

# Logic Synthesis

## -From RTL to Gate-Level

Song Chen

Dept. of Electronic Science and Technology

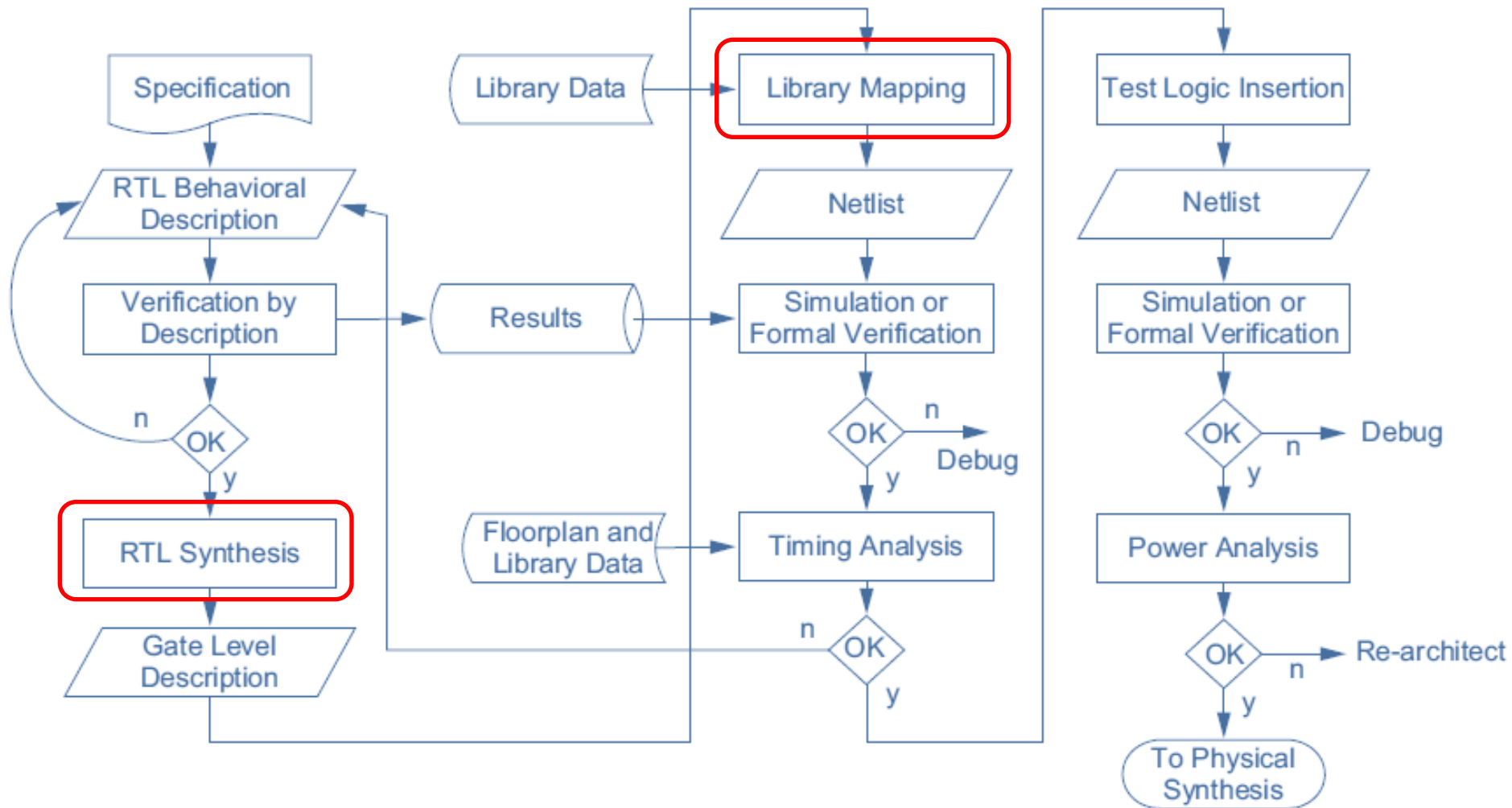
Oct. 29, 2015

<http://staff.ustc.edu.cn/~songch/da-ug.htm>

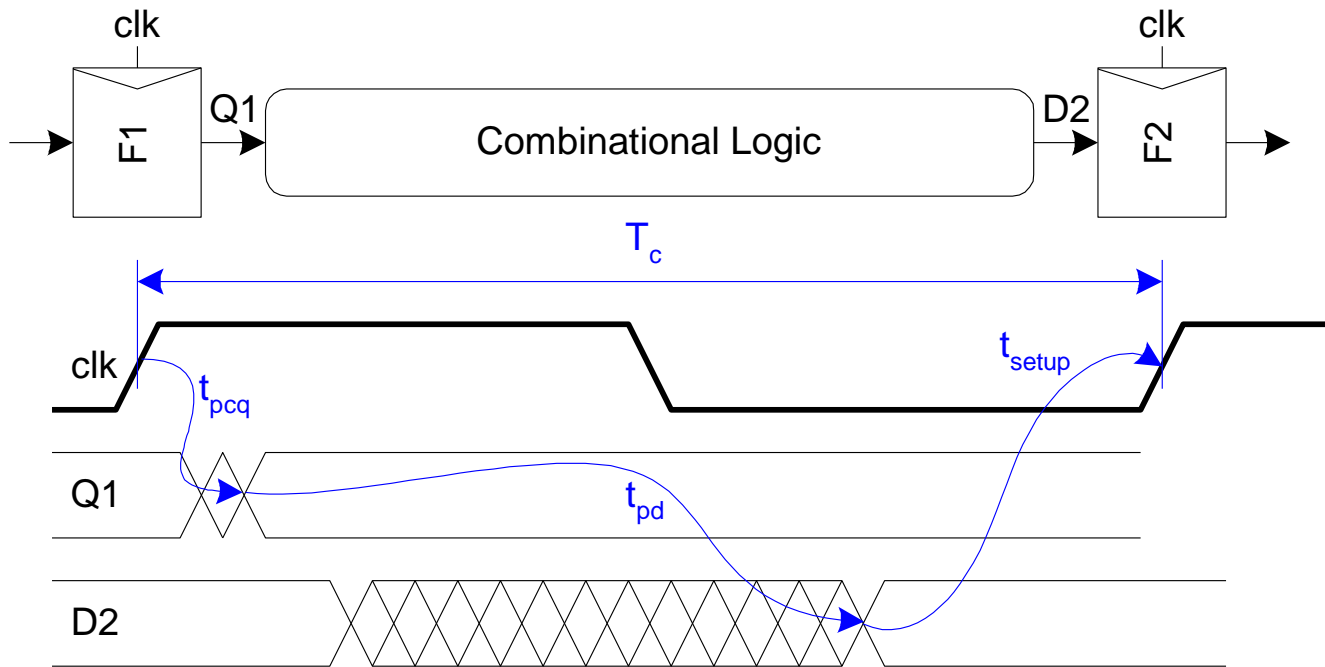
# Outline

- Logic Synthesis
  - Logic Optimization: Technology Independent Optimization
    - Two-level logic optimization
    - Multi-level logic optimization
  - Technology Mapping: Technology Dependent Optimization
    - Mapping to Standard Cells (Standard-Cell based ASIC)
    - Mapping to Look-up tables (FPGA, Optional)
- Appendix – CMOS Gates

# Logic Synthesis Flow

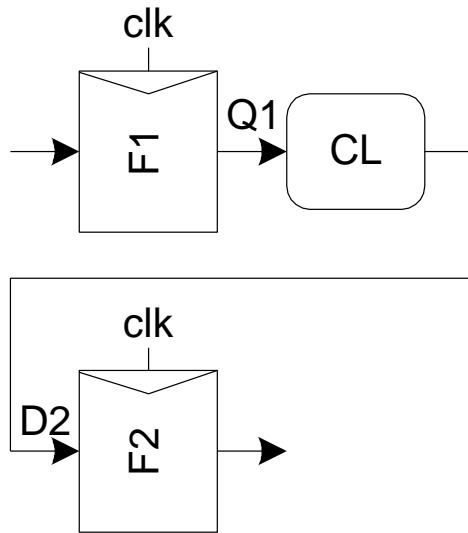


# Max-Delay: Flip-Flops

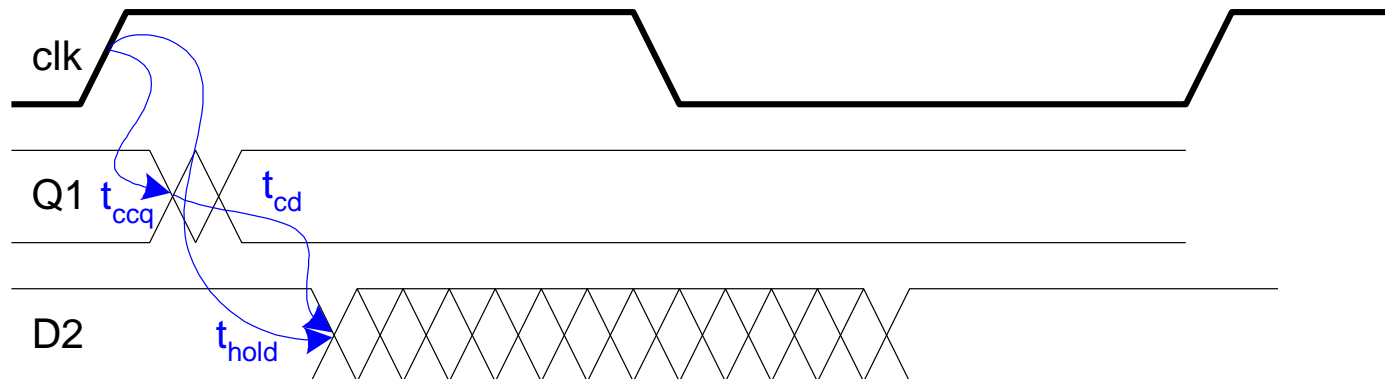


$$t_{pd} \leq T_c - \underbrace{(t_{setup} + t_{pcq})}_{\text{sequencing overhead}}$$

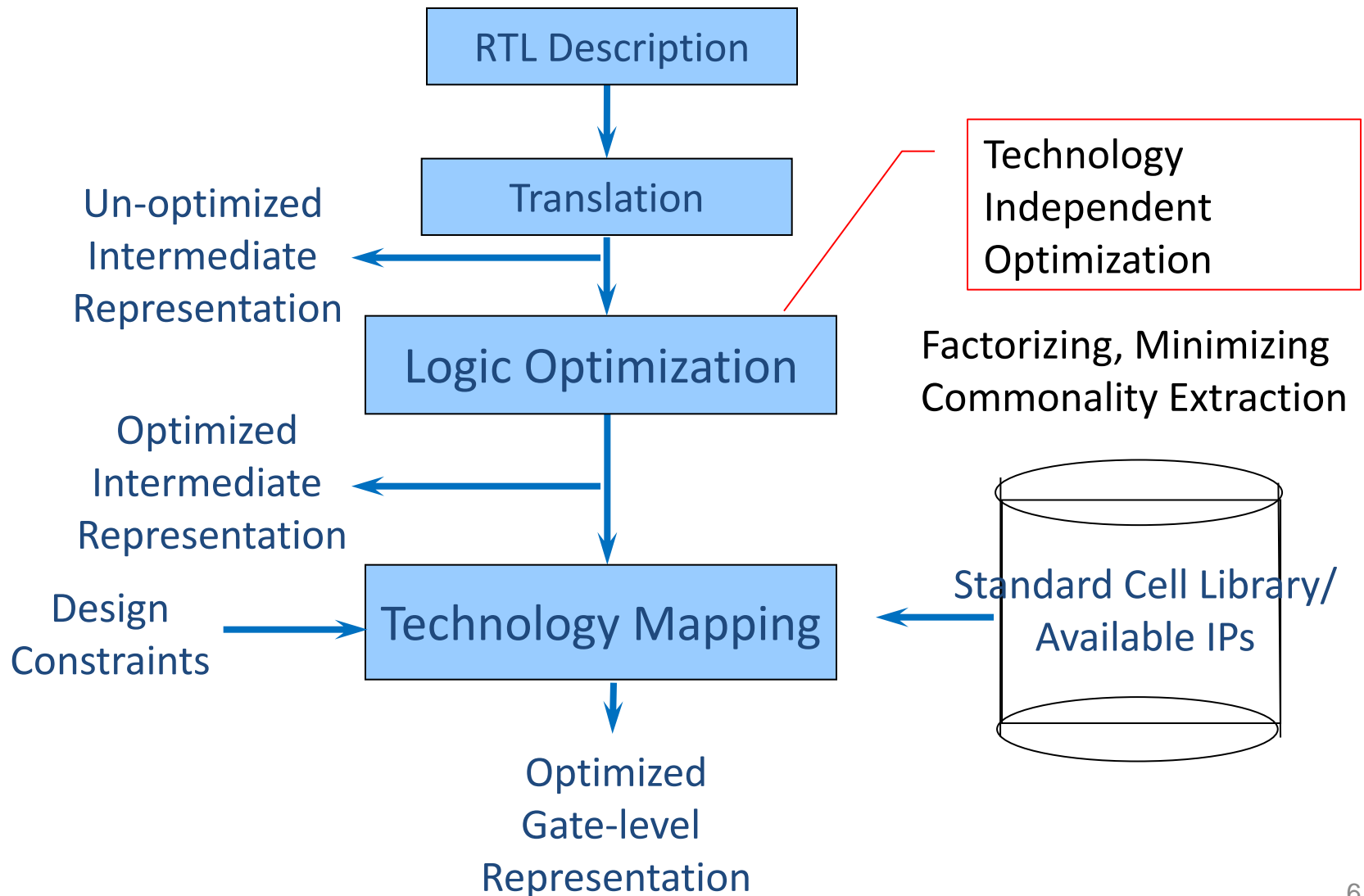
# Min-Delay: Flip-Flops



$$t_{cd} \geq t_{\text{hold}} - t_{ccq}$$



# Logic Synthesis Flow from RTL to gates



# Why logic optimization?

Most important metrics for a chip design:

- **Area:**
  - Size affects manufacturing and packaging costs
- **Performance** (frequency):
  - Does chip meet market/user performance goals?
- **Power:**
  - Peak power affects packaging cost (current supply, heat removal)
  - Energy usage affects battery life

**Area** reduction, **power** reduction, and **delay** reduction improves design

# Two-Level Logic Optimization

- Two-level optimization
  - **Sum of Product (SOP),**
    - First level AND, Second level OR
    - Product of Sum (POS)
  - Smaller/Faster Implementation:
    - Reducing the number of product terms in an equation
    - reducing the size of each product term
  - Find an SOP representation for a Boolean function to minimize (**NP-Hard**)
    - **Number of product terms**
    - **Number of literals, or**
    - **Combination of both.**



# Two-level Boolean Minimization

- Two Steps
  - Generate the set of prime product-terms for the function
  - Select a minimum set of prime terms to cover the function
- Quine-McCluskey Method

# Prime Term Generation

- Express your Boolean function
  - Using **0-terms** (product terms with no don't care entries).
- Include only those entries where the output of the function is 1
  - Label each entry with it's decimal equivalent
- Look for pairs of **0-terms** that
  - Differ in only one bit position and
  - Merge them in a **1-term** (i.e., a term that has exactly one '–' entry).

<i>W</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>label</i>
0	0	0	0	0
0	1	0	1	5
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	1	0	14
1	1	1	1	15

**0-terms:**

0, 8	–000
5, 7	01–1
7, 15	–111
8, 9	100–
8, 10	10–0
9, 11	10–1
10, 11	101–
10, 14	1–10
11, 15	1–11
14, 15	111–

**1-terms:**

# Prime Term Generation

- **Next:**
  - Examine 1-terms in pairs
  - Merge into 2-terms, ...
  - Mark k-terms that get merged into (k+1) terms so we can discard them later
- **Label unmerged terms: these terms are prime!**

<i>W</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>label</i>
0	0	0	0	0
0	1	0	1	5
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	1	0	14
1	1	1	1	15

**0-terms:**

0, 8	-000
5, 7	01-1
7, 15	-111
8, 9	100-
8, 10	10-0
9, 11	10-1
10, 11	101-
10, 14	1-10
11, 15	1-11
14, 15	111-

**1-terms:**

**3-terms:** none!

**2-terms:** 8, 9, 10, 11 10--  
10, 11, 14, 15 1-1-

# Prime Term Table

- An “X” in the prime term table in row  $r$  and column  $c$  signifies that the 0-term corresponding to row  $r$  is contained by the prime corresponding to column  $c$ .

- Goal: select the minimum set of primes (columns) such that there is at least one “X” in every row. This is the classical minimum covering problem.

	A	B	C	D	E	
0000	X	.	.	.	.	→ A is essential
0101	.	X	.	.	.	→ B is essential
0111	.	X	X	.	.	
1000	X	.	.	X	.	
1001	.	.	.	X	.	→ D is essential
1010	.	.	.	X	X	
1011	.	.	.	X	X	
1110	.	.	.	.	X	→ E is essential
1111	.	.	X	.	X	

- Each row with a single X signifies an **essential prime term** since any prime implementation will have to include that prime term because the corresponding 0-term is not contained in any other prime.
- In this example the essential primes “cover” all the 0-terms<sub>2</sub>

# Dominated Columns

- Some functions **may not have essential primes** (Fig. 1),
- So make arbitrary selection of first prime in cover, say A (Fig. 2).
- A column **U** of a prime term table dominates **V** if **U** contains every row contained in **V**. Delete the dominated columns (Fig. 3).

1. Prime table

	A	B	C	D	E	F	G	H
0000	X	.	.	.	.	.	.	X
0001	X	X	.	.	.	.	.	.
0101	.	X	X	.	.	.	.	.
0111	.	.	X	X	.	.	.	.
1000	.	.	.	.	.	.	X	X
1010	.	.	.	.	.	X	X	.
1110	.	.	.	.	X	X	.	.
1111	.	.	.	X	X	.	.	.

2. Table with A selected

	B	C	D	E	F	G	H
0101	X	X	.	.	.	.	.
0111	.	X	X	.	.	.	.
1000	.	.	.	.	.	X	X
1010	.	.	.	.	X	X	.
1110	.	.	.	X	X	.	.
1111	.	.	X	X	.	.	.

C dominates B,  
G dominates H

3. Table with B & H removed

	C	D	E	F	G
0101	X	.	.	.	→ C is essential
0111	X	X	.	.	.
1000	.	.	.	.	→ X → G is essential
1010	.	.	.	X	X
1110	.	.	X	X	.
1111	.	X	X	.	.

Selecting C and G  
shows that only E is  
needed to complete  
the cover

- This gives a prime cover of {A, C, E, G}. Now backtrack to our choice of A and explore a different (arbitrary) first choice; repeat, remembering minimum cover found during search.

# The Quine-McCluskey Method

- **The input to the procedure is the prime term table T**
- **1.** Delete the **dominated primes** (columns) in T. Detect **essential primes** in T by checking to see if any 0-term is contained by a single prime. Add these essential primes to the selected set. Repeat until no new essential primes are detected.
- **2.** If the size of the selected set of primes equals or exceeds the best solution thus far return from this level of recursion. If there are no elements left to be contained, declare the selected set as the best solution recorded thus far.
- **3.** Heuristically select a prime.
- **4.** **Add the chosen prime** to the selected set and create a new table by deleting the prime and all 0terms that are contained by this prime in the original table. Set T to this new table and go to Step 1.
- Then, create a new table by deleting the chosen prime from the original table **without adding it to the selected set**. No 0-terms are deleted from the original table. Set T to new table and go to Step 1.

# The Quine-McCluskey Method

- **The good news**
  - This technique generalizes to multi-output functions (relations).
- **The bad news**
  - The search time grows as  $2^{(2^N)}$  where  $N$  is the number of inputs.
  - So most modern minimization systems use heuristics to make dramatic reductions in the processing time.

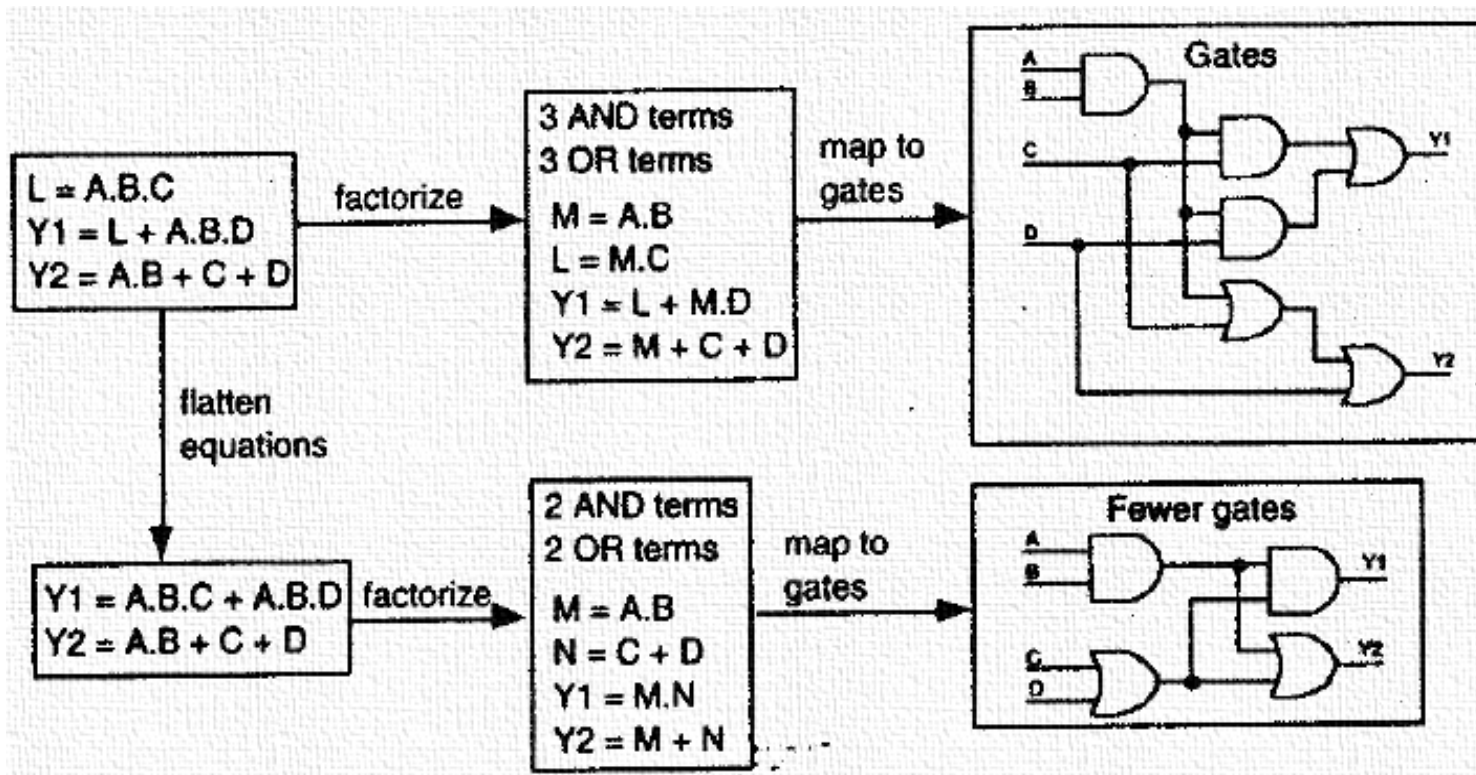
# Multi-level Logic Optimization

- Two-level logic is limited
  - Not all Boolean functions can be efficiently represented in the SOP form
- Multilevel logic implementation of a function
  - Faster and smaller than two-level logic
    - Preferred means of implementing combinational logic in VLSI systems
  - **Reuse sub-circuits,**
    - there are more degrees of freedom in implementing a Boolean function than in the two-level case.
  - However, **Largely expands the search space** in identifying an optimal solution.



# Multilevel logic optimization

- Multiple level (equation) and multiple output

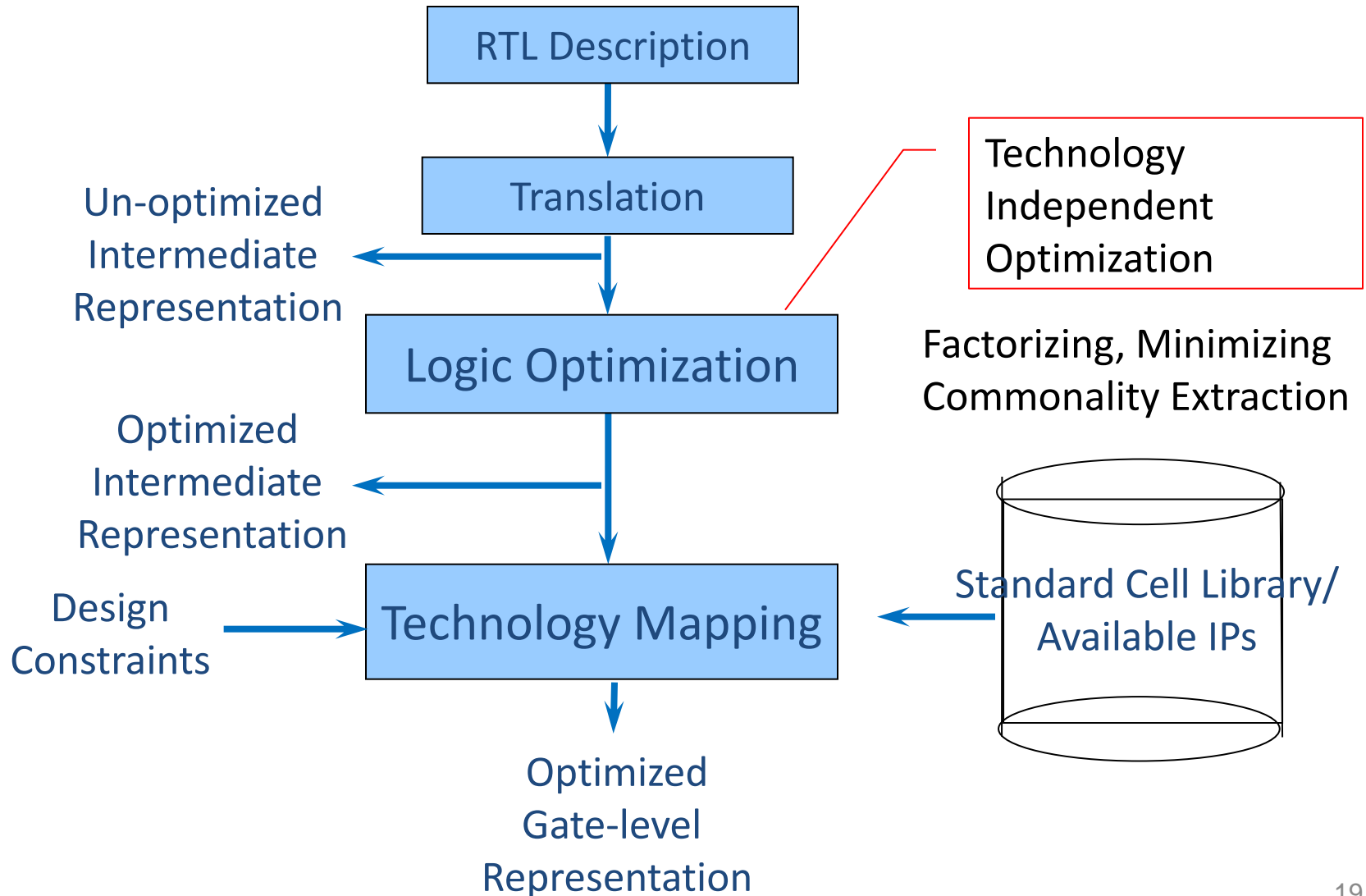


# Multilevel Optimization

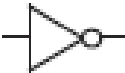



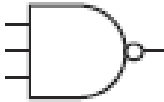
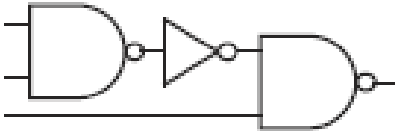
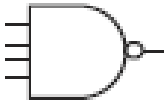
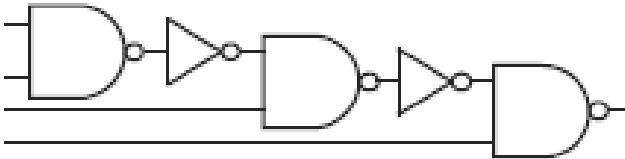
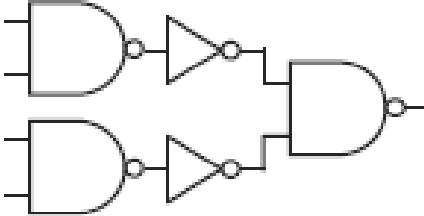
- Flattening
  - The conversion of multiple Boolean equations into a **two-level sum-of-products** form is called flattening. All **intermediate terms** are removed.
- Factoring
  - The factorization of Boolean functions is the process of **adding intermediate terms**.
    - This adds implied logic structure which reduce both the size of the implied circuit and large fan-outs.
  - Adding structure adds levels of logic which tends to make a smaller, but slower operating circuits.

Note: For more details, please refer to the reference book [2]. Chapter 6.3

# Logic Synthesis Flow from RTL to gates



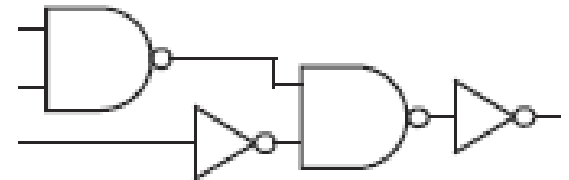
# Sample Library

Gate	Cost	Symbol	Pattern DAG
INV	2		
NAND2	3		
NAND3	4		
NAND4	5		 

# Sample Library (Cont')

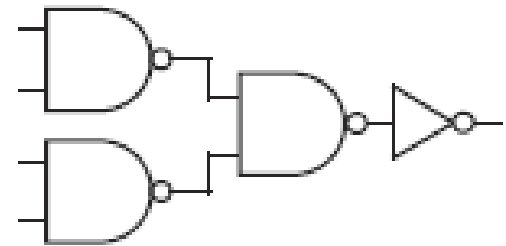
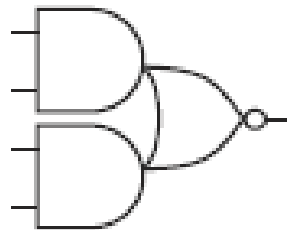
AOI21

4



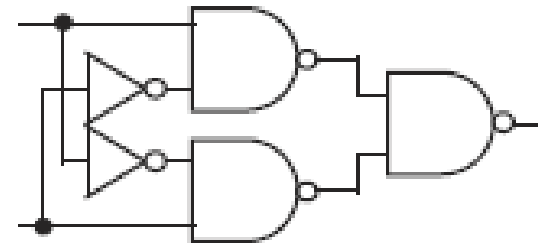
AOI22

5



XOR

4



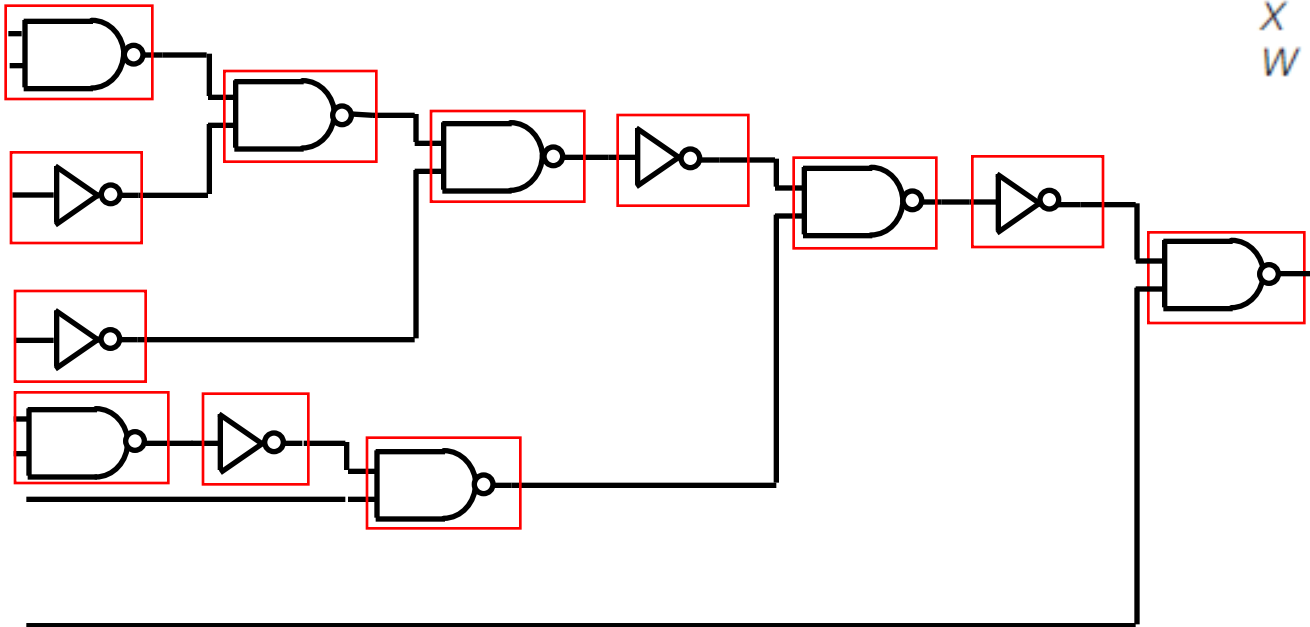
# Optimization Example: Mapping via Directed Acyclic Graph (DAG) Covering

- Represent logic network (netlist) in canonical form  
=> subject DAG
- Represent each library gate with canonical forms  
for the logic function => pattern DAGs
- Each pattern DAG has a cost
- Goal:
  - Find a **minimum cost covering** of the subject DAG by  
the pattern DAGs

# Trivial Covering

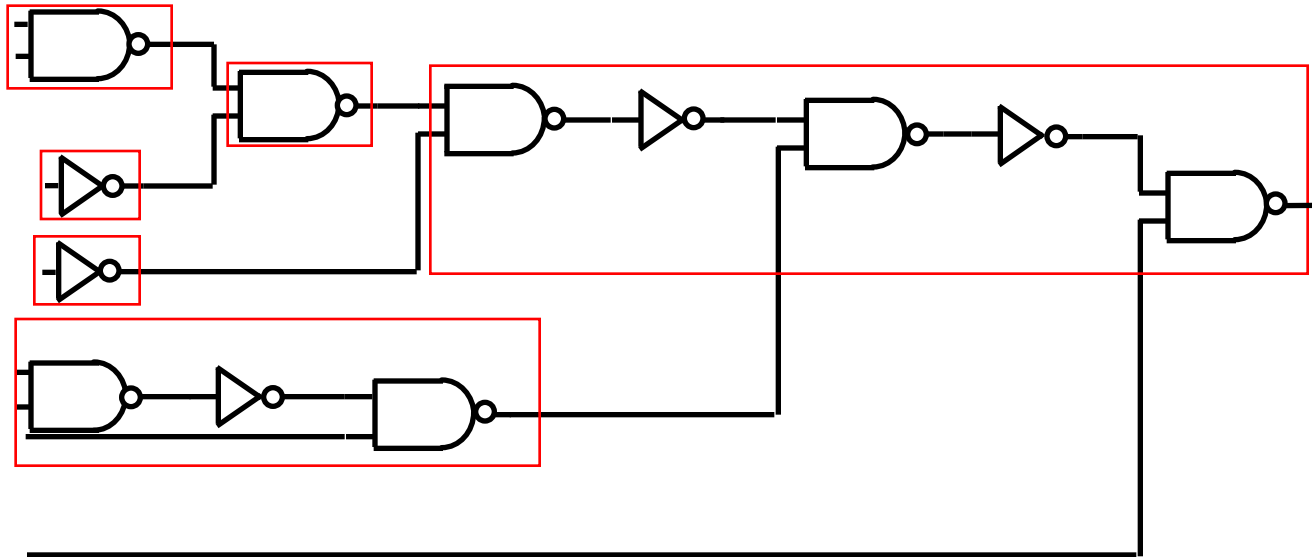
Reduce netlist into NAND2 gates  $\rightarrow$  subject DAG

$$\begin{aligned} Z &= X + \bar{Y} + \bar{h} \\ Y &= W \cdot \bar{d} \\ X &= e \cdot f \cdot g, \text{ and} \\ W &= a \cdot b + c \end{aligned}$$



7	NAND2 = 21
5	INV = 10
	<hr/>
	31 (area cost)

# Covering #1



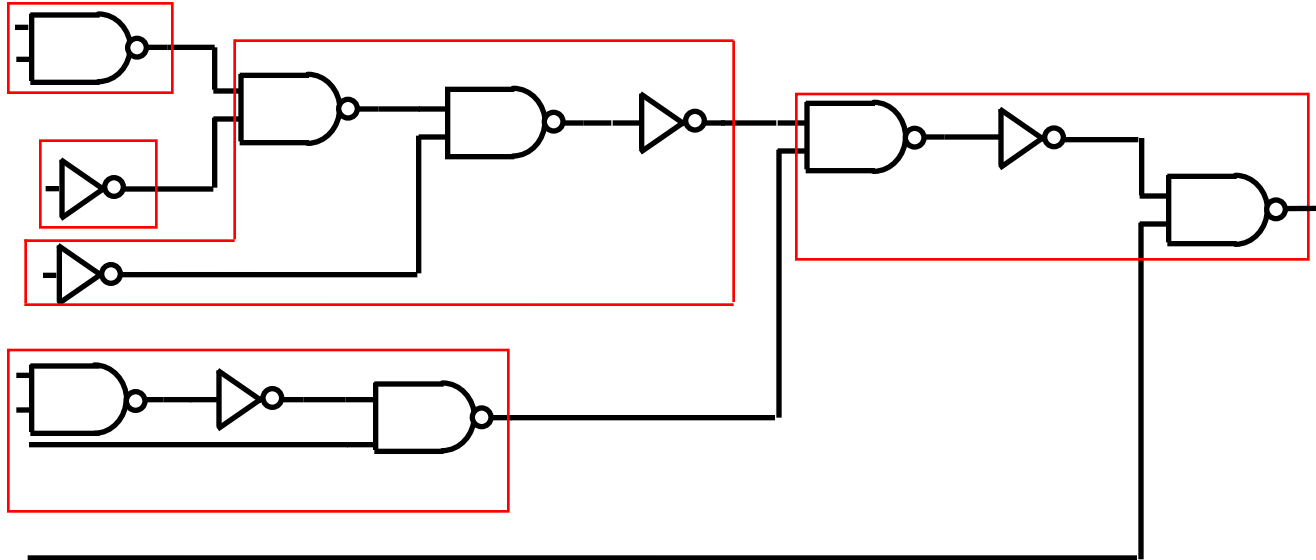
2 INV	= 4
2 NAND2	= 6
1 NAND3	= 4
1 NAND4	= 5

---

19 (area cost)



# Covering #2

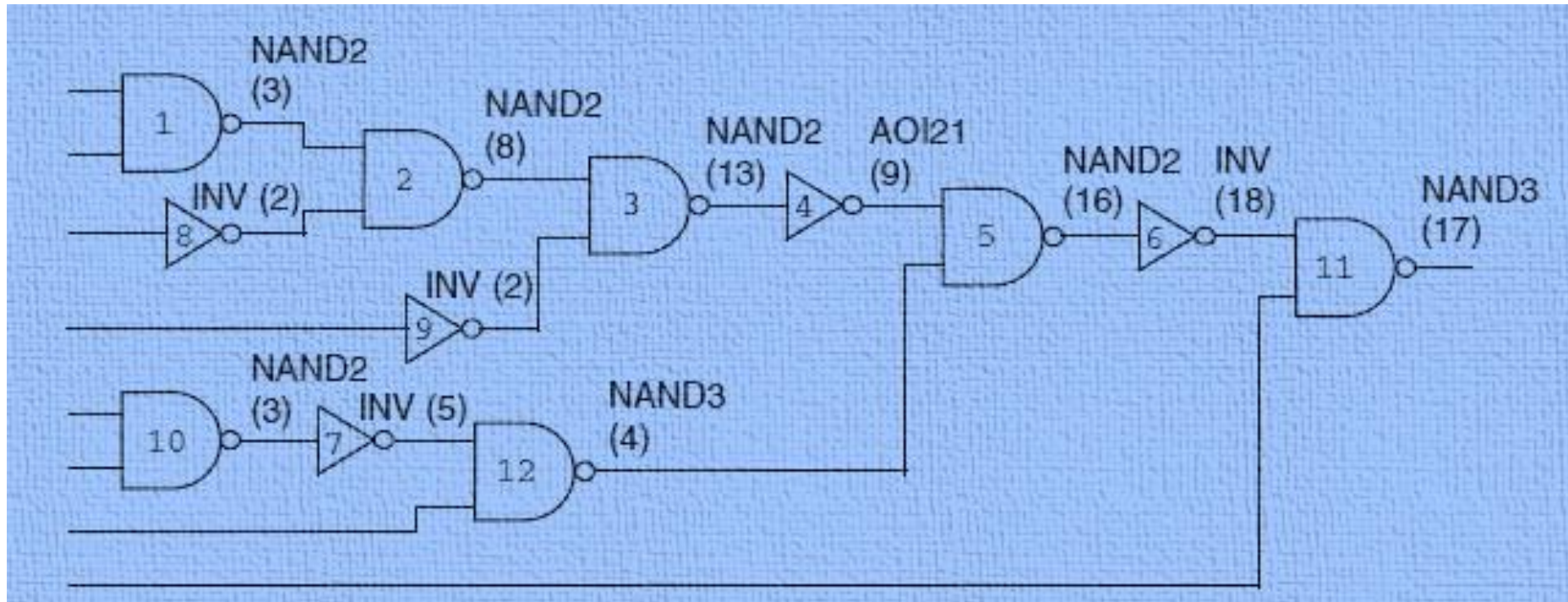


1 INV	= 2
1 NAND2	= 3
2 NAND3	= 8
1 AOI21	= 4

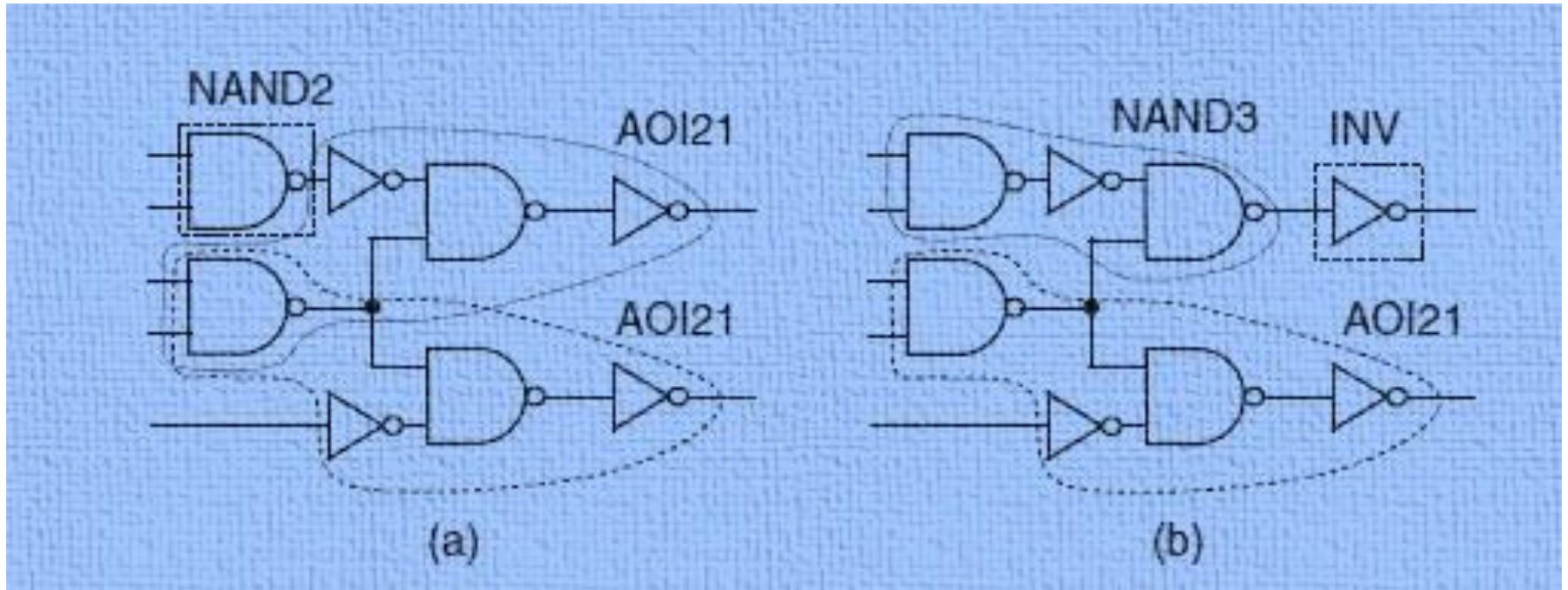
---

17 (area cost)

# Dynamic Programming for Optimum **Tree** Covering



# Examples of Cover: Non-Tree Case

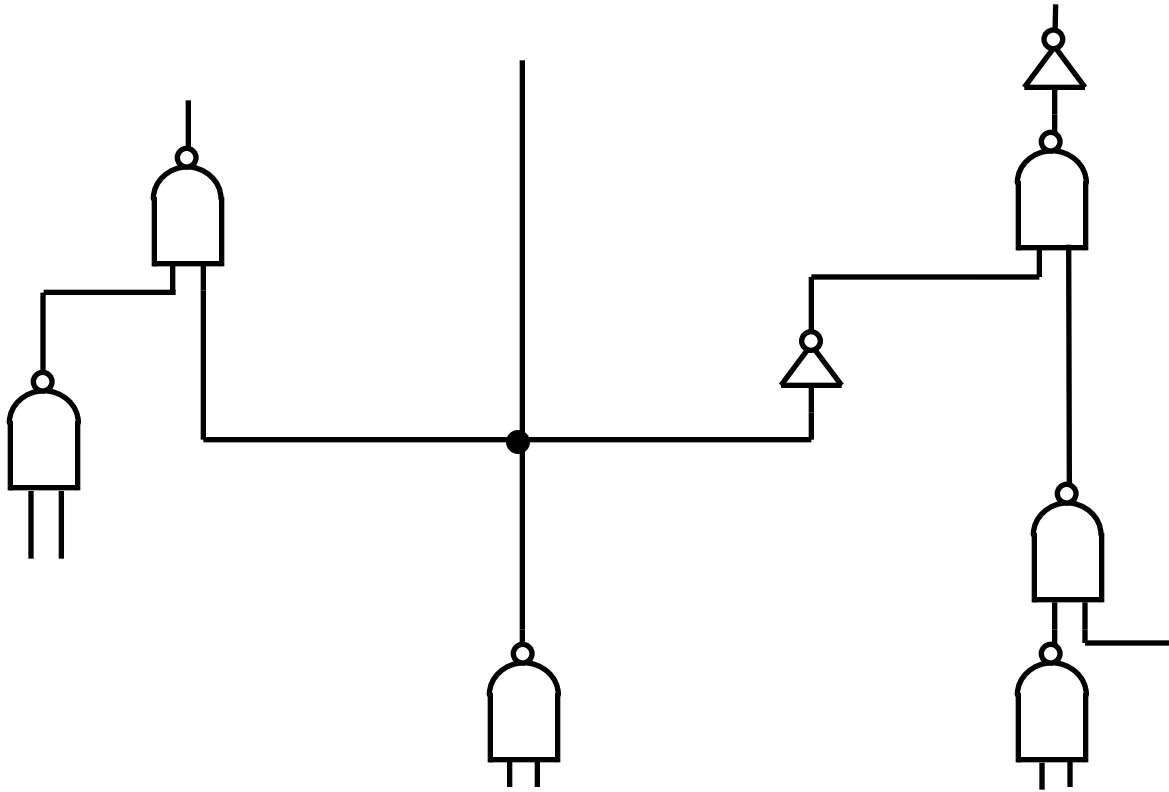


(a). Legal

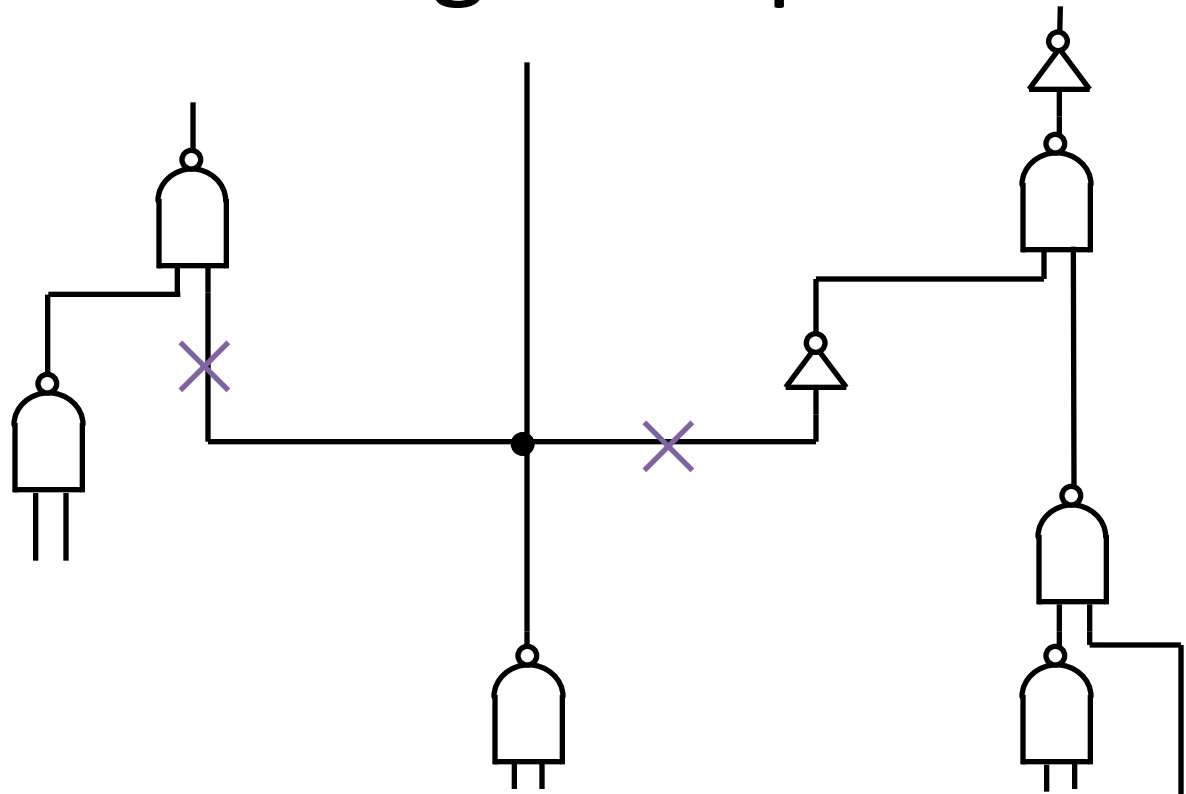
(b). Illegal

# Multiple fan-out

- NP-hard
- Heuristic: partitioning the DAG into a forest

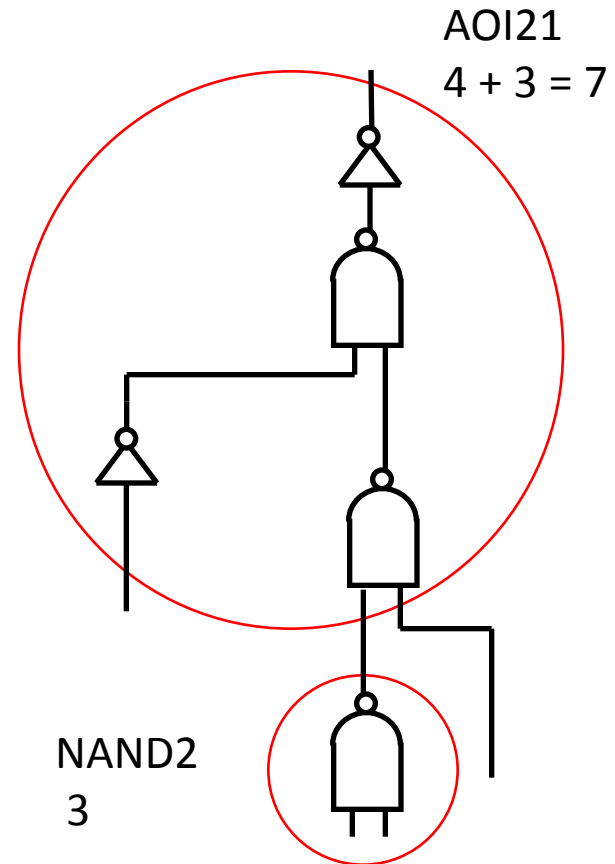
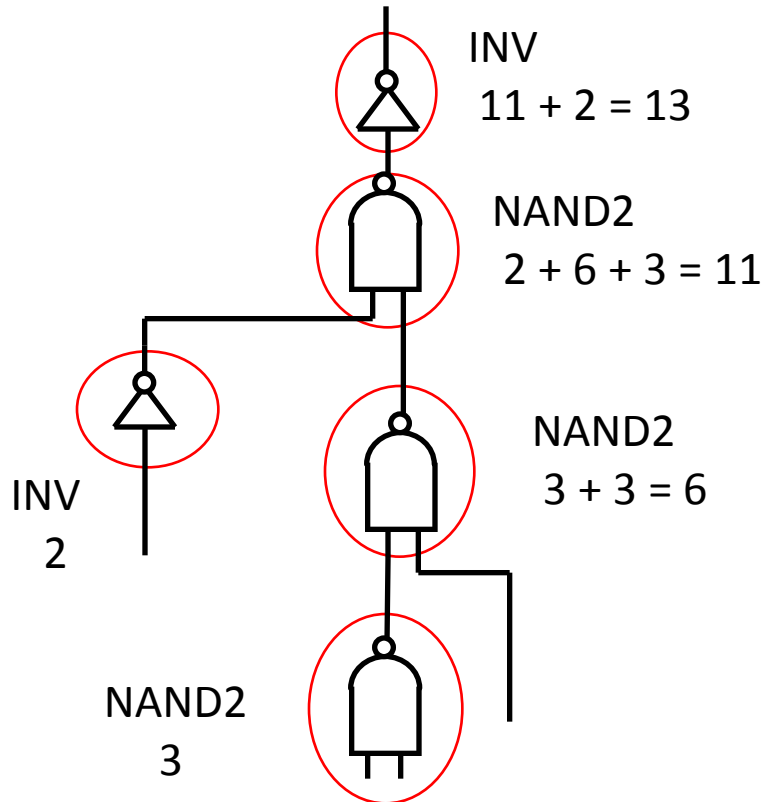


# Partitioning a Graph



1. Break at multiple fanout points
2. Partition input netlist into a forest of trees
3. Solve each tree optimally
4. Stitch trees back together

# Optimum Tree Covering

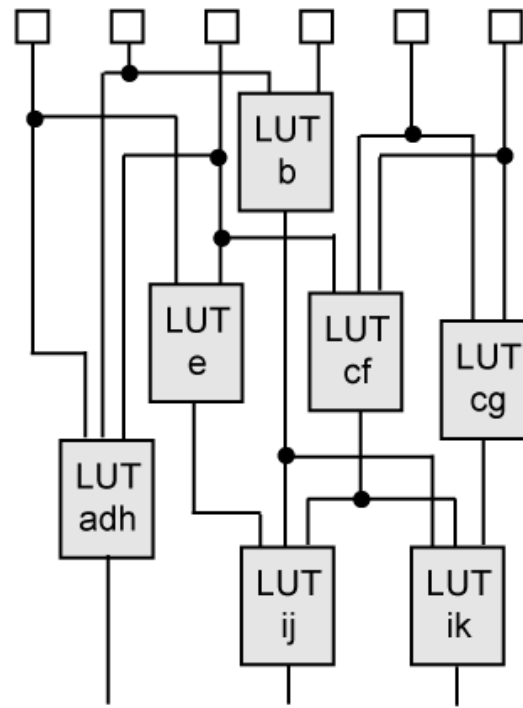
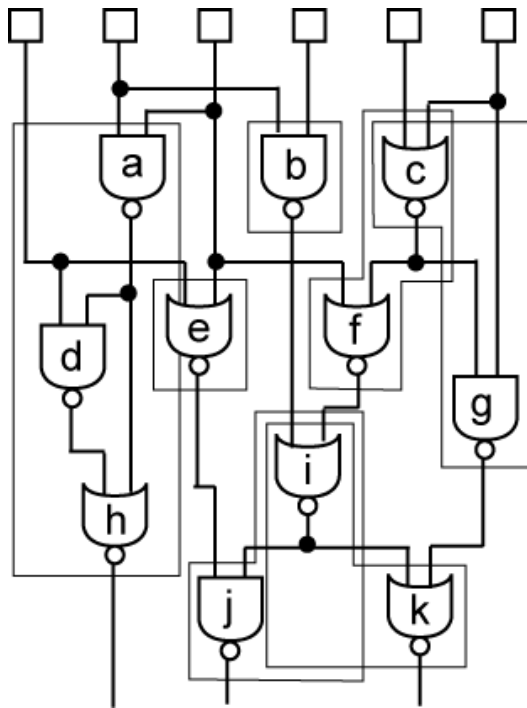


Dynamic Programming: Try all the possibilities at each gate, topological Order

Pls. refer to Reference [2] Chapter 6.4 for more details.

# Mapping to Look-up Tables

- Perform clustering on the following 2-bounded network
  - Intra-cluster and node delay = 0, inter-cluster = 1
  - Pin constraint = 3



Max delay = 2

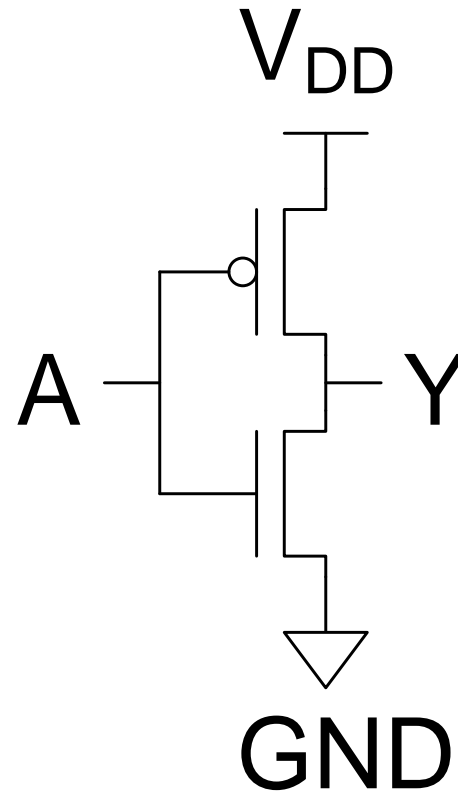
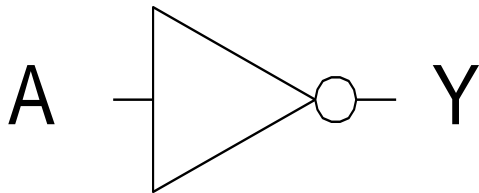
# Reference

- Logic Optimization
  - Reference 2, Chapter 6.3, 6.4



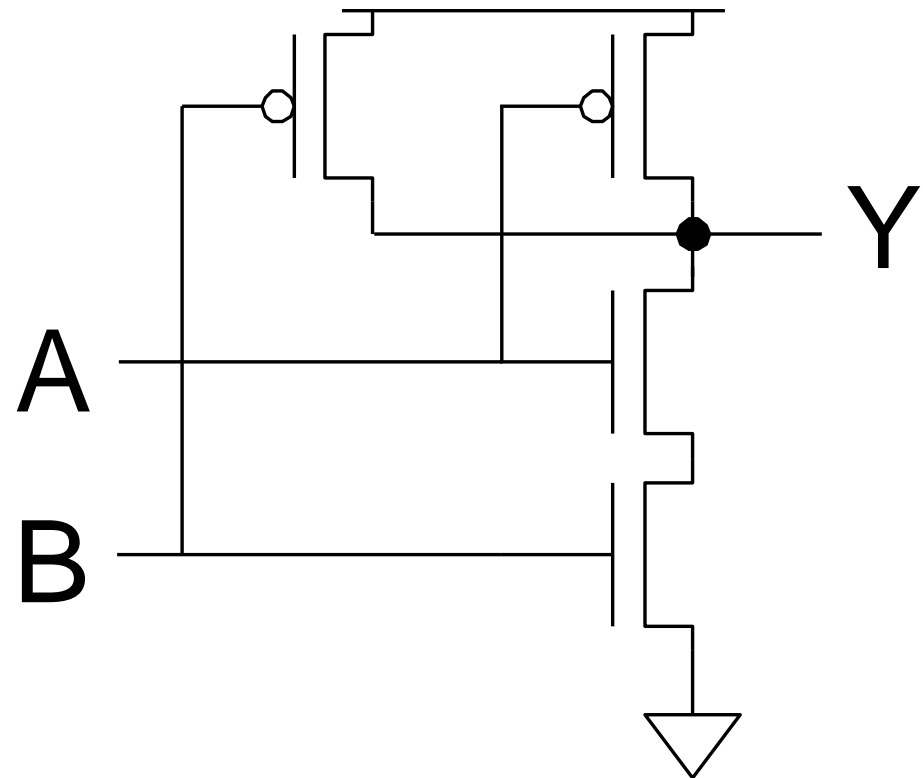
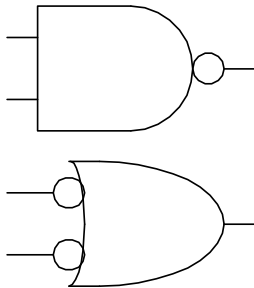
# CMOS Inverter (反向器)

A	Y



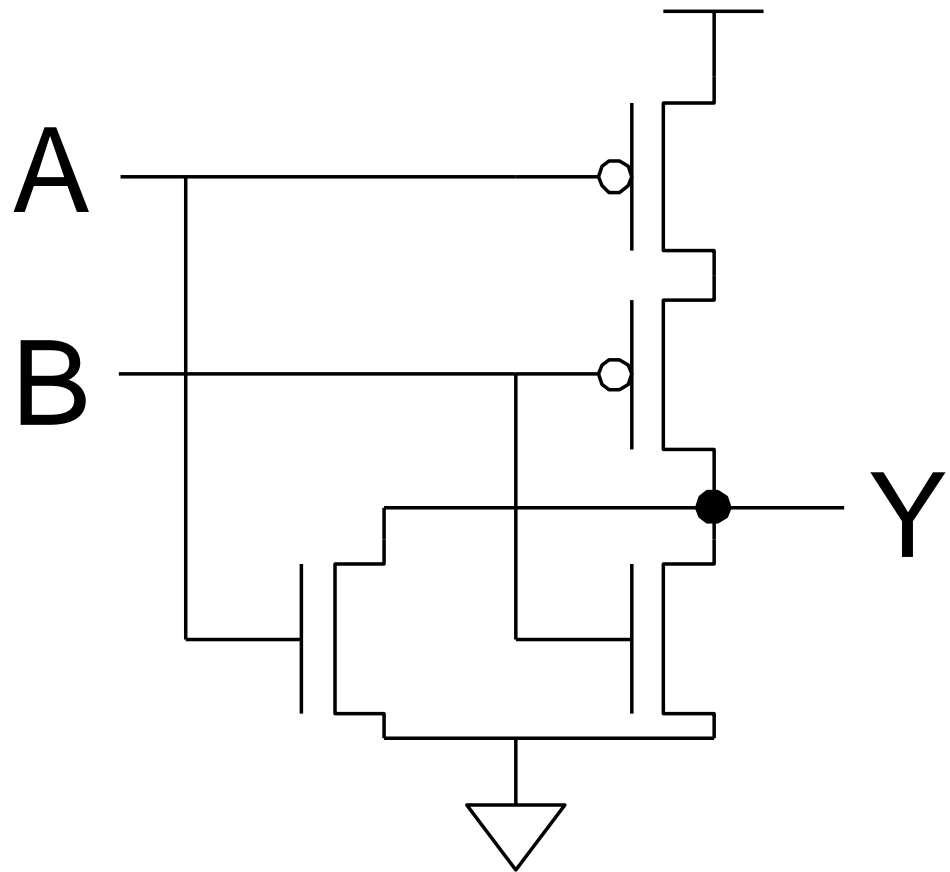
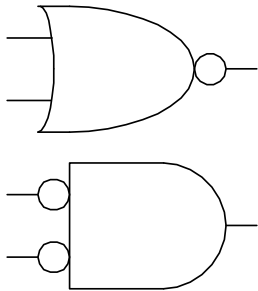
# CMOS NAND Gate

A	B	Y
0	0	
0	1	
1	0	
1	1	



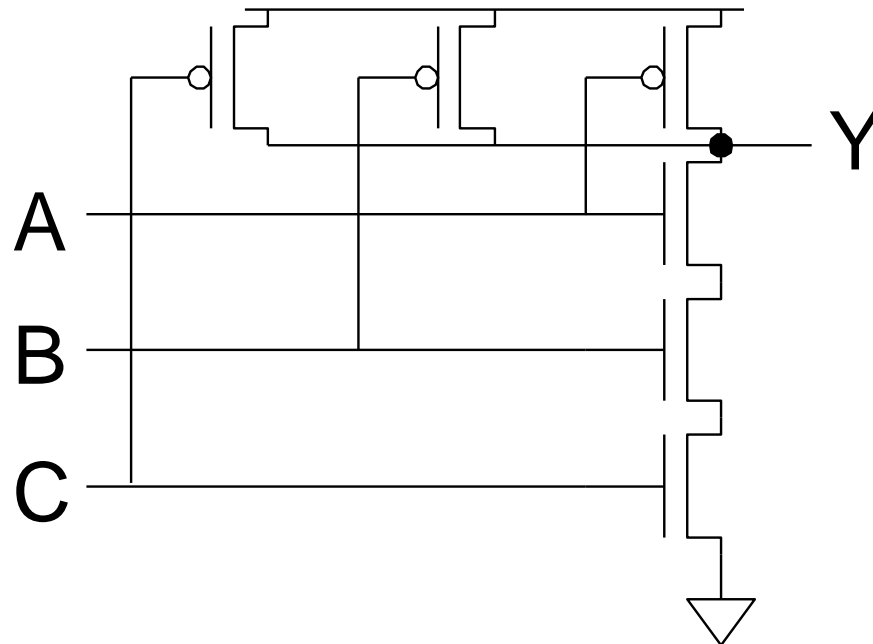
# CMOS NOR Gate

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0



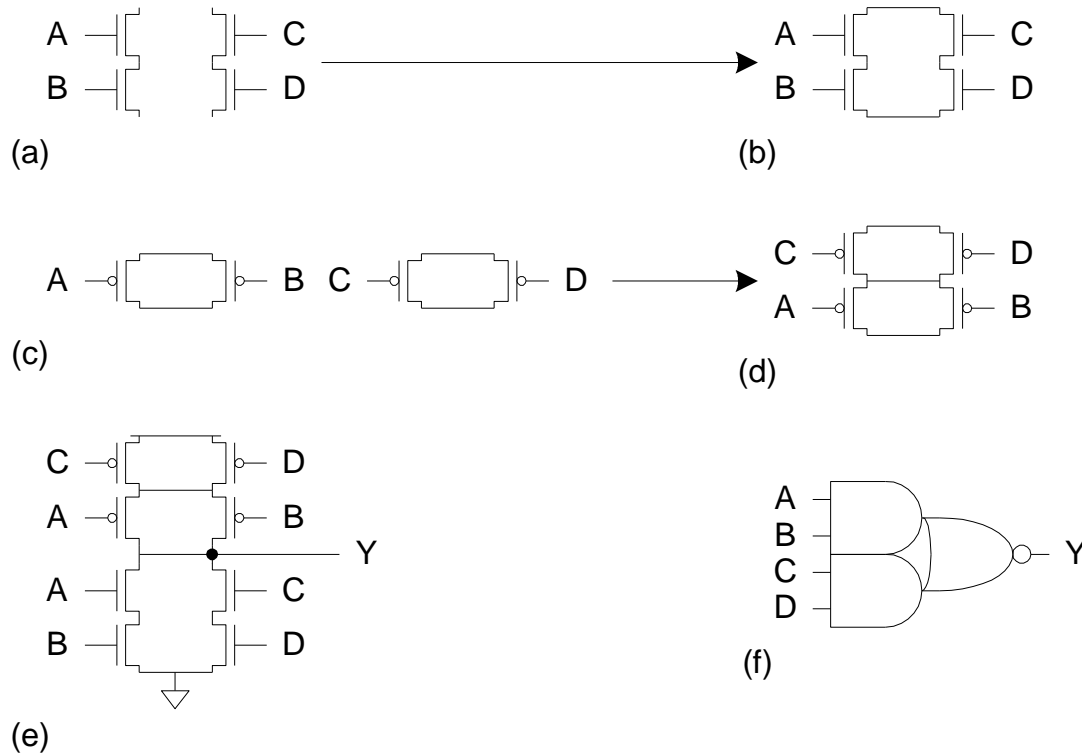
# 3-input NAND Gate

- Y pulls low if ALL inputs are 1
- Y pulls high if ANY input is 0



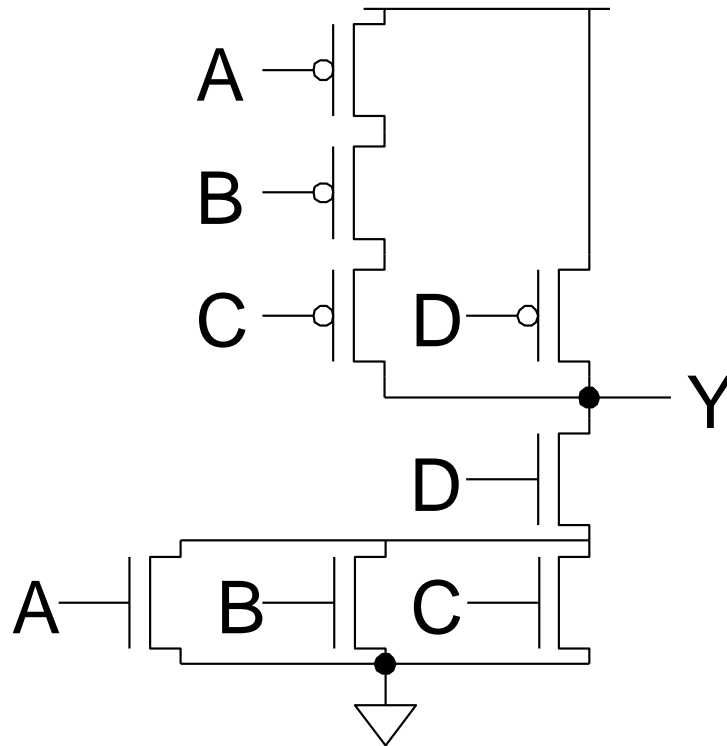
# Compound Gates

- *Compound gates* can do any inverting function
- Ex:  $Y = \overline{A \square B + C \square D}$  (AND-AND-OR-INVERT, AOI22)



# Example: O3AI

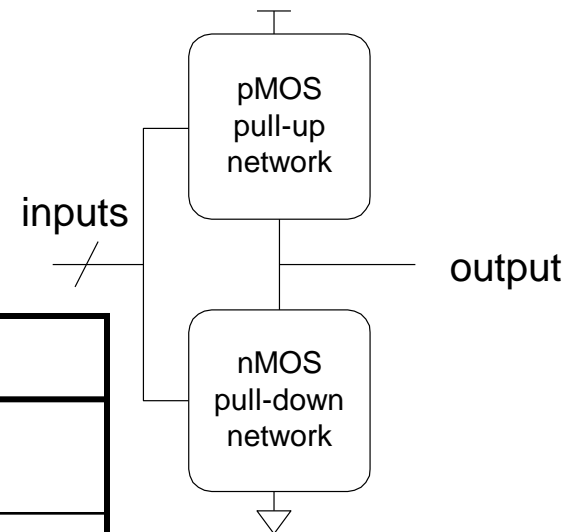
$$Y = \overline{(A + B + C)} \square D$$



# Complementary CMOS

- Complementary CMOS logic gates (cells)
  - nMOS *pull-down network*
  - pMOS *pull-up network*
  - a.k.a. static CMOS

	Pull-up OFF	Pull-up ON
Pull-down OFF	Z (float)	1
Pull-down ON	0	X (crowbar)



# Conduction Complement

- Complementary CMOS gates always produce 0 or 1
- Ex: NAND gate
  - **Series** nMOS:  $Y=0$  when both inputs are 1
  - Thus  $Y=1$  when either input is 0
  - Requires **parallel** pMOS
- Rule of *Conduction Complements*
  - Pull-up network is complement of pull-down
  - Parallel  $\rightarrow$  series, series  $\rightarrow$  parallel

