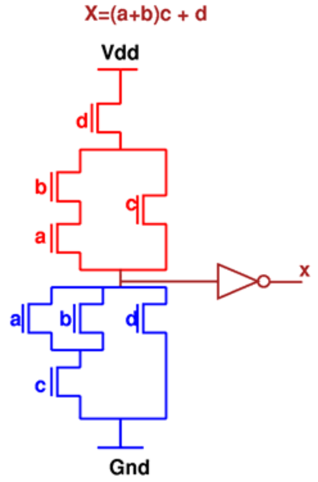
Factor Form

- Factor forms – beyond SOP
  - Example:
    (ad+b'c)(c+d'(e+ac'))+(d+e)fg
- Advantages
  - good representation reflecting logic complexity (SOP may not be representative)
    - E.g., f=ad+ae+bd+be+cd+ce has complement in simpler SOP f'= a'b'c'+d'e'; effectively has simple factor form f=(a+b+c)(d+e)
  - in many design styles (e.g. complex gate CMOS design) the implementation of a function corresponds directly to its factored form
  - good estimator of logic implementation complexity
  - doesn't blow up easily
- Disadvantages
  - not as many algorithms available for manipulation

Factor From

- Factored forms are useful in estimating area and delay in multi-level logic
  - Note: literal count ≈ transistor count ≈ area
    - however, area also depends on wiring, gate size, etc.
    - therefore very crude measure



Factor From

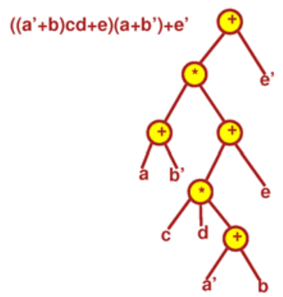
- There are functions whose sizes are exponential in the SOP representation, but polynomial in the factored form
  - Example
    Achilles' heel function

$$\prod_{i=1}^{i=n/2}(x_{2i-1}+x_{2i})$$

There are *n* literals in the factored form and *(n/2)*×$2^{n/2}$ literals in the SOP form.

Factor Form

- Factored forms can be graphically represented as labeled trees, called factoring trees, in which each internal node including the root is labeled with either + or ×, and each leaf has a label of either a variable or its complement
  - Example: factoring tree of *((a'+b)cd+e)(a+b')+e'*



Multi-Level Logic Minimization

- Basic techniques in Boolean network manipulation:
  - structural manipulation (change network topology)
  - node simplification (change node functions)
    - node minimization using don't cares

Multi-Level Logic Minimization Structural Manipulation

Restructuring Problem: Given initial network, find best network.

Example:

$f_1 = abcd+abce+ab'cd'+ab'c'd'+a'c+cdf+abc'd'e'+ab'c'df'$

$f_2 = bdg+b'dfg+b'd'g+bd'eg$

minimizing,

$f_1 = bcd+bce+b'd'+a'c+cdf+abc'd'e'+ab'c'df'$

$f_2 = bdg+dfg+b'd'g+d'eg$

factoring,

$f_1 = c(b(d+e)+b'(d'+f)+a')+ac'(bd'e'+b'df')$

$f_2 = g(d(b+f)+d'(b'+e))$

decompose,

$f_1 = c(b(d+e)+b'(d'+f)+a')+ac'x'$

$f_2 = gx$

$x = d(b+f)+d'(b'+e)$

Two problems:

- find good common subfunctions
- effect the division

Multi-Level Logic Minimization Structural Manipulation

Basic operations:

1. Decomposition (for a single function)

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

⇓

$f = xy+x'y' \quad x = ab \quad y = c+d$

2. Extraction (for multiple functions)

$f = (az+bz')cd+e \quad g = (az+bz')e' \quad h = cde$

⇓

$f = xy+e \quad g = xe' \quad h = ye \quad x = az+bz' \quad y = cd$

3. Factoring (series-parallel decomposition)

$f = ac+ad+bc+bd+e$

⇓

$f = (a+b)(c+d)+e$

Multi-Level Logic Minimization Structural Manipulation

Basic operations (cont'd):

4. Substitution

$f = a+bc \quad g = a+b$

⇓

$f = g(a+c) \quad g = a+b$

5. Collapsing (also called elimination)

$f = ga+g'b \quad g = c+d$

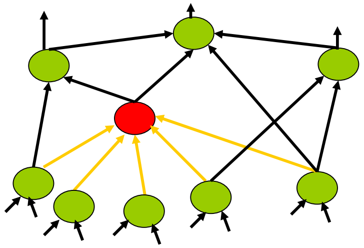
⇓

$f = ac+ad+bc'd' \quad g = c+d$

Note: "division" plays a key role in all these operations

Multi-Level Logic Minimization
Node Simplification

- Goal: For any node of a given Boolean network, find a least-cost SOP expression among the set of permissible functions for the node
  - Don't care computation + two-level logic minimization



combinational Boolean network

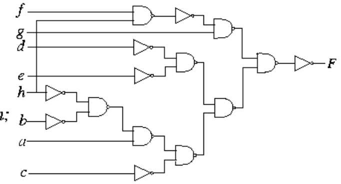
Combinational Logic Minimization

- **Two-level:** minimize #product terms and #literals
  - E.g., $F = x_1'x_2'x_3' + x_1'x_2'x_3 + x_1x_2'x_3' + x_1x_2'x_3 + x_1x_2x_3' \Rightarrow F = x_2' + x_1x_3'$
- **Multi-level:** minimize the # literals (area minimization)
  - E.g., equations are optimized using a smaller number of literals

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

logic optimization →

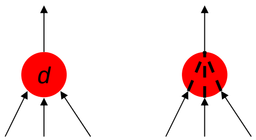
$t1 = d + e;$
$t2 = b + h;$
$t3 = a\ t2 + c;$
$t4 = t1\ t3 + f\ g\ h;$



subject graph for the optimized equations

Timing Analysis and Optimization

- Delay model at logic level
  - Gate delay model (our focus)
    - Constant gate delay, or pin-to-pin gate delay
    - Not accurate
  - Fanout delay model
    - Gate delay considering fanout load (#fanouts)
    - Slightly more accurate
  - Library delay model
    - Tabular delay data given in the cell library
      - Determine delay from input slew and output load
      - Table look-up + interpolation/extrapolation
    - Accurate

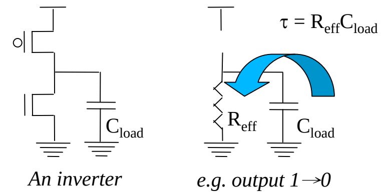


Timing Analysis and Optimization
Gate Delay

The delay of a gate depends on:

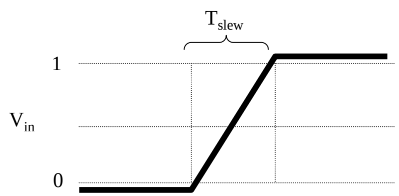
1. Output Load

- Capacitive loading ∝ charge needed to swing the output voltage
- Due to interconnect and logic fanout



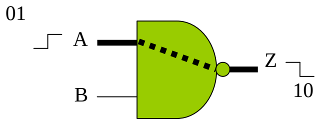
2. Input Slew

- Slew = transition time
- Slower transistor switching ⇒ longer delay and longer output slew



Timing Analysis and Optimization
Timing Library

- Timing library contains all relevant information about each standard cell
  - E.g., pin direction, clock, pin capacitance, etc.
- Delay (fastest, slowest, and often typical) and output slew are encoded for each input-to-output path and each pair of transition directions
- Values typically represented as 2 dimensional look-up tables (of output load and input slew)
  - Interpolation is used




```
Path(
    inputPorts(A),
    outputPorts(Z),
    inputTransition(01),
    outputTransition(10),
    "delay_table_1",
    "output_slew_table_1"
);
```

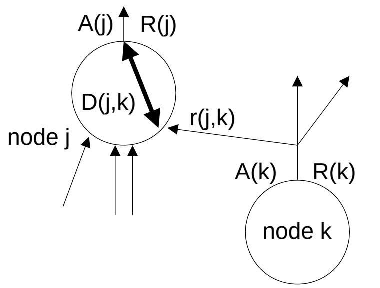
"delay_table_1"

Output load (nF) / Input slew (ns)

Input slew (ns) \ Output load (nF)	1.0	2.0	4.0	10.0
0.1	2.1	2.6	3.4	6.1
0.5	2.4	2.9	3.9	7.2
1.0	2.6	3.4	4.0	8.1
2.0	2.8	3.7	4.9	10.3

Static Timing Analysis

- Arrival time: the time signal arrives
  - Calculated from input to output in the topological order
- Required time: the time signal must ready (e.g., due to the clock cycle constraint)
  - Calculated from output to input in the reverse topological order
- Slack = required time – arrival time
  - Timing flexibility margin (positive: good; negative: bad)

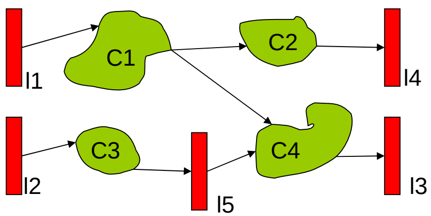


A(j): arrival time of signal j
R(k): required time or for signal k
S(k): slack of signal k
D(j,k): delay of node j from input k

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

Static Timing Analysis

- Arrival times known at I_1 and I_2
- Required times known at I_3, I_4, and I_5
- Delay analysis gives arrival and required times (hence slacks) for combinational blocks C_1, C_2, C_3, C_4



Static Timing Analysis

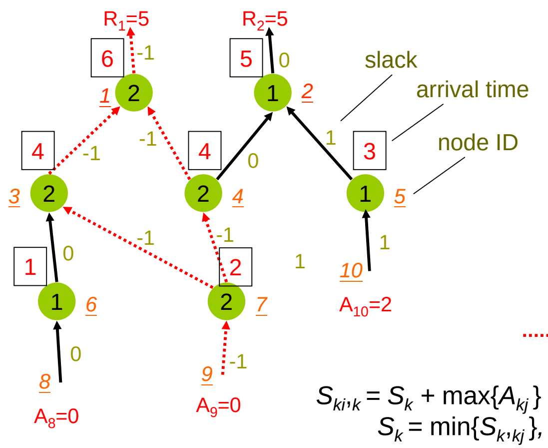
- **Arrival time** can be computed in the topological order from inputs to outputs
  - When a node is visited, its output arrival time is: the max of its fanin arrival times + its own gate delay
- **Required time** can be computed in the reverse topological order from outputs to inputs
  - When a node is visited, its input required time is: the min of its fanout required times – its own gate delay

Static Timing Analysis

- Example

$A_1 = 6$ $R_1 = 5$
$A_2 = 5$ $R_2 = 5$

$S_1 = -1$ $R_3 = 3$
$S_2 = 0$ $R_7 = 1$
$S_{3,1} = -1$ $R_9 = -1$
$S_{4,1} = -1$
$S_{4,2} = 0$
$S_{5,2} = 1$
$S_{6,3} = 0$
$S_{7,3} = -1$
$S_{7,4} = -1$
$S_{7,5} = 1$
$S_{8,6} = 0$
$S_{9,7} = -1$



critical path edges

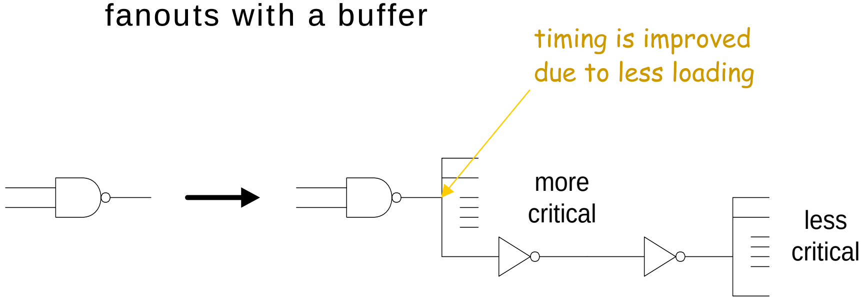
$$S_{k_i,k} = S_k + \max\{A_{k_j}\} - A_{k_i},\ k_j, k_i \in \text{fanin}(k)$$
$$S_k = \min\{S_{k,k_j}\},\ k_j \in \text{fanout}(k)$$

Timing Optimization

- Identify timing critical regions
- Perform timing optimization on the selected regions
  - E.g., gate sizing, buffer insertion, fanout optimization, tree height reduction, etc.

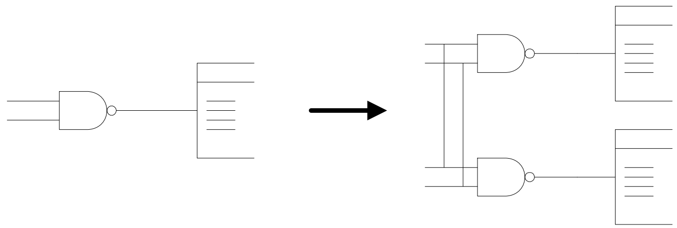
Timing Optimization

- Buffer insertion
  - Divide the fanouts of a gate into critical and non-critical parts, and drive the non-critical fanouts with a buffer



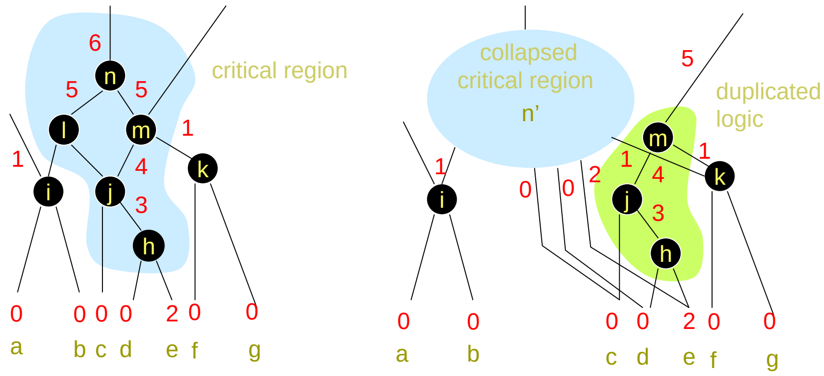
Timing Optimization

- Fanout optimization
  - Split the fanouts of a gate into several parts. Each part is driven by a copy of the original gate.



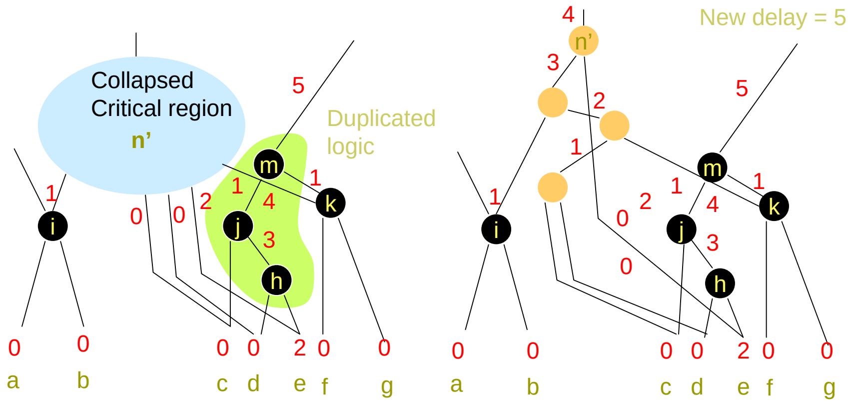
Timing Optimization

- Tree height reduction



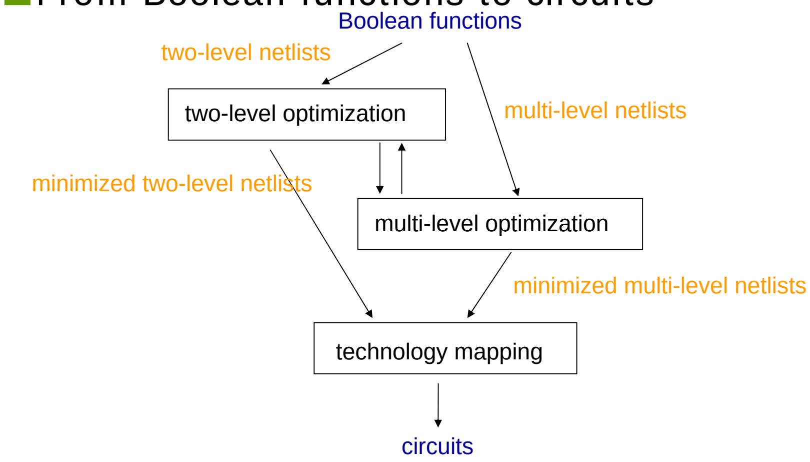
Timing Optimization

- Tree height reduction



Combinational Optimization

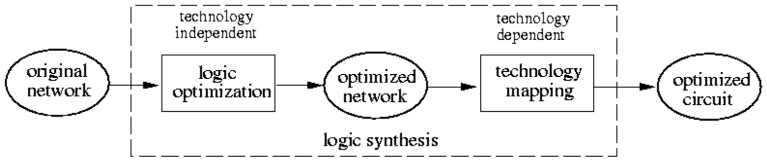
- From Boolean functions to circuits



Technology Independent vs. Dependent Optimization

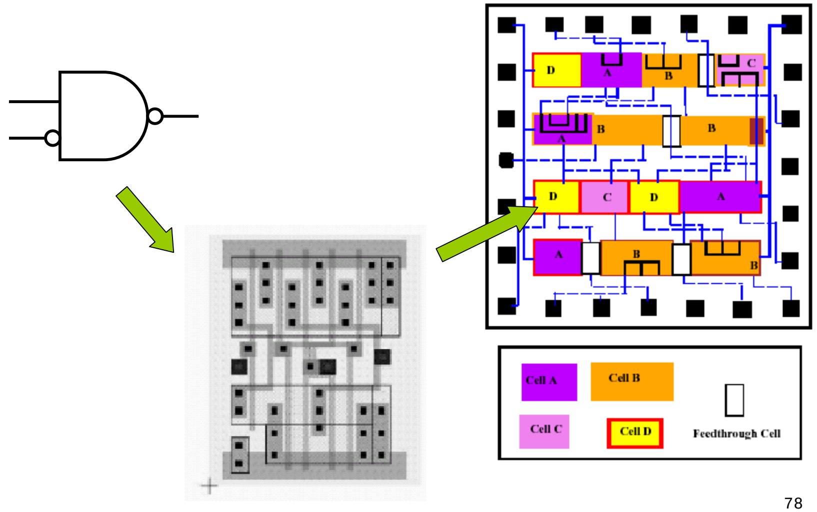
- Technology independent optimization produces a two-level or multi-level netlist where literal and/or cube counts are minimized
- Given the optimized netlist, its logic gates are to be implemented with library cells
- The process of associating logic gates with library cells is technology mapping
  - Translation of a technology independent representation (e.g. Boolean networks) of a circuit into a circuit for a given technology (e.g. standard cells) with optimal cost

Technology Mapping



- **Standard-cell technology mapping:** standard cell design
  - Map a function to a limited set of pre-designed library cells
- **FPGA technology mapping**
  - Lookup table (LUT) architecture:
    - E.g., Lucent, Xilinx FPGAs
    - Each lookup table (LUT) can implement all logic functions with up to k inputs (k = 4, 5, 6)
  - Multiplexer-based technology mapping:
    - E.g., Actel FPGA
    - Logic modules are constructed with multiplexers

Standard-Cell Based Design



Technology Mapping

- Formulation:
  - Choose base functions
    - Ex: 2-input NAND and Inverter
  - Represent the (optimized) Boolean network with base functions
    - Subject graph
  - Represent library cells with base functions
    - Pattern graph
    - Each pattern is associated with a cost depending on the optimization criteria, e.g., area, timing, power, etc.
- Goal:
  - Find a minimal cost covering of a subject graph using pattern graphs

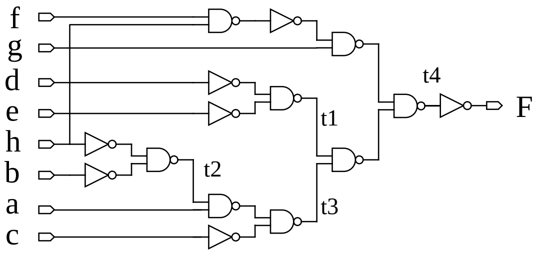
Technology Mapping

- Technology Mapping: The optimization problem of finding a minimum cost covering of the subject graph by choosing from a collection of pattern graphs of 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.

Technology Mapping

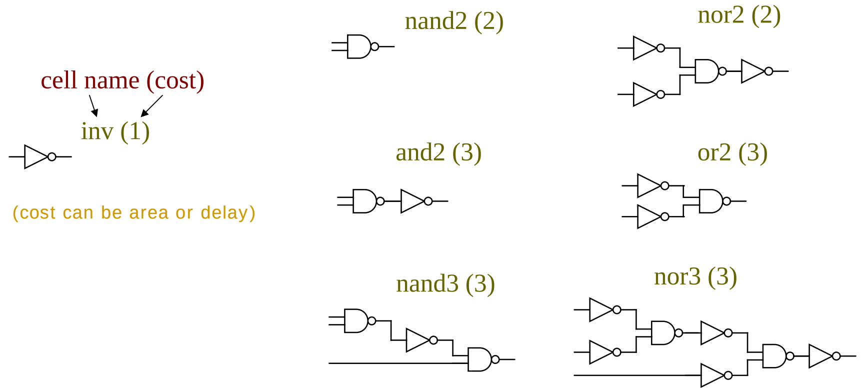
- Example
  - Subject graph

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



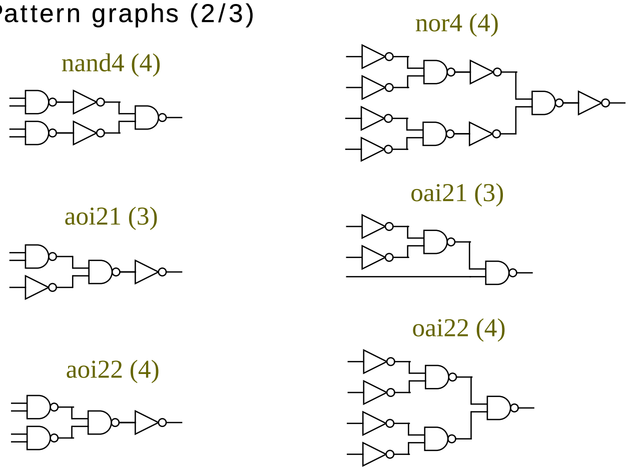
Technology Mapping

- Example
  - Pattern graphs (1/3)



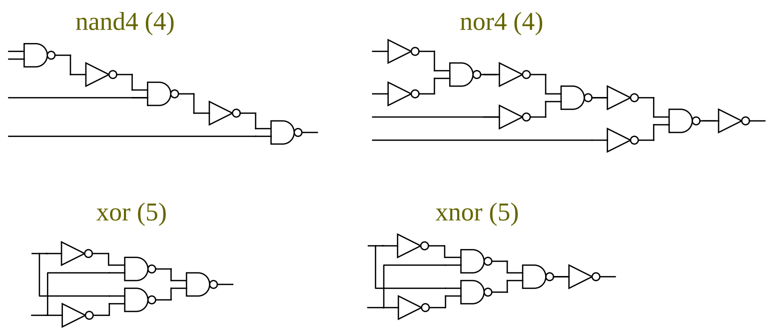
Technology Mapping

- Example
  - Pattern graphs (2/3)



Technology Mapping

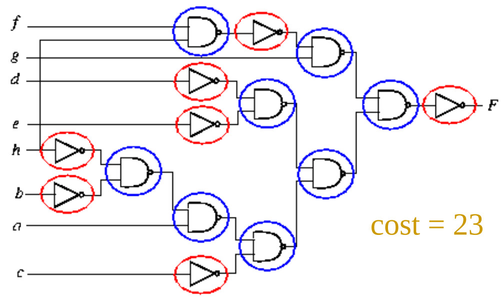
- Example
  - Pattern graphs (3/3)



Technology Mapping

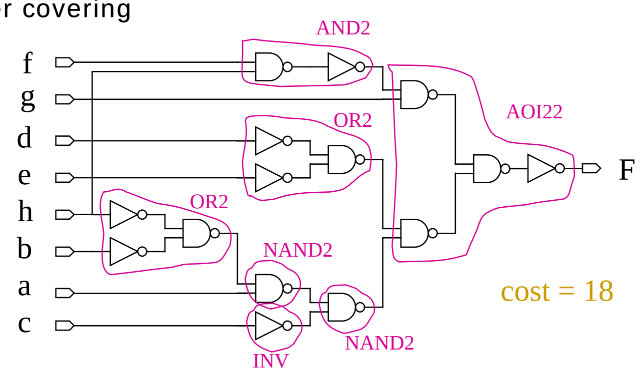
- Example
  - A trivial covering
    - Mapped into NAND2's and INV's
      - 8 NAND2's and 7 INV's at cost of 23

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



Technology Mapping

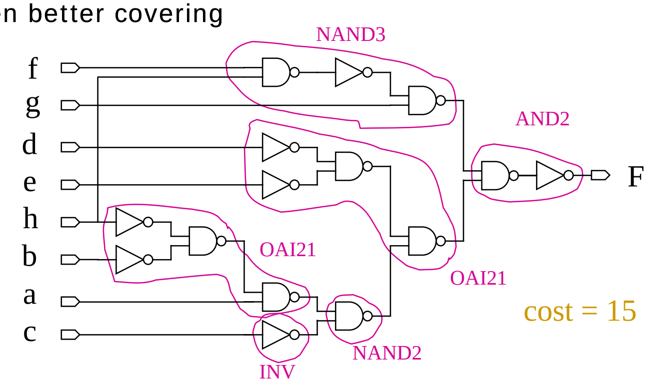
- Example
  - A better covering



For a covering to be legal, every input of a pattern graph must be the output of another pattern graph!

Technology Mapping

- Example
  - An even better covering



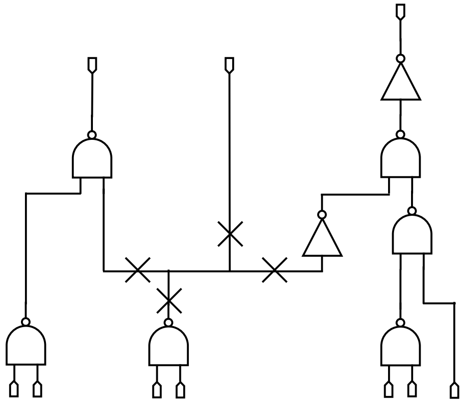
For a covering to be legal, every input of a pattern graph must be the output of another pattern graph!

Technology Mapping

- Complexity of covering on directed acyclic graphs (DAGs)
  - NP-complete
  - If the subject graph and pattern graphs are trees, then an efficient algorithm exists (based on dynamic programming)

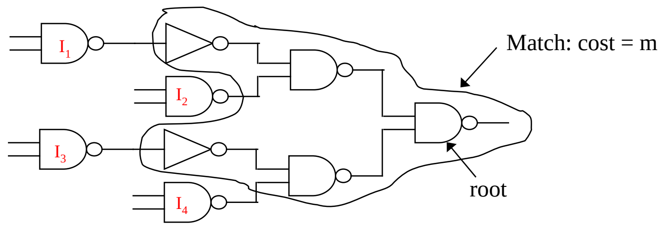
Technology Mapping DAGON Approach

- Partition a subject graph into trees
  - Cut the graph at all multiple fanout points
- Optimally cover each tree using dynamic programming approach
- Piece the tree-covers into a cover for the subject graph



Technology Mapping DAGON Approach

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

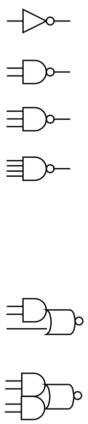


$C(root) = m + C(I_1) + C(I_2) + C(I_3) + C(I_4)$
cost of a leaf (i.e. primary input) = 0

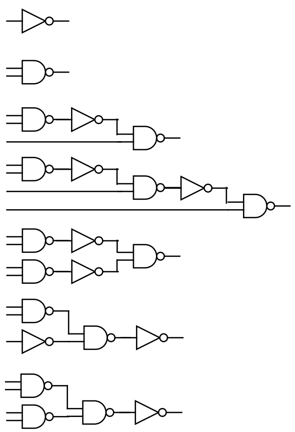
Technology Mapping DAGON Approach

- Example
  - Library

INV	2	a'
NAND2	3	(ab)'
NAND3	4	(abc)'
NAND4	5	(abcd)'
AOI21	4	(ab+c)'
AOI22	5	(ab+cd)'



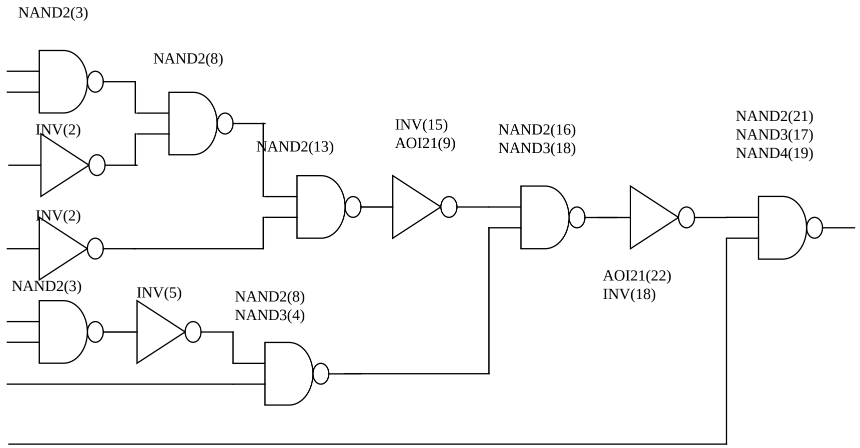
library element



base-function representation

Technology Mapping DAGON Approach

- Example



Technology Mapping DAGON Approach

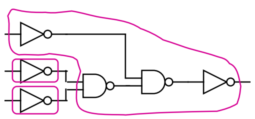
- Complexity of DAGON for tree mapping is controlled by finding all sub-trees of the subject graph isomorphic to pattern trees
- Linear complexity in both the size of subject tree and the size of the collection of pattern trees
  - Consider library size as constant

Technology Mapping DAGON Approach

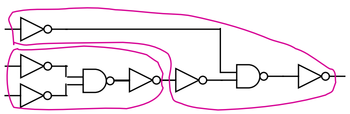
- Pros:
  - Strong algorithmic foundation
  - Linear time complexity
    - Efficient approximation to graph-covering problem
  - Give locally optimal matches in terms of both area and delay cost functions
  - Easily "portable" to new technologies
- Cons:
  - With only a local (to the tree) notion of timing
    - Taking load values into account can improve the results
  - Can destroy structures of optimized networks
    - Not desirable for well-structured circuits
  - Inability to handle non-tree library elements (XOR/XNOR)
  - Poor inverter allocation

Technology Mapping DAGON Approach

- DAGON can be improved by
  - Adding a pair of inverters for each wire in the subject graph
  - Adding a pattern of a wire that matches two inverters with zero cost



2 INV
1 AIO21



2 NOR2

Available Logic Synthesis Tools

- Academic CAD tools:
  - Espresso (heuristic two-level minimization, 1980s)
  - MIS (multi-level logic minimization, 1980s)
  - SIS (sequential logic minimization, 1990s)
  - ABC (sequential synthesis and verification system, 2005-)
    - http://www.eecs.berkeley.edu/~alanmi/abc/