Logic Synthesis and Verification

Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

Fall 2013

Technology Mapping

Reading:

Logic Synthesis in a Nutshell

Section 4

most of the following slides are by courtesy of Andreas Kuehlmann

Technology Independent Optimization

- Example

$t_1 = a + bc$

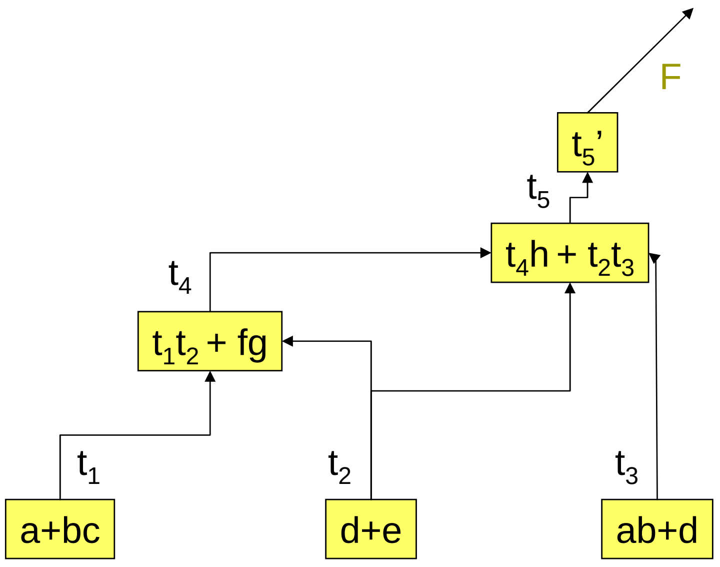
$t_2 = d + e$

$t_3 = ab + d$

$t_4 = t_1t_2 + fg$

$t_5 = t_4h + t_2t_3$

$F = t_5'$



An unoptimized set of logic equations consisting of 17 literals

Technology Independent Optimization

- Example (cont'd)

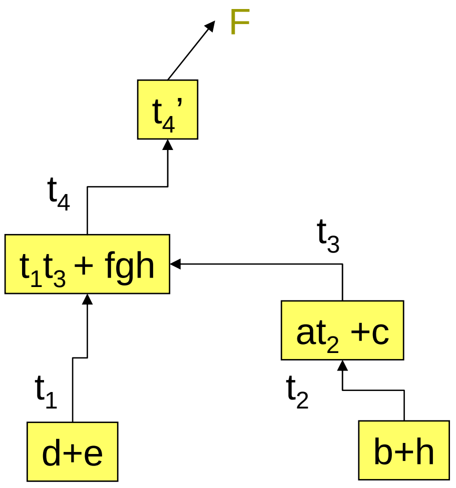
$t_1 = d + e$

$t_2 = b + h$

$t_3 = at_2 + c$

$t_4 = t_1t_3 + fgh$

$F = t_4'$

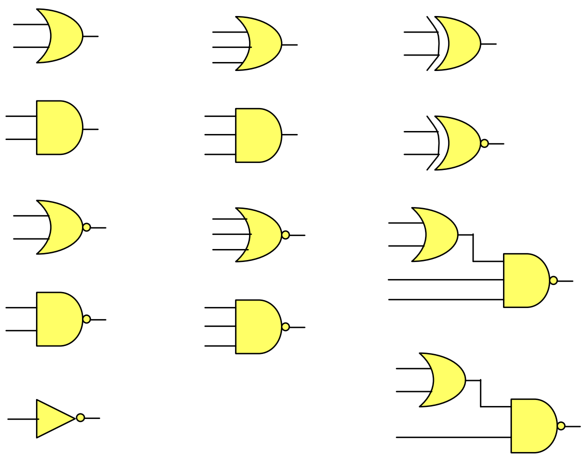


An optimized set of logic equations consisting of 13 literals

Technology Mapping

- Implement an optimized Boolean network using a set of pre-designed and pre-characterized gates from a *library*
  - Each gate has a cost (e.g. area, delay, power, etc.)

library of primitive gates



Technology Mapping

- Two approaches:
  1. Rule based: LSS
  2. Algorithmic: DAGON
     - Represent the netlist to be mapped in terms of a selected set of base functions, e.g., {NAND2, INV}
       - Base functions from a functionally complete set
       - Such a netlist is called the subject graph
     - Each gate in the library is likewise represented using the base functions
       - Represent each gate in all possible ways
       - This generates *pattern graphs*

Algorithmic Technology Mapping

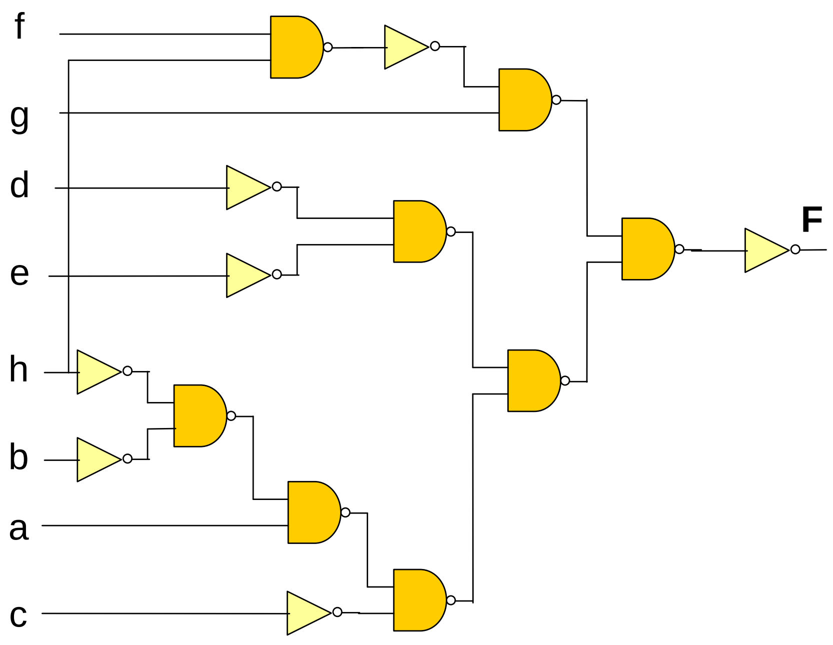
- A *cover* is a collection of pattern graphs such that
  - every node of the subject graph is contained in one (or more) pattern graphs
  - each input required by a pattern graph is actually an output of some other pattern graph (i.e. the inputs of one gate must exist as outputs of other gates)
- For area minimization, the cost of the cover is the sum of the areas of the gates in the cover
- Technology mapping problem:

  Find a minimum cost covering of the subject graph by choosing from the collection of pattern graphs for all the gates in the library

Subject Graph

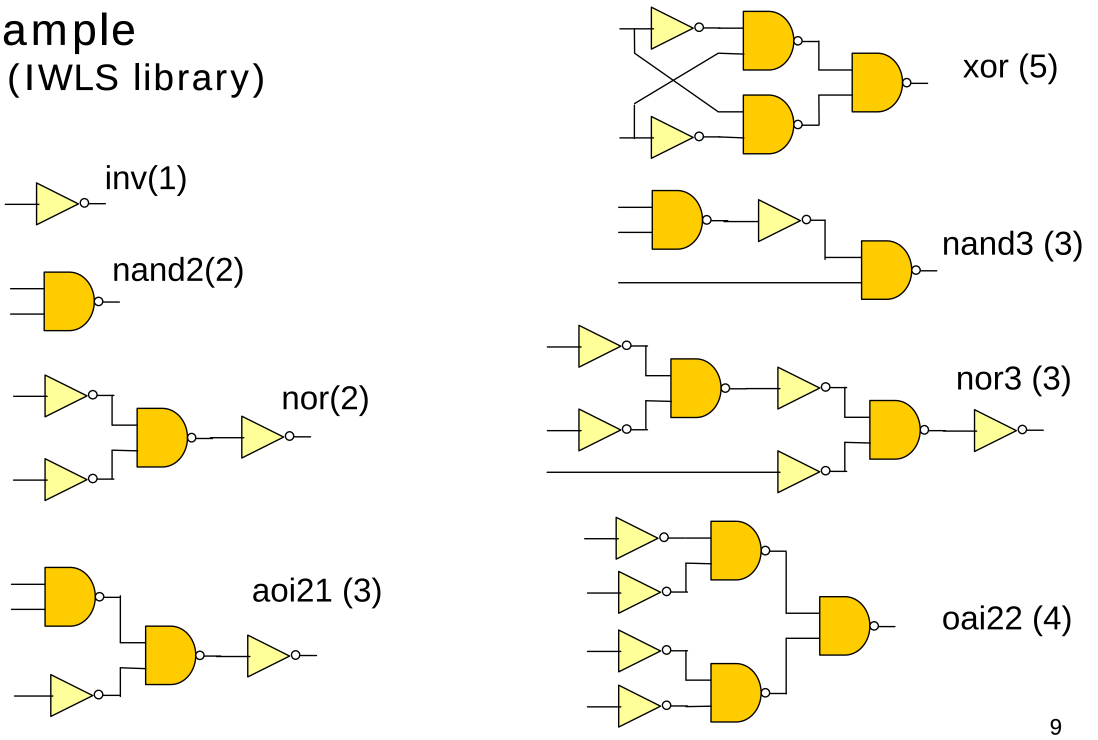
- Example

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



Pattern Graphs

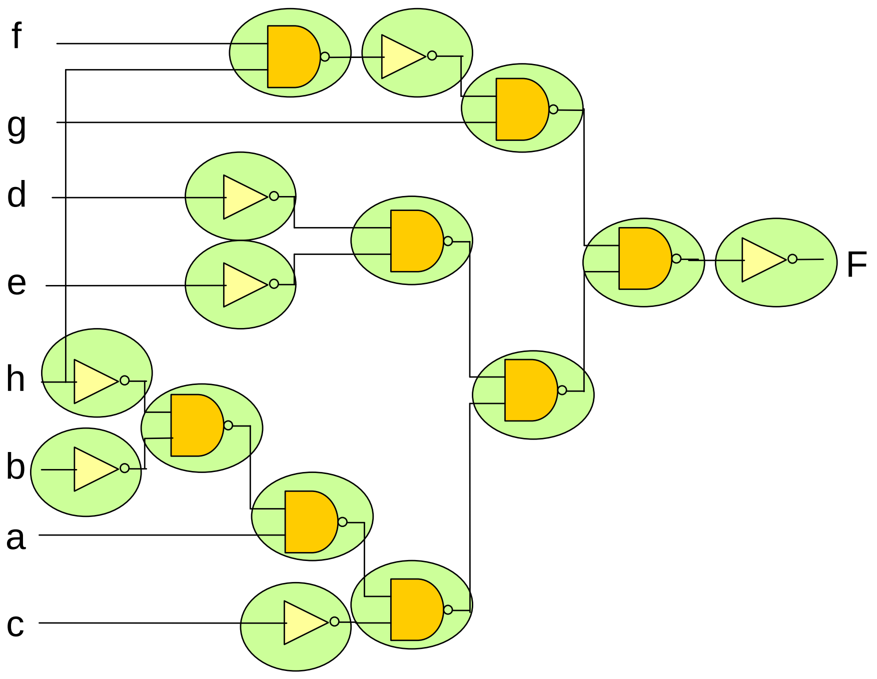
- Example
  - (IWLS library)



Subject Graph Covering (1)

- Example

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

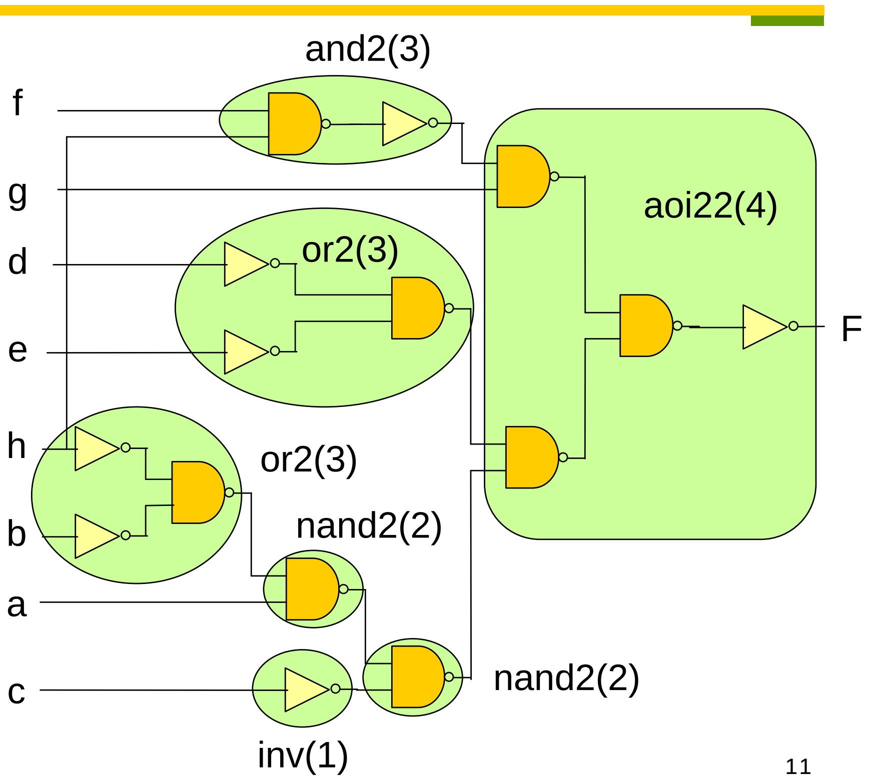


Total cost = 23

Subject Graph Covering (2)

- Example

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

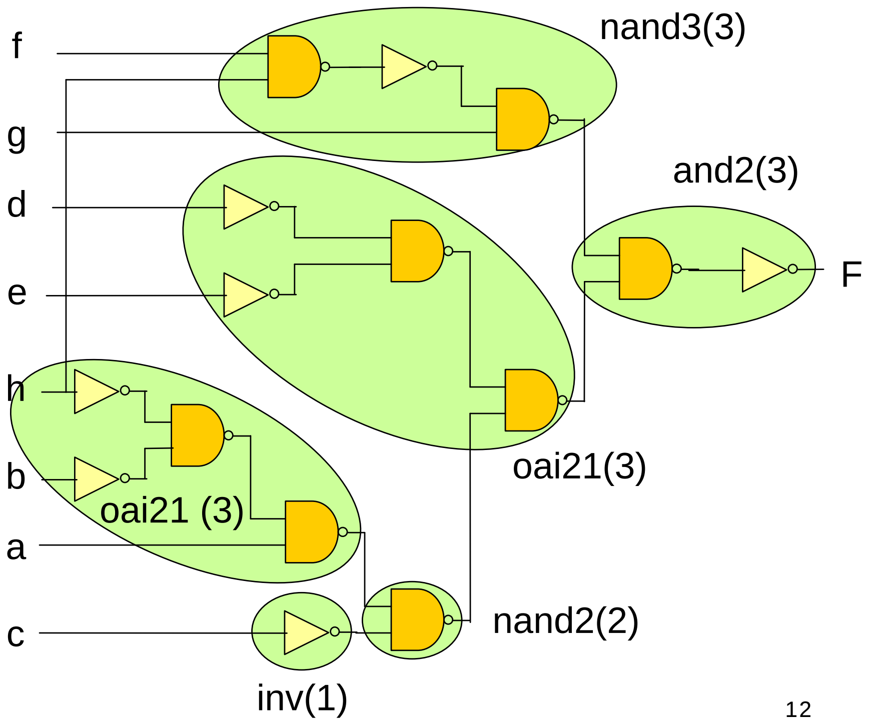


Total cost = 18

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

DAG Covering

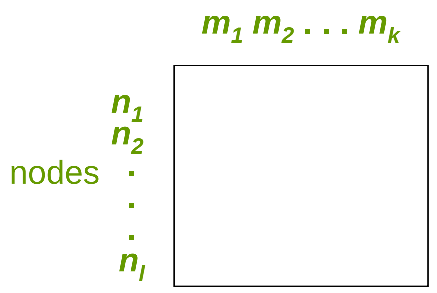
- Input:
  - Logic network after technology independent optimization
  - A library of gates with their costs
- Output:
  - A netlist of gates (from library) which minimizes total cost
- General Approach:
  - Construct a subject DAG (directed acyclic graph) for the network
  - Represent each gate in the 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

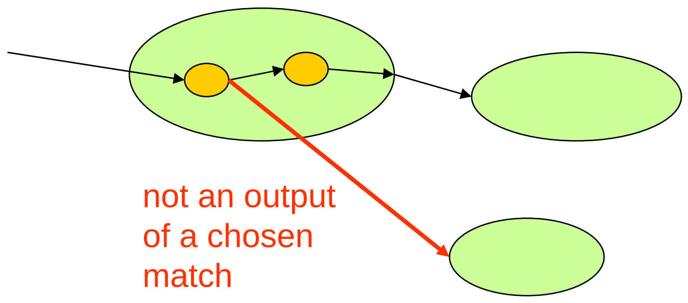
DAG Covering
Binate Covering Approach

- Compute all possible matches $\{m_k\}$ of pattern graphs for each node in the subject graph
- Using a variable m_i for each match of a pattern graph in the subject graph, ($m_i = 1$ if match is chosen)
- Write a clause for each node of the subject graph indicating which matches cover this node (each node has to be covered)
  - e.g., if a subject node is covered by matches $\{m_2, m_5, m_{10}\}$, then the clause would be $(m_2 + m_5 + m_{10})$
- Repeat for each subject node and take the product over all subject nodes (CNF)



DAG Covering
Binate Covering Approach

- Any satisfying assignment guarantees that all subject nodes are covered, but does not guarantee that *other matches chosen create outputs needed as inputs for a given match*



- Resolve this problem by adding additional clauses

DAG Covering
Binate Covering Approach

- Let match m_i have subject nodes $v_{i1},\dots,v_{in}$ as its n inputs. If m_i is chosen, one of the matches that realizes v_{ij} must also be chosen for each input j (if j not a primary input).

- Let S_{ij} be the disjunctive expression in the variables m_k giving the possible matches which realize v_{ij} as an output node. Selecting match m_i implies satisfying each of the expressions S_{ij} for j = 1 … n. This can be written

$$(m_i \Rightarrow (S_{i1} \dots S_{in})) \Leftrightarrow (\overline{m}_i + (S_{i1} \dots S_{in})) \Leftrightarrow ((\overline{m}_i + S_{i1}) \dots (\overline{m}_i + S_{in}))$$

DAG Covering
Binate Covering Approach

- Also, one of the matches for each primary output of the circuit must be selected
- An assignment to variables m_i that satisfies the above covering expression is a legal graph cover
- For area optimization, each match m_i has a cost c_i that is the area of the gate the match represents
- The goal is a satisfying assignment with the least total cost
  - Find a least-cost prime:
    - if a variable $m_i = 0$ its cost is 0, else its cost in c_i
    - $m_i = 0$ means that match i is not chosen

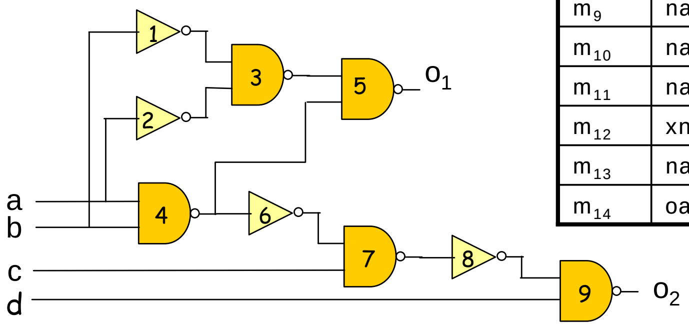
DAG Covering
Binate Covering Approach

- Binate covering is more general than unate covering
  - Unlike unate covering, variables are present in both their true and complemented forms in the covering expression
  - The covering expression is a binate function, and the problem is referred to as the *binate-covering problem*

DAG Covering
Binate Covering Approach

- Example

Match	Gate	Cost	Inputs	Root	Covers
m_1	inv	1	b	g_1	g_1
m_2	inv	1	a	g_2	g_2
m_3	nand2	2	g_1,g_2	g_3	g_3
m_4	nand2	2	a, b	g_4	g_4
m_5	nand2	2	g_3,g_4	g_5	g_5
m_6	inv	1	g_4	g_6	g_6
m_7	nand2	2	g_6,c	g_7	g_7
m_8	inv	1	g_7	g_8	g_8
m_9	nand2	2	g_8,d	g_9	g_9
m_{10}	nand3	3	g_6,c,d	g_9	g_7,g_8,g_9
m_{11}	nand3	3	a,b,c	g_7	g_4,g_6,g_7
m_{12}	xnor2	5	a,b	g_5	g_1,g_2,g_3,g_4,g_5
m_{13}	nand4	4	a,b,c,d	g_9	g_4,g_6,g_7,g_8,g_9
m_{14}	oai21	3	a,b,g_4	g_5	g_1,g_2,g_3,g_5



DAG Covering
Binate Covering Approach

- Example (cont'd)
  - Generate constraints that each node g_i be covered by some match

    $(m_1 + m_{12} + m_{14})\ (m_2 + m_{12} + m_{14})\ (m_3 + m_{12} + m_{14})$
    $(m_4 + m_{11} + m_{12} + m_{13})\ (m_5 + m_{12} + m_{14})$
    $(m_6 + m_{11} + m_{13})\ (m_7 + m_{10} + m_{11} + m_{13})$
    $(m_8 + m_{10} + m_{13})\ (m_9 + m_{10} + m_{13})$
  - To ensure that a cover leads to a valid circuit, extra clauses are generated
    - For example, selecting m_3 requires that
      - a match be chosen which produces g_2 as an output, and
      - a match be chosen which produces g_1 as an output
  - The only match which produces g_1 is m_1, and the only match which produces g_2 is m_2

DAG Covering
Binate Covering Approach

- Example (cont'd)
  - The primary output nodes g_5 and g_9 must be realized as an output of some match
    - The matches which realize g_5 as an output are m_5, m_{12}, m_{14}
    - The matches which realize g_9 as an output are m_9, m_{10}, m_{13}
  - Note:
    - A match which requires a primary input as an input is satisfied trivially
    - Matches $m_1, m_2, m_4, m_{11}, m_{12}, m_{13}$ are driven only by primary inputs and do not require additional clauses

DAG Covering
Binate Covering Approach

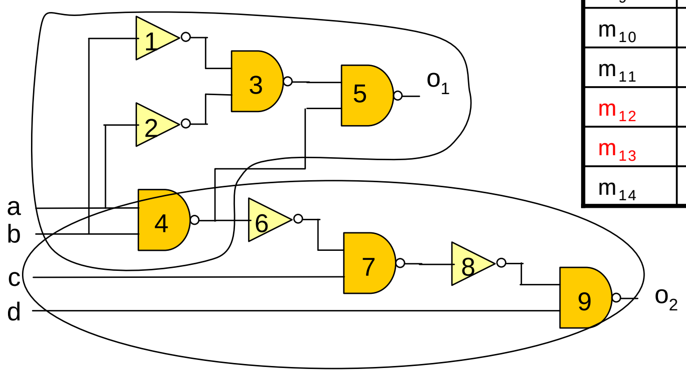
- Example (cont’d)
  - Finally, we get
    $(\overline{m}_3 + m_1)(\overline{m}_3 + m_2)(m_3 + \overline{m}_5)(\overline{m}_5 + m_4)(\overline{m}_6 + m_4)$
    $(\overline{m}_7 + m_6)(\overline{m}_8 + m_7)(m_8 + \overline{m}_9)(\overline{m}_{10} + m_6)$
    $(\overline{m}_{14} + m_4)(m_5 + m_{12} + m_{14})(m_9 + m_{10} + m_{13})$
  - The covering expression has 58 implicants
  - The least cost prime implicant is
    $m_3\ \overline{m}_5\ \overline{m}_6\ \overline{m}_7\ \overline{m}_8\ \overline{m}_9\ \overline{m}_{10}\ m_{12}\ m_{13}\ \overline{m}_{14}$
  - This uses two gates for a cost of 9 gate units. This corresponds to a cover which selects matches m_{12} (xor2) and m_{13} (nand4).

DAG Covering
Binate Covering Approach

- Example (cont’d)

Match	Gate	Cost	Inputs	Root	Covers
m_1	inv	1	b	g_1	g_1
m_2	inv	1	a	g_2	g_2
m_3	nand2	2	g_1,g_2	g_3	g_3
m_4	nand2	2	a, b	g_4	g_4
m_5	nand2	2	g_3,g_4	g_5	g_5
m_6	inv	1	g_4	g_6	g_6
m_7	nand2	2	g_6,c	g_7	g_7
m_8	inv	1	g_7	g_8	g_8
m_9	nand2	2	g_8,d	g_9	g_9
m_{10}	nand3	3	g_6,c,d	g_9	g_7,g_8,g_9
m_{11}	nand3	3	a,b,c	g_7	g_4,g_6,g_7
m_{12}	xor2	5	a,b	g_5	g_1,g_2,g_3,g_4,g_5
m_{13}	nand4	4	a,b,c,d	g_9	g_4,g_6,g_7,g_8,g_9
m_{14}	oai21	3	a,b,g_4	g_5	g_1,g_2,g_3,g_5

$m_3\ \overline{m}_5\ \overline{m}_6\ \overline{m}_7\ \overline{m}_8\ \overline{m}_9\ \overline{m}_{10}\ m_{12}\ m_{13}\ \overline{m}_{14}$



Note: g_4 is covered by both matches

DAG Covering
Binate Covering Approach

- Complexity
  - DAG-covering: covering + implication constraints
  - More general than unate covering
    - Finding least cost prime of a *binate* function
      - Even finding a feasible solution is NP-complete (SAT)
      - For unate covering, finding a feasible solution is easy
  - Given a *subject graph*, the binate covering provides the exact solution to the technology-mapping problem
    - However, better results may be obtained with a different initial decomposition into 2-input NANDs and inverters
  - Methods to solve the binate covering formulation:
    - Branch and bound, BDD-based
    - Expensive even for moderate-size networks

Tree Covering

- When the subject graph and pattern graphs are trees, an efficient algorithm to find the best cover exists
- Solvable with dynamic programming

Tree Covering

1. Partition subject graph into forest of trees
2. Cover each tree optimally using dynamic programming
   - 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*

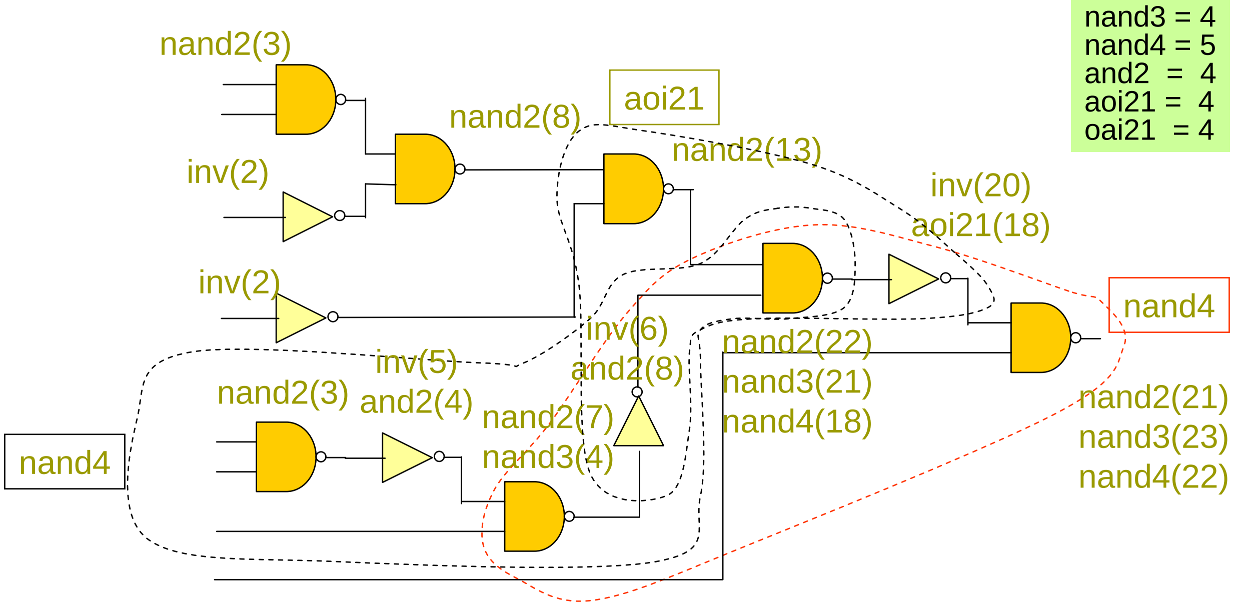
Tree Covering

```
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;
    }
  }
}
```

Tree Covering

- Example

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

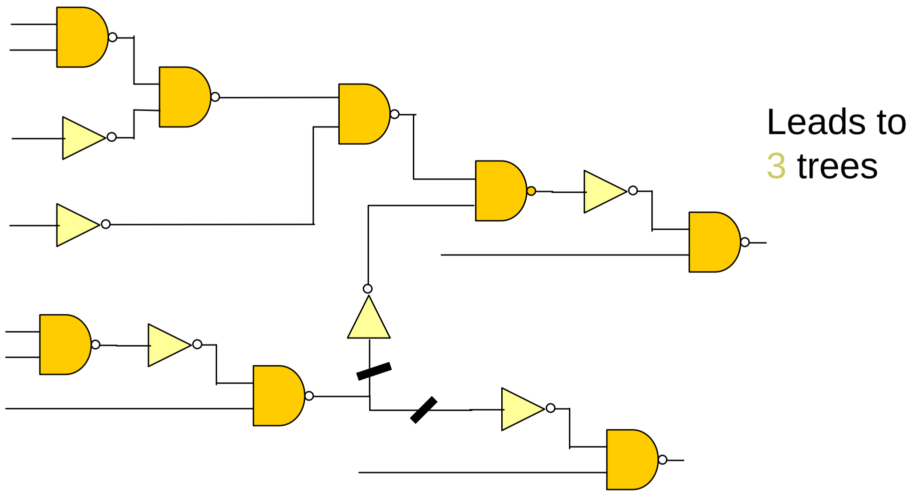


Tree Covering

- 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

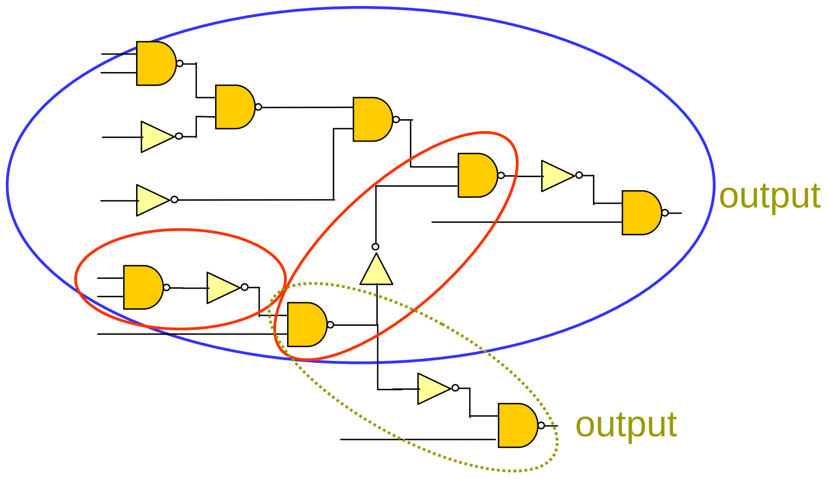
Tree Covering

- 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



Tree Covering

- 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



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

Decomposition and Technology Mapping

Common Approach:

- Phase 1: Technology independent optimization
  - commit to a particular Boolean network
  - algebraic decomposition used
- Phase 2: AND2/INV decomposition
  - commit to a particular decomposition of a general Boolean network using 2-input ANDs and inverters
- Phase 3: Technology mapping (tree-mapping)

Drawbacks:

Procedures in each phase are disconnected:

- Phase 1 and Phase 2 make critical decisions without knowing much about constraints and library
- Phase 3 knows about constraints and library, but solution space is restricted by decisions made earlier

Combined Decomposition and Technology Mapping

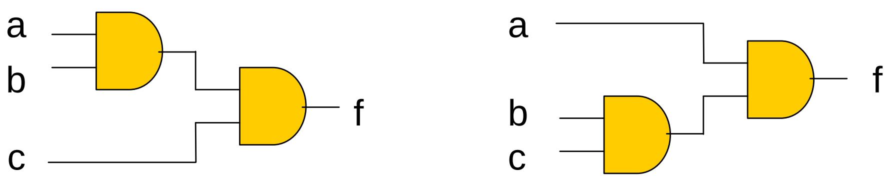
- Incorporate technology independent procedures (Phase 1 and Phase 2) into technology mapping
- Lehman-Watanabe Algorithm:
- Key Idea:
  - Efficiently encode a set of AND2/INV decompositions into a single structure called a *mapping graph*
  - Apply a modified tree-based technology mapper while *dynamically performing algebraic logic decomposition* on the mapping graph

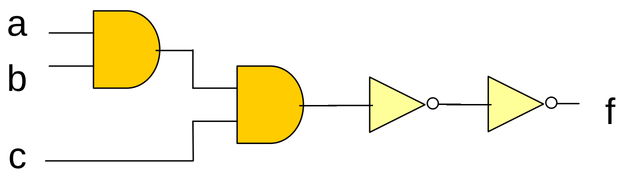
Combined Decomposition and Technology Mapping

- Outline
  - Mapping Graph
    - Encodes a set of AND2/INV decompositions
  - Tree-mapping on a mapping graph: graph-mapping
  - Λ-mapping:
    - without dynamic logic decomposition
    - solution space: Phase 3 + Phase 2
  - Δ-mapping:
    - with dynamic logic decomposition
    - solution space: Phase 3 + Phase 2 + Algebraic decomposition (Phase 1)

Combined Decomposition and Technology Mapping

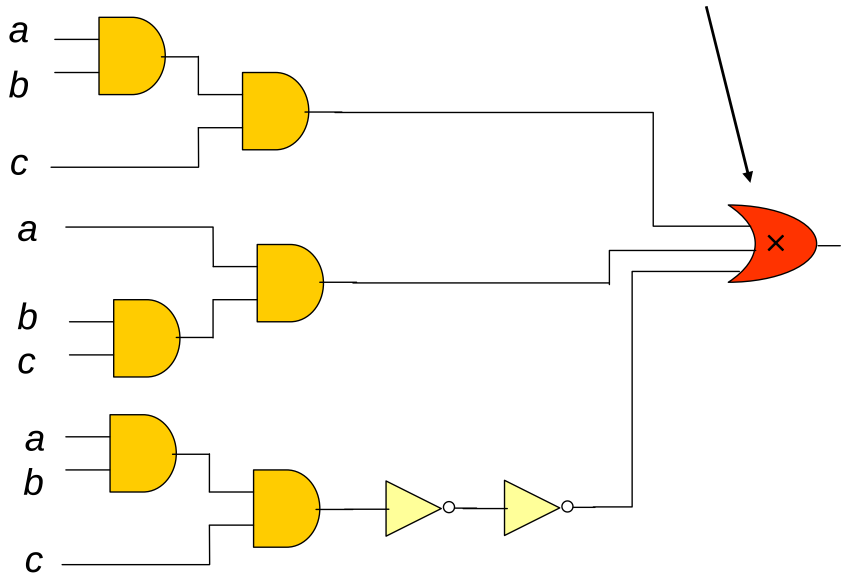
- AND2/INV decomposition
  - E.g., *f* = *abc* can be represented in various ways





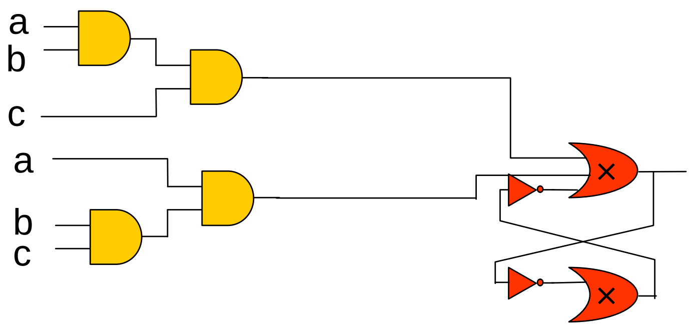
Combined Decomposition and Technology Mapping

- Combine different AND2/INV decompositions with a choice node

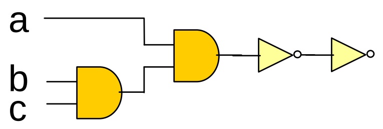


Combined Decomposition and Technology Mapping

- The previous AND2/INV decompositions can be represented more compactly as:

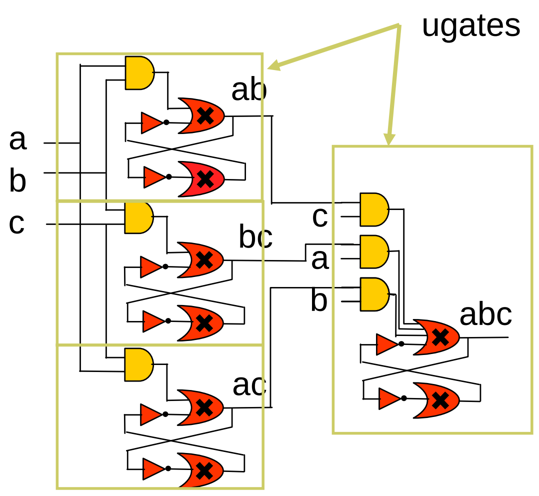


- This representation encodes even more decompositions, e.g.,



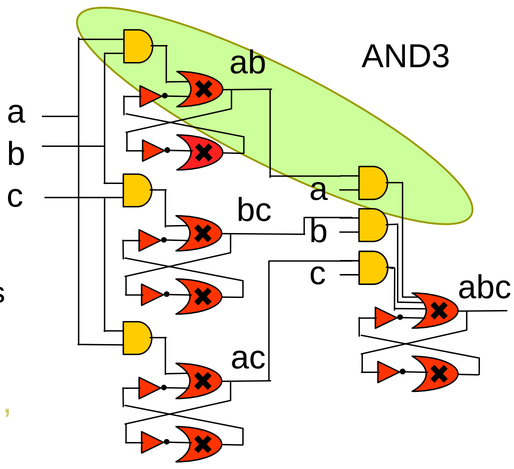
Combined Decomposition and Technology Mapping

- Mapping graph is a Boolean network containing the following four modifications:
  - Choice node: choices on different decompositions
  - Cyclic: functions written in terms of each other, e.g. inverter chain with an arbitrary length
  - Reduced: No two choice nodes with same function. No two AND2s with same fanin. (like BDD node sharing)
  - Ugates: just for efficient implementation - do not explicitly represent choice nodes and inverters
    - For CHT benchmark (MCNC'91), there are 2.2x10^{93} AND2/INV decompositions. All are encoded with only 400 ugates containing 599 AND2s in total.



Combined Decomposition and Technology Mapping

- Graph-Mapping on Trees*:
  Apply dynamic programming from primary inputs:
  - find matches at each AND2 and INV, and
  - retain the cost of a best cover at each node
    - a match may contain choice nodes
    - the cost at a choice node is the minimum of fanin costs
  - fixed-point iteration on each cycle, until costs of all the nodes in the cycle become stable
- Run-time is typically linear in the size of the mapping graph



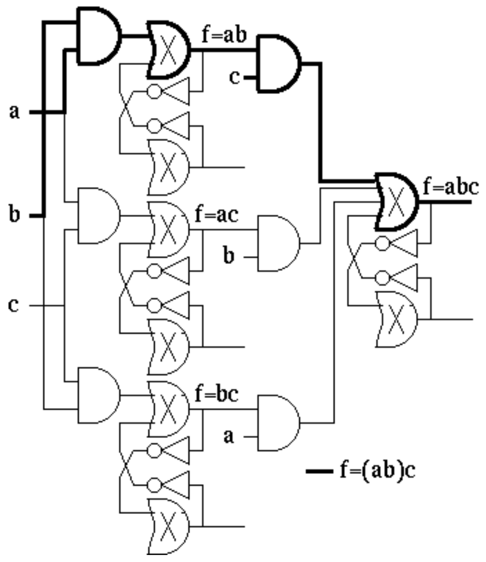
* mapping graph may not be a tree, but any multiple fanout node just represents several copies of same function.

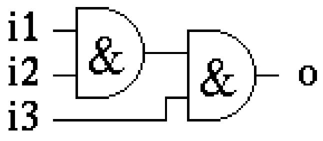
Combined Decomposition and Technology Mapping

- Example
  - Graph mapping on trees for min delay
    - best choice if c is later than a and b.

subject graph

library pattern graph





Combined Decomposition and Technology Mapping

- Graph mapping

  Graph-mapping(μ) = $\min_{\theta\in\mu}$ { tree-mapping(θ) }

  μ: mapping graph

  θ: AND2/INV decomposition encoded in μ

  - Graph-mapping finds an optimal tree implementation for each primary output over all AND2/INV decompositions encoded in μ
  - Graph-mapping is as powerful as applying tree-mapping exhaustively, but is typically exponentially faster

Combined Decomposition and Technology Mapping

- Λ-mapping

  Given a Boolean network η,

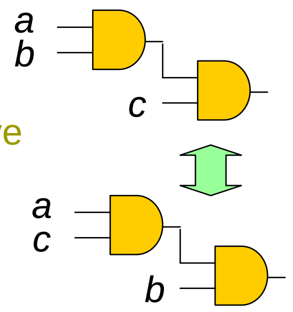
  - Generate a mapping graph μ:
  - For each node of η,
    - encode all AND2 decompositions for each product term
      - E.g., *abc* $\Rightarrow$ 3 AND2 decompositions: *a(bc), c(ab), b(ca)*
    - encode all AND2/INV decompositions for the sum term
      - E.g., *p+q+r* $\Rightarrow$ 3 AND2/INV decompositions:

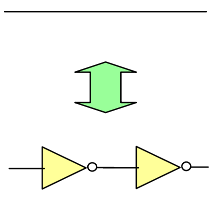
        p+(q+r), r+(p+q), q+(r+p)
    - In practice, η is preprocessed so each node has at most 10 product terms and each term has at most 10 literals
  - Apply graph-mapping on μ

Combined Decomposition and Technology Mapping

- Λ-mapping
  For the mapping graph μ generated for a Boolean network η, let
  - L_η be the set of AND2/INV decompositions encoded in μ
  - Λ_η be the closure of the set of AND2/INV decompositions of η under the associative and inverter transformations:

Associative transform



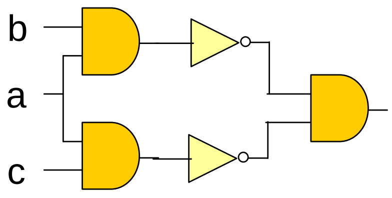


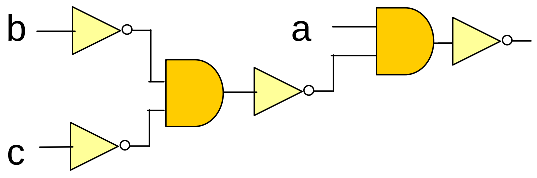
Inverter transform

- Theorem: $\Lambda\eta$ = $L\eta$

Combined Decomposition and Technology Mapping

- Dynamic logic decomposition
  - During graph-mapping, dynamically modify the mapping graph: find D-patterns and add F-patterns



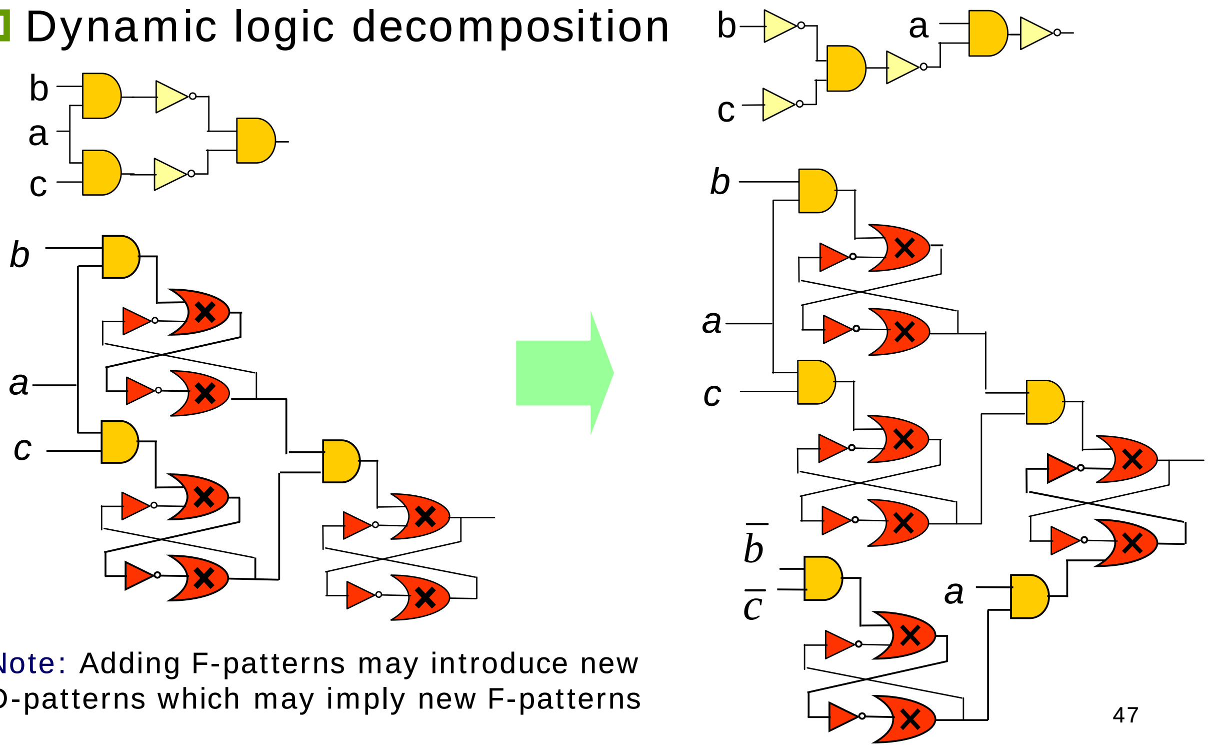


D-pattern: $\overline{ab+ac}$

F-pattern: $\overline{a(b+c)}$

Combined Decomposition and Technology Mapping

- Dynamic logic decomposition



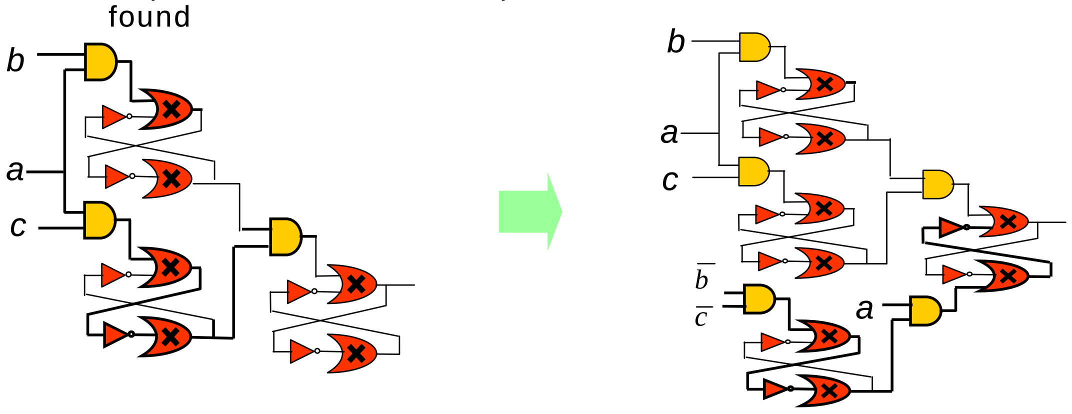
Note: Adding F-patterns may introduce new D-patterns which may imply new F-patterns

Combined Decomposition and Technology Mapping

- Δ-mapping

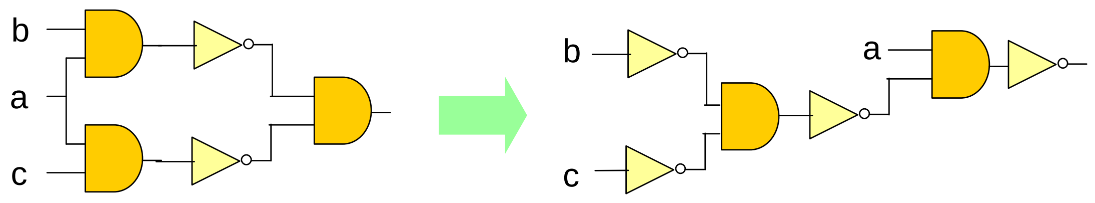
  Given a Boolean network η,

  - Generate a mapping graph μ
  - Iteratively apply graph mapping on μ, while performing dynamic logic decomposition until nothing changes in μ
    - Before finding matches at an AND2 in μ, check if D-pattern matches at the AND2. If so, add the corresponding F-pattern
    - In practice, terminate the procedure when a feasible solution is found



Combined Decomposition and Technology Mapping

- Δ-mapping
  For the mapping graph μ generated for a Boolean network η, let
  - D_η be the set of AND2/INV decompositions encoded in the resulting mapping graph.
  - Δ_η be the closure of Λ_η under the distributive transformation:



- Theorem: $\Delta_\eta = D_\eta$

Combined Decomposition and Technology Mapping

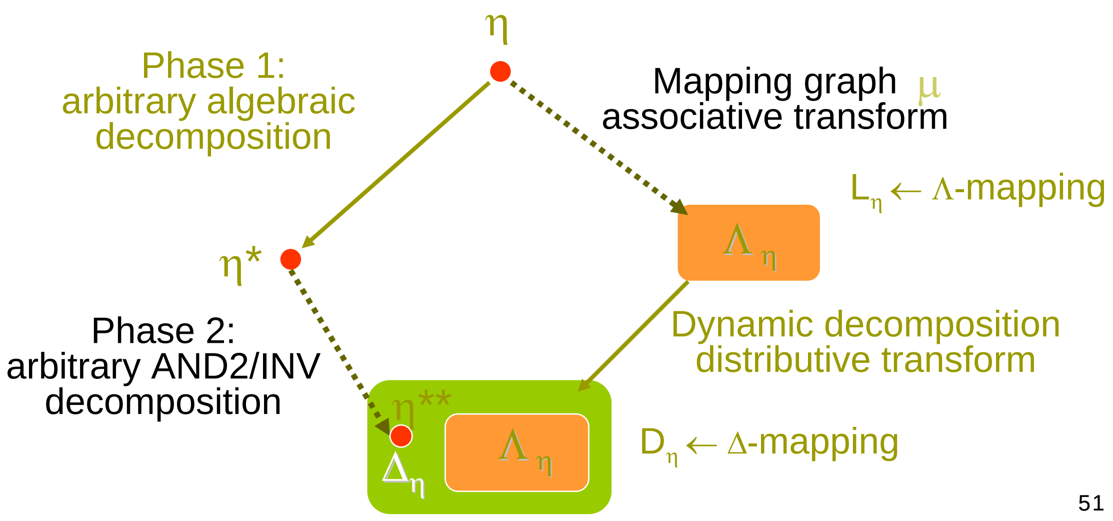
- Theorem: If
  1. η^* is an arbitrary Boolean network obtained from η by algebraic decomposition, and
  2. θ is an arbitrary AND2/INV decomposition of η^*

  then $\theta \in D_\eta$

- The resulting mapping graph encodes all the AND2/INV decompositions of all algebraic decompositions of η

Combined Decomposition and Technology Mapping

- Λ-mapping captures all AND2/INV decompositions of η:
  Phase 2 (subject graph generation) is subsumed
- Δ-mapping captures all algebraic decompositions:
  Phase 2 and Phase 1 are subsumed


η
Phase 1: arbitrary algebraic decomposition
Mapping graph μ associative transform
L_η ← Λ-mapping
Λ_η
η*
Dynamic decomposition distributive transform
Phase 2: arbitrary AND2/INV decomposition
η**
Δ_η
Λ_η
D_η ← Δ-mapping

Combined Decomposition and Technology Mapping

- Summary
  - Logic decomposition during technology mapping
    - Efficiently encode a set on AND2/INV decompositions
    - Dynamically perform logic decomposition
  - Two mapping procedures
    - Λ-mapping: optimal over all AND2/INV decompositions (associative rule)
    - Δ-mapping: optimal over all algebraic decompositions (distributive rule)
  - Was implemented and used for commercial design projects (in DEC/Compac alpha)
  - Extended for sequential circuits:
    - considers all retiming possibilities (implicitly) and algebraic factors across latches