ECE 667

Synthesis and Verification of Digital Systems

Click to add Text

ABC System

Combinational Logic Synthesis

Slides adapted from Alan Mishchenko, *UC Berkeley* 2010+

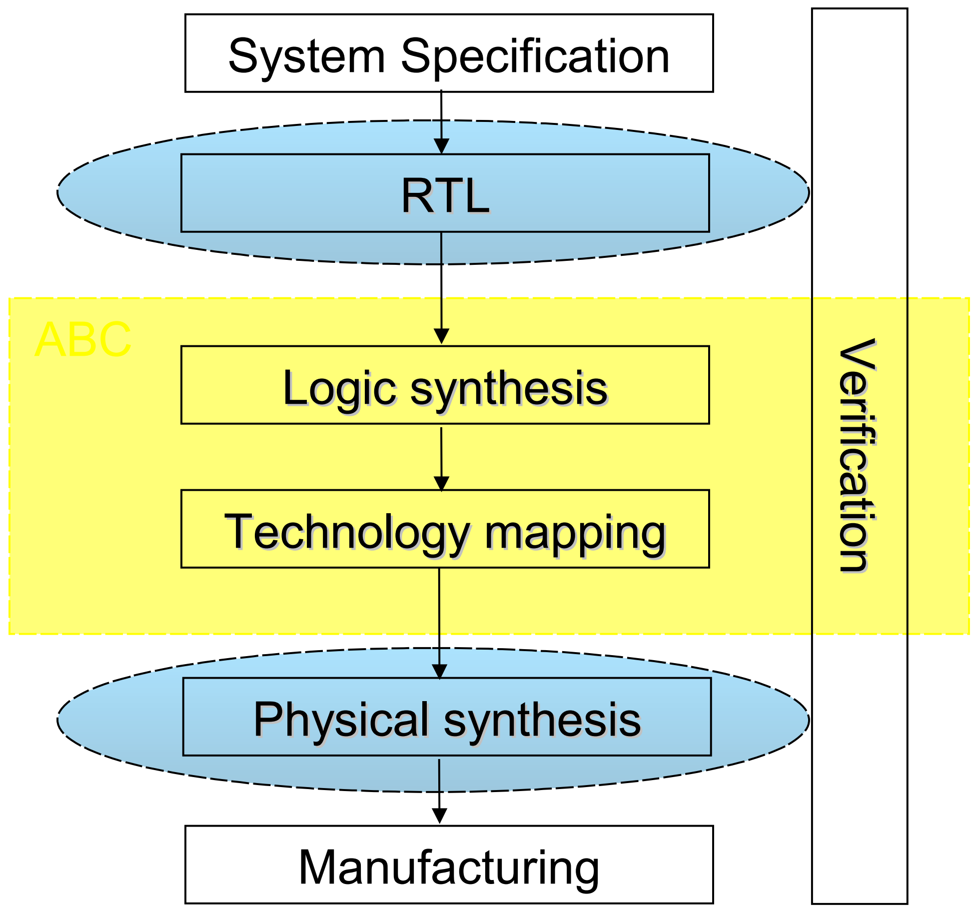
Outline

- ABC System
- And-Inverter Graph (AIG)
  - AIG construction
  - AIG optimization
    - Rewriting
    - Substitution
    - Redundancy removal
- Technology mapping
  - Boolean matching
  - Cut-based mapping
- Sequential optimization
  - Integration: logic optimization + mapping + retiming

What Is Berkeley ABC?

- A system for logic synthesis and verification
  - Fast
  - Scalable
  - High quality results (industrial quality)
  - Exploits synergy between synthesis and verification
- A programming environment
  - Open-source
  - Evolving and improving over time

Design Flow


System Specification
RTL
ABC
Logic synthesis
Technology mapping
Physical synthesis
Manufacturing
Verification

Areas Addressed by ABC

- Combinational synthesis
  - AIG rewriting
  - technology mapping
  - resynthesis after mapping
- Sequential synthesis
  - retiming
  - structural register sweep
  - merging seq. equiv. nodes

- Formal verification
  - combinational equivalence checking
  - bounded sequential verification
  - unbounded sequential verification
  - equivalence checking using synthesis history

ABC vs. Other Tools

- Industrial
  - + well documented, fewer bugs
  - - black-box, push-button, no source code, often expensive
- SIS
  - + traditionally very popular
  - - data structures / algorithms outdated, weak sequential synthesis
- VIS
  - + very good implementation of BDD-based verification algorithms
  - - not meant for logic synthesis, does not feature the latest SAT-based implementations
- MVSIS
  - + allows for multi-valued and finite-automata manipulation
  - - not meant for binary synthesis, lacking recent implementations

Existing Capabilities

Combinational logic synthesis
Fast, scalable, good quality

Technology mapping with structural choices
Cut-based, heuristic, good area/delay, flexible

ABC

Sequential verification
Integrated, interacts with synthesis

Sequential synthesis
Innovative, scalable, verifiable

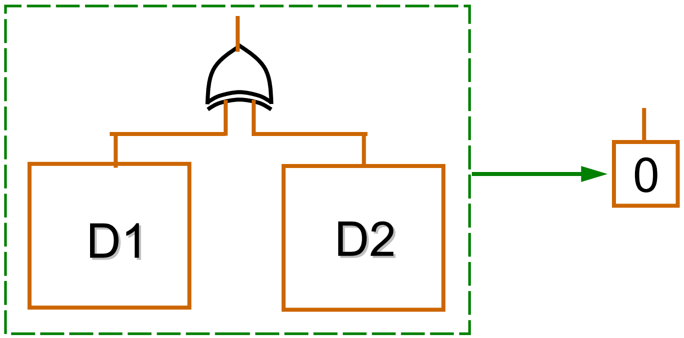
Formal Verification

- Equivalence checking
  - Takes two designs and makes a miter (AIG)
- Model checking *safety* properties
  - Takes design and property and makes a miter (AIG)

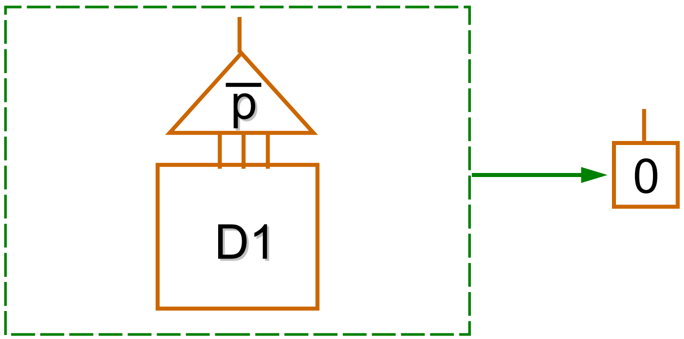
The goals are the same: to transform AIG until the output is proved constant 0

ABC won a model checking competition at CAV in August 2008

Equivalence checking



Property checking



And-Inverter Graphs (AIG)

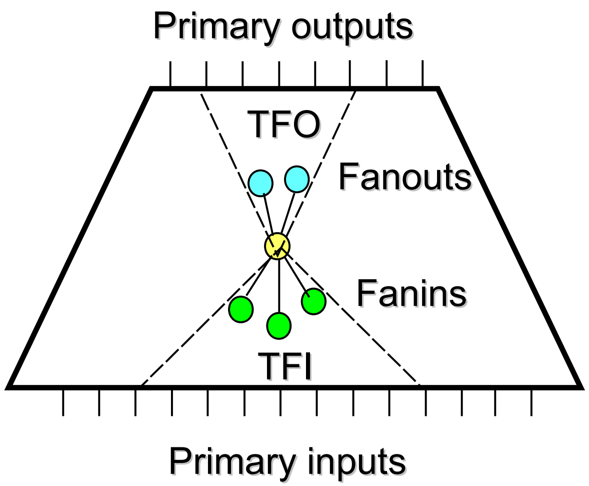
Click to add Text

And-Invert Graph (AIG)

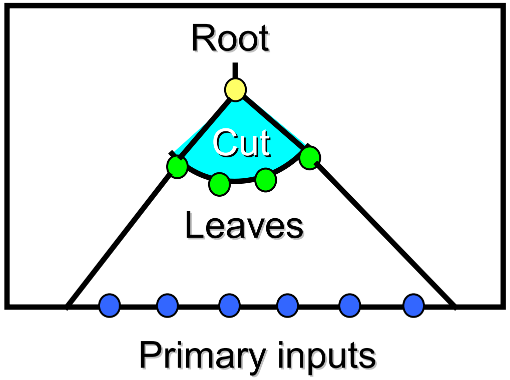
- AIG is a Boolean network with two types of nodes:
  - two-input ANDs, nodes
  - Inverters (NOT)
- Any Boolean function can be expressed using AIGs
  - For many practical functions AIGs are smaller than BDDs
  - Efficient graph representation (structural)
  - Very good correlation with design size
- AIGs are <u>not canonical</u>
  - For one function, there may be many structurally-different AIGs
  - Functional reduction and structural hashing can make them "canonical enough"

Terminology

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

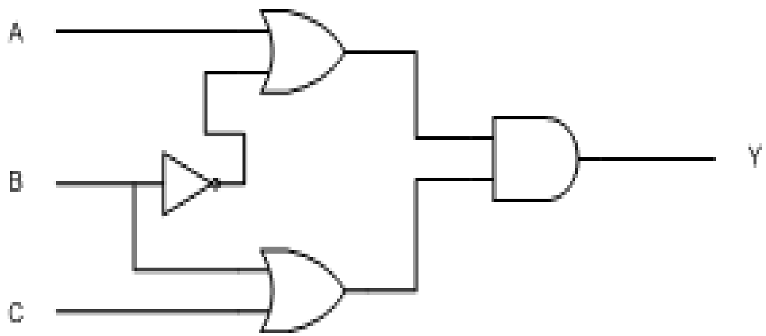


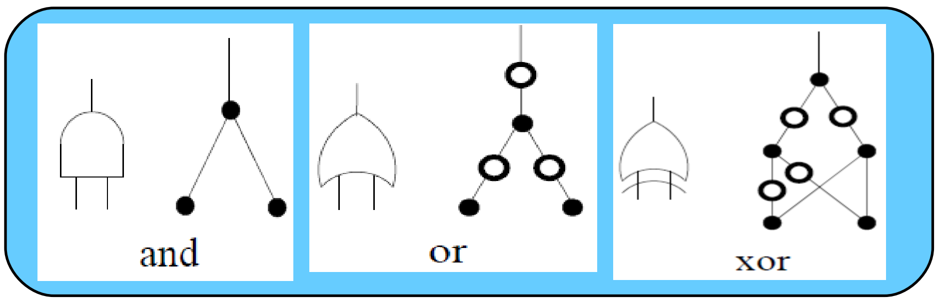
- Structural cut of a node
  - Cut is a boundary in the network separating the node from the PIs
  - Boundary nodes are the leaves
  - The node is the root of the cut
  - k-feasible cut has *k* or less leaves
  - Function of the cut is function of the root in terms of the leaves

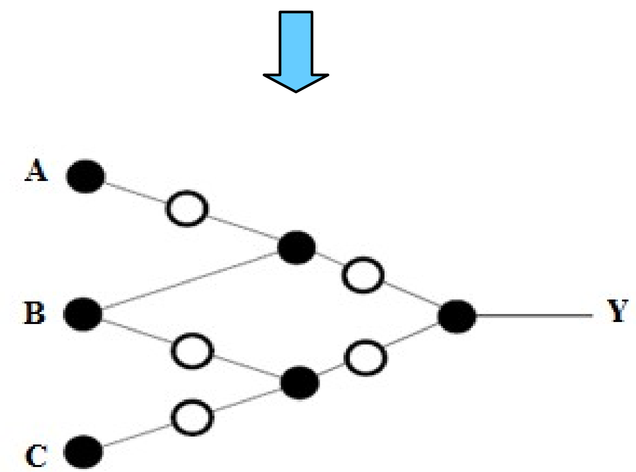


Create Starting AIG

- AIGs are constructed from the Boolean network and reduced to FRAIGs to minimize the AIG size.
- Constructed from the netlist available fror technology independent logic synthesis

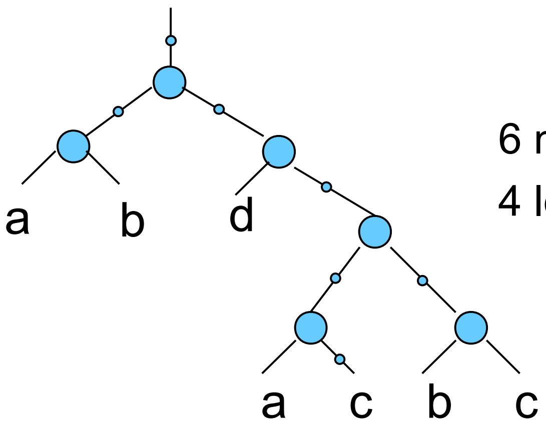




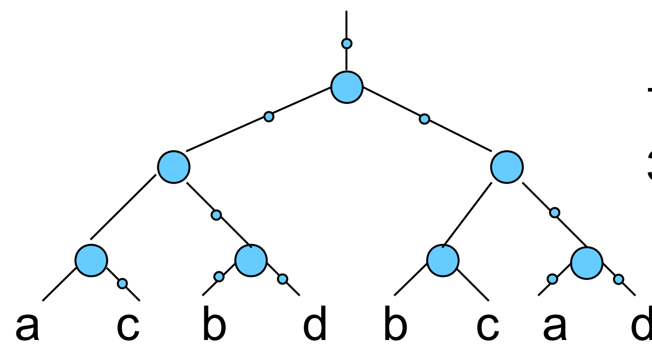


AIG Non-canonicity

- AIGs are **not** canonical
  - same function represented by two functionally equivalent AIGs with different structures
  - BDDs – canonical for same variable ordering
  - But they are “canonical enough” (A. Mishchenko)



6 nodes
4 levels

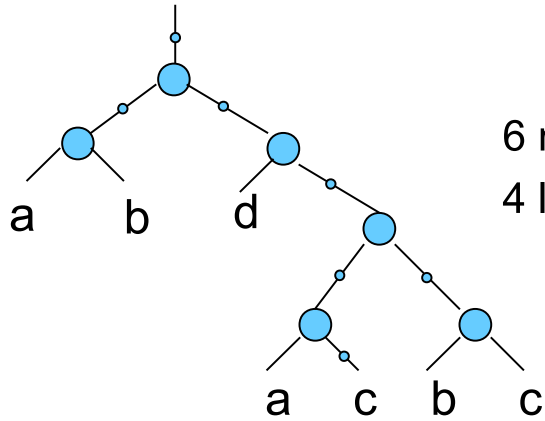


7 nodes
3 levels

AIG Example

cd \ ab	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	0	1	1	0
10	0	0	1	0

$F(a,b,c,d) = ab + d(ac'+bc)$

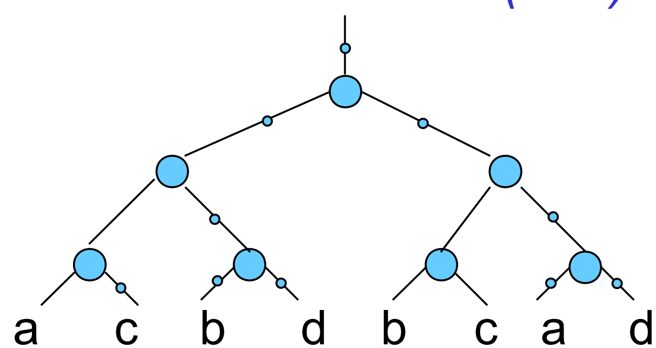


6 nodes

4 levels

cd \ ab	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	0	1	1	0
10	0	0	1	0

$$F(a,b,c,d) = ac'(b'd')' + cb(a'd')'$$
$$= ac'(b+d) + bc(a+d)$$

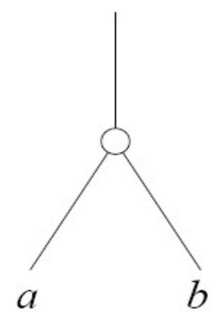


7 nodes

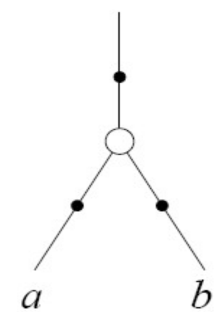
3 levels

Basic Logic Operations

- Converting logic function into AIG graph
  - Inversion $\neg a$ $\neg a$
  - Conjunction $a \wedge b$ (ab) $a \wedge b$
  - Disjunction $a \vee b$ $(a+b)$ $\neg(\neg a \wedge \neg b)$
  - Implication $a \Rightarrow b$ $\neg(a \wedge \neg b)$
  - Equivalence $a \Leftrightarrow b$ $\neg(a \wedge \neg b) \wedge \neg(\neg a \wedge b)$
  - a XOR b $\neg(\neg(a \wedge \neg b) \wedge \neg(\neg a \wedge b))$



$a \wedge b$

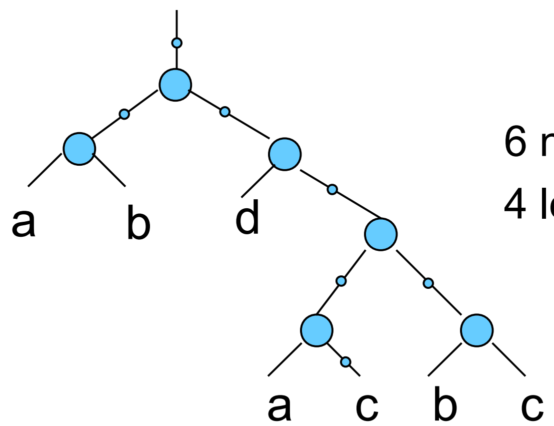


$a \vee b$

AIG Attributes

- AIG size
  - Measured by number of AND nodes

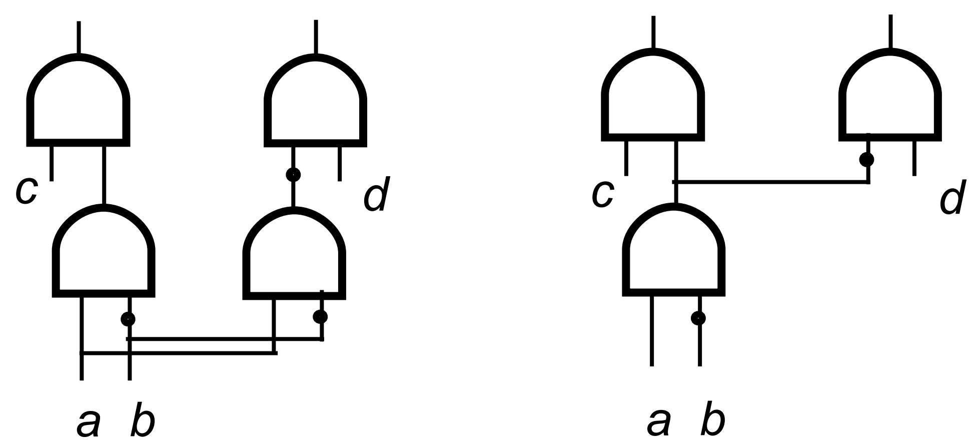
- AIG depth
  - Number of logic levels = number of AND-gates on longest path from a primary input to a primary output
  - The inverters are ignored when counting nodes and logic levels



6 nodes
4 levels

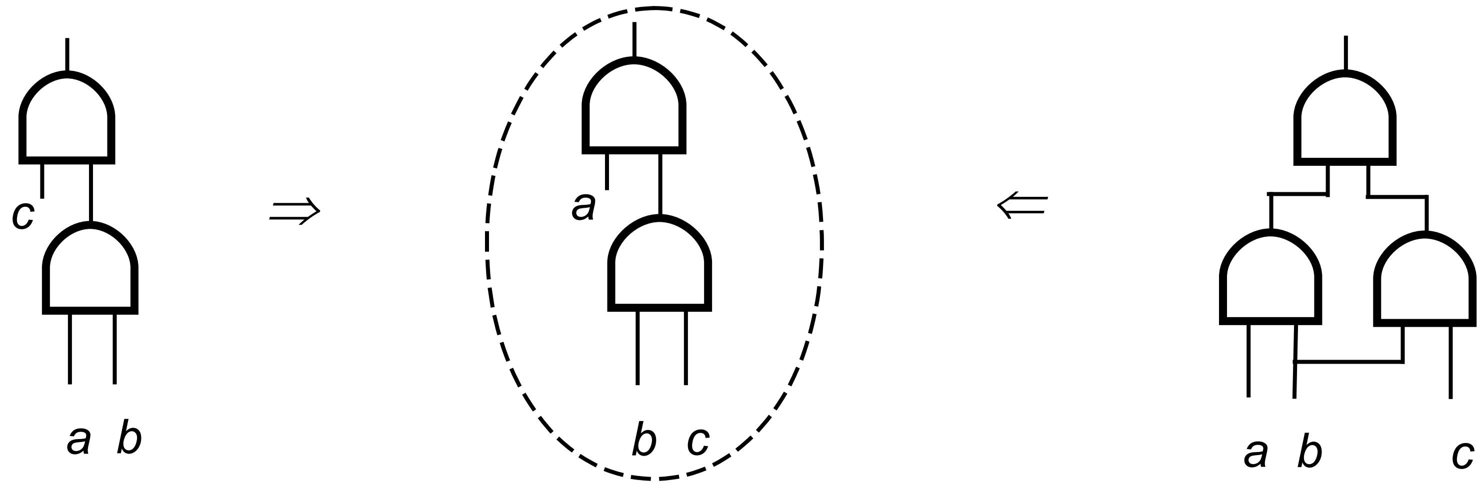
Structural Hashing (Strashing)

- When building AIGs, always add AND node
  - When an AIG is constructed without strashing, AND-gates are added one at a time without checking whether AND-gate with the same fanins already exists
- One-level strashing
  - when adding a new AND-node, check the hash table for a node with the same input pair (fanin)
  - if it exists, return it; otherwise, create a new node



Two-Level Structural Hashing

- When adding a new AND-node
  - Consider two levels of its predecessors
  - Hash the three AND-gates into a representative ("canonical") form
  - This offers partial canonicity

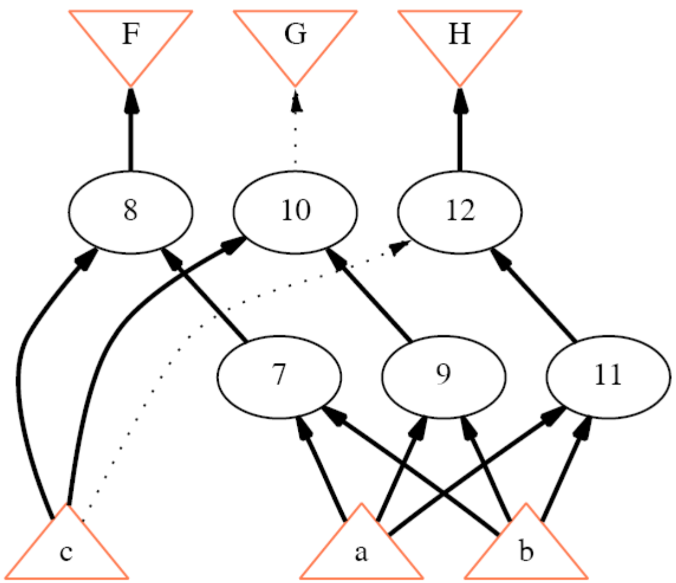


"canonical" form

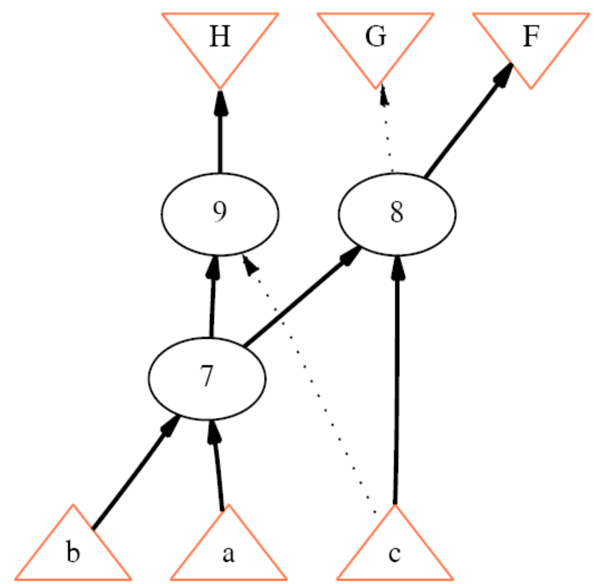
Strashing- example

F = abc   *G = (abc)'*   *H = abc'*

Initial AIG

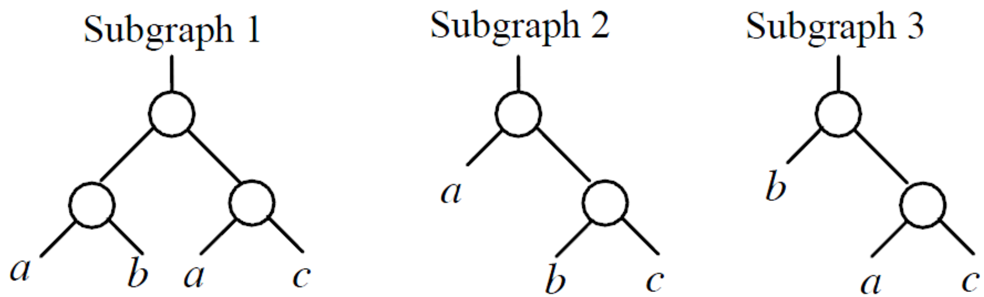


AIG after strashing



Functional Reduction

- AIGs are not canonical – may contain syntactically distinct but functionally equivalent (redundant) internal nodes.



Different AIG structures for function $F = abc$.

- Operations on such AIGs are inefficient and time consuming.
- Detecting and merging functionally equivalent nodes is called *functional reduction*.
- Achieved by bit-parallel simulation + SAT (explain !)

"DAG-Aware AIG Rewriting A Fresh Look at Combinational Logic Synthesis" - Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, Robert

AIG Functional Reduction - Previous Work

AIGs are first built using structural hashing (*strashing*) and post-processed optionally to enforce functional reduction.

- BDD Sweeping [1]
  - Constructs BDDs of the network nodes in terms of primary inputs (PI) and intermediate variables
  - A pair of network nodes with same BDDs are merged
  - Resource limits restrict BDD size
- SAT Sweeping [2]
  - Achieves the same by solving topologically ordered SAT problems designed to <u>prove or disprove equivalence</u> of cut-point pairs
  - Equivalence candidate pairs are detected using random or exhaustive <u>simulation</u> (bit-parallel)

[1] A. Kuehlmann, et.al., "Robust boolean reasoning for equivalence checking and functional property verification", *IEEE Trans. CAD, Vol. 21(12), 2002*

[2] A. Kuehlmann, "Dynamic Transition Relation Simplification for Bounded Property Checking". *Proc. ICCAD '04.*

Functional Reduction (FRAIG)

- Outline of the algorithm:
  - When a new AND-node is added, perform structural hashing
  - When a new node is created, check for the node with the same functionality (up to complementation)
    - If such a node exists, return it
    - If the node does not exist, return the new node
- The resulting functionally-reduced AIGs are “canonical” in the following sense
  - Each node has a unique functionality
  - Structural representation of each function is not fixed
    - Adding nodes in different order may lead to a different graph
    - They can be always mapped to a representative form

AIG Rewriting

Fast greedy algorithm to minimize AIG size (# nodes)

- Iteratively selects AIG subgraphs up to cut size 4
- Replaces each subgraph by precomputed subgraphs (same funcion and number of levels)
- Uses NPN classes, hashed by truth table

Use of 4-input cuts

- The cut computation starts at the PIs and proceeds in topological order
- For an internal node *n* with two fanins, *a* and *b*, the cuts *C*(*n*) are computed by *merging* the cuts of *a* and *b.*
- For each cut, all pre-computed subgraphs are considered . The new subgraph that leads to the largest improvement at a node if chosen.

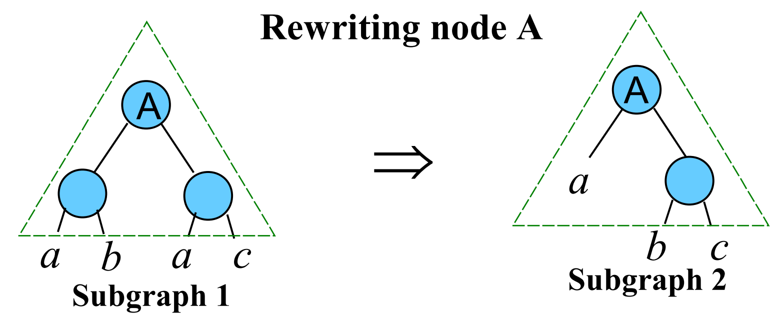
Delay-aware AIG rewriting

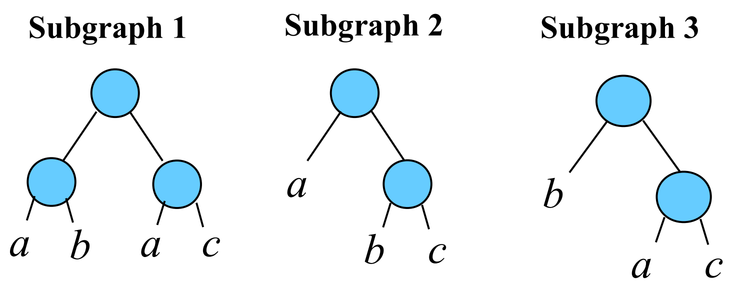
- AIG refactoring; AIG balancing

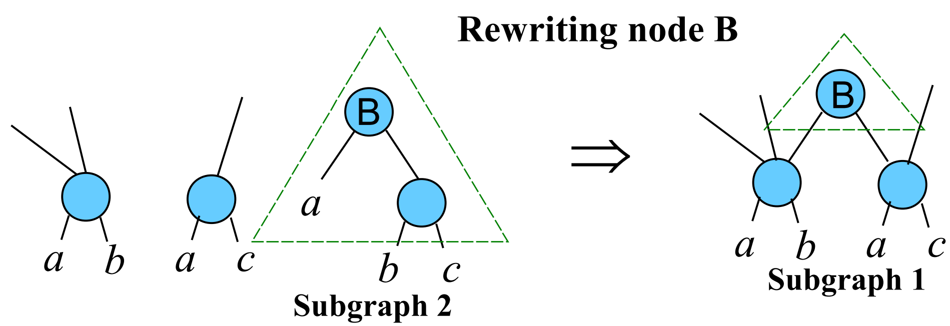
Combinational Synthesis

- AIG rewriting minimizes the number of AIG nodes without increasing the number of AIG levels
- Pre-computing AIG subgraphs
  - Consider function $f = abc$

Rewriting AIG subgraphs



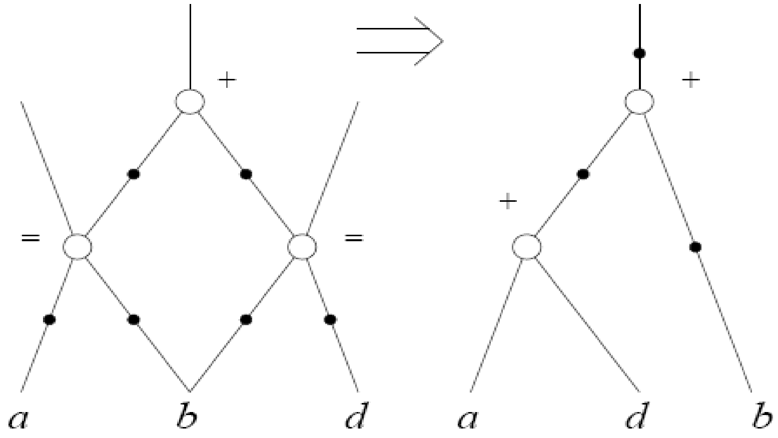




In both cases one node is saved

AIG Optimization

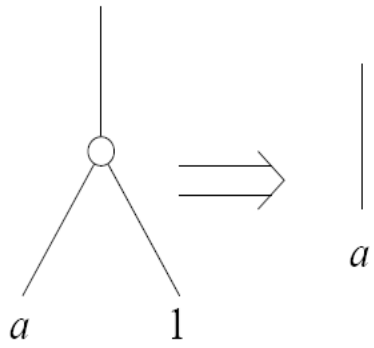
- AIG optimization is based on AIG rewriting, from one form to a simpler form



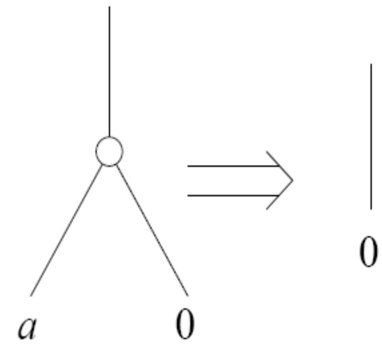
distributivity law

(a+b)(b+d) = ad+b

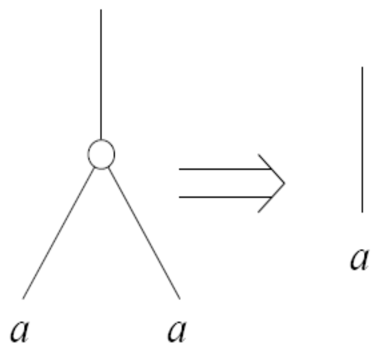
Level -1 Optimization



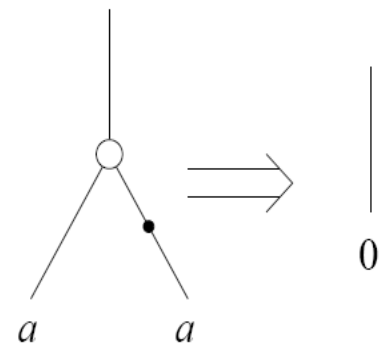
$a * 1 = 1$



$a * 0 = 0$

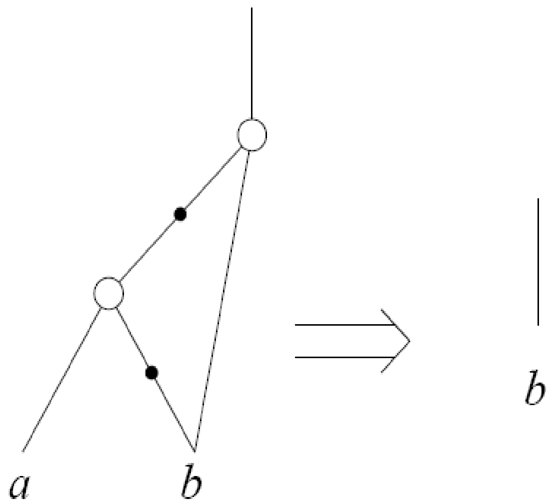


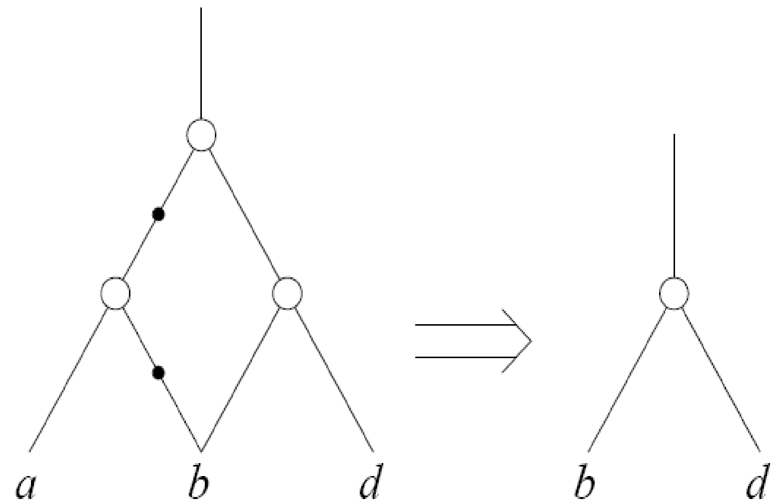
$a * a = a$



$a * \neg a = 0$

Level 2 Optimization



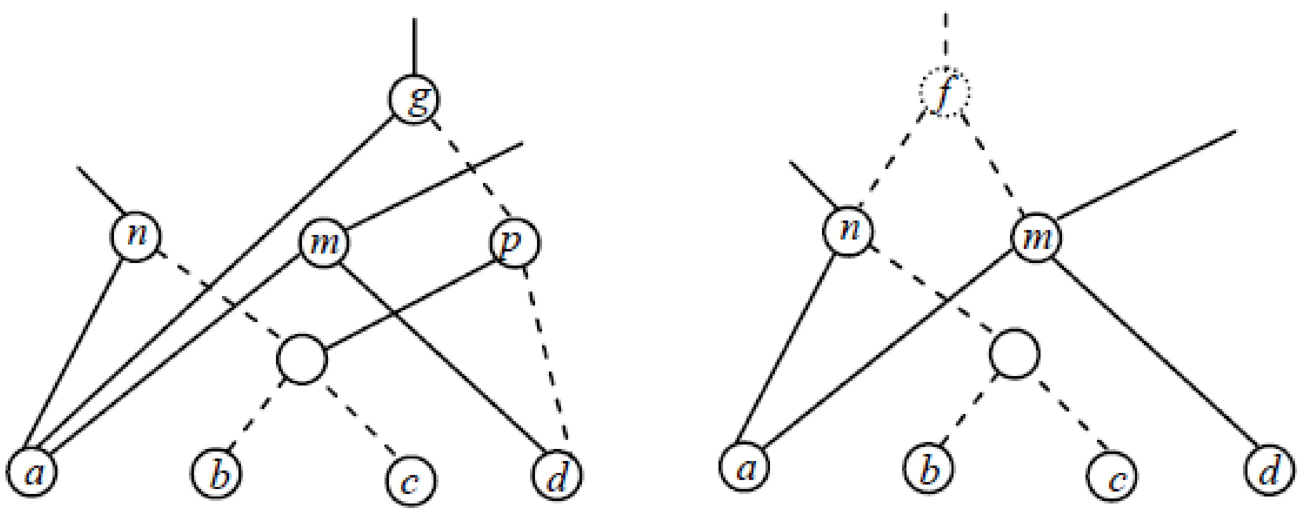


$(\neg a+b)b = b$

$((\neg a+b)b)\ d = bd$

Resubstitution

- Express the function of the node using other nodes (divisors).
- 0-level resubstitution: replace a logic cone (MFFC) by another node
- 1-level resubstitution: replace function of the node by two existing nodes + new node (AND). Example:
  - Replace function $g = a(b+c+d)$ by $f' = n + m = a(b+c) + (a\ d) = a(b+c+d)$ in the context of the network where $n = a\ (b+c)$ and $m = a\ d$.

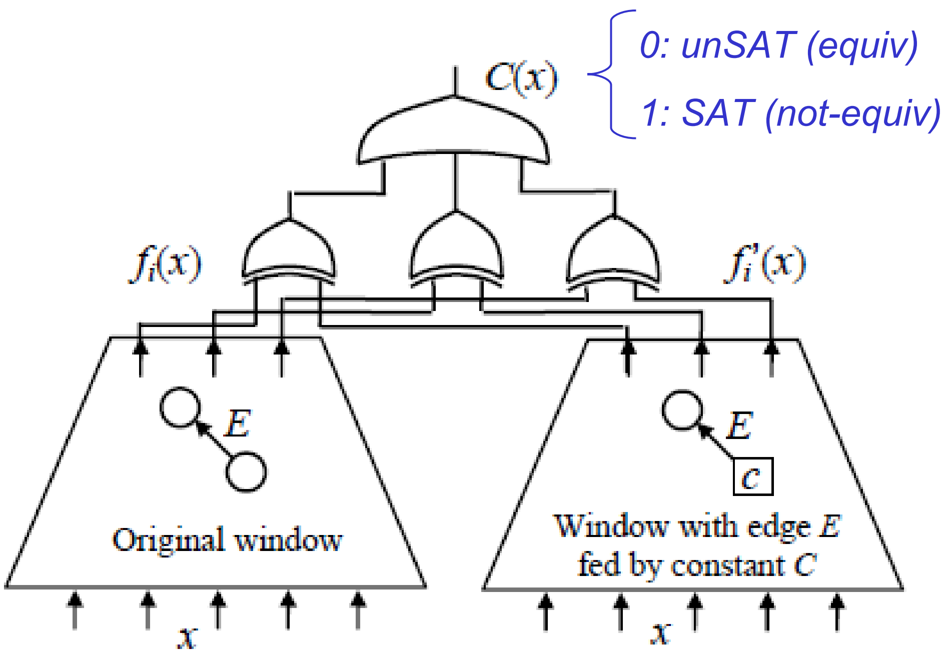


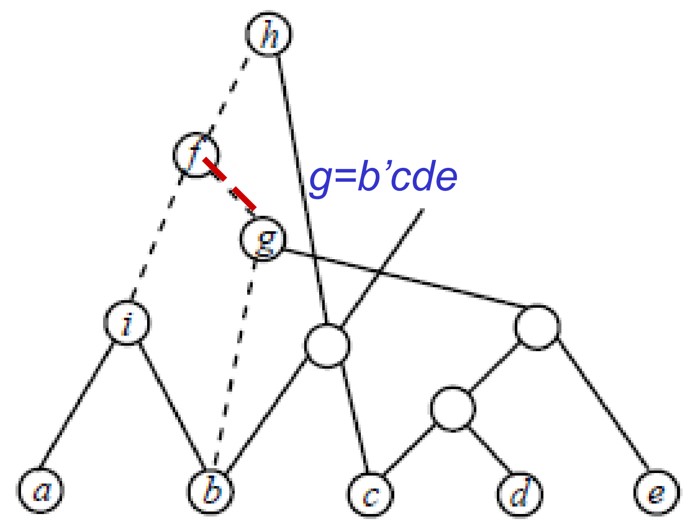
AIG is reduced by 1 node (p)

Redundancy Removal

- Fast bit-parallel, random *simulation* used for early detection of non-redundancy
- SAT used to prove or disprove redundancy (equivalence)
- Edge $g \rightarrow f$ is redundant (remove it, set $g=0$)

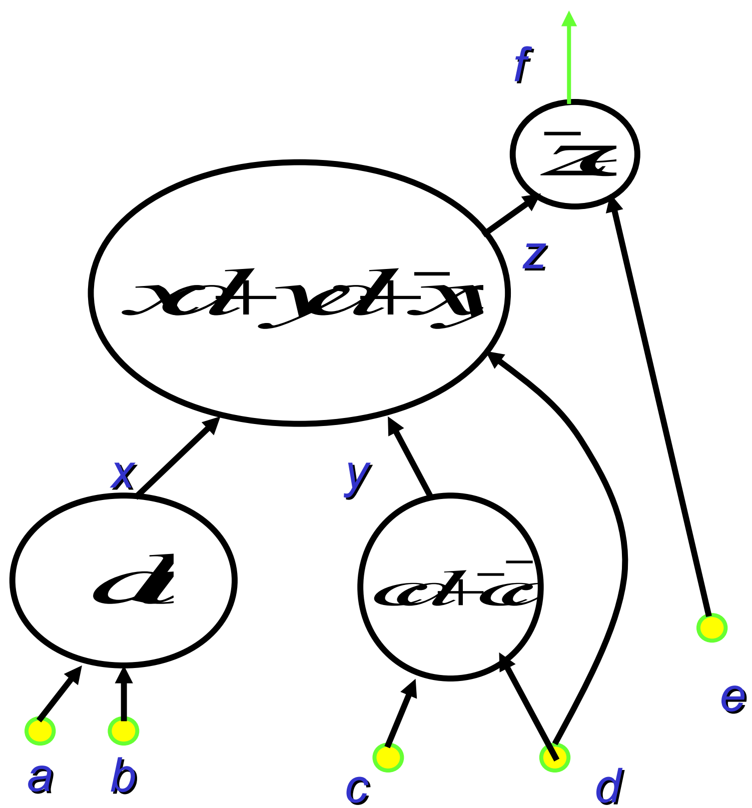
  $h = f'bc = (ab + b'cde)bc = abc$



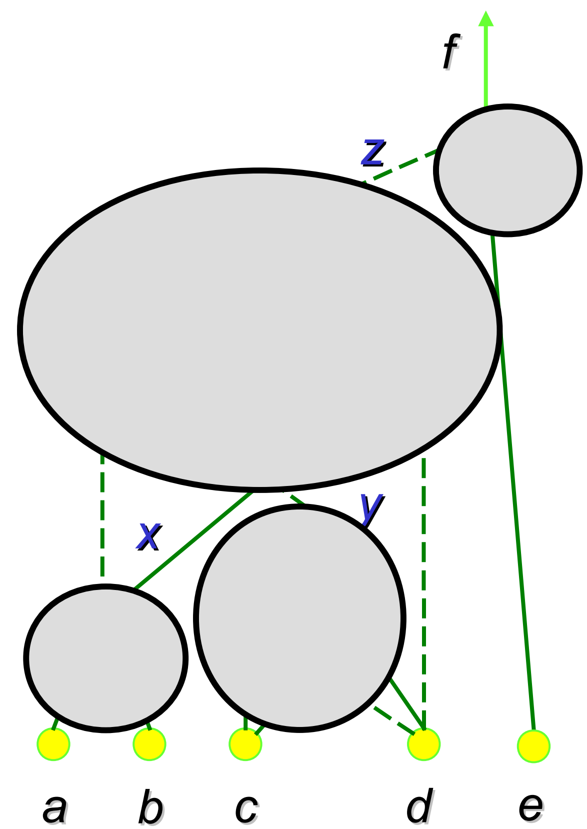


How Is ABC Different From SIS?

Boolean network in SIS



Equivalent AIG in ABC



AIG is a Boolean network of 2-input AND nodes and invertors (dotted lines)

Comparison of Two Synthesis Systems

"Classical" synthesis (SIS)

- Boolean network
- Network manipulation (algebraic)
  - Elimination
  - Factoring/Decomposition
  - Speedup
- Node minimization
  - Espresso
  - Don't cares computed using BDDs
  - Resubstitution
- Technology mapping
  - Tree based

ABC "contemporary" synthesis (ABC)

AIG network

DAG-aware AIG rewriting (Boolean)

Several related algorithms

Rewriting

Refactoring

Balancing

Speedup

Node minimization

Boolean decomposition

Don't cares computed using simulation and SAT

Resubstitution with don't cares

Technology mapping

Cut based with choice nodes

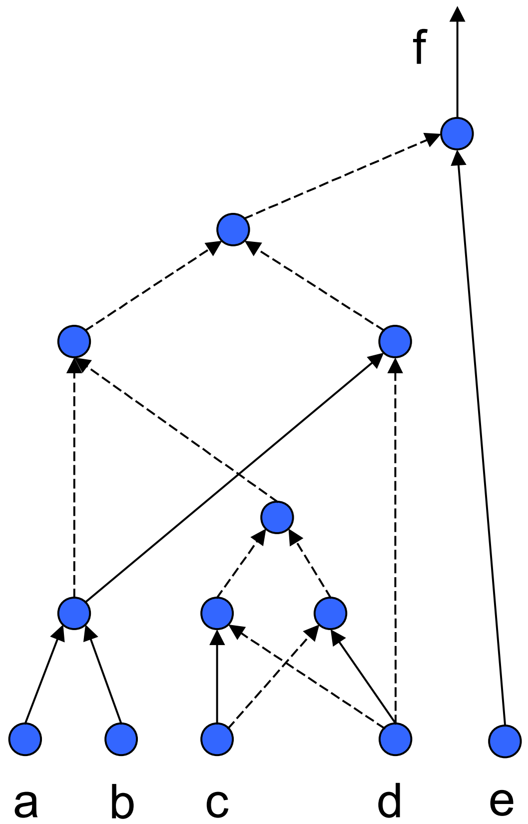
Cut-based Technology Mapping

Click to add Text

Technology Mapping

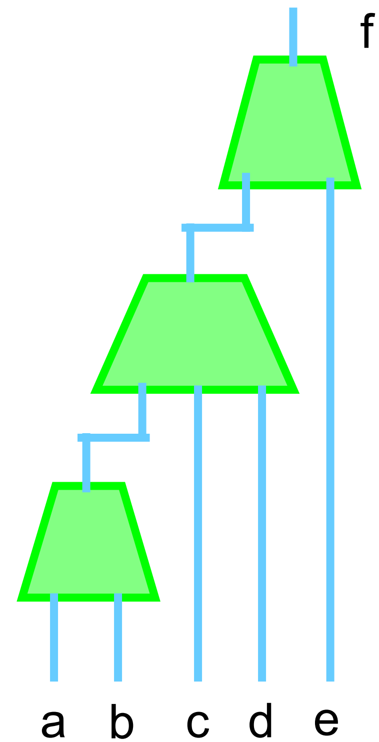
Input: A Boolean network (And-Inverter Graph)

Output: A netlist of *K*-LUTs implementing AIG and optimizing some cost function



The subject graph

Technology Mapping

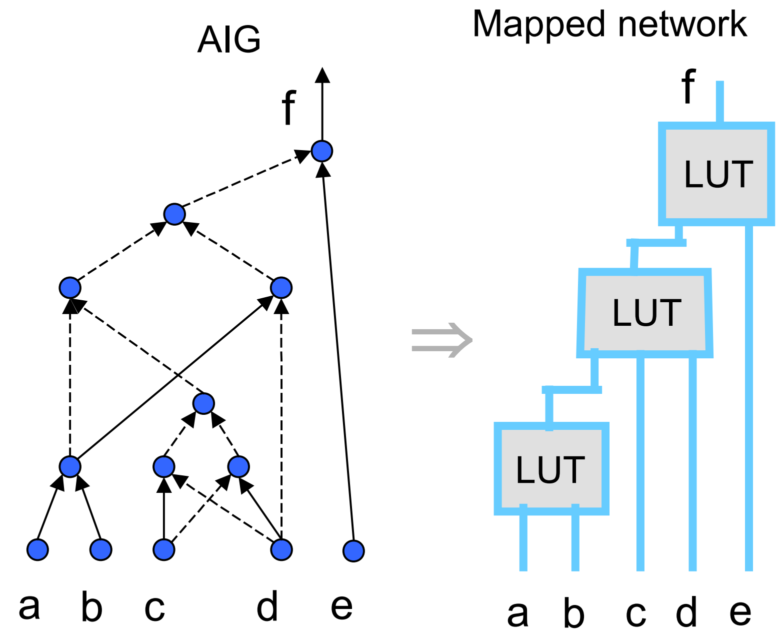


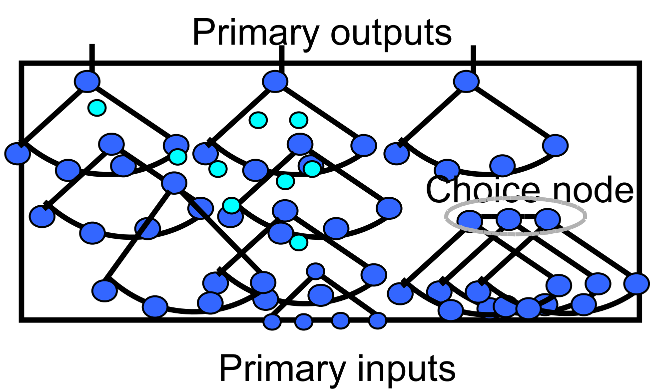
The mapped netlist

Mapping in a Nutshell

- AIGs represent logic functions
  – A good subject graph for mapping
- Technology mapping expresses logic functions to be implemented
  – Uses a description of the technology
- Technology
  – Primitives with delay, area, etc

- Structural mapping
  – Computes a cover of AIG using primitives of the technology (standard cell or LUT)
- 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)





LUT Mapping Algorithm (min delay)

Input: Structural representation of the circuit
(AIG or Boolean network)

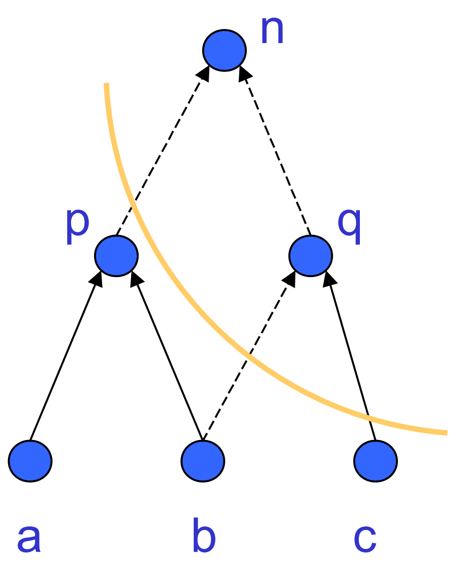
1. Compute all *k*-feasible cuts for each node and match them against gates from library
   - FPGA: structural matching (*k*-input LUTs)
   - ASIC: functional matching (truth tables)
2. Compute best arrival time at each node
   - In topological order (from PI to PO)
     compute the depth of all cuts and choose the best one
3. Perform area recovery
4. Chose the best cover
   - In reverse topological order (from PO to PI) choose best cover

Output: Mapped netlist

Structural Cuts in AIG

A cut of 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.

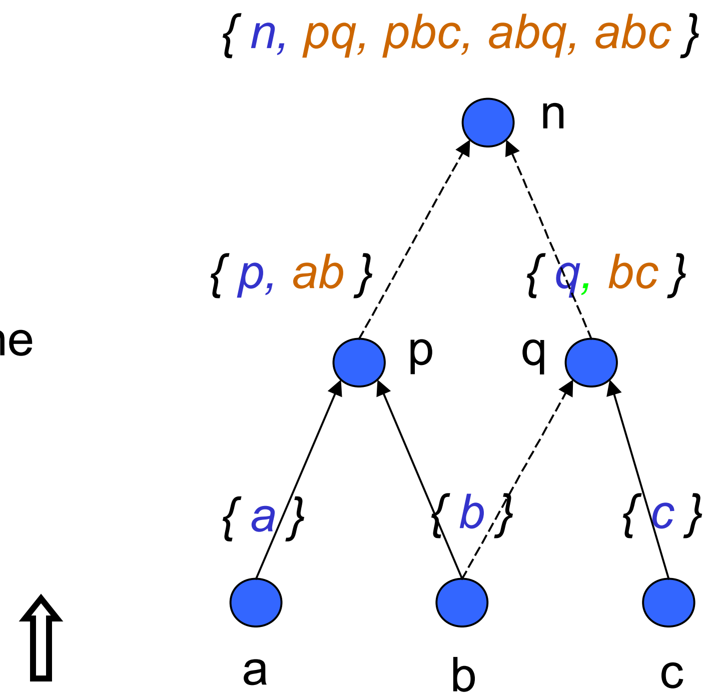


The set *{pbc}* is a 3-feasible cut of node *n*. (It is also a 4-feasible cut.)

k-feasible cuts are important in LUT mapping because the logic between root *n* and the cut leaves *{pbc}* can be replaced by a *3*-LUT.

Exhaustive Cut Enumeration

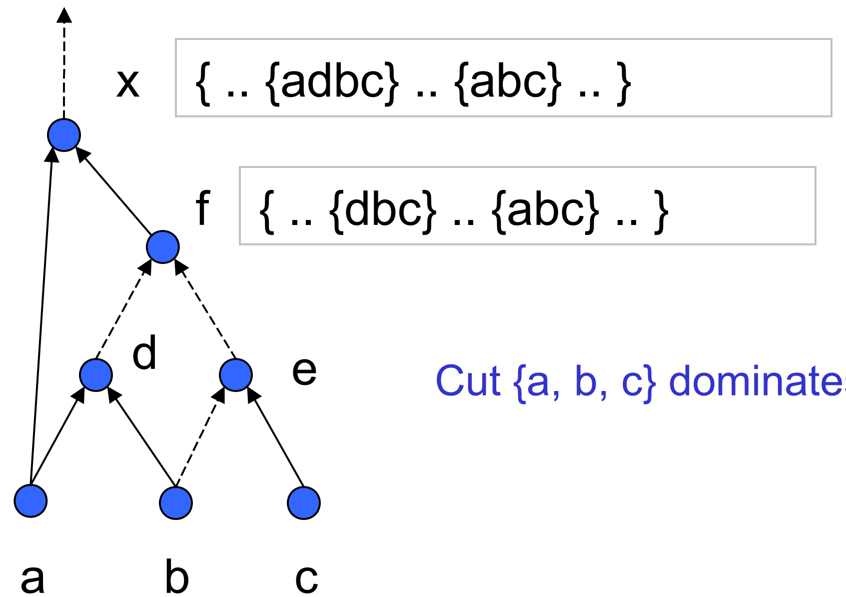
- All *k*-feasible cuts are computed in one pass over the AIG
  - Assign elementary cuts for primary inputs
  - For each internal node
    - merge the cut sets of children
    - remove duplicate cuts
    - add the elementary cut composed of the node itself



Computation is done bottom-up, from PIs to Pos.
Any cut that is of size greater than *k* is discarded

Cut Filtering

Bottom-up cut computation in the presence of re-convergence might produce *dominated* cuts

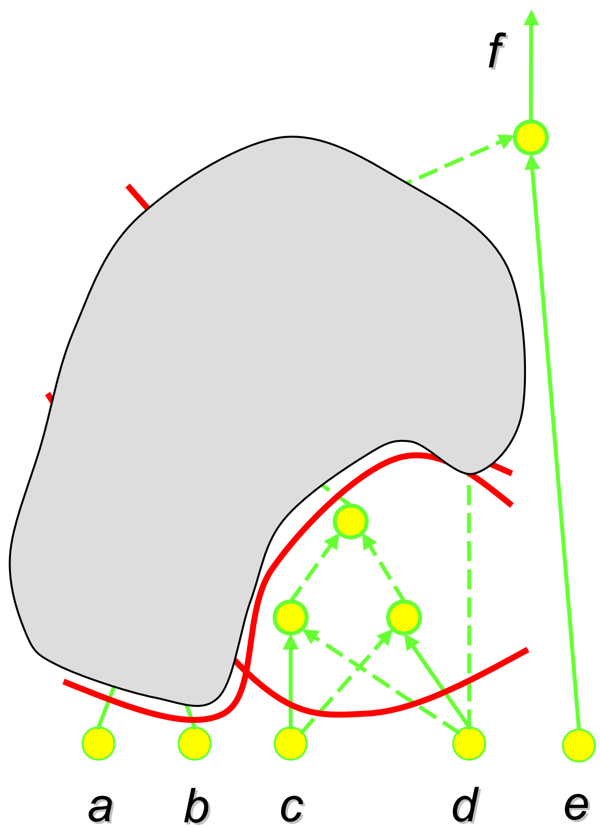


Cut {a, b, c} dominates cut {a, d, b, c}

- The "good" cut *{abc}* is present
- But the "bad" cut *{adbc}* may be propagated further (a run-time issue)
- It is important to discard dominated cuts ***quickly***

One AIG Node – Many Cuts

Combinational AIG



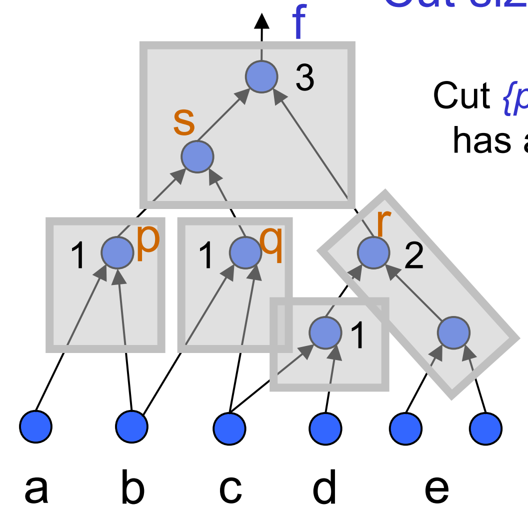
Different cuts for the same node

- Manipulating AIGs in ABC
  - Each node in an AIG has many cuts
  - Each cut is a different SIS node
  - There are no a priori fixed boundaries
- Implies that AIG manipulation with cuts is equivalent to working on *many* Boolean networks at the same time

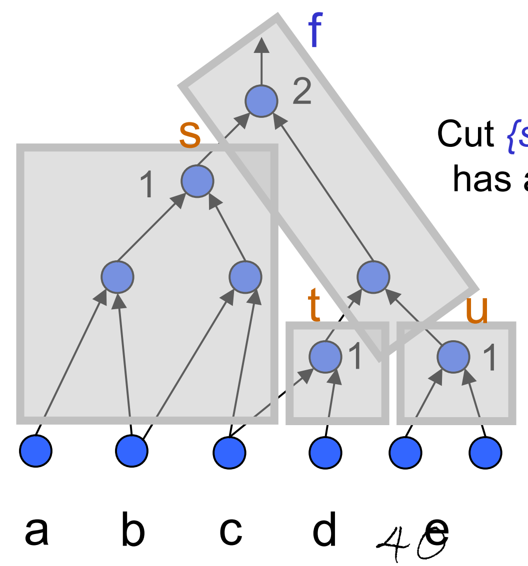
Delay-Optimal Mapping

- Input:
  - AIG and *k*-cuts computed for all nodes
- Algorithm:
  - For all nodes in a topological order
    - Compute arrival time of each cut using fanin arrival times
    - Select one cut with min arrival time
    - Set the arrival time of the node to be the arrival time of this cut
- Output:
  - Delay-optimal mapping for all nodes

Cut size *k* = 3



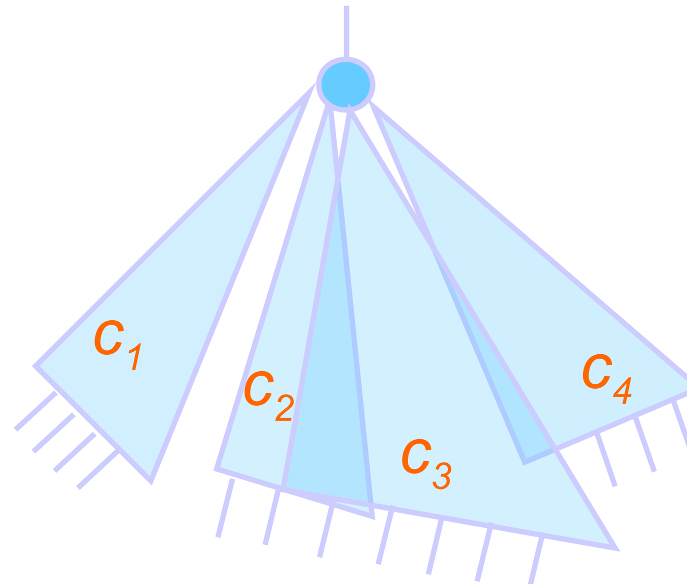
Cut *{pqr}* of node f has arrival time 3



Cut *{stu}* of node f has arrival time 2

Selecting Delay Optimal Cuts

- Computing Boolean function of a cut
  - Express the root of the cut as *f (leaves)*
- Matching cuts with the target device
  - <u>ASIC</u>: associate the cut with a gate from the library and look up its delay
  - <u>FPGA</u>: assign a *k*-feasible cut with a *k*-input LUT (delay and area are const)
- Assigning arrival times:
  for each node, from PIs to POs
  - Compute the arrival times of each cut
  - Select the best cut for optimum delay
  - When arrival times are equal, use area as a tie-breaker
  - Compute arrival times at the outputs



If $T_{c2} < T_{c3} < T_{c1} < T_{c4}$

C_2 is the best cut

Boolean Matching (standard cells)

- Comparing the Boolean function of the cut with those of the library gates
  - Represent the function of the cut output as truth table disregarding interconnect structure of internal nodes
  - Compare to truth tables of gates from library
  - Uses phase assignment
- All Boolean function with *k* variables are divided into *N*-equivalence classes

- NPN equivalence
  - Two Boolean function are *NPN equivalent* if one of them can be derived from another by selectively complementing inputs (N), permuting inputs (P) and optionally complementing output (N)

$f = x_1 x'_3 + x_2$ *and* $g = x_3 x'_1 + x_2$

are *N-equivalent* (input complementation)

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

NPN equivalence

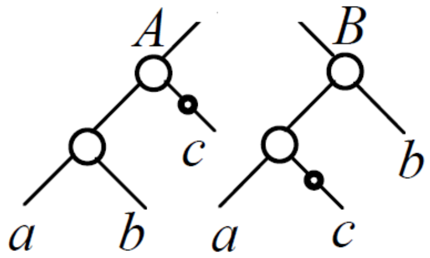
- Functions F and G are NPN equivalent if

  F can be derived from G by selectively complementing the inputs (N), permuting the inputs (P), and optionally complementing the output (N).

Examples:

$F1 = (a\ b)\ c'$ and $F2 = (a\ c')\ b$

are *P-equivalent* (permutation)



$f = x_1x'_3 + x_2$ and $g = x_3x'_1 + x_2$

are *N-equivalent* (input complementation)

N-Equivalence

function $f = x_1\overline{x}_3 + x_2$

Function $f = x_1x'_3 + x_2$ represented by bit-string <00111011>

Phase <001> transforms the truth table <00111011> into <00110111>

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Truth Table of f

c_1	c_2	c_3	Truth Table	Integer
0	0	0	<00111011>	59
0	**0**	**1**	**<00110111>**	**55**
0	1	0	<11001110>	206
0	1	1	<11001101>	205
1	0	0	<10110011>	179
1	0	1	<01110011>	115
1	1	0	<11101100>	236
1	1	1	<11011100>	220

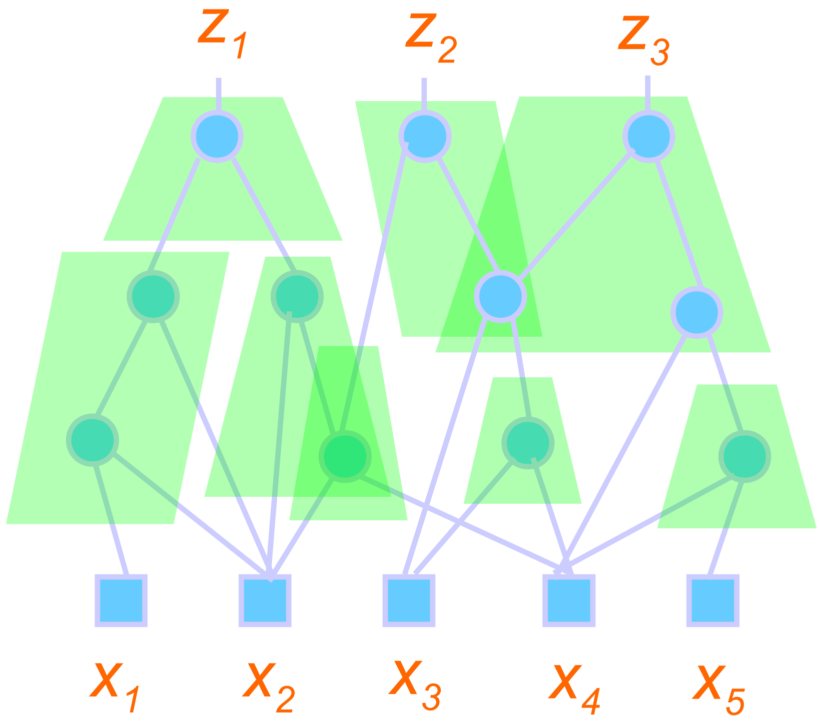
Canonical form of f

Canonical form: representative of N-equivalence class, phase assignment with smallest integer value (here <00110111>=55)

ABC pre-computes truth tables of all gates from the library and their N canonical forms.

Selecting Final Mapping (Covering)

- Once the best matches are assigned to each node
- Going from POs to PIs, extract the final mapping
  - Select the best match for each primary output node
  - Recursively, for each fanin of a selected match, select its best matches



Area Recovery During Mapping

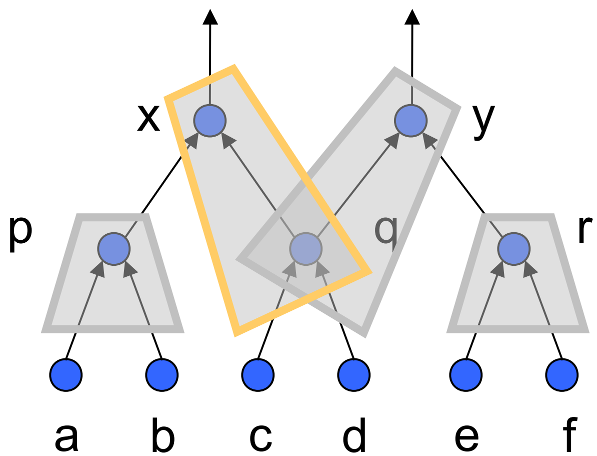
- Delay-optimal mapping is performed first
  - Best match is assigned at each node
  - Some nodes are used in the mapping; others are not used
- Arrival and required times are computed for all AIG nodes
  - Required time for all used nodes is determined
  - If a node is not used, its required time is set to $+\infty$
- Slack is a difference between required time and arrival time
- If a node has *positive slack*, its current best match can be updated to reduce the total area of mapping
  - This process is called area recovery
- Exact area recovery is exponential in the circuit size
  - A number of area recovery heuristics can be used
- Heuristic area recovery is iterative
  - Typically involved 3-5 iterations
- Next, we discuss cost functions used during area recovery
  - They are used to decide what is the best match at each node

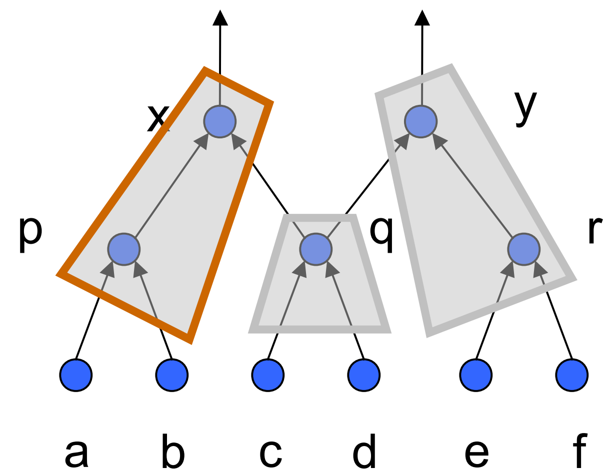
How to Measure Area?

Suppose we use the naïve definition:

Area (cut) = 1 + [Σ area (fanin)]

(assuming that each LUT has one unit of area)





Area of cut *{pcd}*
= 1 + [1 + 0 + 0] = 2

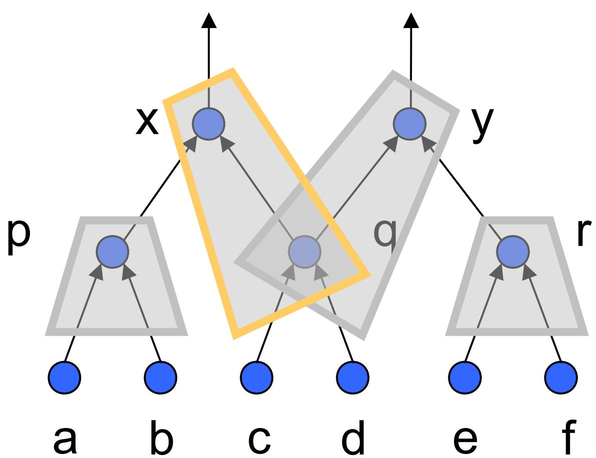
Area of cut *{abq}*
= 1 + [0 + 0 + 1] = 2

Naïve definition says both cuts are equally good in area

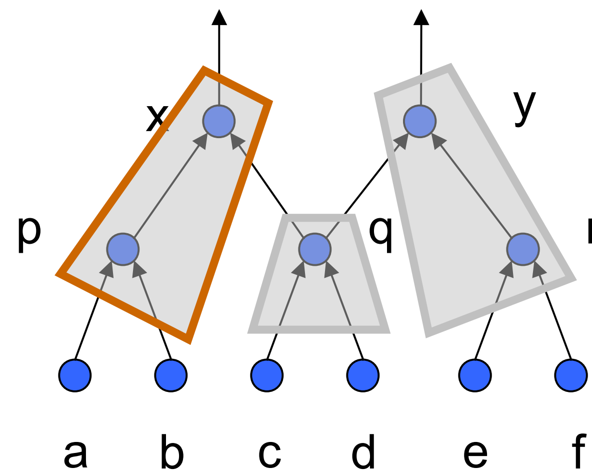
But this ignores sharing due to multiple fanouts

Area-flow

Area-flow (cut) = 1 + [Σ (area-flow (fanin) / fanout_num(fanin))]



Area-flow of cut *{pcd}*
= 1 + [1 + 0 + 0] = 2



Area-flow of cut *{abq}*
= 1 + [0/1 + 0/1 + ½] = 1.5

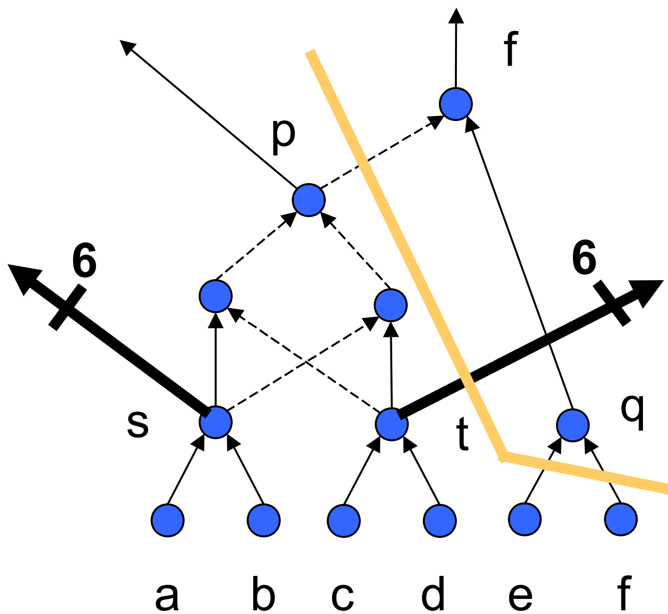
Area-flow recognizes that cut *{abq}* is better

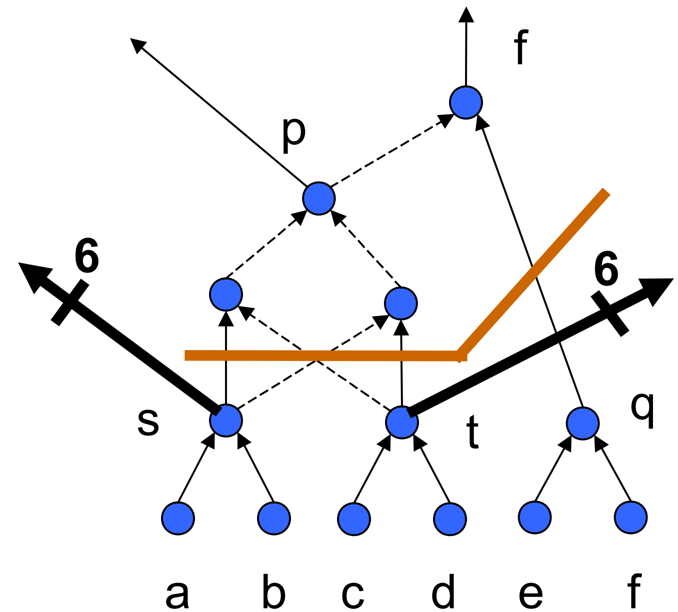
Area-flow "correctly" accounts for sharing

(Cong '99, Manohara-rajah '04)

Exact Local Area

Exact-local-area (cut) = 1 + [Σ exact-local-area (fanin with no other fanout)]





Cut *{pef}*

Area flow = 1+ [(.25+.25+3)/2] = 2.75

Exact area = 1 + 0 (p is used elsewhere)

Exact area will choose this cut.

Cut *{stq}*

Area flow = 1+ [.25+.25 +1] = 2.5

Exact area = 1 + 1 = 2 (due to *q*)

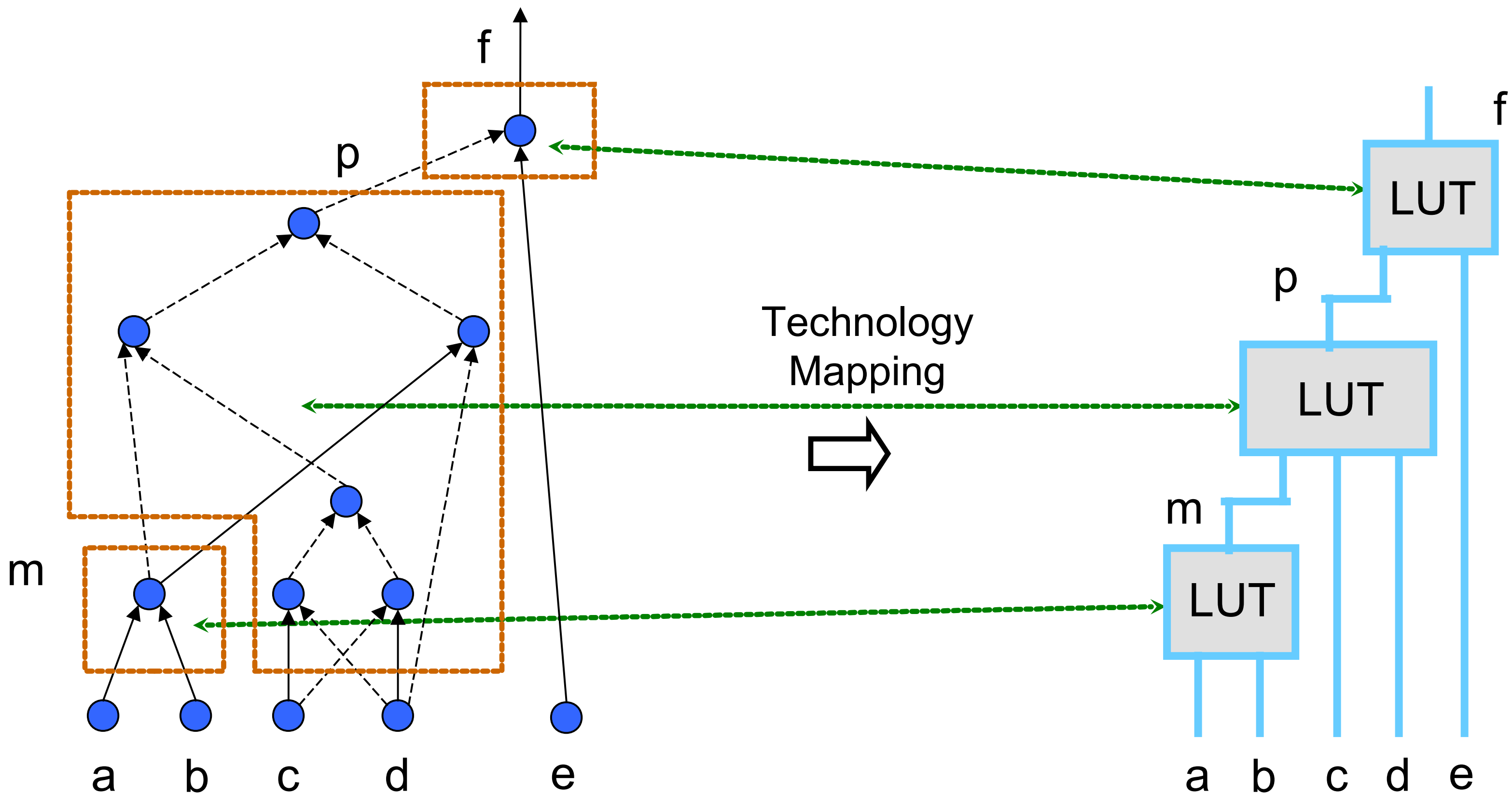
Area flow will choose this cut.

Area Recovery Summary

- Area recovery heuristics
  - Area-flow (global view)
    - Chooses cuts with better logic sharing
  - Exact local area (local view)
    - Minimizes the number of LUTs by looking one node at a time
- The results of area recovery depends on
  - The order of processing nodes
  - The order of applying two passes
  - The number of iterations
  - Implementation details
- This scheme works for the constant-delay model
  - Any change off the critical path does not affect critical path

Structural Bias

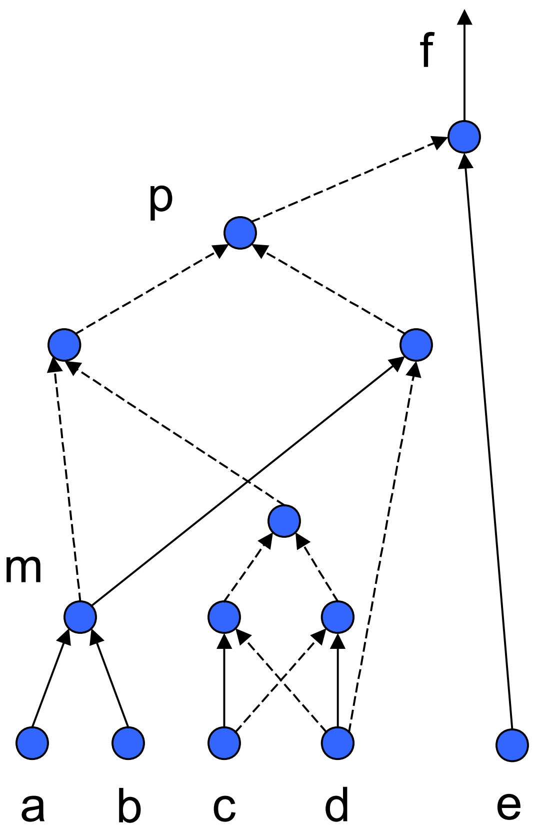
The mapped netlist very closely resembles the subject graph

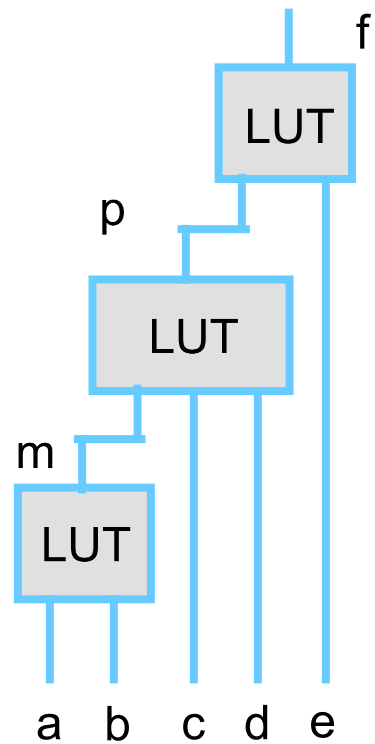


Every input of every LUT in the mapped netlist must be present in the subject graph - otherwise technology mapping will not find the match

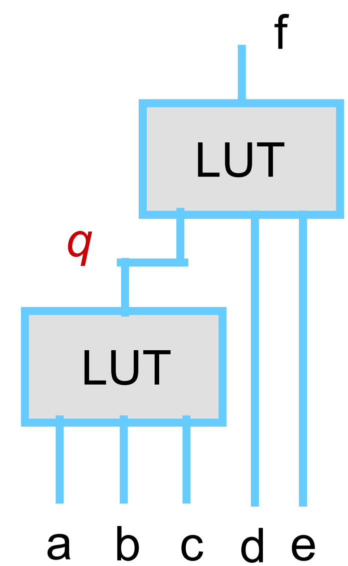
Example of Structural Bias

A better match may not be found





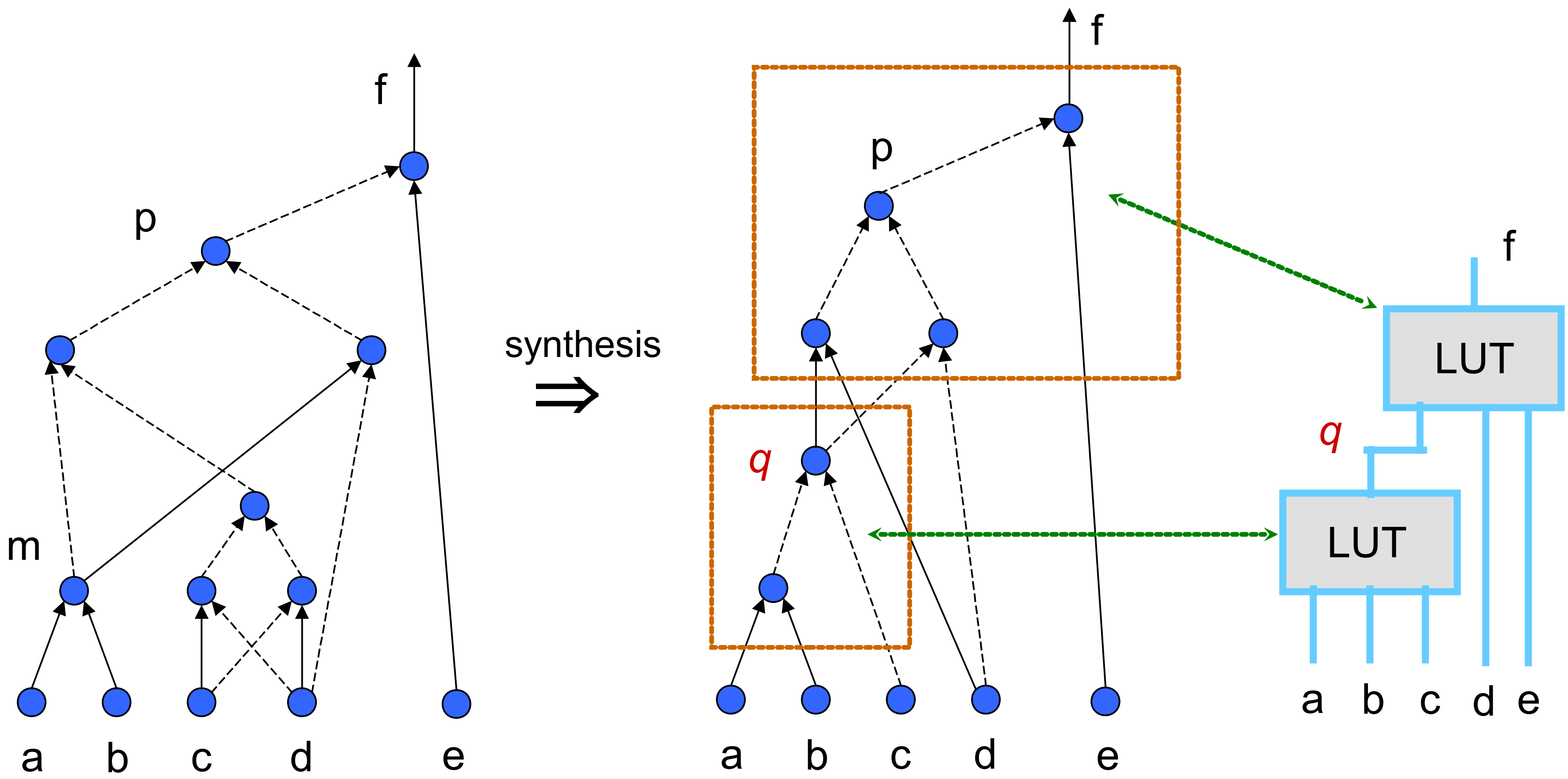
This match is not found



Since the point ***q*** is ***not*** present in the subject graph,
the match on the right is ***not*** found

Example of Structural Bias

The better match can be found with a different subject graph



Summary

Tech Mapping for Combinational Logic Circuits

- Derive balanced AIG
- Compute *k*-feasible cuts
- Compute Boolean functions of all cuts (truth tables)
  - needed only for standard cell designs
- Find matching for each cut
- Assign optimal matches at each node (from PIs to POs)
  - LUTs: delay optimal
  - Gates: area optimal
- Recover area on non-critical paths
- Choose the final mapping

To Learn More

- Visit ABC webpage http://www.eecs.berkeley.edu/~alanmi/abc
- Read recent papers http://www.eecs.berkeley.edu/~alanmi/publications
- Send email
  - alanmi@eecs.berkeley.edu
  - brayton@eecs.berkeley.edu