EE2026
Digital Design

LOGIC GATES

Massimo ALIOTO
Dept of Electrical and Computer Engineering
Email: massimo.alioto@nus.edu.sg

Get to know the latest silicon system breakthroughs from our labs in 1-minute video demos

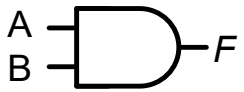
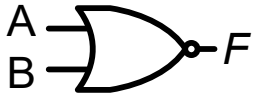
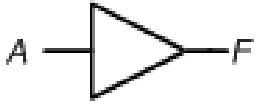
Explore | Follow | Discover more | What's happening | Stay tuned | Be connected | Communicate

Outline

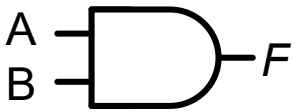
- Logic gate introduction
  - AND/NAND, OR/NOR, NOT/buffer, XOR/XNOR
- Levels of abstraction: Boolean function, truth table, graphical, Verilog
- Implementation of Boolean function using gates
- Design simplification via algebraic manipulations
- Positive and negative logic
- Implementation of Boolean function with gate-level netlist

Introduction to Logic Gates

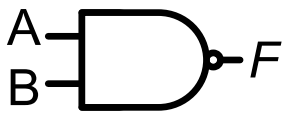
- Logic gates are digital circuits implementing fundamental Boolean operators or some simple combination of them
  - Abstraction: actually made up of transistors (not shown here), closer to physical implementation of digital systems

logic gate	symbol	function (*F*)	logic gate	symbol	function (*F*)
AND		$A \cdot B$	NAND		$\overline{A \cdot B}$
OR		$A + B$	NOR		$\overline{A + B}$
NOT		$\bar{A}$	Buffer		A

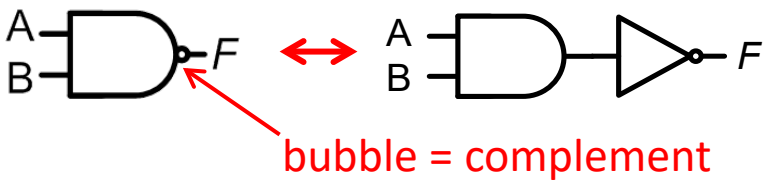
AND and NAND Gates



$$F = A \cdot B$$



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



Truth table (AND, NAND)

A	B	$A \cdot B$	$\overline{A \cdot B}$
0	0	0	1
1	0	0	1
0	1	0	1
1	1	1	0

AND

- ***F*** is 1 only when both ***A*** and ***B*** are 1

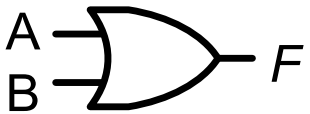
```
module andgate(A, B, F);
    input A, B;
    output F;
    assign F = A & B;
endmodule
```

NAND

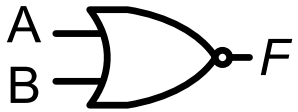
- ***F*** is 0 only if both ***A*** and ***B*** are 1

```
module nandgate(A, B, F);
    input A, B;
    output F;
    assign F = ~(A & B);
endmodule
```

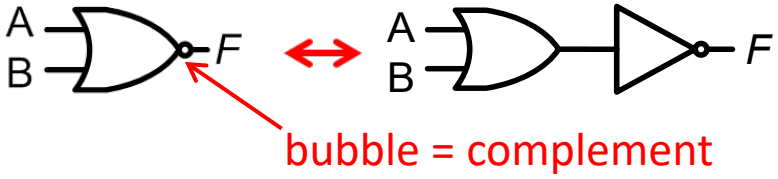
OR and NOR Gates



$$F = A + B$$



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



Truth table (OR, NOR)

A	B	$A + B$	$\overline{A + B}$
0	0	0	1
1	0	1	0
0	1	1	0
1	1	1	0

OR

- ***F*** is 1 when either ***A*** or ***B*** are 1

NOR

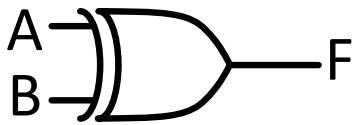
- ***F*** is 1 only if both ***A*** and ***B*** are 0

```
module orgate(A, B, F);
    input A, B;
    output F;
    assign F = A | B;
endmodule
```

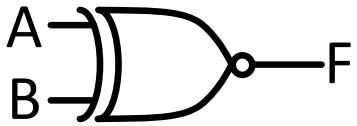
```
module norgate(A, B, F);
    input A, B;
    output F;
    assign F = ~(A | B);
endmodule
```

XOR and XNOR Gates

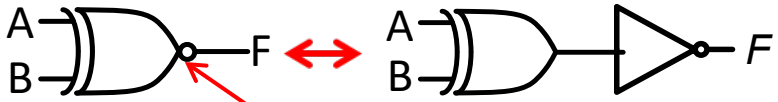
- Logic gate that is not fundamental in Boolean algebra
  - But very useful (e.g., arithmetic circuits – see week 4)



$$F = A\overline{B} + \overline{A}B = A \oplus B$$



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



XOR

bubble = complement

Truth Table (XOR, XNOR)

A	B	$A \oplus B$	$\overline{A \oplus B}$
0	0	0	1
1	0	1	0
0	1	1	0
1	1	0	1

XNOR

- ***F*** is 1 when either ***A*** or ***B*** (exclusively) are 1, or equivalently different from each other

```
module xorgate(A, B, F);
    input A, B;
    output F;
    assign F = A ^ B;
endmodule
```

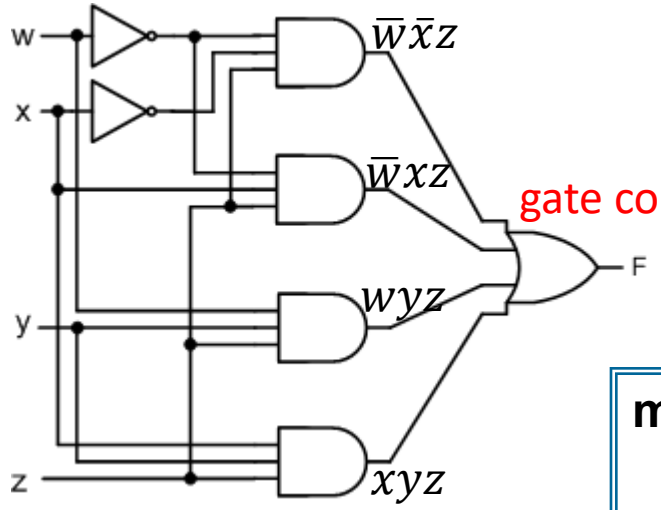
- ***F*** is 1 when either ***A*** or ***B*** (exclusively) are 0, or equivalently equal

```
module xnorgate(A, B, F);
    input A, B;
    output F;
    assign F = ~(A ^ B);
endmodule
```

Implementation of Boolean Functions with Logic Gates

- Translate Boolean function into gate-level implementation
  - Logic gates as building blocks of any digital system
- Start simple: SOP form → gate-level design
  - Example of SOP with constraint: max number of logic gate inputs is 4 (fan-in)

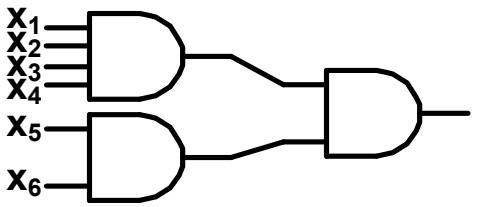
$$F(w,x,y,z) = \bar{w}\bar{x}z + \bar{w}xz + wyz + xyz$$



gate count = 7

if AND5 or more is needed: two-level ANDing (same for OR):

$x_1 \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5 \cdot x_6 = (x_1 \cdot x_2 \cdot x_3 \cdot x_4) \cdot (x_5 \cdot x_6)$



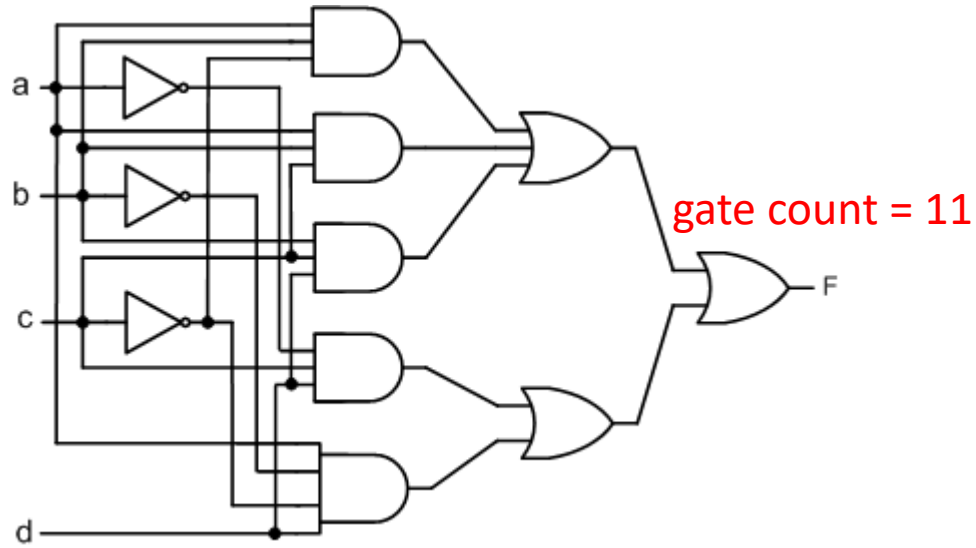
parentheses (~w & x & z) not needed in SOP, as precedence order is ~, &, ^, |

```
module func(w,x,y,z,F);
    input w, x, y, z;
    output F;
    assign F = ~w & ~x & z | ~w & x & z | w & y & z | x & y & z;
endmodule
```

Implementation of Boolean Functions with Logic Gates

- Another example of SOP with constraint: max number of logic gate inputs is 4

$$F(a,b,c,d) = ab\bar{c} + abc + bcd + \bar{a}cd + a\bar{b}\bar{c}d$$

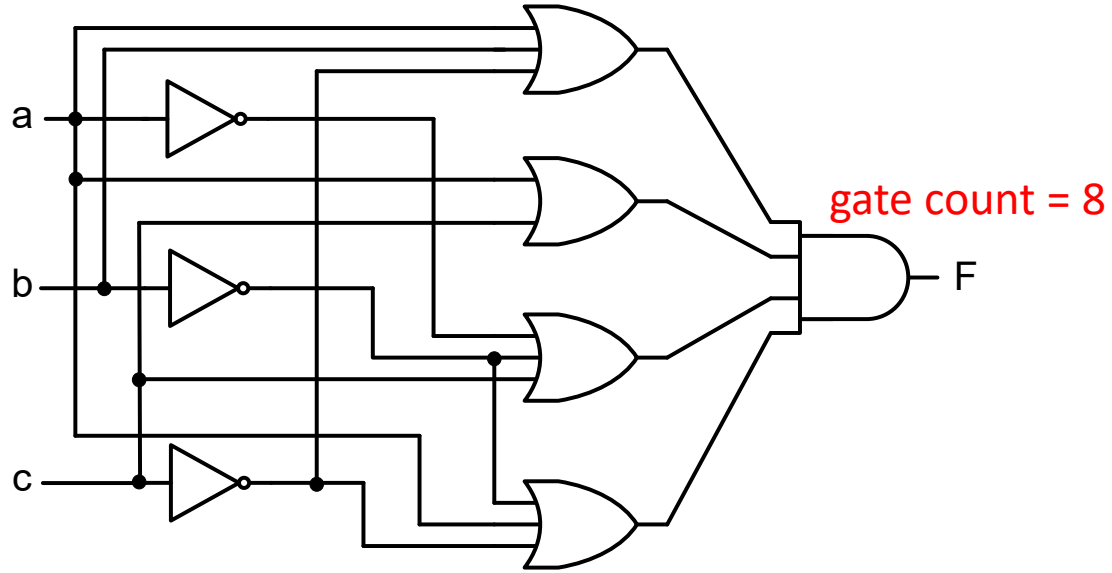



```
module func(a,b,c,d,F);
    input a, b, c, d;
    output F;
    assign F = a & b & ~c | a & b & c | b & c & d | ~a & c & d | a & ~b & ~c & d;
endmodule
```

Implementation of Boolean Functions with Logic Gates

- Example of POS with constraint: max number of logic gate inputs is 4

$$F(a,b,c) = (a + b + \bar{c})(a + c)(\bar{a} + \bar{b} + c)(a + \bar{b} + \bar{c})$$



parentheses needed in POS,
as precedence order is ~, &, ^, |

```
module func(a,b,c,F);
    input a, b, c;
    output F;
    assign F = (a | b | ~c) & (a | c) & (~a | ~b | c) & (a | ~b | ~c);
endmodule
```

Boolean Function Simplification

- To reduce the hardware cost, the Boolean function must be simplified before gate-level implementation
  - Eliminate redundancies, minimize gate count
- Definition of simplified Boolean Function
  - It contains a minimal number of **terms** and **literals** in each term, such that no other expression with fewer literals and terms will represent the original function

literals (variables, complemented or not)

$$F_2(A,B,C) = \bar{A}B\bar{C} + A\bar{B} + \bar{B}C + AB\bar{C}$$

terms

literals

$$F_2(A,B,C) = (A + B + C) \cdot (A + \bar{B} + \bar{C})$$

terms

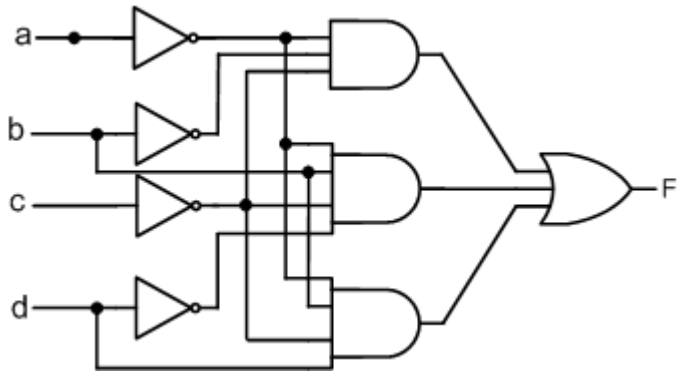
- Simplification can be carried out via
  - Algebraic manipulations using postulates and theorems
  - Karnaugh maps

Boolean Function Simplification using Algebraic Manipulations

- Example: SOP

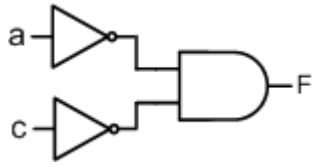
$$
\begin{aligned}
F(a,b,c,d) &= \bar{a}\bar{b}\bar{c} + \bar{a}b\bar{c}\bar{d} + \bar{a}b\bar{c}d \\
&= \bar{a}\bar{b}\bar{c} + \bar{a}b\bar{c}(\bar{d} + d) \leftarrow (A + \bar{A} = 1) \\
&= \bar{a}\bar{c}(\bar{b} + b) \leftarrow (A + \bar{A} = 1) \\
&= \bar{a}\bar{c} \leftarrow (A + \bar{A} = 1)
\end{aligned}
$$

Before simplification



gate count = 8

After simplification



gate count = 3

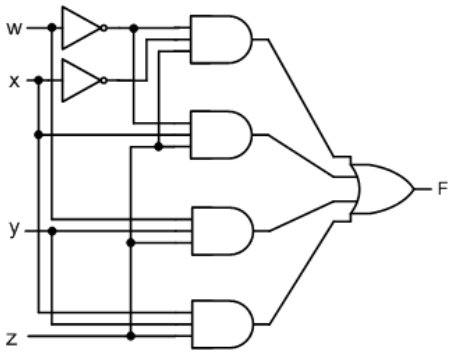
(62.5% reduction!)

Boolean Function Simplification using Algebraic Manipulations

- Another example: SOP

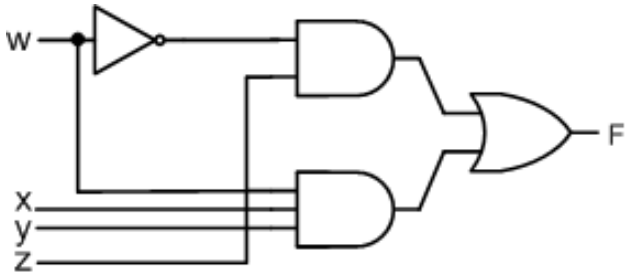
$$
\begin{aligned}
F(x,y,z) &= \bar{w}\bar{x}z + \bar{w}xz + xyz + wxy \\
&= \bar{w}z(\bar{x} + x) + w(xy) + z(xy) && \leftarrow (A + \bar{A} = 1) \\
&= \bar{w}z + w(xy) + z(xy) && \leftarrow (A + \bar{A} = 1) \\
&= \bar{w}z + wxy && \leftarrow (AB + \bar{A}C + BC = AB + \bar{A}C) \text{ - consensus}
\end{aligned}
$$

Before simplification



gate count = 7

After simplification



gate count = 4

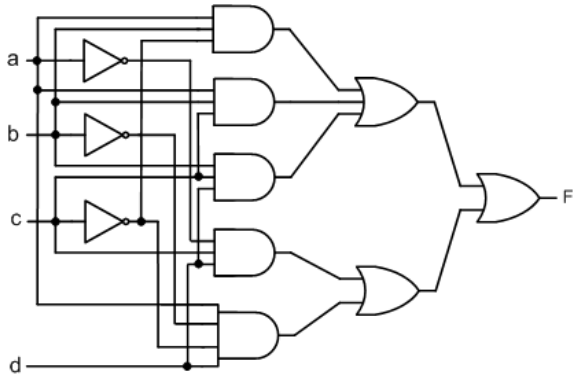
(43% reduction!)

Boolean Function Simplification using Algebraic Manipulations

- Another example: SOP

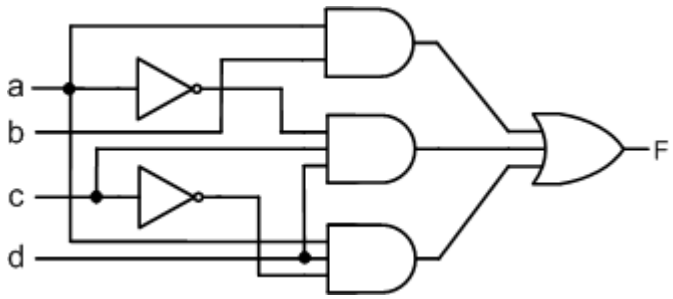
$$
\begin{aligned}
F(a,b,c,d) &= ab\bar{c} + abc + bcd + \bar{a}cd + a\bar{b}\bar{c}d \\
&= ab(\bar{c} + c) + bcd + \bar{a}cd + a\bar{b}\bar{c}d && \leftarrow (A + \bar{A} = 1) \\
&= a\left[b + \bar{b}(\bar{c}d)\right] + bcd + \bar{a}cd && \leftarrow (A + \bar{A} \cdot B = A + B) \\
&= a(b + \bar{c}d) + bcd + \bar{a}cd \\
&= [ab + \bar{a}(cd) + b(cd)] + a\bar{c}d && \leftarrow (AB + \bar{A}C + BC = AB + \bar{A}C) \text{ - consensus} \\
&= ab + \bar{a}cd + a\bar{c}d
\end{aligned}
$$

Before simplification



gate count = 11

After simplification



gate count = 6

(45.5% reduction!)

Boolean Function Simplification using Algebraic Manipulations

- Disadvantage of algebraic manipulations: not systematic, tedious, no guarantee of minimal function
- Proposed procedure to somewhat minimize Boolean functions using algebraic manipulations

1) $AB + A\bar{B} = A$ (Logical adjacency)
2) $A + \bar{A} \cdot B = A + B$
3) $AB + \bar{A}C + BC = AB + \bar{A}C$ (Consensus)

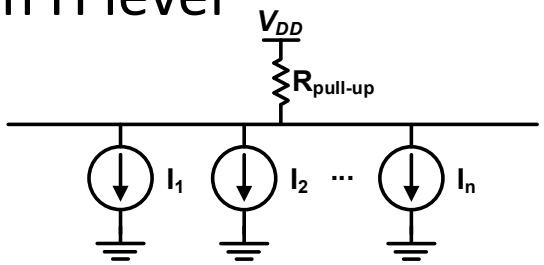
- Apply (1) until it cannot be applied further
- Apply (2) until it cannot be applied further
- Go back to (1) and then (2) until they can no longer be applied
- Apply (3) until it cannot be applied further
- Go back to (1), (2) and then (3) until none of them can be applied

Interfaces and Legacy Systems: Negative Logic

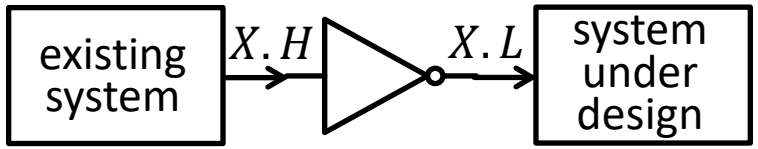
- Physical logic gates: voltage input and output levels low (L) and high (H)
  - Binary values 0 and 1 can be mapped in two ways
    - positive logic ("active high"): ex. X.H
    - negative logic ("active low"): ex. X.L
    - convert one to another via simple inverter

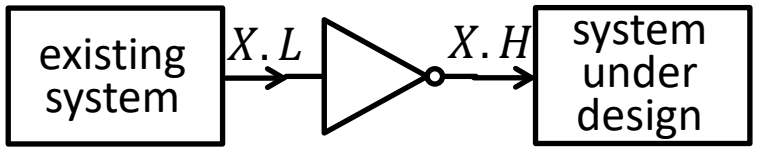
voltage level	positive logic value (X.H)	negative logic value (X.L)
H	1	0
L	0	1

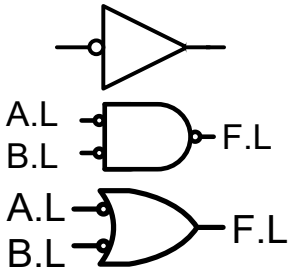
  - In the (distant) past, active-low preferred because of lower power in H level
    - normally-off signals (e.g., reset) were set to H (e.g., TTL logic, open-collector)
    - it was easier to merge normally-off signals into one occasionally-on signal
    - today used only in system resets, interrupts and I^2C busses



  - Today (including FPGAs), no preference
    - think in terms of positive logic, negate any active-low input/output signal if necessary
    - graphically add "bubble" to signals to remind about the complement







Bubble Pushing Rule

- Graphical interpretation of De Morgan's law(s), useful to
  - Manipulate gate-level netlists directly without Boolean expressions (tedious)
  - Transform logic gates from one type (e.g., NAND) into another one (NOR)
- Graphically, bubble = complement
  - If a bubble is needed, it can always be created anywhere (but in pairs)
  - Vice versa, two adjacent bubbles can always be dissolved

$\overline{\overline{A}} = A$

$A = \overline{\overline{A}}$

  - Bubbles at input of AND gate can be "pushed" at its output, and the gate is transformed into a NOR gate (similarly, NAND becomes OR)

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

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

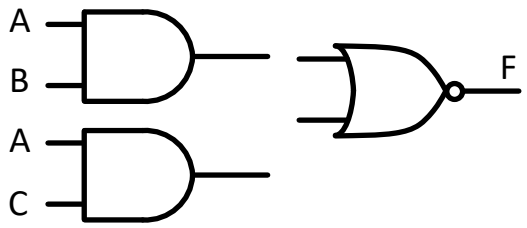
and vice versa (push from output to input)

Bubble Pushing Rule

- Example: implement Boolean function using only NOR gates and inverter gates

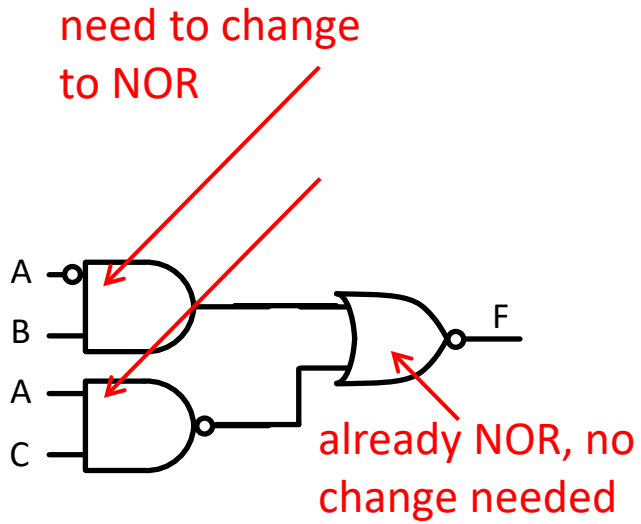
$$F = \overline{\left(\overline{A} \cdot B + \overline{A \cdot C}\right)}$$

Step 1: place logic gates



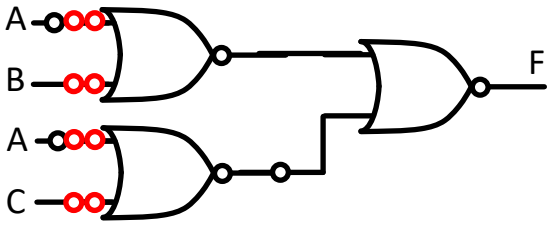
Step 2: add bubbles and connect

(add the complement where needed for the correct function)



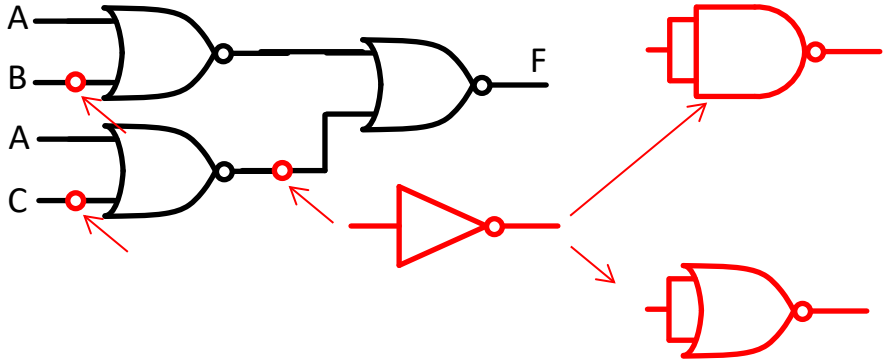
Step 3: bubble pushing

(i) Replace gates with targeted ones
(ii) balance the bubbles using inverters to maintain the correct functionality



Step 4: simplify

(eliminate redundant pairs of bubbles)



A	B	$\overline{A \cdot B}$
0	0	1
1	1	0

A	B	$\overline{A + B}$
0	0	1
1	1	0

Bubble Pushing Rule

- Another example: implement Boolean function with logic gates with active-low output, using only NAND gates (no inverter gates)

In Summary

- Logic gates
  - Building blocks of any digital system
  - Different levels of abstraction (Boolean, truth table, graphical, Verilog)
- Implementation of Boolean function using gates
- Design simplification via algebraic manipulations (limitations motivate introduction of K-maps)
- Positive $\rightarrow$ negative logic conversion for legacy systems
- Bubble pushing for direct manipulation of gate-level netlists (with no intermediate Boolean manipulations)