Boolean Algebra and Logic Synthesis

Reminders:

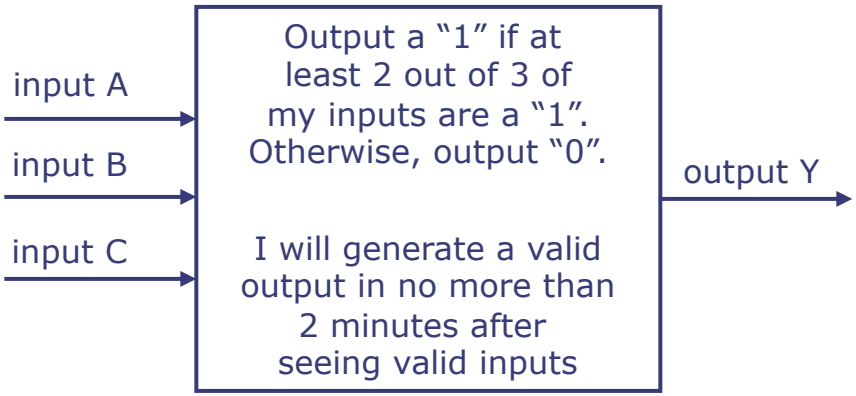
- Lab 2 due Thursday
- Bash tutorial in lab on Thursday 7:30pm

Reminder: Combinational Devices

A combinational device is a circuit element that has

Static discipline

- 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 a minimum) of a *propagation delay* (t_{PD}): an upper bound on the required time to produce valid, stable output values from an arbitrary set of valid, stable input values

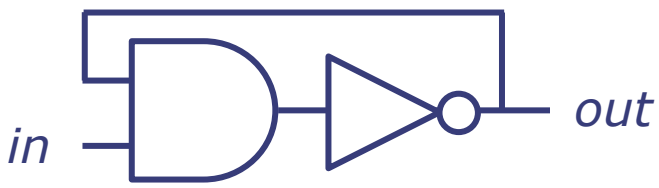


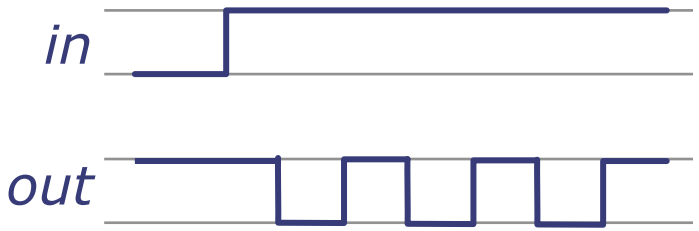
Reminder: Composing Combinational Devices

A set of interconnected elements is a combinational device if

- each circuit element is combinational
- every input is connected to exactly one output or to a constant (0 or 1)
- the circuit contains no directed cycles

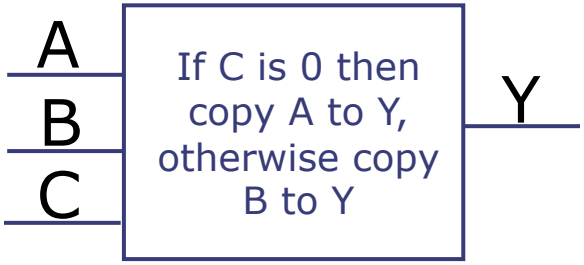
Cycles can cause feedback loops that prevent output from reaching a known or stable value! Example:





Functional Specifications

- There are many ways to specify the function of a combinational device



- We will use two systematic approaches:
  - Truth tables enumerate the output values for all possible combinations of input values
  - Boolean expressions are equations containing binary (0/1) variables and three operations: AND (·), OR (+), and NOT (overbar)

$$Y = \bar{C} \cdot A + C \cdot B$$

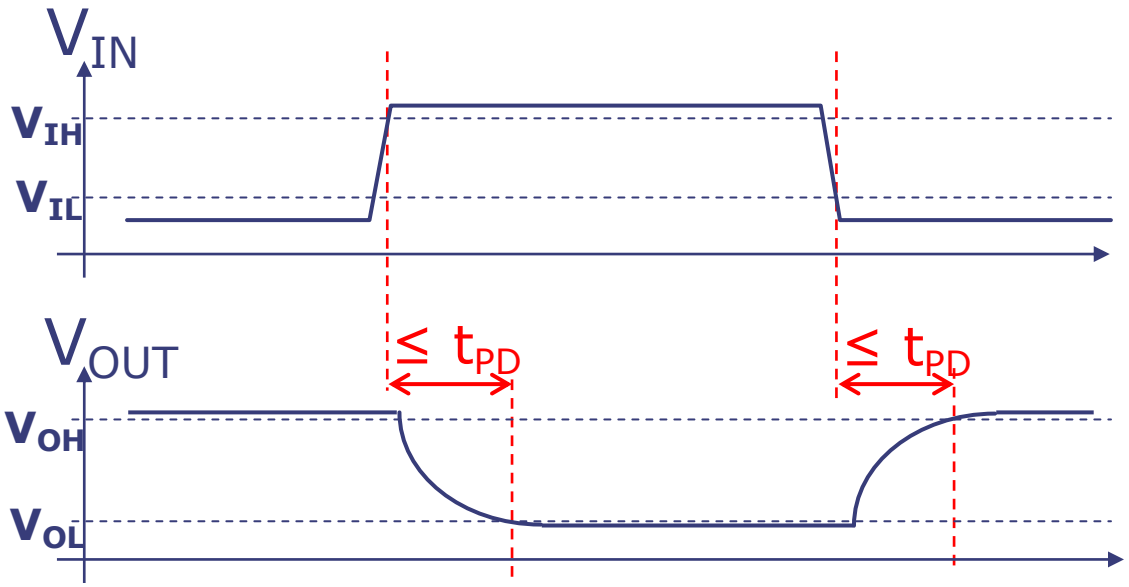
Truth Table

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

Any combinational function can be specified as a truth table or Boolean expression

Timing Specifications

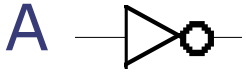
- Propagation delay (t_{PD}): An upper bound on the delay from valid inputs to valid outputs



Goal: Minimize t_{PD} !

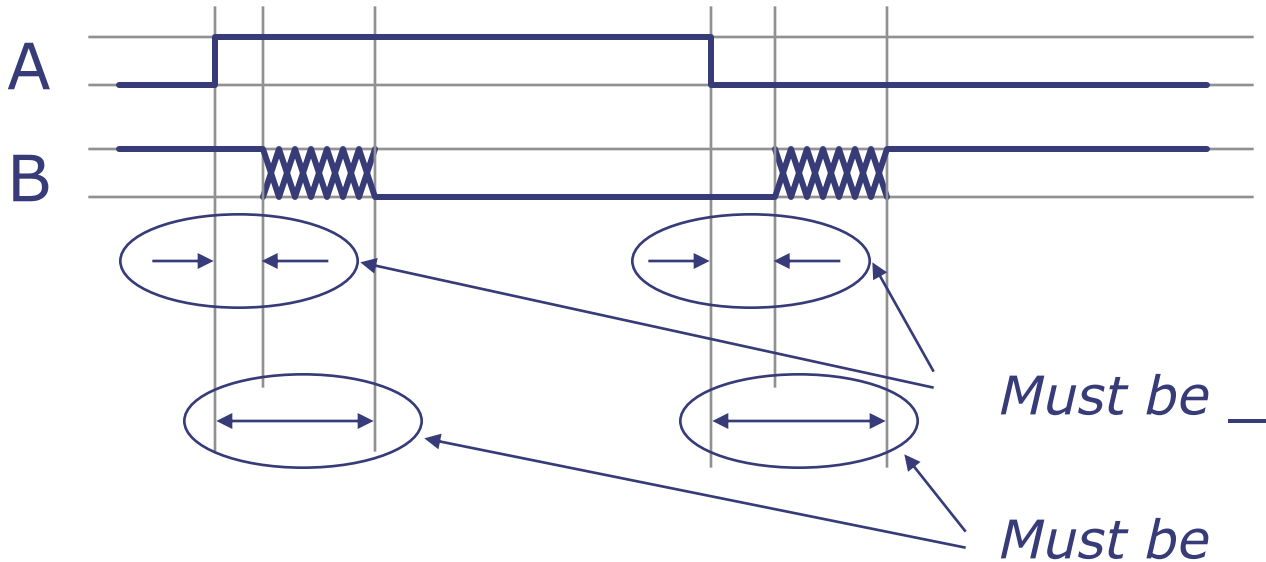
- Contamination delay (t_{CD}): A lower bound on the delay from invalid inputs to invalid outputs
  - Used later (for sequential logic), can ignore for now

The Combinational Contract



A	B
0	1
1	0

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



No promises during 

Boolean Algebra

How to interpret and manipulate Boolean expressions

Boolean Algebra

- Boolean algebra comprises
  - Two elements, 0 and 1
  - Two binary operators, AND (·) and OR (+)
  - One unary operator, NOT (overbar)

a	b	a·b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a+b
0	0	0
0	1	1
1	0	1
1	1	1

a	$\overline{a}$
0	1
1	0

- All of Boolean algebra can be derived from the definitions of AND, OR, and NOT

Boolean Algebra Axioms

- Instead of using truth tables to define AND, OR, and NOT, we can derive all of Boolean algebra using a small set of axioms:

identity	$a \cdot 1 = a$	$a + 0 = a$
null	$a \cdot 0 = 0$	$a + 1 = 1$
negation	$\overline{0} = 1$	$\overline{1} = 0$

- Duality principle: If a Boolean expression is true, then replacing $0 \leftrightarrow 1$ and AND $\leftrightarrow$ OR yields another expression that is true
  - This principle holds for the axioms → Holds for all expressions
  - Halves the number of expressions you have to learn ☺

Useful Boolean Algebra Properties

- Using the axioms, we can derive several useful properties to manipulate and simplify Boolean expressions:

commutative	$a \cdot b = b \cdot a$	$a + b = b + a$
associative	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	$a + (b + c) = (a + b) + c$
distributive	$a \cdot (b + c) = a \cdot b + a \cdot c$	$a + b \cdot c = (a + b) \cdot (a + c)$
complements	$a \cdot \overline{a} = 0$	$a + \overline{a} = 1$
absorption	$a \cdot (a + b) = a$	$a + a \cdot b = a$
reduction	$a \cdot b + a \cdot \overline{b} = a$	$(a + b) \cdot (a + \overline{b}) = a$
DeMorgan's Law	$\overline{a \cdot b} = \overline{a} + \overline{b}$	$\overline{a + b} = \overline{a} \cdot \overline{b}$

Useful Boolean Algebra Properties

- Many of these properties are easy to remember because they match the ones for integer algebra, but be aware of the differences
  - e.g., distributive property for Boolean "+" $a+b\cdot c = (a+b)\cdot(a+c)$ does not hold for integer "+"!
- To familiarize yourself with the properties, we recommend that you simply prove them
  - *Example: DeMorgan's Law*

a	b	$\overline{a\cdot b}$	$\overline{a}+\overline{b}$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Equivalence and Normal Form

- Given a truth table, it is easy to derive an equivalent Boolean expression: write a sum of products where each product term covers a single 1 in the 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

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

- This representation is called the function's normal form
  - It is unique, but there may be simpler expressions

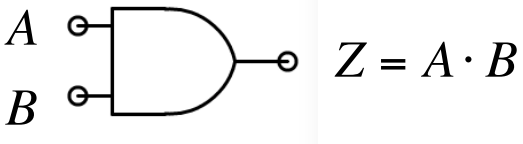
Logic Synthesis

Building logic circuits from Boolean expressions

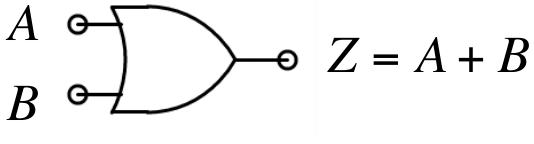
From Boolean Algebra to Gates

- A logic diagram represents a Boolean expression as a circuit schematic with logic gates and wires
- Basic logic gates:

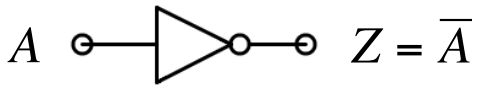
AND



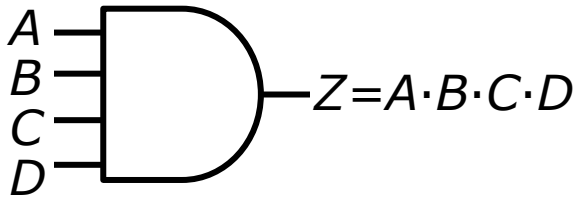
OR



Inverter



  - We often use AND and OR gates with more than two inputs



- AND, OR, and NOT are universal: They can implement any combinational function

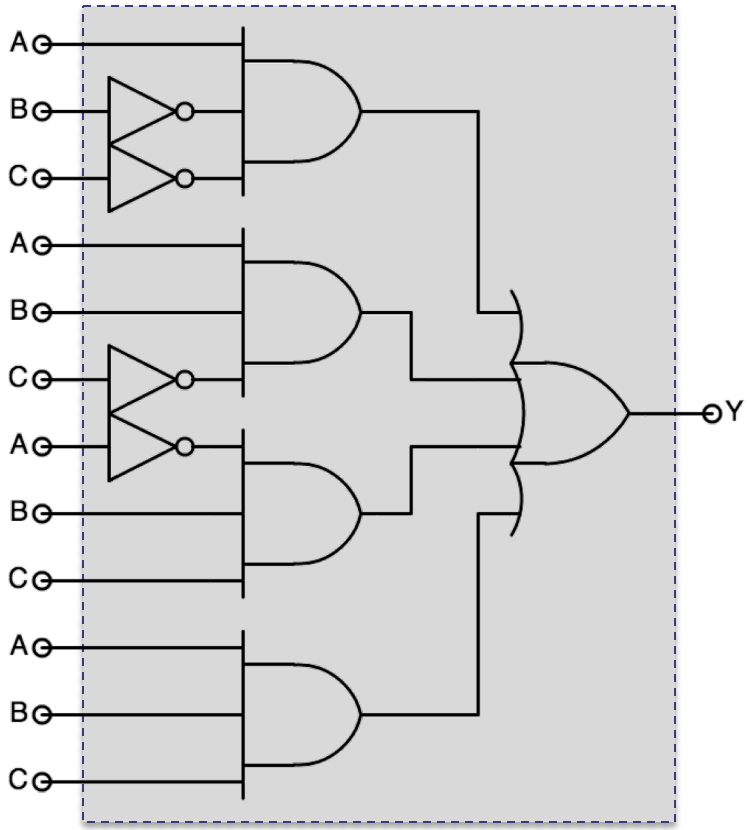
Why?

Straightforward Logic Synthesis

- We can implement any sum-of-products (SOP) Boolean expression with three levels of gates:
  1. Inverters
  2. ANDs
  3. OR

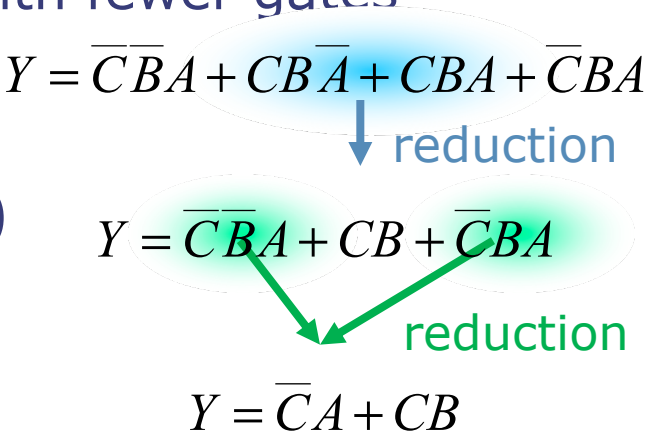
- However, we can often implement the same function with fewer gates. This requires simplifying its Boolean expression to use fewer operations.

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



Boolean Simplification of SOPs

- A minimal sum-of-products is a sum-of-products expression that has the smallest possible number of AND and OR operators
  - Unlike the normal form, it is not unique (a function may have multiple minimal SOPs)
  - Minimal SOPs can be implemented with fewer gates
- Simple algebraic manipulation (using the properties we've seen) is sufficient to minimize small expressions (3-4 variables)

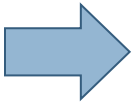


- More sophisticated techniques exist (e.g., K-maps), but we will not need them in this course

Truth Tables with "Don't Cares"

Another way to reveal simplification is to rewrite the truth table using "don't cares" (--, X, or ?) to indicate when the value of a particular input is irrelevant in determining the value of the output.

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



C	B	A	Y	
0	X	0	0	
0	X	1	1	$\rightarrow \overline{C}A$
1	0	X	0	
1	1	X	1	$\rightarrow CB$
X	0	0	0	
X	1	1	1	$\rightarrow BA$

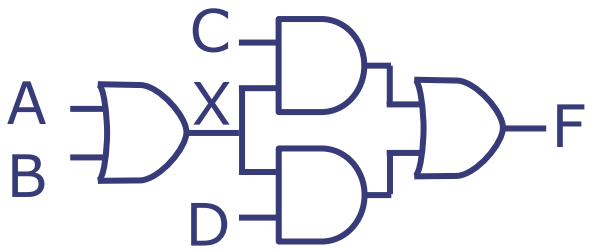
Note: Some input combinations (e.g., 000) are matched by more than one row in the "don't care" table. It would be a bug if all matching rows didn't specify the same output value!

Multi-Level Boolean Simplification

- We can often reduce the number of gates by using more logic levels than an SOP
  - Find common subexpressions and factor them out into independent variables
- Example: $F = A \cdot C + B \cdot C + A \cdot D + B \cdot D$ (minimal SOP)

$$F = (A+B) \cdot C + (A+B) \cdot D \quad \text{(not SOP)}$$

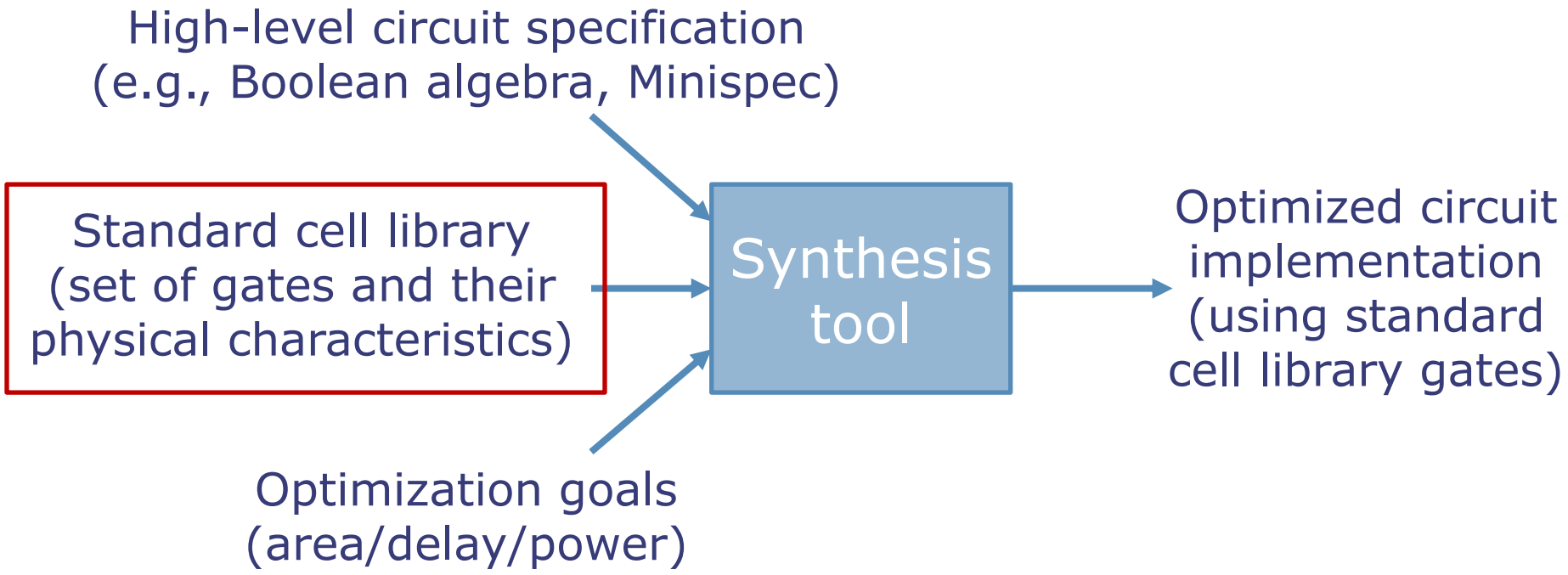
$$X = A + B$$
$$F = X \cdot C + X \cdot D$$



- Multi-level simplification has no well-defined optimum
  - Adding levels may reduce gates but increase delay

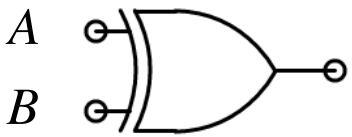
Logic Optimization

- In practice, tools use Boolean simplification and other techniques to synthesize a circuit that meets certain area, delay, and power goals:



Other Common Gates

- XOR (Exclusive-OR)

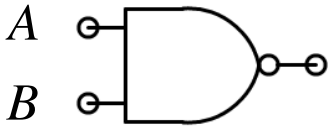


$$Z = A \oplus B = \bar{A} \cdot B + A \cdot \bar{B}$$

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

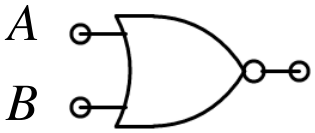
- Inverting logic

NAND



$$Z = \overline{A \cdot B}$$

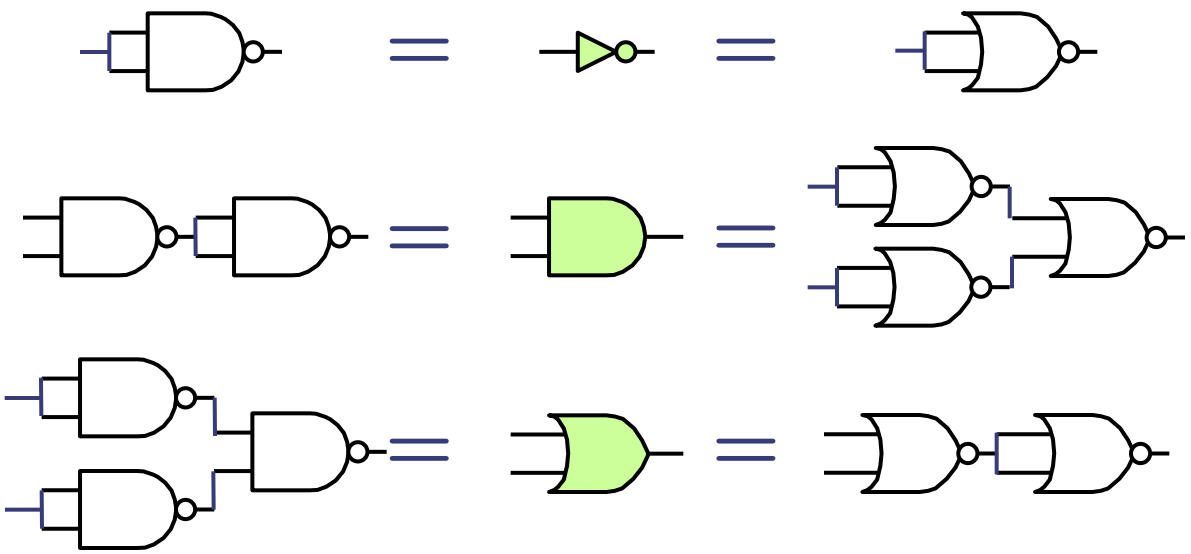
NOR



$$Z = \overline{A + B}$$

Universal Building Blocks

- NANDs and NORs are universal:



- Any logic function can be implemented using only NANDs (or, equivalently, NORs)

Standard Cell Library

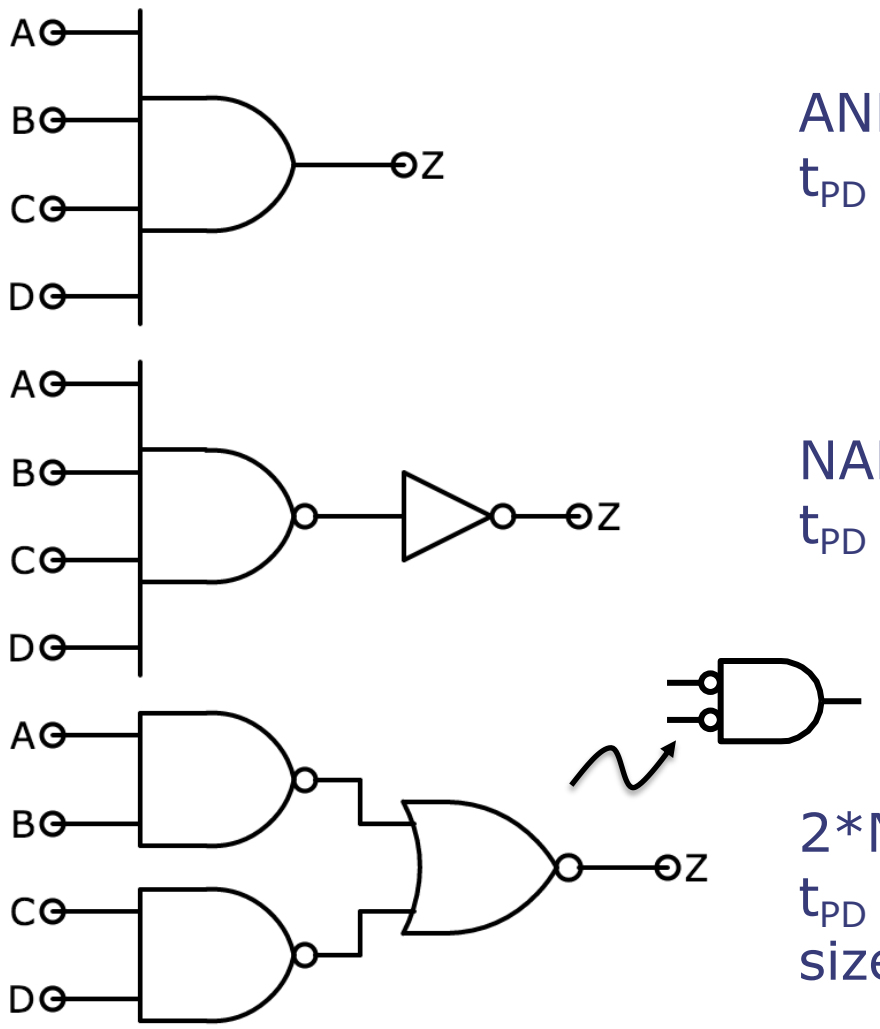
- Library of gates and their physical characteristics
- Example:

Gate	Delay (ps)	Area (μ^2)
Inverter	20	10
Buffer	40	20
AND2	50	25
NAND2	30	15
OR2	55	26
NOR2	35	16
AND4	90	40
NAND4	70	30
OR4	100	42
NOR4	80	32

Observations:

1. In current technology (CMOS), inverting gates are faster and smaller
2. Delay and area grow with number of inputs

Design Tradeoffs: Delay vs Size



AND4:
t_{PD} = 90 ps, size = 40μ²

NAND4 + INV:
t_{PD} = 90 ps, size = 40μ²

Demorgan's Laws:

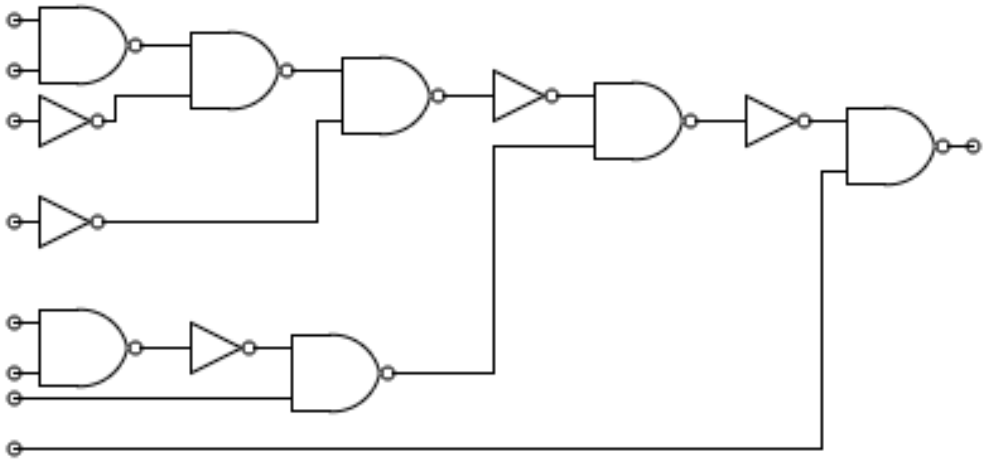
$$\overline{A} \cdot \overline{B} = \overline{A + B}$$

$$\overline{A} + \overline{B} = \overline{A \cdot B}$$

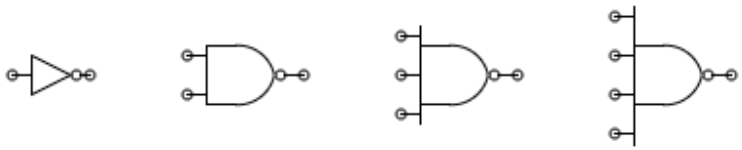
2*NAND2 + NOR2:
t_{PD} = 1 NAND2 + NOR2 = 65 ps,
size = 2 NAND2 + NOR2 = 46μ²

Example: Mapping a Circuit to a Standard Cell Library

Find an implementation of a circuit, e.g.,



Using gates from a standard cell library, e.g.,

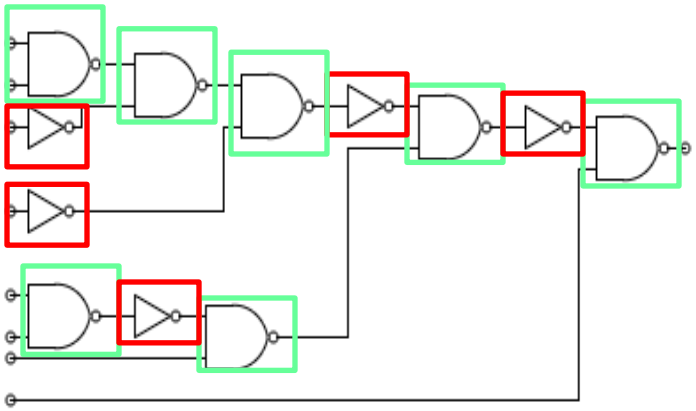


Area	2	3	4	5

That optimizes for some goal, e.g., minimum area

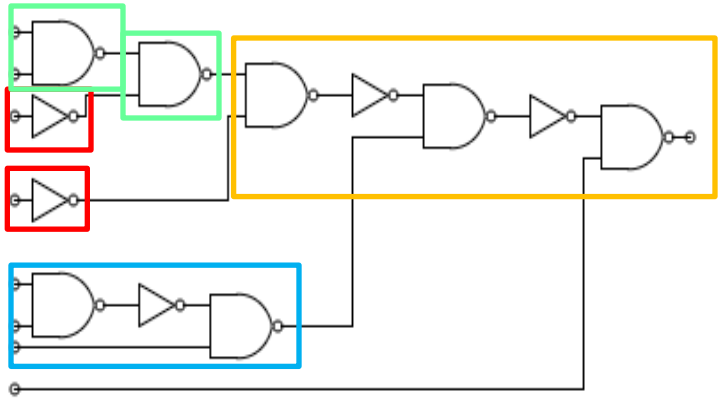
Example: Mapping a Circuit to a Standard Cell Library

Possible implementations:



7 NAND2 (3) = 21
5 INV (2) = 10

Total area cost: 31



2 INV = 4
2 NAND2 = 6
1 NAND3 = 4
1 NAND4 = 5

Total area cost: 19

Logic Optimization Takeaways

- Synthesizing an optimized circuit is a very complex problem
  - Boolean simplification
  - Mapping to cell libraries with many gates
  - Multidimensional tradeoffs (e.g., minimize area-delay-power product)

- Infeasible to do by hand for all but the smallest circuits!

- Instead, hardware designers write circuits in a hardware description language, and use a synthesis tool to derive optimized implementations

Summary

- Any combinational (Boolean) function can be specified by a truth table or a Boolean expression (binary literals and AND, OR, NOT, which form a Boolean algebra)
- Any combinational function can be expressed as a sum-of-products (SOP) and implemented with three levels of logic gates (NOTs, ANDs, OR)
- Boolean simplification (finding a minimal SOP, multi-level simplification) results in simpler circuits
- There are MANY design tradeoffs in mapping Boolean functions to gates. We will use synthesis tools to find optimized circuit implementations

Thank you!

Next lecture:
CMOS