Synthesis

EDA 導論
Introduction to EDA

黃鐘揚 Chung-Yang (Ric) Huang

Feb, 2006

Switching to use Prof. Ric Huang's slides (with my updates, including Prof. Jiang's, Prof. Keutzer's and Nowick's teaching)

Thanks to all the professors contributing to this synthesis area!!

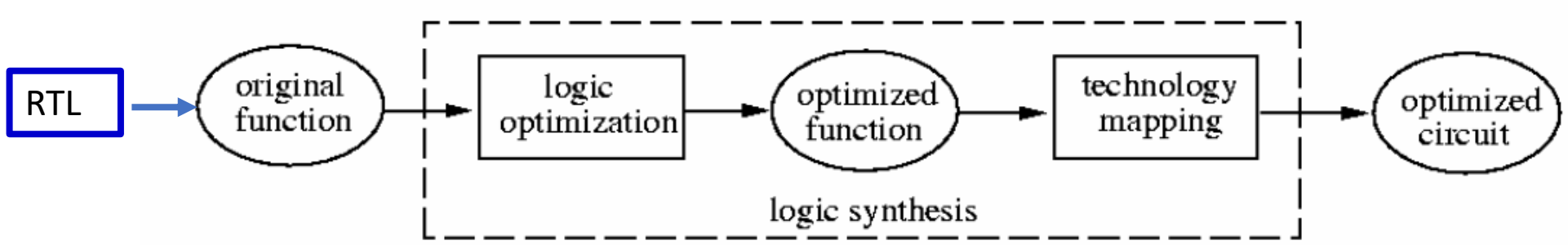
Outline

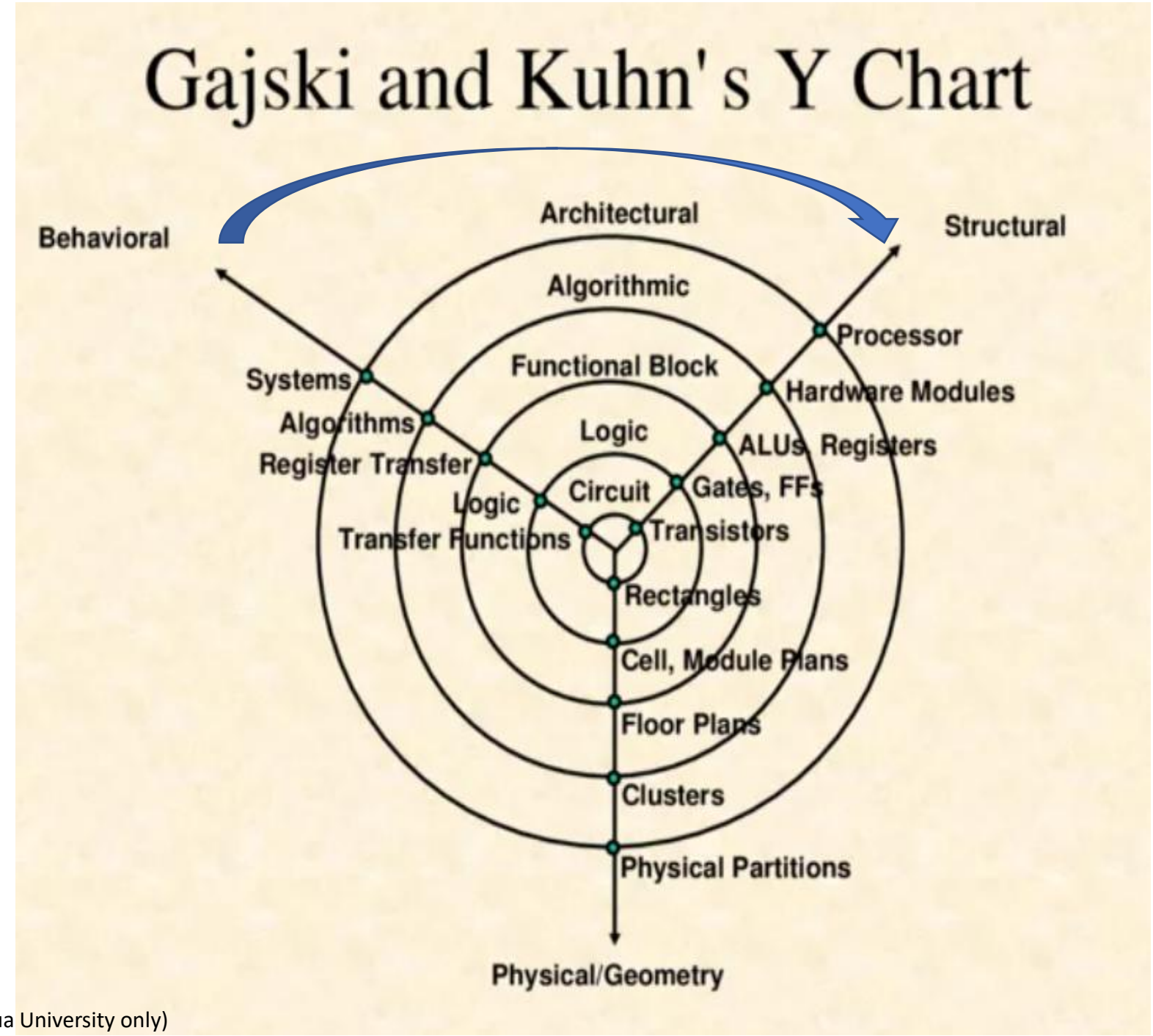
- Course contents
  - Logic synthesis basics
  - Boolean Logic
  - Logic optimization
  - Technology mapping
  - Timing & Power optimization
  - High-level synthesis basics
  - Data flow
  - Scheduling/allocation/assignment

skip

LOGIC SYNTHESIS
AND
VERIFICATION ALGORITHMS
Gary D. Hachtel
Fabio Somenzi

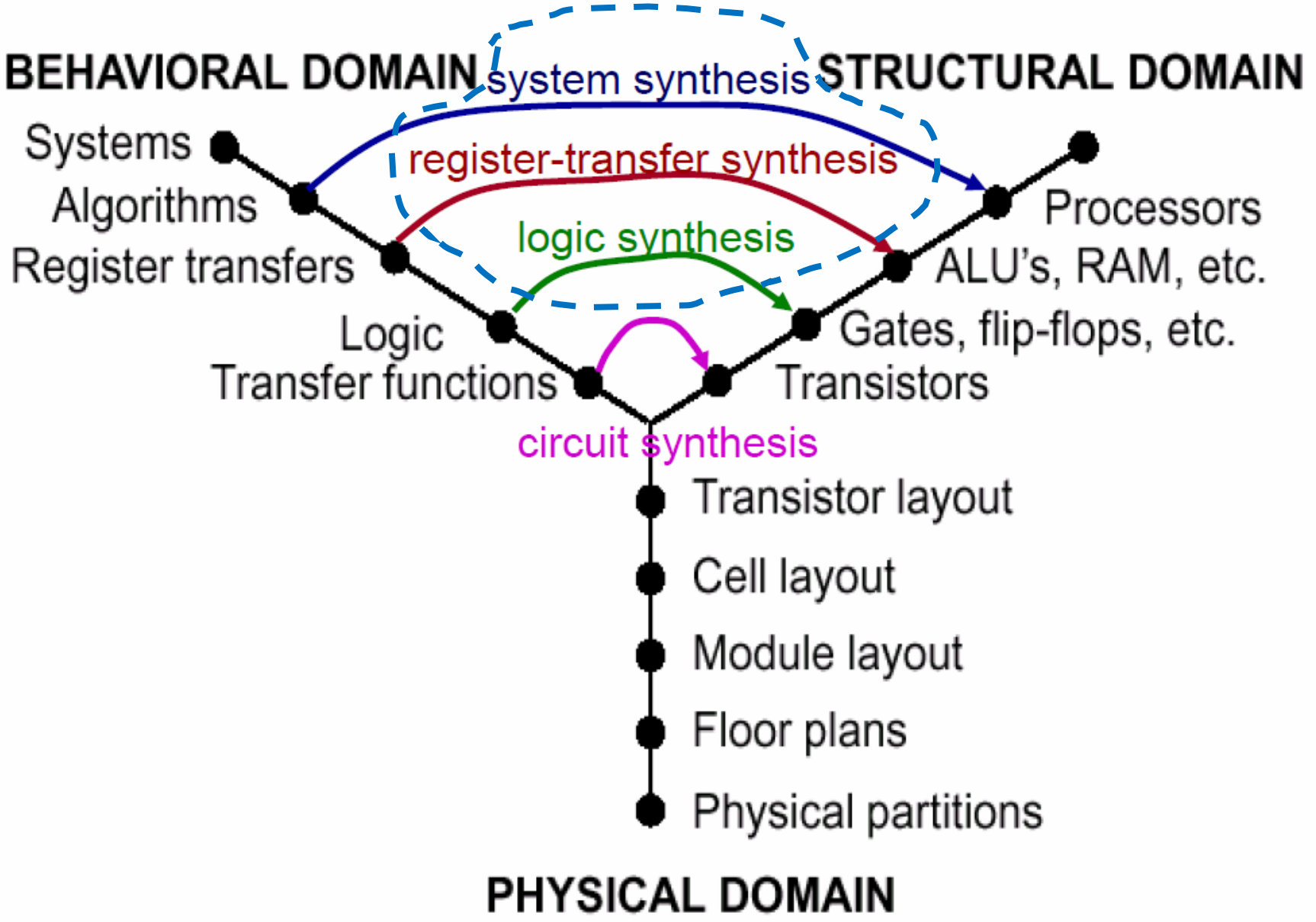
- Readings
  - Chapter 11, 12





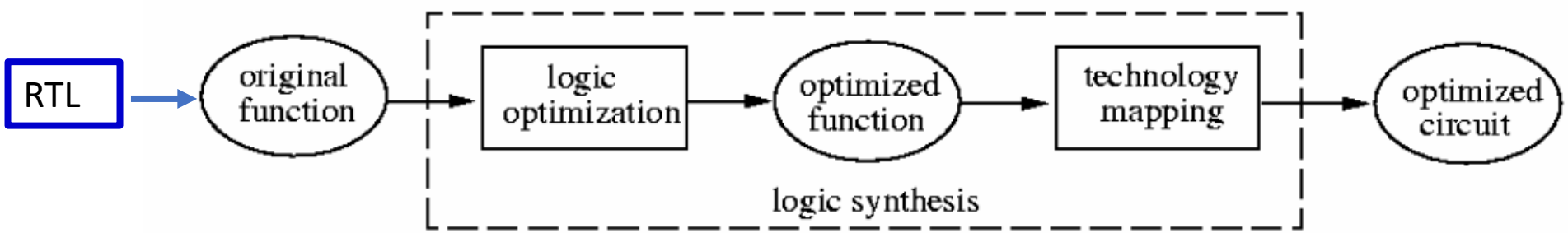
Types of synthesis – from behavioral to structural

Type of Synthesis



Logic Synthesis & Verification

- **Logic synthesis** programs transform Boolean expressions or register-transfer level (RTL) description (in Verilog/VHDL/C) into logic gate networks (netlist) in a particular library.
  - Three different tasks
    - two-level combinational synthesis
    - multilevel combinational synthesis
    - sequential synthesis
  - Optimization goals: minimize area, delay, and power, etc
- **Verification:** Checks the equivalence of a specification and an implementation.



HDL Synthesis

HDL Synthesis = Domain Translation + Optimization

Variable (A,B,C,D,Y) are assigned to registers

Domain translation

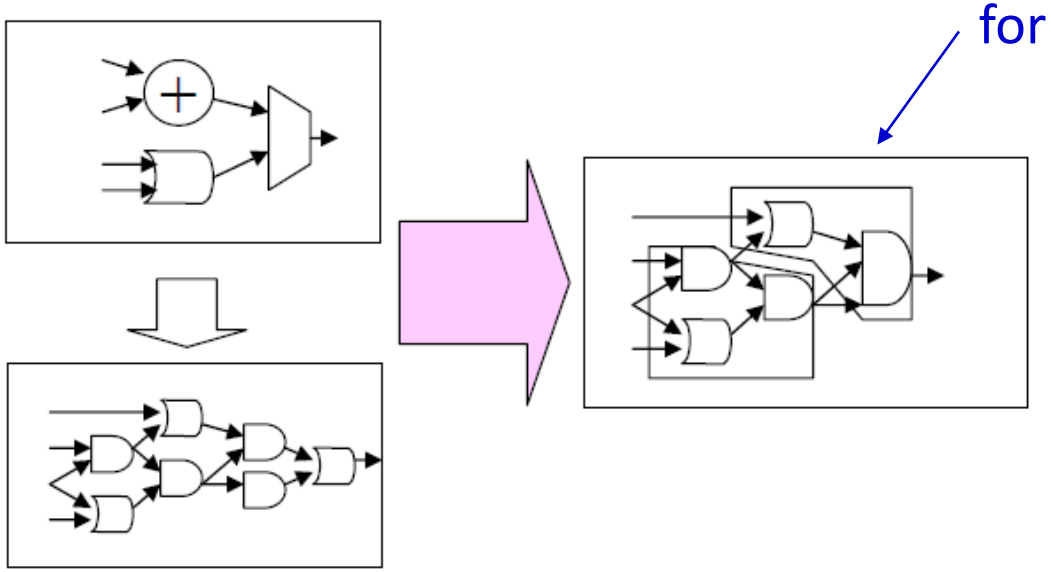
Optimization (area, timing, power...)

Logic gates can have various sizes for strengths

```
--VHDL                          //Verilog
If (A='1') then                 if (A==1)
  Y <= C + D;                     Y = C + D;
elseif  (B='1') then            else if (B==1)
  Y<= C or D;                     Y = C | D;
else Y<= C;                     else Y = C;
endif
```

Behavioral domain

RTL synthesis

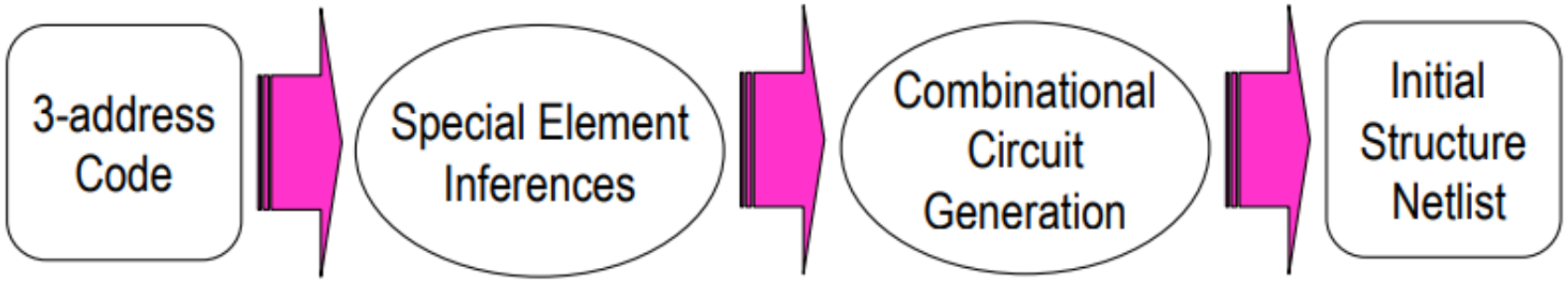


Structural domain

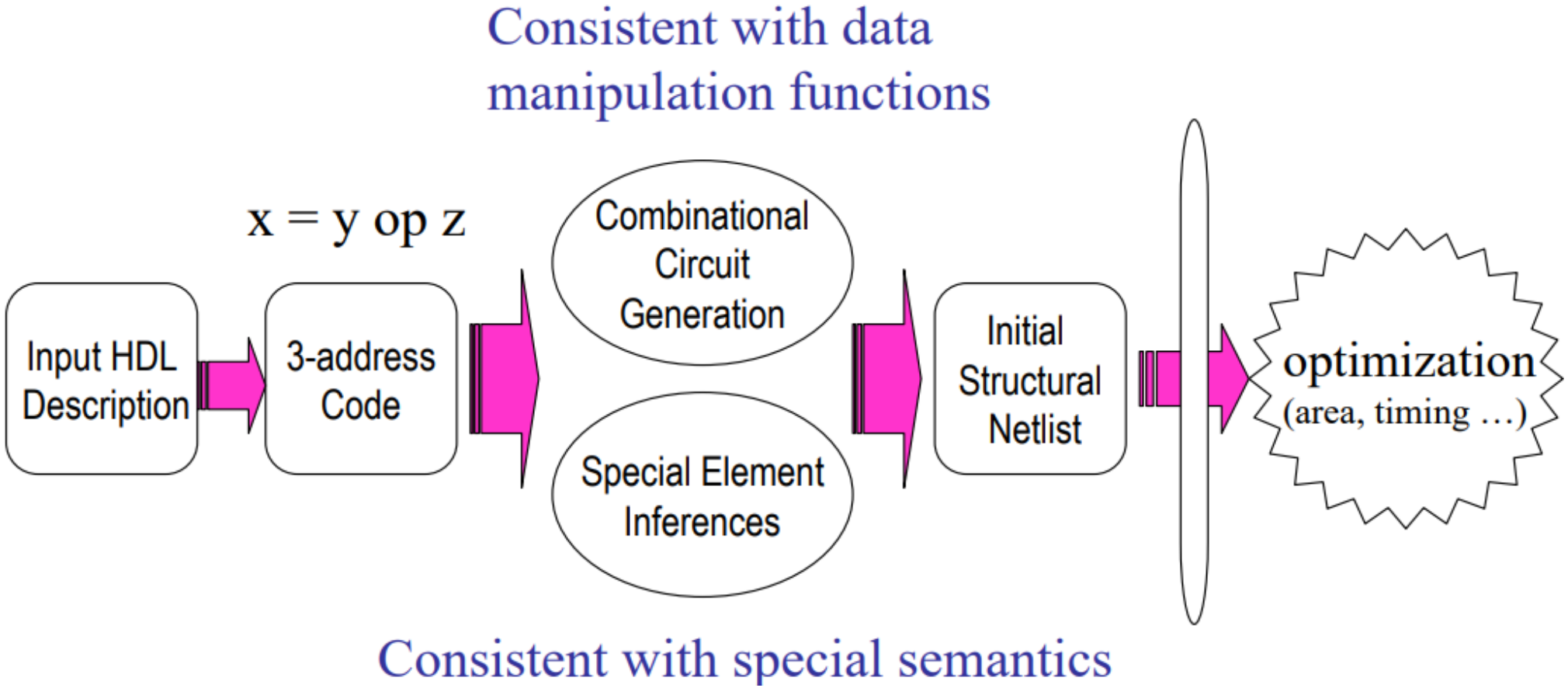
MUX is needed

Typical Domain Translation Flow

- Translate original HDL code into 3-address format
- Conduct special element inferences before combinational circuit generation
- Conduct special element inferences process by process (local view)

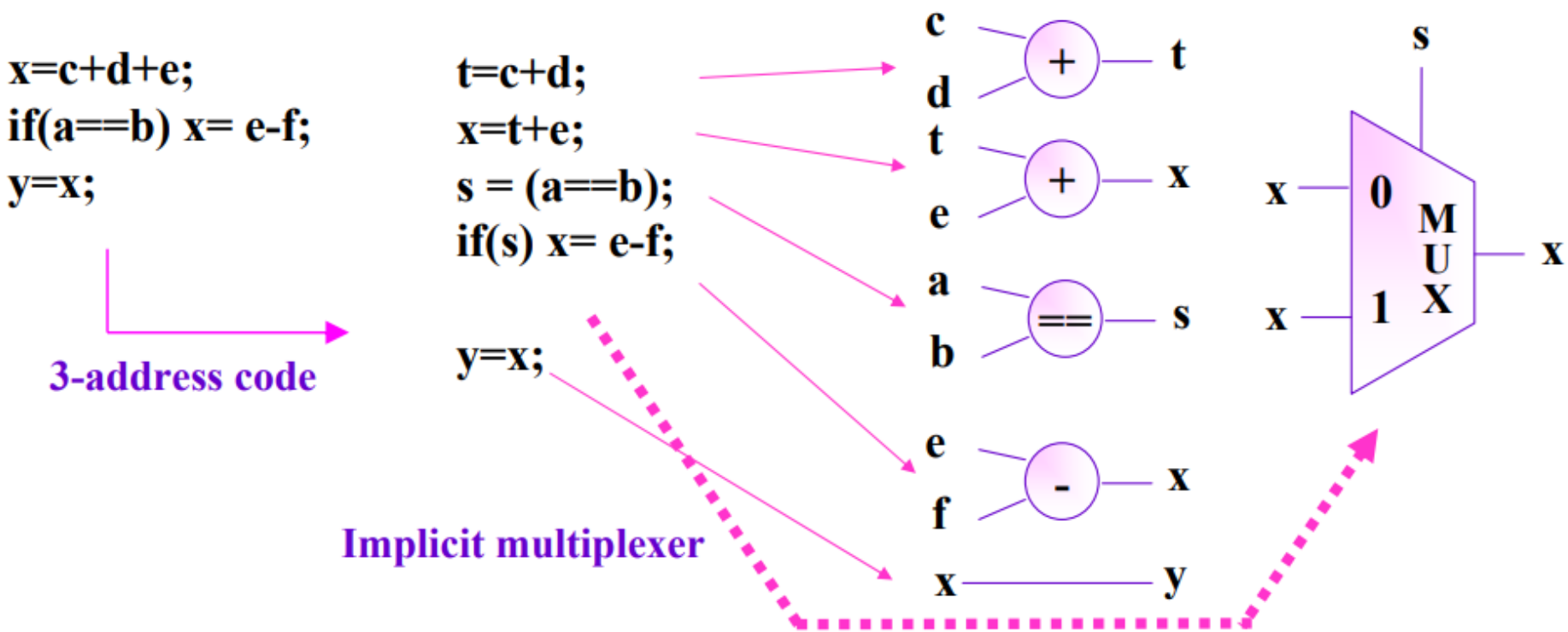


Domain Translation



Functional Unit Allocation

- 3-address code
  - x = y op z in general form
  - Function unit op with inputs y and z and output x



Combinational Circuit Generation

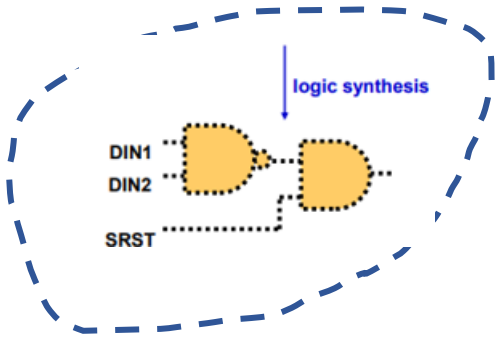
- Functional unit allocation
  - Straightforward mapping with 3-address code
- Interconnection binding
  - Using control/data flow analysis

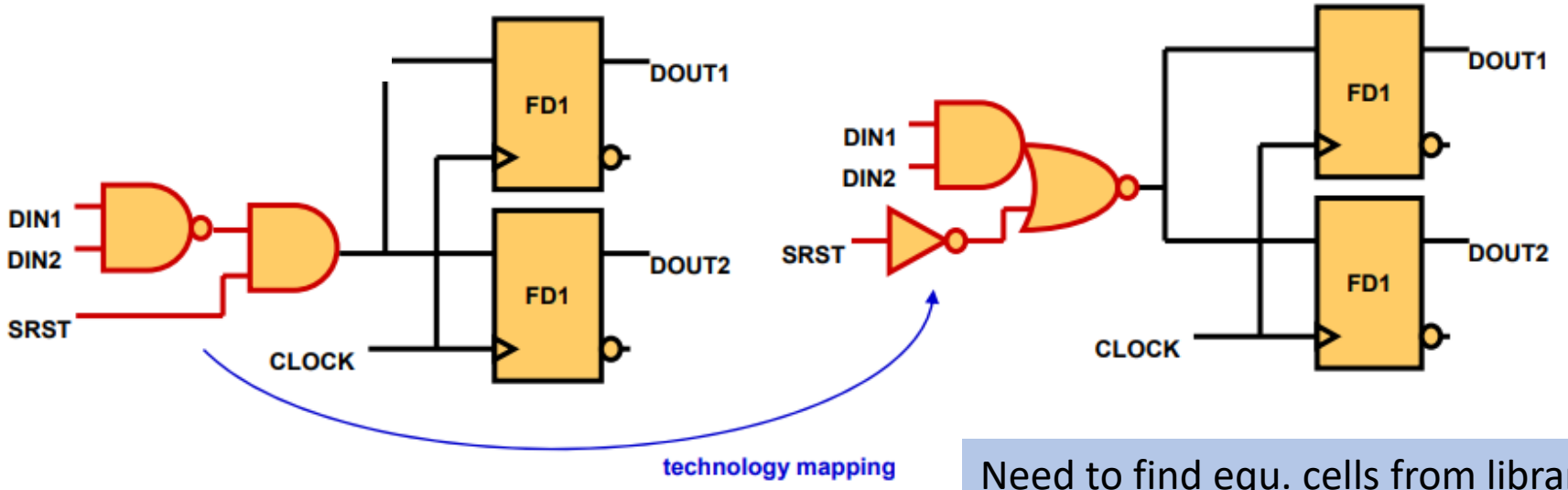
Interconnection Binding

- Need the dependency information among functional units
  - Using **control/data flow analysis**
  - A traditional technique used in compiler design for a variety of code optimizations
  - Statically analyze and compute the set of assignments reaching a particular point in a program

Technology Mapping Problem

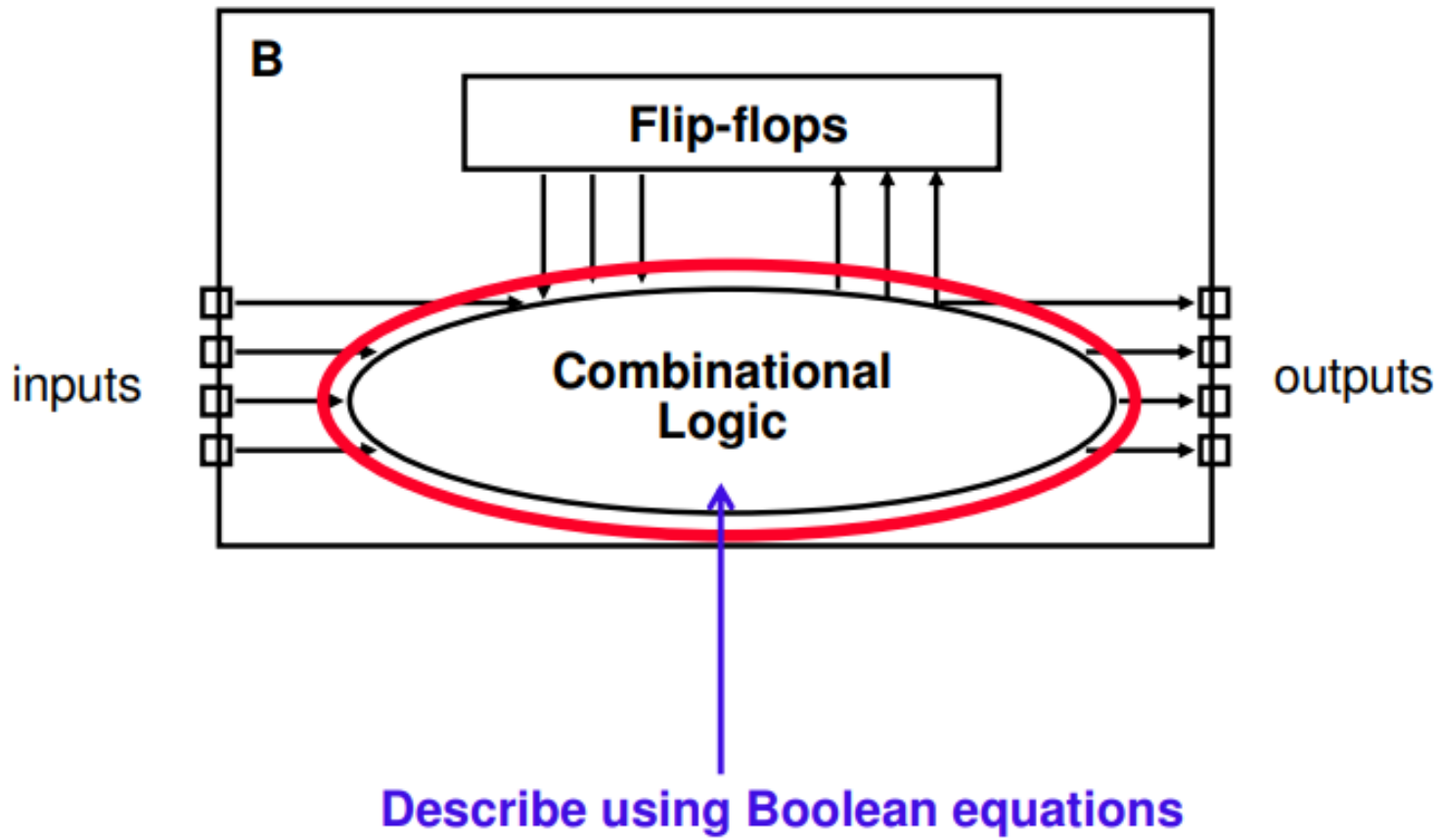
```
process
begin
    wait until rising_edge(CLOCK);
    if (SRST='0') then
        DOUT1 <= '0';
    else
        DOUT1 <= DIN1 nand DIN2;
    end if;
    DOUT2 <= SRST and (DIN1 nand DIN2);
end process;
```





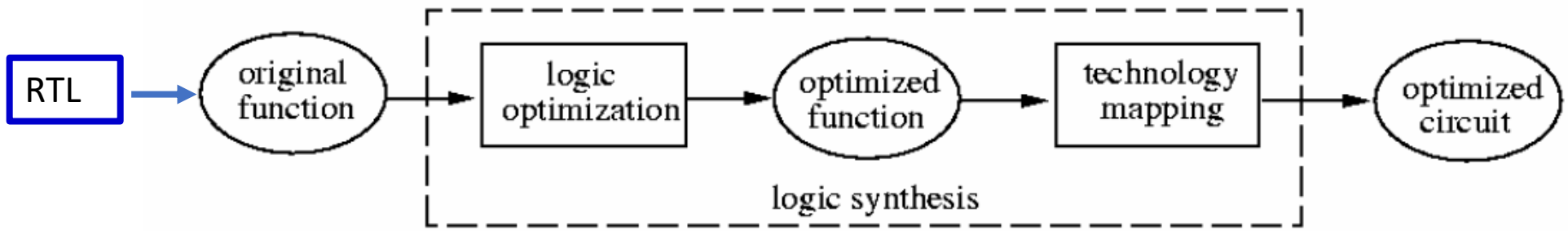
Need to find equ. cells from library

Synthesize Logic to Implement equations



Logic Optimization

- **Technology-independent** optimization
  - Works on Boolean expression equivalent.
  - Estimates size based on # of literals.
  - Uses don't-cares, common factor extraction (factorization), etc. to optimize logic.
  - Uses simple delay models.
- **Technology-dependent** optimization: **technology mapping/library binding**
  - Maps Boolean expressions into a particular cell library.
  - May perform some optimizations in addition to simple mapping.
  - Uses more accurate delay models based on cell structures.



Adapted from: Prof. Y.-W. Chang's slides

Two-Level Logic Optimization

- Two-level logic representations
  - Sum-of-product form
  - Product-of-sum form
- Two-level logic optimization
  - Key technique in logic optimization
  - Many efficient algorithms to find a near minimal representation in a practical amount of time
  - In commercial use for several years
  - Minimization criteria: **number of product terms**
- Example: $F = XYZ + X\bar{Y}\bar{Z} + X\bar{Y}Z + \bar{X}YZ + XY\bar{Y}Z$

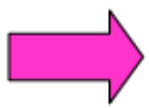
$F = X\bar{Y} + YZ$

Multi-Level Logic Optimization

- Translate a combinational circuit to meet performance or area constraints
  - Two-level minimization
  - Common factors or kernel extraction
  - Common expression resubsitution
- In commercial use for several years
- Example:

$$f1 = abcd + abce + \bar{a}bcd + a\bar{b}\bar{c}\bar{d} + \bar{a}c + cdf + ab\bar{c}\bar{d}e + a\bar{b}cdf$$

$$f2 = bdg + \bar{b}dfg + \bar{b}\bar{d}g + b\bar{d}eg$$



$$f1 = c(\bar{a} + x) + a\bar{c}\bar{x}$$

$$f2 = gx$$

$$x = d(b + f) + \bar{d}(\bar{b} + e)$$

Boolean Functions

- $B = \{0,1\}$, $Y = \{0,1,D\}$
- A Boolean function f: $B^m \rightarrow Y^n$
  - $f = \overline{x}_1 \overline{x}_2 + \overline{x}_1 \overline{x}_3 + \overline{x}_2 x_3 + x_1 x_2 + x_2 \overline{x}_3 + x_1 x_3$ **Sum of products**
- Input variables: x_1, x_2, …
- The value of the output partitions B^m into three sets
  - the on-set
  - the off-set
  - the dc-set (don't-care set)

Boolean algebra such as:

$a + (b * c) = (a + b) * (a + c)$

$$\overline{a \cdot b} = \overline{a} + \overline{b} \qquad \overline{a + b} = \overline{a} \cdot \overline{b}$$

$$(a + b) \cdot c = a * c + b * c \qquad a \cdot b + c = (a + c) \cdot (b + c)$$

$$a \cdot a = a \qquad a + a = a$$

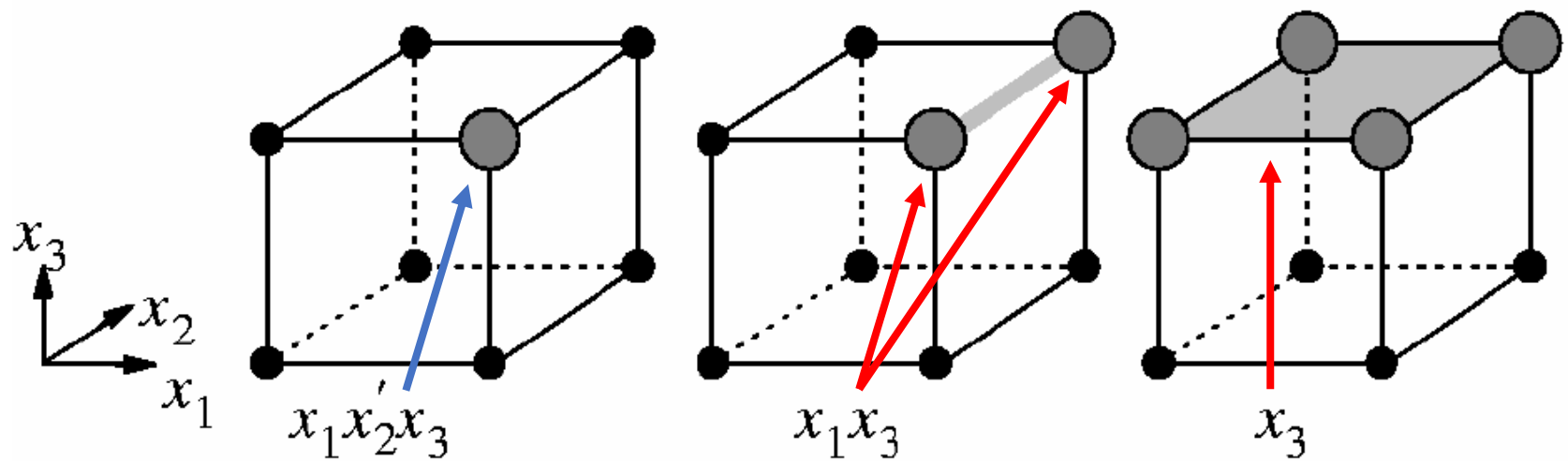
$$a \cdot \overline{a} = \text{'0'} \qquad a + \overline{a} = \text{'1'} \qquad (11.4)$$

$$a \cdot \text{'0'} = \text{'0'} \qquad a + \text{'0'} = a$$

$$a \cdot \text{'1'} = a \qquad a + \text{'1'} = \text{'1'}$$

Minterms and Cubes

- A minterm is a product of **all** input variables or their negations.
  - A minterm corresponds to a single point in B^m.
- A cube is a product of the input variables or their negations.
  - The fewer the number of variables in the product, the bigger the space covered by the cube.



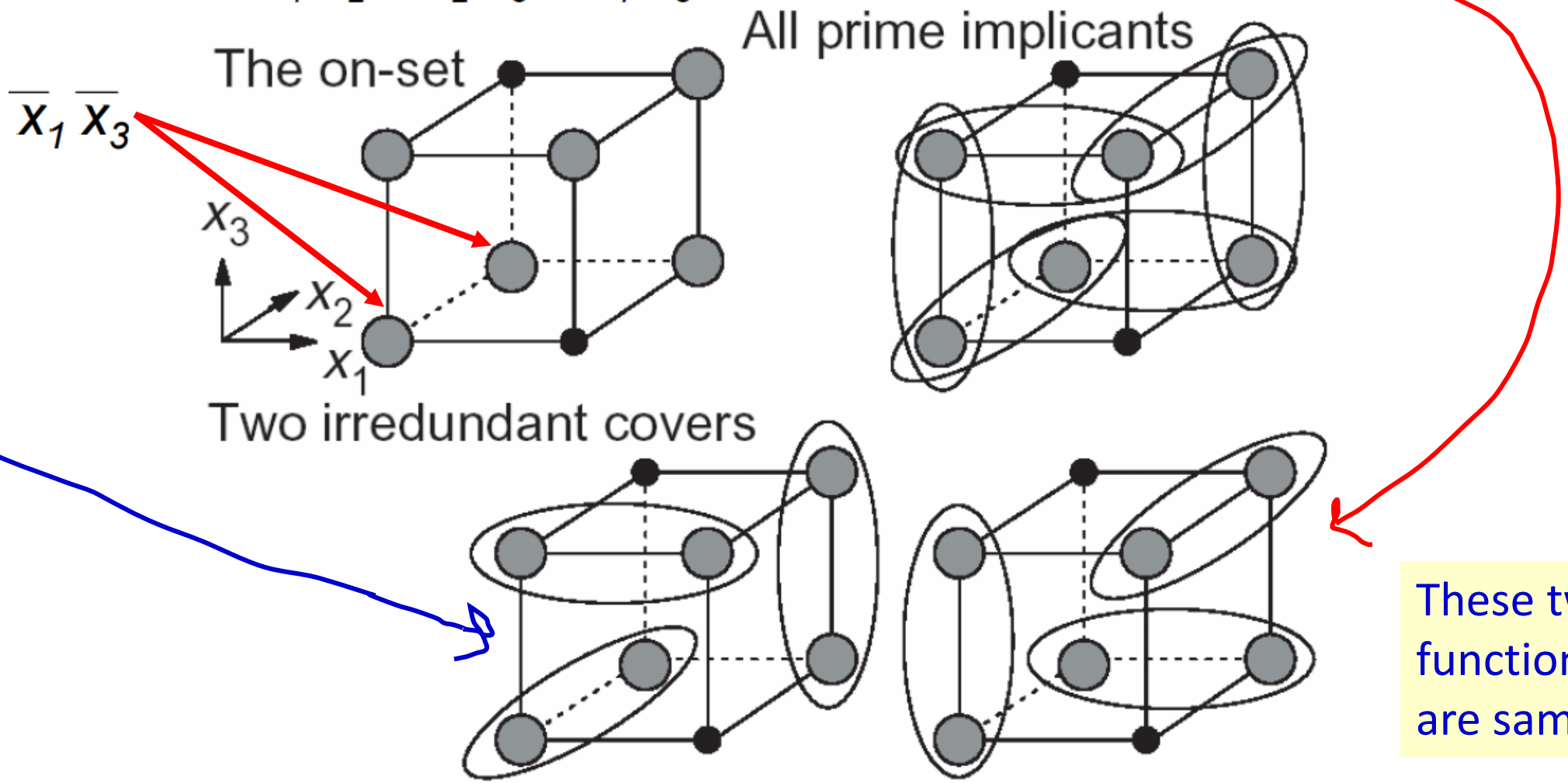
Product of 1 0 1

Implicant and Cover

- An **implicant** is a cube whose points are either in the on-set or the dc-set.
- A **prime implicant** is an implicant that is not included in any other implicant.
- A set of prime implicants that together cover all points in the on-set (and some or all points of the dc-set) is called a prime cover.
- A prime cover is irredundant when none of its prime implicants can be removed from the cover.
- An irredundant prime cover is minimal when the cover has the minimal number of prime implicants.

Cover Examples

- $f = \overline{x}_1\,\overline{x}_3 + \overline{x}_2\,x_3 + x_1\,x_2$
- $f = \overline{x}_1\,\overline{x}_2 + x_2\,\overline{x}_3 + x_1\,x_3$



These two functions are same

Canonical Forms

- A **canonical form** of a Boolean function is a unique representation of the function.
  - It can be used for verification purposes.
- The **truth table** or the **sum of minterms** are canonical forms
  - They grow exponentially with the number of input variables.
- An irredundant prime cover is not a canonical form.
- **Reduced ordered binary decision diagram (ROBDD)**: a canonical form that is interesting from a practical point of view.

Binary Decision Diagram (BDD)

- BDD was proposed by R.E. Bryant in 1986
  - "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE TC*, Vol. C-35, Aug. 1986.
- BDD is a Directed Acyclic Graph (DAG) used to represent a Boolean function f: $B^m \rightarrow B^n$
- Each non-terminal node is a decision node associated with an input variable with two branches: 0-branch and 1-branch
- There are two terminal nodes: 0-terminal and 1-terminal
- Example:

Binary-Decision Diagram (BDD) Principles

Restriction means the substitution of a constant value for one of the variables

- **Restriction** resulting in the **positive** and **negative cofactors** of a Boolean function:

$$f_{x_i} = f(x_1, \ldots, x_{i-1}, '1', x_{i+1}, \ldots, x_m)$$
$$f_{\overline{x_i}} = f(x_1, \ldots, x_{i-1}, '0', x_{i+1}, \ldots, x_m)$$

  - $f = \overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}\,x_2\,\overline{x_3} + \overline{x_1}\,\overline{x_2}\,x_3 + x_1\,\overline{x_2}\,x_3 + x_1\,x_2\,\overline{x_3} + x_1\,x_2\,x_3$

    $f_{x1} = \overline{x_2}\,x_3 + x_2\,\overline{x_3} + x_2\,x_3$

    $f_{\overline{x1}} = \overline{x_2}\,\overline{x_3} + \overline{x_2}\,x_3 + x_2\,\overline{x_3}$

- **Shannon expansion** (already known to Boole) states:

$$f = x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}}$$

- A complete expansion can be obtained by successively applying Shannon expansion on all variables of a function until either of the constant functions ’0’ or ’1’ are reached.

Shannon Expansion

- $F(x_1, \ldots, x_n) = x_i \cdot F_{x_i} + x_i' \cdot F_{x_i'}$

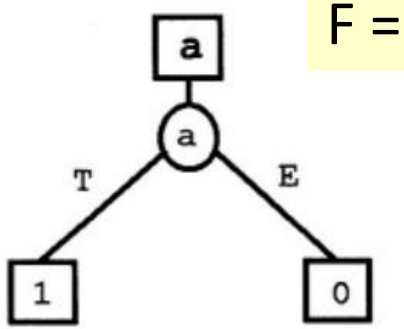
Shannon expansion with many variables

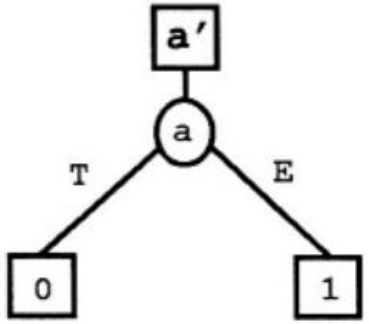
- $F(x, y, z, w) =$

  $xy\, F_{xy} + x'y\, F_{x'y} + xy'\, F_{xy'} + x'y'\, F_{x'y'}$

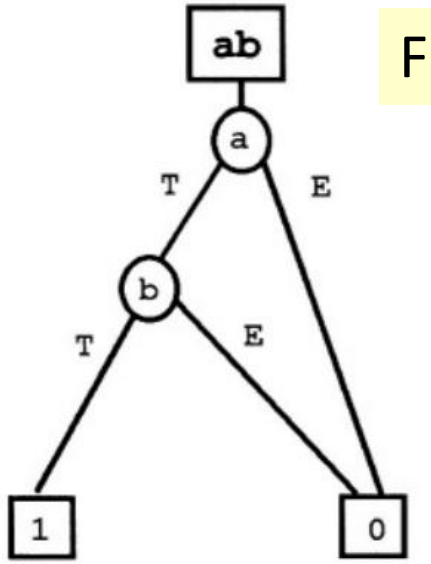
Figure 6.5: BDDs for typical functions.

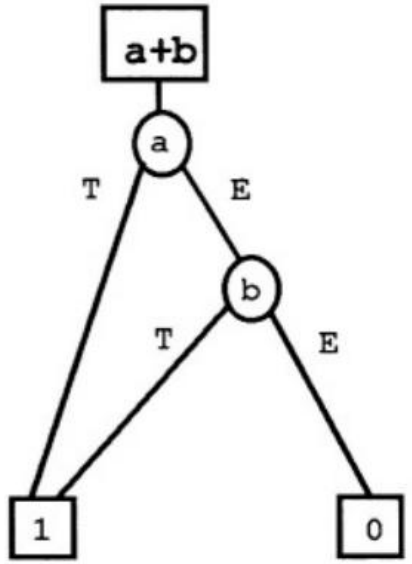
F = a



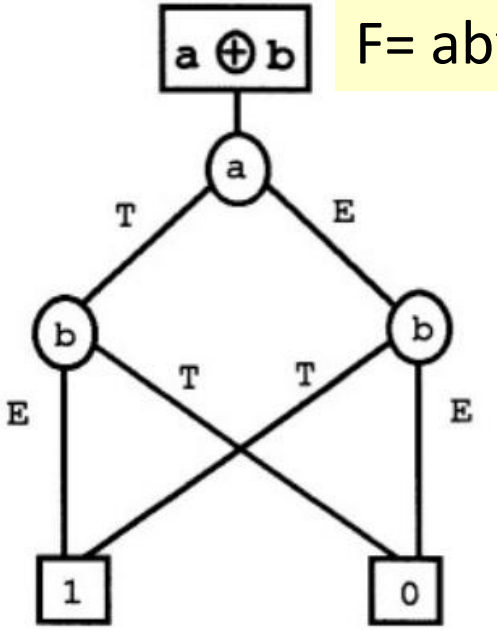


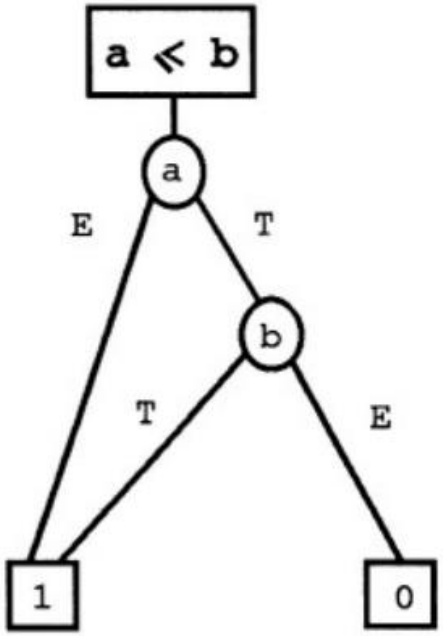
F = ab





F= ab'+a'b





Reduced Ordered BDDs

- Introduced by Randal E. Bryant in mid-80s
  - IEEE Transactions on Computers 1986 paper is one of the most highly cited papers in EECS
- Useful data structure to represent Boolean functions
  - Applications in logic synthesis, verification, program analysis, AI planning, …
- Commonly known simply as BDDs
  - Lee [1959] and Akers [1978] also presented BDDs, but not **RO**BDDs
- Many variants of BDDs have also proved useful
- Links to coding theory (trellises), etc.

https://people.eecs.berkeley.edu/~sseshia/219c/lectures/BinaryDecisionDiagrams.pdf
About BDD (46 pages)

BDDs from Truth Tables

w	x	y	z	f
1	1	0	-	1
1	-	0	1	1
1	-	-	-	1
-	1	-	1	1
0	1	1	-	1
0	0	-	-	1
0	0	1	0	1

Truth Table

↓

Binary Decision Tree

↓

Binary Decision Diagram (BDD)

↓

Ordered Binary Decision Diagram (OBDD)

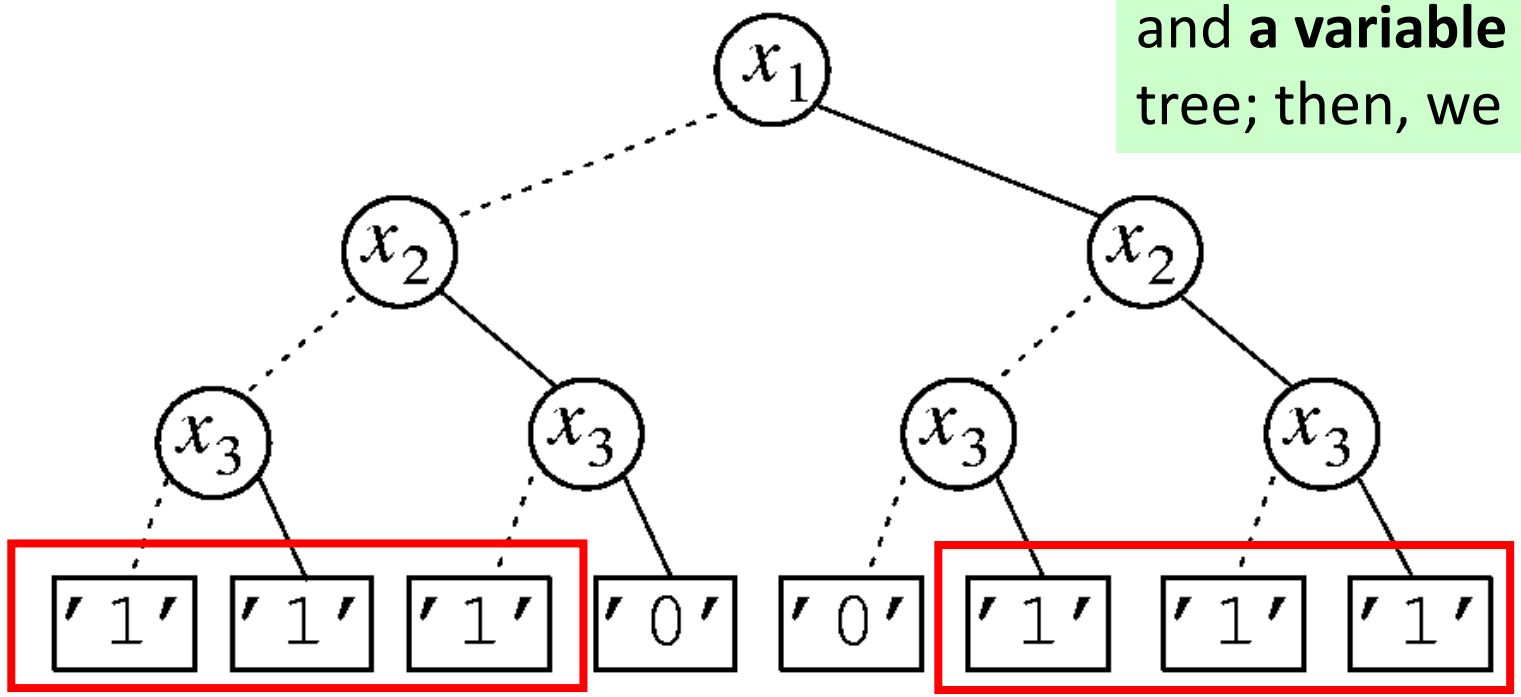
↓

Reduced Ordered Binary Decision Diagram
(ROBDD, simply called BDD)

Example Ordered Binary-Decision Diagram (OBDD)

- The complete Shannon expansion can be visualized as a tree (solid lines correspond to the positive cofactors and dashed lines to negative cofactors).

From its truth table (or logic function) and **a variable ordering**, we get this tree; then, we reduce it



$$f = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3$$

$$f = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3$$

Reduction

- Identify Redundancies

- 3 Rules:

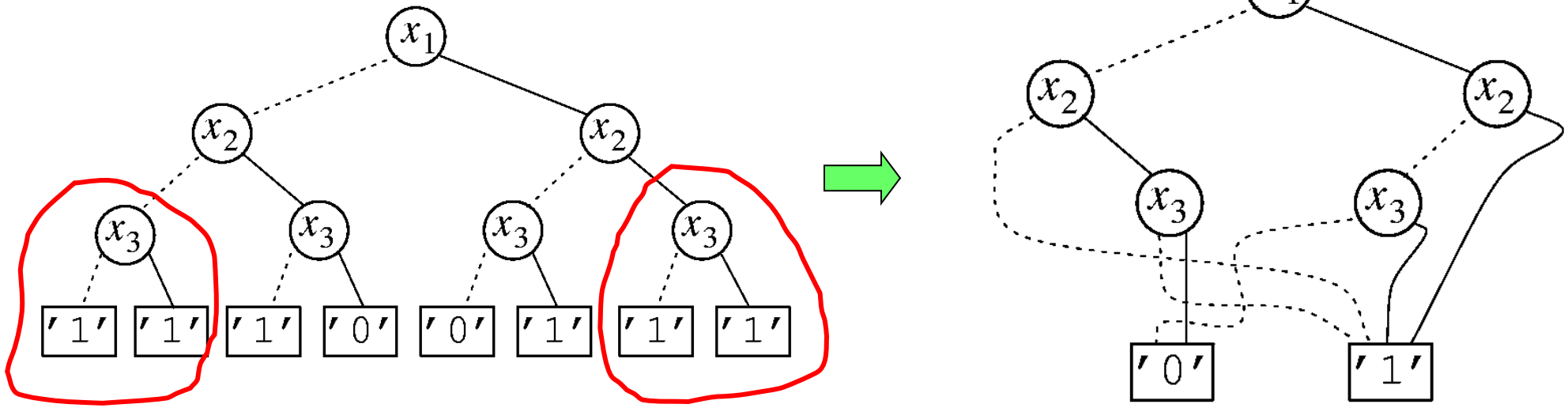
1. Merge equivalent leaves
2. Merge isomorphic nodes
3. Eliminate redundant tests

Core Operators

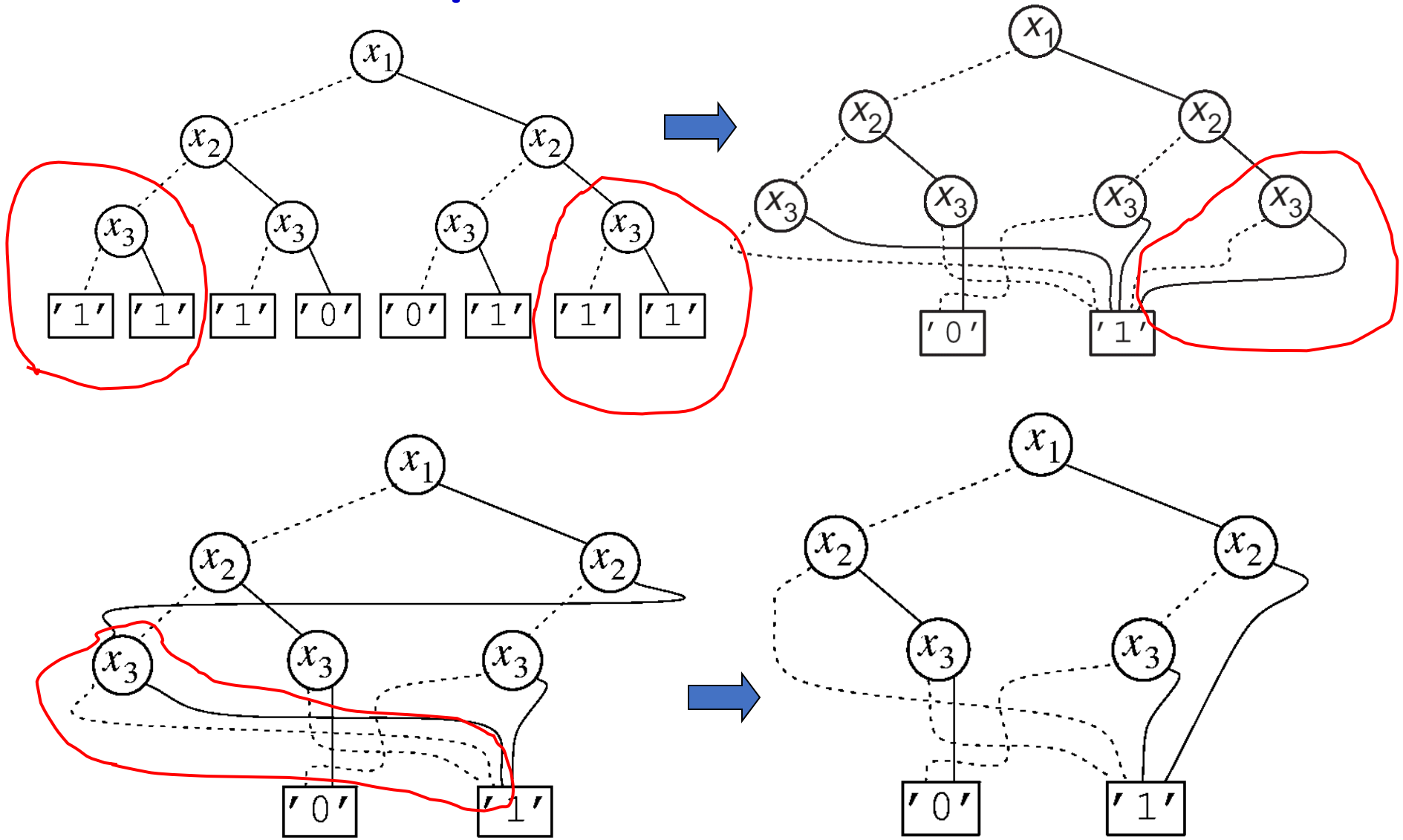
Just two of them!

1. Restrict(Function F, variable v, constant k)
   - Shannon cofactor of F w.r.t. v=k

2. ITE(Function I, Function T, Function E)
   - “if-then-else” operator



Reduction Example



Logic Synthesis (by Gary Hachtel, Fabio Somenzi)
(page 245)

As we shall see, BDDs have two **remarkable** properties.

First, they are **canonical**, so if you correctly build the BDDs for two circuits, the two circuits are equivalent if and only if the BDDs are identical. This has led to significant breakthroughs in circuit optimization, testing, and equivalence checking.

Second, BDDs are amazingly effective at representing combinatorially large sets

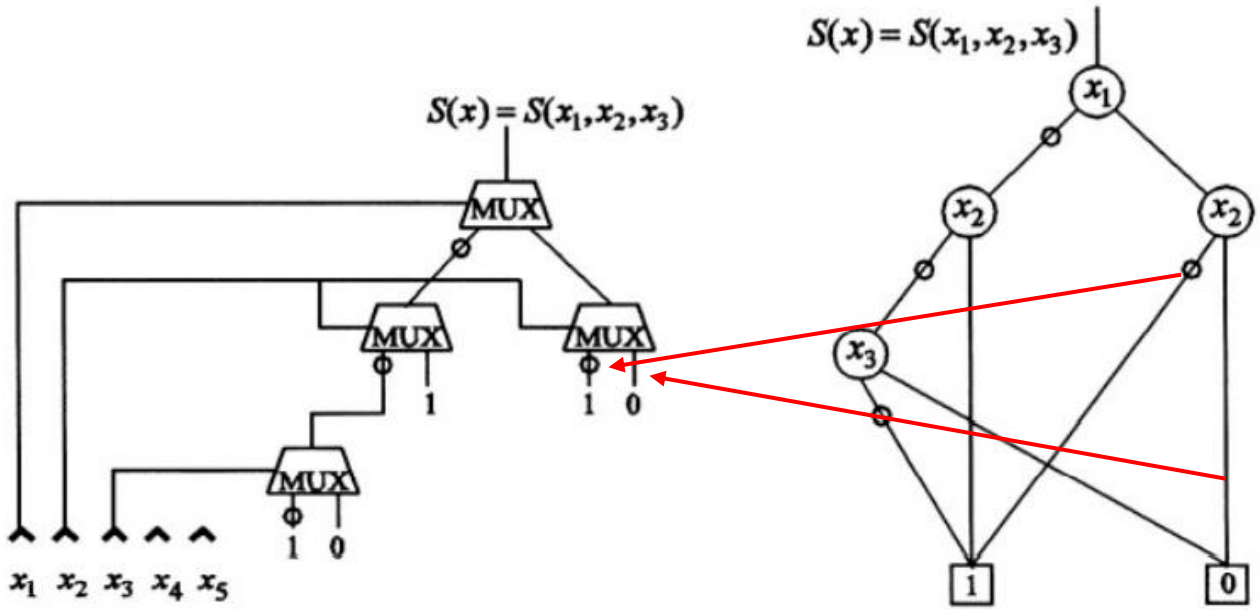


Figure 6.1: A MUX circuit and the corresponding BDD.

S= x1'X2 + x1'x2'x3' +x1x2'

https://people.eecs.berkeley.edu/~sseshia/219c/lectures/BinaryDecisionDiagrams.pdf
About BDD (46 pages)

How to Build The BDD? An Example

- BDDS can be built from recursive use of Boole's expansion theorem

$$f = abc + b'd + c'd$$

under the variable ordering:

$$b \le c \le d \le a.$$

the cofactors of with respect to **b**:

$$f_b = ac + c'd$$

$$f_{b'} = d + c'd.$$

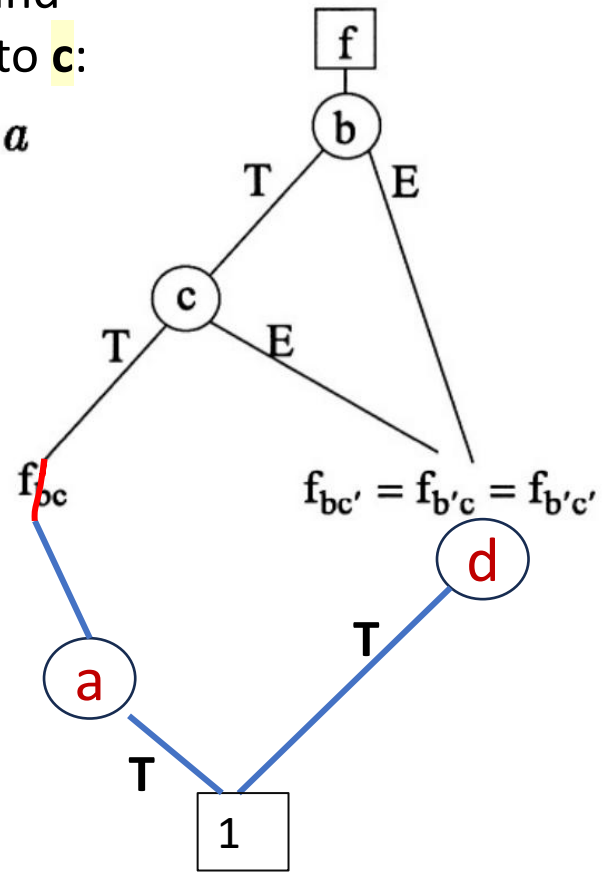
cofactors of and with respect to **c**:

$$(f_b)_c = f_{bc} = a$$

$$f_{bc'} = d$$

$$f_{b'c} = d$$

$$f_{b'c'} = d$$



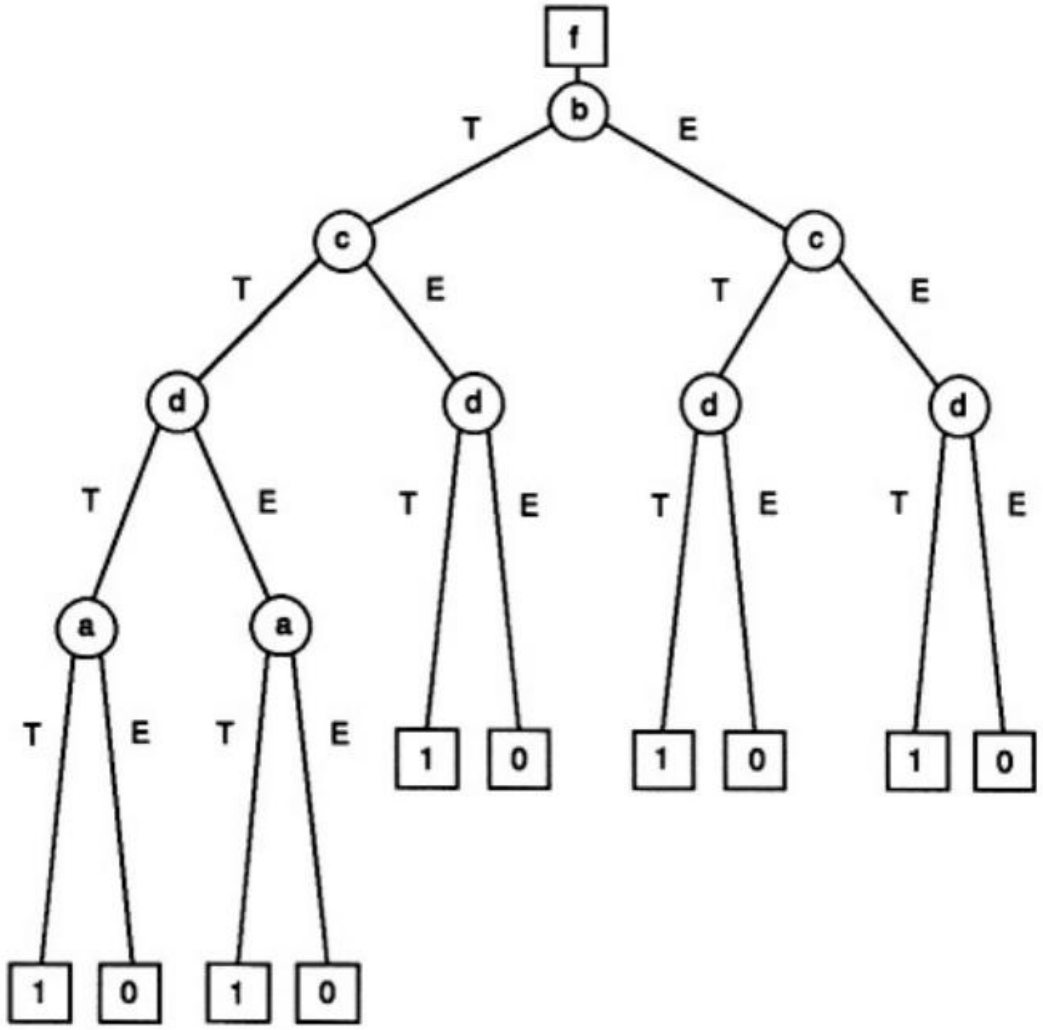


Figure 6.9: Non-reduced BDD.

Reduction

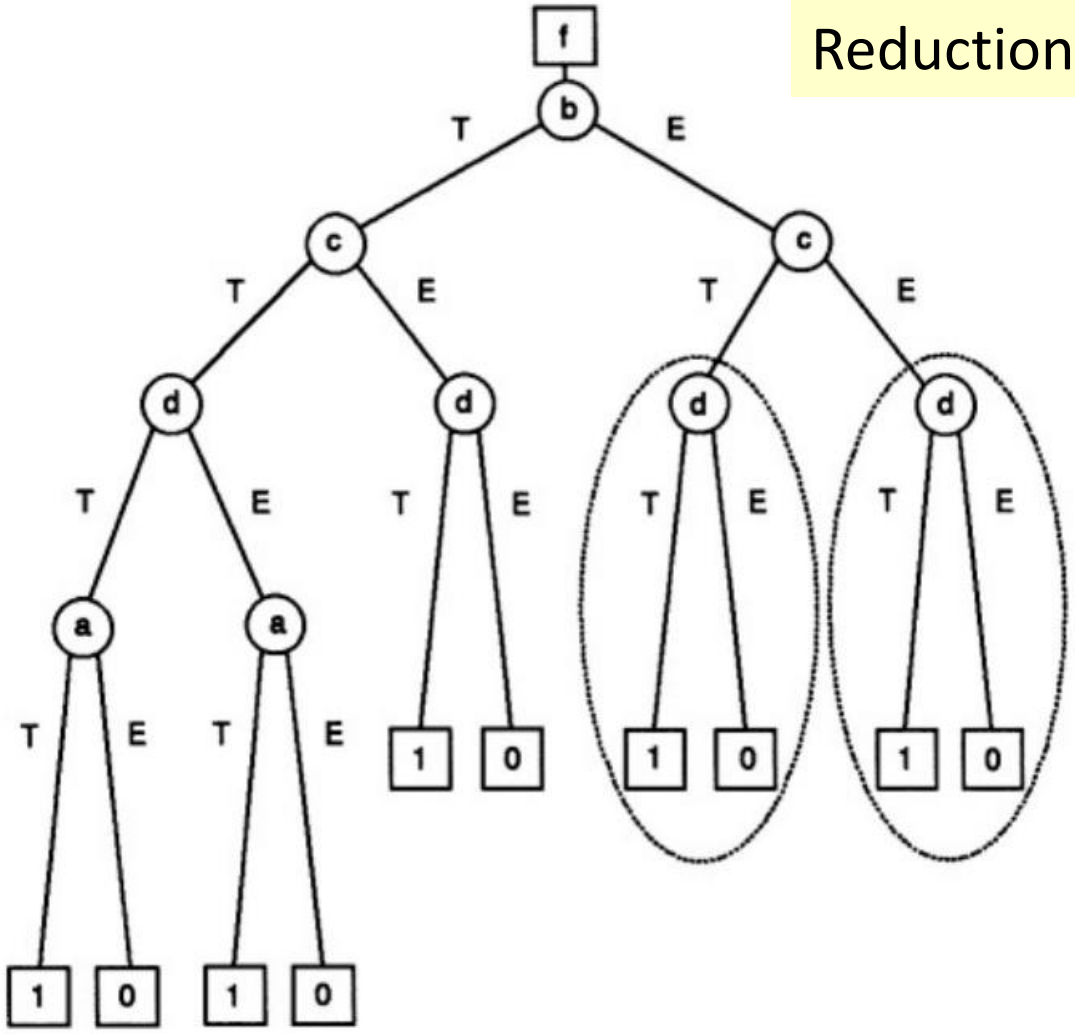


Figure 6.10: Two isomorphic subgraphs.

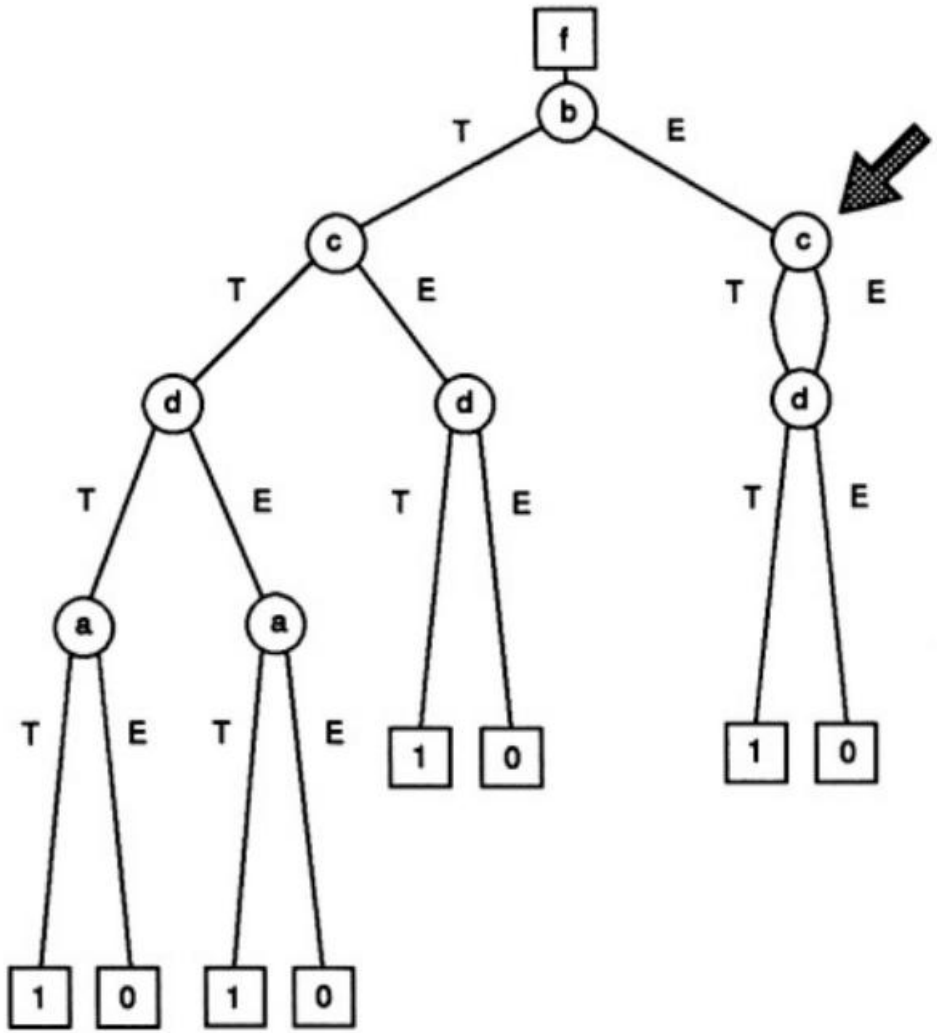


Figure 6.11: Merging two isomorphic subgraphs.

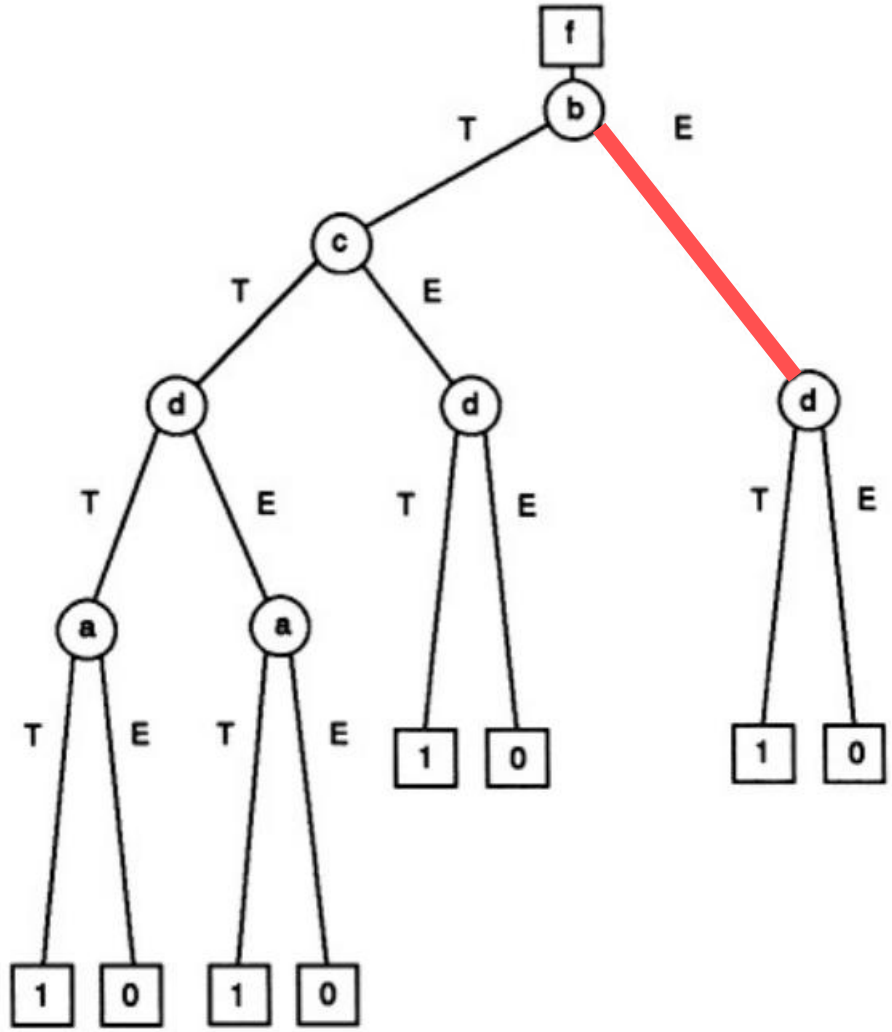


Figure 6.12: Elimination of a redundant node.

$$f = abc + b'd + c'd$$

under the variable ordering:

$$b \leq c \leq d \leq a.$$

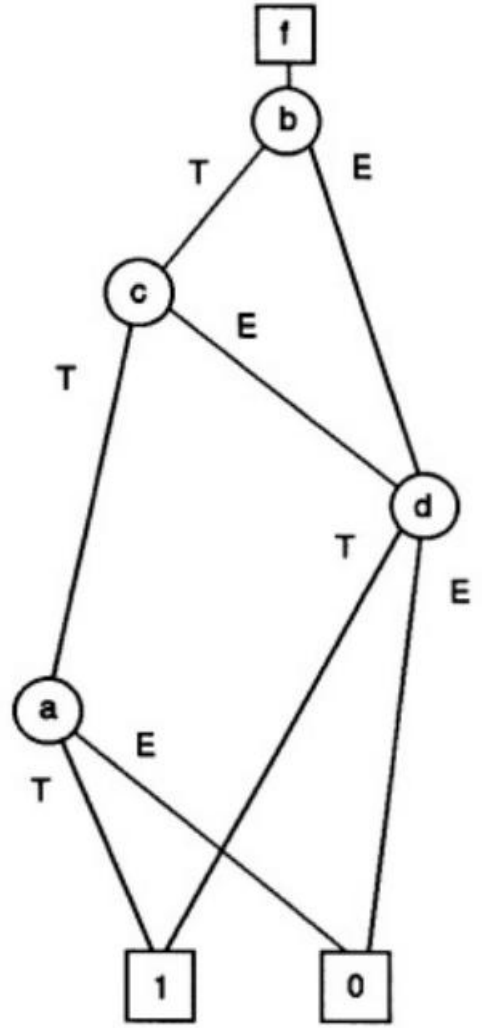
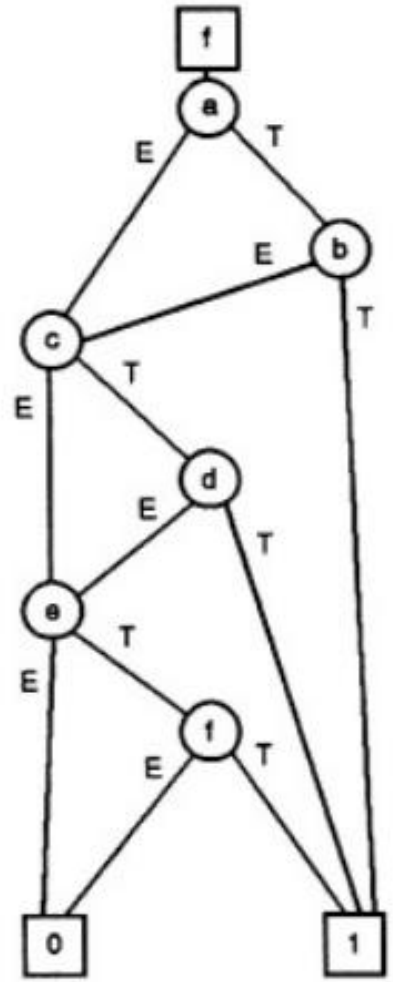


Figure 6.8: Final BDD.

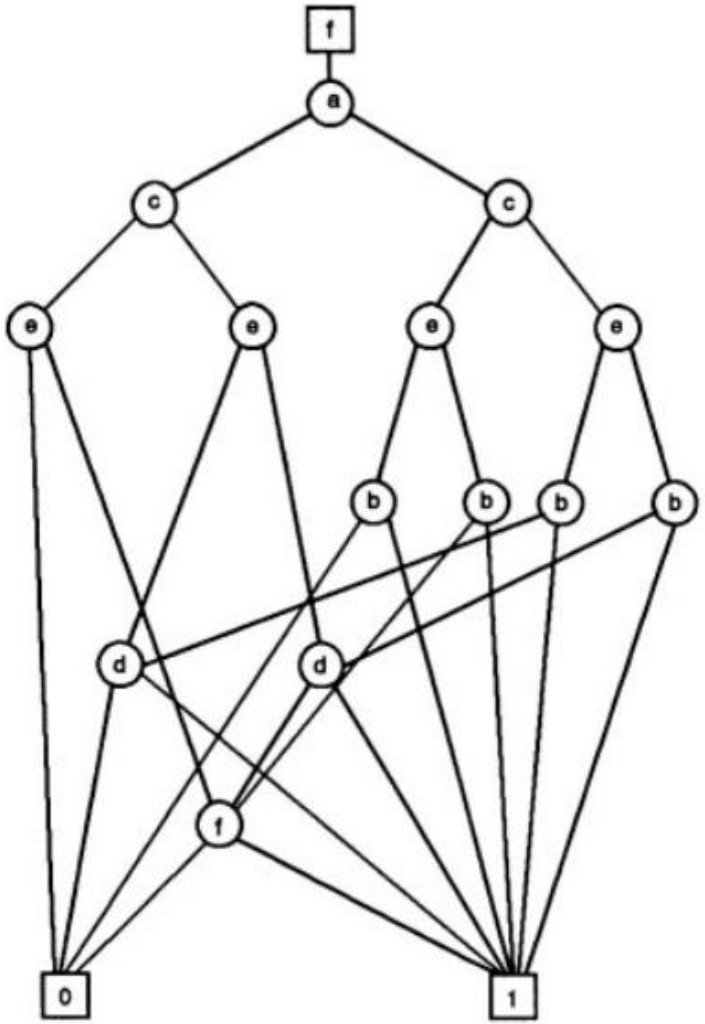
Ordering Is Important

$$f = ab + cd + ef$$

$a \leq b \leq c \leq d \leq e \leq f.$



$a \leq c \leq e \leq b \leq d \leq f.$



Second, let us consider the following function, given in SOP form:

$$f = abc + b'd + c'd.$$

Variable ordering matters

A BDD for this function is given in Figure 6.2. If we want to know the value of f for a

a<b<c<d

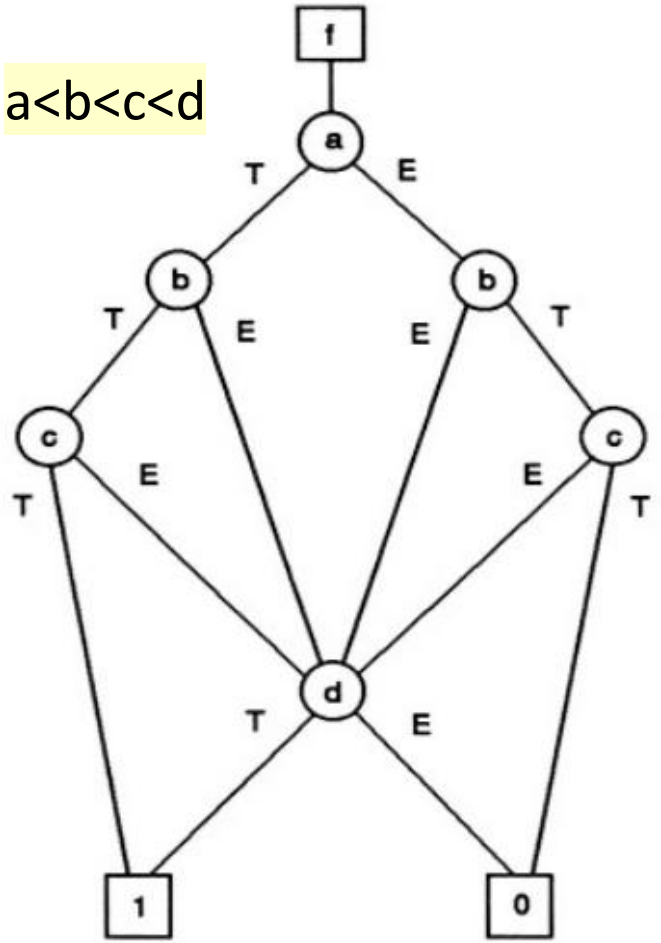


Figure 6.2: A binary decision diagram.

a<d<b<c

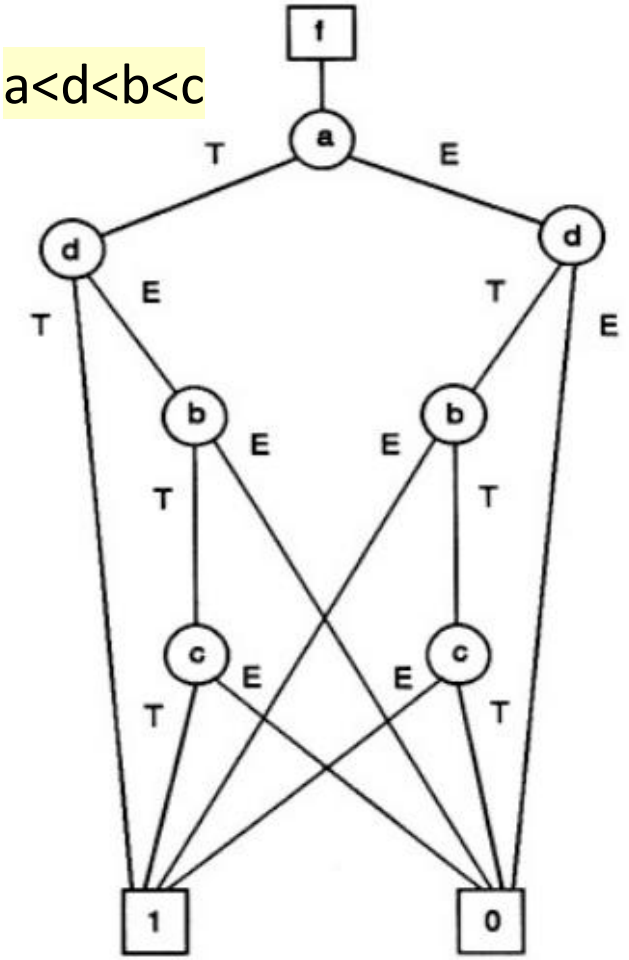


Figure 6.3: Another BDD.

b<c<a<d

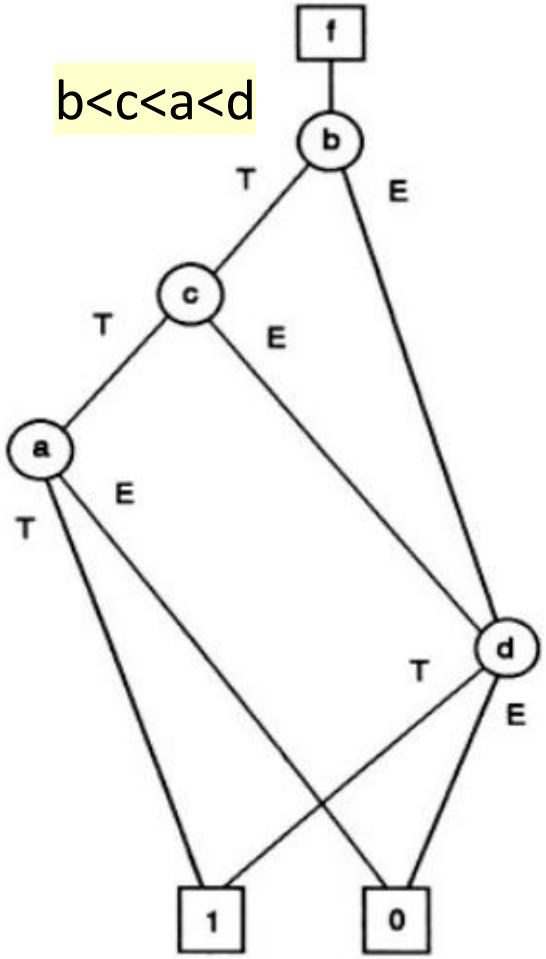


Figure 6.4: An optimal BDD.

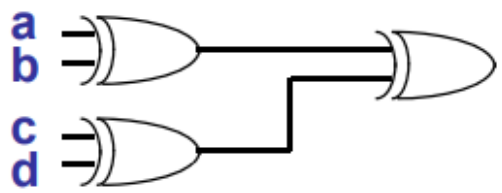
Some BDD packages

- CUDD – from Colorado University, Fabio Somenzi's group
  - Front-ends to CUDD in various languages including Perl and Python
- BuDDy – from IT Univ. of Copenhagen

https://davidkebo.com/cudd?fbclid=IwAR15HjgUfWraRYOhKfB9kn09k8cTWKAiVVsoABZ46TFalEEfRiAq0AC5X6o

Good CUDD tutorial (worthwhile)

Example: Odd Parity Function



Edge labels along a root-leaf path form an assignment to a,b,c,d

Binary Decision Tree

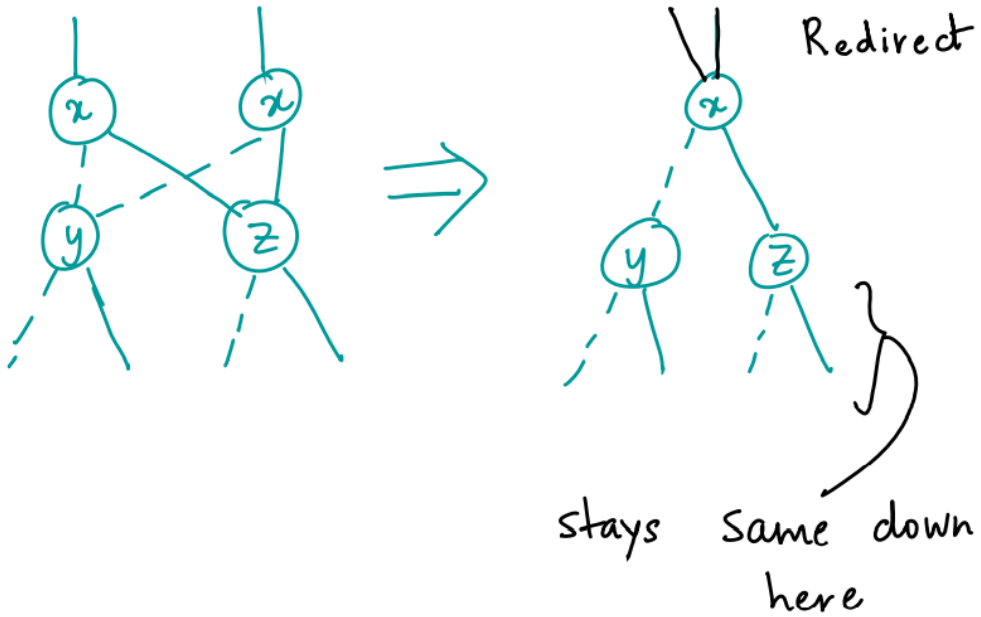
Reduction

- Identify Redundancies

- 3 Rules:

1. Merge equivalent leaves
2. Merge isomorphic nodes
3. Eliminate redundant tests

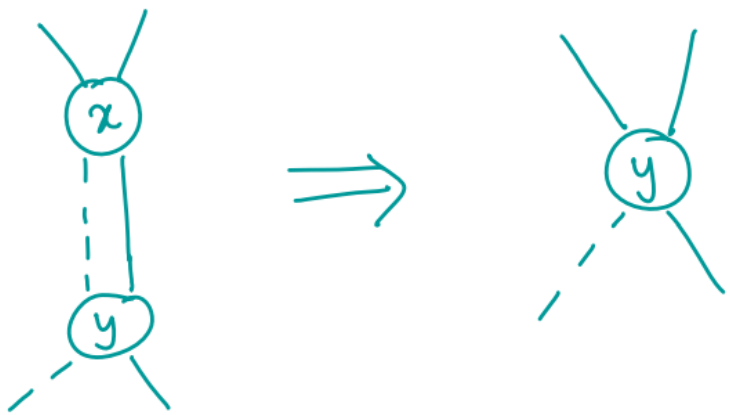
Merge Isomorphic Nodes



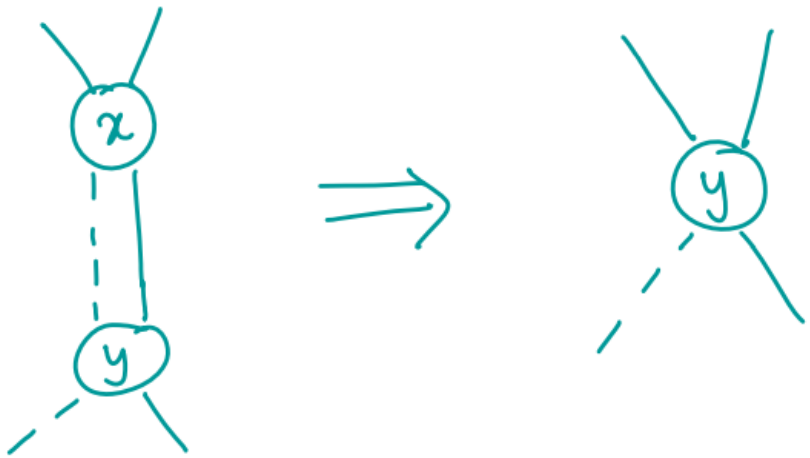
Merge Equivalent Leaves

"a" is either 0 or 1

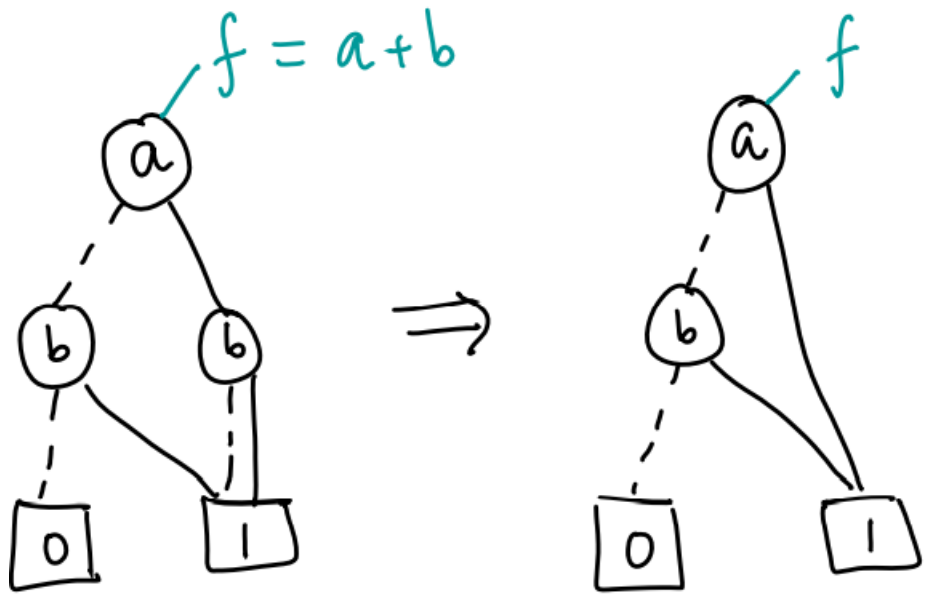
Eliminate Redundant Tests



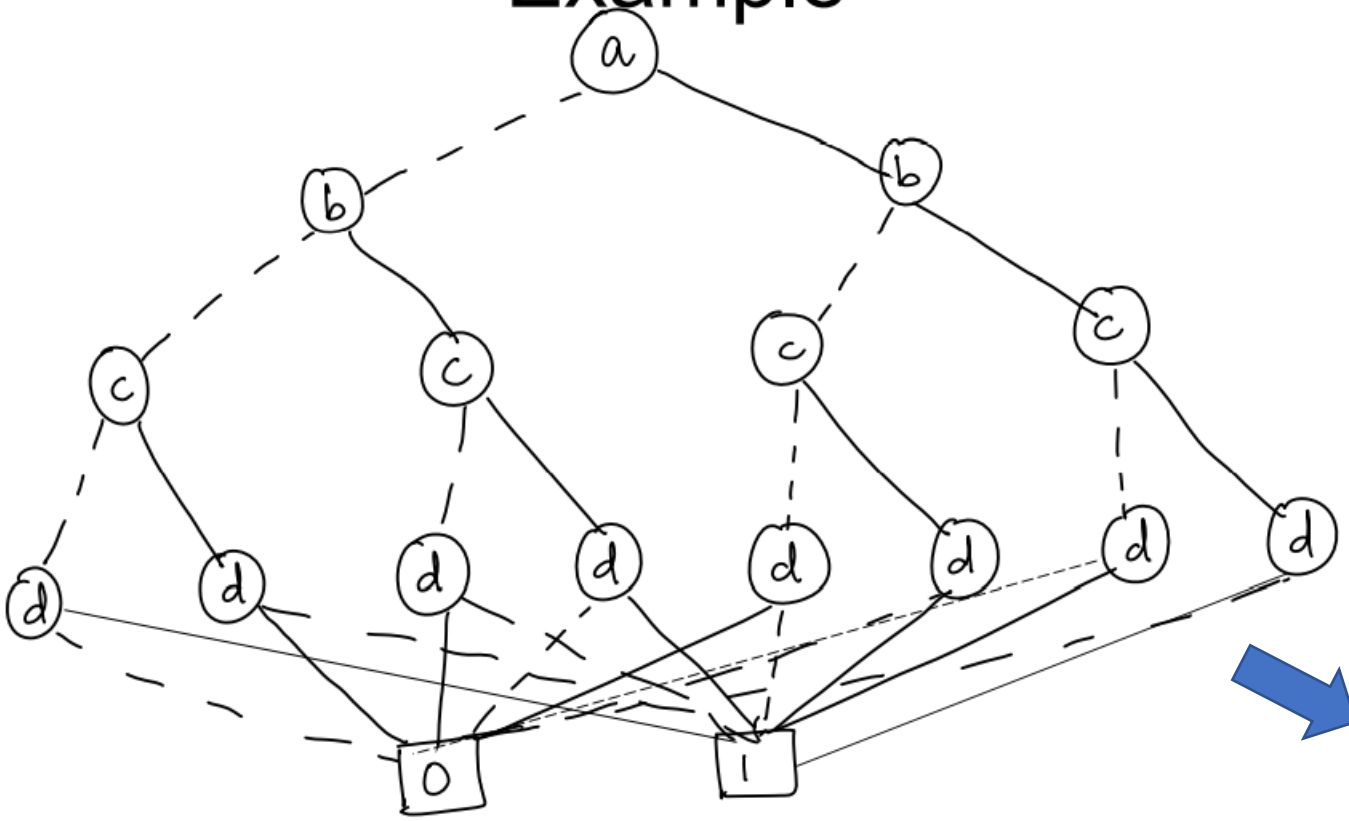
Eliminate Redundant Tests



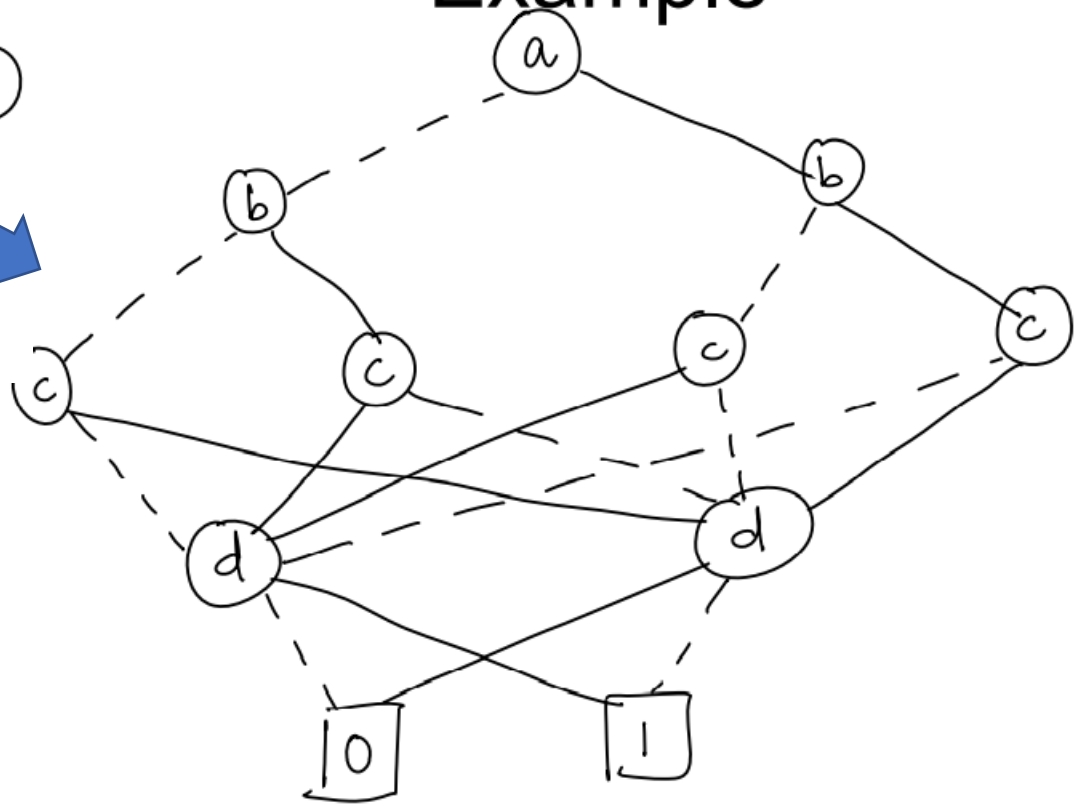
Example of Rule 3



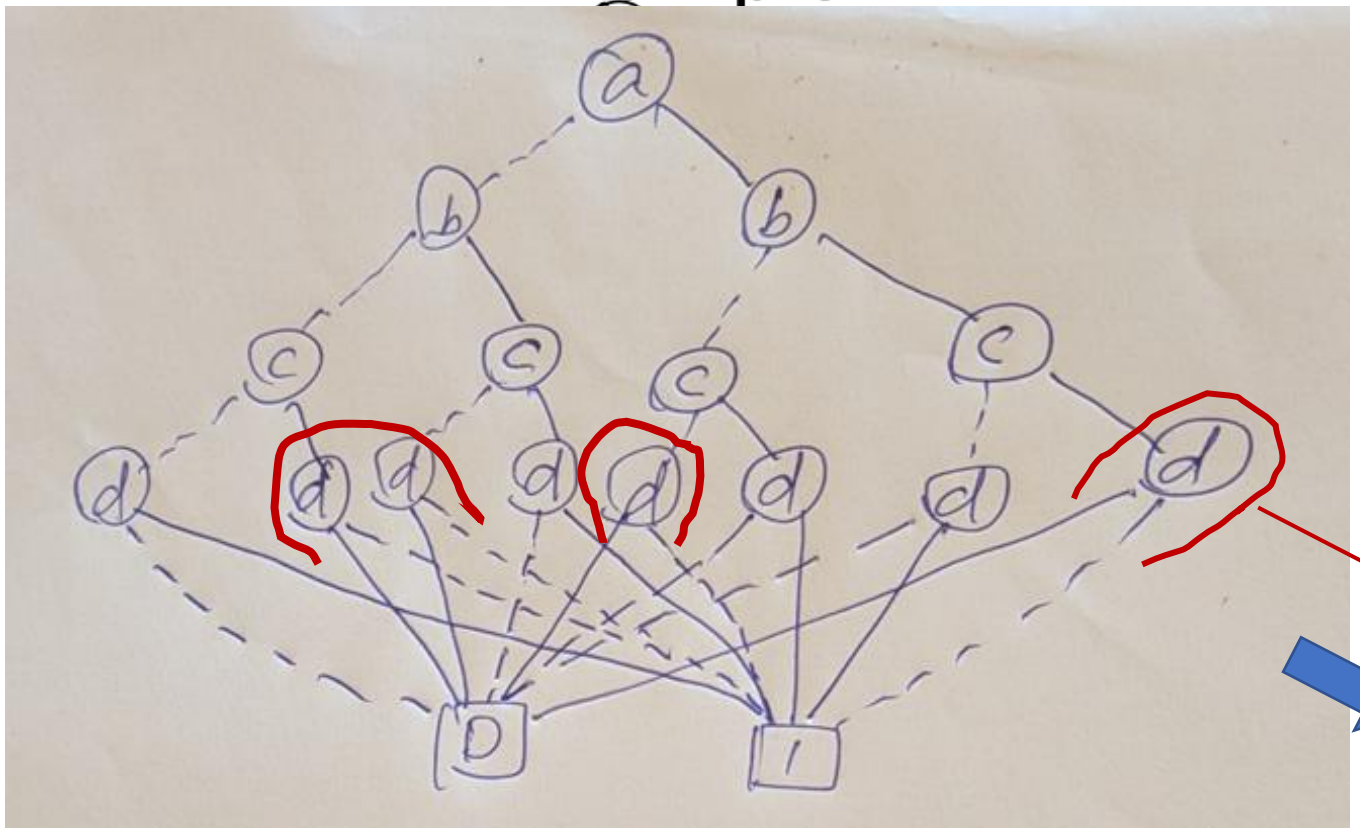
Example



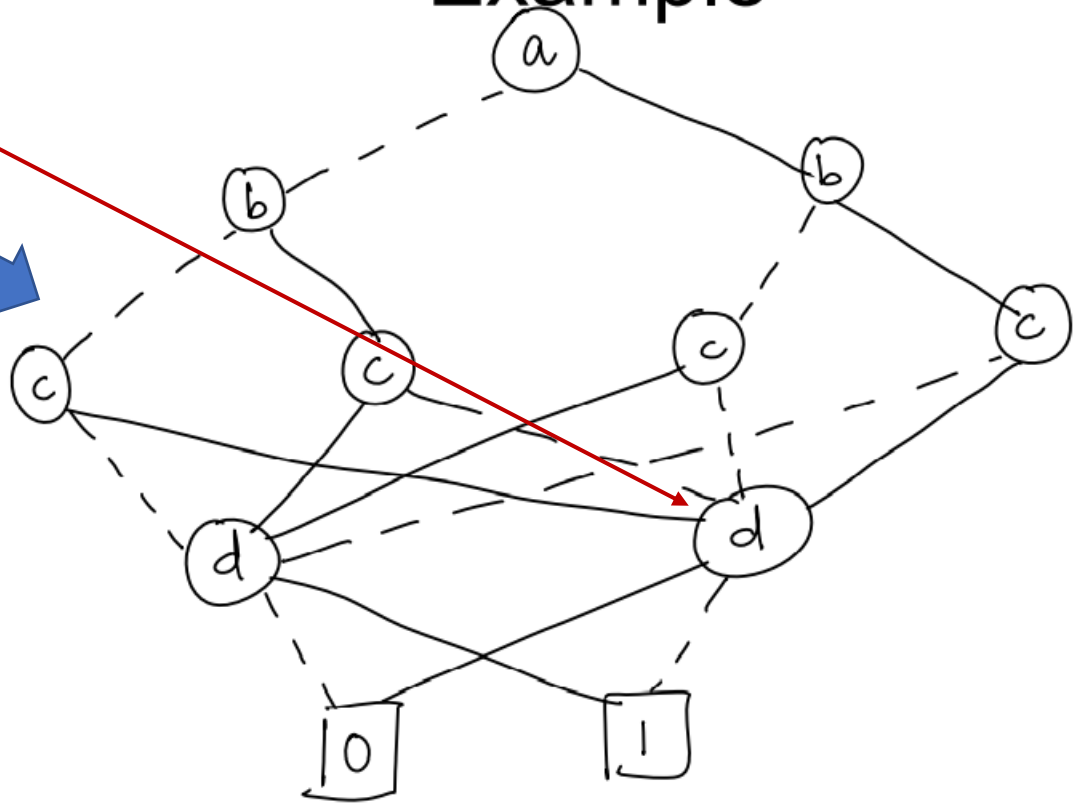
Example



Example

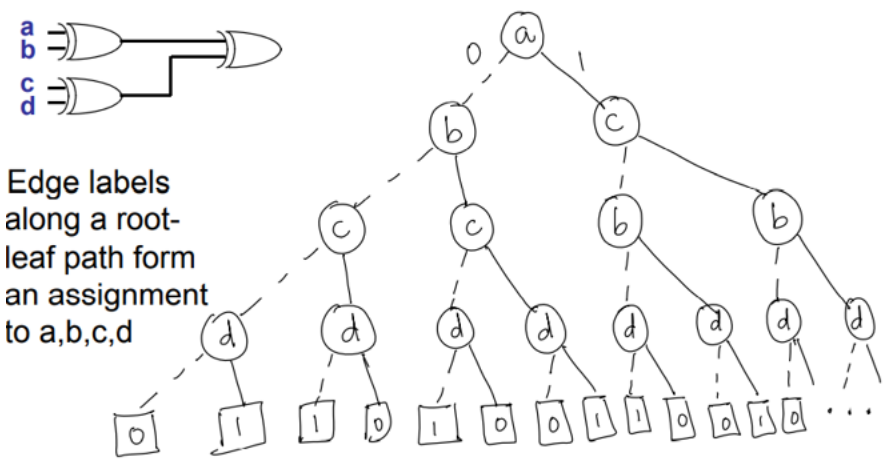


Example

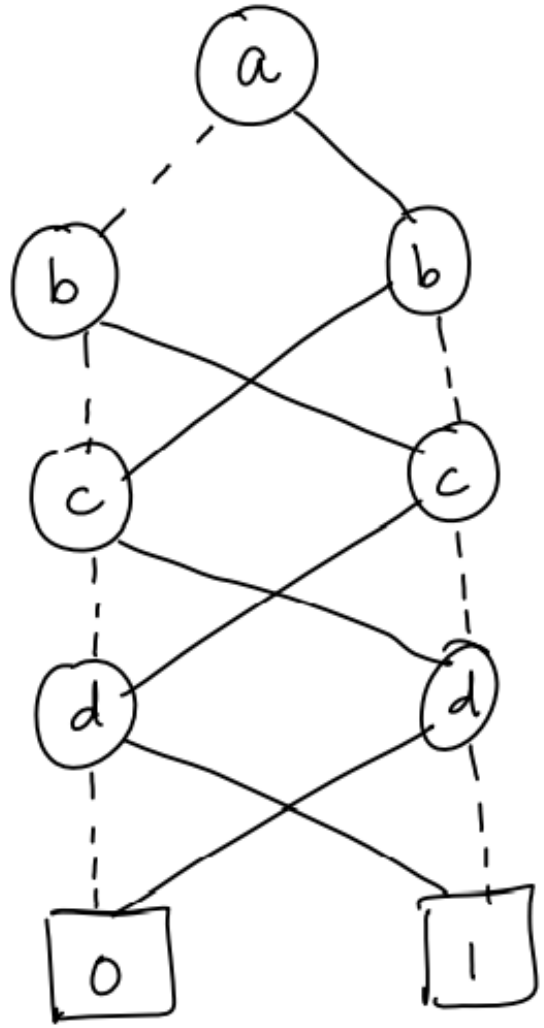


Final ROBDD for Odd Parity Function

Example: Odd Parity Function



Binary Decision Tree



2n-1 non-terminal nodes

What can BDDs be used for?

- Uniquely representing a Boolean function
  - And a Boolean function can represent sets
- Symbolic simulation of a combinational (or sequential) circuit
- Equivalence checking and verification
  - Satisfiability (SAT) solving
- Finding / counting *all* solutions to a SAT (combinatorial) problem
- Operations on “quantified” Boolean formulas

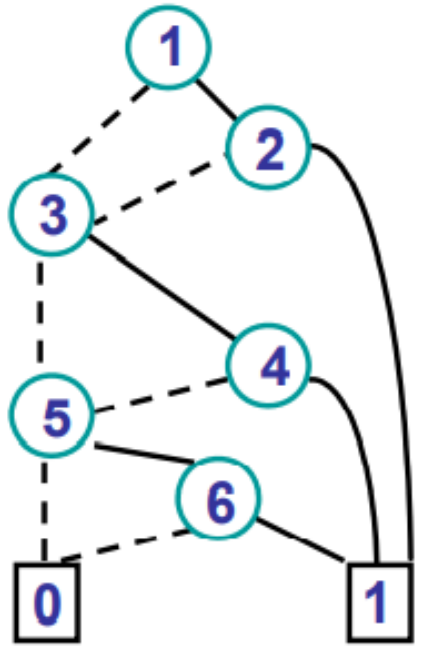
(RO)BDDs are canonical

- Theorem (R. Bryant): If G, G' are ROBDD's of a Boolean function f with k inputs, using same variable ordering, then G and G' are identical.

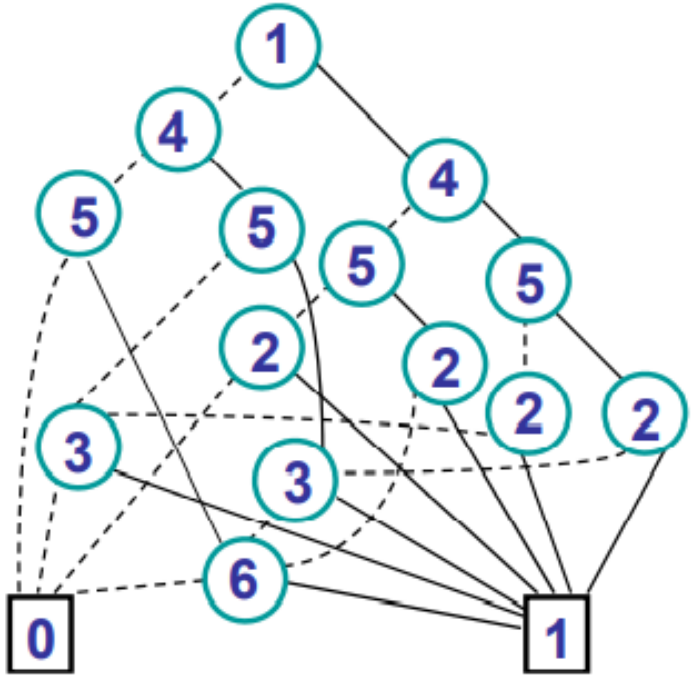
Sensitivity to Ordering

- Given a function with n inputs, one input ordering may require exponential # vertices in ROBDD, while other may be linear in size.
- Example: $f = x_1 x_2 + x_3 x_4 + x_5 x_6$

$x_1 < x_2 < x_3 < x_4 < x_5 < x_6$



$x_1 < x_4 < x_5 < x_2 < x_3 < x_6$



To find an ordering that results in the smallest ROBDD, unfortunately, this is an intractable problem (the problem is co-NP-complete to be precise

Constructing BDDs in Practice

- Strategy: Define how to perform basic Boolean operations
- Build a few core operators and define everything else in terms of those

Advantage:
- **Less programming work**
- **Easier to add new operators later by writing "wrappers"**

Core Operators

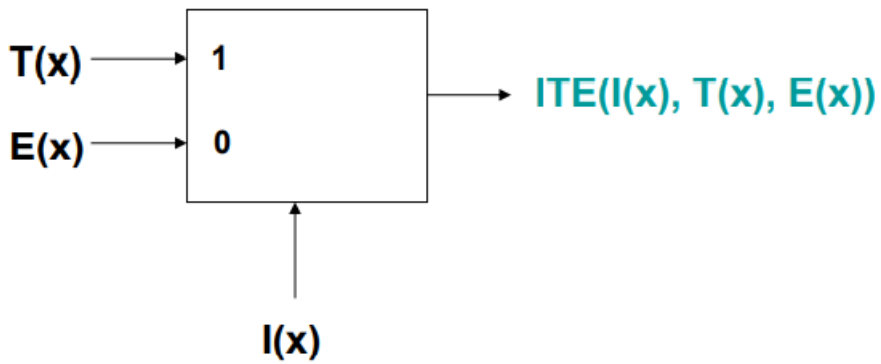
Just two of them!

1. Restrict(Function F, variable v, constant k)
   - Shannon cofactor of F w.r.t. v=k

2. ITE(Function I, Function T, Function E)
   - "if-then-else" operator

ITE

- Just like:
  - "if then else" in a programming language
  - A mux in hardware
- ITE(I(x), T(x), E(x))
  - If I(x) then T(x) else E(x)



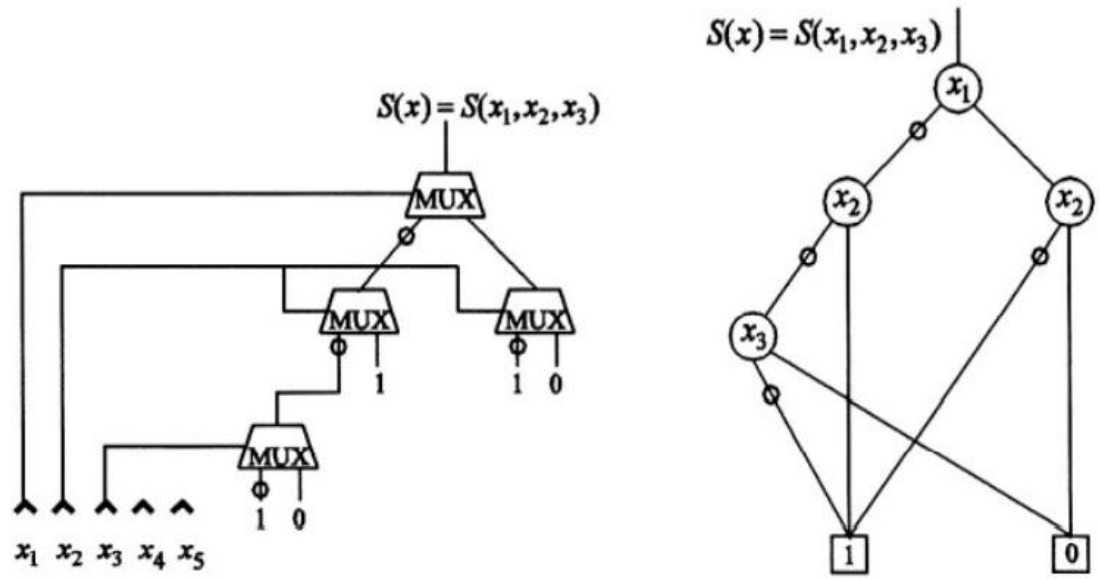


Figure 6.1: A MUX circuit and the corresponding BDD.

The ITE Function

ITE(I(x), T(x), E(x))

=

I(x) . T(x) + I'(x). E(x)

How do we implement ITE?

- Divide and conquer!
- Use Shannon cofactoring…
- Recall: Operator of cofactors is Cofactor of operators…

Size Reduction Of An OBDD To ROBDD

1. Replace all leaf vertices v with identical $\phi(v)$ by a single vertex and redirect all edges incident to the original vertices to this single vertex.

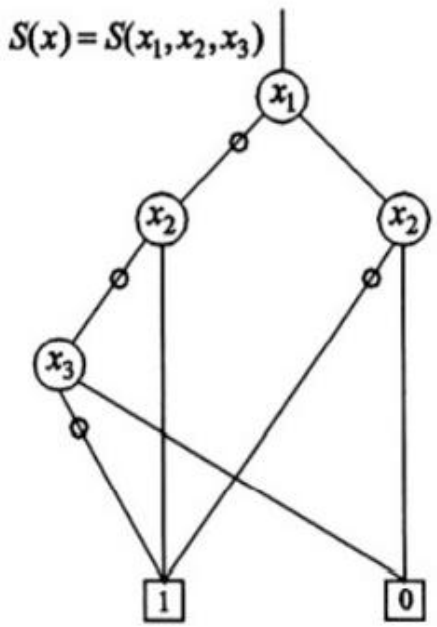
2. Process all vertices from bottom to top. If two vertices u and v are found for which $\phi(u) = \phi(v)$, $\eta(u) = \eta(v)$ and $\lambda(u) = \lambda(v)$, remove v and redirect to u all edges originally incident to v.

3. If edges v exist for which $\eta(v) = \lambda(v)$, remove v and redirect to $\eta(v)$ all edges originally incident to v.

ROBDDs and Satisfiability

- A Boolean function is **satisfiable** if an assignment to its variables exists for which the function becomes '1'
- Any Boolean function whose ROBDD is unequal to '0' is satisfiable.
- Suppose that choosing a Boolean variable x_i to be '1' costs c_i. Then, the **minimum-cost satisfiability** problem asks to minimize: $\sum_{i=1}^{n} c_i \mu(x_i)$

  where $\mu(x_i) = 1$ when $x_i =$ '1' and $\mu(x_i) = 0$ when $x_i =$ '0'.
- Solving minimum-cost satisfiability amounts to computing the shortest path in an ROBDD, which can be solved in linear time.
  - Weights: $w(v, \eta(v)) = c_i$, $w(v, \lambda(v)) = 0$, variable $x_i = \phi(v)$.



Project

- Bringing up CUDD and play around a few Boolean equations

Applications to Combinatorial Optimization

- **Zero-one integer linear programming** can be formulated as a minimum-cost satisfiability problem.
- Consider the (standard form) constraint: $x_1 + x_2 + x_3 + x_4 = 3$.
- It can be written as:

$$(x_1 + x_2) \cdot (x_1 + x_3) \cdot (x_1 + x_4) \cdot (x_2 + x_3) \cdot (x_2 + x_4) \cdot (x_3 + x_4) \cdot (\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + \overline{x}_4)$$

  - The first 6 sums in the product: at least 3 of the 4 variables are 1.
  - The last sum: at least one of the variables is 0.
- Many combinatorial optimization problems can also be directly formulated in terms of the satisfiability problem.

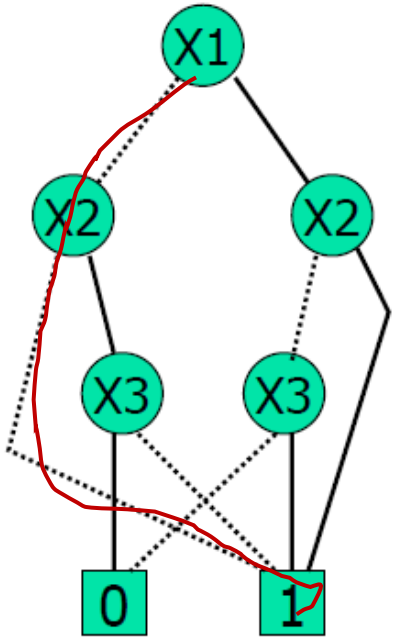
Only 3 variables can be 1

Use of ROBDD for Two-Level Logic Implementation

- We have learned ROBDD is a compact and canonical representation of Boolean functions

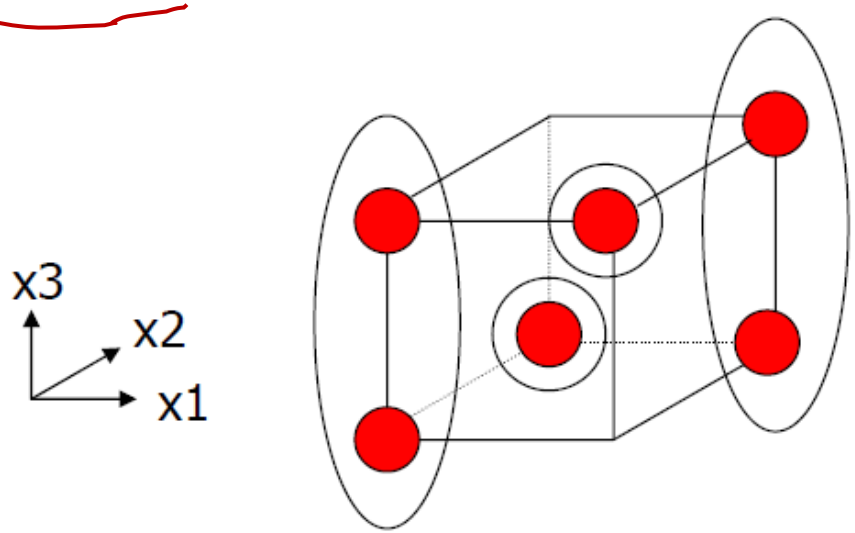
➔ Can we directly use the BDD representation of a Boolean function for two-level logic implementation?

- Consider this example ---



Enumerate the cubes:

x1'x2' + x1'x2x3' + x1x2'x3 + x1x2



Currently 4 product terms. Can we do better? (finding minimal cover)

A Heuristic Based on ROBDDs

- "isop" (Irredundant Sum of Products) proposed by Minato
  - Note: cubes in BDD are irredundant, but they are NOT prime implicant
  - For logic optimization: need to collect irredundant prime implicants → irredundant prime cover
  - Recursive call: $f = x' f_0 + x f_1 + f_d$

    where $f_0 \subset f_{x'}$, $f_1 \subset f_x$, $f_d \subset (f_{x'} \cap f_x)$

    and f_0, f_1, f_d are irredundant prime cover
  - Can expanded to incomplete specified function
  - (Details not covered here. Please refer to Section 11.3.2)

Logic Synthesis in Practice

1. Specify the logic-level behavioral description of the circuit in some hardware-description language.
2. Extract from this description the Boolean expressions related to the logic and represent them in some suitable internal form.
3. Manipulate these expressions to obtain an optimized (two-level or multilevel) representation.
4. Perform **technology mapping**, a mapping from the abstract optimized representation to a netlist of cells from a library.

Two-Level Logic Synthesis

- Any Boolean function can be realized in two levels: AND-OR (sum of products), NAND-NAND, etc.
- Direct implementation of two-level logic using PLAs (programmable logic arrays) is not as popular as in the nMOS days.
- Classic problems, solved e.g. by Karnaugh maps [1953] or the *Quine-McCluskey* algorithm [1956].
  - Basic idea: Boolean law $x+x'=1$ allows for grouping $x_1x_2+x_1x'_2=x_1$
- Popular cost function: the number of literals in the sum of products expression.
- The goal is to find a minimal irredundant prime cover.

Two-Level Logic Optimization

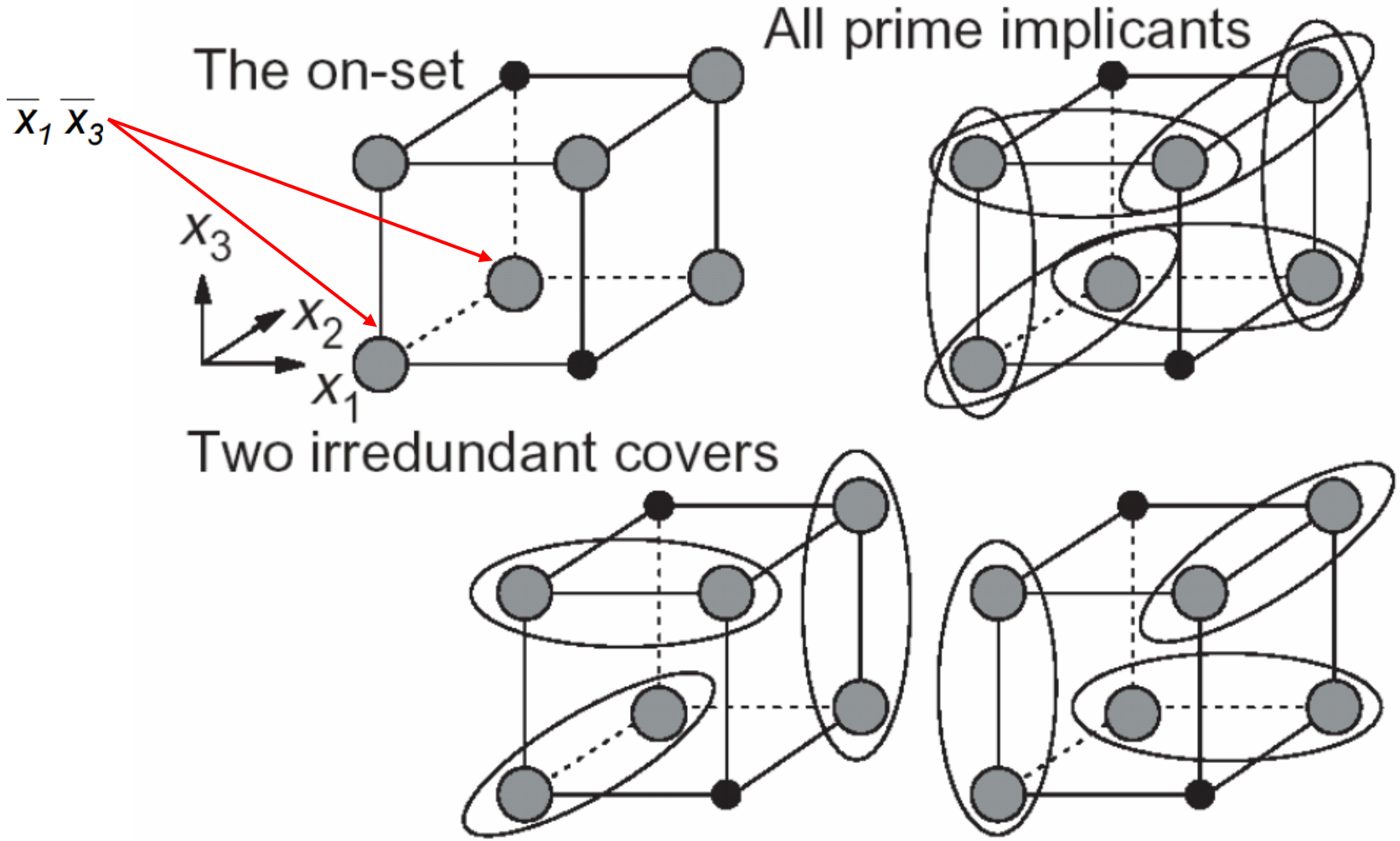
- Two-level logic representations: Sum-of-product form, product-of-sum form, NAND-NAND, NOR-NOR
- Two-level logic optimization
  - Key technique in logic optimization
  - Many efficient algorithms to find a near minimal representation in a practical amount of time
  - In commercial use for several years
  - Minimization criteria: **number of product terms**
- Example:

$$F = \bar{x_1}\bar{x_2}\bar{x_3} + \bar{x_1}\bar{x_2}x_3 + x_1\bar{x_2}\bar{x_3} + x_1\bar{x_2}x_3 + x_1x_2\bar{x_3} \Rightarrow F = \bar{x_2} + x_1\bar{x_3}.$$

- Methods/CAD tools: The Quine-McCluskey method (exponential-time exact algorithm), Espresso (heuristics)

Cover Examples

- $f = \overline{x_1}\,\overline{x_3} + \overline{x_2}\,x_3 + x_1\,x_2$
- $f = \overline{x_1}\,\overline{x_2} + x_2\,\overline{x_3} + x_1\,x_3$

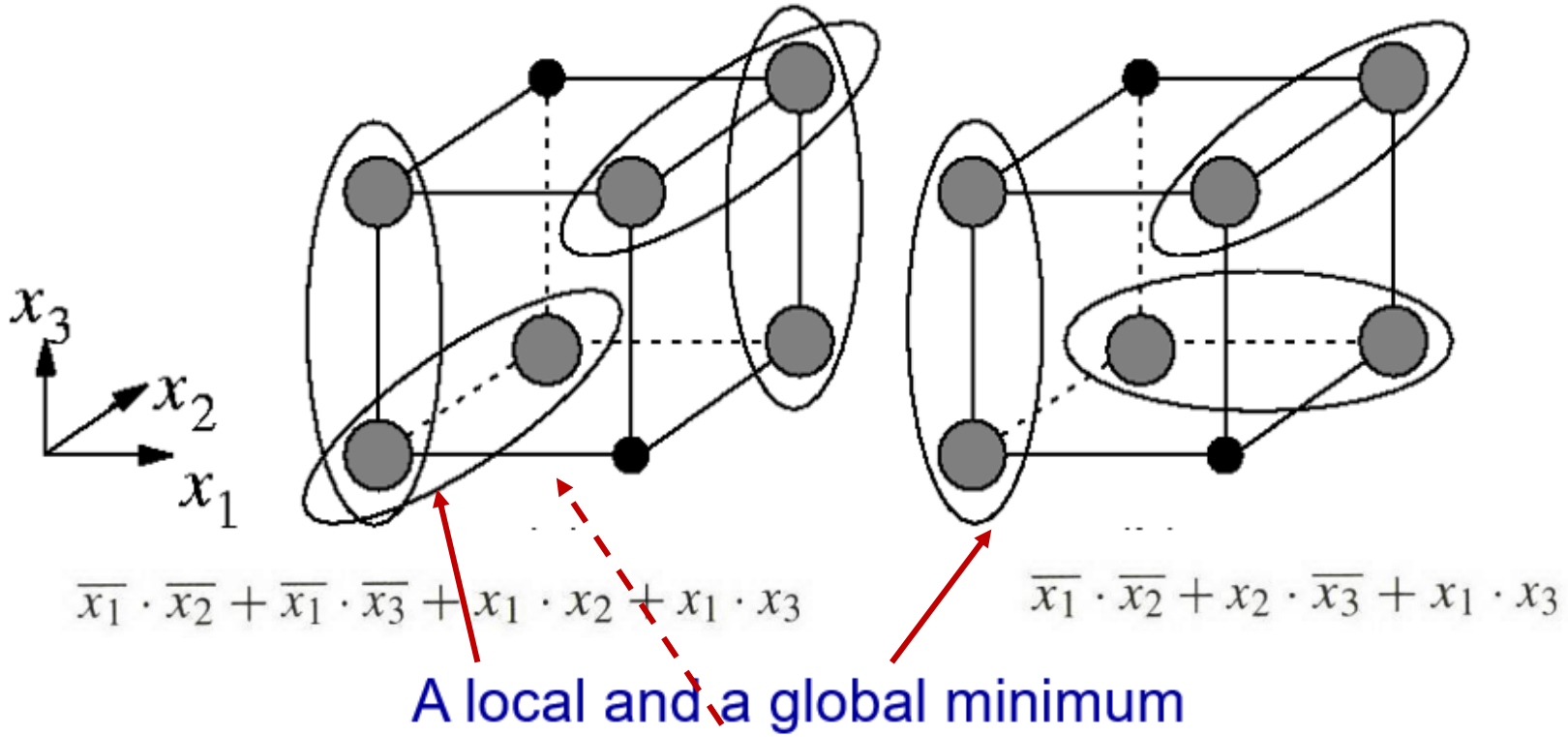


Quine-McCluskey Algorithm:

- Get all prime implicants
- Find minimal cover of all minterms
- Construct covering matrix
- Detecting essential columns, row and column dominance

Adapted from: Prof. Y.-W. Chang's slides

Optimality in Two-Level Logic Synthesis



It has one cube less which gives a cost of 6 (due to 3 product terms with 2 variables). The existence of two solutions shows that an irredundant prime cover is not necessarily a globally optimal solution.

Set Covering

A combinatorial optimization problem

- Given a set $S = \{s_1, \ldots, s_m\}$ and a set $K = \{K_1, \ldots, K_n\}$ where each K_j $(1 \leq j \leq n)$ is a subset of S, find a subset Γ of K such that the **union** of the elements Γ covers S.
- The cost of a cover is the sum of the costs c_j of the elements K_j of Γ.
- Multiple cost functions are possible. E.g., $c_j = 1$ or $c_j = |K_j|$.
- The problem is NP-complete for most cost functions.

Covering Matrix

(page 221 (233 of pdf))

	K_1	K_2	K_3	K_4	K_5	K_6
s_1	1	1	0	0	0	1
s_2	1	0	0	1	1	1
s_3	0	1	1	0	1	0
s_4	0	1	0	1	0	1

- $\Gamma = \{K_3, K_6\}$ is the optimal solution when $c_j = |K_j|$.
- For solution $\Gamma = \{K_1, K_2, K_3\}$, K_3 is redundant.

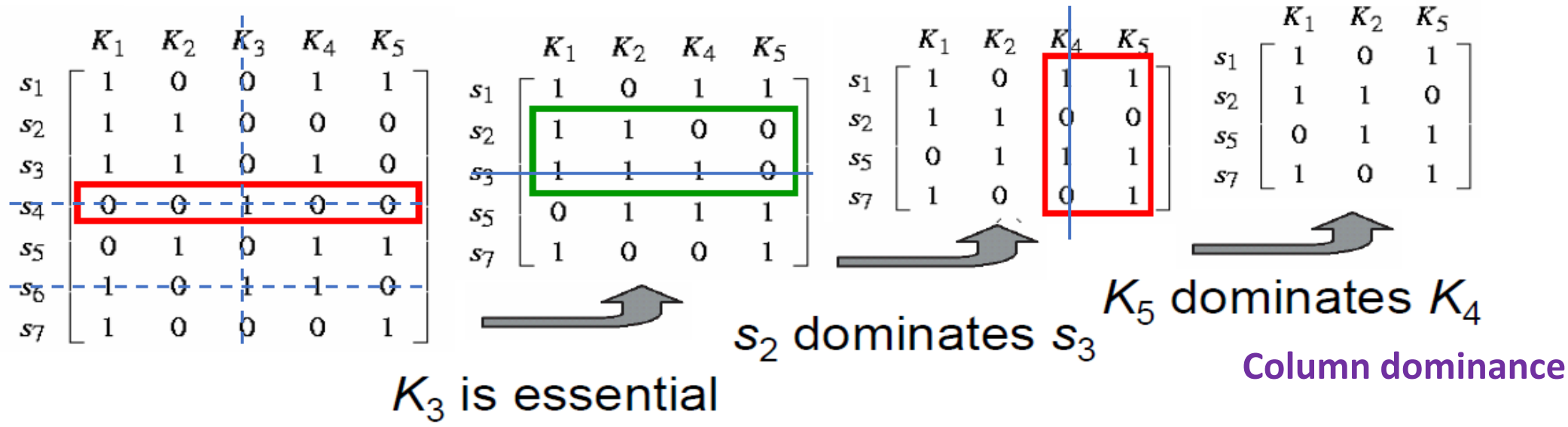
C3=1
C6=3
Optimal cost = 4

C1=2
C2=3
C1+C2=5
C3=1

S1=x1+x2+x6
S2=x1+x4+x5+x6
S3=x2+x3+x5
S4=x2+x4+x6

- A covering problem can be formulated as a satisfiability problem by associating variables x_j with the sets K_j: $(x_1 + x_2 + x_6) \cdot (x_1 + x_4 + x_5 + x_6) \cdot (x_2 + x_3 + x_5) \cdot (x_2 + x_4 + x_6)$.
- This type of covering is called **unate**.
- A **binate** covering problem has an expression where complemented variables are allowed.

Example Simplification Rules in Covering



K3 is **essential**, since s4 has only 1.
Take K3 out (since it is a must)
→ s4 and s6 are covered

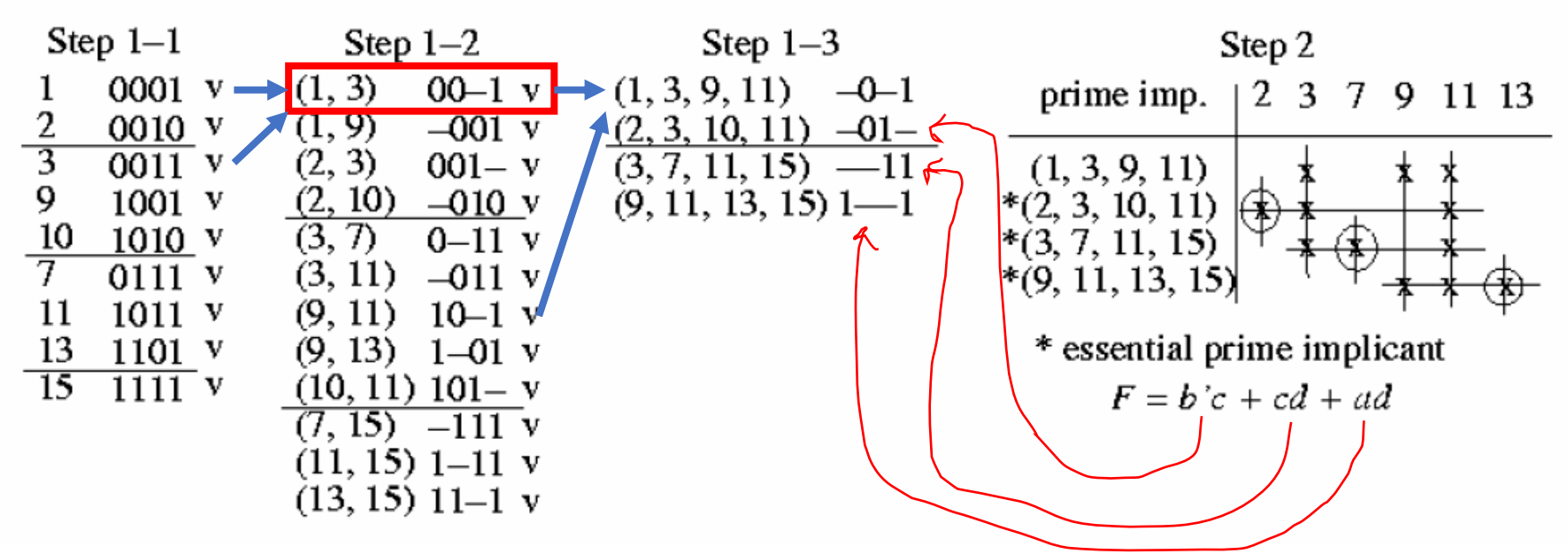
S2 is covered by K1, K2; both covers s3 (**row dominance**)

The Quine-McCluskey Algorithm

- Calculate all prime implicants (of the union of the on-set and dc-set).
- Find the minimal cover of all minterms in the on-set by prime implicants.
- Construct the covering matrix.
- Simplify the covering matrix by detecting **essential** columns, **row and column dominance**.
- What is left is the **cyclic core** of the covering matrix.
  - The covering problem can then be solved by a branch-and-bound algorithm.
- Other methods do not first enumerate all prime implicants; they use an implicit representation by means of ROBDDs.

The Quine-McCluskey Algorithm

- $F(a, b, c, d) = \sum_m(2, 3, 7, 9, 11, 13) + \sum_d(1, 10, 15)$
- Step 1: Group minterms to find prime implicants by applying $xy + xy' = x$.
- Step 2: Select a minimum set of prime implicants (minimum # of literals) to implement the original function.
- Exponential-time exact algorithm, huge amounts of memory!



Quine-McCluskey Exercises

http://www.cs.columbia.edu/~cs6861/handouts/quine-mccluskey-handout.pdf
(Worthwhile. This one is better. We are using this.)

https://www.tutorialspoint.com/digital_circuits/digital_circuits_quine_mccluskey_tabular_method.htm

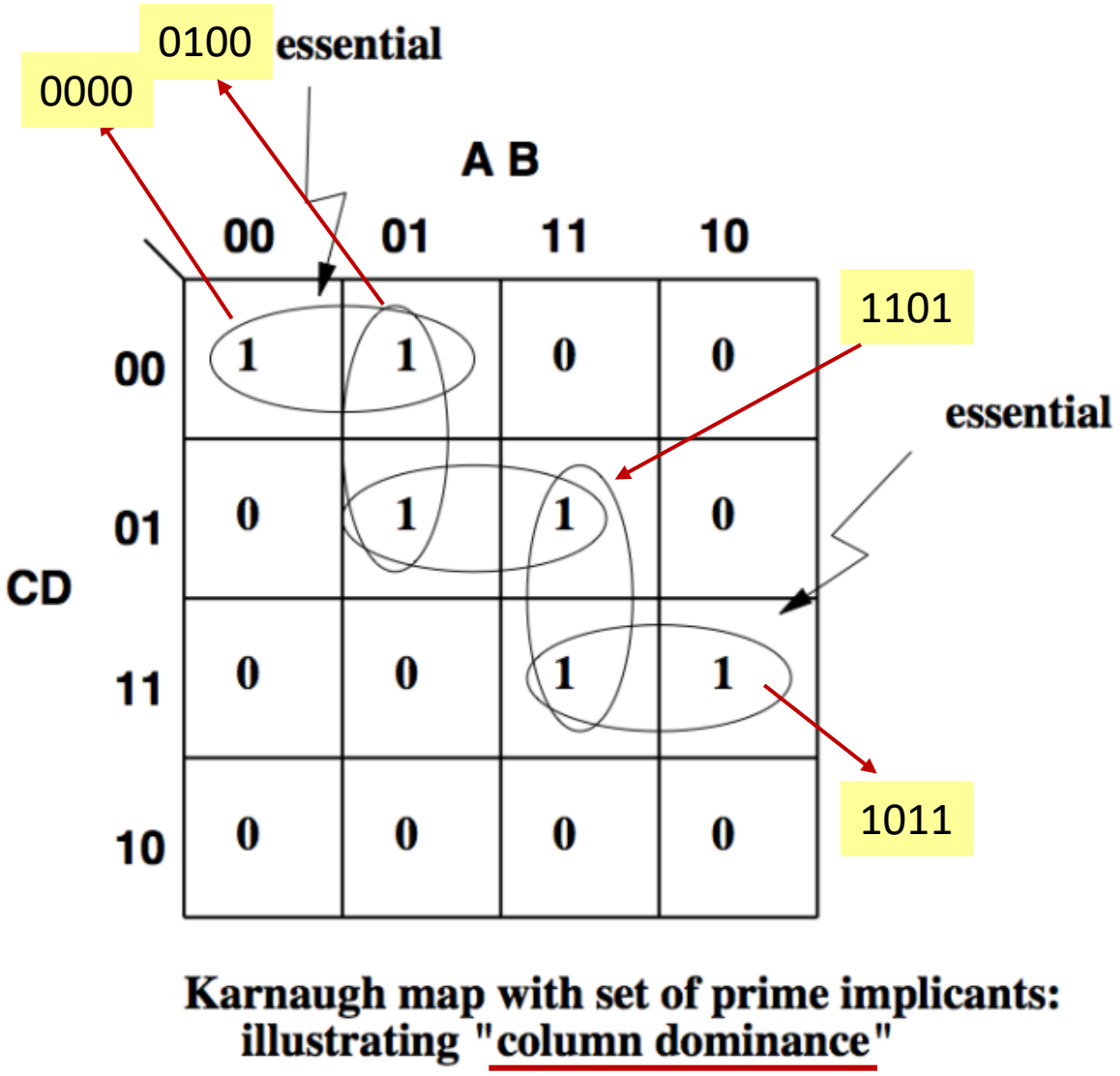
https://atozmath.com/example/KMap.aspx?he=e&q=quine

https://www.youtube.com/watch?v=6JzBPVr7yCM (by Prof. 王俊堯)

https://www.youtube.com/watch?v=l1jgq0R5EwQ

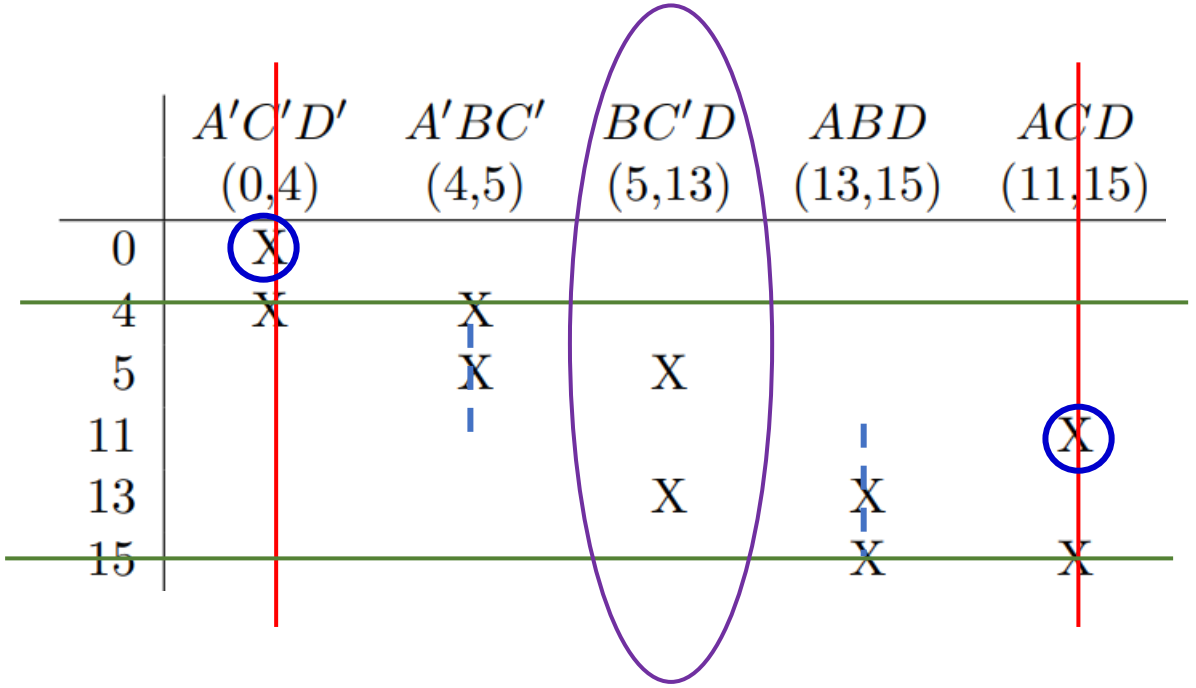
https://zh.wikipedia.org/zh-tw/%E5%A5%8E%E5%9B%A0%EF%BC%8D%E9%BA%A6%E5%85%8B%E6%8B%89%E6%96%AF%E5%9F%BA%E7%AE%97%E6%B3%95

From Prof. Nowick's handout



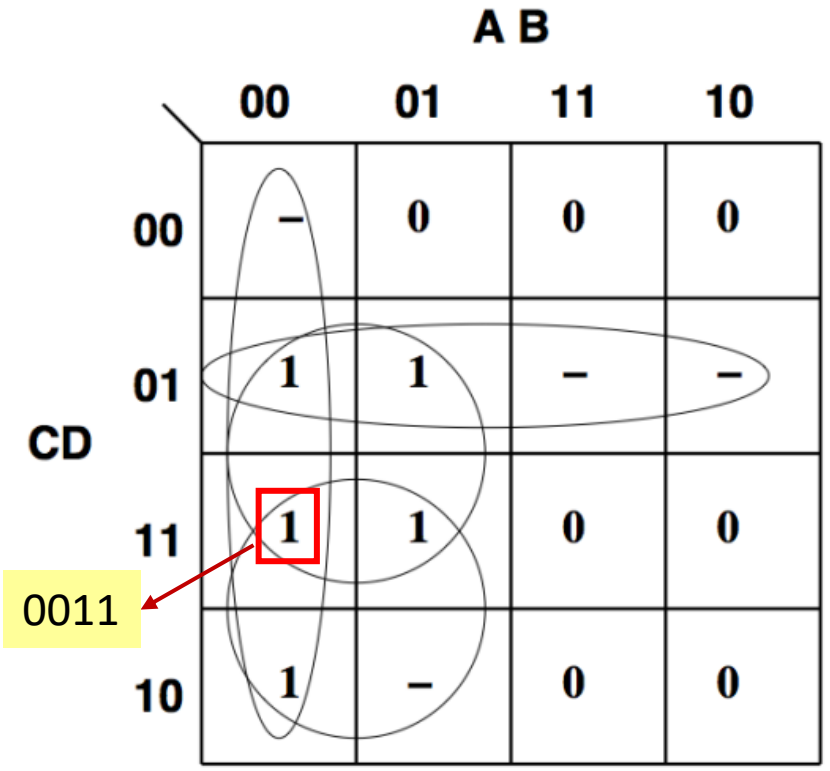
Karnaugh map with set of prime implicants: illustrating "column dominance"

○ : essential (looking for a row with only one X; Then, that column (prime implicant) is needed.



Column dominance

F=A'C'D' + ACD + BC'D

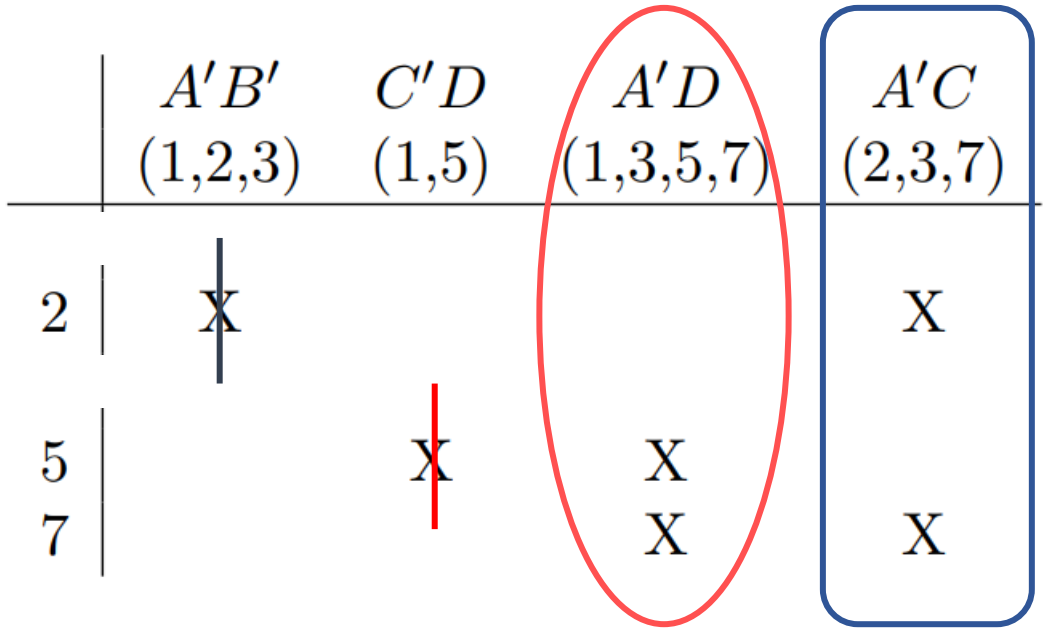


Karnaugh map with set of prime implicants: illustrating "row dominance"

no essential prime implicants

	$A'B'$ (1,2,3)	$C'D$ (1,5)	$A'D$ (1,3,5,7)	$A'C$ (2,3,7)
1	X	X	X	
2	X			X
3	X		X	X
5		X	X	
7			X	X

Any prime implicant which contains row 2 also contains row 3.
Any prime implicant which contains row 7 also contains row 3.
Row 3 row dominates.
The **reverse** of column dominance: we cross out the dominating (larger) row. (As long as we cover row 2, we cover 3)
Every product which covers row-2 also covers row-3. If some product covers row-2, row-3 is guaranteed to be covered.
After crossing out row-3, row-1 dominates row-5.



F = A’D + A’C

F(A, B, C, D) = Σm(0, 2, 5, 6, 7, 8, 10, 12, 13, 14, 15)

○ distinguished row

Step 1: Generate Prime Implicants

Column I			*Column II*			*Column III*	
0	0000	√	(0,2)	00-0	√	(0,2,8,10)	-0-0
2	0010	√	(0,8)	-000	√	~~(0,8,2,10)~~	~~-0-0~~
8	1000	√	(2,6)	0-10	√	(2,6,10,14)	–10
5	0101	√	(2,10)	-010	√	~~(2,10,6,14)~~	~~–10~~
6	0110	√	(8,10)	10-0	√	(8,10,12,14)	1–0
10	1010	√	(8,12)	1-00	√	~~(8,12,10,14)~~	~~1–0~~
12	1100	√	(5,7)	01-1	√	(5,7,13,15)	-1-1
7	0111	√	(5,13)	-101	√	~~(5,13,7,15)~~	~~-1-1~~
13	1101	√	(6,7)	011-	√	(6,7,14,15)	-11-
14	1110	√	(6,14)	-110	√	~~(6,14,7,15)~~	~~-11-~~
15	1111	√	(10,14)	1-10	√	(12,13,14,15)	11–
			(12,13)	110-	√	~~(12,14,13,15)~~	~~11–~~
			(12,14)	11-0	√		
			(7,15)	-111	√		
			(13,15)	11-1	√		
			(14,15)	111-	√		

3 1's

Step 2: Construct Prime Implicant Table.

	$B'D'$ (0,2,8,10)	CD' (2,6,10,14)	BD (5,7,13,15)	BC (6,7,14,15)	AD' (8,10,12,14)	AB (12,13,14,15)
0	X					
2	X	X				
5			X			
6		X		X		
7			X	X		
8	X				X	
10	X	X			X	
12					X	X
13			X			X
14		X		X	X	X
15			X	X		X

Column: B'D', BD are essential prime implicant;
Then, cross out the rows with X in these two columns

(ii) Row Dominance:

Row 14 dominates both row 6 and row 12.

row 14 can be crossed out

	CD' (2,6,10,14)	BC (6,7,14,15)	AD' (8,10,12,14)	AB (12,13,14,15)
6	X	X		
12			X	X
14	X	X	X	X

(iii) Column Dominance

	CD' (2,6,10,14)	BC (6,7,14,15)	AD' (8,10,12,14)	AB (12,13,14,15)
6	X	X		
12			X	X

Such columns are said to co-dominate each other.

dominated column is crossed out.

Iteration #2.

(i) Remove Secondary Essential Prime Implicants

	$CD'(**)$ (2,6,10,14)	$AD'(**)$ (8,10,12,14)
(○)6	X	
(○)12		X

** indicates a secondary essential prime implicant

○ indicates a distinguished row

the minimum-cost solution consists of the primary and secondary essential prime implicants

$$F = B'D' + BD + CD' + AD'$$

- Welcome to go through example #2, #3

Example #2:

$$F(A, B, C, D) = \Sigma m(0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)$$

Example #3: Don't-Cares (Roth example, pp. 163-4).

$$F(A, B, C, D) = \Sigma m(2, 3, 7, 9, 11, 13) + \Sigma d(1, 10, 15)$$

- Pondering how this kind of algorithms is implemented?
  - GitHub has some programs

More Examples

- Welcome to go through example #2, #3

Example #2:

$$F(A, B, C, D) = \Sigma m(0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)$$

- Generate prime implicants

Step 2: Construct Prime Implicant Table.

	$A'D'$	$B'D'$	$C'D'$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
0	X	X	X						
2	X	X		X	X				
3				X	X				
4	X		X			X	X		
5						X	X		
6	X			X		X			
7				X		X			
8		X	X					X	X
9								X	X
10		X			X			X	
11					X			X	
12			X				X		X
13							X		X

No primary **essential** prime implicants, i.e., no row of single minterm. Each row is covered by at least 2 products. No column dominance.

(ii) Row Dominance

	$A'D'$	$B'D'$	$C'D'$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
0	X	X	X						
2	X	X		X	X				
3				X	X				
4	X		X			X	X		
5						X	X		
6	X			X		X			
7				X		X			
8		X	X					X	X
9								X	X
10		X			X			X	
11					X			X	
12			X				X		X
13							X		X

Row 2 dominates 3
Row 4 dominates 5
6 dominates 7,
8 dominates 9,
10 dominates 11
12 dominates 13

Remove dominating rows

(iii) Column Dominance

	$A'D'$	$B'D'$	$C'D'$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
0	X	X	X						
3				X	X				
5						X	X		
7				X		X			
9								X	X
11					X			X	
13							X		X

(i) Remove Secondary Essential Prime Implicants

	$A'D'(**)$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
(○)0	X						
3		X	X				
5				X	X		
7		X		X			
9						X	X
11			X			X	
13					X		X

** indicates a secondary essential prime implicant

○ indicates a distinguished row

Product A'D' is a secondary essential prime implicant; it is removed from the table

(ii) Row Dominance

No further row dominance is possible.

(iii) Column Dominance

No further column dominance is possible.

Step 4: Solve Prime Implicant Table.

	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
3	X	X				
5			X	X		
7	X		X			
9					X	X
11		X			X	
13				X		X

The remaining covering problem is called a **cyclic covering** problem. A solution can be obtained using one of two methods: (i) Petrick's method or (ii) the branching method.

Quick Overview Of Petrick's Method

Step 4: Solve Prime Implicant Table.

	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
3	X	X				
5			X	X		
7	X		X			
9					X	X
11		X			X	
13				X		X

Using Boolean expression

- p1=A'C, p2=B'C, p3=A'B, p4=BC', p5=AB', p6=AC'
- Boolean variable **P1** is true, when prime implicant p1 (lower-case p1) is included in the solution.
- **P1** is called Boolean proposition (true/false). P1=1 means "I select prime implicant p1 in the cover."

- To cover row-3, we need (**P1+P2**)
- To cover row-7, we need (**P1+P3**)
- The covering requirements can be captured by the Boolean equation: $P = (P_1 + P_2)(P_3 + P_4)(P_1 + P_3)(P_5 + P_6)(P_2 + P_5)(P_4 + P_6)$

$$P = (P_1 + P_2)(P_3 + P_4)(P_1 + P_3)(P_5 + P_6)(P_2 + P_5)(P_4 + P_6)$$

$$P = P_1P_4P_5 + P_1P_3P_5P_6 + P_2P_3P_4P_5 + P_2P_3P_5P_6 + P_1P_2P_4P_6 + P_1P_2P_3P_6 + P_2P_3P_4P_6 + P_2P_3P_6$$

- Looking for the product terms with **minimum # of P variables**
- There are: $P_1P_4P_5$ and $P_2P_3P_6$ p1=A'C, p2=B'C, p3=A'B, p4=BC', p5=AB', p6=AC'
- Both solutions have a minimal number of prime implicants, so either can be used; then, add essential one A'D' to get

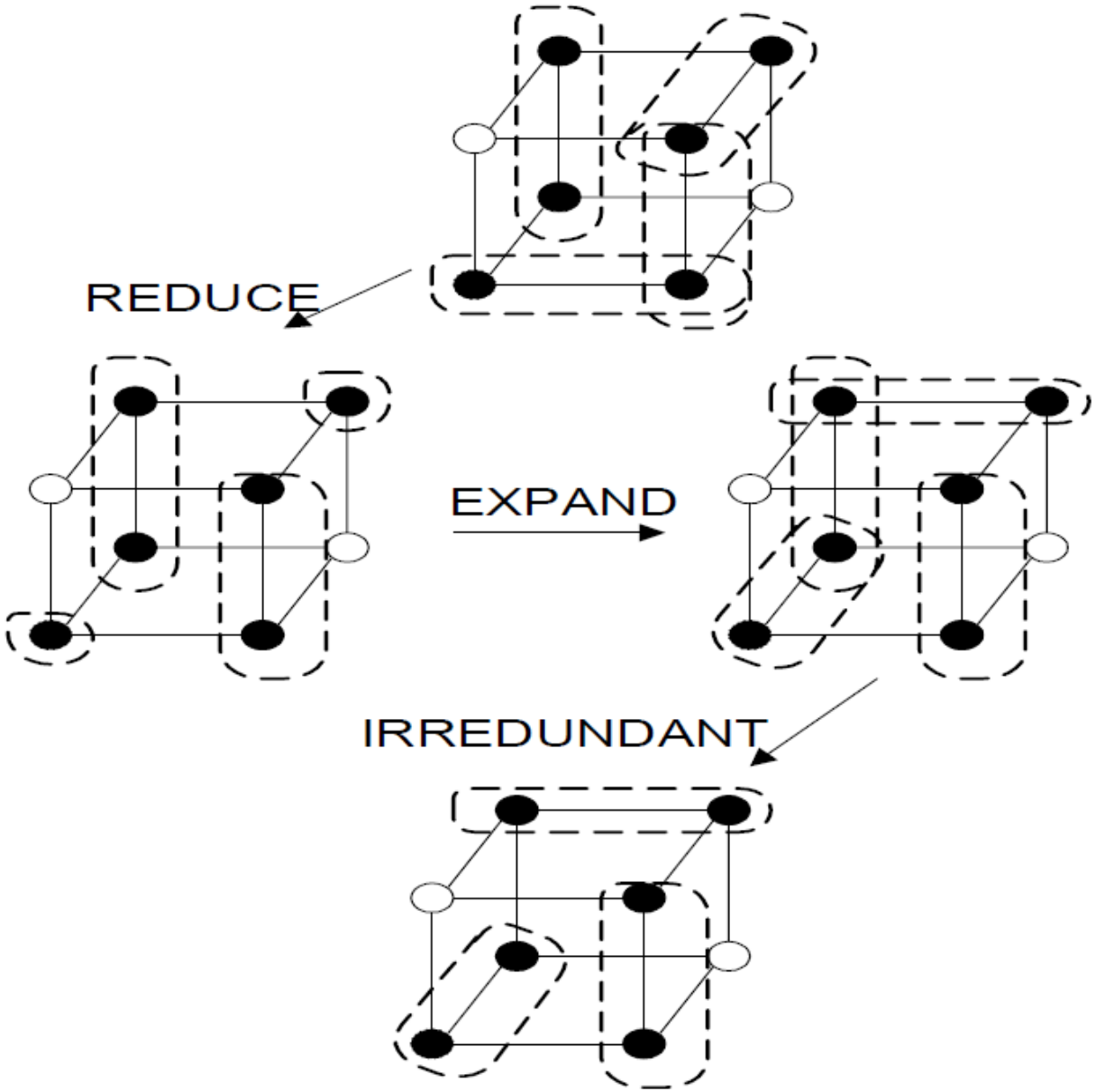
$$F = A'D' + A'C + BC' + AB'$$
$$F = A'D' + B'C + A'B + AC'$$

Heuristic Optimization

- Generation of ***all*** prime implicants is impractical
  - # of prime implicants for a function with *n* variables is exponential
- Finding an *exact* minimum cover is NP-hard
- Heuristic method: avoid generation of all prime implicants
- Procedure
  - A minterm of the ON set is selected and expanded until it becomes a prime implicant
  - The prime implicant is put in the final cover, and all minterms covered by this prime implicant are removed
  - Iterated until all minterms of the ON set are covered
- "Espresso" developed by UC Berkeley
  - The kernel of modern synthesis tools

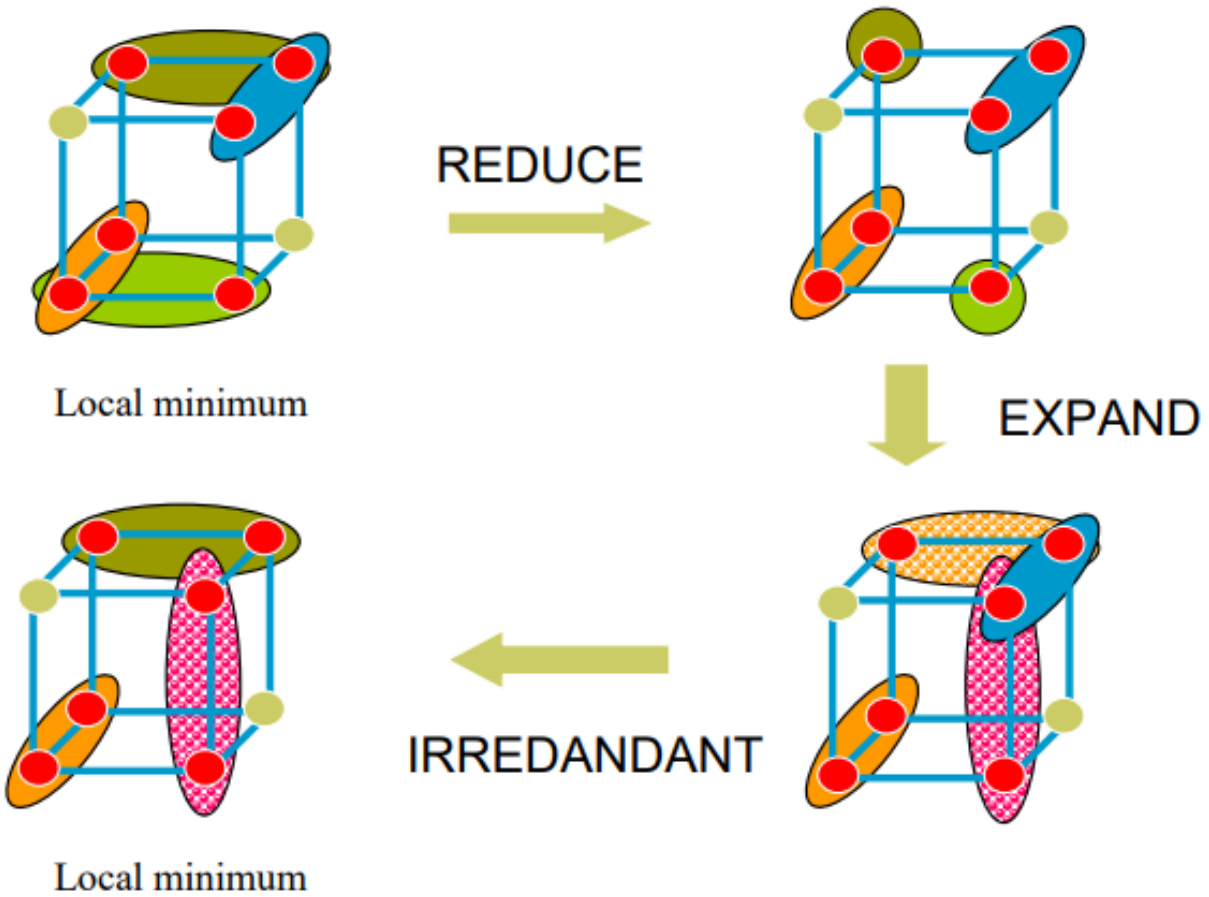
https://github.com/classabbyamp/espresso-logic
A modern (2017) compilable re-host of the Espresso heuristic logic minimizer.
The original source code comes from the University of California, Berkeley.

Illustration of Espresso



Adapted from: Prof. Y.-W. Chang's slides

Two-Level Logic Minimization
ESPRESSO

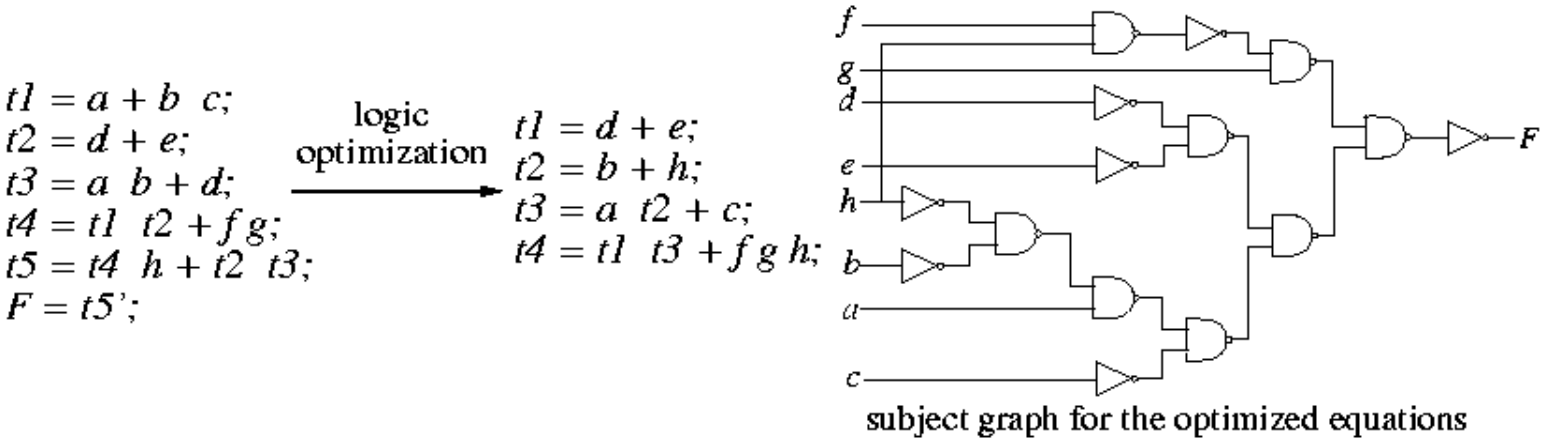


So Far We Had Covered

- Brief domain translation
- Binary decision tree → Binary decision diagram BDD (Reduced)
  - CUDD → welcome to provide more logic equations and vary variable ordering
- Two-level logic optimization (Quine-McClusky)
  - Essential, column dominance, row dominance, Petrick's Boolean Proposition

Technology-Independent Logic Optimization

- **Two-level:** minimize the # of product terms.
  - $F = \bar{x_1}\bar{x_2}\bar{x_3} + \bar{x_1}\bar{x_2}x_3 + x_1\bar{x_2}\bar{x_3} + x_1\bar{x_2}x_3 + x_1x_2\bar{x_3} \Rightarrow F = \bar{x_2} + x_1\bar{x_3}$.
- **Multi-level:** minimize the #'s of literals, variables.
  - E.g., equations are optimized using a smaller number of literals.



subject graph for the optimized equations

- Methods/CAD tools: The Quine-McCluskey method (exponential-time exact algorithm), Espresso (heuristics for two-level logic), MIS (heuristics for multi-level logic), Synopsys, etc.

https://en.wikipedia.org/wiki/Espresso_heuristic_logic_minimizer

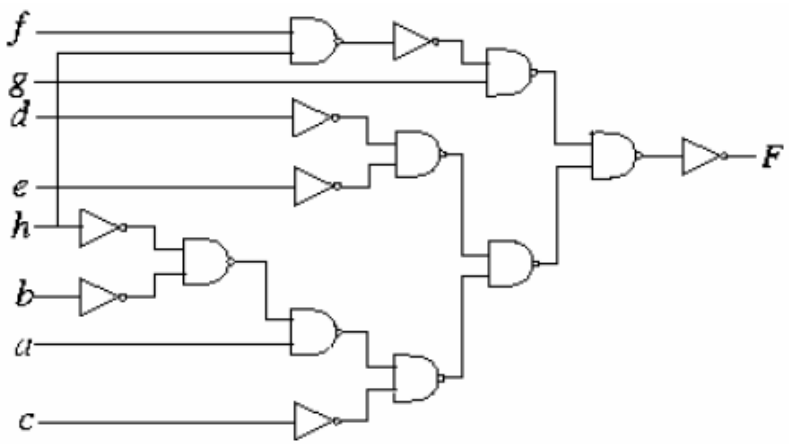
Multi-Level Logic Optimization

- Translate a combinational circuit to meet performance or area constraints
  - Common factors extraction
  - Common expression resubsitution
  - minimize the #'s of literals / variables, e.g., equations are optimized using a smaller number of literals
- In commercial use for several years: Synopsys, MIS
- Example:

$t1 = a + b\ \ c;$
$t2 = d + e;$
$t3 = a\ \ b + d;$
$t4 = t1\ \ t2 + f\ g;$
$t5 = t4\ \ h + t2\ \ t3;$
$F = t5';$

logic optimization →

$t1 = d + e;$
$t2 = b + h;$
$t3 = a\ \ t2 + c;$
$t4 = t1\ \ t3 + f\ g\ h;$

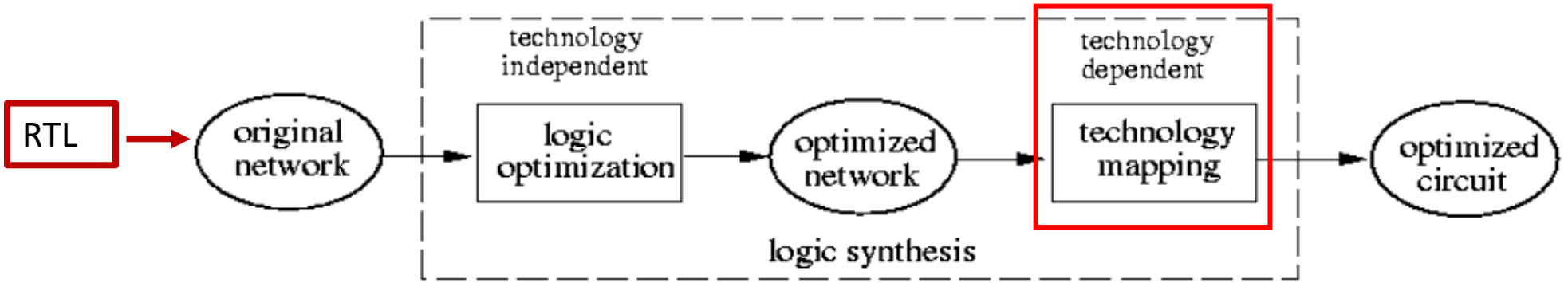


subject graph for the optimized equations

https://people.eecs.berkeley.edu/~alanmi/abc/

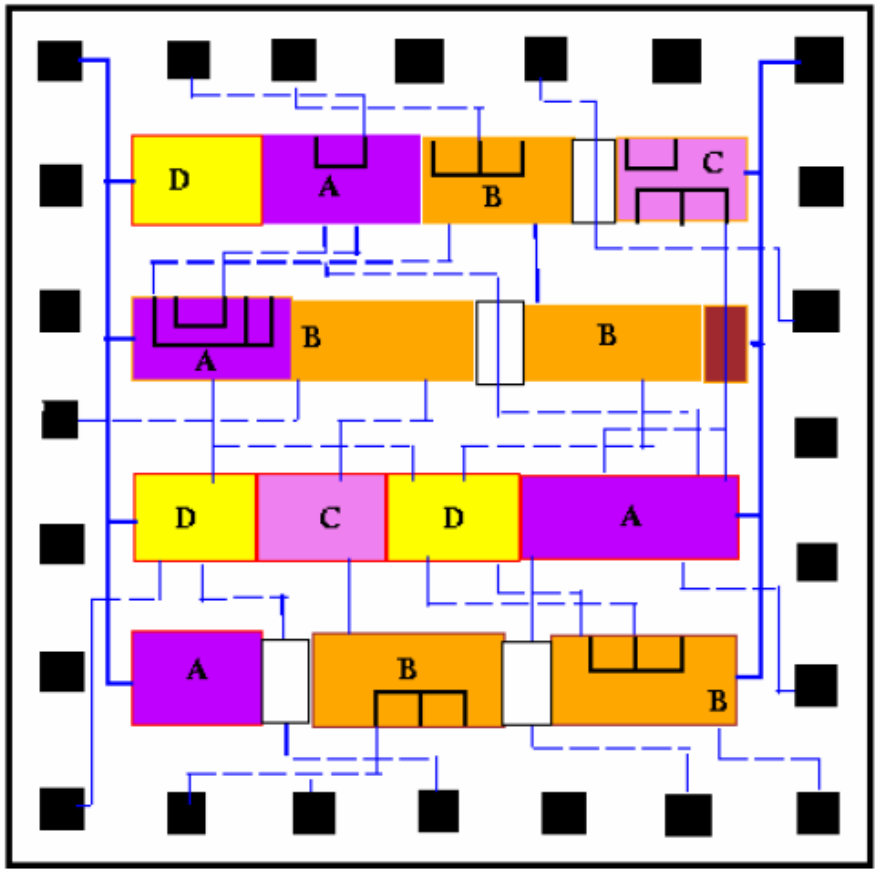
A System for Sequential Synthesis and Verification
Berkeley Logic Synthesis and Verification Group

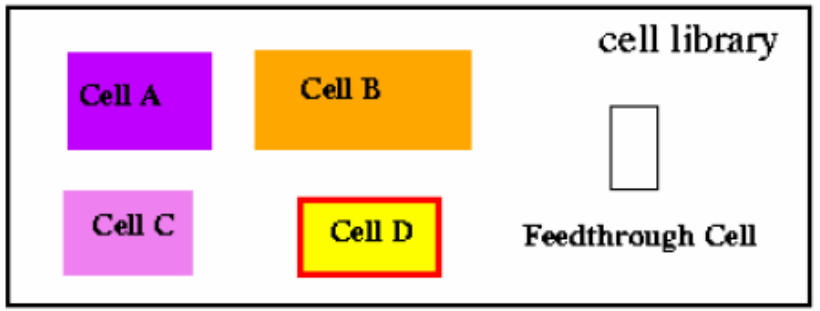
Technology Mapping



- **Library-based technology mapping:** standard cell design.
  - Map a function to a limited set of pre-designed cells
- **Lookup table-based technology mapping:** Altera, Lucent, Xilinx CPLDs/FPGAs, etc.
  - Each lookup table (LUT) can implement a very large number of functions (e.g., all functions with 4 inputs and 1 output)
- **Multiplexer-based technology mapping:** Actel FPGAs, etc.
  - Logic modules are constructed with multiplexers.

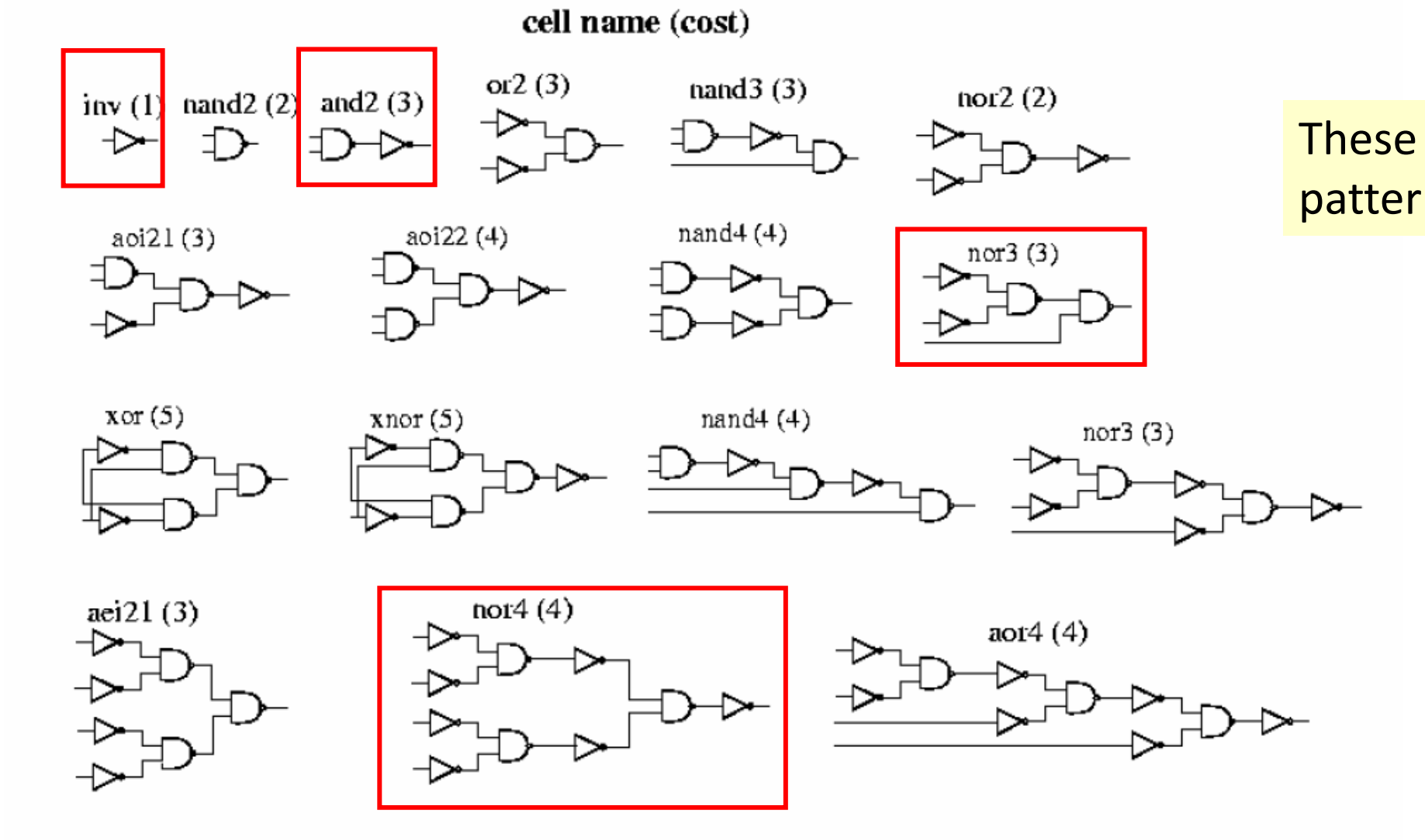
Standard Cell Revisited





Adapted from: Prof. Y.-W. Chang's slides

Pattern Graphs for an Example Library

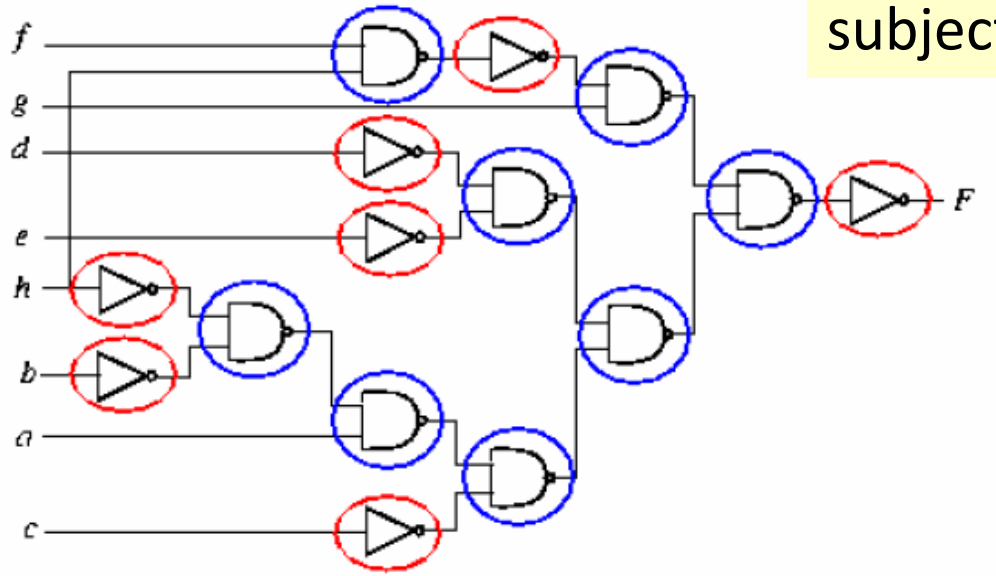


These are called pattern graph

Trivial Covering

- Mapped into 2-input NANDs and 1-input inverters.
- 8 2-input NAND-gates and 7 inverters for an area cost of 23.
- Best covering?

$$t1 = d + e;$$
$$t2 = b + h;$$
$$t3 = a\ \ t2 + c;$$
$$t4 = t1\ \ t3 + f\ g\ h;$$



Circuit is called subject graph

Technology Mapping

- **Technology Mapping:** The optimization problem of finding a minimum cost covering of the subject graph by choosing from the collection of pattern graphs for all gates in the library.
- A **cover** is a collection of pattern graphs such that every node of the subject graph is contained in one (or more) of the pattern graphs.
- The cover is further constrained so that each input required by a pattern graph is actually an output of some other pattern graph.

Ok to be duplicated

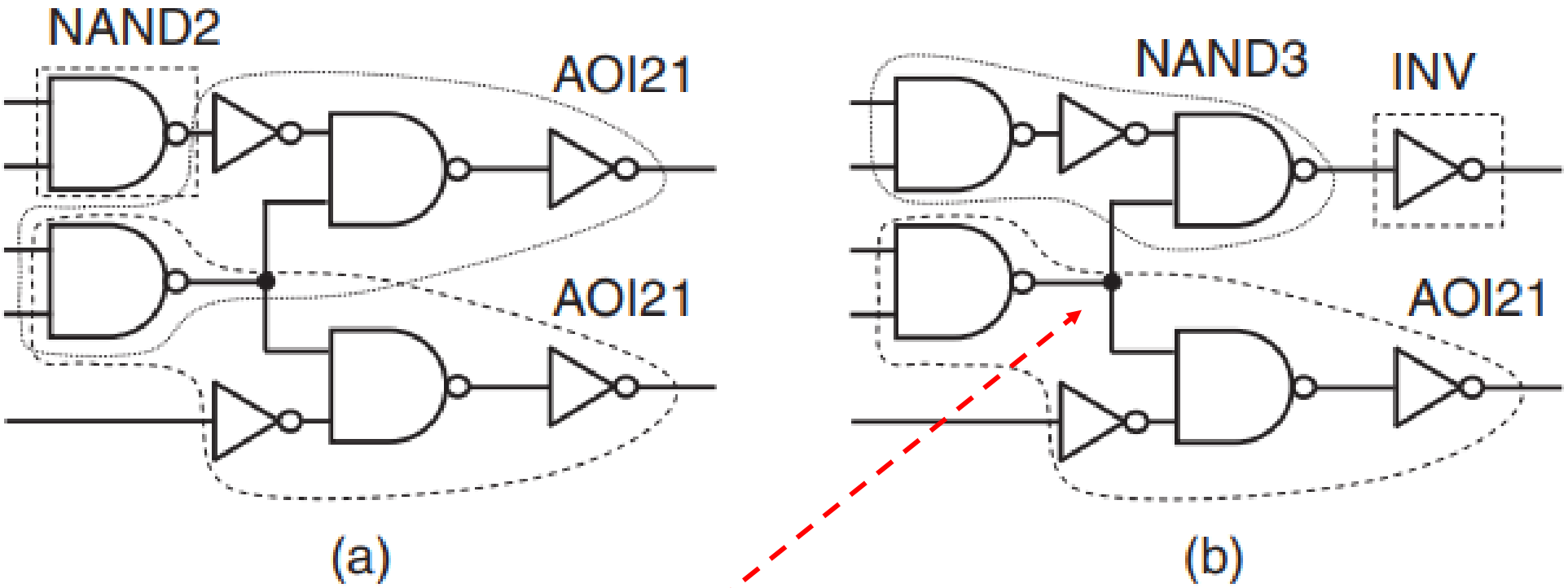
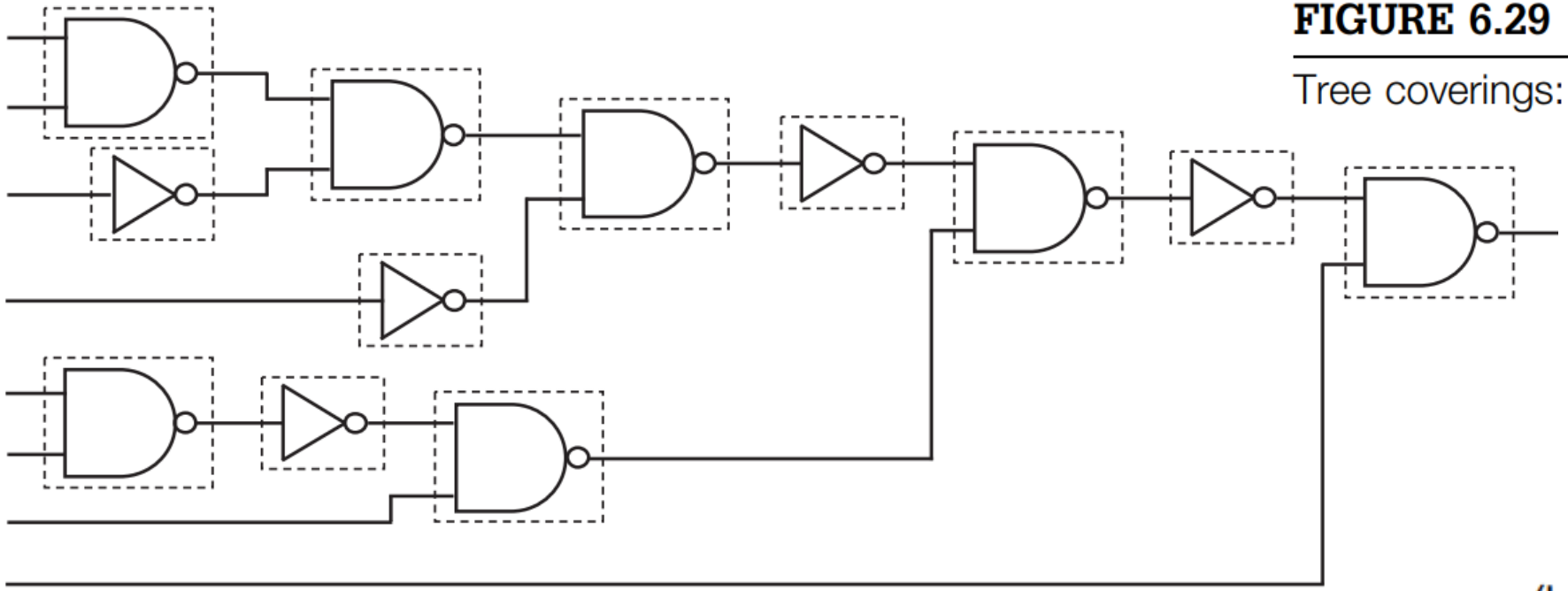


FIGURE 6.27

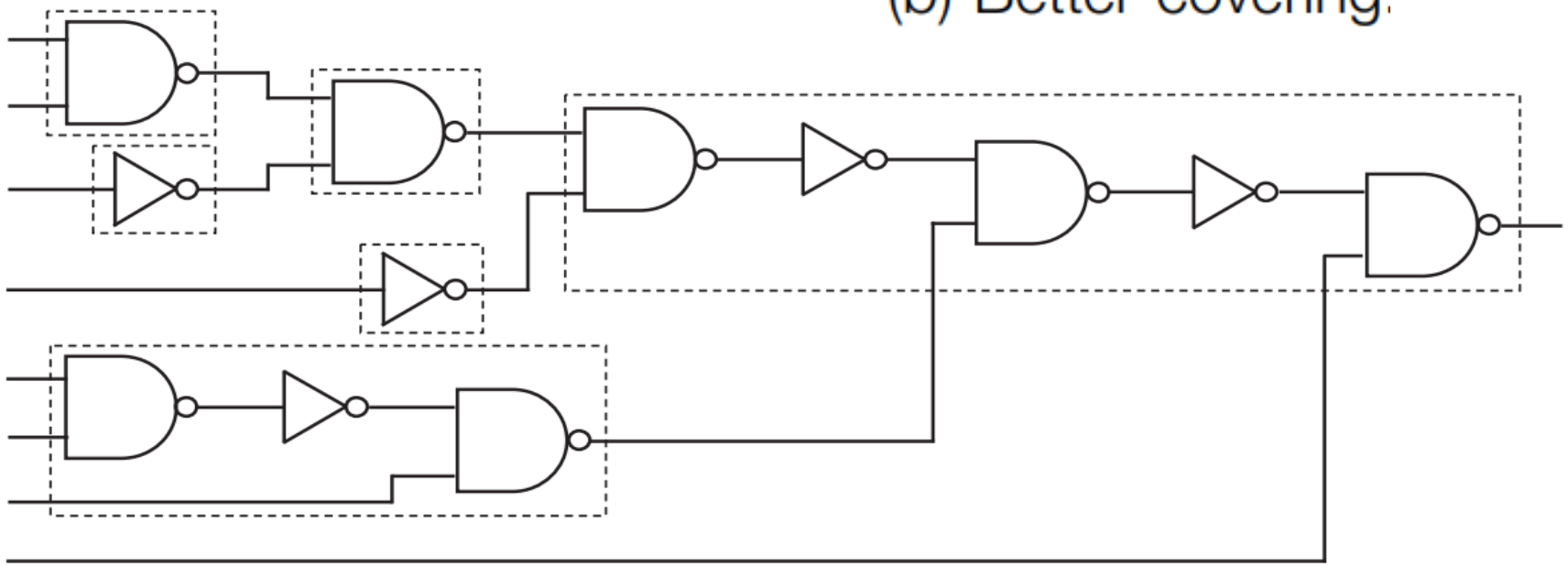
Graph coverings: (a) legal and (b) illegal.

FIGURE 6.29

Tree coverings: (a) Trivial covering.

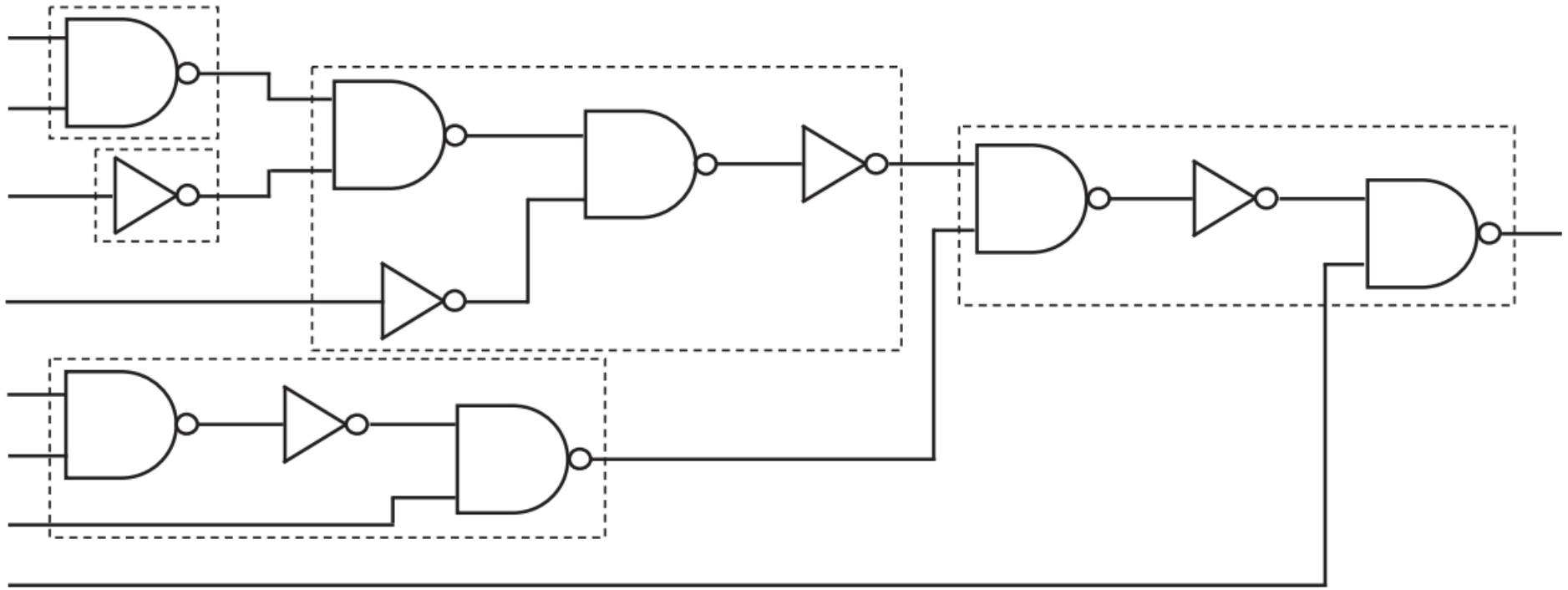


(b) Better covering.



(b)

(c) Optimum covering.

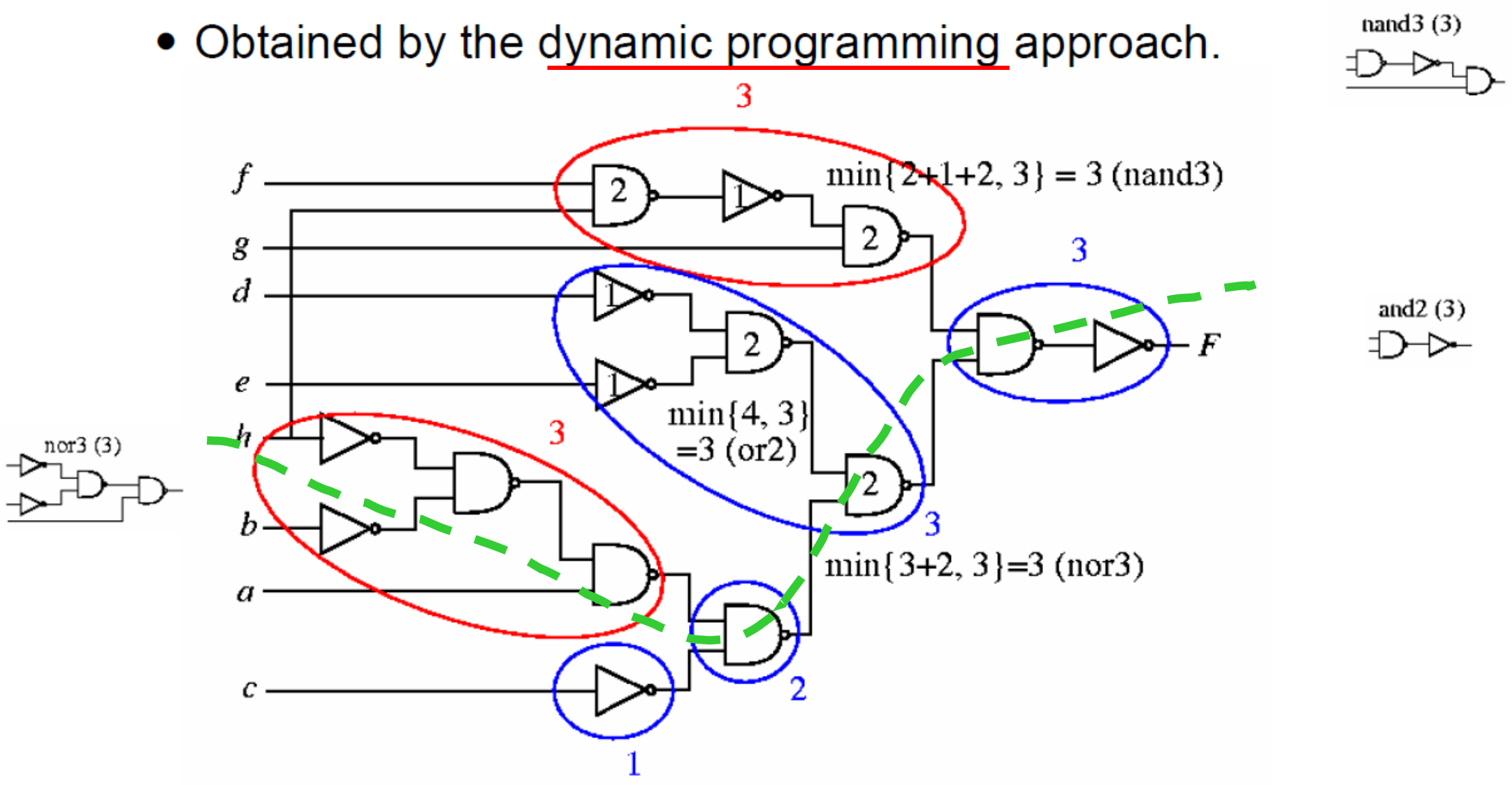


(c)

How to obtain this?

Best Covering

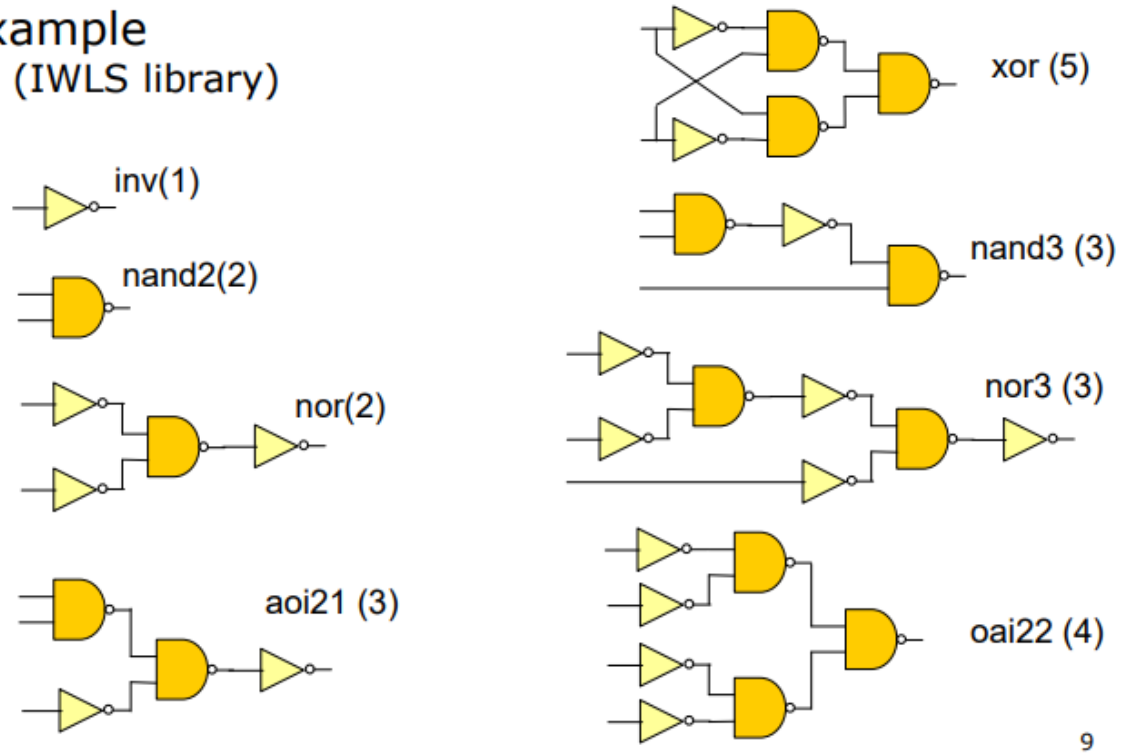
- A best covering with an area of 15.
- Obtained by the dynamic programming approach.



The optimization problem of technology mapping can now be stated as:
Find a minimum cost covering of the subject DAG by the pattern DAGs

Pattern Graphs

- Example
  - (IWLS library)

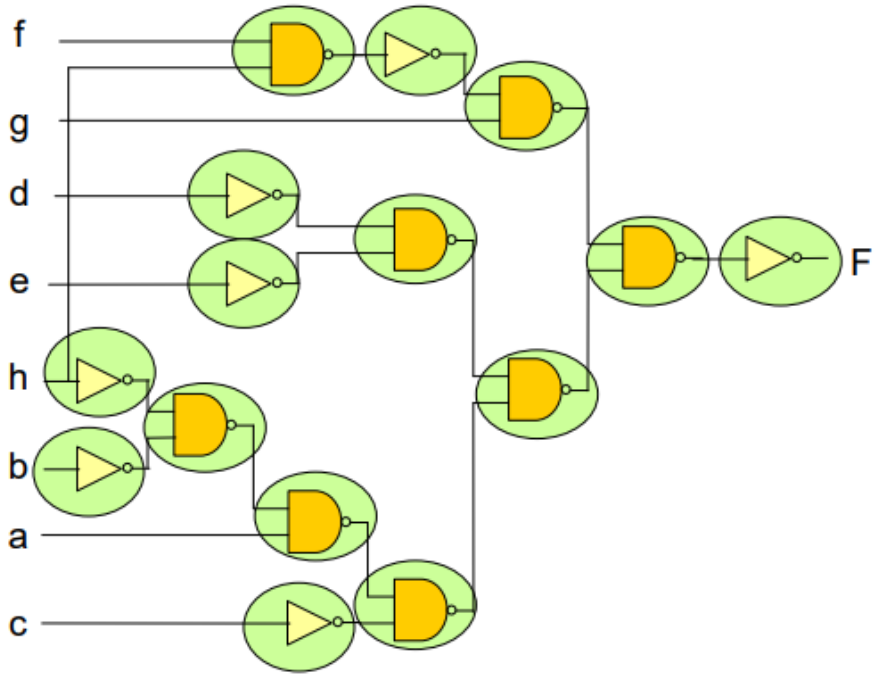


Subject Graph Covering (1)

- Example

$t_1 = d + e$
$t_2 = b + h$
$t_3 = at_2 + c$
$t_4 = t_1t_3 + fgh$
$F = t_4'$

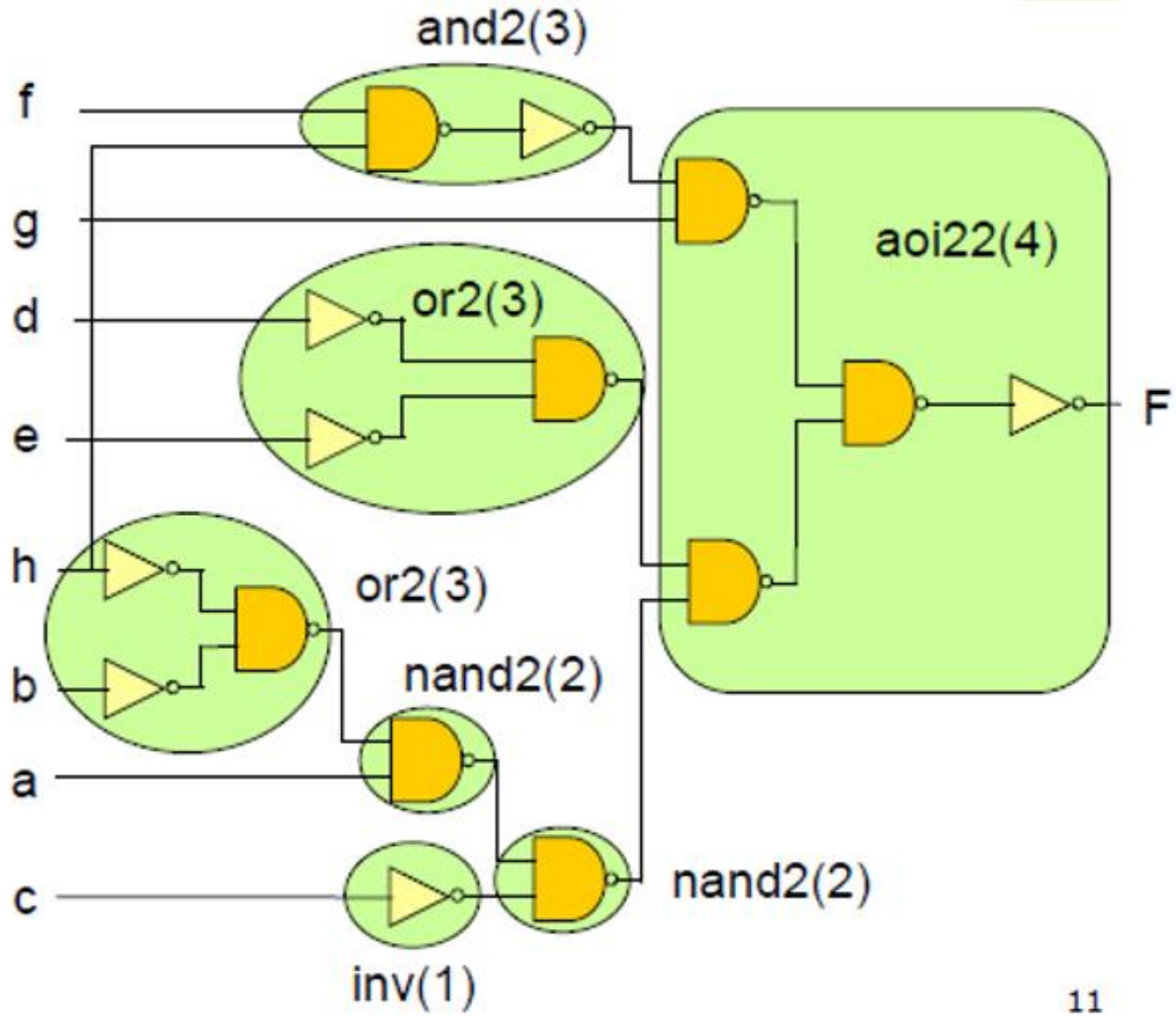
Total cost = 23



Subject Graph Covering (2)

- Example

  $t_1 = d + e$
  $t_2 = b + h$
  $t_3 = at_2 + c$
  $t_4 = t_1t_3 + fgh$
  $F = t_4'$

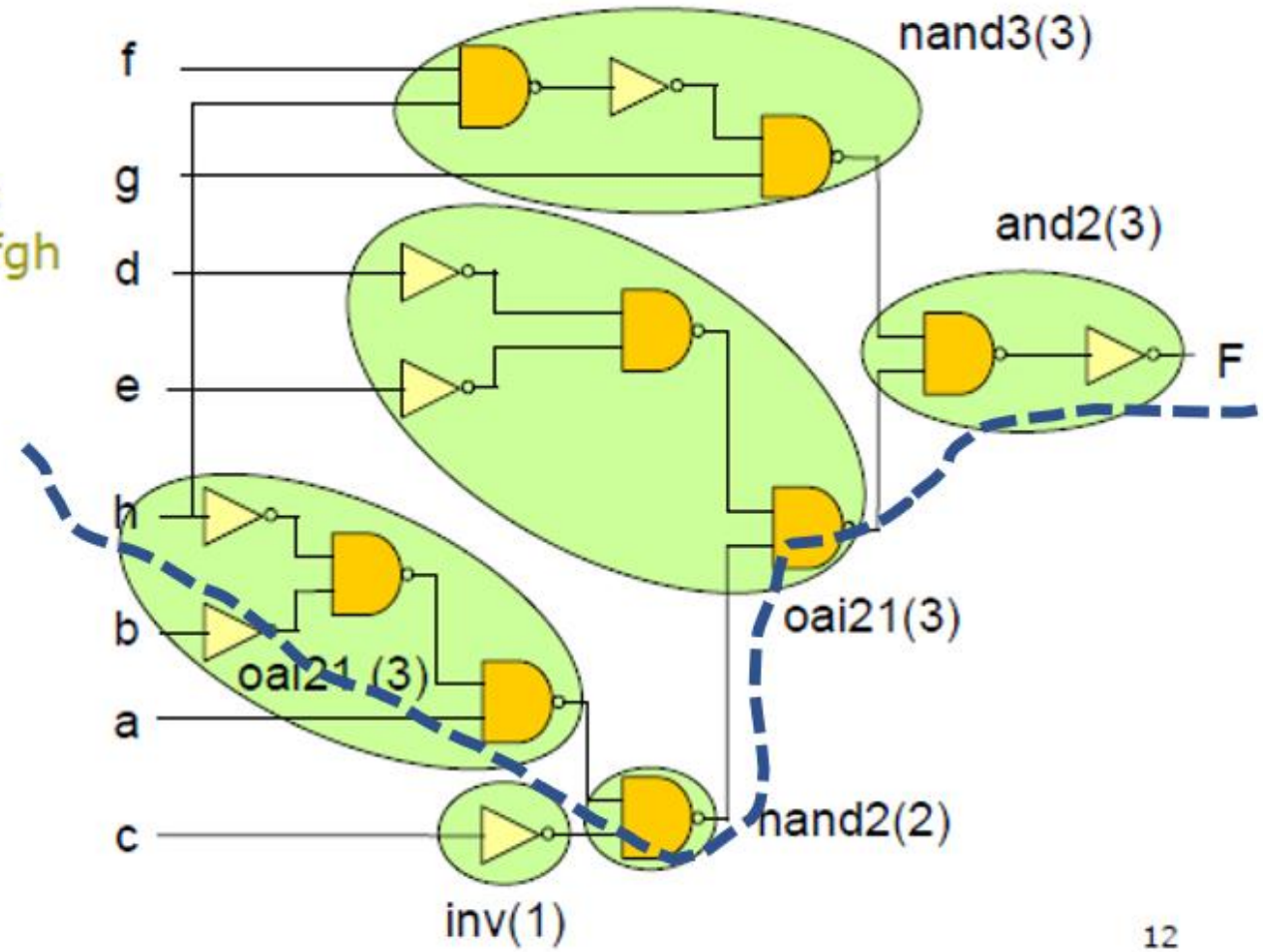


Total cost = 18

Subject Graph Covering (3)

- Example

$t_1 = d + e$
$t_2 = b + h$
$t_3 = at_2 + c$
$t_4 = t_1t_3 + fgh$
$F = t_4'$



Total cost = 15

Going thru 4 “logic gates”

DAG Covering

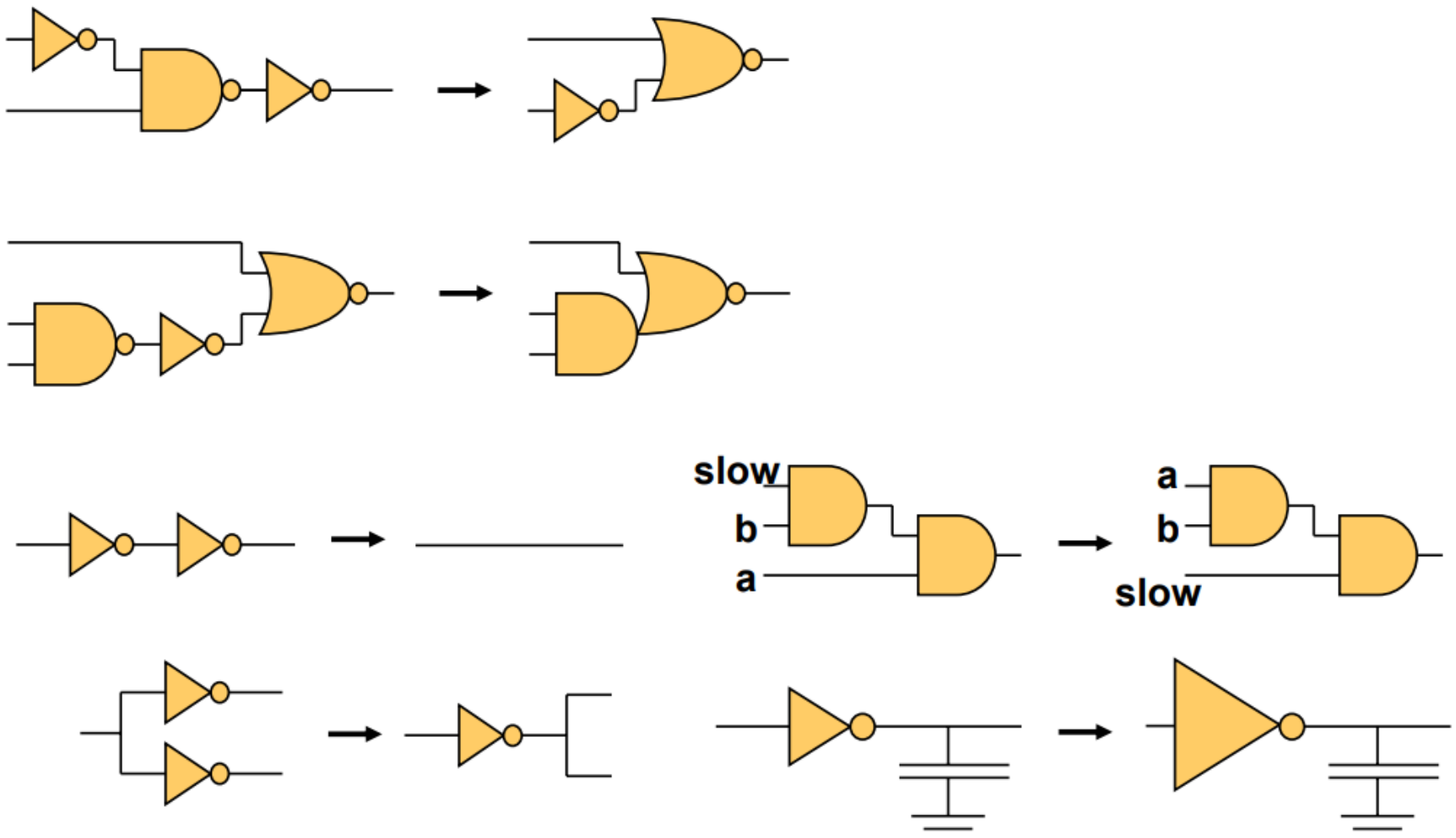
- Input:
  - Logic network after technology independent optimization
  - A library of gates with their costs
- Output:
  - A netlist of gates (from library) which minimizes total cost
- General Approach:
  - Construct a subject DAG (directed acyclic graph) for the network
  - Represent each gate in the target library by pattern DAG's
  - Find an optimal-cost covering of subject DAG using the collection of pattern DAG's

DAG Covering

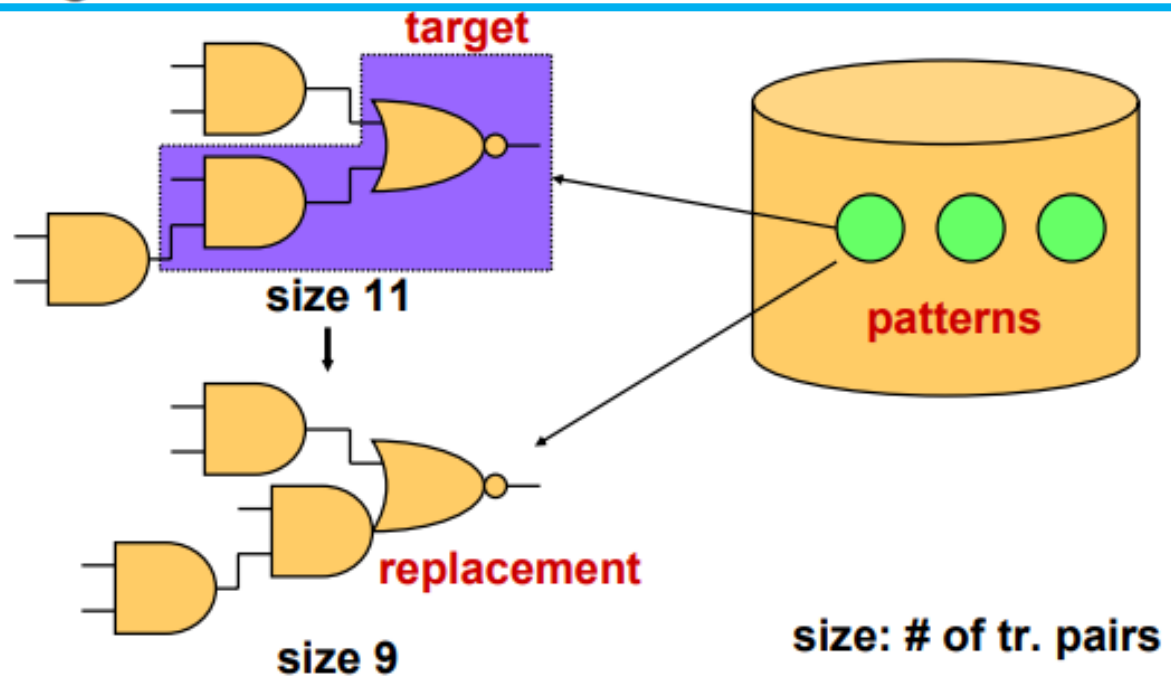
- Complexity
  - NP-hard
  - Remains NP-hard even when the nodes have out-degree ≤ 2
  - If subject DAG and pattern DAG's are trees, efficient algorithms exist

    Dynamic programming

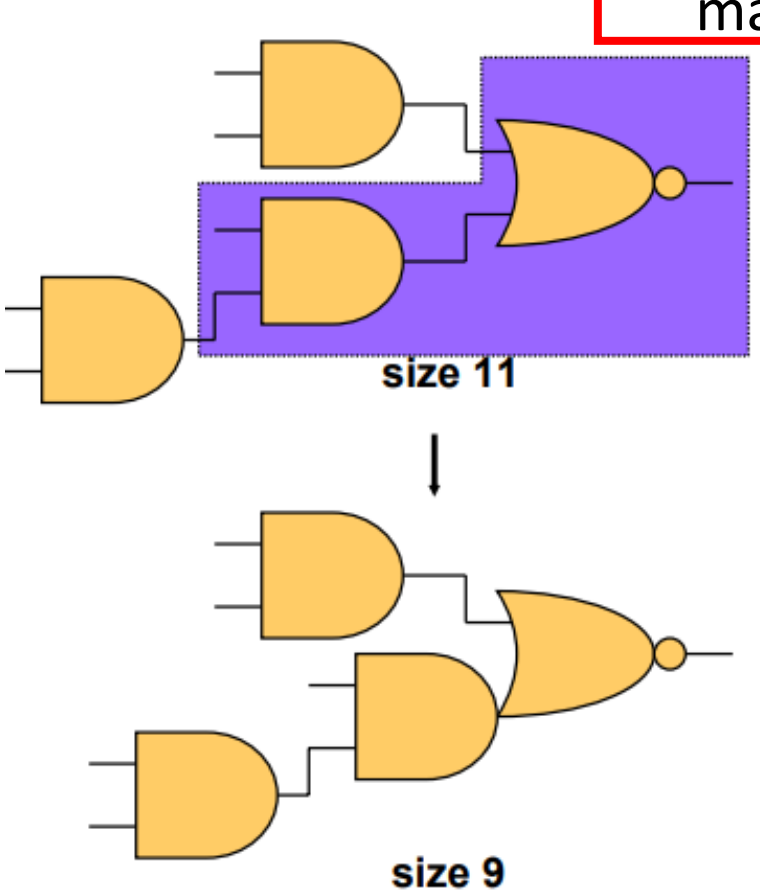
- **Technology Mapping**
  - **A series of local transformations based on rules**
  - **Example rules:**

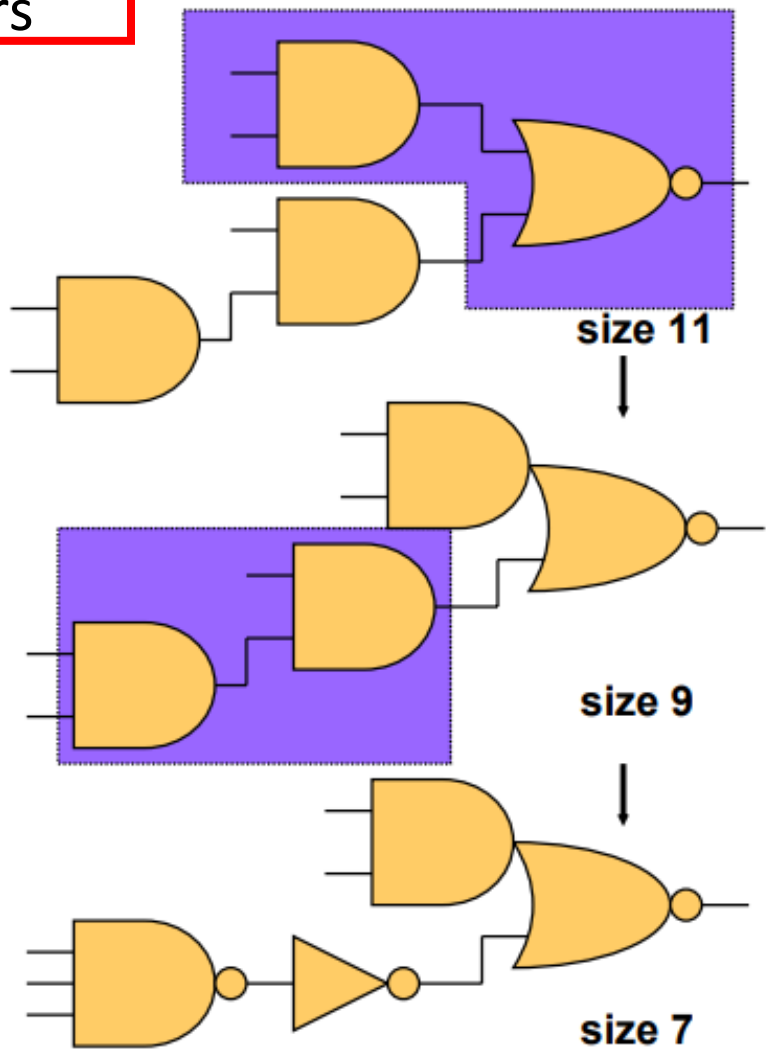


- **Main Operations**
  - Matching: Find a number of rules that apply to the present network
  - Cost function evaluation: For each potential rule application, a cost function is computed to determine the quality of the resulting circuit
  - Selection: Decide which rule should be applied
  - Replacement: Perform network transformation by applying the selected rule



Rule ordering matters





Search Strategies

- State space search problem
- Greedy algorithm

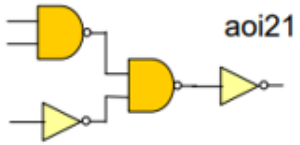
```
Order rules in the knowledge base.
do {
   for each rule R {
      for each gate G {
         if rule R matches at gate G
            if cost improves
               apply_rule (R, G)
      }
   }
} while improving
--> local minimum
```

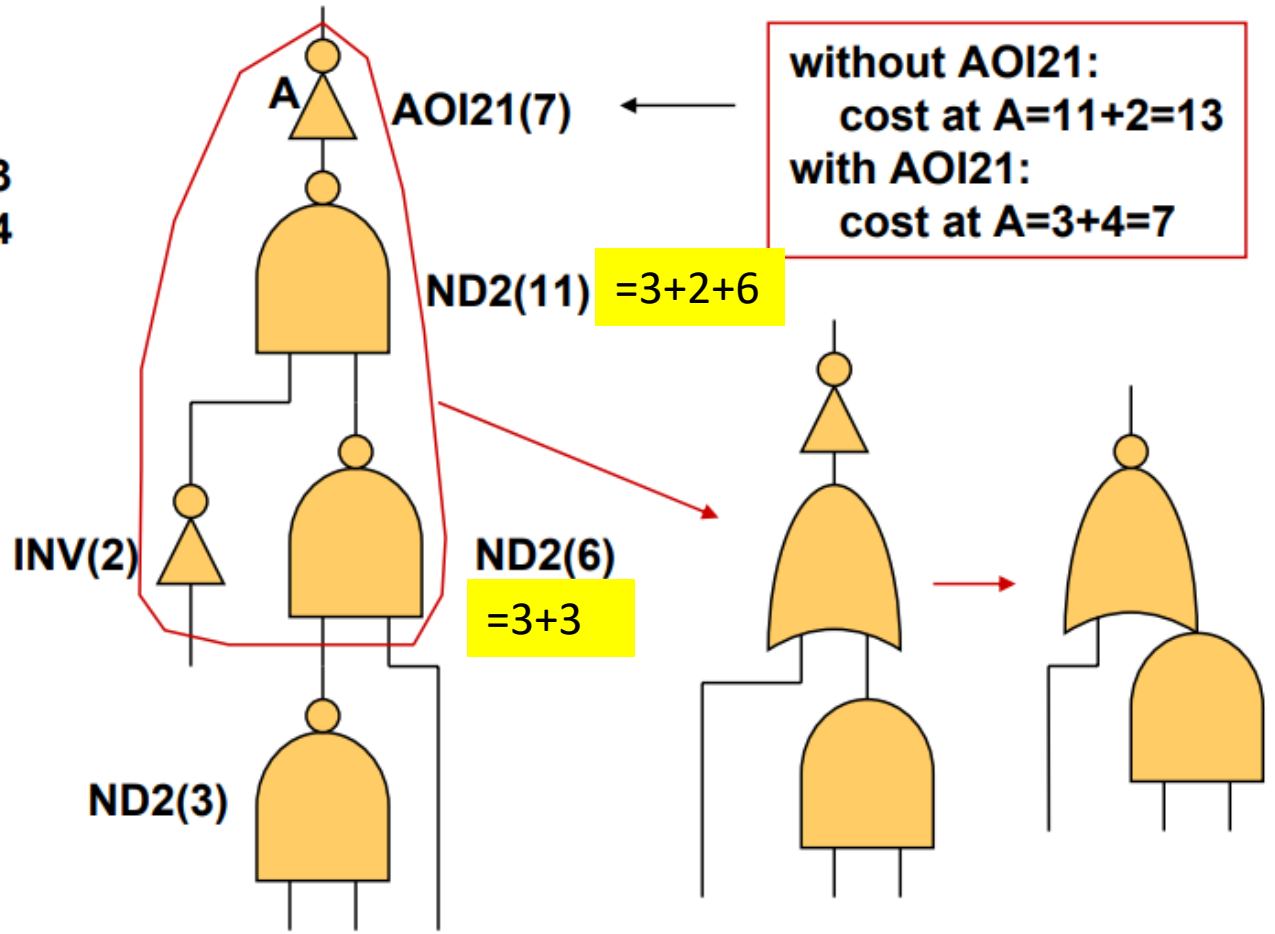
Graph Covering

- DAGON
  - K. Keutzer, "DAGON: technology binding and local optimization by DAG matching," Proc. 24th Design Automation Conference, 1987
- Problem
  - Given
    - Boolean network (represented by a DAG)
    - Library (each cell is a DAG with a cost)
  - Find minimal cost covering of the Boolean network
    - Requires DAG matching (NP-complete)

- **Tree Matching**
  - Compute all matching at each node
  - Select best matching (depth-first)
    - Leaf: Cost of a NAND or a NOT
    - Internal node: Cost of matching tree + cost of sub-trees
  - Example

cost(INV)=2
cost(NAND)=3
cost(AOI21)=4





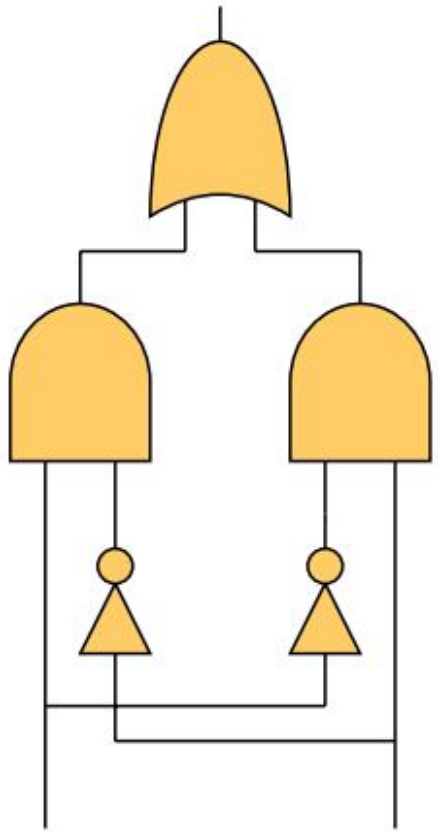
11 is from ND2(11)
2 is from cost-INV(2)

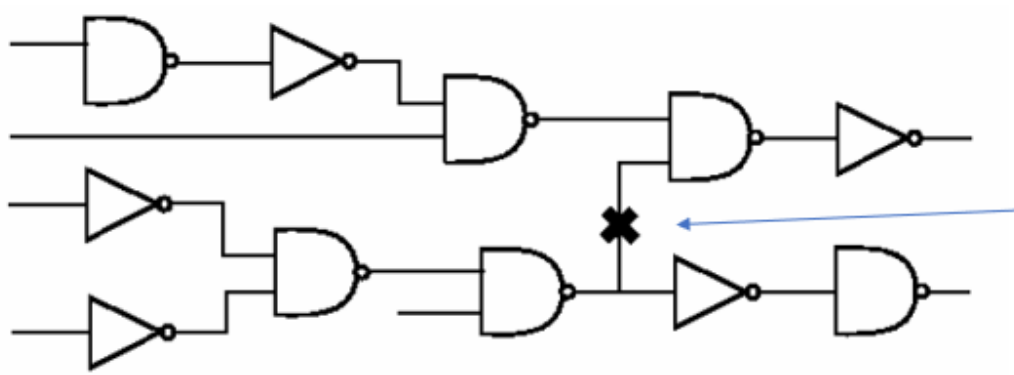
3 is from ND2(3)
4 is cost-AOI21(4)

Pattern matching starts from primary inputs; and follows topological order

How to know the circuit circled is same as circuit at the left?

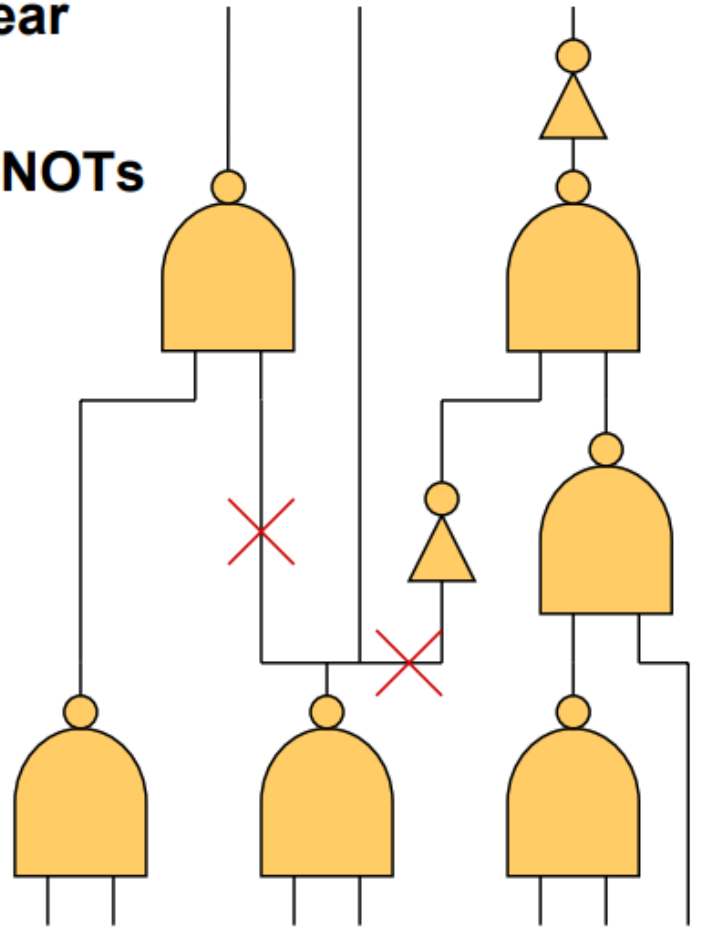
- **Weak Points**
  - **Partitioning network into trees**
    --> local optimum
  - **Representing cells by trees**
    --> No way of representing XORs with a tree





X's marks partition point

- **Simplification**
  - Represent a network by a forest by partitioning DAG
  - Partitioning
    - If a node has fanout greater than one, cut the graph there
    - Gate with fanout>1 becomes a root
  - Complexity of the partitioning: linear
  - Represent library cells by trees
  - Use canonical forms: NANDs and NOTs
  - Match trees by trees
  - Dynamic programming for finding a minimal cost match



X marks partition point

Boolean matching is the problem of determining whether two Boolean functions are functionally equivalent under the permutation and negation of inputs and outputs.

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=263624
https://web.eecs.umich.edu/~imarkov/pubs/book/lsynth-match.pdf

From **Professor Jie-Hong Roland Jiang**:
BDD and/or SAT are often applied to check circuit equivalence, assuming that the input correspondence between the two circuits under checking is known. In technology mapping, in general, we don't know the input correspondence between the two circuits under comparison. In this case, we need to perform the so-called "Boolean matching" to test if the two circuits are equivalent up to input permutation and/or input/output negation). We call such equivalence as "P-equivalence," i.e., equivalent up to input permutation; "NP-equivalence," i.e., equivalent up to input negation and input permutation; or "NPN-equivalence," i.e., equivalent up to input negation, input permutation, and output negation. Indeed, **BDD and SAT** can be used for the computation, but require many more iterations of comparison.

http://cc.ee.ntu.edu.tw/~jhjiang/instruction/courses/fall10-lsv/lec07_4p.pdf

- Let’s use what Prof. Keutzer taught at Berkeley 7-1-techmap.pdf

Technology Dependent Logic Optimization

Prof. Kurt Keutzer

EECS

University of California

Berkeley, CA

Thanks to S. Devadas

Thank Prof. Keutzer and S. Devadas

Reasonable Library

Inverter, Buffer

ND2-ND4; NOR2-NOR4; AND2- AND4;

AOI21 - AOI333; OAI21 - OAI333

XOR, XNOR

MUX, Full Adder

Neg-Edge Triggered D-Flip-Flop

Pos-Edge Triggered D-FF

J-K FF

Above with various clears, enables

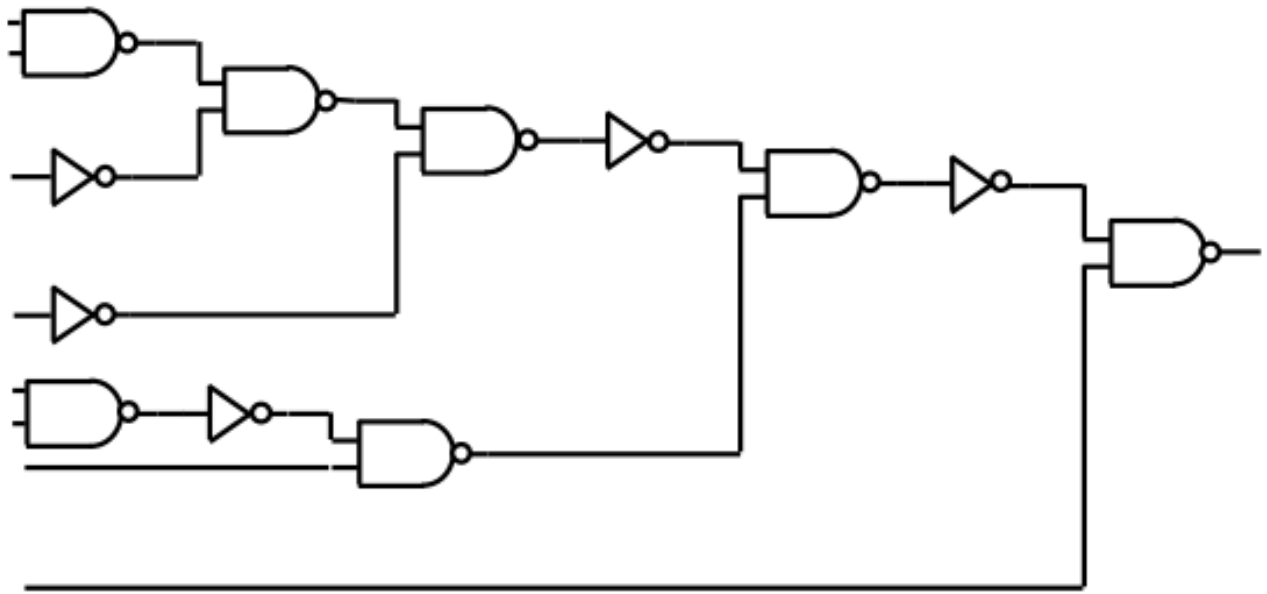
Scan versions of each of the above

Most of the above in 6 different power sizes:

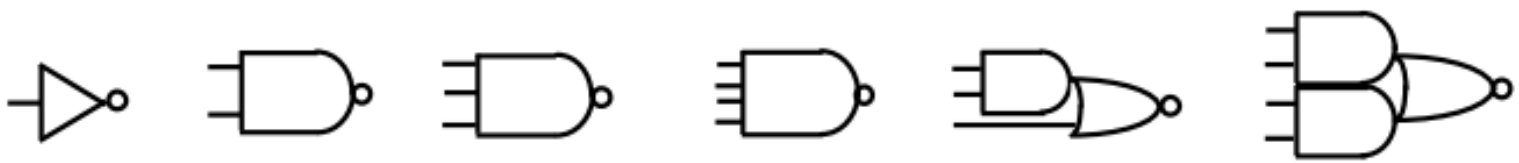
- 1x, 2x, 4x, 6x, 8x, 16x

Problem statement

Find an ``optimal'' (in area, delay, power) mapping of a circuit



into the technology library (simple example below):



History of the Problem - 1

Technology mapping in 1986 was a big problem

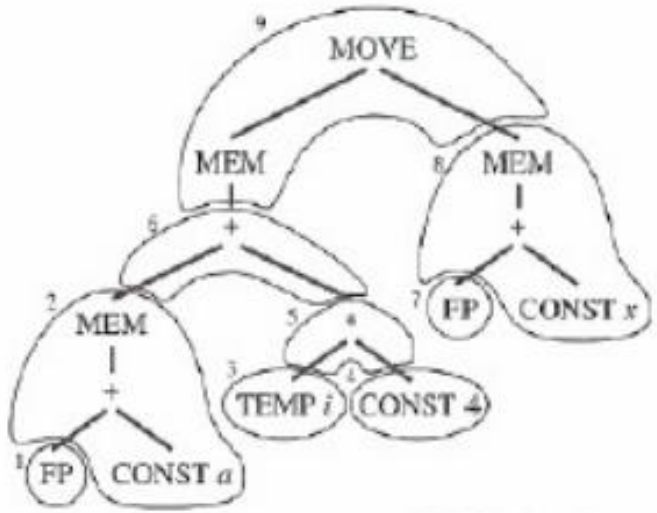
- Almost every design group (e.g. AT&T) had their own library
  - ASIC – 400 cells
  - Microprocessor/DSP – 200 base cells
  - Government – 200+ cells
- Every group had their own approach to mapping
  - ``Do what you have to do!'' – handcrafted mappers tied to particular libraries and optimization tools
  - ``Rule-based'' systems – e.g. GE Socrates – very slow ``expert systems'' that made no guarantee on final quality of result

History of the Problem - 2

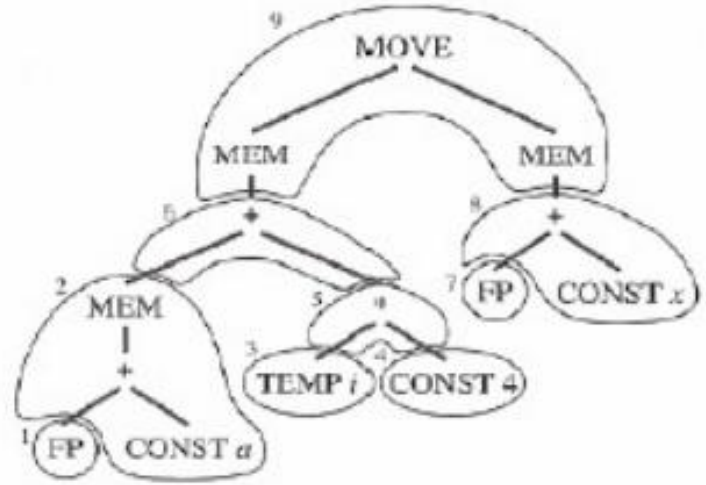
Yes, there are two problems:

- **Technology mapping can significant affect the area, speed, and power dissipation of a circuit**
- **There are over 200 different semiconductors each with multiple internal libraries – how to create a tool that can utilize a diverse set of libraries??**

A similar problem – code generation



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	LOAD	$r_2 \leftarrow M[\mathbf{fp} + x]$
9	STORE	$M[r_1 + 0] \leftarrow r_2$



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	ADDI	$r_2 \leftarrow \mathbf{fp} + x$
9	MOVEM	$M[r_1] \leftarrow M[r_2]$

Example of code generation in compilers using tree-covering

- **Handles complex instruction sets → Handles complex libraries**
- **Easily portable to other instruction sets → Easily portable to**

Problem Formulation: DAG Covering

Represent input netlist in normal form
⇒ subject DAG

Represent each library gate with normal forms for the logic function
⇒ primitive DAGs

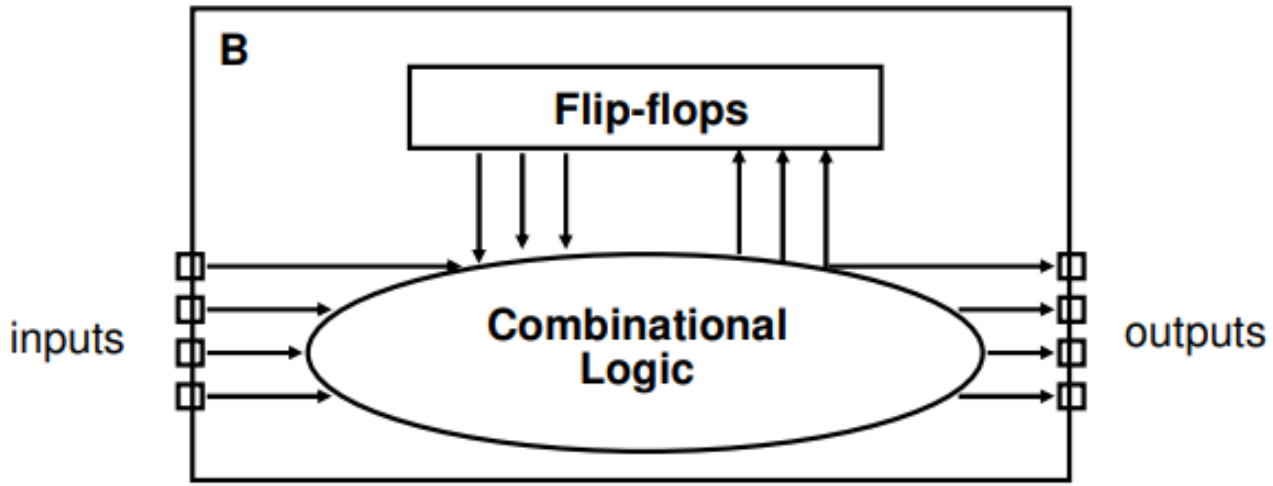
Each primitive DAG has a cost

Goal: Find a minimum cost covering of the subject DAG by the primitive DAGs

Normal form: 2-input NAND gates and inverters

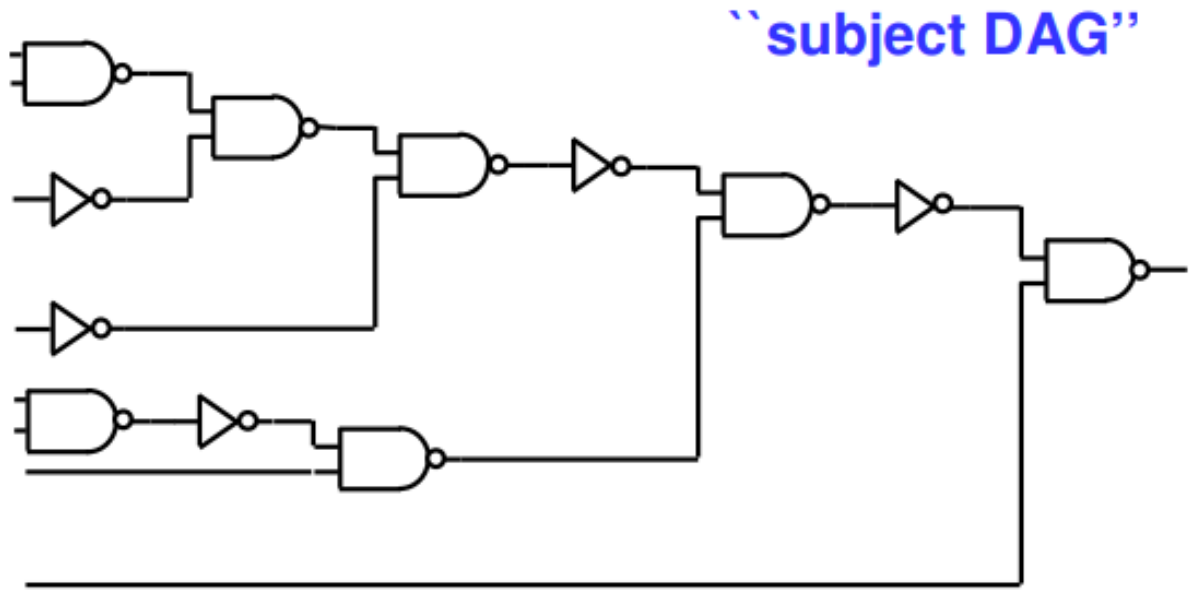
K. Keutzer, *DAGON: Technology Binding and Local Optimization by DAG Matching,* in Proceedings of the 24th Design Automation Conference, 1987 and 25 Years of Design Automation

Step 1: Extract Combinational Logic



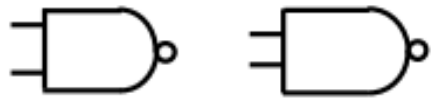
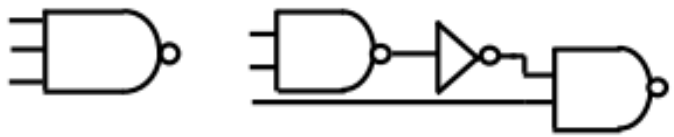
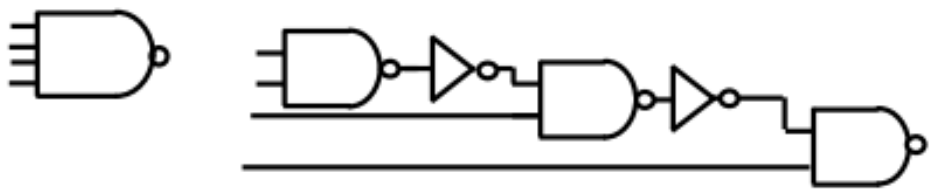
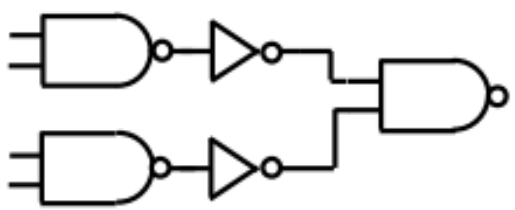
Since FF's don't need to be optimized with surrounding combinational logic we can partition them out

Step 2: Normalize Circuit Netlist

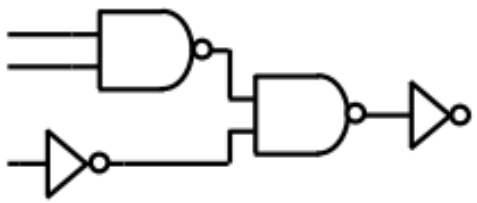
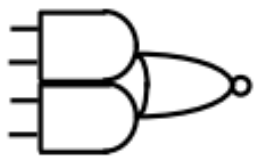
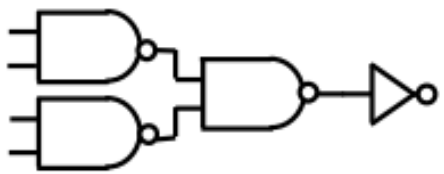


Reduce the netlist into ND2 gates

Step 3a: Normalize library

	Element/Area Cost	Tree Representation (normal form)
INVERTER	2	
NAND2	3	
NAND3	4	
NAND4	5	
		

Step 3b: Normalize library

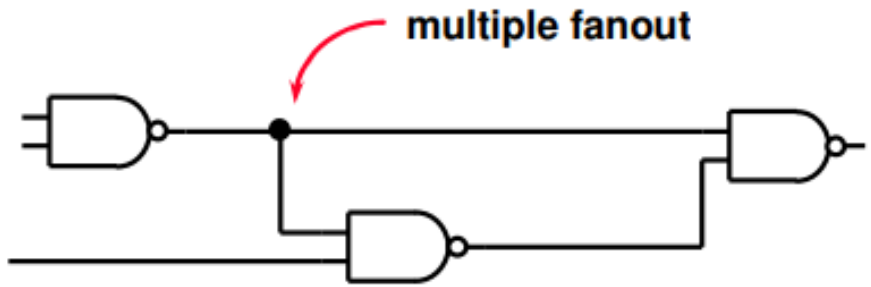
	Element/Area Cost		Tree Representation (normal form)
AOI21	4		
AOI22	5		

Step 4: DAG Covering

Sound Algorithmic approach

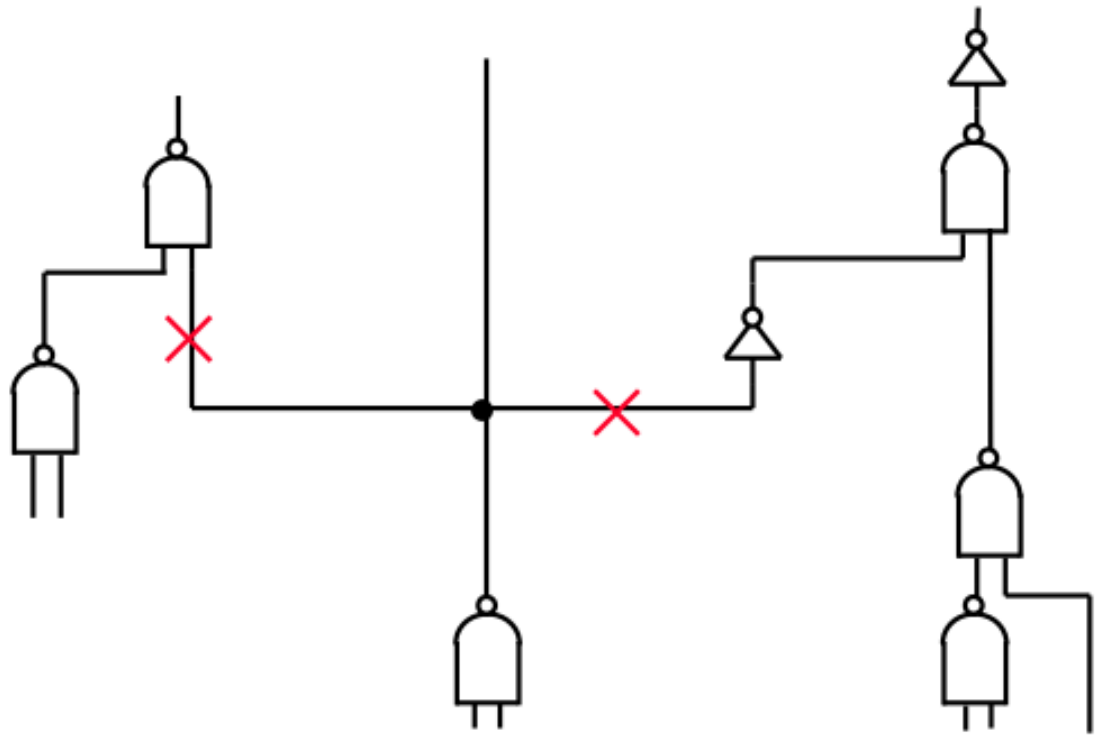
NP-hard optimization problem

K. Keutzer, D. Richards, *Computation Complexity of Logic Synthesis and Optimization,* in Proceedings of the International Workshop on Logic Synthesis, 1989



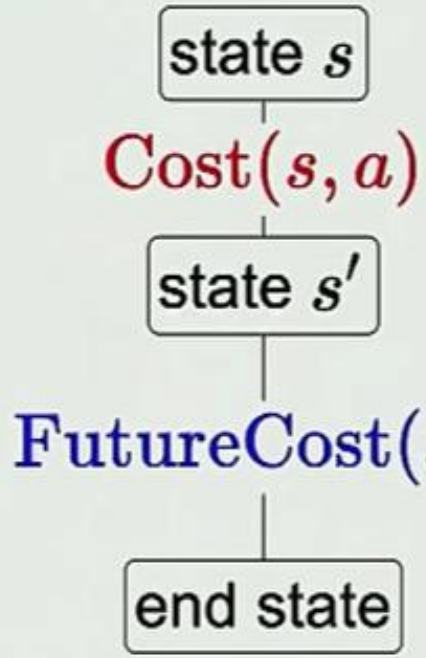
Tree covering heuristic: If subject and primitive DAGs are trees, efficient algorithm can find optimum cover ⇒ dynamic programming formulation

Solution formulation



1) Partition input netlist into forest of *trees*
2) Solve each tree *optimally* using tree covering
3) Stitch trees back together

Dynamic programming



Minimum cost path from state s to a end state:

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

a: action

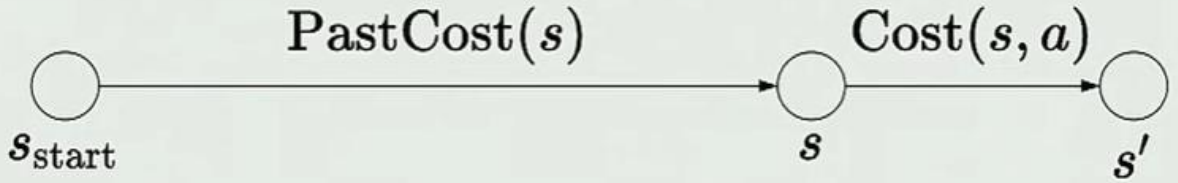
Succ: successor after taking action a

Tree search

Dynamic programming

Uniform cost search

Observation: prefixes of optimal path are optimal

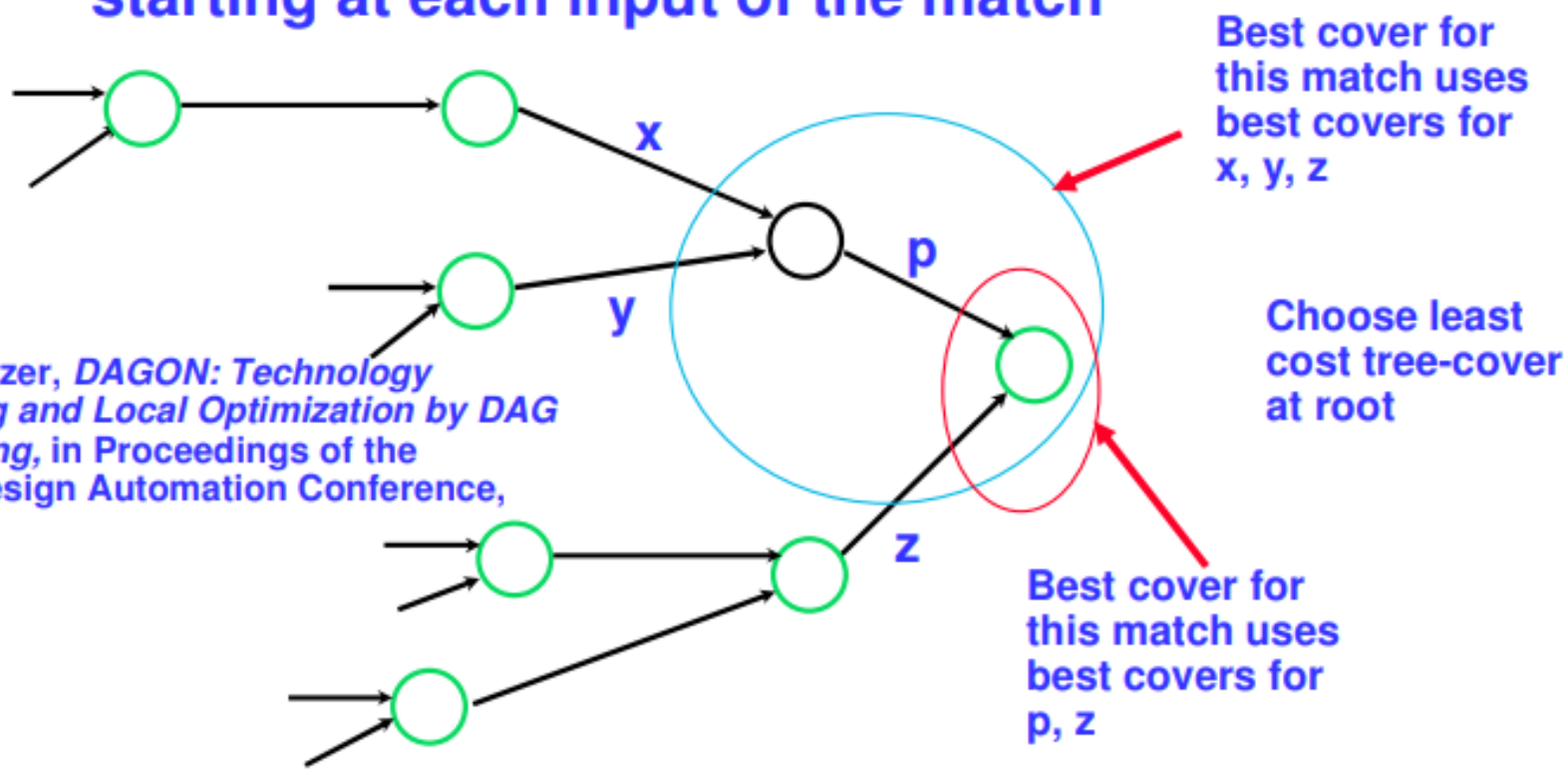


Key: if graph is acyclic, dynamic programming makes sure we compute $\text{PastCost}(s)$ before $\text{PastCost}(s')$

If graph is cyclic, then we need another mechanism to order states...

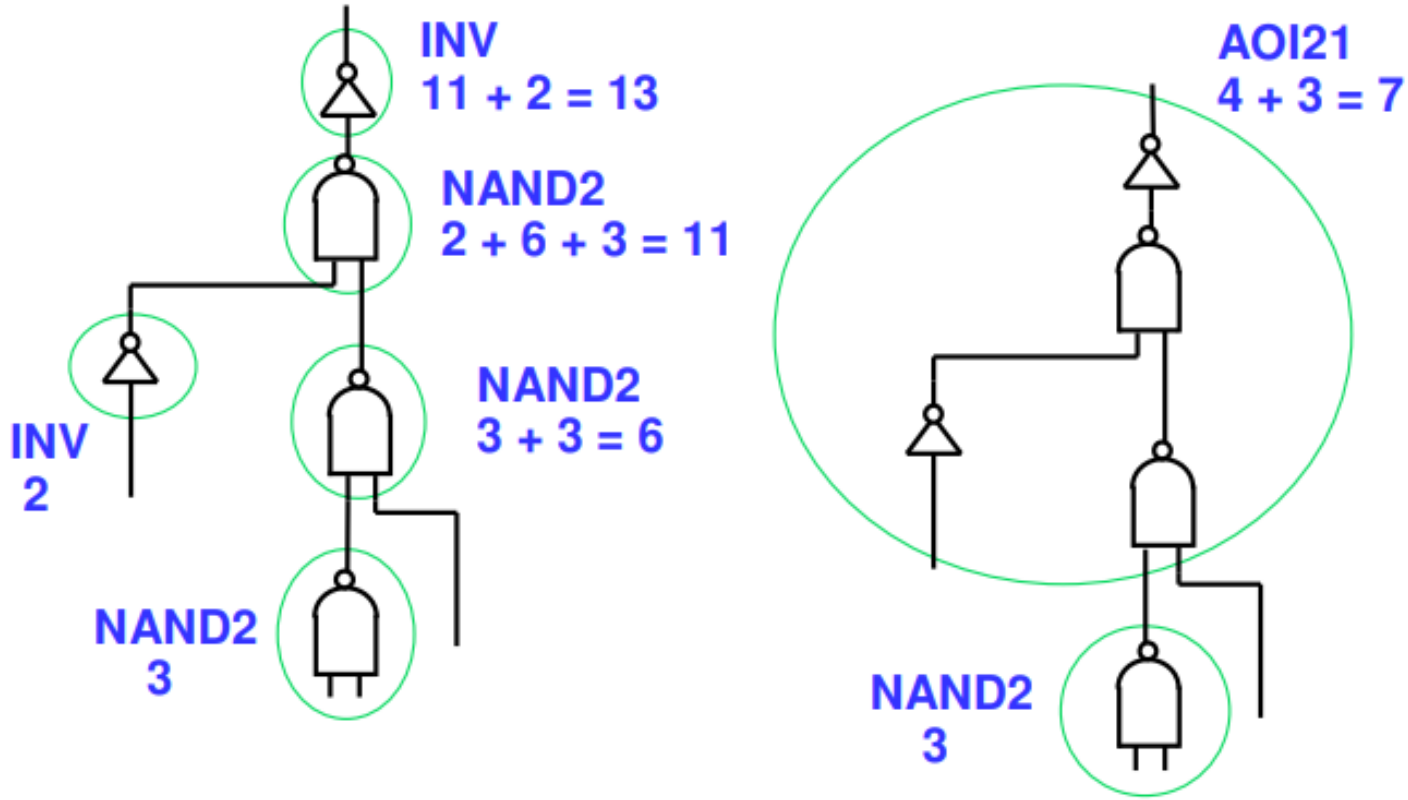
For each tree - Dynamic Programming

Principle of optimality: Optimal cover for a tree consists of a best match at the root of the tree plus the optimal cover for the sub-trees starting at each input of the match



K. Keutzer, *DAGON: Technology Binding and Local Optimization by DAG Matching,* in Proceedings of the 24th Design Automation Conference, 1987

Example of Optimal Tree Covering



DAG covering in detail

1) partition DAG into a forest of trees

2) normalize netlist

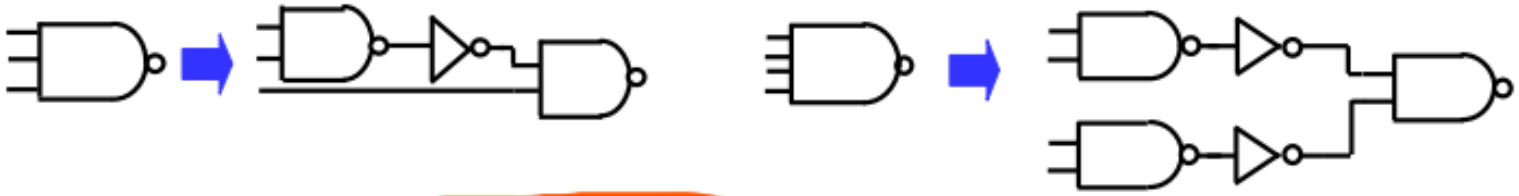
3) optimally cover each tree

   a) generate all candidate matches

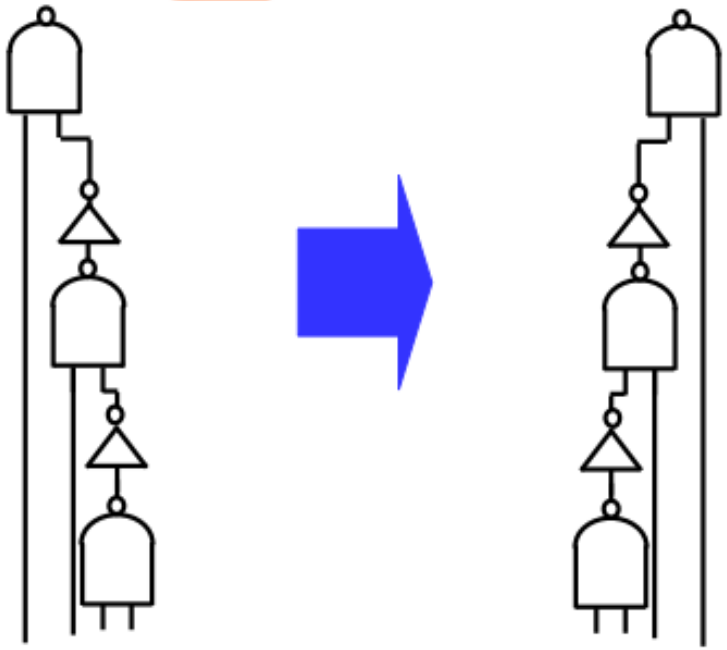
   b) find the optimal match using dynamic programming

Normalize netlist

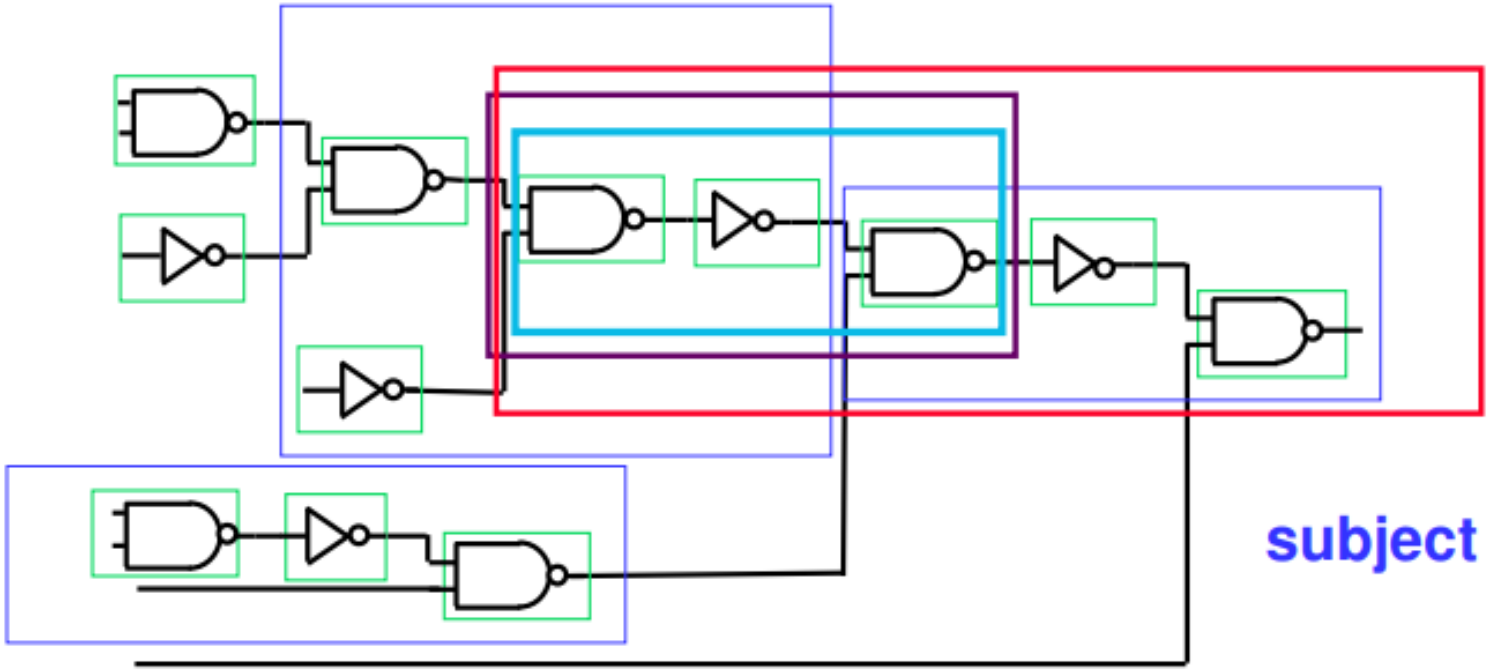
Re-express netlist into 2-input Nand gates and Inverters



Make each tree left-oriented



Generate candidate matches - 1



subject tree

At the end of this segment each gate in the subject tree is annotated with every possible library cell that could be rooted at that gate

What are some ways we can generate matches?

Generating candidate matches -2

Naïve approach -

try to match each cell in the library with each node of the tree (libraries can be large! - beware of large constants!!)

Better approach

build tables such that only potential candidate matches are checked

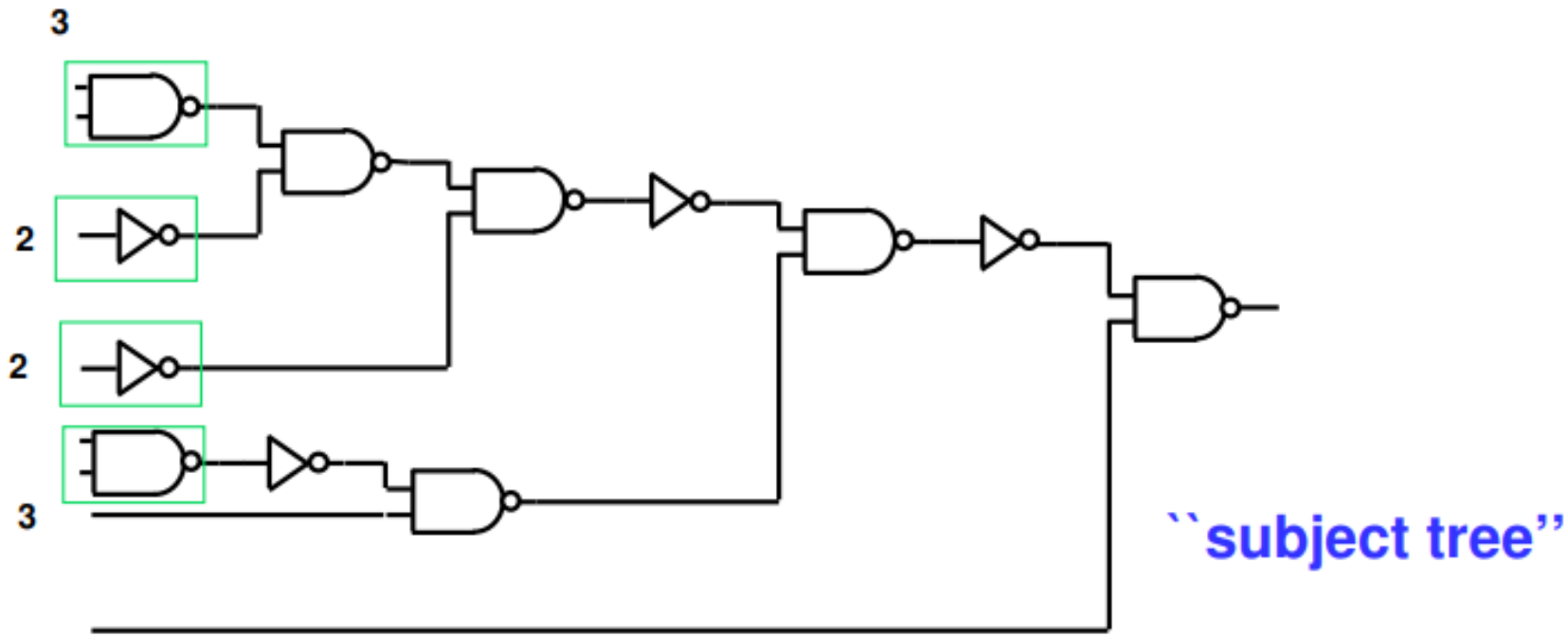
Best approach

fancy string matching - pp. 862-869

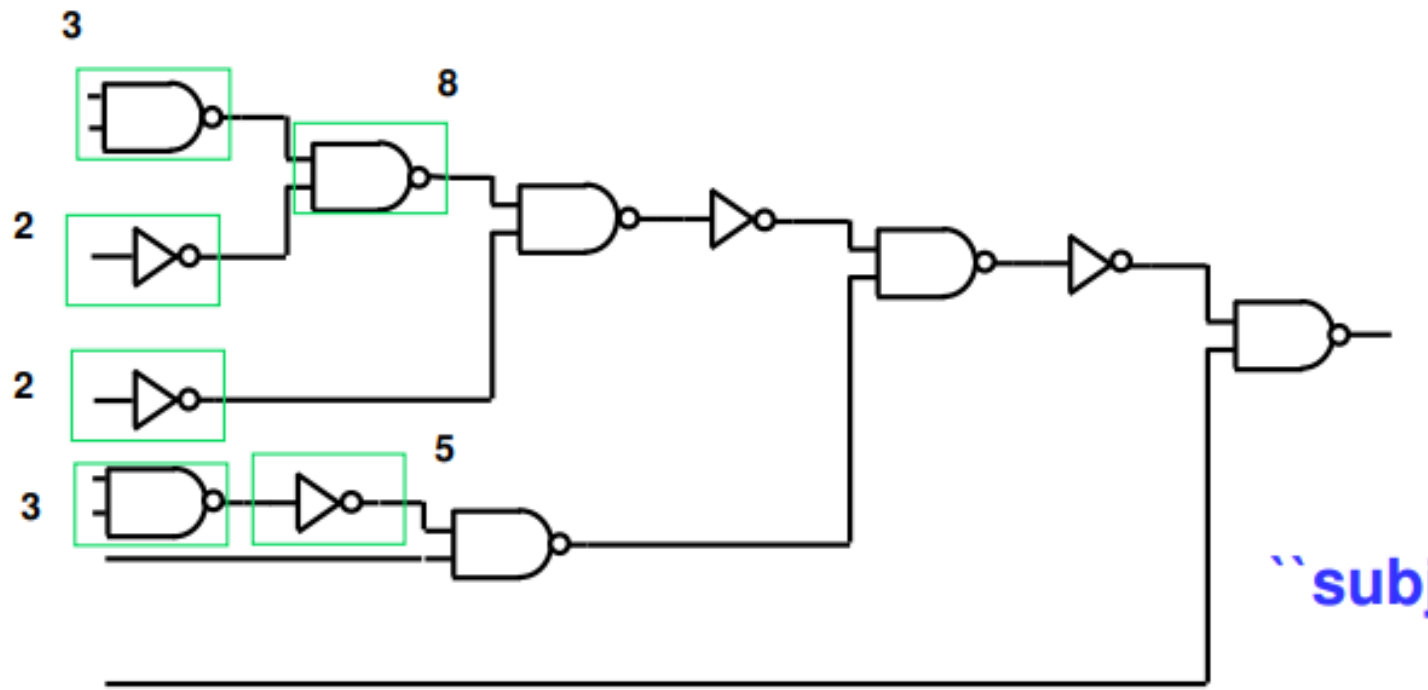
Introduction to Algorithms, T. Cormen, C. Lesierson, R. Rivest, The MIT Press, Second Printing, 1996. - pp. 862-869

What's the complexity of each approach?

Optimal tree covering - 1

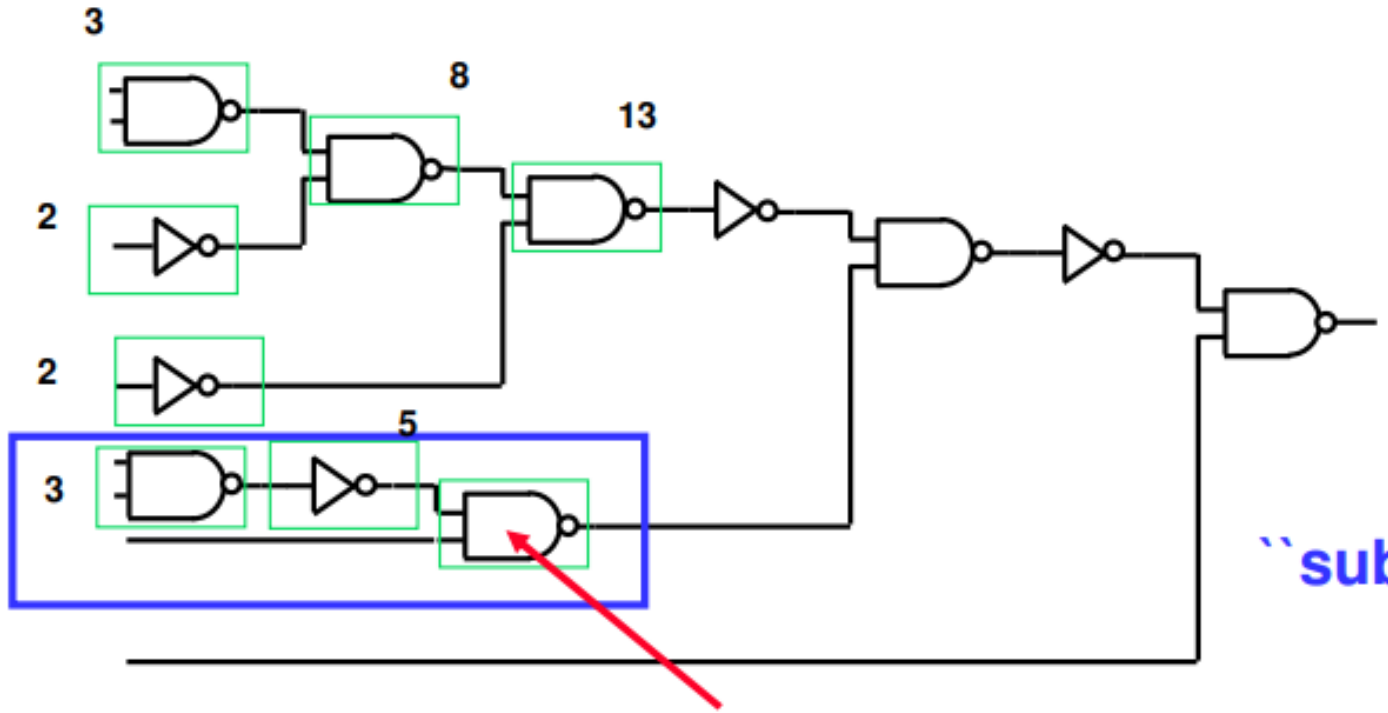


Optimal tree covering - 2



``subject tree''

Optimal tree covering - 3



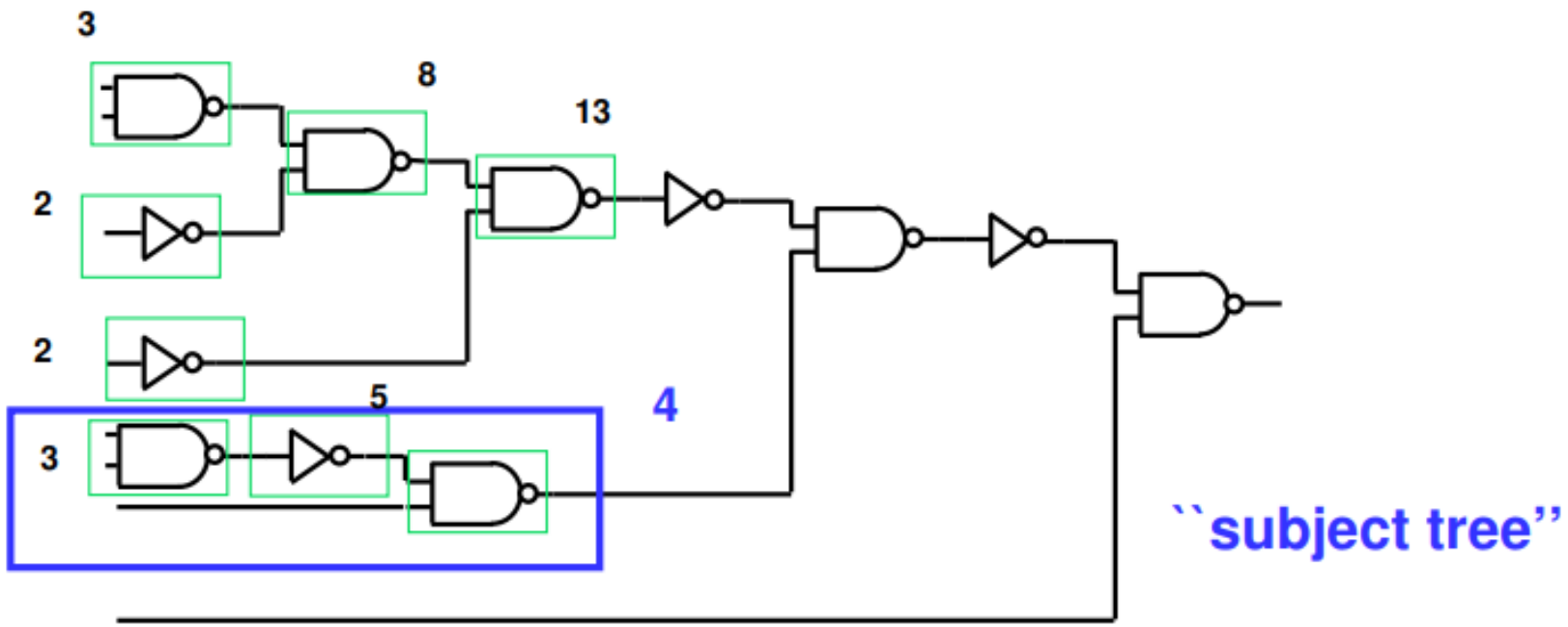
``subject tree''

Cover with ND2 or ND3 ?

1 NAND2	3	1 NAND3	= 4
+ subtree	5		

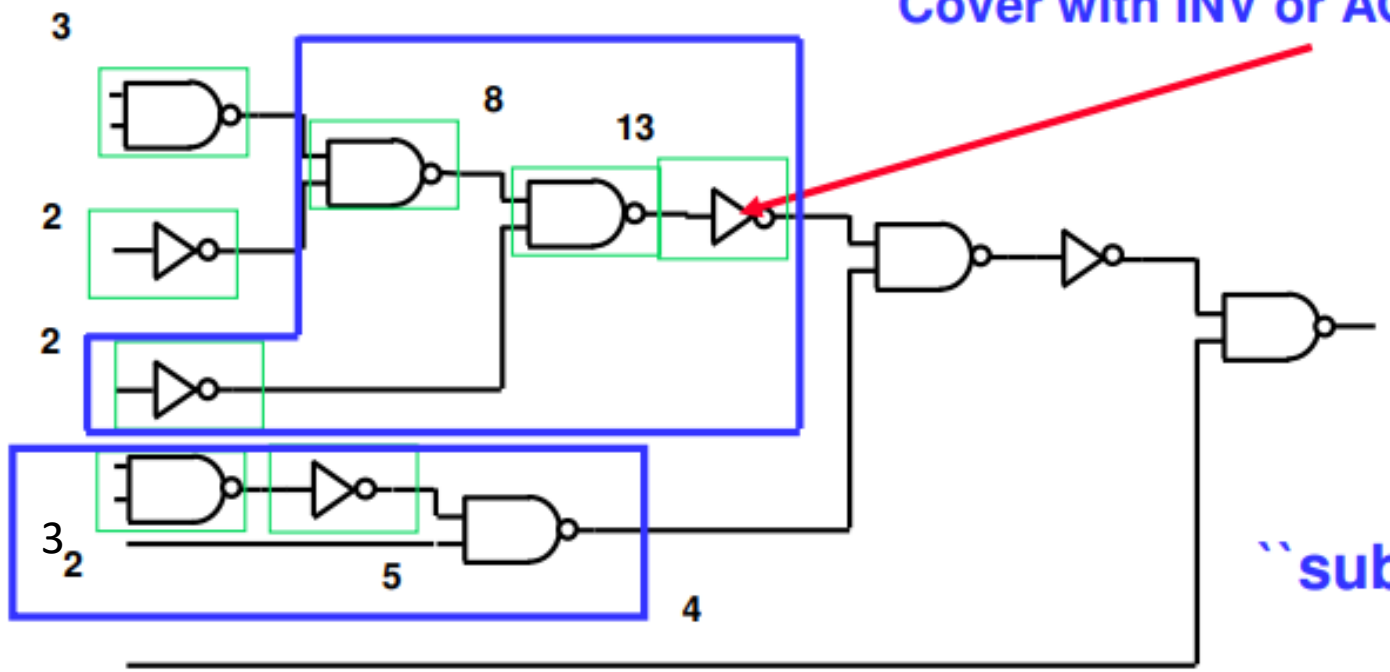
Area cost 8

Optimal tree covering – 3b



Label the root of the sub-tree with optimal match and cost

Optimal tree covering – 4a



Cover with INV or AO21 ?

``subject tree''

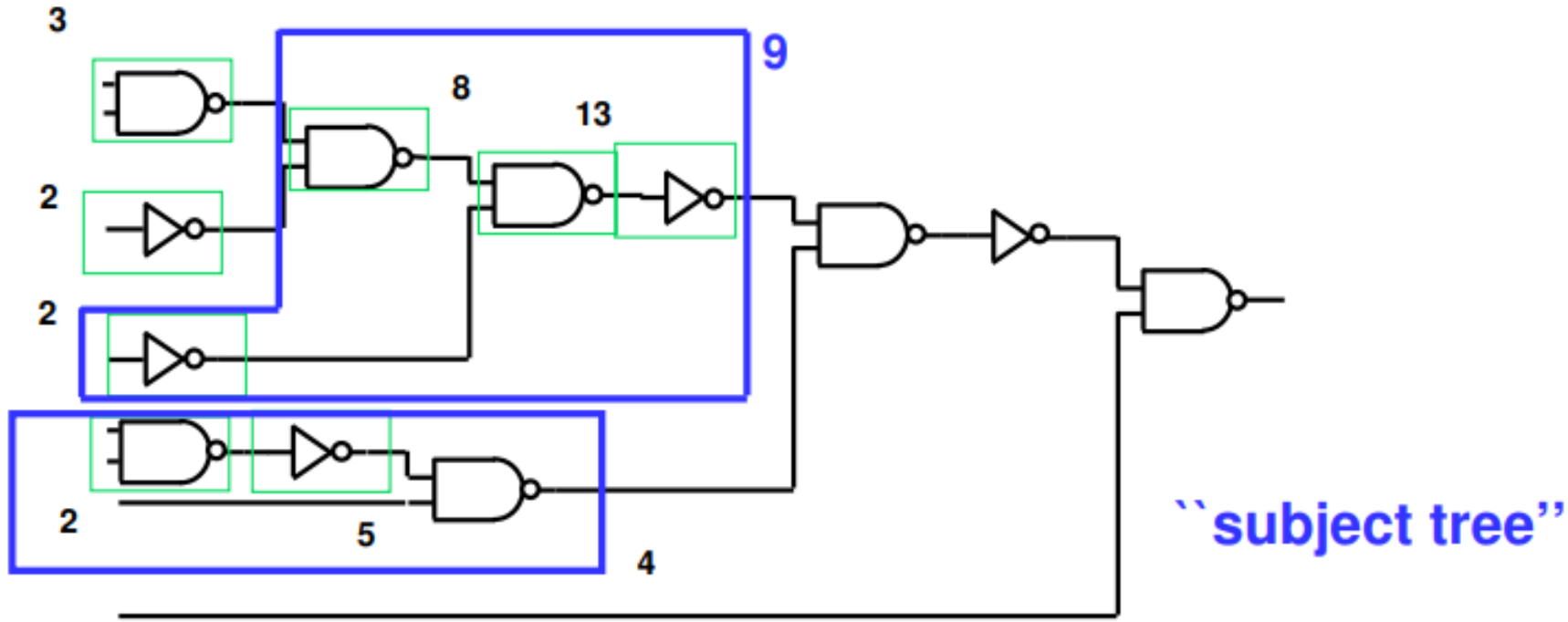
1 Inverter	2
+ subtree	13

Area cost 15

1 AO21	4
+ subtree 1	3
+ subtree 2	2

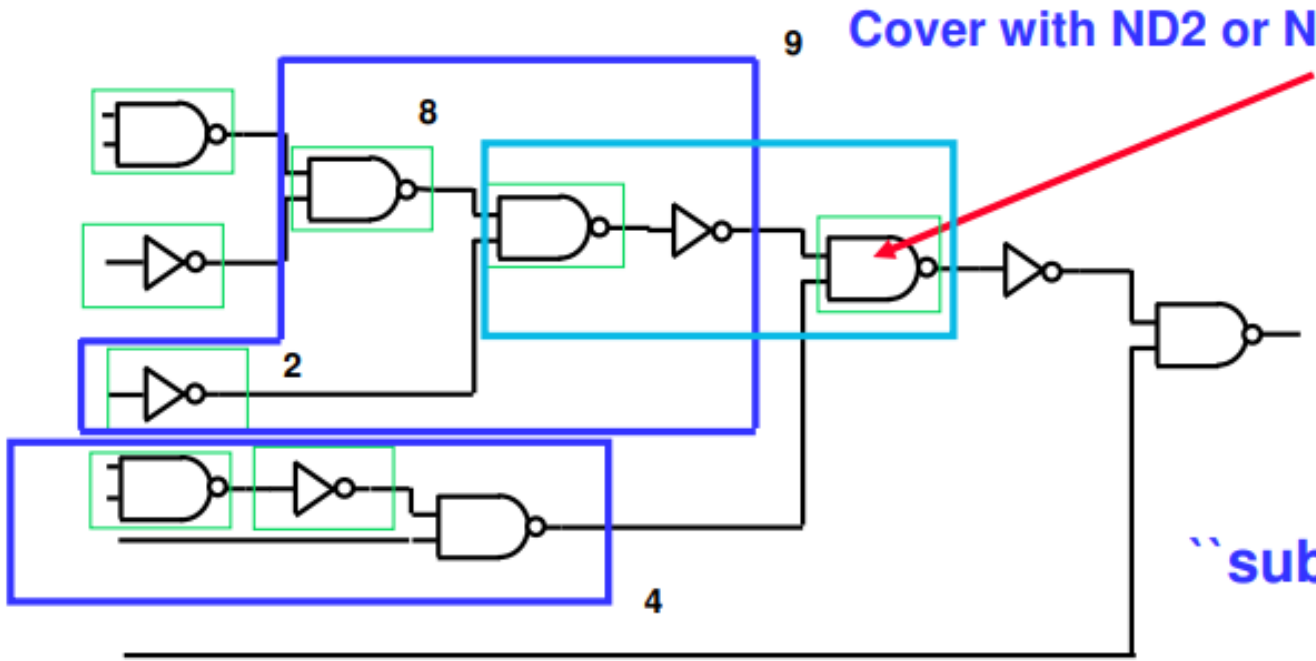
Area cost 9

Optimal tree covering – 4b



Label the root of the sub-tree with optimal match and cost

Optimal tree covering - 5

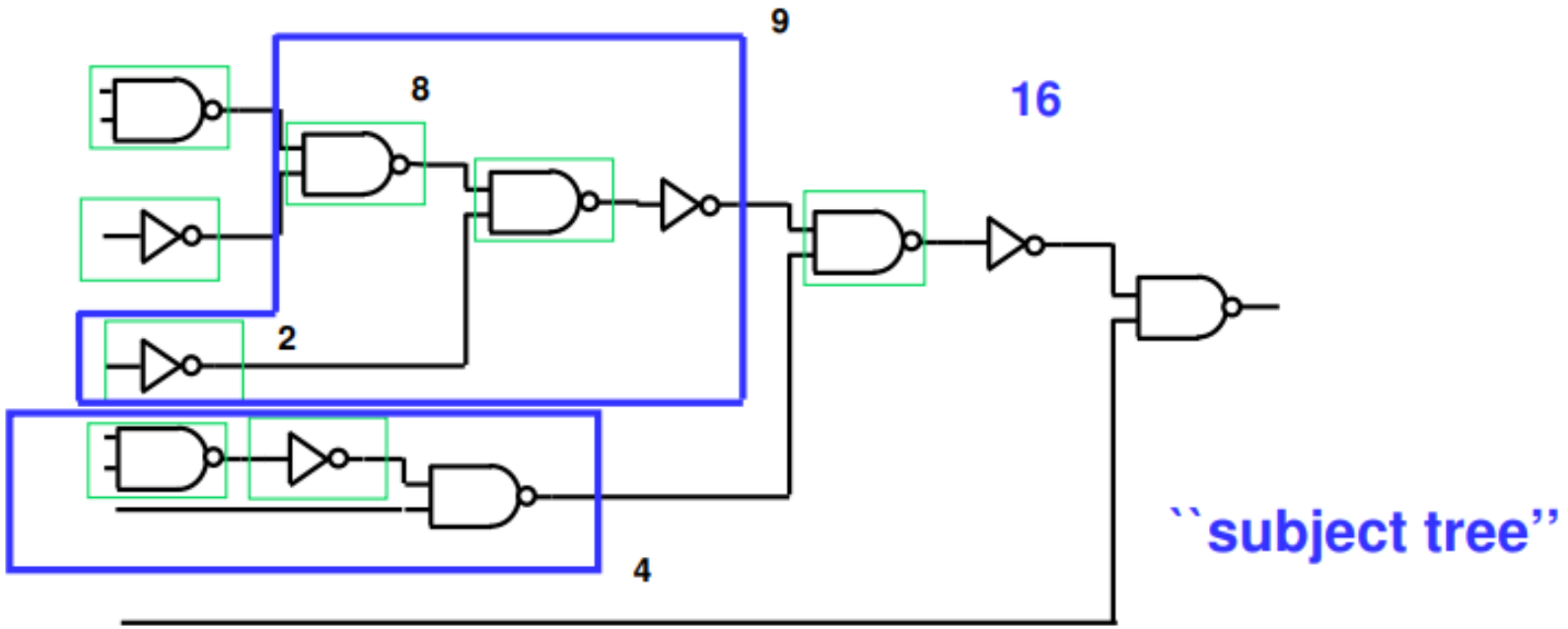


Cover with ND2 or ND3 ?

``subject tree''

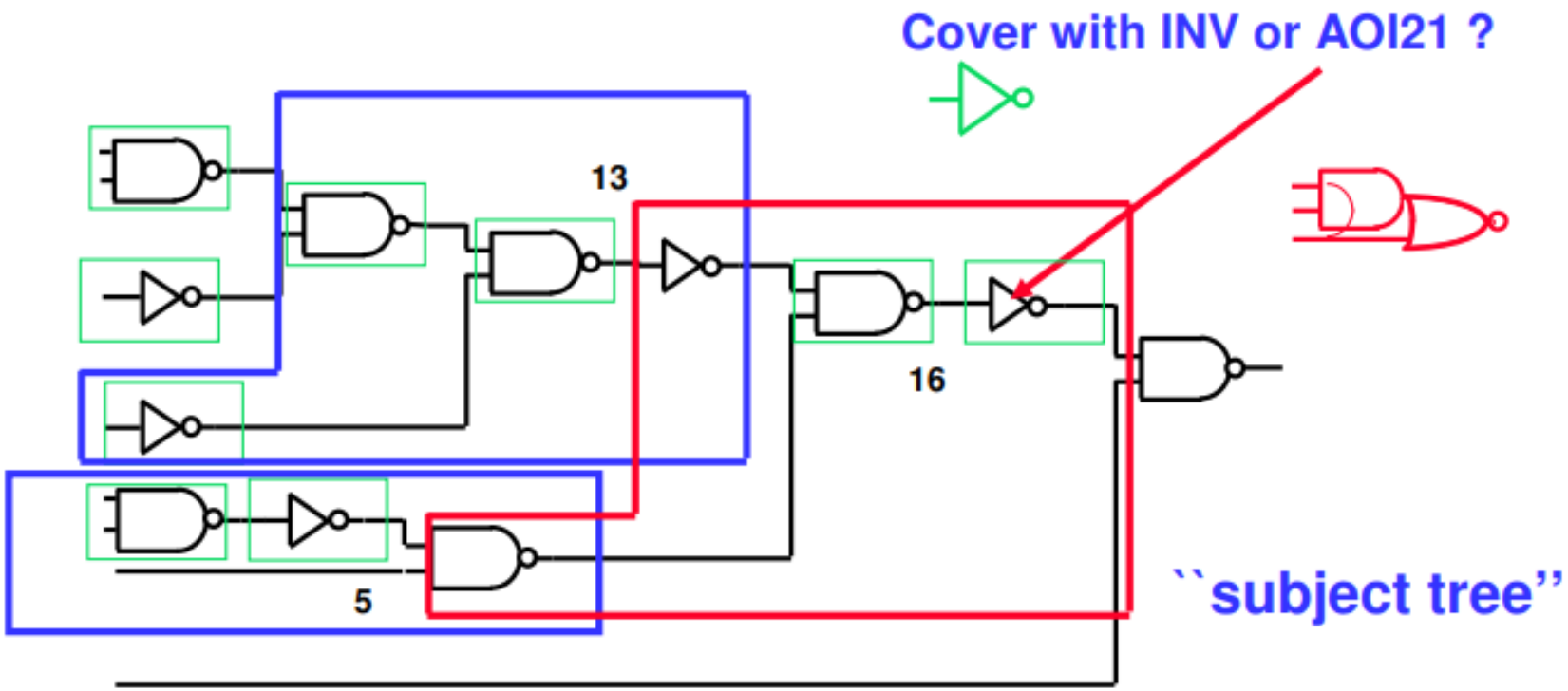
NAND2			NAND3		
	subtree 1	9		subtree 1	8
	subtree 2	4		subtree 2	2
	1 NAND2	3		subtree 3	4
				1 NAND3	4
	Area cost 16			Area cost 18	

Optimal tree covering – 5b



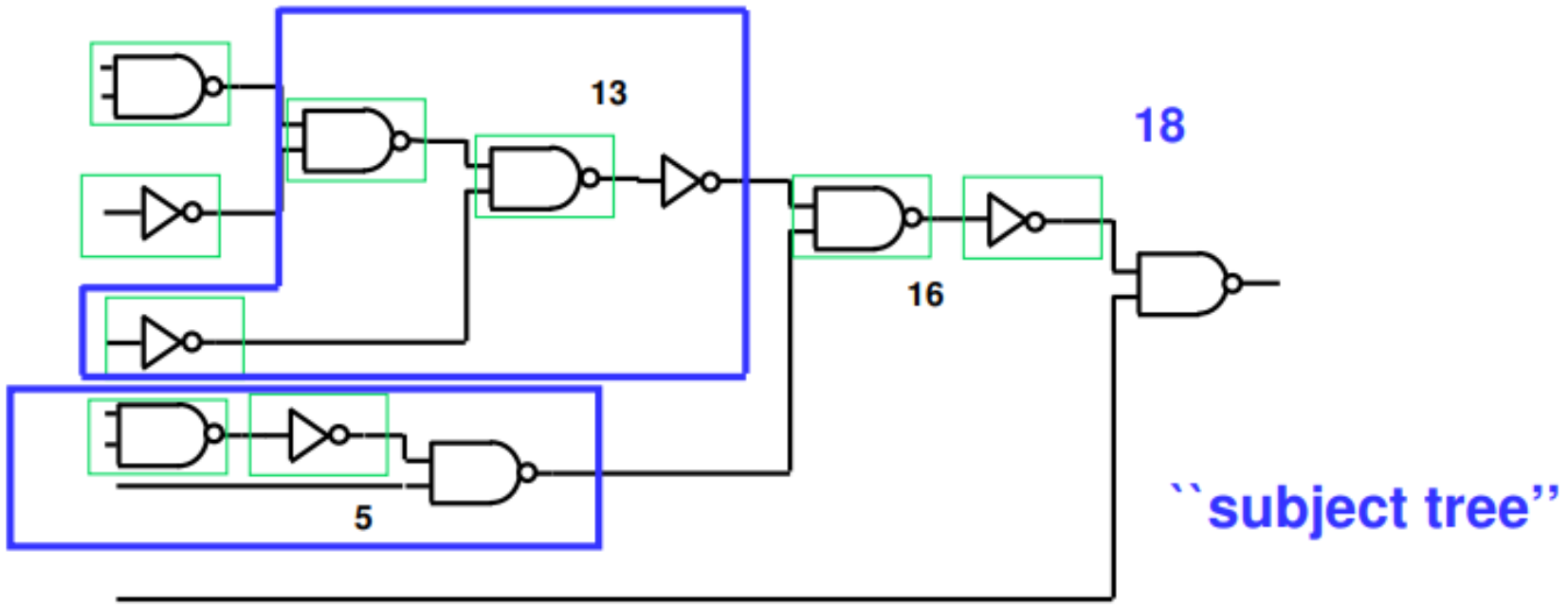
Label the root of the sub-tree with optimal match and cost

Optimal tree covering - 6



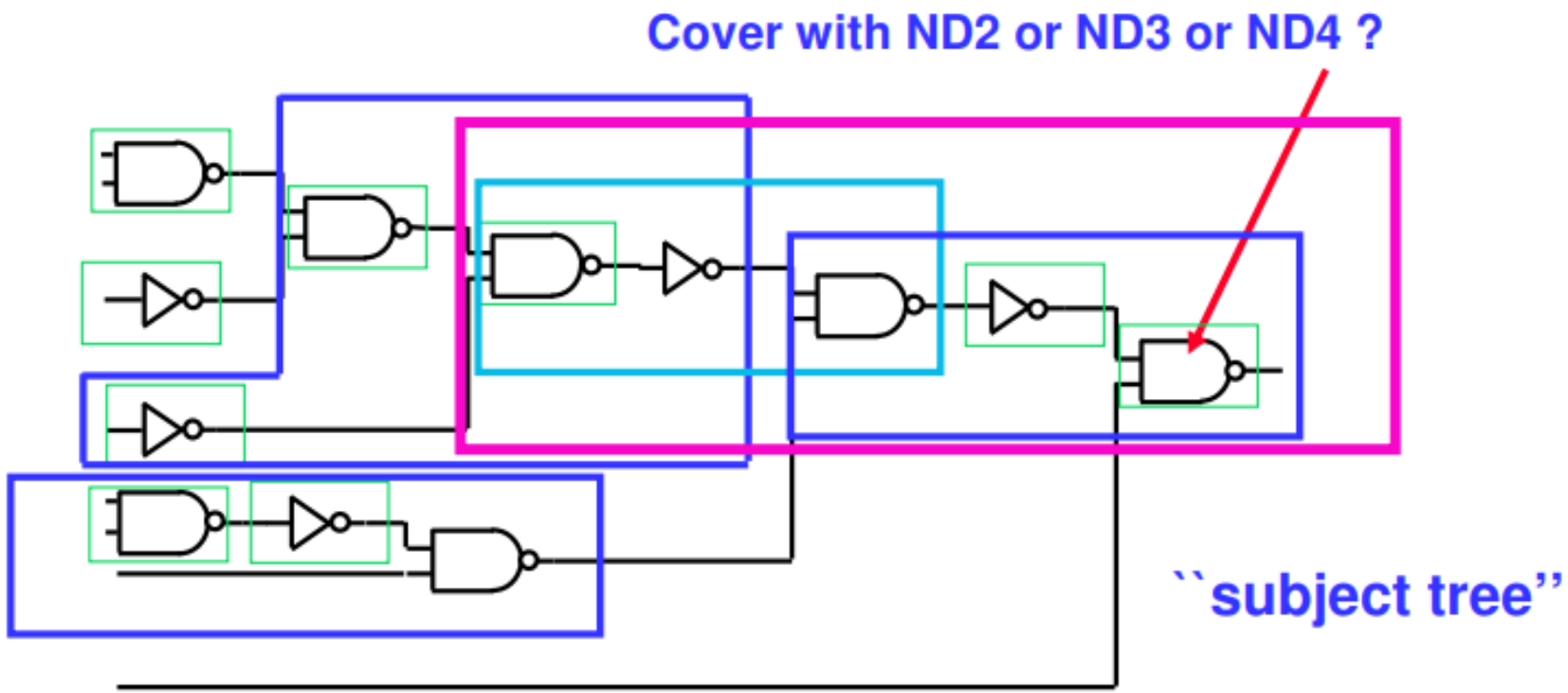
INV	subtree 1	16	AOI21	subtree 1	13
	1 INV	2		subtree 2	5
				1 AOI21	4
	Area cost 18			Area cost 22	

Optimal tree covering – 6b

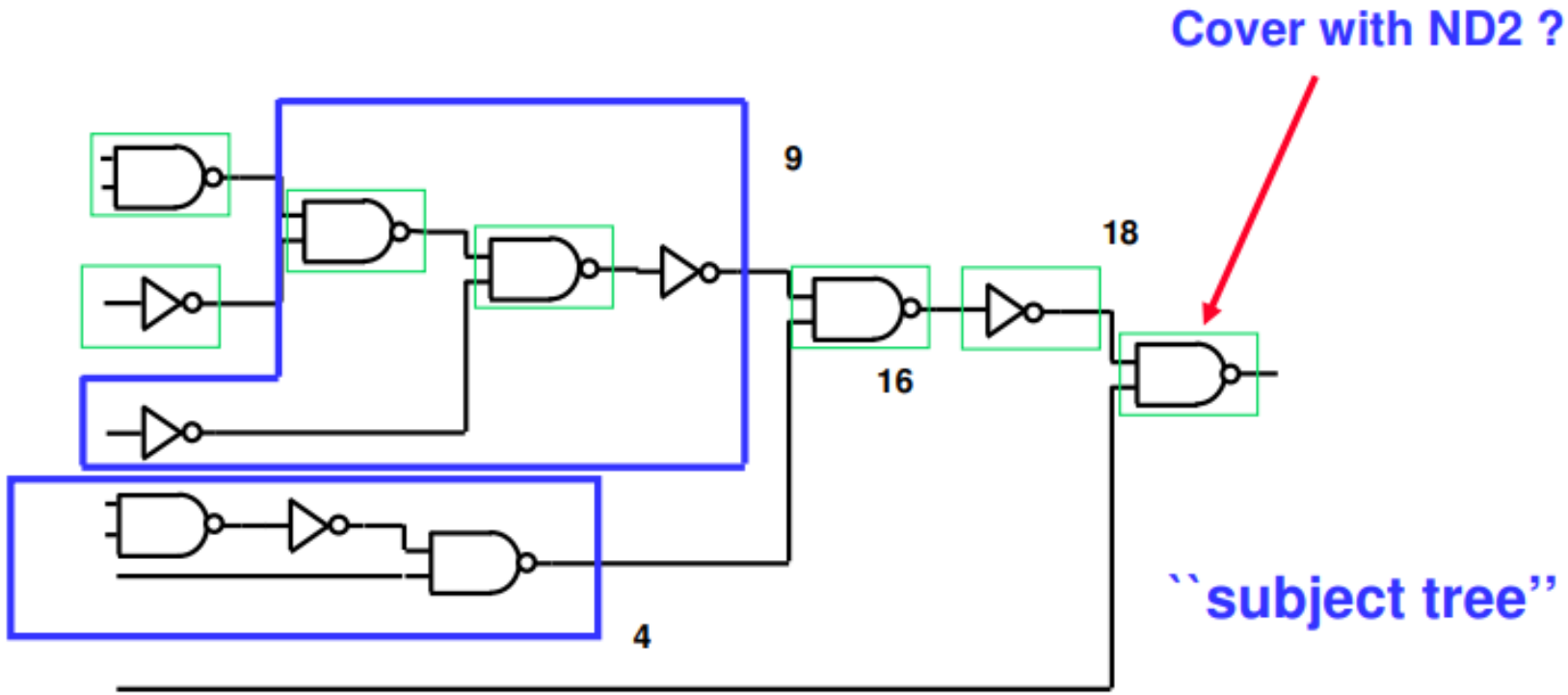


Label the root of the sub-tree with optimal match and cost

Optimal tree covering - 7

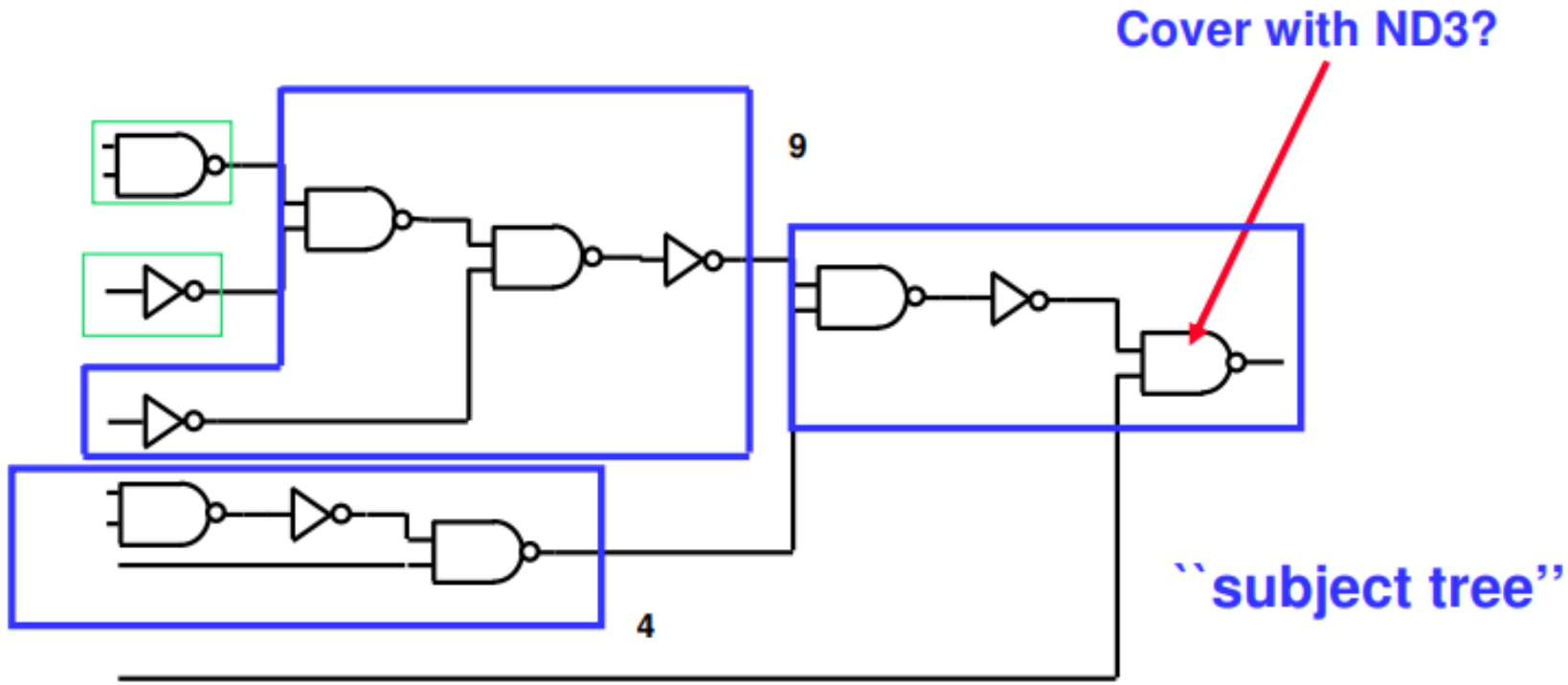


Cover 1 - NAND2



subtree 1	18
subtree 2	0
1 NAND2	3
	Area cost 21

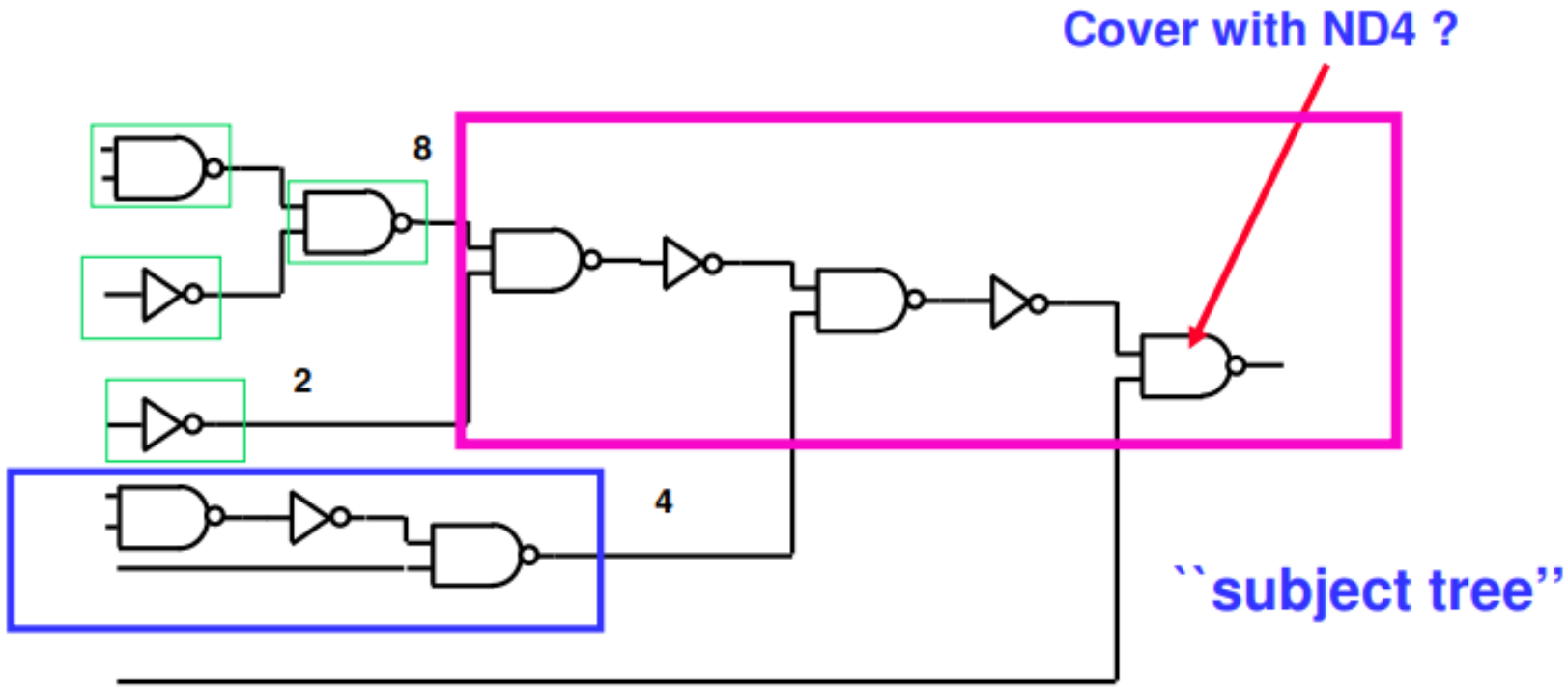
Cover 2 - NAND3



subtree 1	9
subtree 2	4
subtree 3	0
1 NAND3	4

Area cost 17

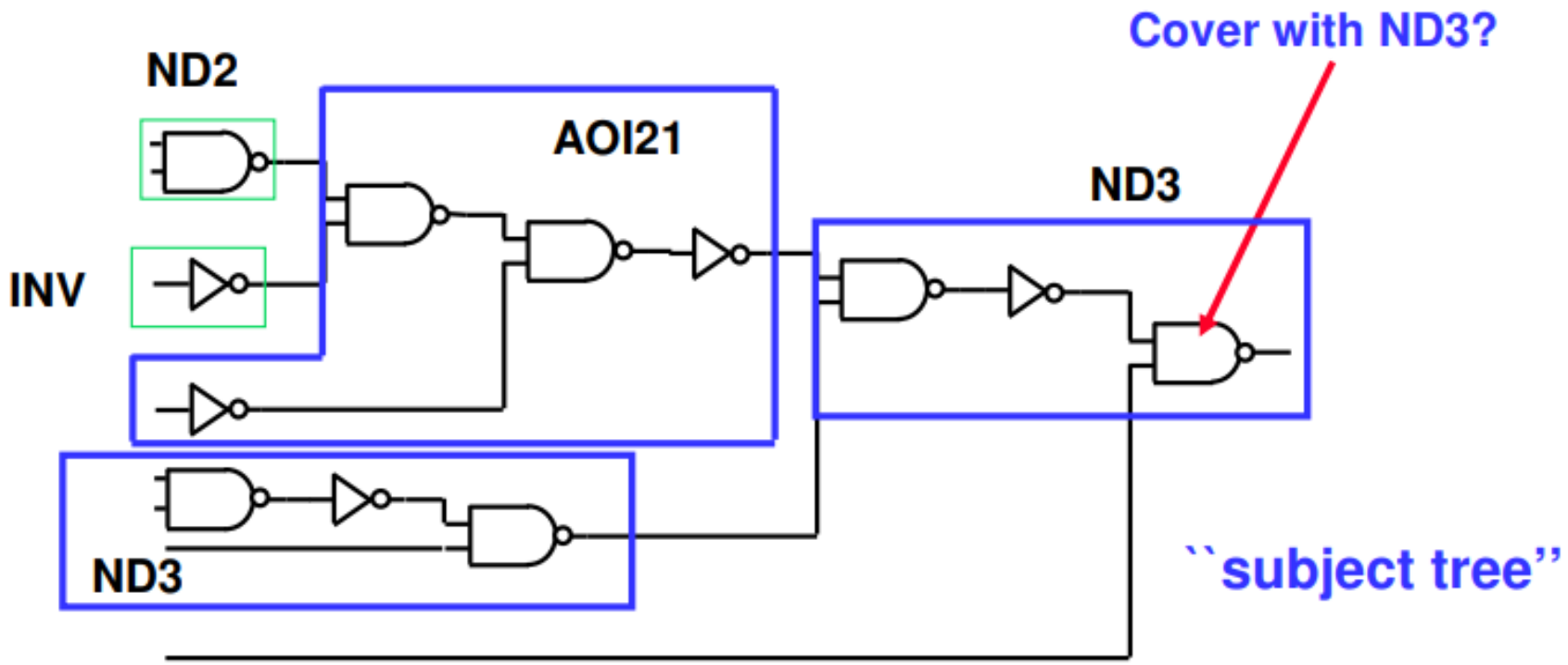
Cover - 3



subtree 1	8
subtree 2	2
subtree 3	4
subtree 4	0
1 NAND4	5

Area cost 19

Optimal Cover was Cover 2



``subject tree''

Clear that greedy doesn't work well
What's the complexity?

INV	2
ND2	3
2 ND3	8
AOI21	4
	Area cost 17

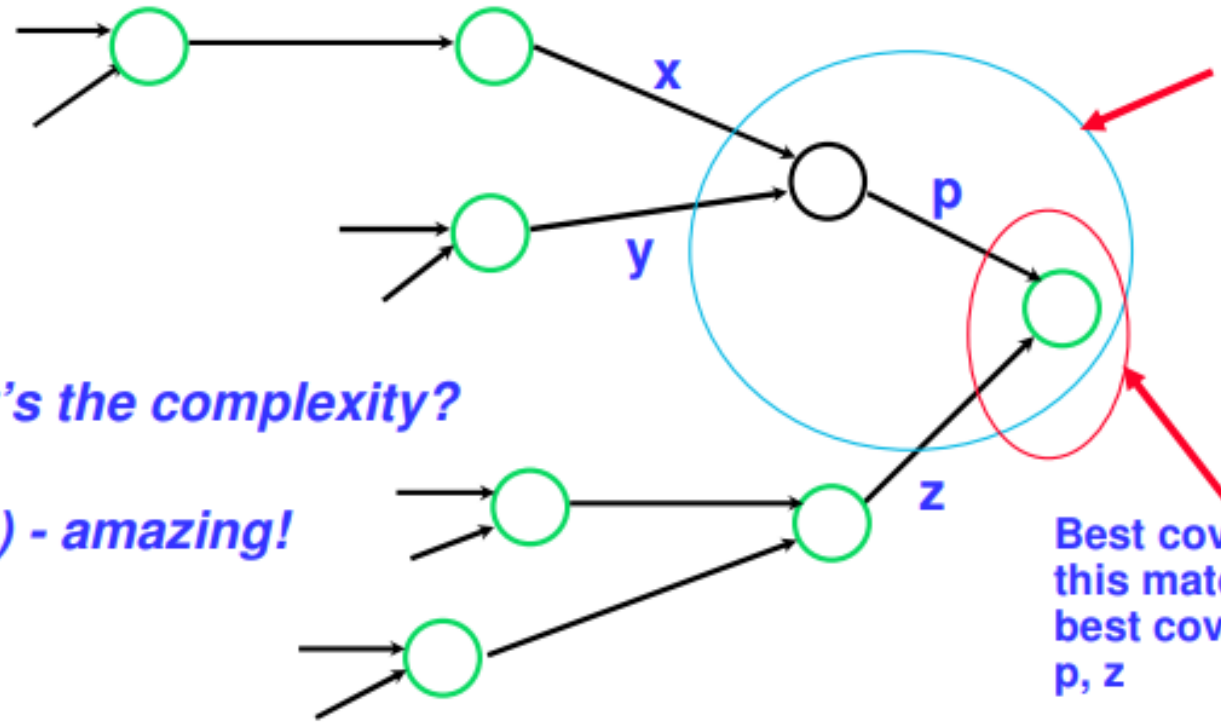
Computational Complexity

To determine the optimal cover for a tree we only need to consider a best cost match at the root of the tree

This is constant-time in the number of matched cells

Plus the optimal cover for the sub-trees starting at each input of the match

This is constant-time in the indegree/fan-in of each match



Best cover for this match uses best covers for x, y, z

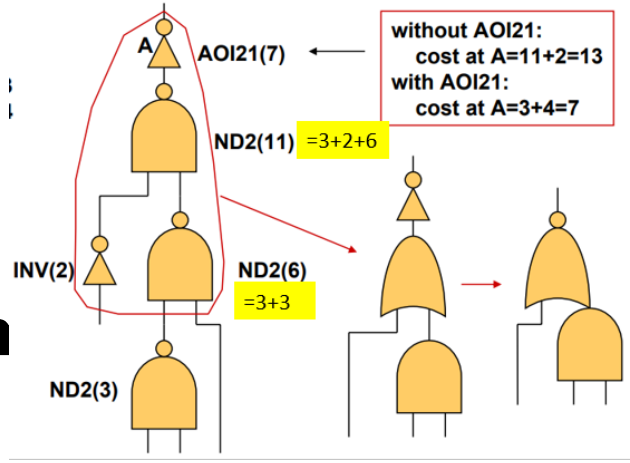
Choose least cost tree-cover at root

Best cover for this match uses best covers for p, z

What's the complexity?

O(n) - amazing!

Tree Covering



1. Partition subject graph into forest of trees
2. **Cover each tree optimally using dynamic programn**
   - Given:
     - Subject trees (networks to be mapped)
     - Forest of patterns (gate library)

   - For each node *N* of a subject tree
     - Recursive Assumption: for all children of *N*, a best cost match (which implements the node) is known
     - Compute cost of each pattern tree which matches at *N*,
       Cost = SUM of best costs of implementing each input of pattern plus the cost of the pattern
       - Cost of a leaf of the tree is 0

     - Choose least cost matching pattern for implementing *N*

Tree Covering

```
Algorithm OPTIMAL_AREA_COVER(node) {
foreach input of node {
   OPTIMAL_AREA_COVER(input);//satisfies recur. assumption
}
// Using these, find the best cover at node
node→area = INFINITY;
node→match = 0;
foreach match at node {
   area = match→area;
   foreach pin of match {
      area = area + pin→area;
   }
   if (area < node→area) {
      node→area = area;
      node→match = match;
      }
   }
}
```

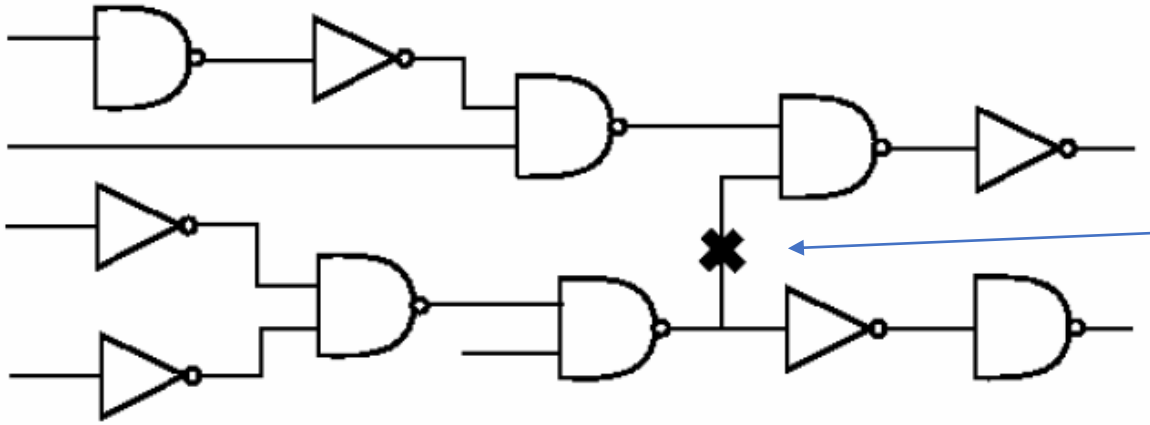
- Complexity
  - Complexity is controlled by finding all sub-trees of the subject graph which are isomorphic to a pattern tree
  - Linear complexity in both size of subject tree and size of collection of pattern trees

Optimal Tree Covering by Dynamic Programming

- If the subject directed acyclic graph (DAG) is a tree, then a polynomial-time algorithm to find the minimum cover exists.
  - Based on dynamic programming: optimal substructure? overlapping subproblems?
- Given: subject trees (networks to be mapped), library cells
- Consider a node *n* of the subject tree
  - Recursive assumption: For all children of *n*, a best match which implements the node is know.
  - Cost of a leaf is 0.
  - Consider each pattern tree which matches at *n*, compute cost as the cost of implementing each node which the pattern requires as an input plus the cost of the pattern.
  - Choose the lowest-cost matching pattern to implement *n*.

Tree-Covering by Dynamic Programming (DAGON)

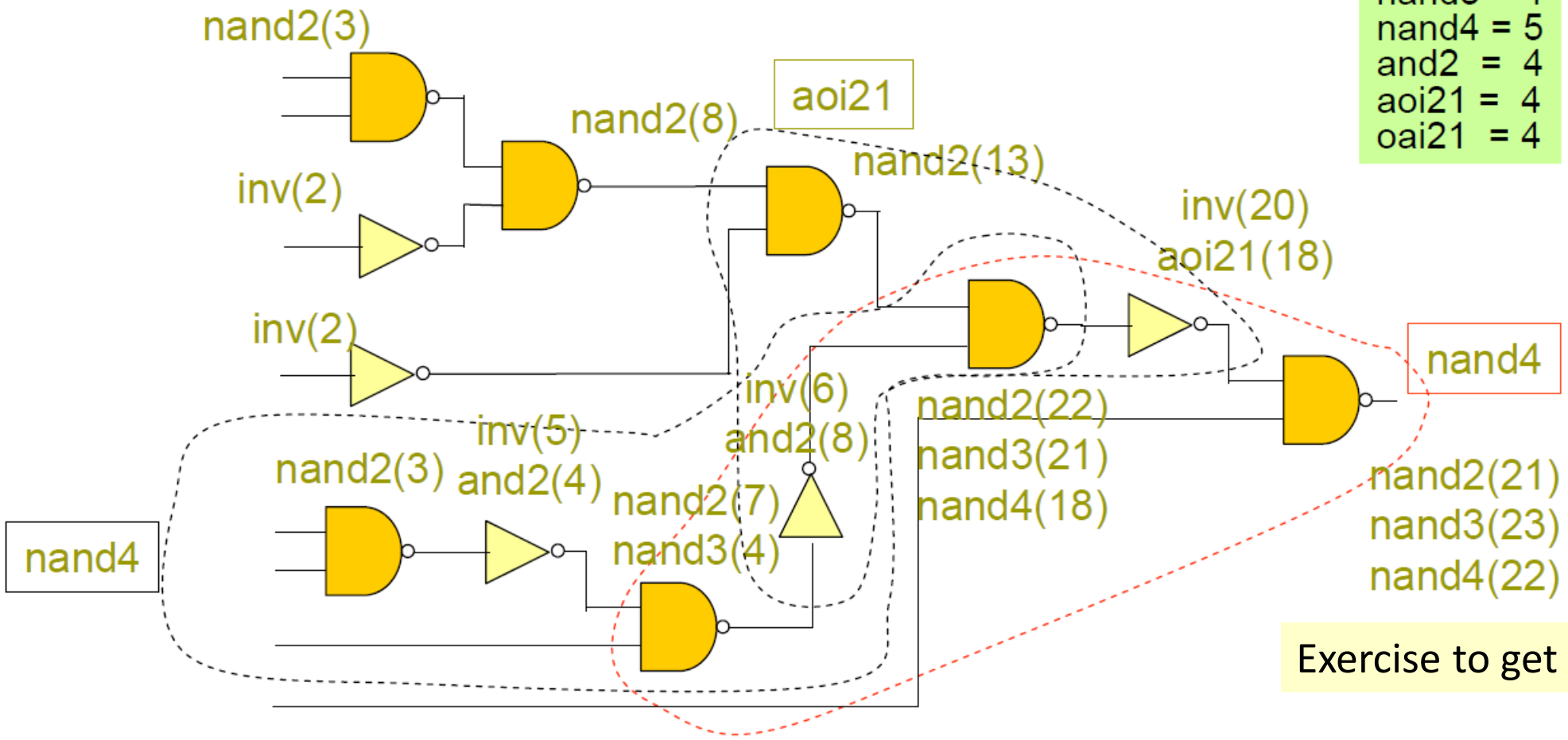
- If the subject DAG is not a tree
  - Partition the subject graph into forest of trees
  - Cover each tree optimally using the dynamic programming.
  - Overall solution is only an approximation.
- Optimality
  - An optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.
  - The minimum area cover for a tree *T* can be derived from the minimum area covers for every node below the root of *T*.



X's marks partition point

Tree Covering

- Example

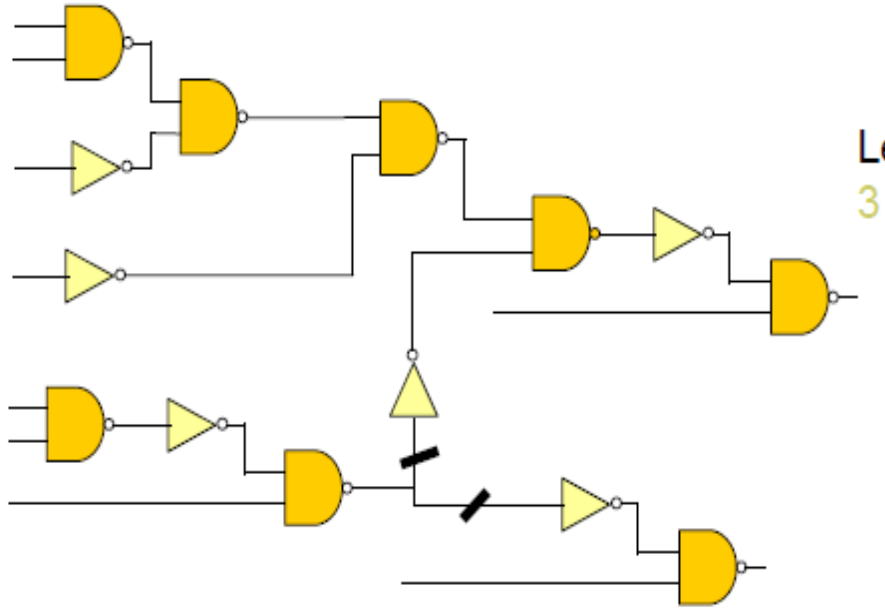


Library:
nand2 = 3
inv = 2
nand3 = 4
nand4 = 5
and2 = 4
aoi21 = 4
oai21 = 4

Exercise to get the cost

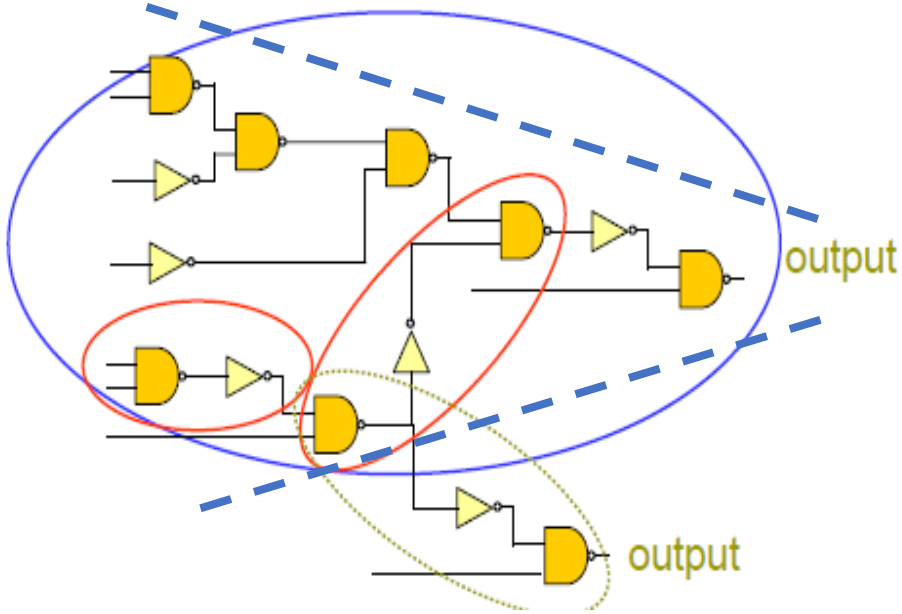
Partition Affects Results

- Partition subject DAG into trees
  - Trivial partition: : break the graph at all multiple-fanout points
    - no duplication or overlap in the resulting trees
    - drawback - sometimes results in many small trees



Leads to 3 trees

- Partition subject DAG into trees
  - Single-cone partition: : from a single output, form a large tree back to the primary inputs
    - map successive outputs until they hit match output formed from mapping previous primary outputs
      - Duplicates some logic (where trees overlap)
      - Produces much larger trees, potentially better area results



How Logic Functions Described For Pattern Graph

Syntax

```
function : "Boolean expression" ;
```

```
pin (Y) {
  direction : output
  function : "(A * B)'"
  timing() {...}
}
pin (Z){
  direction : output
  function : "(A * B)"
  timing() {...}
}
function : "A S + B S'" ;
function : "A & S | B & !S" ;
function : "(A * S) + (B * S')" ;
```

- From Liberty Timing Reference

Min-Delay Technology Mapping

- For trees:
  - identical to min-area covering
  - use optimal delay values within the dynamic programming paradigm
- For DAGs:
  - if delay does not depend on number of fanouts:
    use dynamic programming as presented for trees
  - leads to optimal solution in polynomial time
    - Assume logic replication is okay
- Combined objective
  - e.g. apply delay as first criteria, then area as second
  - combine with static timing analysis to focus on critical paths

Digital Timing Models

- CCS: composite current source
- ECSM: effective current source model
- NLDM: non-linear delay model

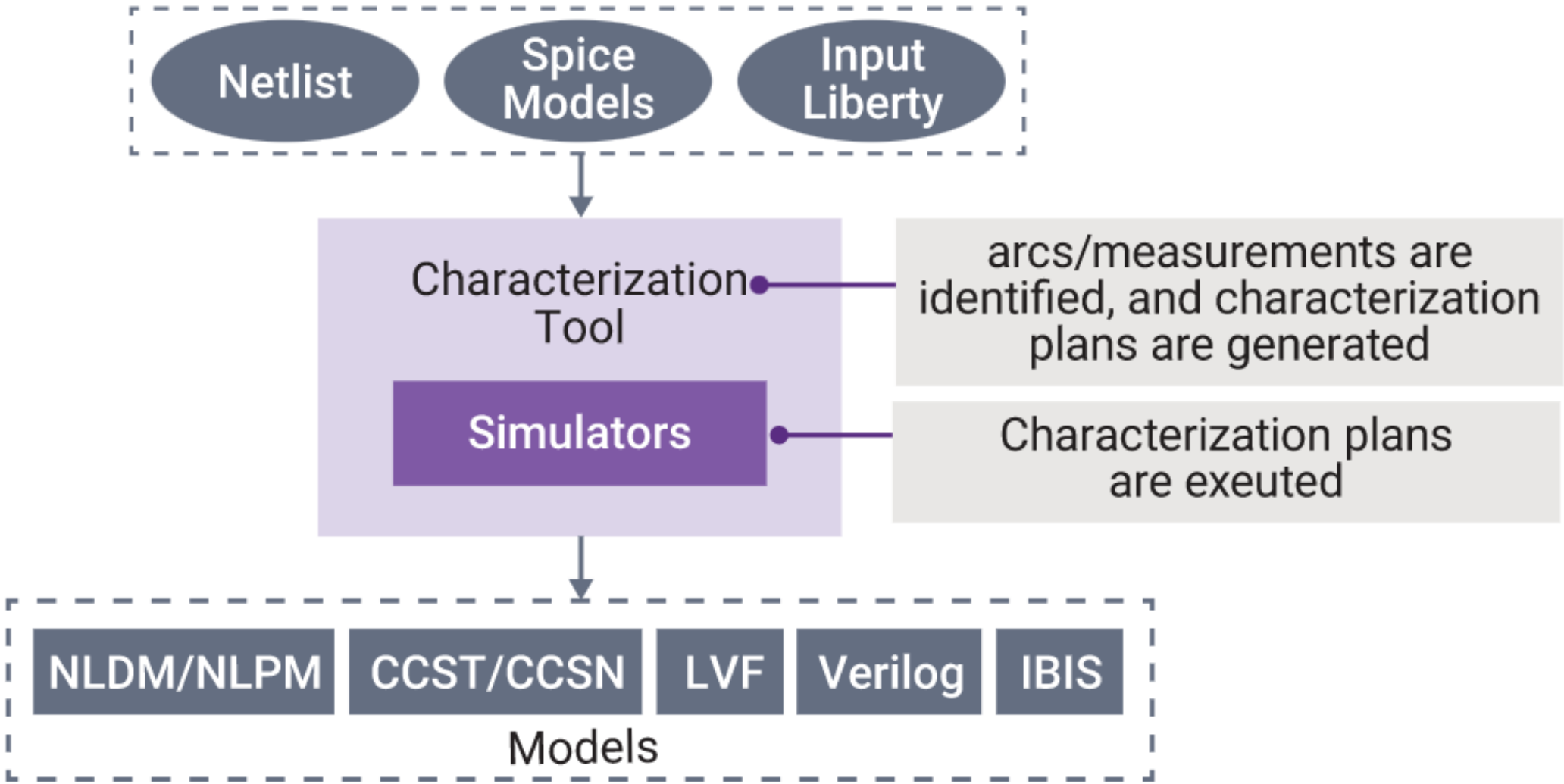
- Many corners (TypicalT, SlowS, FastF)
- On chip variation (OCV)
- Liberty variation format (LVF)

```
/* Define template of 2D polynomial */
poly_template(cell_template) {
  variables(input_net_transition, total_output_net_capacitance) ;
  variable_1_range(0.0, 1.5) ;
  variable_2_range(0.0, 4.0) ;
}

 /* Define template of size 2 x 2*/
 lu_table_template(cell_template) {
   variable_1 : input_net_transition;
   variable_2 : total_output_net_capacitance;
   index_1 ("0.0, 1.5");
   index_2 ("0.0, 4.0");
 }

pin(my_outpin) {
  direction : output;
  timing() {
      related_pin : b;
      timing_sense : non_unate;
      mode(rw, read);
      cell_rise(delay3x3) {
        values("1.1, 1.2, 1.3", "2.0, 3.0, 4.0", "2.5, 3.5, 4.5");
      }
      rise_transition(delay3x3) {
        values("1.0, 1.1, 1.2", "1.5, 1.8, 2.0", "2.5, 3.0, 3.5");
      }
      cell_fall(delay3x3) {
        values("1.1, 1.2, 1.3", "2.0, 3.0, 4.0", "2.5, 3.5, 4.5");
      }
      fall_transition(delay3x3) {
        values("1.0, 1.1, 1.2", "1.5, 1.8, 2.0", "2.5, 3.0, 3.5");
      }
  }
}
```

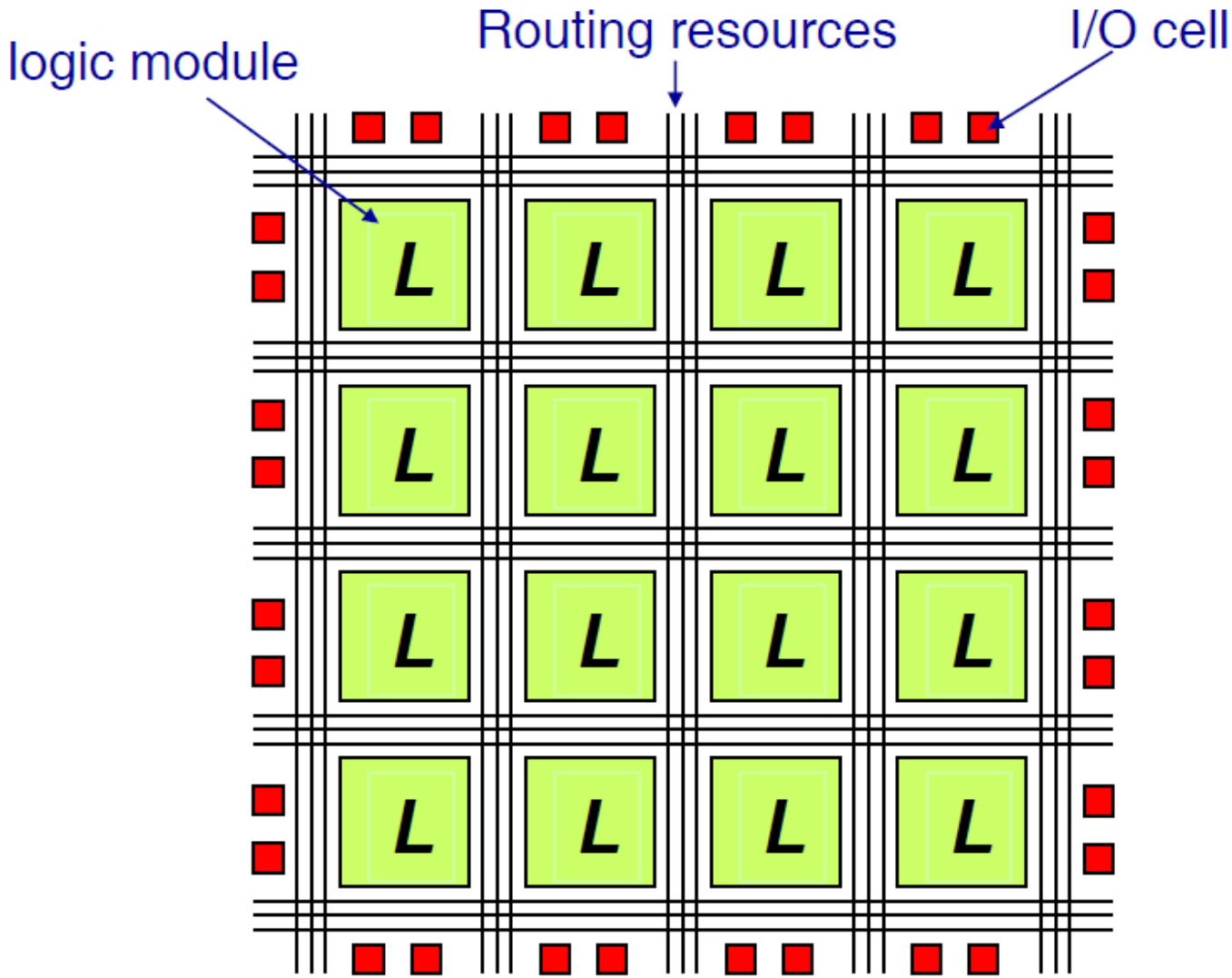
SYNOPSYS®


Netlist
Spice Models
Input Liberty
Characterization Tool
arcs/measurements are identified, and characterization plans are generated
Simulators
Characterization plans are exeuted
NLDM/NLPM
CCST/CCSN
LVF
Verilog
IBIS
Models

- Technology mapping for
  - Area
  - Timing
  - FPGA
  - Power

- But how about tech mapping for area first? Then, optimize for timing, for power?
  - Which way is better?

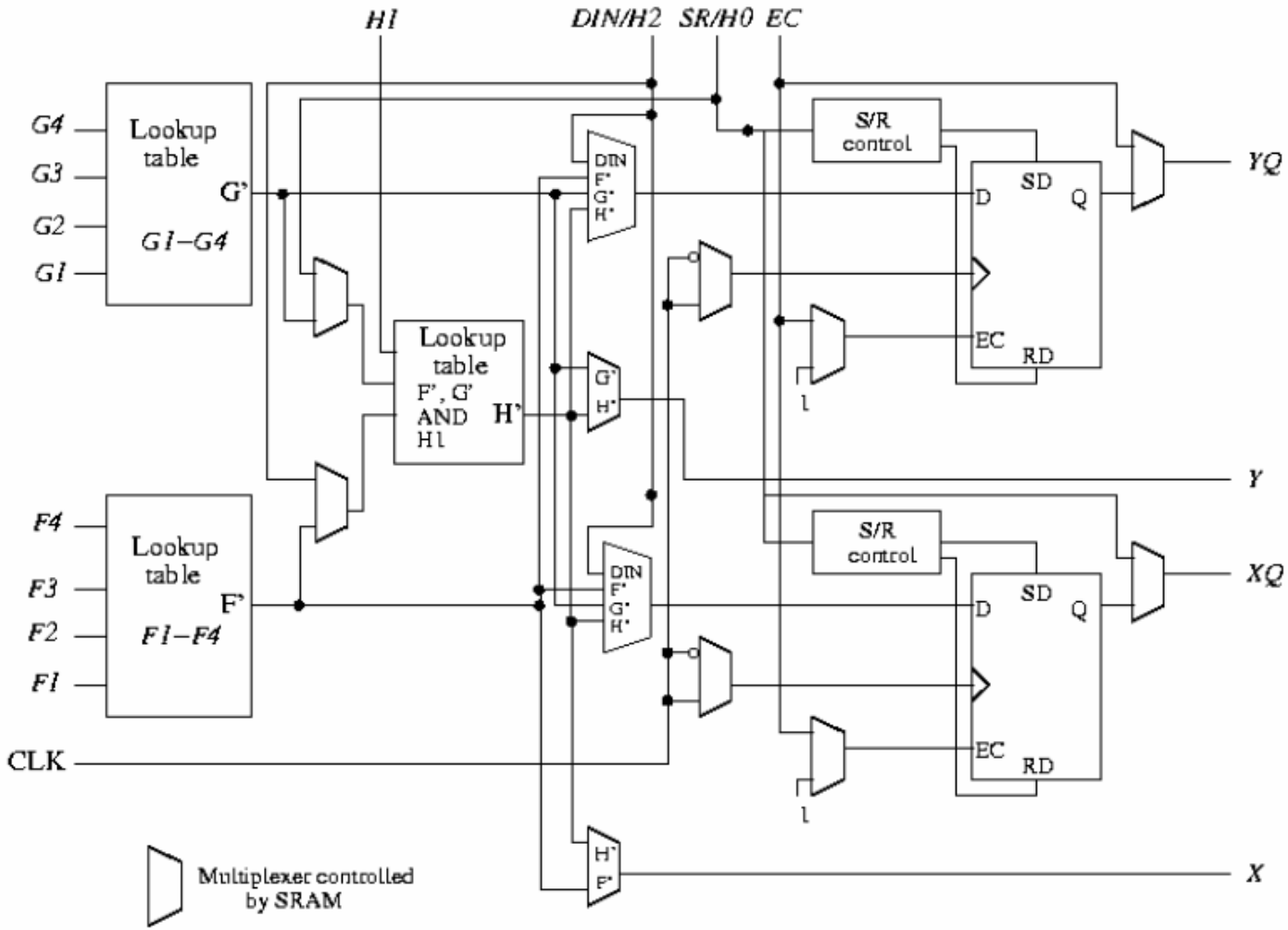
Conceptual FPGA Architecture



Logic modules + Routing resources + I/O cells = FPGAs

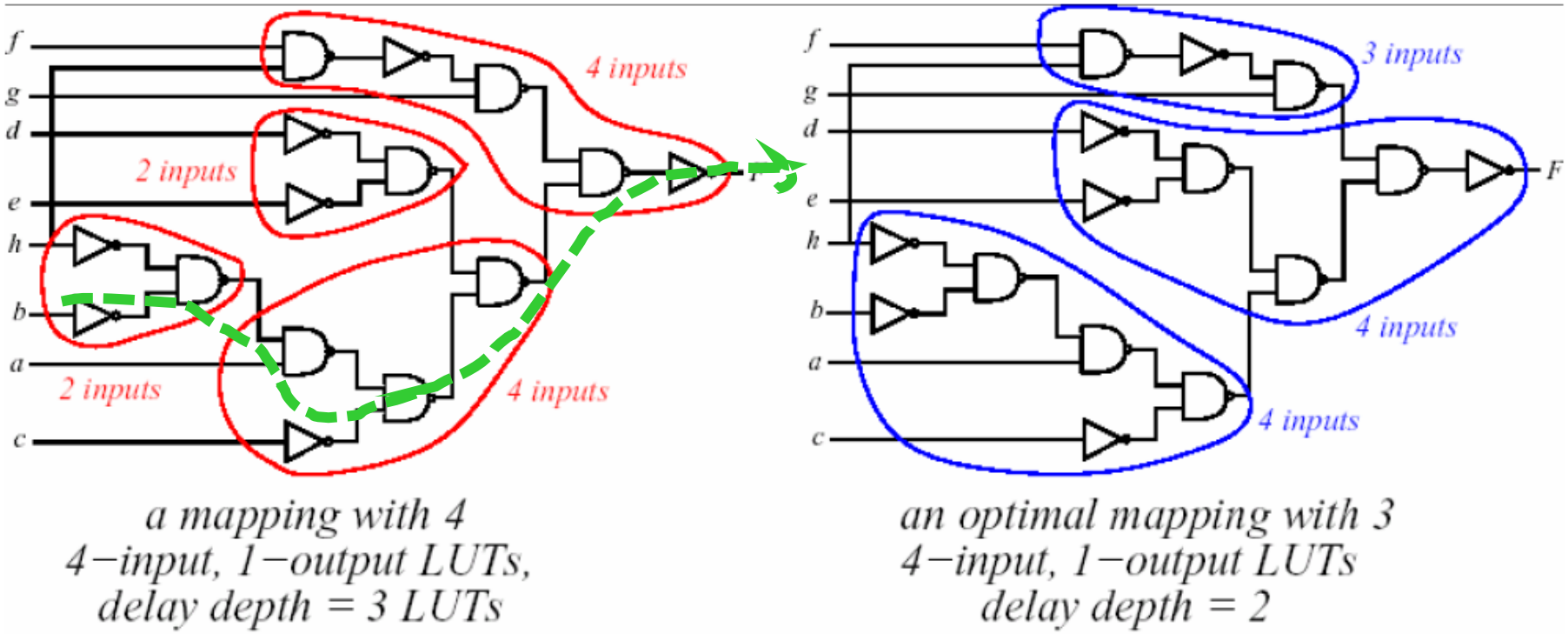
Xilinx XC4000 FPGA Logic Module Architecture

- Each contains two 4-input LUTs, one 3-input LUT, and two DFFs.
- Can implement any 2 functions of up to 4 variables, one function of up to 5 variables, or selected functions of up to 9 variables.



Lookup Table-Based Technology Mapping

- A *k*-input LUT (*k*-LUT) can implement any function of up to *k* inputs.



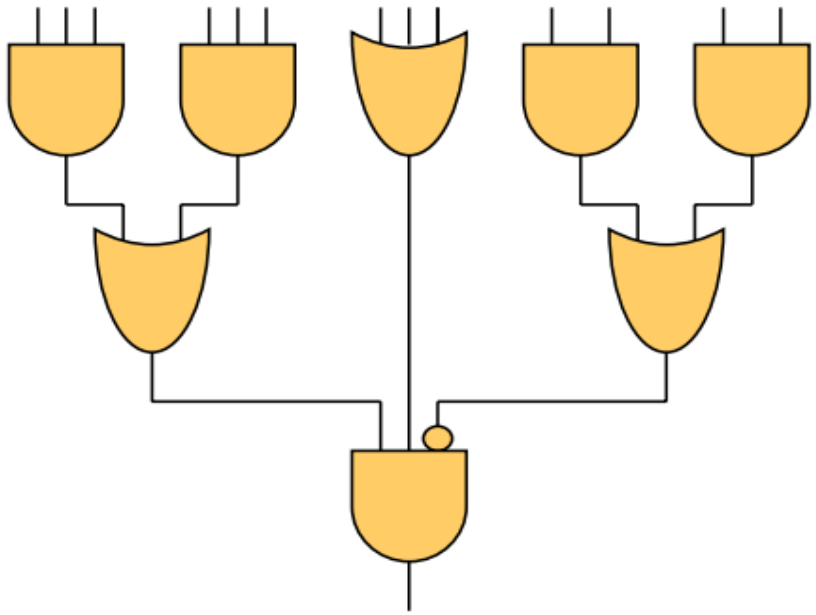
a mapping with 4 4−input, 1−output LUTs, delay depth = 3 LUTs

an optimal mapping with 3 4−input, 1−output LUTs delay depth = 2

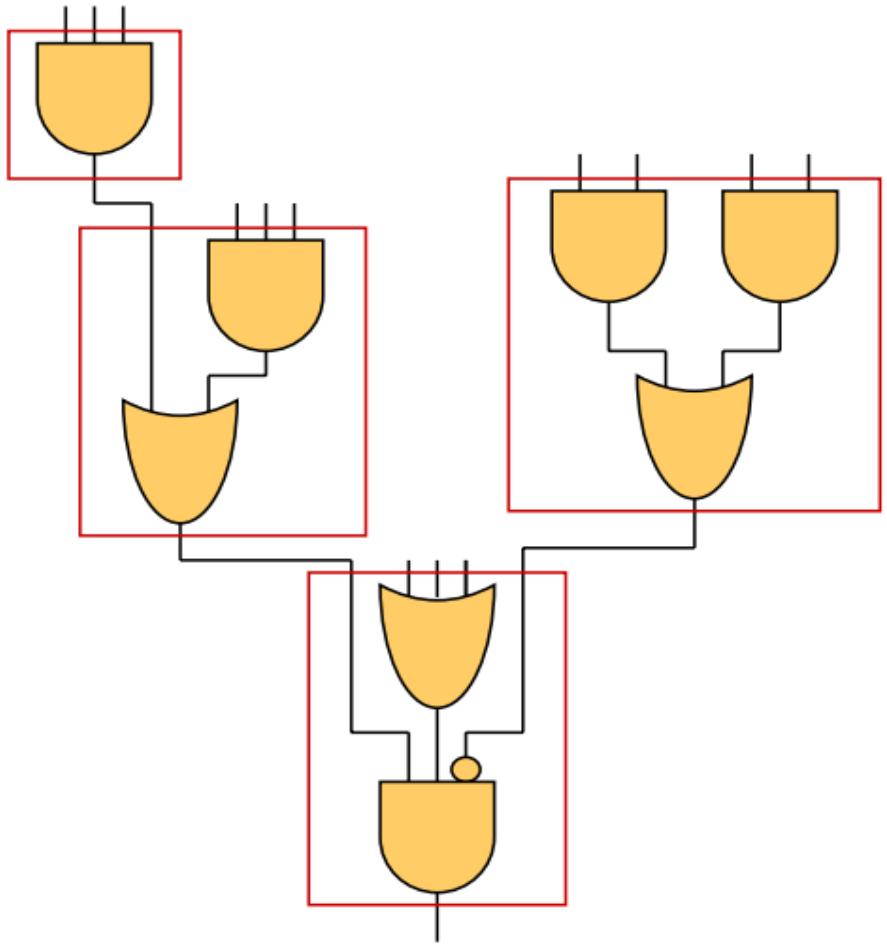
Delay path shown in green

Technology Mapping for FPGAs

- **LUT-based FPGA**
  - **Mapping**



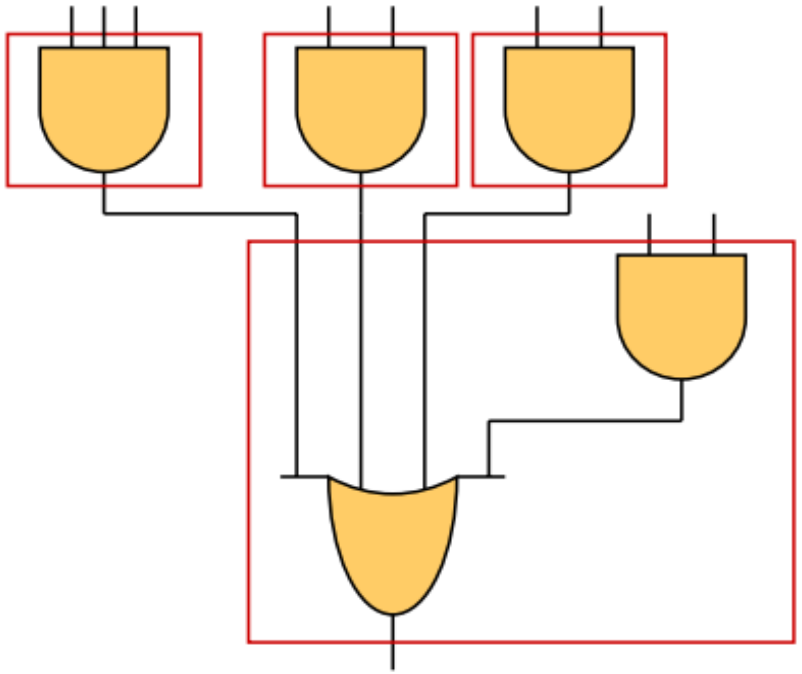
Boolean network



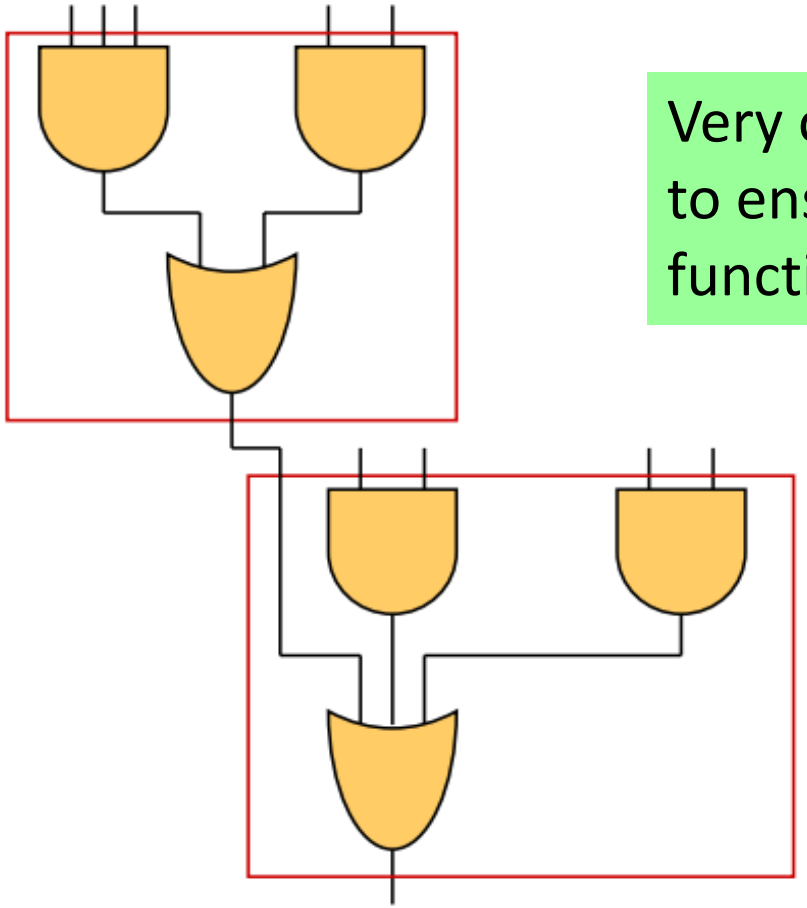
Mapping to 5-input LUTs

– Decomposition

5-inputs LUTs



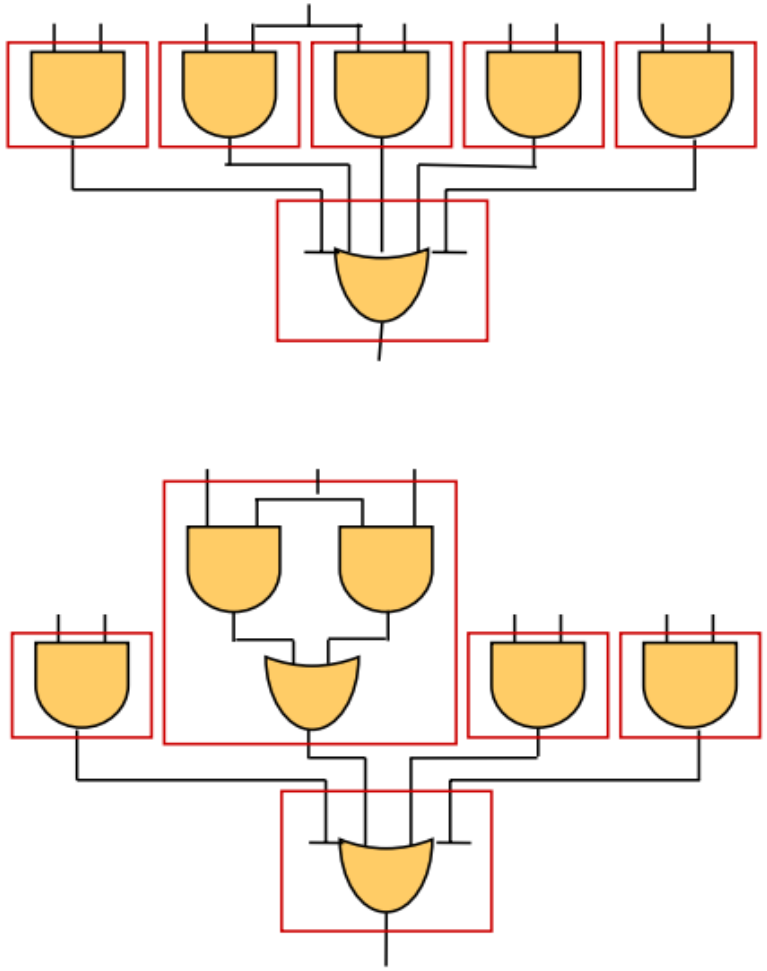
Without decomposition
4 LUTs



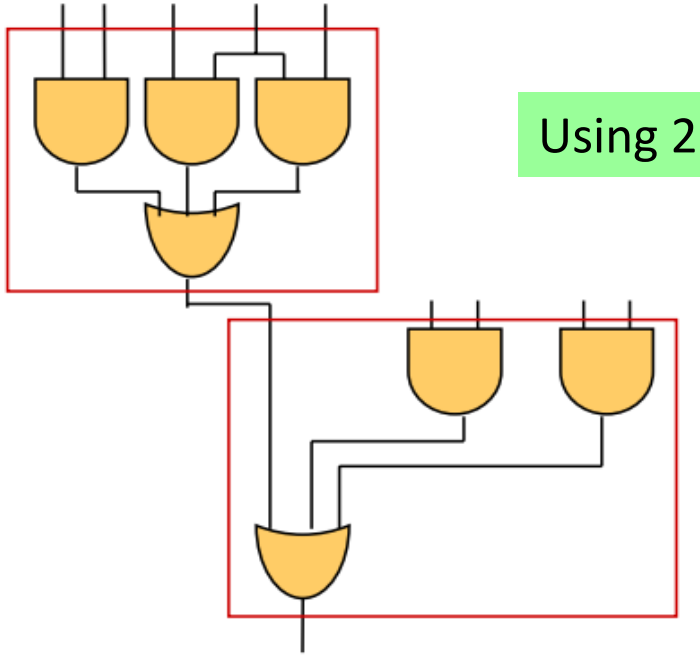
With decomposition
2 LUTs

Very often we need to ensure two logic functions are same

– Local reconvergent paths

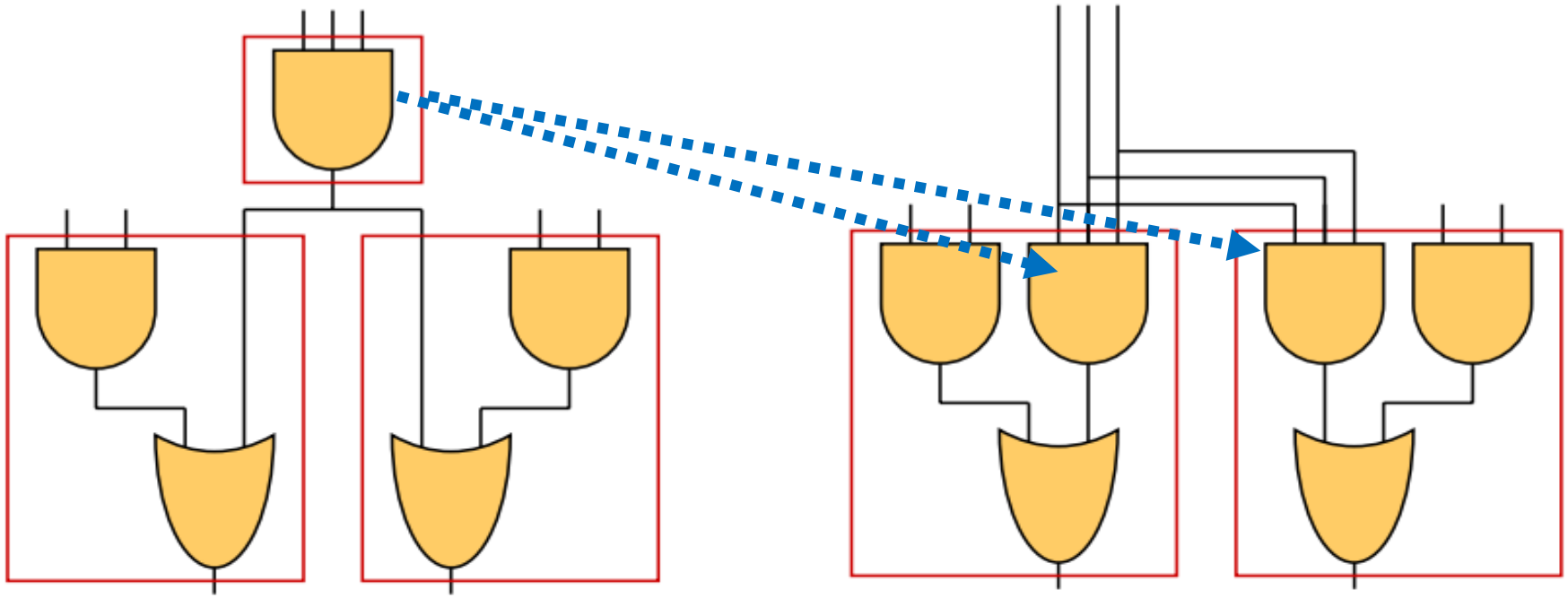


Reconvergent paths
realized within one LUT



Using 2 LUTs

– Replication of logic

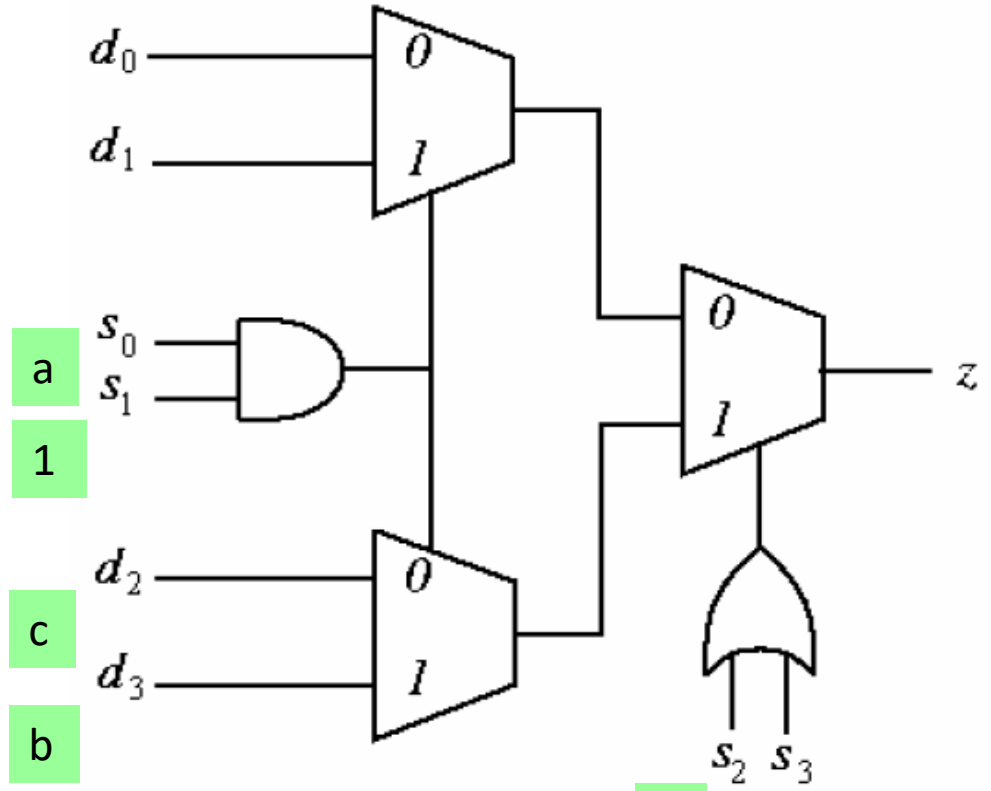


Without replicated logic
3 LUTs

With replicated logic
2 LUTs

Multiplexer-Based Technology Mapping

- To implement $f = ab + \bar{a}c$, set $d_0 = d_1 = s_3 = x$, $d_2 = c$, $d_3 = b$, $s_0 = a$, $s_1 = s_2 = 1$.

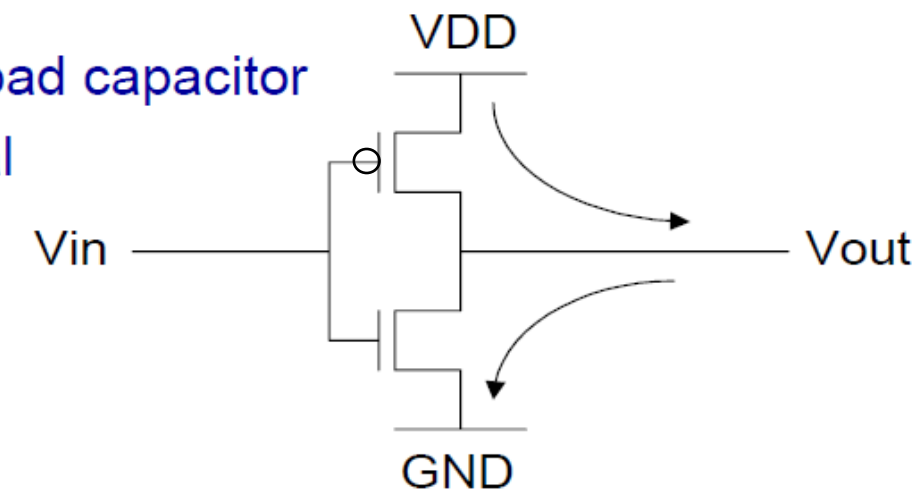


More specific to the architecture

$$z = \overline{(s_3 + s_2)}\,\overline{s_1 s_0}\,d_0 + \overline{(s_3 + s_2)}\,s_1 s_0 d_1 + (s_3 + s_2)\,\overline{s_1 s_0}\,d_2 + (s_3 + s_2)\,s_1 s_0 d_3.$$

Power Dissipation

- Leakage power
  - Static dissipation due to leakage current
  - Typically a smaller value compared to other power dissipation
  - Gets larger and larger in deep-submicron process
- Short-circuit power
  - Due to the short-circuit current when both PMOS and NMOS are open during transition
  - Typically a smaller value compared to dynamic power
- Dynamic power
  - Charge and discharge of a load capacitor
  - Usually the major part of total power consumption



Power Dissipation Model

$$P = \frac{1}{2} C V_{dd}^2 f$$

- In traditional application, ***dynamic power*** is used to represent total power dissipation

  P: the power dissipation for a gate

  C: the load capacitance

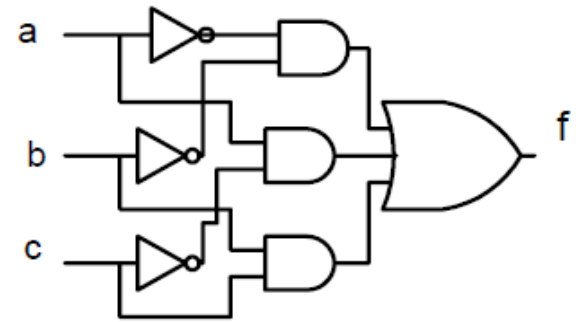
  V_{dd}: the supply voltage

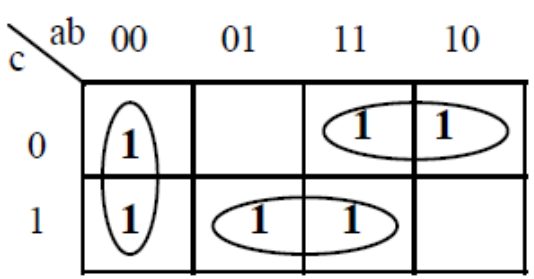
  f: the transition frequency

- To obtain the power dissipation of the circuit, we need
  - node capacitance of each node (obtained from layout)
  - transition frequency of each node (obtained by computation)

Logic Optimization for Low Power

- Consider an example:

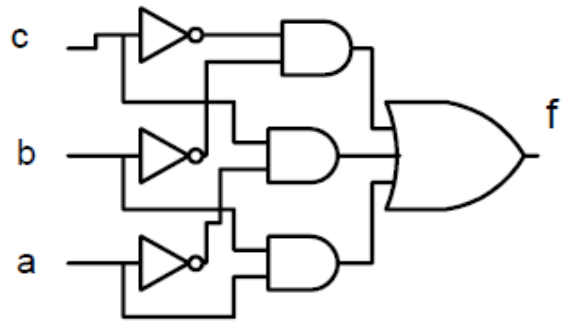


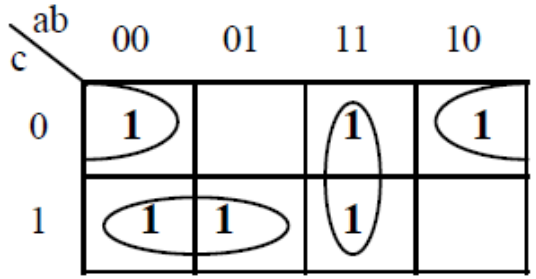


f = a'b' + ac' + bc

P = 108.7 μW

(a)





f = b'c' + a'c + ab

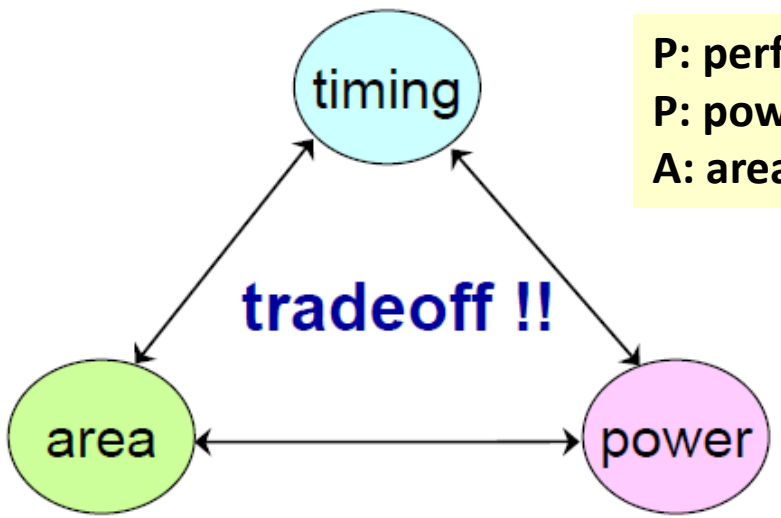
P = 115.5 μW

(b)

- Different choices of the covers may result in different power consumption

Low Power Is a 1st-Order Cost Metric

- Typically, the objective of logic minimization is to minimize
  - the number of product terms
  - the number of literals in the input parts
  - the number of literals in the output parts
- For low power synthesis, the power dissipation has to be added into the cost function for logic optimization

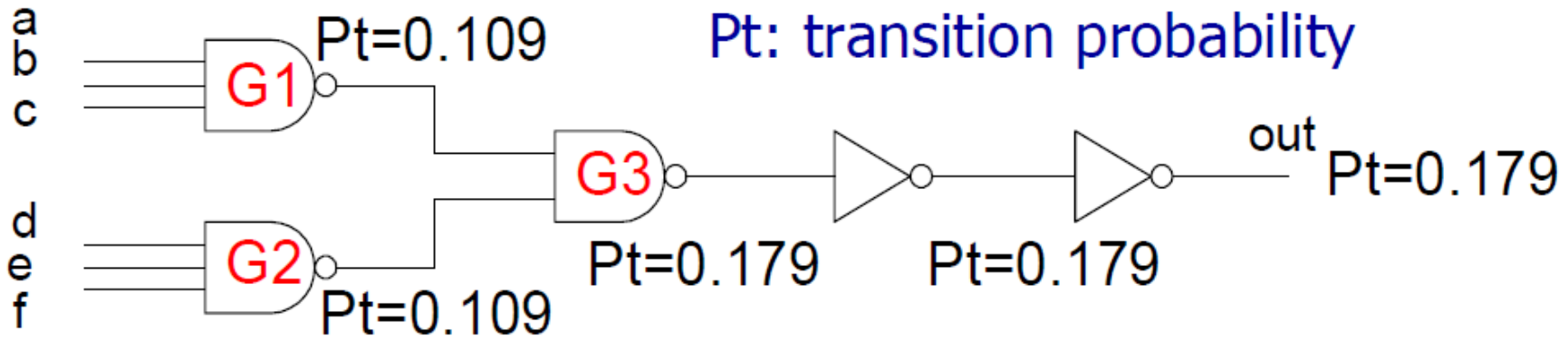


P: performance
P: power
A: area

Adapted from: Prof. Y.-W. Chang's slides

Technology Mapping for Low Power

$$P = \frac{1}{2} C V_{dd}^2 f$$

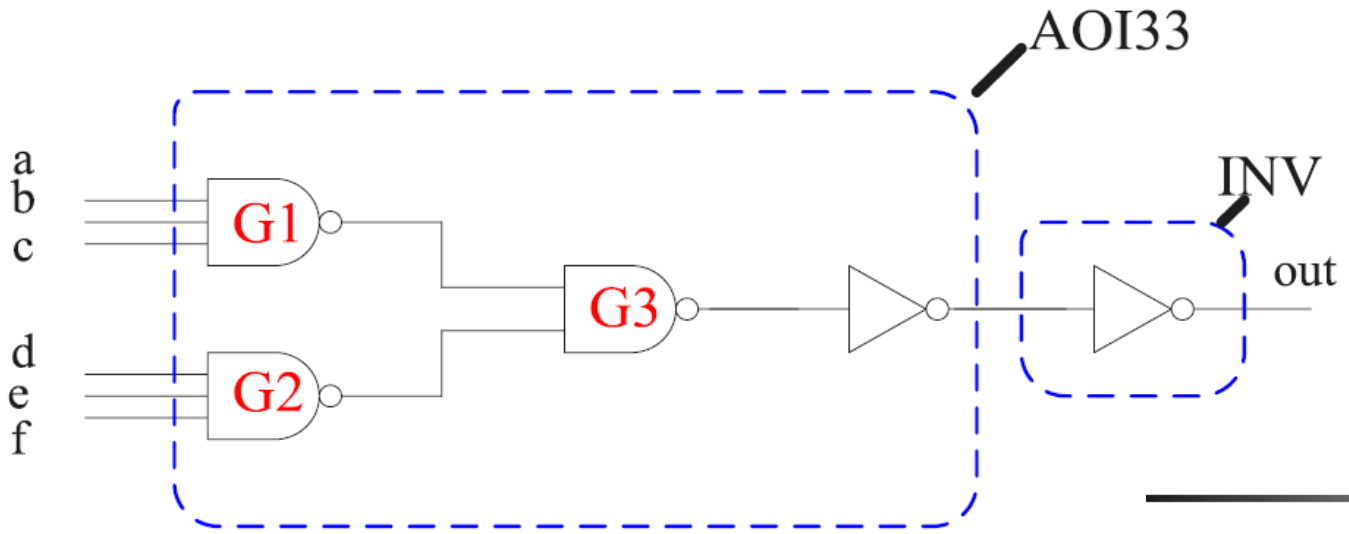


(a) Circuit to be mapped

Gate Type	Area	Intrinsic Cap.	Input Load
INV	928	0.1029	0.0514
NAND2	1392	0.1421	0.0747
NAND3	1856	0.1768	0.0868
AOI33	3248	0.3526	0.1063

(b) Characteristics of Library

Minimum-Area Mapping

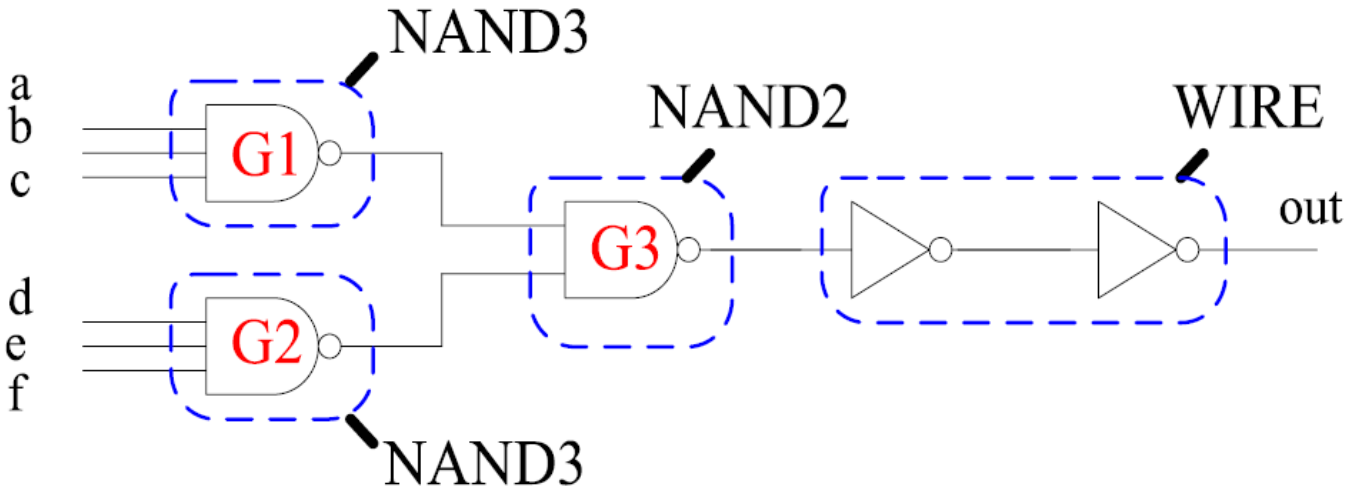


Area Cost: 4176
Power Cost: 0.0907

$$P = \frac{1}{2} C V_{dd}^2 f$$

Let's think about how the power cost is calculated

Minimum-Power Mapping

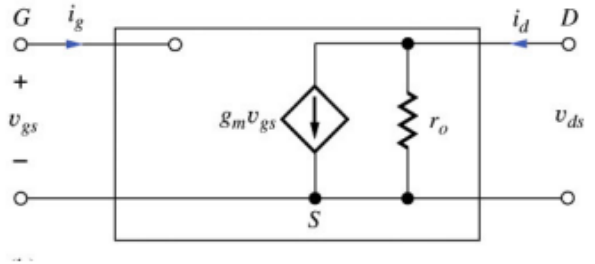


Area Cost: 5104
Power Cost: 0.0803

My Thought About Power Cost

- First, refer to the small signal model for MOSFET

Small-Signal Operation
MOSFET Small-Signal Model - Summary



- **Since gate is insulated from channel by gate-oxide input resistance of transistor is infinite.**
- **Small-signal parameters are controlled by the Q-point.**
- **For the same operating point, MOSFET has lower transconductance and an output resistance that is similar to the BJT.**

$$I_G = 0$$

$$I_D = \frac{K_n}{2}(V_{GS} - V_{TN})^2(1 + \lambda V_{DS})$$

Transconductance:

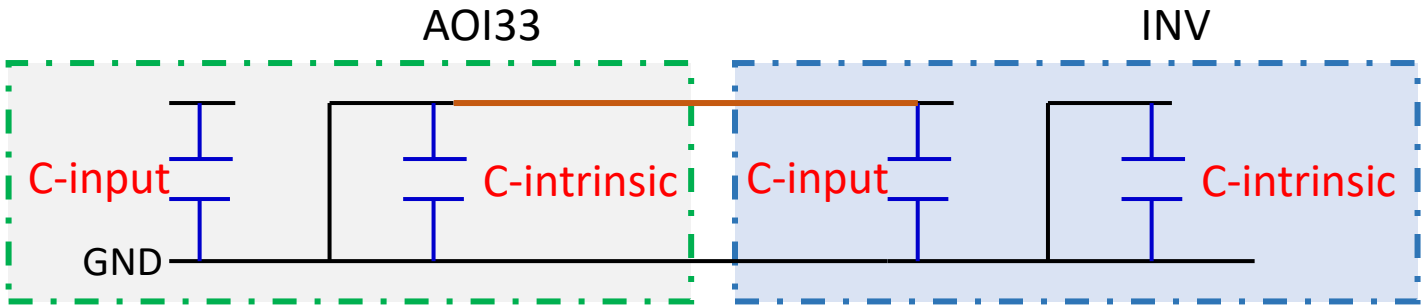
$$g_m = \frac{2I_D}{V_{GS} - V_{TN}} = \sqrt{2K_n I_D}$$

Output resistance:

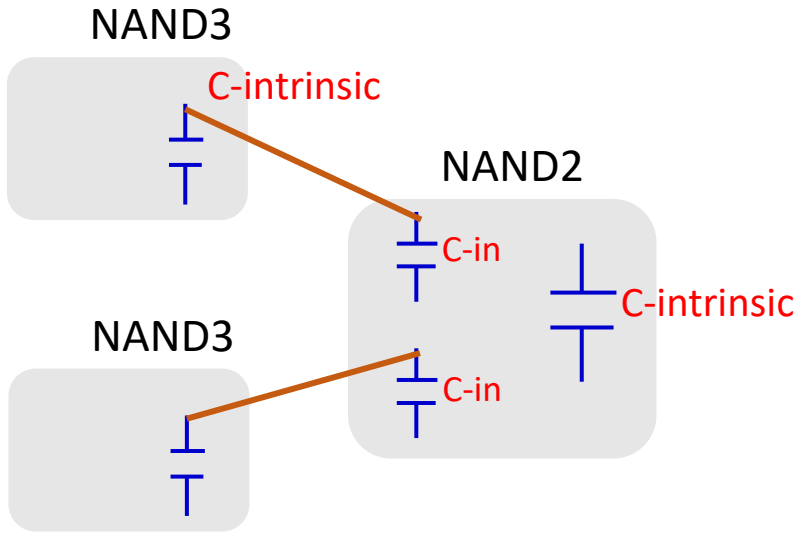
$$r_o = \frac{1}{g_o} = \frac{1 + \lambda V_{DS}}{\lambda I_D} \cong \frac{1}{\lambda I_D}$$

Amplification factor for $\lambda V_{DS} \ll 1$:

$$\mu_f = g_m r_o = \frac{1 + \lambda V_{DS}}{\lambda I_D} \cong \frac{1}{\lambda}\sqrt{\frac{2K_n}{I_D}}$$



AOI33		f	P	totP
intrinsic	0.3526			
INV				
inputC	0.0514	0.179		
totC	0.404		0.072316	
INV				
intrinsic	0.1029	0.179	0.018419	0.09074



NAND3		f	P	totP
intrinsic	0.1768	0.109		
NAND2				
inputC	0.0747			
totC	0.2515		0.027414	
the other NAND3 + NAND2				
			0.027414	
NAND2		f	P	
intrinsic	0.1421	0.179	0.025436	0.080263

But how about tech mapping for area first? Then, optimize for timing, for power? Which way is better?

https://www.wsj.com/articles/the-chips-that-rebooted-the-mac-11650081649?mod=e2li

The Chips That Rebooted the Mac

Any thoughts about how to reduce power consumption?

https://www.youtube.com/watch?v=UdhWvg5mycY

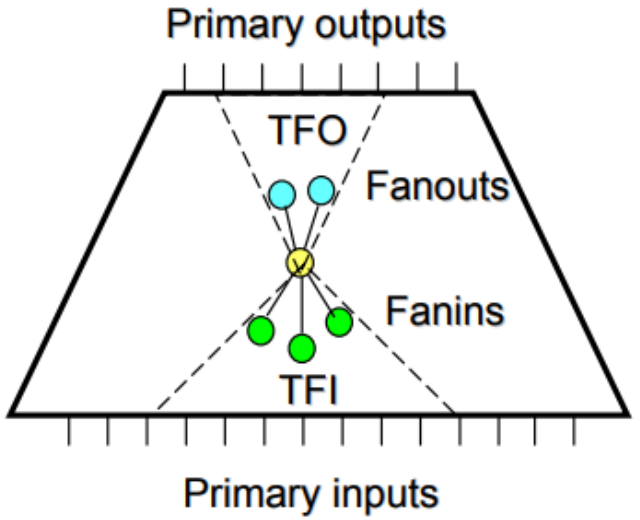
CNBC罕見走進蘋果晶片測試實驗室，一睹由台積電代工的M3晶片，如何在蘋果加州實驗室進行檢測？ ... 以快轉在3:21和7:05左右參觀

http://cc.ee.ntu.edu.tw/~jhjiang/instruction/courses/fall10-lsv/lec07_4p.pdf

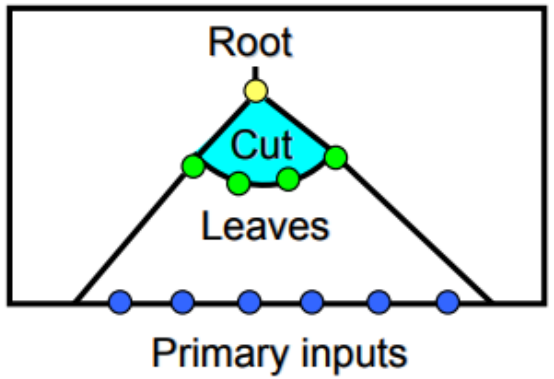
Terminology

Materials used by J-H Roland Jiang. Boolean matching needs it.

- Logic network
  - Primary inputs/outputs (PIs/POs)
  - Logic nodes
  - Fanins/fanouts
  - Transitive fanin/fanout cone (TFI/TFO)



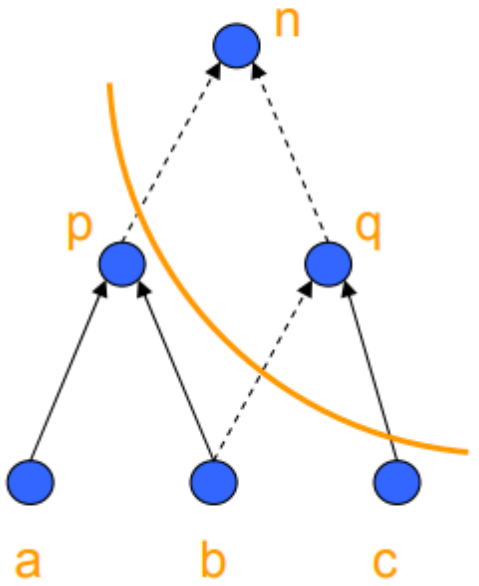
- Structural cut of a node
  - Cut is a boundary in the network separating the node from the PIs
  - Boundary nodes are the leaves
  - The node is the root
  - K-feasible cut has K or less leaves
  - Function of the cut is function of the root in terms of the leaves



Structural Cuts in AIG

A cut of a node *n* is a set of nodes in transitive fanin such that every path from the node to PIs is blocked by nodes in the cut.

A k-feasible cut has no more than k leaves.



The set {pbc} is a 3-feasible cut of node n. (It is also a 4-feasible cut.)

k-feasible cuts are important in LUT mapping because the logic between root n and the cut leaves {pbc} can be replaced by a *k*-LUT.

Example. In the AIG of Figure 1, $\{n_2\}$, $\{n_4, n_5\}$, $\{n_4, x_2, x_3\}$ $\{n_5, x_1, x_2\}$, $\{x_1, x_2, x_3\}$ are all the 3-feasible cuts of n_2.

We compute all k-feasible cuts of every node in the network by the simple bottom-up traversal algorithm shown in Figure 2. Although in general a graph may have exponentially many cuts, most test-cases have between 20 and 30 5-feasible cuts per node.

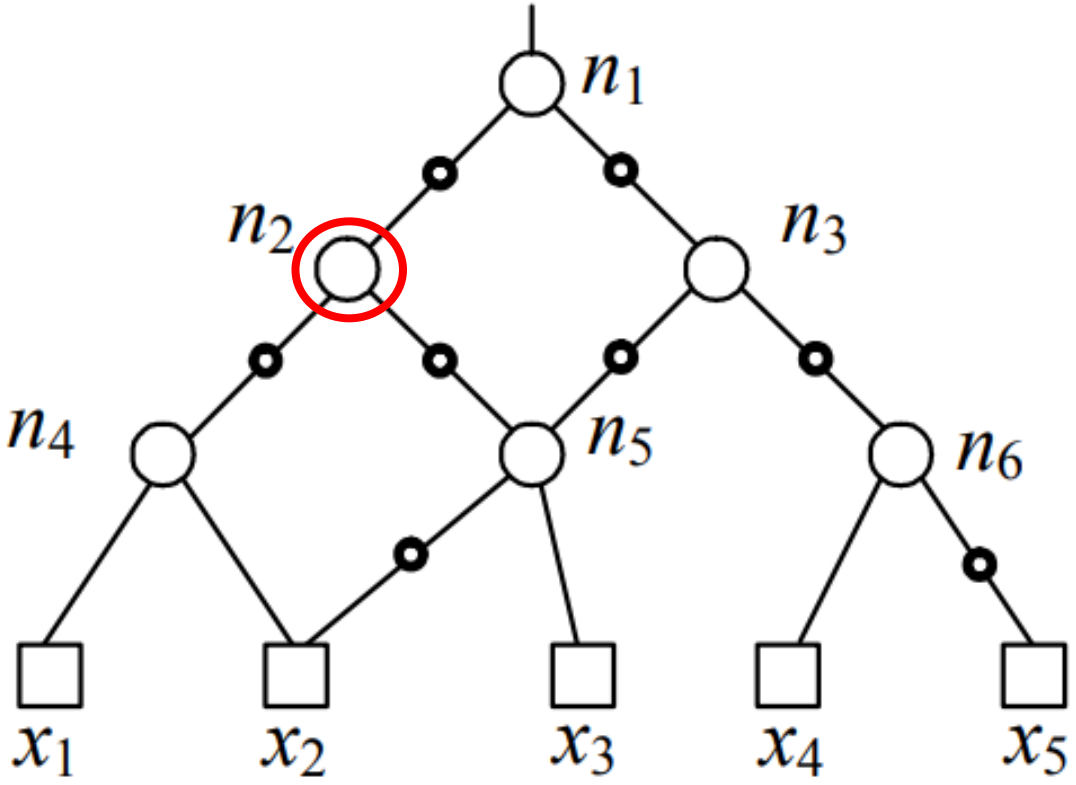
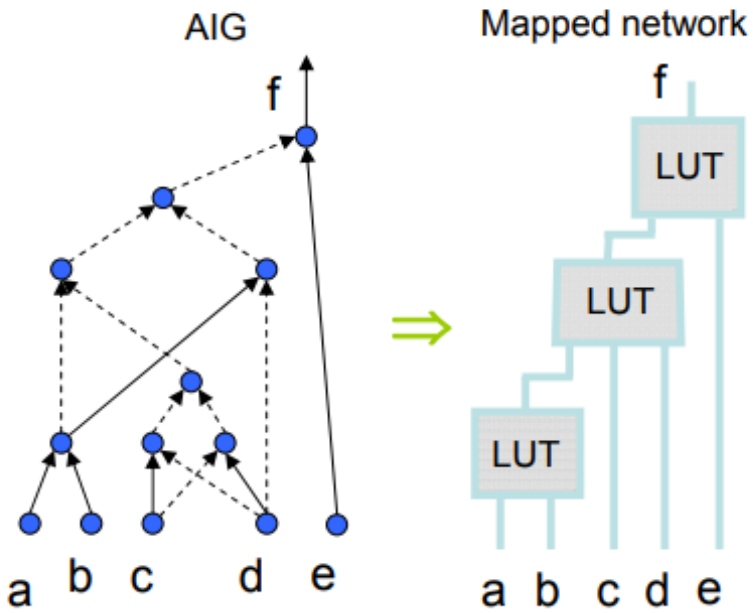
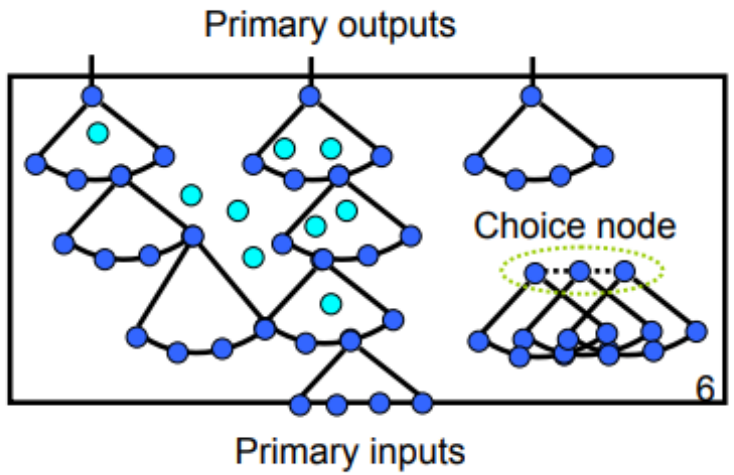


Figure 1: An example of an AIG.

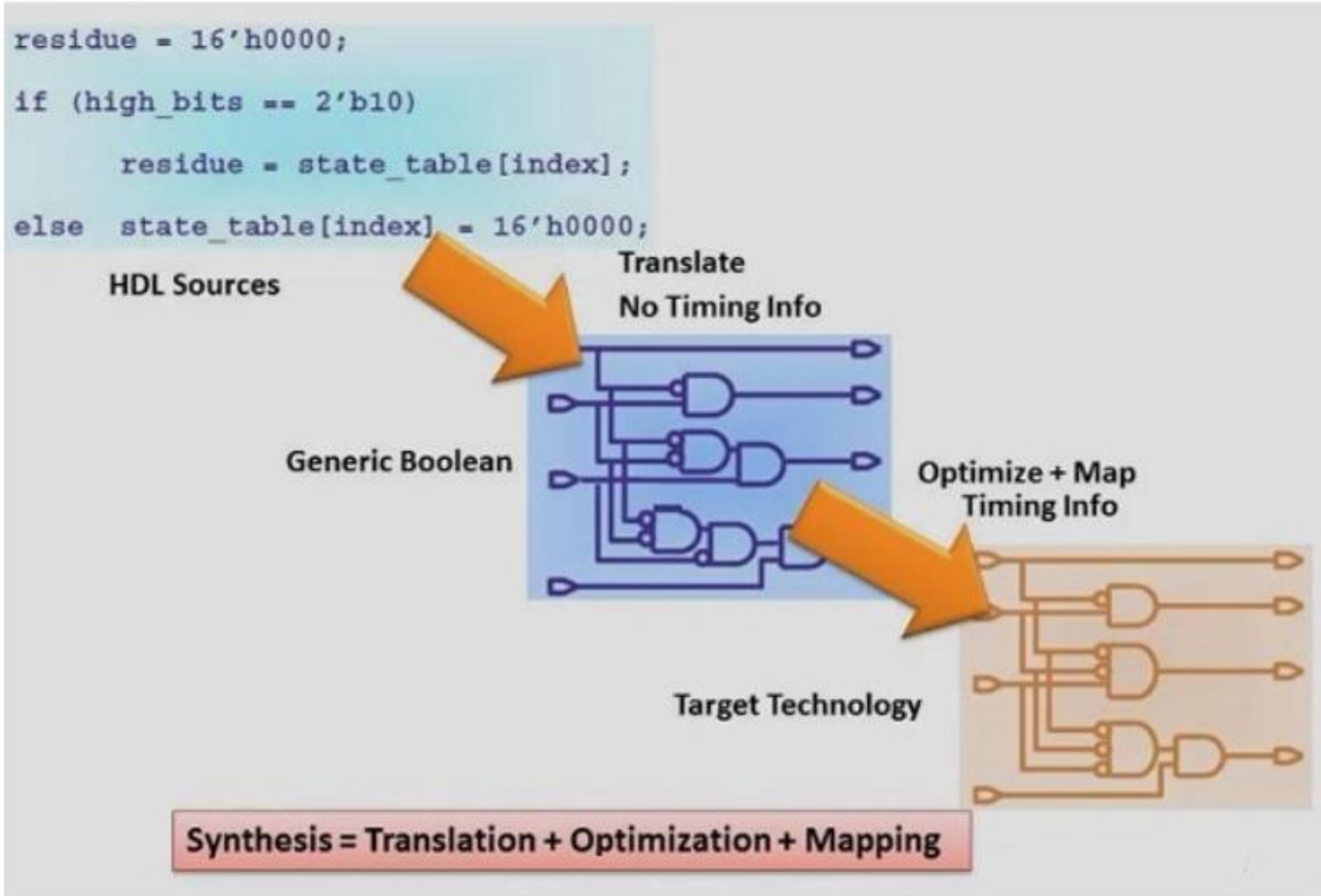
Mapping in a Nutshell

- AIGs reprsent logic functions
  - A good subject graph for mapping
- Technology mapping expresses logic functions to be implemented
  - Uses a description of a technology
- Technology
  - Primitives with delay, area, etc
- Structural mapping
  - Computes a cover of AIG using primitives of the technology
- Cut-based structural mapping
  - Computes cuts for each AIG node
  - Associates each cut with a primitive
  - Selects a cover with a minimum cost
- Structural bias
  - Good mapping cannot be found because of the poor AIG structure
- Overcoming structural bias
  - Need to map over a number of AIG structures (leads to choice nodes)

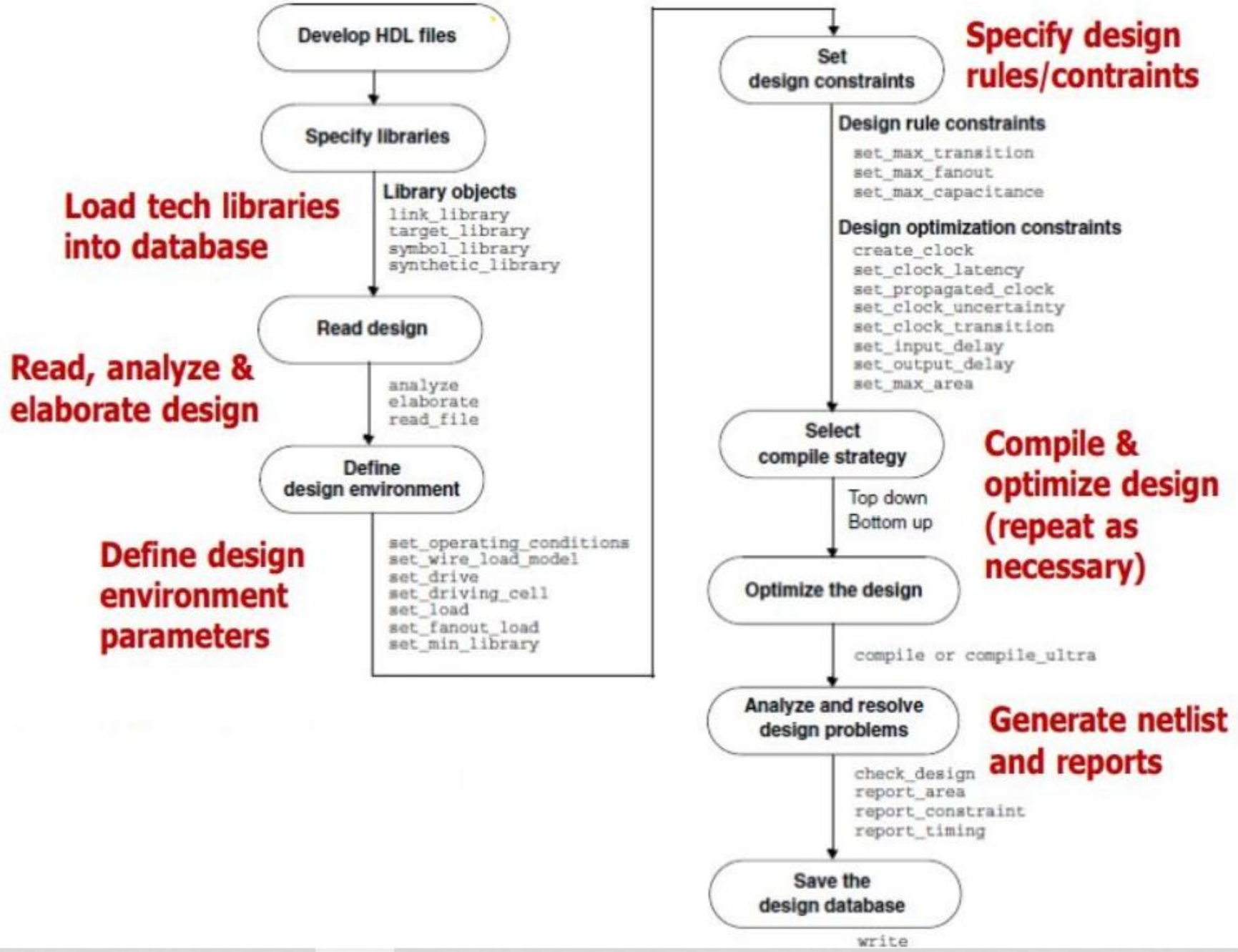




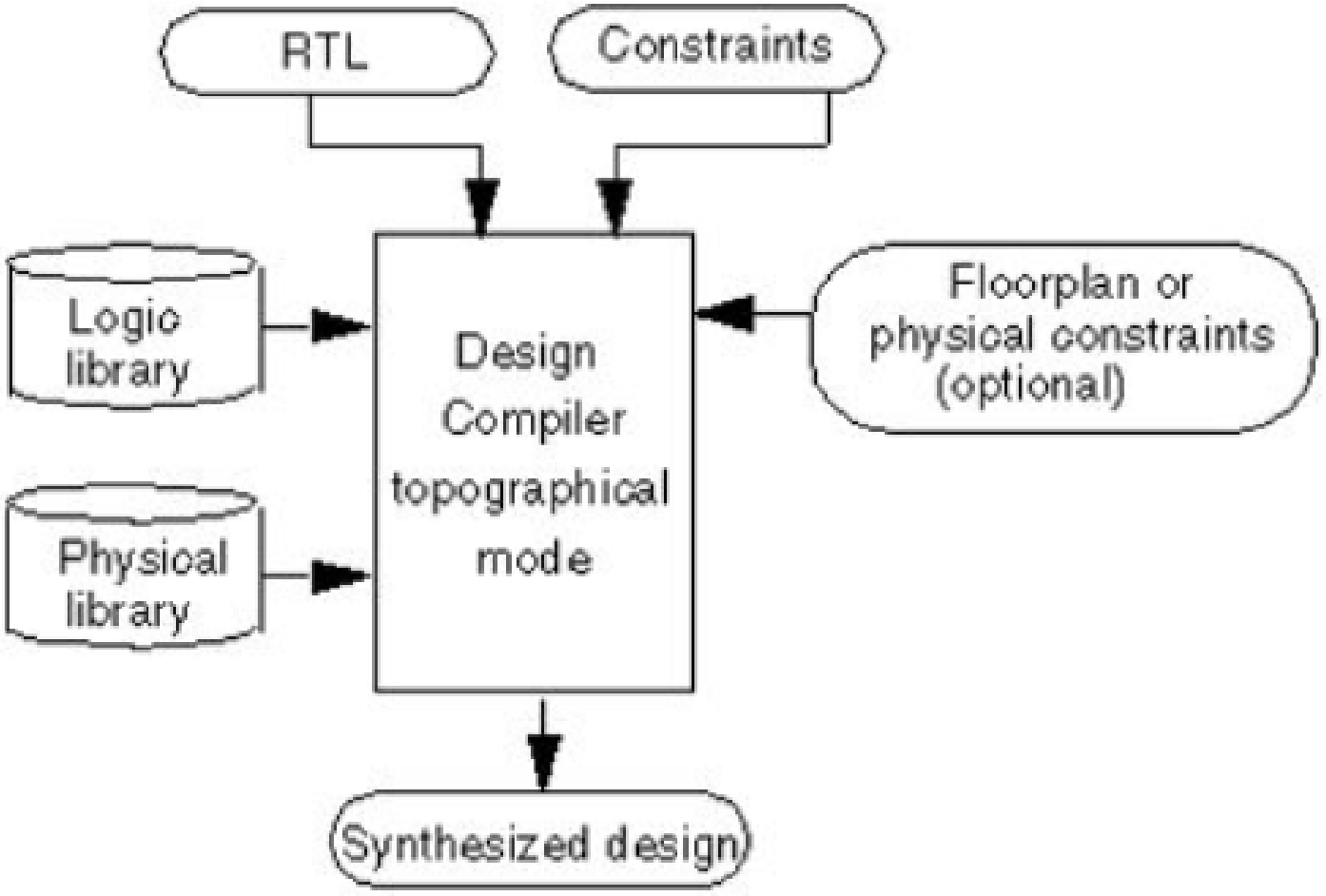
HDL > Generic Boolean logic > Optimize > Map > Netlist



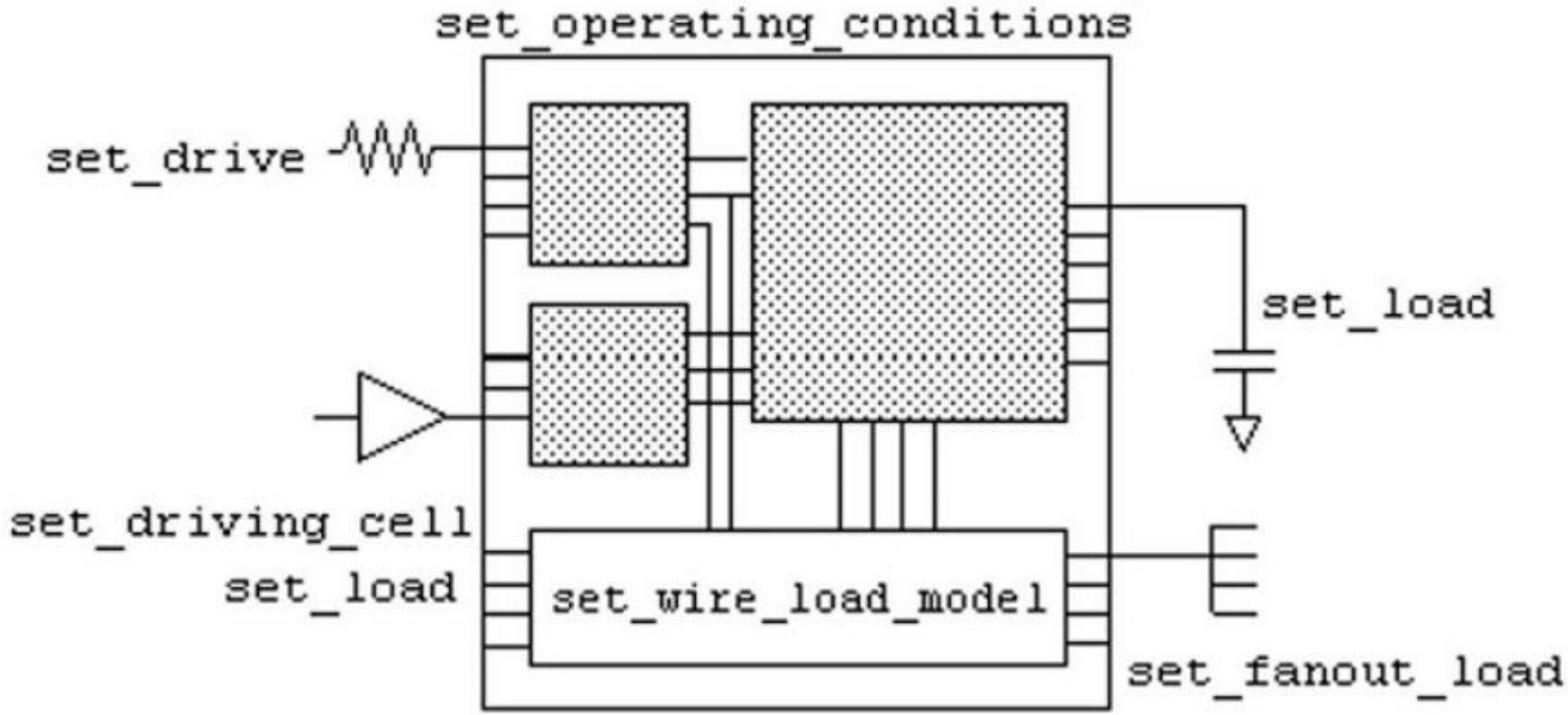
Synopsys Design Compiler



- **Inputs of Synthesis**
  - RTL: HDL Files
  - Libraries
  - Constraints
  - UPF (Power intent for power aware synthesis
- **Ouputs of Synthesis**
  - Netlist
  - Reports (QOR, Area, Timing etc.)
  - UPF: Unified Power Format



Inputs and Outputs in DC topographical mode



Commands Used to Define the Design Environment

Delay Model at Logic Level

- Unit delay model
  - Assign a delay of 1 to *each gate*
- Unit fanout delay model
  - Incorporate an *additional* delay for *each fanout*
- Library delay model
  - Use delay data in the library to provide more accurate delay value
  - May use linear or non-linear (tabular) models

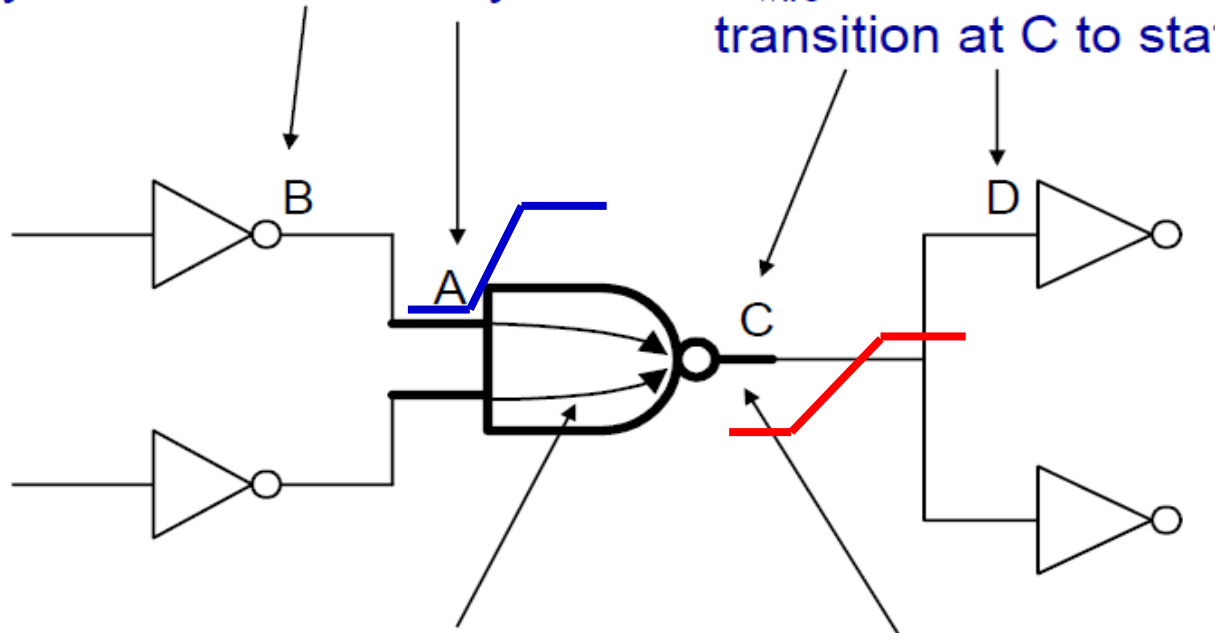
Linear Delay Model

Through Cells

$$\text{Delay} = D_{slope} + D_{intrinsic} + D_{transition} + D_{wire}$$

D_{slope} (Slope delay): delay at input A caused by the transition delay at B

D_{wire} (Wire delay): time from state transition at C to state transition at D



$D_{intrinsic}$ (Intrinsic delay): incurred from cell input to cell output

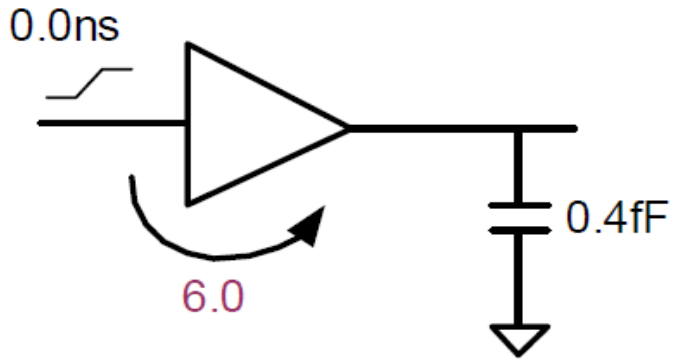
$D_{transition}$ (Transition delay): output pin loading, output pin drive

Timing Library: many spice sims to be run

Adapted from: Prof. Y.-W. Chang's slides

Tabular Delay Model

- Delay values are obtained by a look-up table
  - Two-dimensional table of delays (m by n)
    - W.r.t. input slope (m) and total output capacitance (n)
  - One dimensional table model for output slope (n)
    - W.r.t. total output capacitance (n)
  - Each value in the table is obtained by real measurement



Total Output Load (fF)

Input Transition (ns)	0.2	0.3	**0.4**	0.5
0	3	4.5	**6**	7
0.1	5	8	10.7	13

Cell Delay (ps)

- Can be more precise than the linear delay model
  - table size ↑ ➔ accuracy ↑
- Require more space to store the table

Digital Timing Models

- CCS: composite current source
- ECSM: effective current source model
- NLDM: non-linear delay model

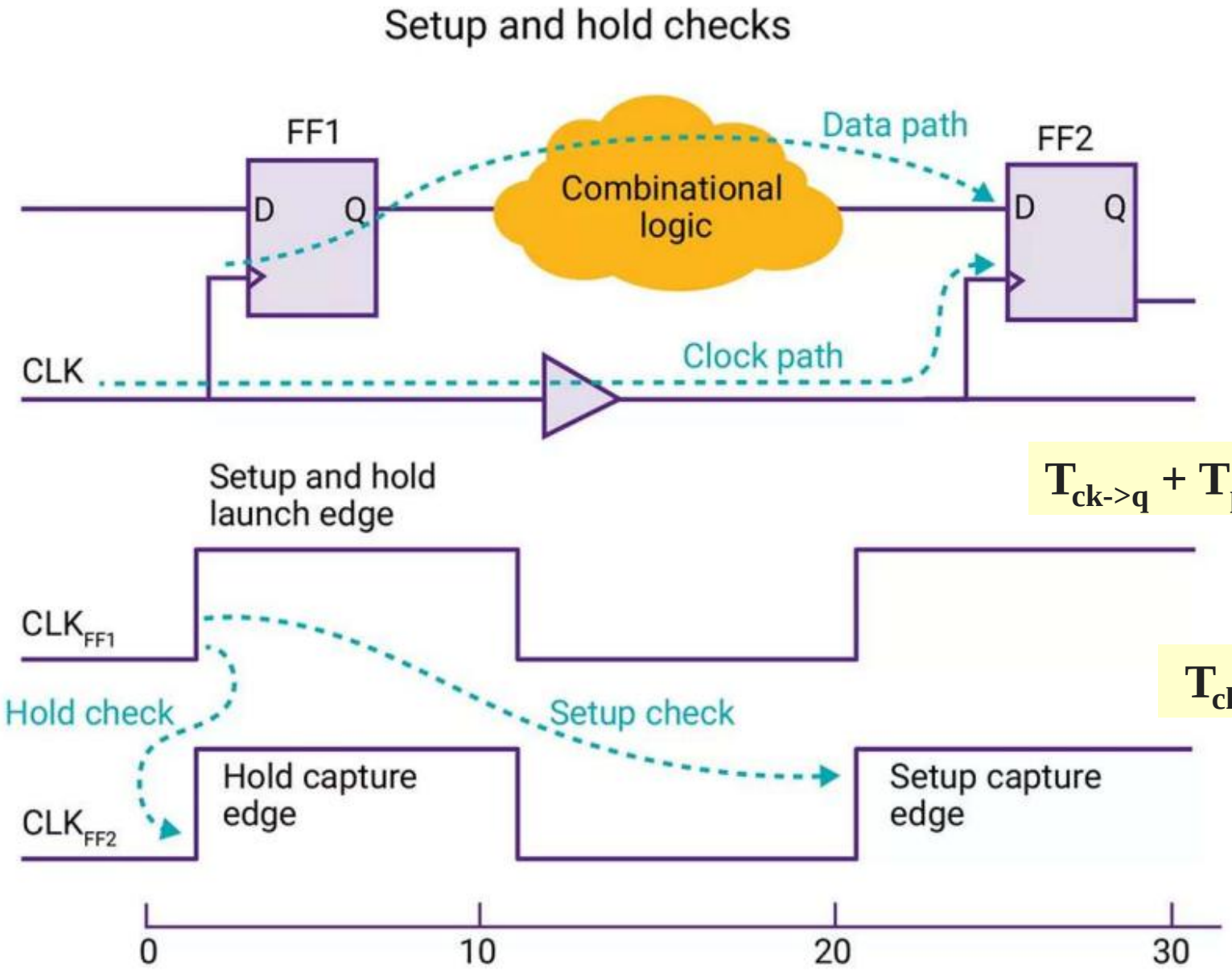
- Many corners (TypicalT, SlowS, FastF)
- On chip variation (OCV)
- Liberty variation format (LVF)

```
/* Define template of 2D polynomial */
poly_template(cell_template) {
  variables(input_net_transition, total_output_net_capacitance) ;
  variable_1_range(0.0, 1.5) ;
  variable_2_range(0.0, 4.0) ;
}

 /* Define template of size 2 x 2*/
 lu_table_template(cell_template) {
   variable_1 : input_net_transition;
   variable_2 : total_output_net_capacitance;
   index_1 ("0.0, 1.5");
   index_2 ("0.0, 4.0");
 }

pin(my_outpin) {
  direction : output;
  timing() {
      related_pin : b;
      timing_sense : non_unate;
      mode(rw, read);
      cell_rise(delay3x3) {
        values("1.1, 1.2, 1.3", "2.0, 3.0, 4.0", "2.5, 3.5, 4.5");
      }
      rise_transition(delay3x3) {
        values("1.0, 1.1, 1.2", "1.5, 1.8, 2.0", "2.5, 3.0, 3.5");
      }
      cell_fall(delay3x3) {
        values("1.1, 1.2, 1.3", "2.0, 3.0, 4.0", "2.5, 3.5, 4.5");
      }
      fall_transition(delay3x3) {
        values("1.0, 1.1, 1.2", "1.5, 1.8, 2.0", "2.5, 3.0, 3.5");
      }
  }
}
```

Setup Time & Hold Time Check



$$T_{ck->q} + T_{prop} + T_{setup} - Tskew < T_{period}$$

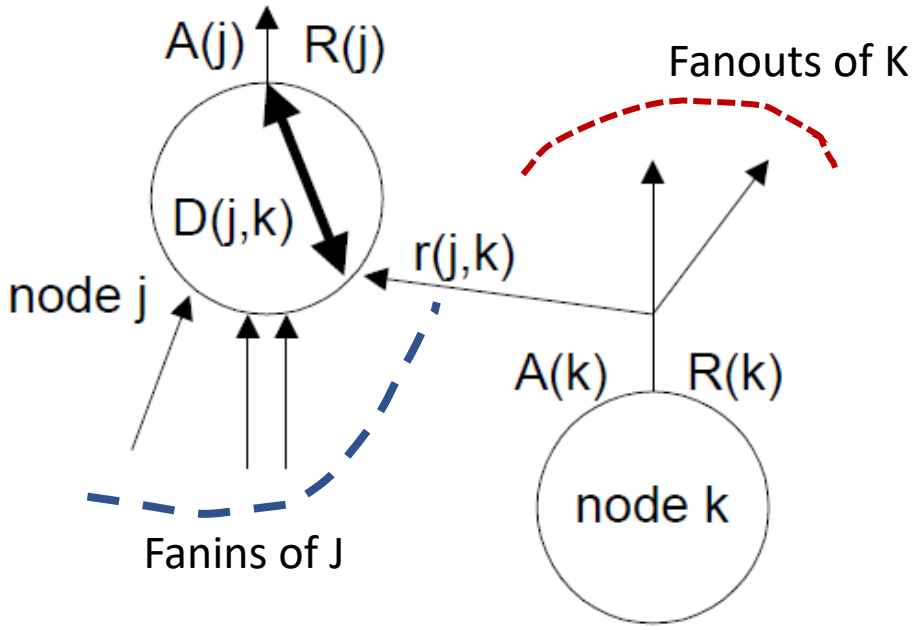
$$T_{ck->q} + T_{prop} > T_{hold} + T_{skew}$$

Arrival Time and Required Time

- arrival time : calculated from input to output
- required time : calculated from output to input
- slack = required time - arrival time

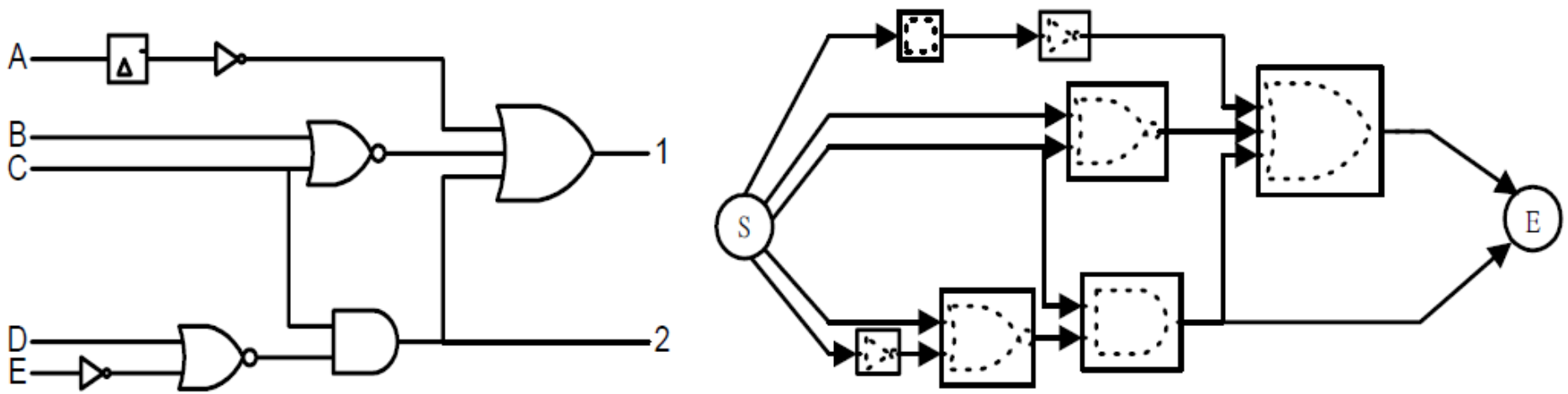
A(j): arrival time of signal j
R(k): required time or for signal k
S(k): slack of signal k
D(j,k): delay of node j from input k

$A(j) = \max_{k \in FI(j)} [A(k) + D(j,k)]$
$r(j,k) = R(j) - D(j,k)$
$R(k) = \min_{j \in FO(k)} [r(j,k)]$
$S(k) = R(k) - A(k)$



Delay Graph

- Replace logic gates with delay blocks
- Add start (S) and end (E) blocks
- Indicate signal flow with directed arcs

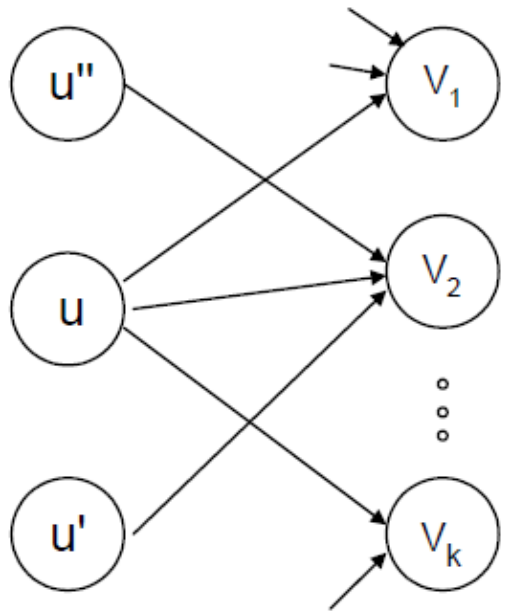


Longest and Shortest Path

- If we visit vertices in precedence order, the following code will need executing only once for each *u*

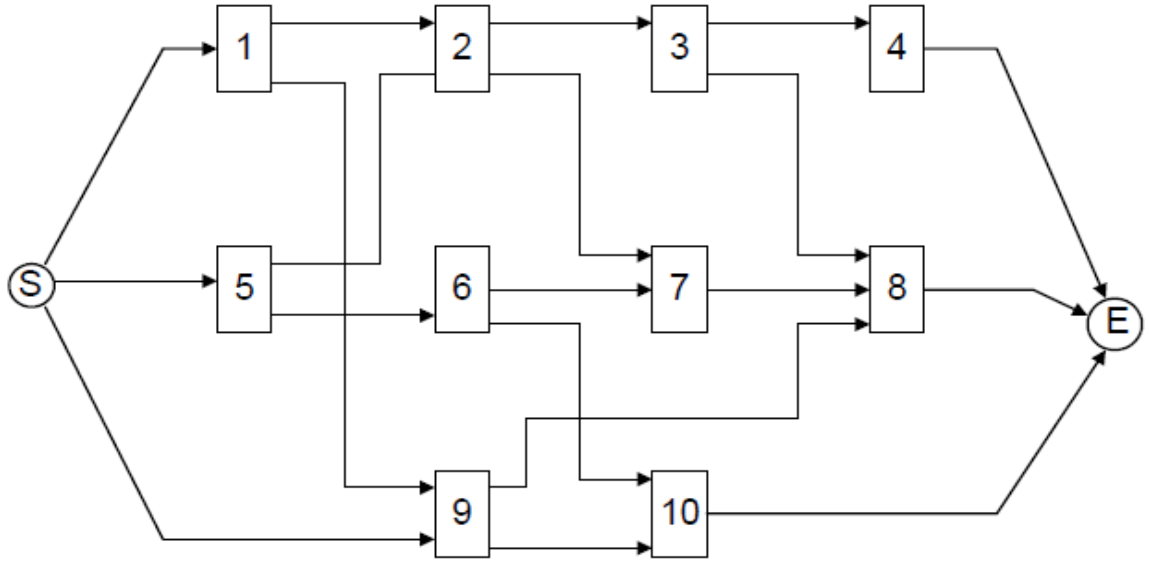
Update Successors[u]

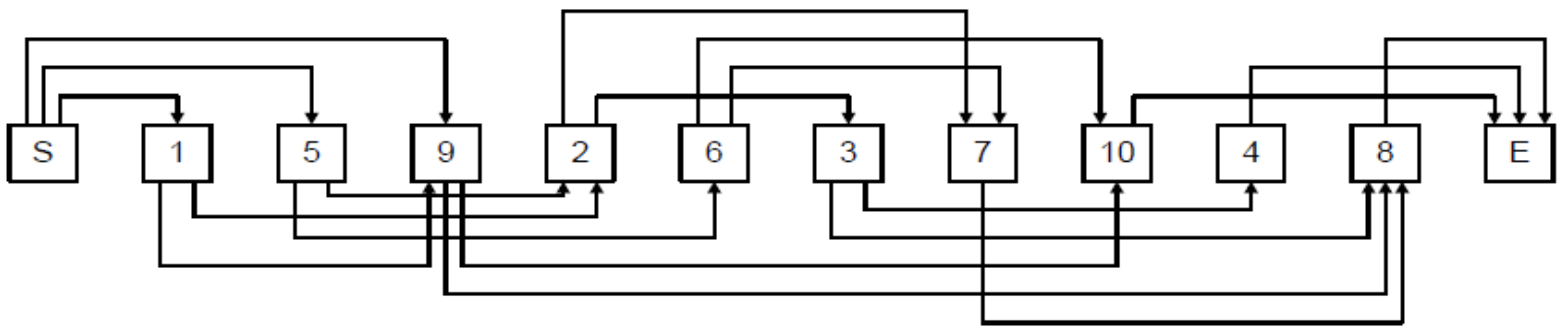
```
1  for each vertex v ∈ Adj[u] do
2      if  A[v] < A[u] + Δ[u]   /* longest */
3          then A[v] ← A[u] + Δ [u]
4               LP[v] ← u
5      if  a[v] > a[u] + δ[u]   /* shortest */
6          then a[v] ← a[u] + δ[u]
7               SP[v] ← u
```



Delay Graph and Topological Sort

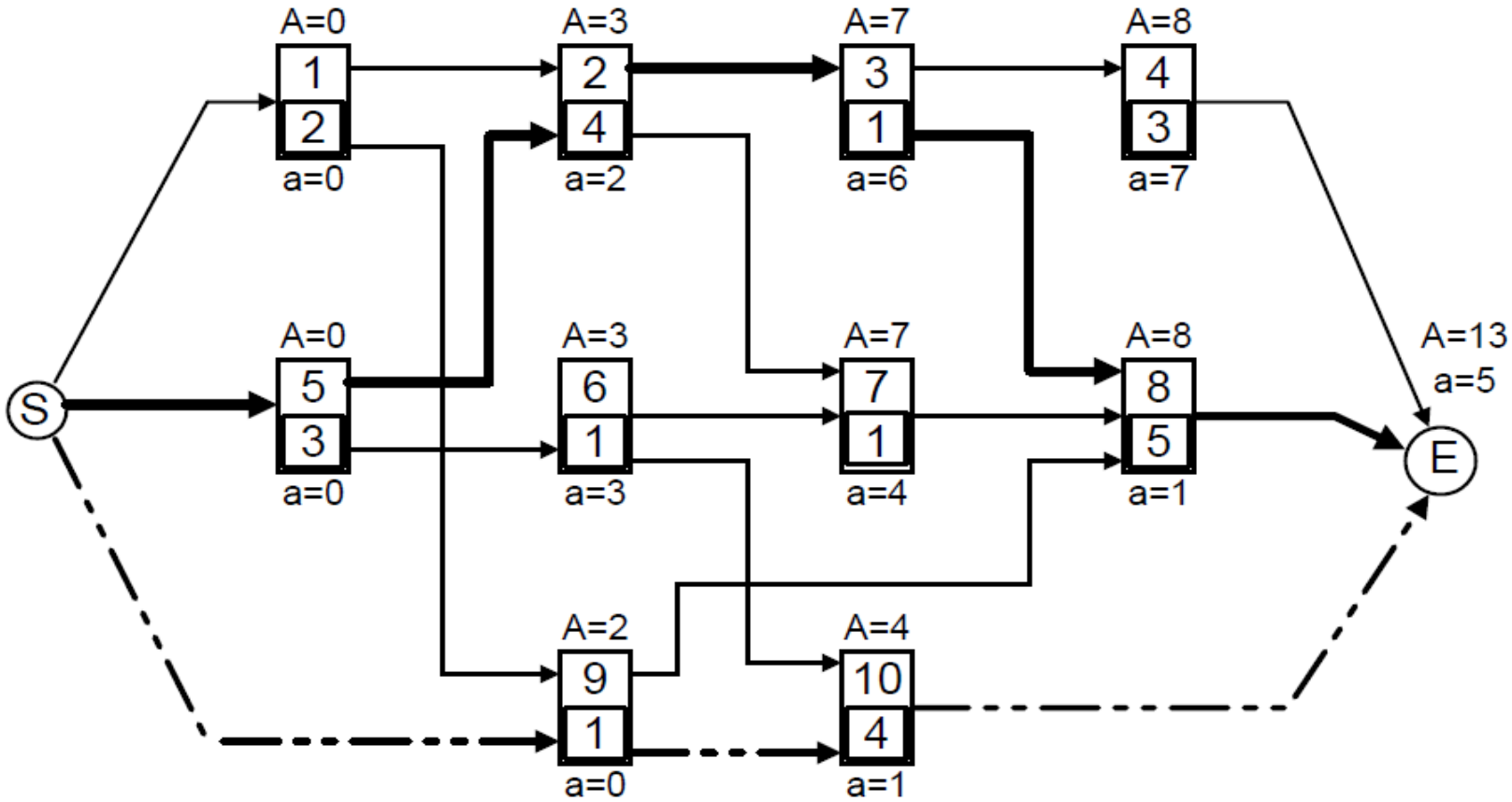
- Extract the topological order from the delay graph





topological order

Delay Calculation



A=3 → longest path delay

2 → node number

4 → gate delay

a=2 → shortest path delay

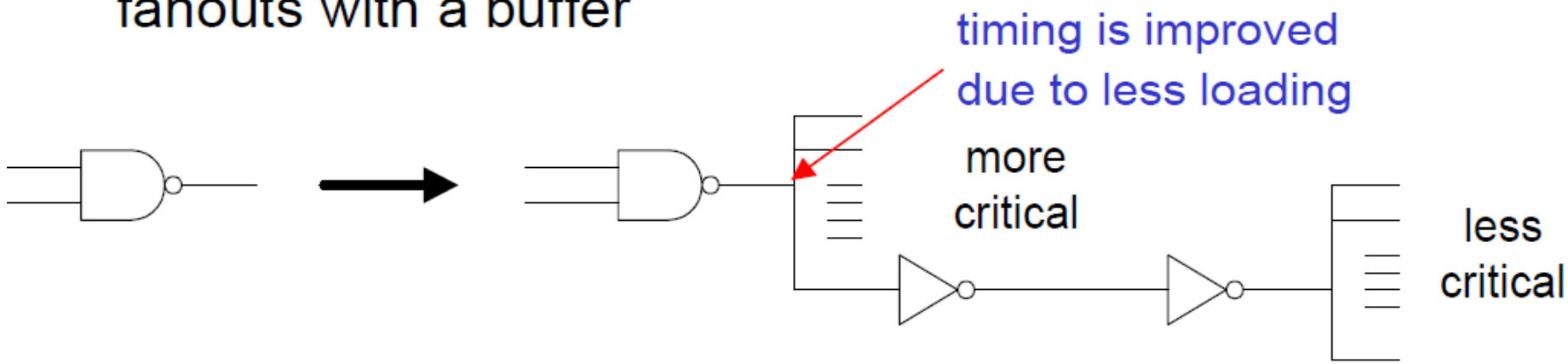
P.S: The longest delay and shortest delay of each gate are assumed to be the same.

Timing Optimization Techniques

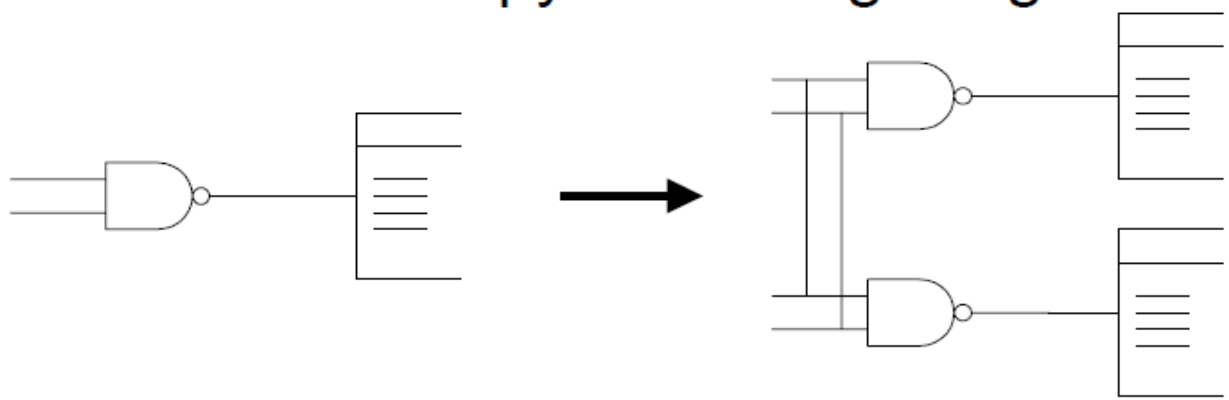
- Fanout optimization
  - Buffer insertion
  - Split
- Timing-driven restructuring
  - Critical path collapsing
  - Timing decomposition
- Misc
  - De Morgan
  - Repower
  - Down power
- Most of them will increase area to improve timing
  - Have to make a good trade-off between them

Fanout Optimization for Timing

- **Buffer insertion**: divide the fanouts of a gate into critical and non-critical parts and drive the non-critical fanouts with a buffer

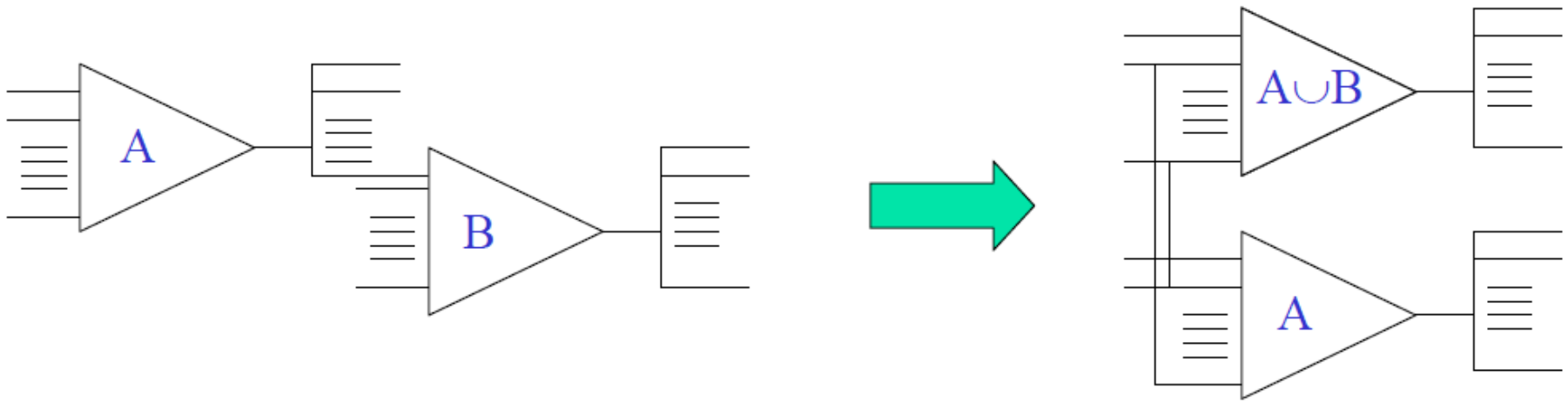


- **Split**: split the fanouts of a gate into several parts. Each part is driven with a copy of the original gate.



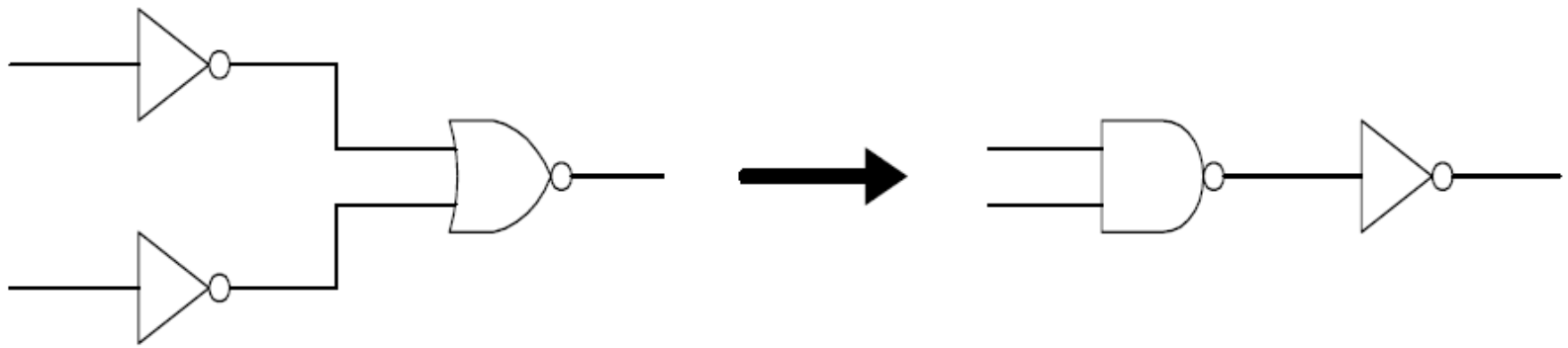
Path Collapsing for Timing Optimization

- **Critical path collapsing**: reduce the depth of logic networks



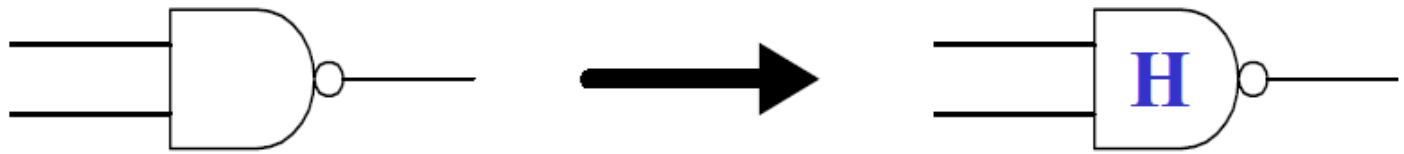
De Morgan Law for Timing Optimization

- **De Morgan**: replace a gate with its dual and reverse the polarity of inputs and output
  - NAND gate is typically faster than NOR gate

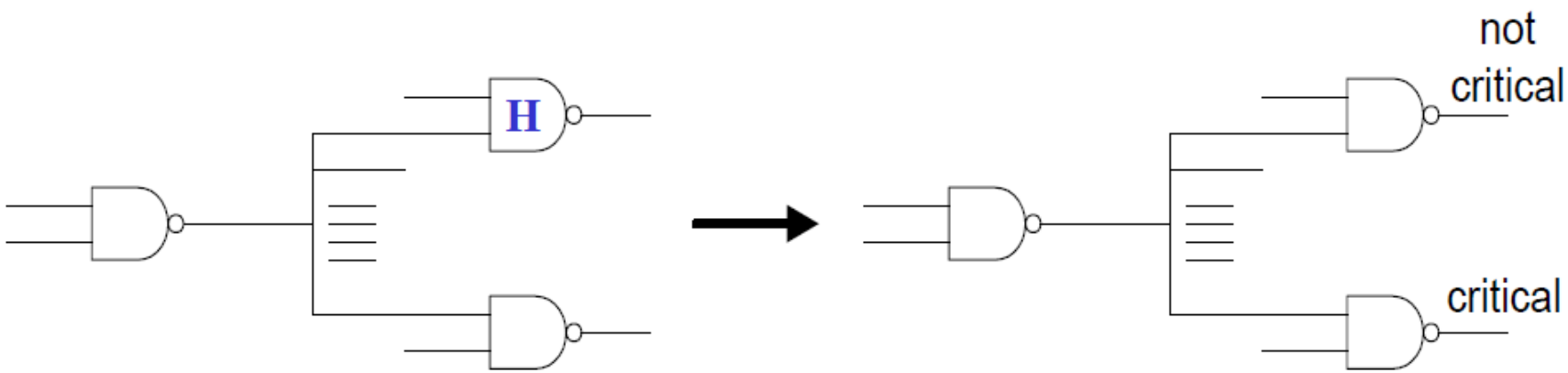


Repowering for Timing Optimization

- **Repower**: replace a gate with one of the other gate in its logic class with higher driving capability



- **Down power**: reducing gate size of a non-critical fanout in the critical path

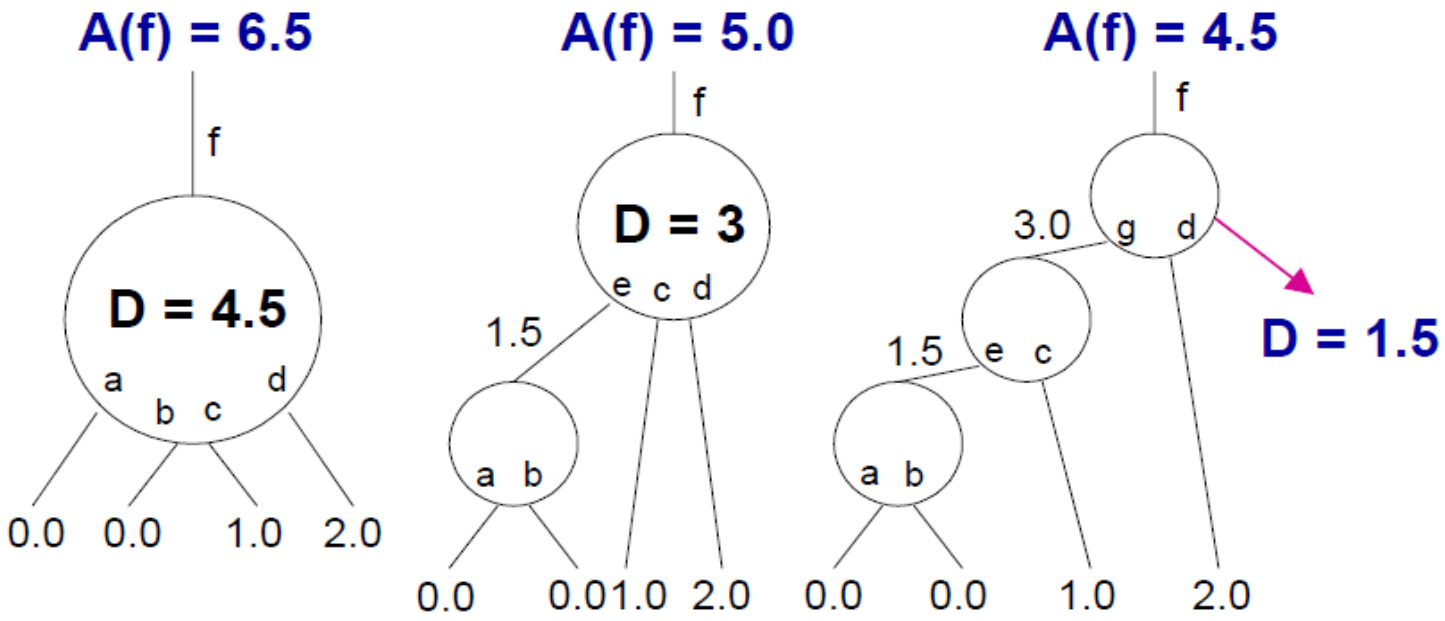


Adapted from: Prof. Y.-W. Chang's slides

Logic Restructuring for Timing Optimization

- **Timing decomposition**: restructuring the logic networks to minimize the arrival time

More advanced way



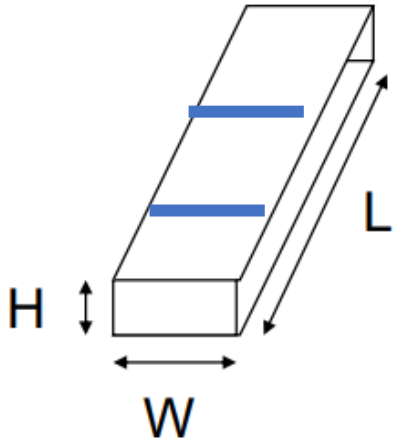
f = abcd

e = ab
f = ecd

e = ab
g = ce
f = dg

Elmore Delay (For Wiring)

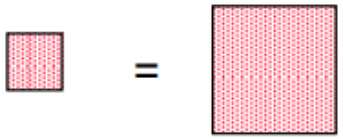
Wire Resistance



$$R = \frac{\rho L}{A} = \frac{\rho}{H} \frac{L}{W}$$

Sheet Resistance $R_{\square}$

$R_{1\square} = R_{2\square}$

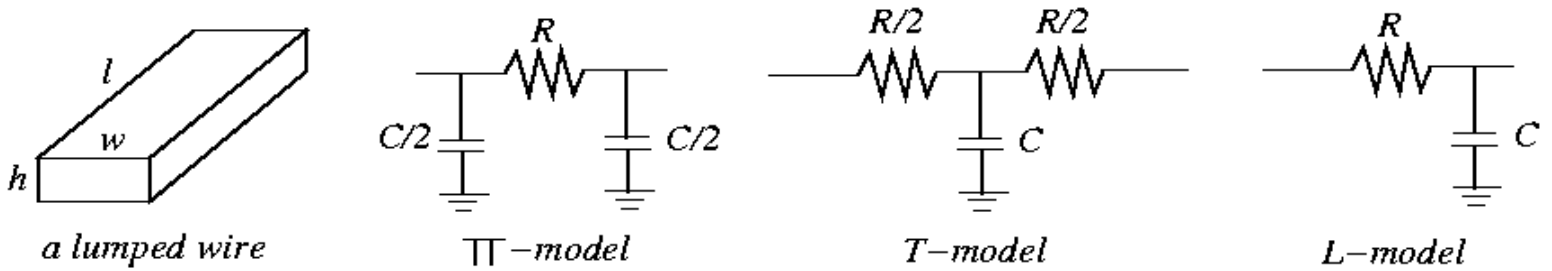


Material	ρ(Ω-m)
Silver (Ag)	1.6×10^{-8}
Copper (Cu)	1.7×10^{-8}
Gold (Au)	2.2×10^{-8}
Aluminum (Al)	2.7×10^{-8}
Tungsten (W)	5.5×10^{-8}

Material	Sheet Res. (Ω/❐)
n, p well diffusion	1000 to 1500
n+, p+ diffusion	50 to 150
n+, p+ diffusion with silicide	3 to 5
polysilicon	150 to 200
polysilicon with silicide	4 to 5
Aluminum	0.05 to 0.1

Wire Models

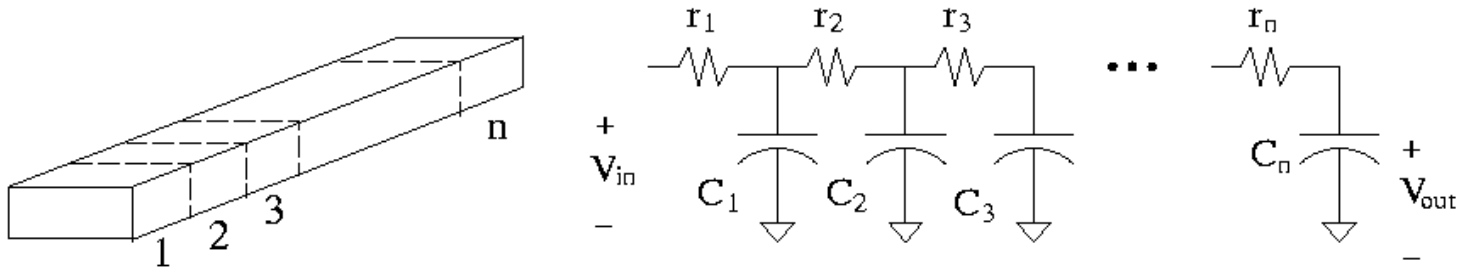
- Lumped circuit approximations for distributed RC lines: **π-model** (most popular), *T*-model, *L*-model.



Elmore Delay: Nonlinear Delay Model

- Parasitic resistance and capacitance start to dominate delay in deep submicron wires.
- Resistor r_i must charge all downstream capacitors.
- **Elmore delay:** Delay can be approximated as sum of sections: resistance × downstream capacitance.

$$\delta = \sum_{i=1}^{n} \left(r_i \sum_{k=i}^{n} c_k \right) = \sum_{i=1}^{n} r(n-i+1)c = \frac{n(n+1)}{2} rc.$$



- Delay (delta) grows as **square** of wire length.

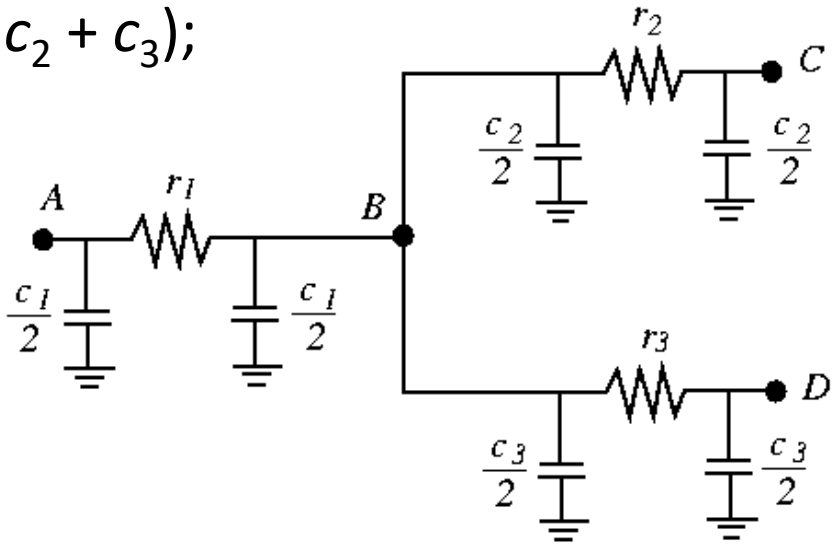
Wire Models

- π-model: If no capacitive loads for *C pin* and *D pin*,

  A to *B delay*: $\delta_{AB} = r_1 (c_1/2 + c_2 + c_3)$;

  B to *C*: $\delta_{BC} = r_2 (c_2/2)$;

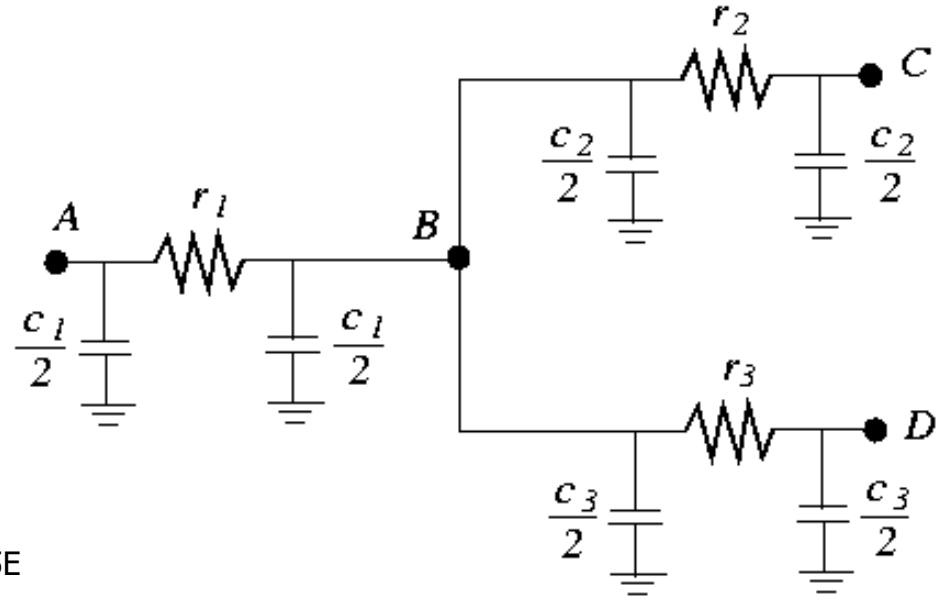
  B to *D*: $\delta_{BD} = r_3 (c_3/2)$.

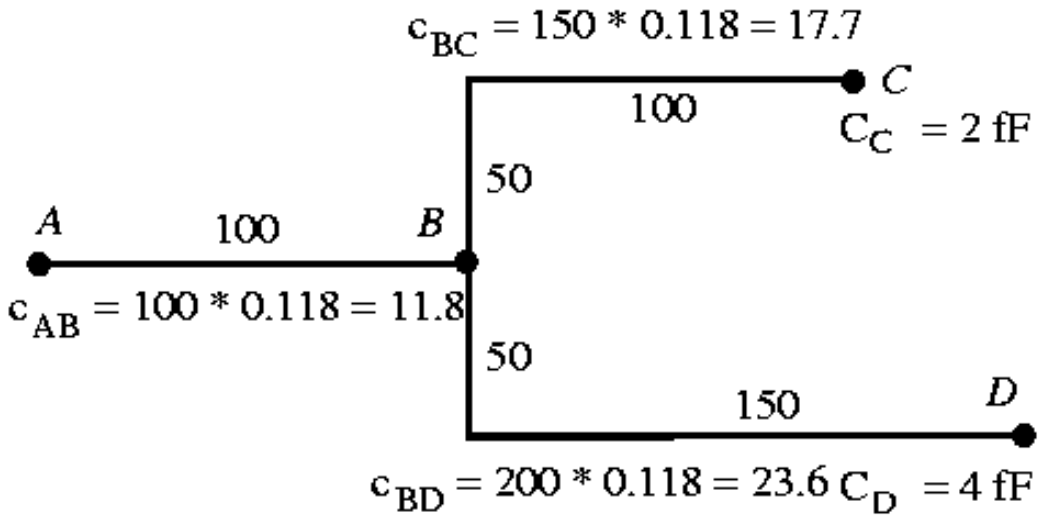


Example Elmore Delay Computation

- 0.18 μm technology.: unit resistance $\hat{r}$ = 0.075 $\Omega / \mu m$; unit capacitance $\hat{c}$ = 0.118 $fF/\mu m$.
  - Assume C_C = 2 fF, C_D = 4 fF.
  - $\delta_{BC} = r_{BC}\,(c_{BC}/2 + C_C) = 0.075 \times 150\,(17.7/2 + 2) = 120$ fs
  - $\delta_{BD} = r_{BD}\,(c_{BD}/2 + C_D) = 0.075 \times 200\,(23.6/2 + 4) = 240$ fs
  - $\delta_{AB} = r_{AB}\,(c_{AB}/2 + C_B) = 0.075 \times 100\,(11.8/2 + 17.7 + 2 + 23.6 + 4) = 400$ fs
  - Critical path delay: $\delta_{AB} + \delta_{BD}$ = 640 fs.

B-to-C: wire length 150um
Cbc=150um*0.118=17.7





Summary So Far

- We covered BDD, Quine-McClusky logic minimization, technology mapping for standard cells, briefly on FPGA lookup table,
- Touched upon delay analysis
- Also briefly talked about timing optimization, low power

- Very important area