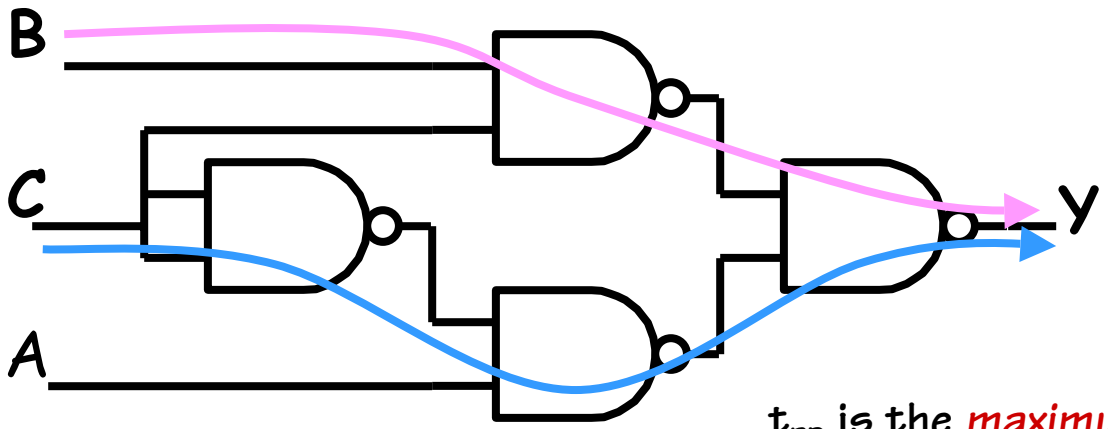
# Example: Timing Analysis

If NAND gates have a $t_{PD}$ = 4nS and $t_{CD}$ = 1nS

$t_{PD}$ = 12 nS

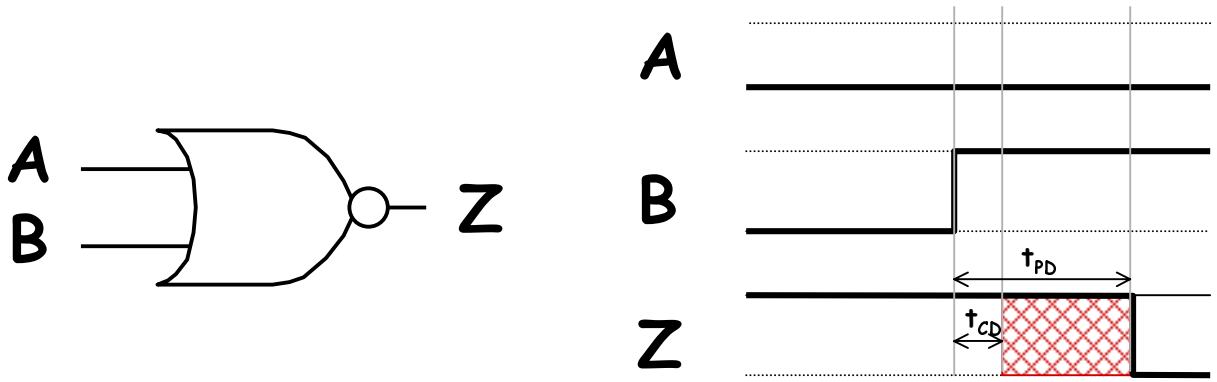
$t_{CD}$ = 2 nS

$t_{CD}$ is the *minimum* cumulative contamination delay over all paths from inputs to outputs




$t_{PD}$ is the *maximum* cumulative propagation delay over all paths from inputs to outputs

# One last issue...

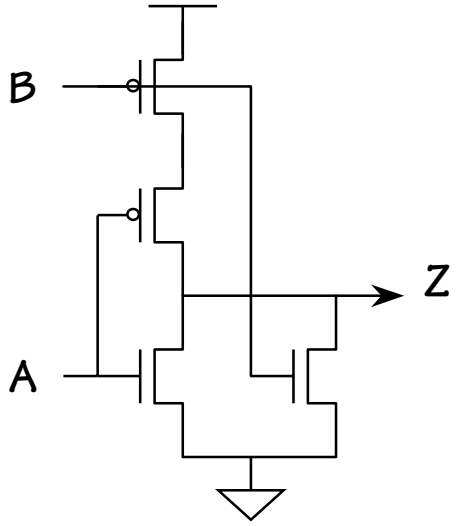


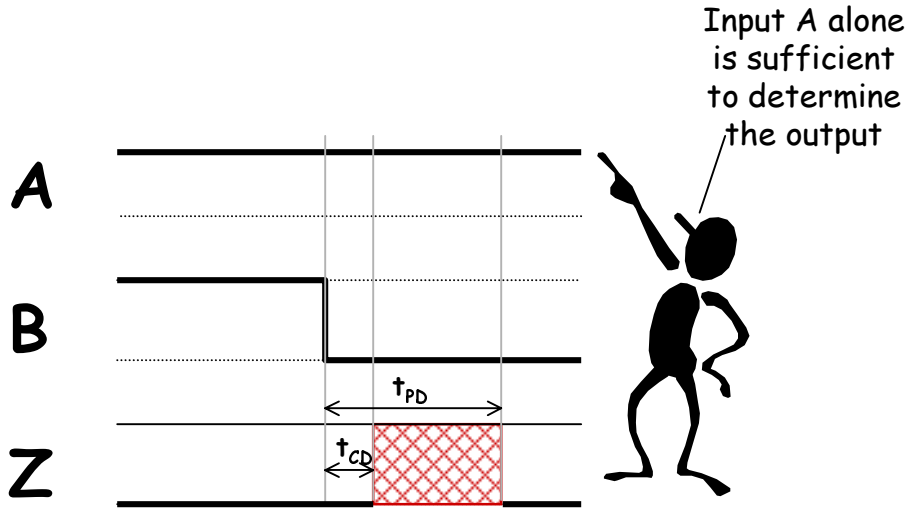

Recall the rules for *combinational devices*:

> Output guaranteed to be valid when all inputs have been valid for at least $t_{PD}$, and, outputs may become invalid no earlier than $t_{CD}$ after an input changes!

Many gate implementations--e.g., CMOS—
adhere to even tighter restrictions.

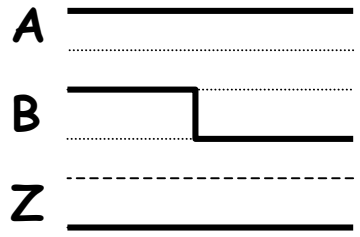
# What happens in this case?



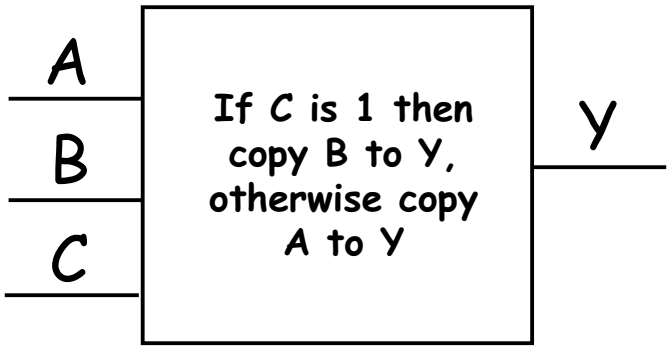

*LENIENT* Combinational Device:

Output guaranteed to be valid when any combination of inputs sufficient to determine output value has been valid for at least $t_{PD}$. *Tolerates transitions -- and invalid levels -- on irrelevant inputs!*



# Now let's design stuff!

We need to start somewhere -- usually it's the functional specification





If you are like most engineers you'd rather see a formula that parse a logic puzzle. The fact is, **any combinational function can be expressed as a table**.

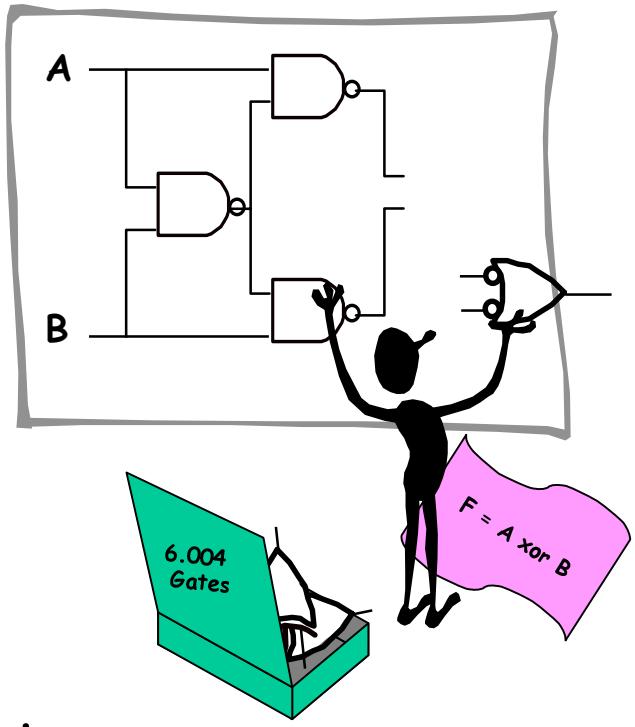
These "**truth tables**" are a concise description of the combinational system's function. Conversely, any computation performed by a combinational system can expressed as a truth table.

## Truth Table

| C | B | A | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

# Where do we start?

We have a bag of gates.




We have a spec.

What do we do?

Did I mention we have gates?

We need

... a systematic approach for designing logic

# A Slight Diversion

Are we sure we have all the gates we need?

Just how many two-input gates are there?

| AND | | OR | | NAND | | NOR | |
|---|---|---|---|---|---|---|---|
| AB | Y | AB | Y | AB | Y | AB | Y |
| 00 | 0 | 00 | 0 | 00 | 1 | 00 | 1 |
| 01 | 0 | 01 | 1 | 01 | 1 | 01 | 0 |
| 10 | 0 | 10 | 1 | 10 | 1 | 10 | 0 |
| 11 | 1 | 11 | 1 | 11 | 0 | 11 | 0 |


Hum… all of these have 2-inputs (no surprise)

… each with 4 permutations, giving $2^2$ output cases

How many permutations of 4 outputs are there? $4^2$

# There are only so many gates

How many of these gates can be implemented using a single CMOS gate?

There are only 16 possible 2-input gates

... some we know already, others are just silly

| INPUT AB | ZERO | AND | A > B | A | B > A | B | XOR | OR | NOR | XNOR | NOT 'B' | A <= B | NOT 'A' | B <= A | NAND | ONE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 01 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 11 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

Do we need all of these gates?

**Nope. After all, we describe them all using AND, OR, and NOT.**

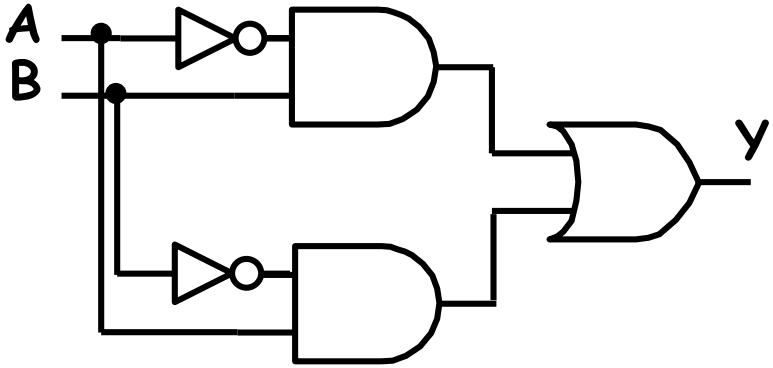
# We can make most gates out of others

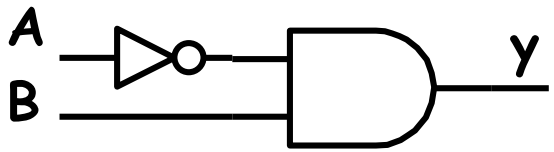
**B>A**

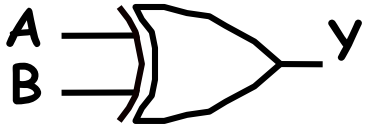
| AB | Y |
|---|---|
| 00 | 0 |
| 01 | 1 |
| 10 | 0 |
| 11 | 0 |

**XOR**

| AB | Y |
|---|---|
| 00 | 0 |
| 01 | 1 |
| 10 | 1 |
| 11 | 0 |



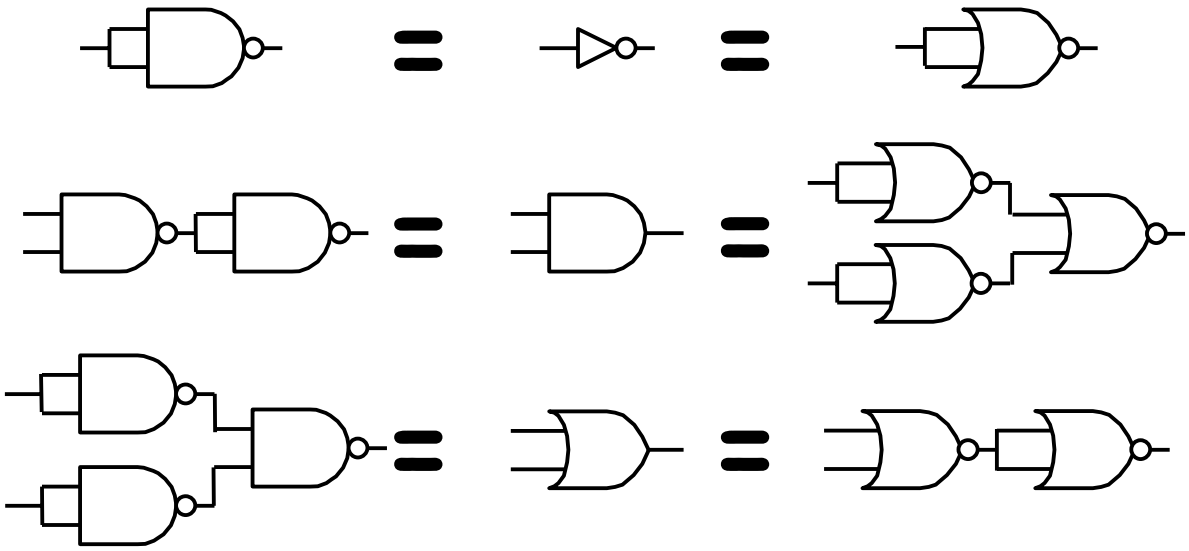




How many different gates do we really need?

# One will do!

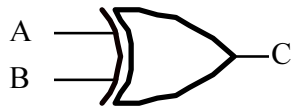
NANDs and NORs are universal



Ah!, but what if we want more than 2-inputs

# Stupid Gate Tricks

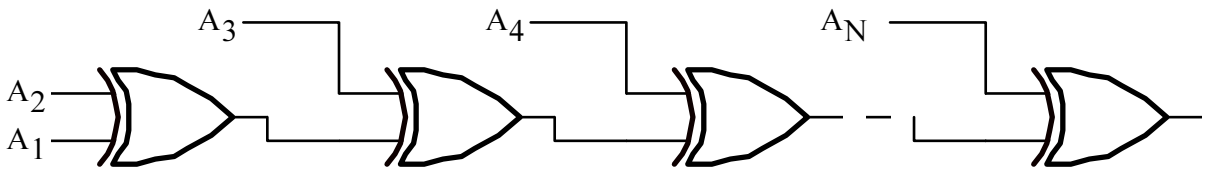
Suppose we have some 2-input XOR gates:




$t_{pd} = 1$
$t_{cd} = 0$

| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

And we want an N-input XOR:



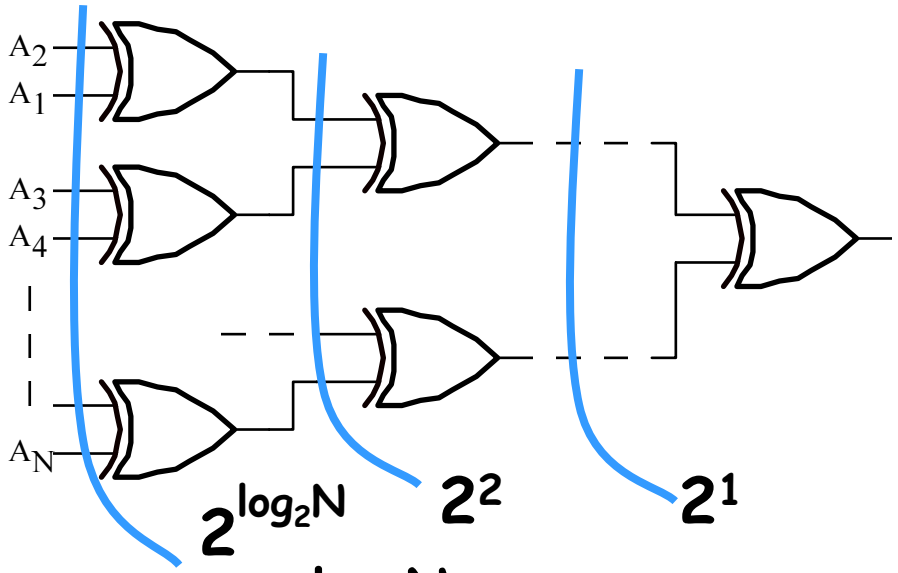

output = 1
iff number of 1s
input is ODD
("ODD PARITY")

$t_{pd}$ = O( N ) -- WORST CASE.

Can we compute N-input XOR faster?

# I think that I shall never see

## a circuit lovely as...




N-input TREE has O( log N ) levels...

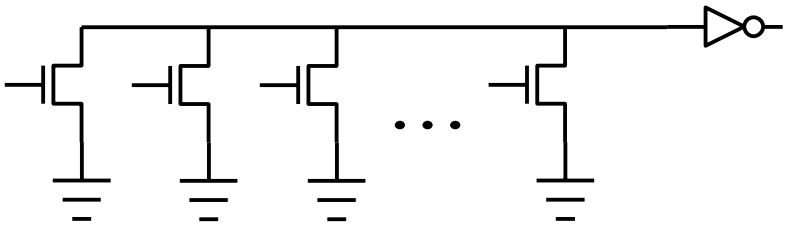
Signal propagation takes O( log N ) gate delays.

Question: Can EVERY N-Input Boolean function be implemented as a tree of 2-input gates?

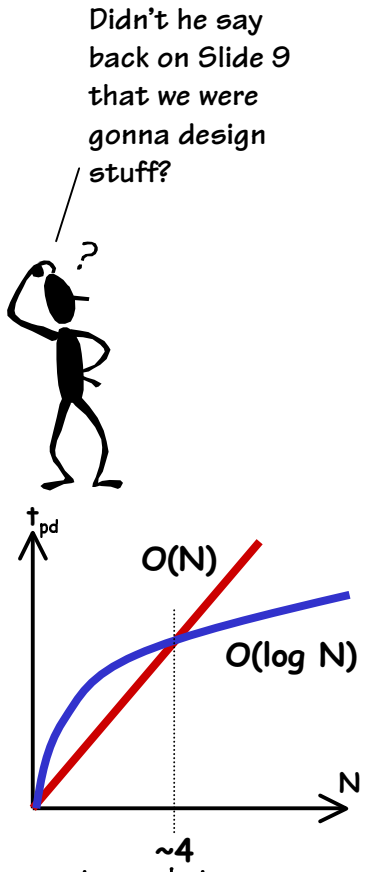
# Are Trees Always Best?

## Alternate Plan: Large Fan-in gates

- N pulldowns with complementary pullups
- Output HIGH if any input is HIGH = "OR"



- Propagation delay: O( **N** )
  since each additional MOSFET adds C




Don't be mislead by the "big O" stuff… the constants in this case can be much smaller… so for small N this plan might be the best.

# Here's a Design Approach

## Truth Table

| C | B | A | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |


1) Write out our functional spec as a truth table
2) Write down a Boolean expression for every '1' in the output

$$Y = \overline{C}\overline{B}A + \overline{C}B\overline{A} + C\overline{B}\overline{A} + CBA$$

3) Wire up the gates, call it a day, and go home!

This approach will always give us logic expressions in a particular form: SUM-OF-PRODUCTS

# Straightforward Synthesis

We can implement

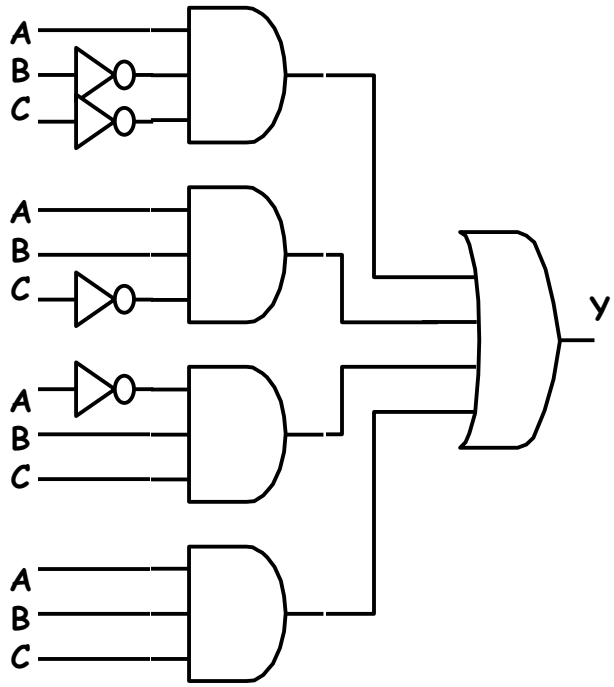
SUM-OF-PRODUCTS

with just three levels of logic.

INVERTERS/AND/OR

Propagation delay --

No more than 3 gate delays (ignoring fan-in)



# Logic Simplification

Can we implement the same function with fewer gates?

Before trying we'll add a few more tricks in our bag.

BOOLEAN ALGEBRA:

| | |
|---|---|
| OR rules: | $a + 1 = 1,\ a + 0 = a,\ a + a = a$ |
| AND rules: | $a1 = a,\ a0 = 0,\ aa = a$ |
| Commutative: | $a + b = b + a,\ ab = ba$ |
| Associative: | $(a + b) + c = a + (b + c),\ (ab)c = a(bc)$ |
| Distributive: | $a(b+c) = ab + ac,\ a + bc = (a+b)(a+c)$ |
| Complements: | $a + \bar{a} = 1, \quad a\bar{a} = 0$ |
| Absorption: | $a + ab = a, \quad a + \bar{a}b = a + b$ |
| | $a(a + b) = a, \quad a(\bar{a} + b) = ab$ |
| Reduction: | $ab + \bar{a}b = b, \quad (a + b)(\bar{a} + b) = b$ |
| DeMorgan's Law: | $\bar{a} + \bar{b} = \overline{ab}, \quad \bar{a}\bar{b} = \overline{a + b}$ |

# Minimal Sum-of-Products

Once a logic expression is in Sum-of-Product (SOP) form, it is easy to minimize it further. In fact, we can reduce it to minimal set of products.

Here's the trick:

Factor out common terms and collapse groups of "don't care" bits.

$$Y = \overline{C}\,\overline{B}A + \overline{C}B\overline{A} + CB\overline{A} + CBA$$

$$Y = \overline{C}A(\overline{B} + B) + CB(\overline{A} + A)$$

$$Y = \overline{C}A + CB$$

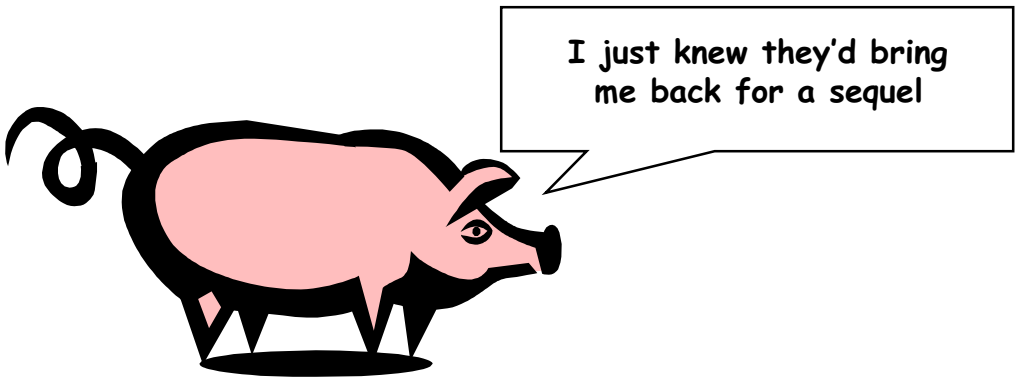
How do we find these "don't care" groups

# A Geometric Approach

It is easy to identify clusters of "don't care" bits geometrically.

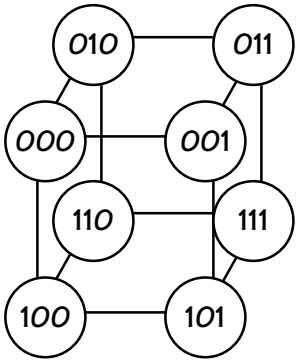
Products with these clusters, have terms that differ in exactly one bit position. So if we could arrange all possible products so that those which differ by exactly one bit are adjacent to one another then we could see these clusters easily.

We've seen such an arrangement of codings before when we computed Hamming distances.

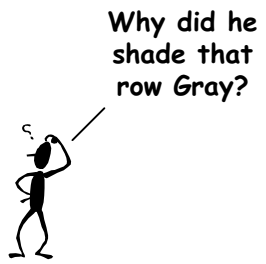


# Karnaugh Maps

Here's the layout of a 3-variable K-map filled in with the values from our truth table

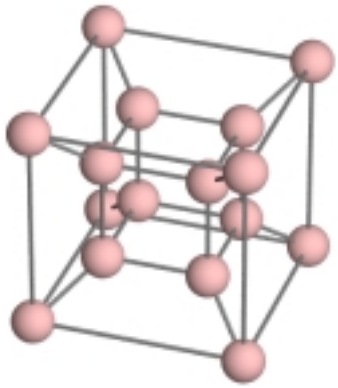



| B \ A | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| C\AB | | | | |
| 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 |




It's cyclic. The left edge is adjacent to the right edge. (It's really just a flattened out cube).

# Jump to Hyperspace



4-variable K-map for a multipurpose logic gate:

$$Y = \begin{cases} A \cdot B & \text{if } CD = 00 \\ A + B & \text{if } CD = 01 \\ \overline{B} & \text{if } CD = 10 \\ A \oplus B & \text{if } CD = 11 \end{cases}$$

| CD\AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 0 |
| 01 | 0 | 1 | 1 | 1 |
| 11 | 0 | 1 | 0 | 1 |
| 10 | 1 | 0 | 0 | 1 |

Again it's cyclic. The left edge is adjacent to the right edge,and the top is adjacent to the bottom.

# Finding Subcubes

We can identify clusters of "don't care" bits by circling adjacent subcubes of 1s. A subcube is just a lower dimensional cube.

| C\AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 |

| CD\AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 0 |
| 01 | 0 | 1 | 1 | 1 |
| 11 | 0 | 1 | 0 | 1 |
| 10 | 1 | 0 | 0 | 1 |

The best strategy is generally a greedy one.

Find, the largest subcube and circle it first. Continue circling the largest subcubes possible (even if it has some overlap with a previous one). Then proceed onto smaller and smaller subcubes until no 1s are left.

# Write Down Equations

Write down a product term for the portion of each cluster/subcube that is invariant.

| C\AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 |

$Y = \bar{C}A + CB$

| \AB CD\ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 0 |
| 01 | 0 | 1 | 1 | 1 |
| 11 | 0 | 1 | 0 | 1 |
| 10 | 1 | 0 | 0 | 1 |

$Y = AB\bar{C} + \bar{A}BD + A\bar{B}D + \bar{B}C\bar{D}$

Go home!

# Review: K-map minimization

1) Copy truth table into K-Map

2) Identify subcubes,

   selecting the largest available at each step (even if it involves some overlap) until all ones are covered

3) Write down the answer

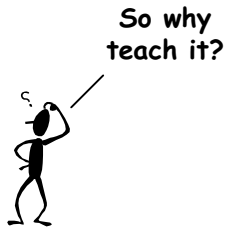
Does it always work? Not really…

   There's still a bit of art to it

| C\AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 |

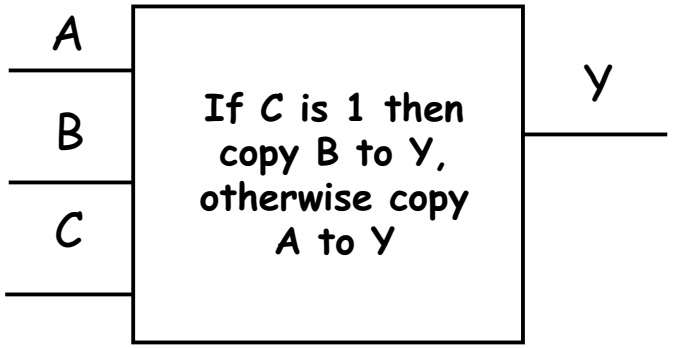
# Are K-maps really all that useful?

- They are only manageable for small circuits
  (4-5 inputs at most)
- Sometimes you pick the wrong set of subcubes
- There are better techniques
  (better for computer's that is)
- SOP realizations aren't all that relevant
- We've got gates to burn
- Low fan-in gates are better suited to current technologies that SOP (FPGAs, Standard Cells)
- Sometimes minimal circuits are glitchy
  (more about this later)
- Some important circuits aren't amenable to minimal SOP realizations



# That Gate has a Name!

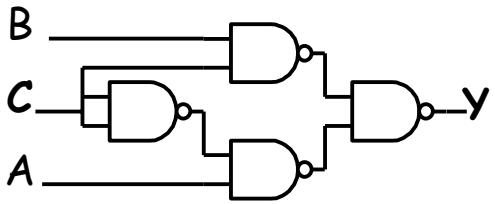
The gate we've been designing for this entire lecture is a relatively important one.



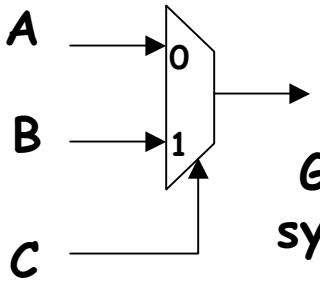

## Truth Table

| C | B | A | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

## 2-input Multiplexer




schematic




Gate symbol

# Summary

- Timing specs
  - $t_{PD}$: upper bound on time from valid inputs to valid outputs
  - $t_{CD}$: lower bound on time from invalid inputs to invalid outputs
  - If not specified, assume $t_{CD} = 0$
- Combinational logic
  - Any function that can be specified by a truth table
  - Expressed in terms of AND/OR/NOT
  - Minimally, we can get away with just 2-input NANDs or NORs
  - Max number of inputs = 4? Then use multiple levels of logic
- Sum-of-products canonical form
  - Comes directly from truth table
  - "3-level" implementation of any logic function
  - Limitations on number of inputs (fan-in) increases depth
- Next time: logic simplification, other canonical forms