State-of-the-Art in Equivalence Checking

Wolfgang Kunz
University of Kaiserslautern, Germany

Overview

Verification gap:

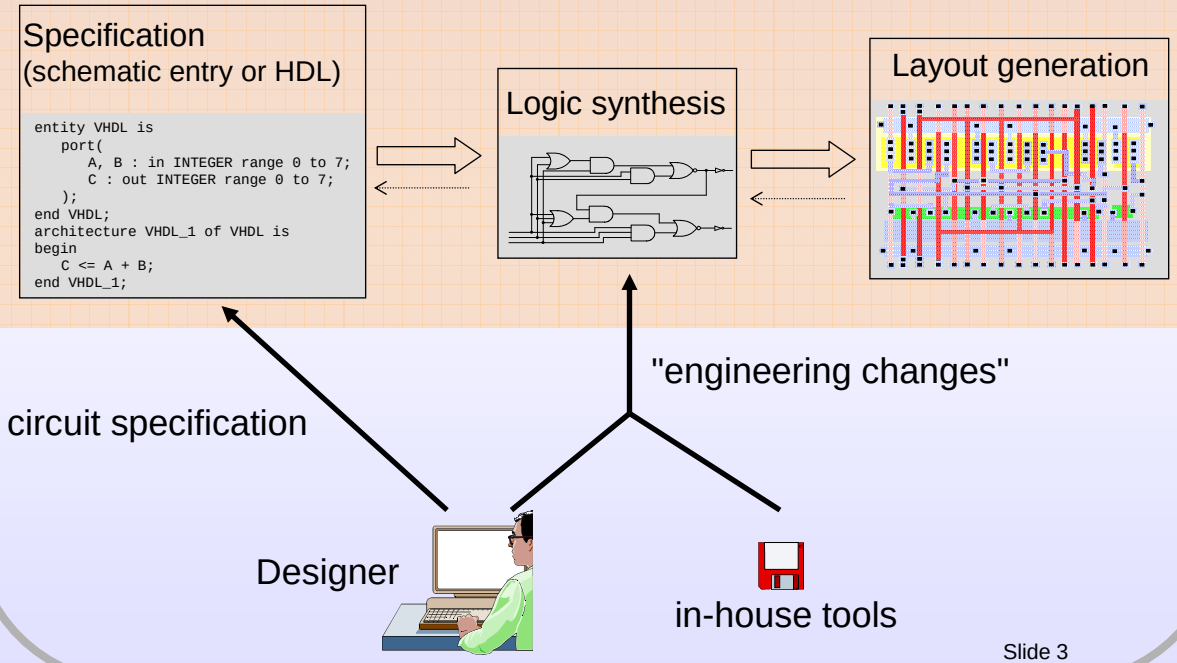
SIA Roadmap: 60% of design resources are consumed by verification!

Formal methods – the answer?

- what tools exist?
- what is the underlying technology?
- equivalence checking and property checking

Background

Typical design flow using a standard synthesis tool



Task of Verification

1. **Does the formal specification fullfill the expectations of the designer?**

   - can unexpected situations occur? (e.g.: deadlocks)
   - are there unexpected side effects?
   - do certain things always work?

   "*design verification*"

Verification method: "*property checking*" or "*model checking*"

Task of Verification

2. Has the specification been correctly translated into an implementation?

error sources:

- bugs in synthesis tool
- problems with interfacing several tools
- manual interaction of the designer

"*implementation verification*"

Verification method: "*equivalence checking*"

Industrial Practice

Implementation verification: *equivalence checking*

mostly based on SAT, ATPG and local BDDs

usage:

- check equivalence of specification and implementation
- verify implementation repeatedly after synthesis and engineering changes,
- can be integrated into regression testing environment,
- feasible for 1,000,000 gates and more for combinational equivalence checking

The engines

binary decision diagrams (BDDs)

- special graph representation of Boolean function
- many problems that are difficult to solve for conventional representations of Boolean functions are easy to solve for BDDs

SAT-solving methods (SAT-solvers)

- decide whether or not a Boolean function can be 1
- many verification problems can be mapped to a SAT-problem

Decision diagrams (DDs)

Important contribution to handle complexity:

graph representation of Boolean functions (OBDDs ("ordered binary decision diagrams"), [Brya86])

many types of decision diagrams:

OBDDs, FDDs, OKFDDs, IBDDs, ADDs, MTBDDs, EVBDDs, BMDs, *BMDs, K*BMDs, PHDDs u.a.

"alphabet soup DDs"

Decision diagrams (DDs)

Important properties of a Boolean function representation:

- Canonicity (uniqueness)
  e.g. truth table
- Compactness
  e.g. multi-level netlist (Boolean network)
- easy to perform Boolean operations (AND, OR, NOT)
  e.g. multi-level netlist
- easy to solve Boolean decision problems (SAT)
  e.g. truth table

DDs attempt to find the ideal compromise between these requirements.

Decision diagrams (DDs)

for most DDs:

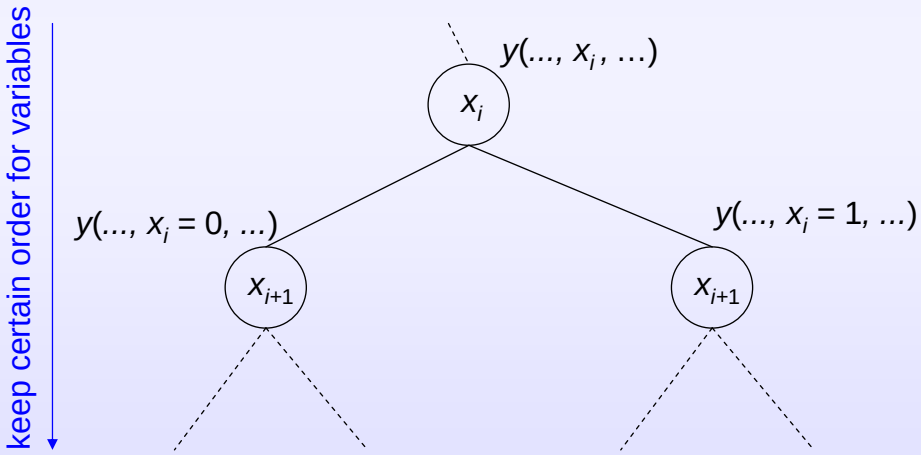
- directed acyclic graph with one or several root nodes
- every sub-graph rooted in some node represents a Boolean function
- every node corresponds to a decomposition of its associated function
- the terminal nodes of the graph correspond to the terminal cases of the decomposition (e.g. the output values of the function)

Ordered binary decision diagrams (OBDDs)

underlying decomposition: Shannon decomposition

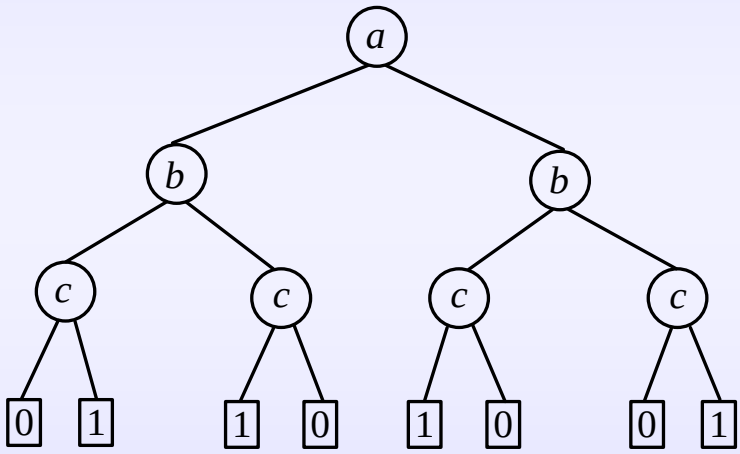
$y(x_1, \ldots, x_n) = x_i \cdot y(x_1, \ldots, x_i = 1, \ldots, x_n) + \overline{x_i} \cdot y(x_1, \ldots, x_i = 0, \ldots, x_n)$

short notation: $y = x \cdot y|_{x=1} + \overline{x} \cdot y|_{x=0}$



Example

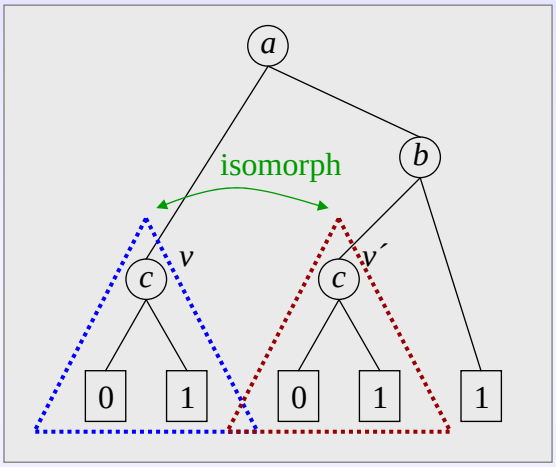
$$y = abc + a\overline{b}\overline{c} + \overline{a}b\overline{c} + \overline{a}\overline{b}c$$



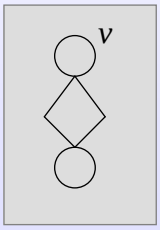
binary decision diagram (Shannon-tree) for 3-bit parity function

Reduction

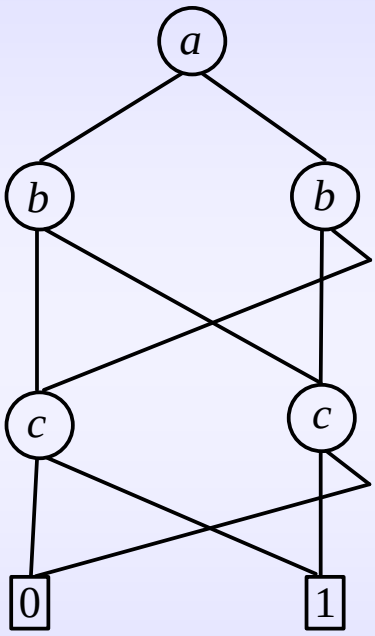
- nodes with isomorphic sub-graphs can be merged



- nodes with only one immediate successor can be eliminated



Example



Reduced ordered binary decision diagram (ROBDD) for 3-bit parity function

Canonicity

A reduced ordered binary decision diagram (ROBDD) is a canonical representation of a Boolean function.

Theorem (Bryant, 86):

Two Boolean functions are functionally equivalent if and only if their ROBDDs are isomorphic.

Boolean manipulation

If-Then-Else-Operator: $ite(f, g, h) = f \cdot g + \bar{f} \cdot h$
„if f then g, else h"

Every Boolean function of two variables, i.e. every binary operation can be expressed by the ite-Operator:

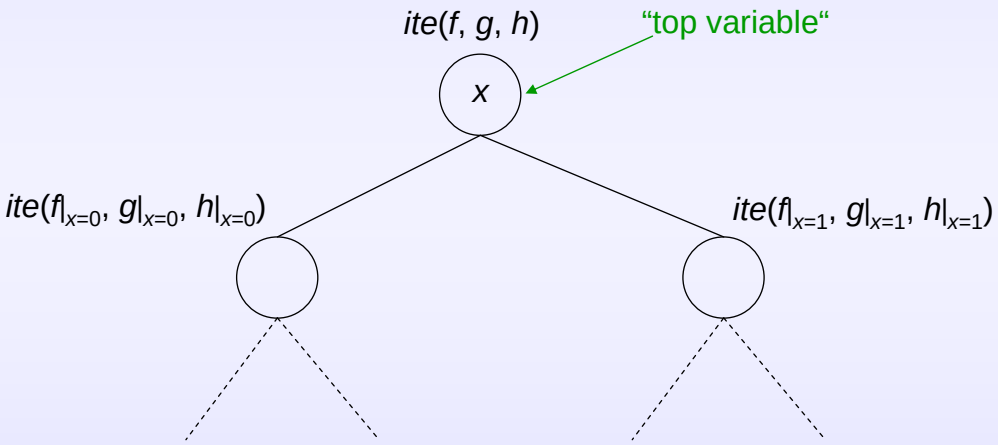
e.g.: $f \cdot g = ite(f, g, 0)$, $f + g = ite(f, 1, g)$,

Let x be the “top-variable” of the ROBDD that shall be formed out of f, g, and h. It is:

$$ite(f, g, h) = ite(x, ite(f|_{x=1}, g|_{x=1}, h|_{x=1}), ite(f|_{x=0}, g|_{x=0}, h|_{x=0}))$$

Boolean manipulation

$ite(f, g, h) = ite(x, ite(f|_{x=1}, g|_{x=1}, h|_{x=1}), ite(f|_{x=0}, g|_{x=0}, h|_{x=0}))$



Terminal cases of recursion:

a) $ite(1, f, g) = ite(0, g, f) = ite(f, 1, 0) = ite(g, f, f) = f$

b) $ite(f, 0, 1) = \overline{f}$

Important operations

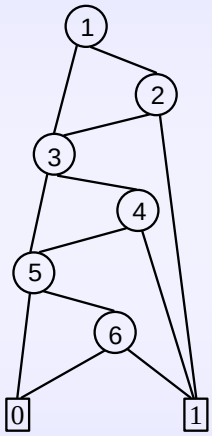
Let f be a Boolean function represented as OBDD with $|f|$ nodes:

	operation	complexity
reduce:	makes OBDD canonical	$O(\|f\|)$
apply:	$y = y(f_1, f_2)$ (beruht auf *ite*-Operator)	$O(\|f_1\| \cdot \|f_2\|)$
restrict:	$f = (x_1, \ldots x_i = V, \ldots x_n)$, $V \in \{0, 1\}$	$O(\|f\|)$
compose:	$f_1 = (x_1, \ldots x_i = f_2, \ldots x_n)$	$O(\|f_1\|^2 \cdot \|f_2\|)$

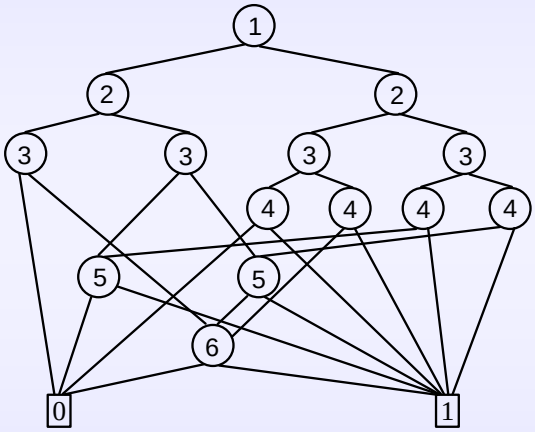
Operations on OBDDs

Variable order

Variable order is important for many functions!



a) $y = x_1x_2 + x_3x_4 + x_5x_6$



b) $y = x_1x_4 + x_2x_5 + x_3x_6$

For some functions ROBDD grows exponentially with number of variables for every variable order, e.g. multiplier.

OFDDs (ordered functional decision diagrams)

underlying decomposition: (positive) Reed-Muller-decomposition (Davio-decomposition)

$$y(x_1,\dots x_n) = y(x_1,\dots x_i = 0,\dots x_n) \oplus x_i \ (y(x_1,\dots x_i = 1,\dots x_n) \oplus y(x_1,\dots x_i = 0,\dots x_n))$$

short notation: $y = y|_{x=0} \oplus x(y|_{x=0} \oplus y|_{x=1})$

- sometimes more compact
- Boolean manipulations have exponential worst-case complexity
- multipliers still explode

Equivalence Checking

given: two sequential circuit descriptions
(RTL-level or gate-level)

prove equivalence of the two designs

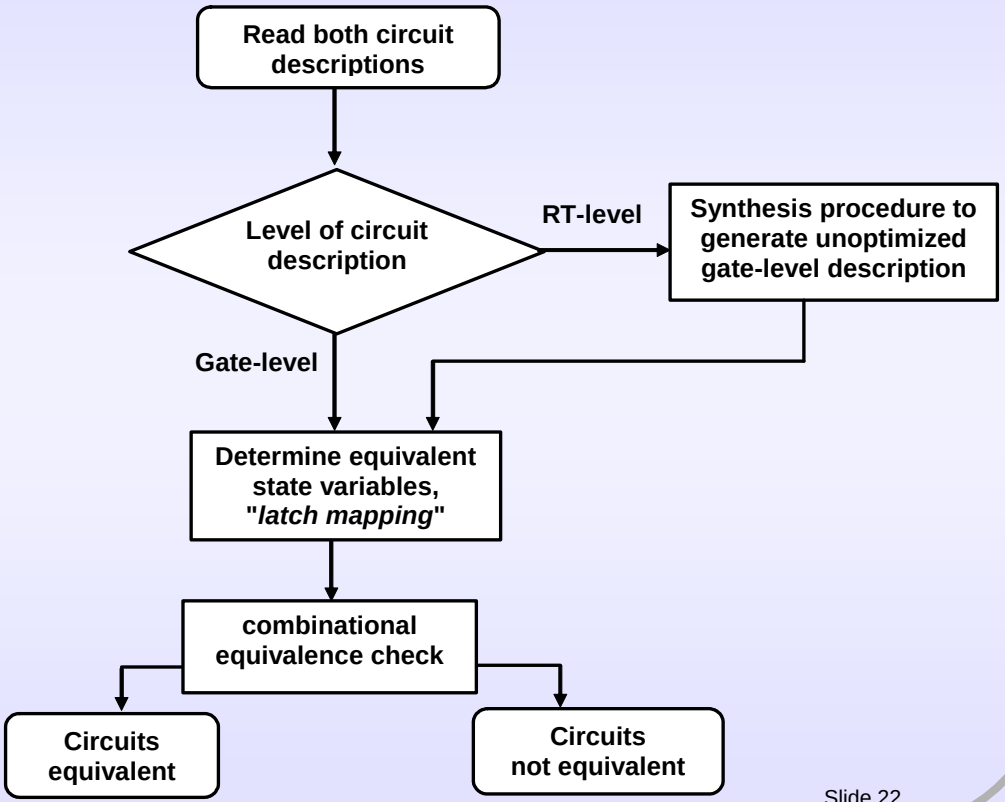
Combinational equivalence checking:

circuits have the same state encoding

⇒ check whether or not combinational blocks implement the same Boolean function

Only *combinational* equivalence checking is sufficiently mature for industrial use!

General Procedure



"Latch Mapping" or "State Matching"

Identify equivalent state variables

- manually by user interaction
- automatically by name
- automatically by function

State matching

Basic procedure

1. Put all state variables into one equivalence class
2. Prove non-equivalence of some state variables
3. Split equivalence class(es) accordingly

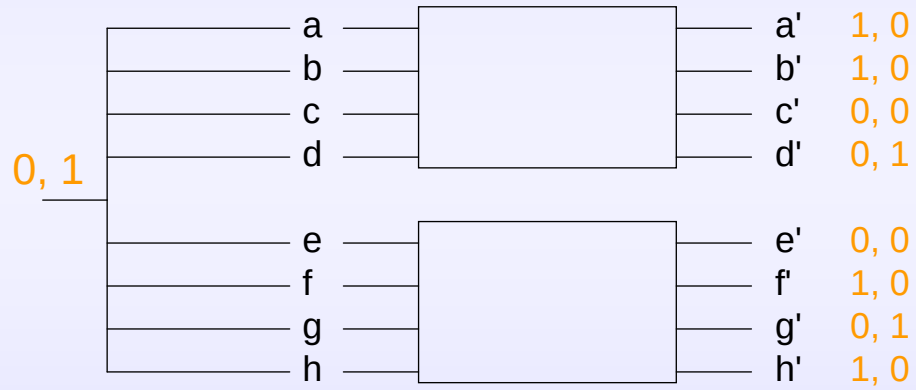
Iterate Step 2 and 3 until no further refinement is obtained.

How to prove non-equivalent state variables?

State matching

Example

put all state variables into one class and simulate



present state variables

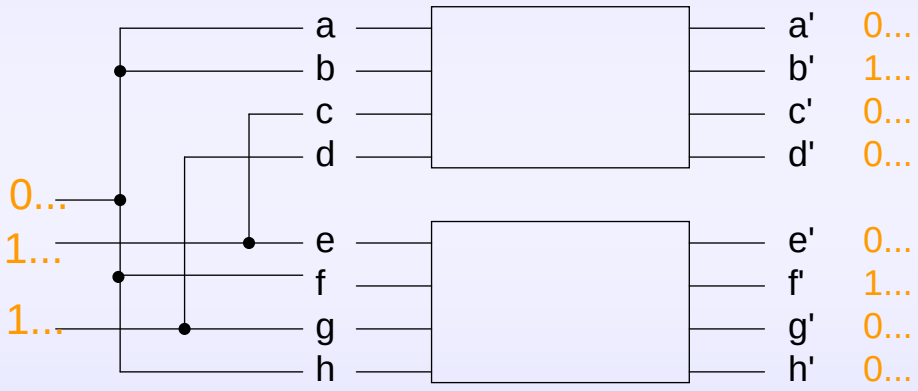
next state variables

split classes:
{a, b, f, h}
{c, e}
{d, g}

State matching

Example

refine equivalence class and simulate again



8 patterns possible

present state variables

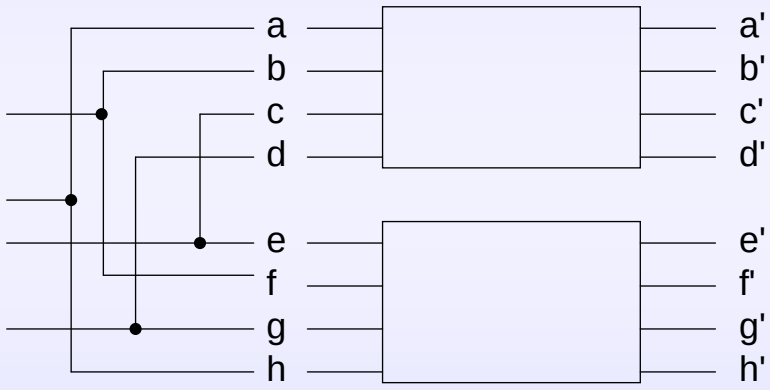
next state variables

split classes:
{a, h}
{b, f}
{c, e}
{d, g}

State matching

Example

latch mapping complete



present state variables

next state variables

State matching

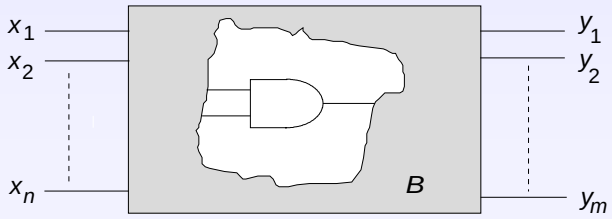
Basic procedure

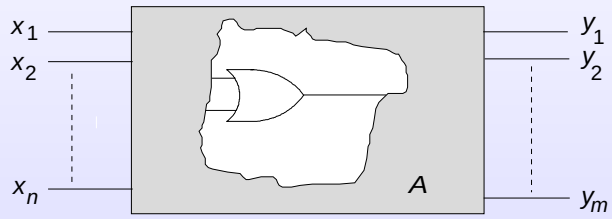
Non-equivalence of state variables can be proved by

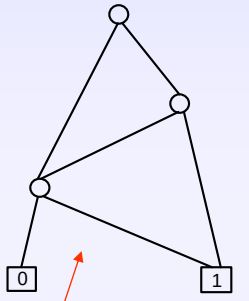
- analyzing circuit structure
- logic simulation
- SAT (satisfiability solving) or ATPG (automatic test pattern generation)
- BDDs

Equivalence Checking

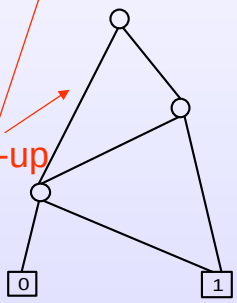
The "functional paradigm": build canonical representation (e.g. ROBDD, Bryant 86)





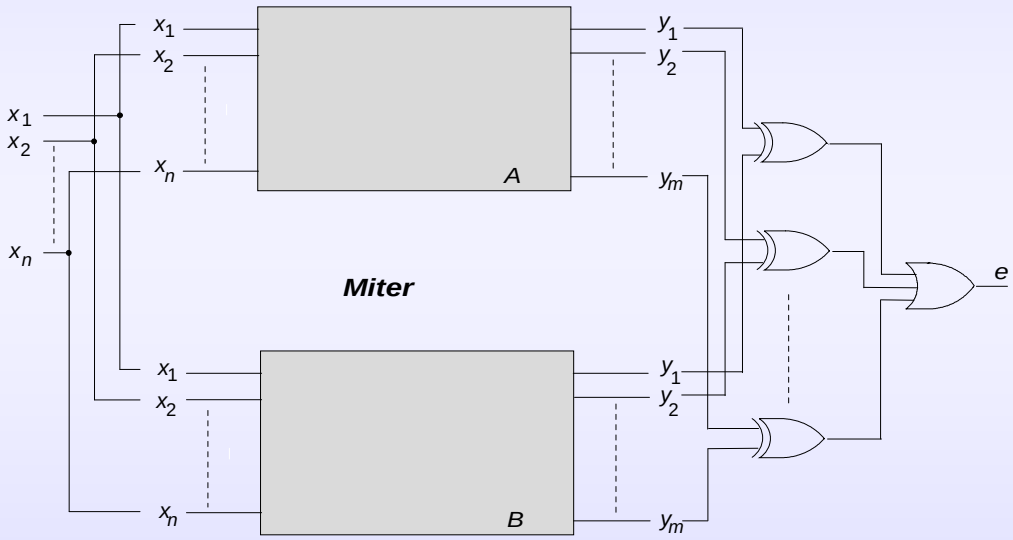


Problem:
BDD-blow-up



Equivalence Checking

The SAT-based approach



A and *B* are not equivalent if and only if *e* is satisfiable.

SAT-solving / ATPG

What problems are hard and what problems are easy?

Stuck-at test generation is easy.

Combinational equivalence checking is hard.

Why?

Source of many problems:

reconvergent fanout causing redundant logic

SAT-solving / ATPG

Combinational stuck-at test generation

- operates on circuit under test (CUT)
- CUT technically makes sense, i.e., redundant structures are fairly local
- problems can be localized by SAT- / ATPG-algorithms

SAT and ATPG work quite robustly!

SAT-solving / ATPG

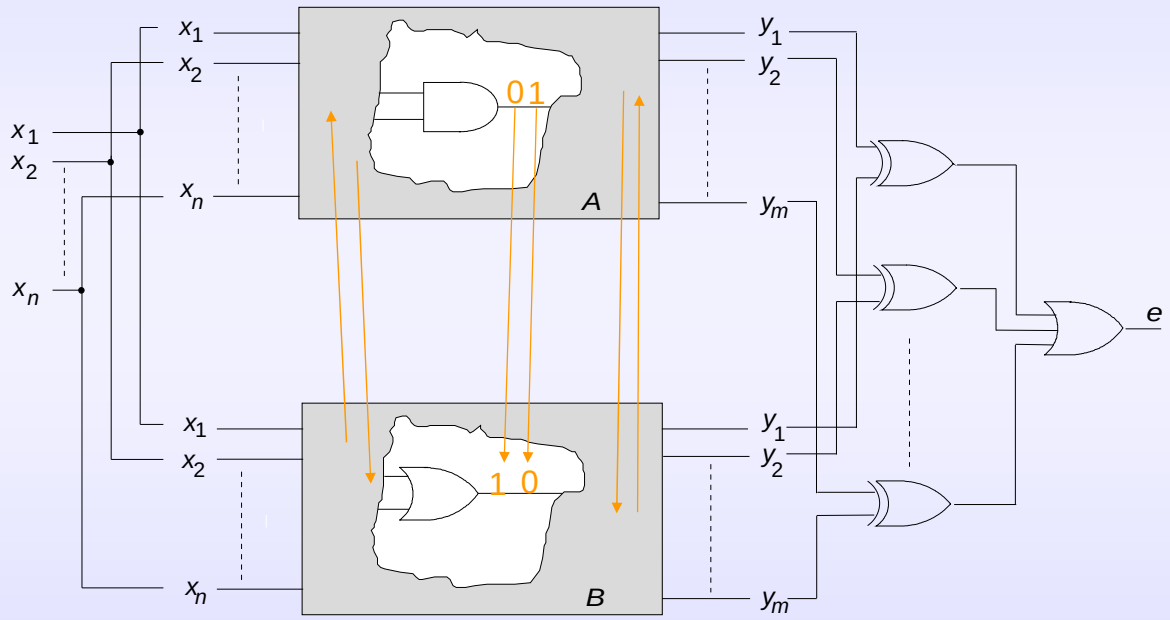
Combinational equivalence checking

- operates on miter
- miter technically does not make sense
- huge reconvergent fanout structure,
  if circuits are equivalent miter is a redundant implementation of a constant 0
- problem cannot be localized easily

SAT and ATPG often fail!

Equivalence Checking

The "structural paradigm": exploit structural similarity of designs



Circuits contain internal equivalences

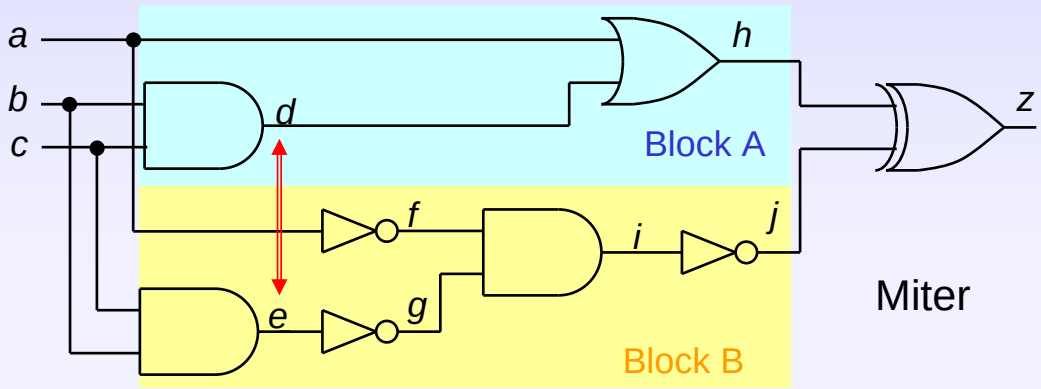
Exploiting internal equivalences

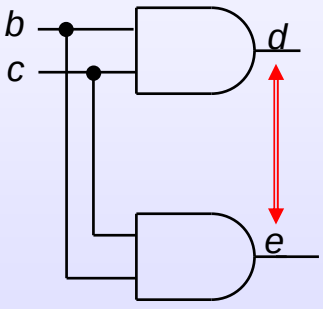
Starting from inputs and moving towards outputs:

- identify equivalences at internal signals in miter
- substitute nodes corresponding to equivalences

Prove/disprove satisfiability of output of modified miter

Example

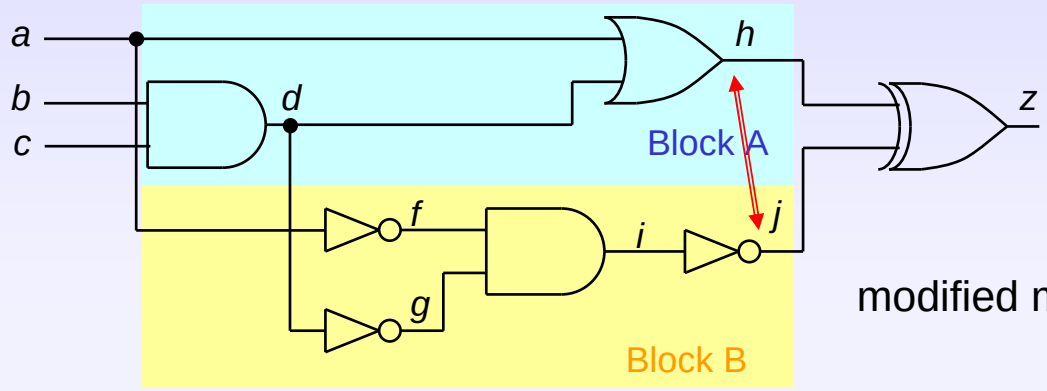




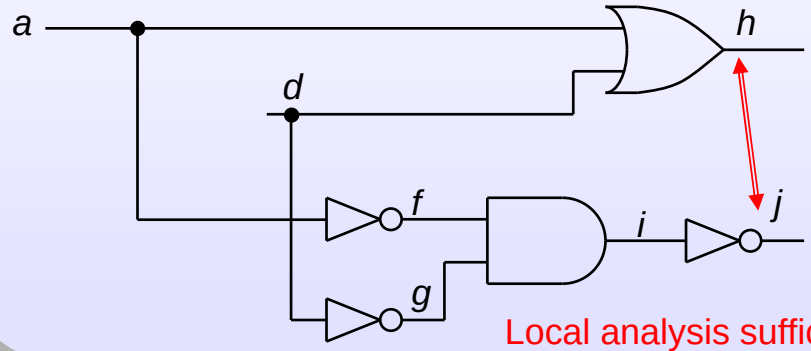
circuit partition relevant to determine internal equivalence $d = e$

Local analysis sufficient !

Example



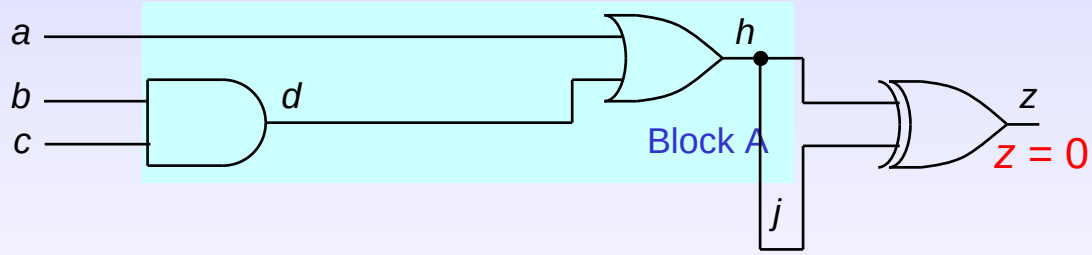
modified miter



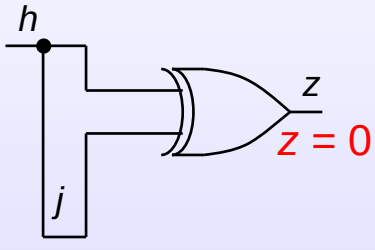
circuit partition relevant to determine internal equivalence $h = j$

Local analysis sufficient !

Example



modified miter



circuit partition relevant to determine that $z = 0$

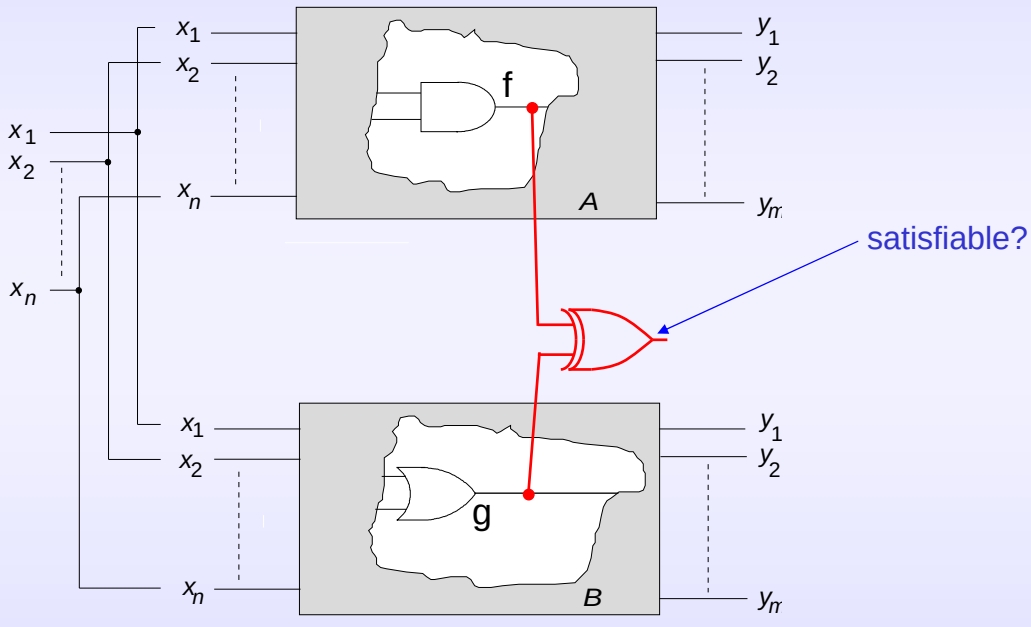
Local analysis sufficient !

Identifying internal equivalences

Methods to identify internal equivalences:

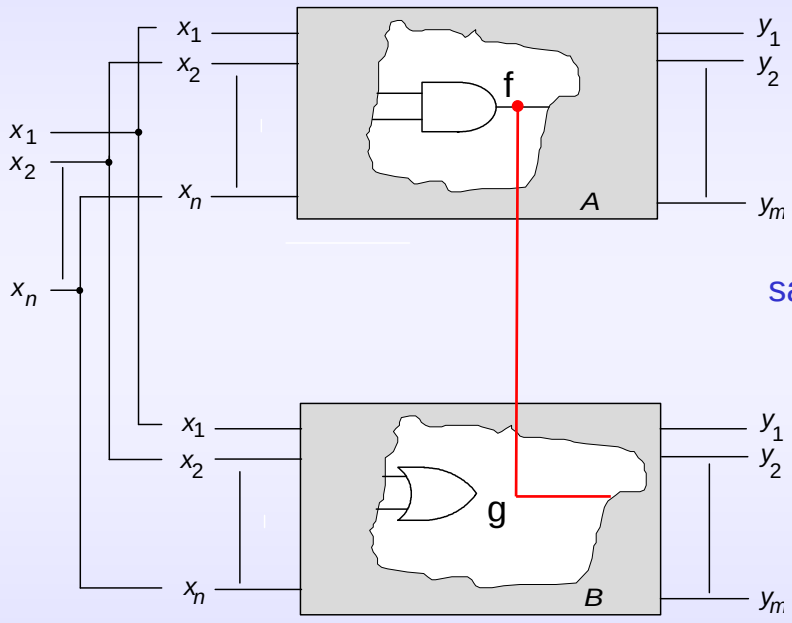
- apply random patterns to determine candidates for equivalent signal pairs,
  then prove equivalence by SAT or ATPG [Bran93]
- variable probing, implications [Ku93]
- local BDDs for relevant circuit partitions [JaMu95], [Mats96], [KuKr97]

Identifying internal equivalences by SAT



Signals *f* and *g* are equivalent if and only if output of XOR is not satisfiable.

Identifying internal equivalences by SAT



satisfiable?

No: *f* can replace *g*

Identifying internal equivalences by BDDs

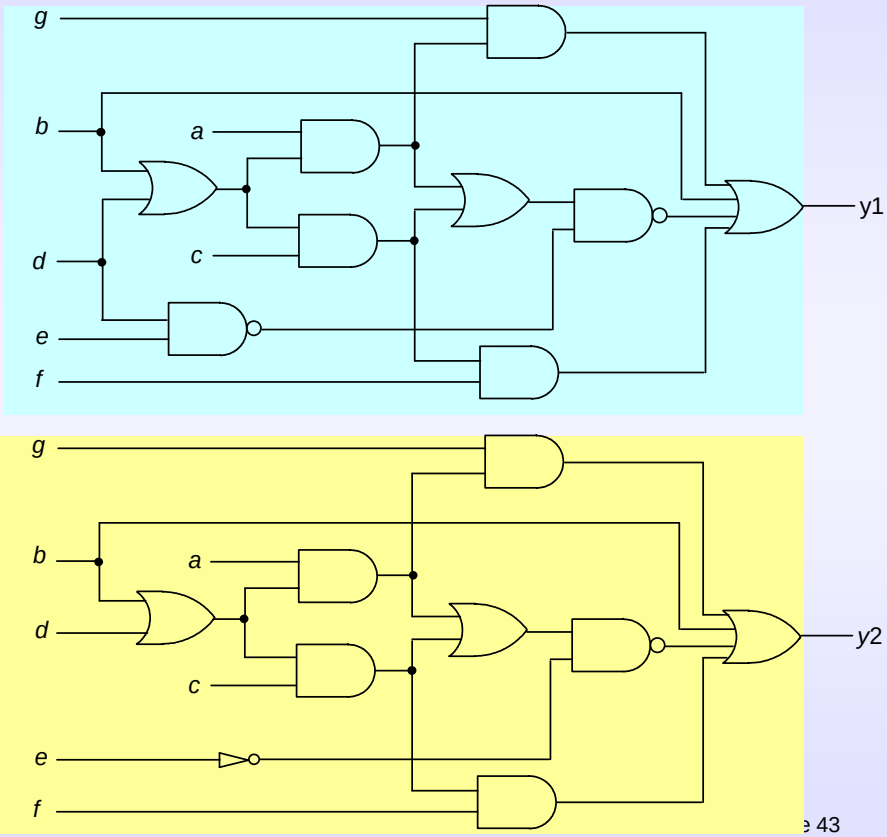
Idea: use BDDs restricted to relevant circuit partitions

- determine internal equivalences starting from primary inputs
- use known equivalences as internal "cut points", i.e., as input variables for circuit partitions to be checked for equivalence

Problem: false negatives

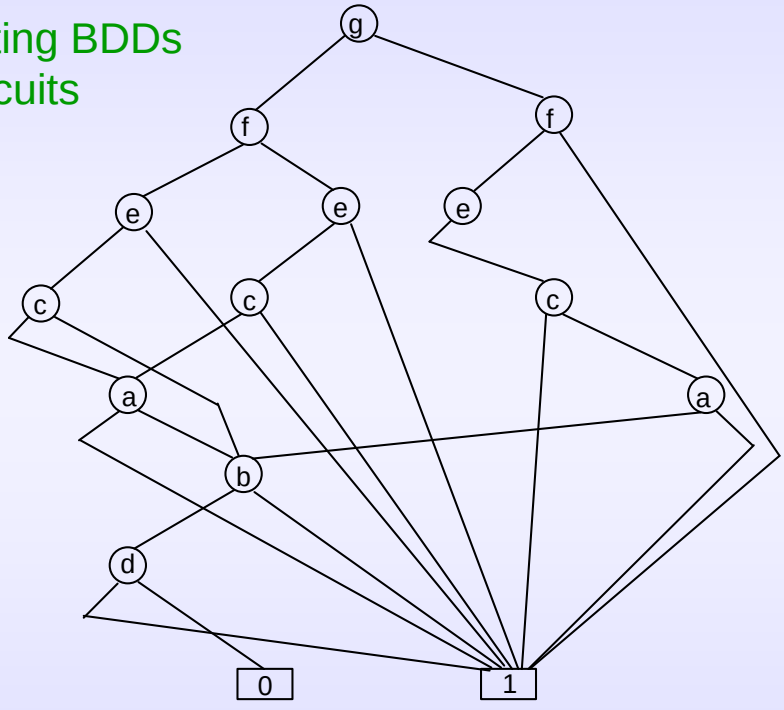
Example

Circuits to be compared



Example

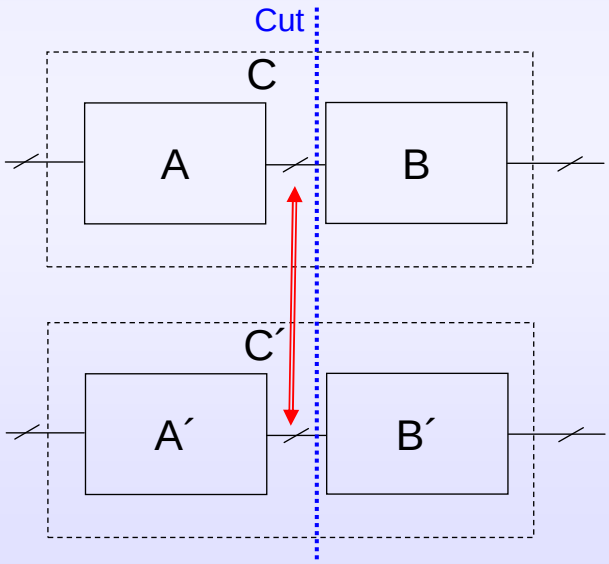
Constructing BDDs for full circuits



ROBDD for y_1 and y_2 (circuits are equivalent)

Partitioning using internal equivalences

Prove equivalence of C and C´ by first proving the equivalence of A and A´ and then the equivalence of B and B

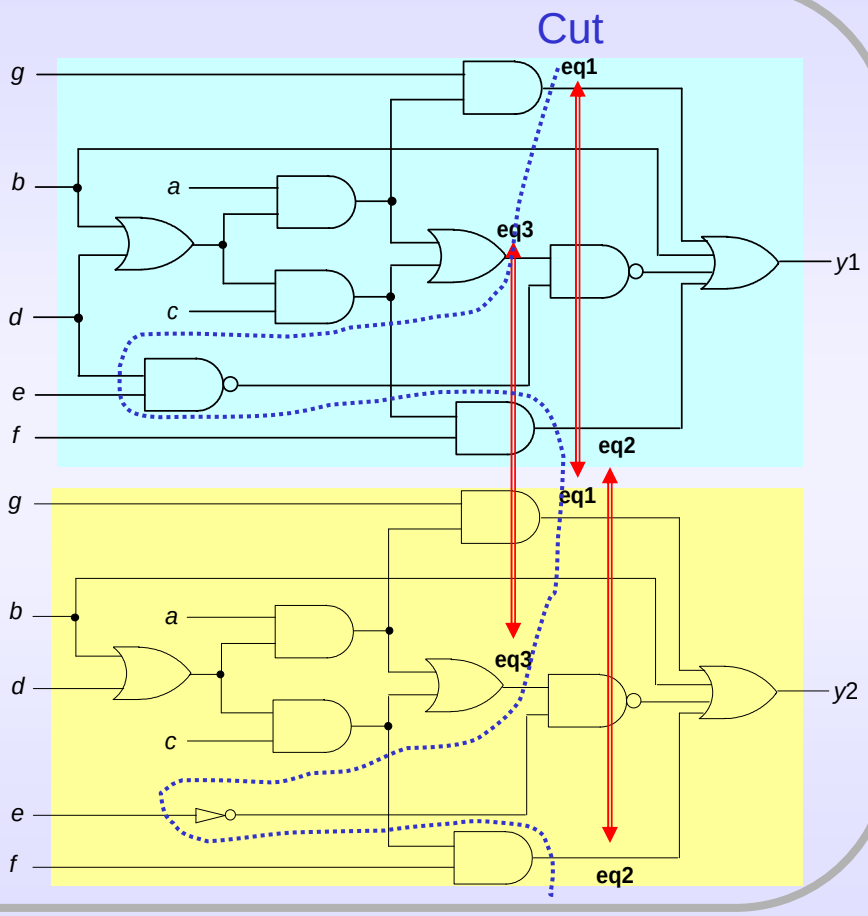


Example

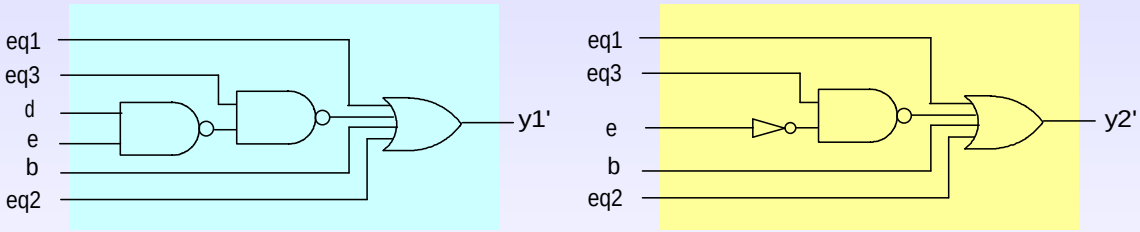
Prove equivalence of eq_1, eq_2, eq_3 by building BDDs in terms of primary inputs

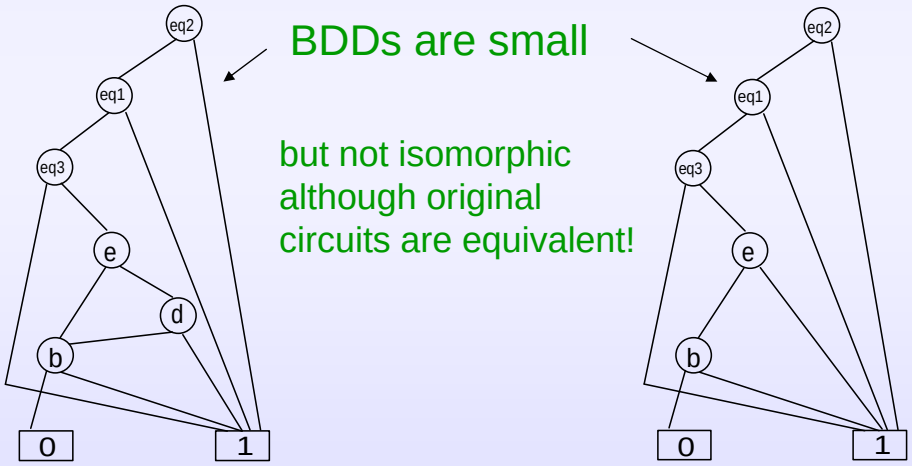
Choose cut through circuit, cut line must only contain primary inputs or internal equivalences

Build BDDs for y_1 and y_2 based on cut line



Example



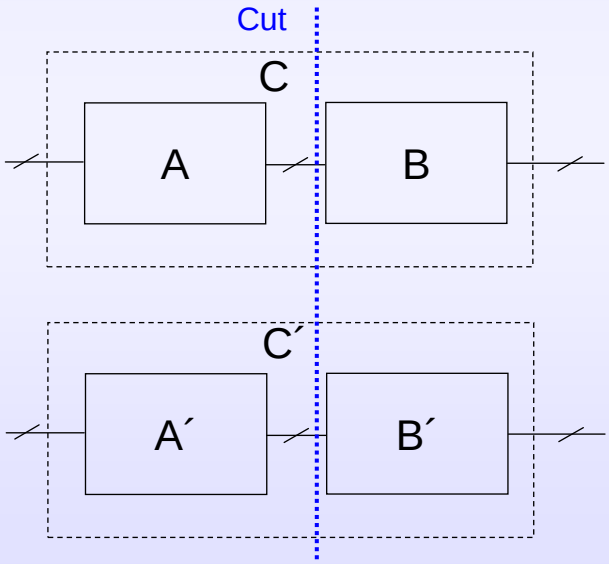


False negative!

Partitioning using internal equivalences

Note: C = C´ and A = A´ does not B = B´ !

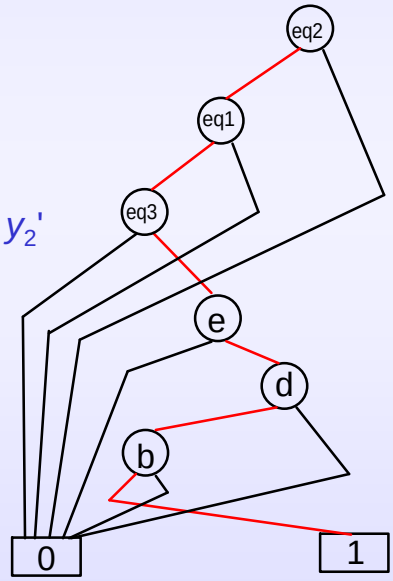
Reason: Some combinations of value assignments may not be possible at the cutline (don‘t cares). For those combinations B and B´ can be different without violating C = C´.



False negatives

Reason: cut points are treated as independent input signals

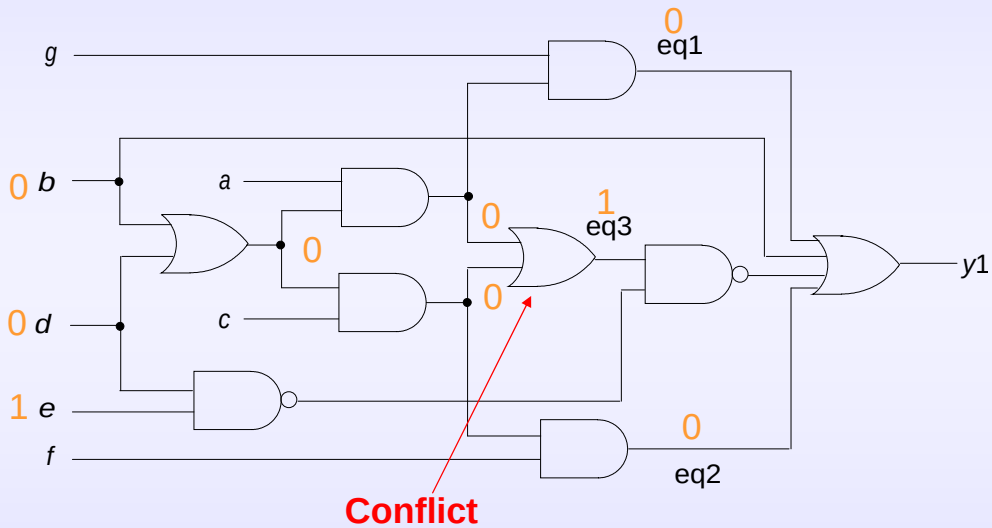
OBDD for $y_1' \oplus y_2'$



Every path to terminal node 1 corresponds to a (internal) set of value assignments for which the two circuits have different output.

In case of a *false negative* these value assignments are not satisfiable (cannot be justified from the inputs).

Example



Conflict in original circuit ⇒ false negative

Handling false negatives

Techniques:

- explicit satisfiabilty check of all distinguishing vectors in original circuit [KuPr96]

  too complex if many such vectors exist!

- minimize OBDDS by exploiting don't care conditions at cut line [KuPr96]

  not exact, also very complex!

Handling false negatives

Techniques (contd.):

- use compose-Operation and move cut line back-wards until false negative has disappeared [Mats96]

  BDDs may blow up, the right cut line may not be found!

- avoid false negatives right from the beginning by building BDDs for overlapping layers of circuit partitions, overlap creates robustness in case of false negatives [KuKr97]

  many BDDs need to be built, right cut might still be missed!

SAT versus BDD

Identifying internal equivalences

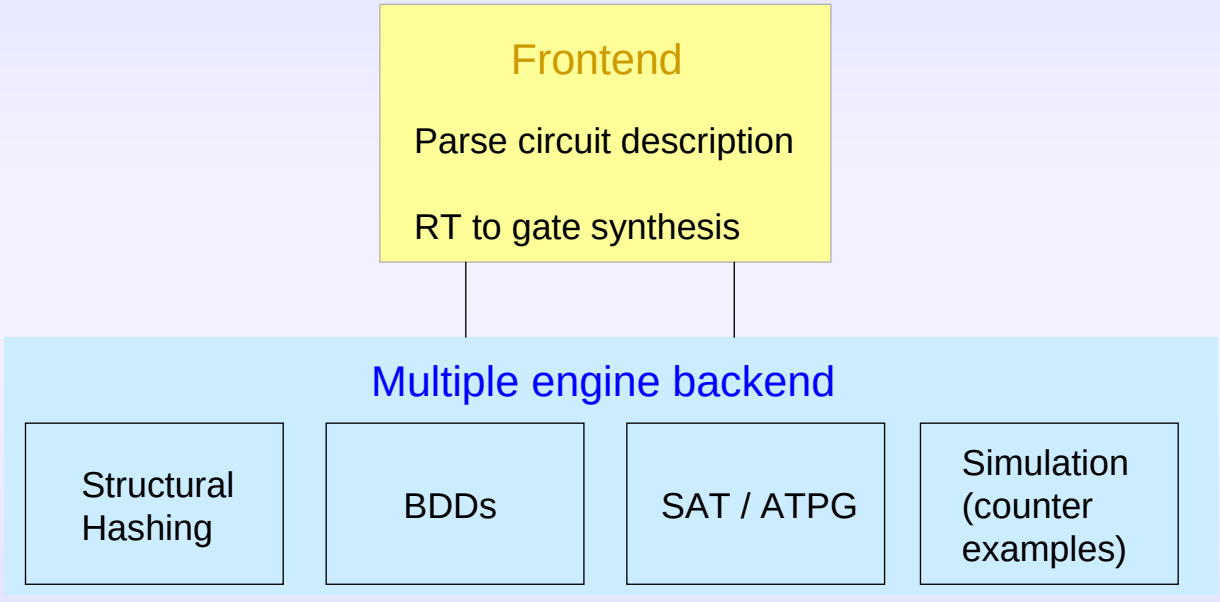
SAT-based techniques avoid the false negative problem, circuit does not need to be cut !

BDD-approaches suffer severely from the false negative problem !

But: BDDs are much faster than SAT!

Equivalence checker

Modules



Equivalence Checking

Typical procedure - Example

increasing complexity

represent circuit as combinational netlist of 2-input NAND gates and inverters "*signed AND-graph* (SAG)" [KuKr97]

traverse SAG from inputs to outputs, determine isomorphic sub-graphs and reduce SAG accordingly ("*structure hashing*")

starting at the inputs, determine BDDs for the nodes of the SAG

if two nodes have isomorphic BDDs they are *cut points*, reduce SAG accordingly ("*BDD hashing*"), then use structure hashing for further reduction [KuKr97]

the cut points identified serve as additional variables for the BDDs

Equivalence Checking

Typical procedure - Example (contd.)

increasing complexity

simulate random vectors, counter example?

determine candidates for equivalent node pairs from results of random simulation

check pairs by SAT / ATPG, reduce SAG if equivalence is proved [Bran93], [Ku93]

run SAT / ATPG on the outputs of the miter [KuGa01]

try to build BDD for complete circuit employing sophisticated variable ordering

Interleaving between different steps possible !

Open Problems

Fairly mature technology but some problems remain:

- Robust latch mapping for faulty circuits
- Equivalence checking for multipliers
- Sequential Equivalence Checking

Literature

Combinational Equivalence Checking (1)

[AaSt94] Aas E. J., Steen T., and Klingsheim K: "Quantifying Design Quality Through Design Experiments", *IEEE Design & Test of Computers*, vol. 11, pp. 27-38, Spring 1994.

[BeTr89] Berman C. L., and Trevillyan L. H.: "Functional Comparison of Logic Designs for VLSI Circuits", *Proc. Intl. Conf. on Comp.-Aided Design* (ICCAD), pp. 456-459, 1989.

[Bran93] Brand D.: "Verification of Large Synthesized Designs", *Proc. IEEE International Conference on Computer-Aided Design* (ICCAD), Santa Clara, pp. 534-537, Nov. 1993.

[JaMu95] Jain J., Mukherjee R., and Fujita M.: "Advanced Verification Techniques Based on Learning", *Design Automation Conference* (DAC), pp. 420 - 426, June 1995.

Literature
Combinational Equivalence Checking (2)

[KuKr97] Kühlmann A., and Krohm F.: "Equivalence Checking using Cuts and Heaps", *Proc. Design Automation Conference*, San Fransisco, Juni 1997.

[KuGa01] Kühlmann A., Ganai M., Paruthi V.: "Circuit-based Boolean Reasoning", *Design Automation Conference* (DAC), pp. 232 - 237, June 2001.

[Ku93] Kunz W.: "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning", *Proc. Intl. Conference on Computer-Aided Design* (ICCAD), Santa Clara, pp. 538-543, Nov. 1993.

[KuPr96] Kunz W., Pradhan D., and Reddy S.: "A Novel Framework for Logic Verification in a Synthesis Environment", *IEEE Transactions on Computer-Aided Design*, vol. 15, no. 1, pp. 20 - 33, January 1996.

Literature
Combinational Equivalence Checking (3)

[Mats96] Matsunaga Y.: "An efficient equivalence checker for combinational circuits", *Proc. Design Automation Conference*, pp. 629-634, Las Vegas, 1996.

[MSGl99] Marquez-Silva J., Glass T.: "Combinational Equivalence Checking Using Satisfiability and Recursive Learning", *Proc. Design, Automation and Test in Europe Conf.* (DATE), 1999.