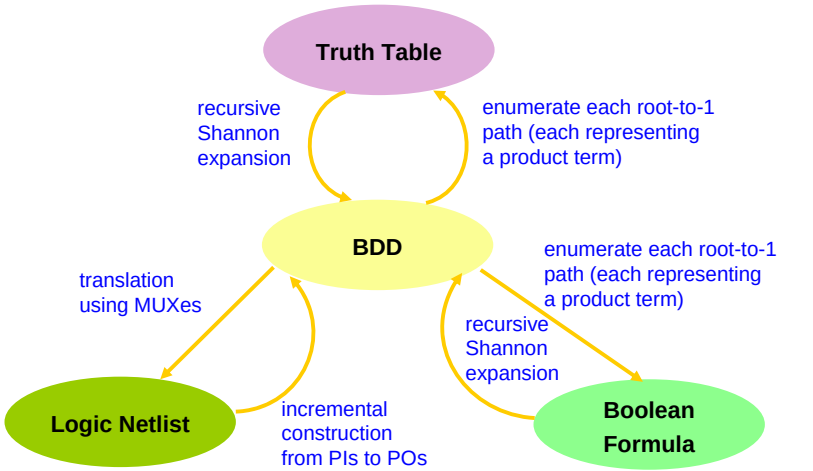
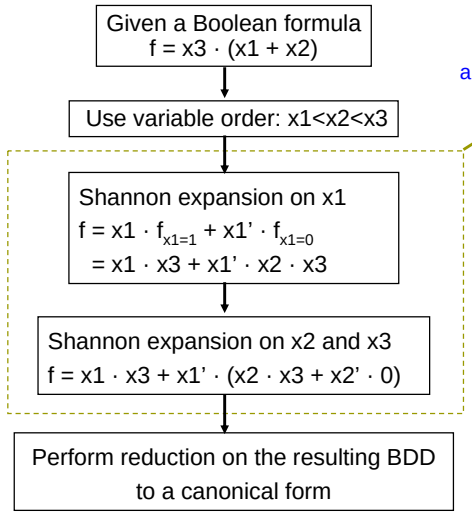
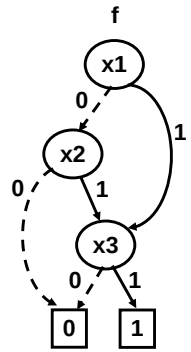
Data Type Conversion

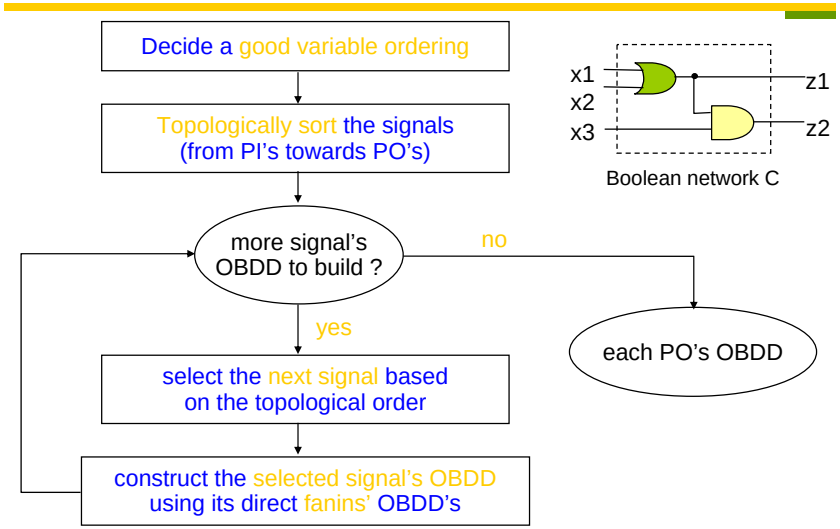


Formula to BDD





Netlist to BDD

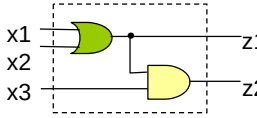


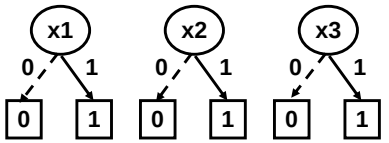
Netlist to BDD

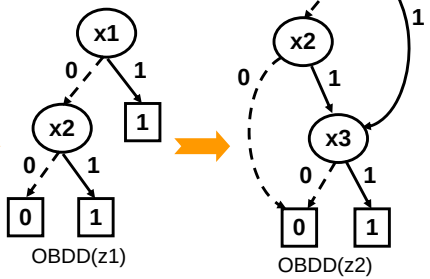
- Example

Topological order: {x1,x2,x3,z1,z2}
variable order: x1<x2<x3

OBDD(z2) = OBDD(x3) · OBDD(z1)

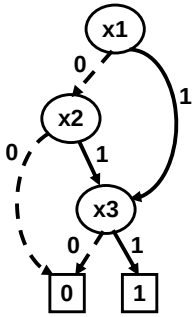


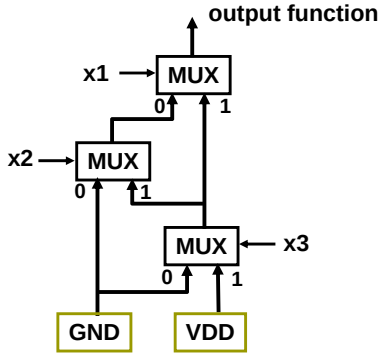




BDD to Netlist

- MUX-based translation
  - replace each decision node by a MUX
  - replace 0-terminal by GND, and 1-terminal by VDD
  - reverse the direction of every edge
  - specify the root node as the output node





BDD Features

- Strengths
  - ROBDD is a compact representation for many Boolean functions
  - ROBDD is canonical, given a fixed variable ordering
  - Many Boolean operations are of polynomial time complexity in the input BDD sizes
- Weaknesses
  - In the worst case, the size of a BDD is $O(2^n)$ for n-input Boolean functions

BDD Applications

- Boolean function verification
  - Compare a specification f to an implementation g, assuming their ROBDDs are F and G, respectively.
    - For fully specified functions f and g, the verification is trivial (pointer comparison) because of the strong canonicity of the ROBDD
      - Strong canonicity: the representations of identical functions are the same
    - For an incompletely specified function $I = (f, d, \neg(f+d))$ with onset f, dc-set d, and offset $\neg(f+d)$. A completely specified function g correctly implements I if $(d + f \cdot g + \neg f \cdot \neg g)$ is a tautology, that is, $f \Rightarrow g \Rightarrow (f+d)$
- Satisfiability checking
  - A Boolean function f is satisfiable if there exists an input assignment for which f evaluates to '1'
  - Any Boolean function whose ROBDD is not equal to '0' is satisfiable

BDD Applications

- Min-cost satisfiability
  - Suppose that choosing a Boolean variable x_i to be '1' costs c_i. Then, the minimum-cost satisfiability problem asks to minimize: $\sum_i c_i \cdot u_i(x_i)$
    where $\mu(x_i) = 1$ when $x_i =$ '1' and $\mu(x_i) = 0$ when $x_i =$ '0'.
  - Solving minimum-cost satisfiability amounts to computing the shortest path in an ROBDD with weights: $w(v, \eta(v)) = c_i$, $w(v, \lambda(v)) = 0$, *variable* $x_i = \phi(v)$, which can be solved in linear time
- Combinatorial optimization
  - Many combinatorial optimization problems can also be formulated in terms of the satisfiability problem
  - 0-1 integer linear programming can be formulated as a minimum-cost satisfiability problem although the translation may not be efficient
    - E.g., the constraint: $x_1 + x_2 + x_3 + x_4 = 3$ can be written as $(x_1+x_2)(x_1+x_3)(x_1+x_4)(x_2+x_3)(x_2+x_4)(x_3+x_4)(\neg x_1+\neg x_2+\neg x_3+\neg x_4)$

Outline

- Introduction
- Boolean reasoning engines
  - BDD
  - SAT
- Equivalence checking
- Property checking

SAT Solving

- SAT problem: Given a Boolean formula φ in CNF, find an input assignment such that φ valuates to true
- SAT solving is a decision procedure over CNFs

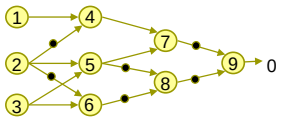
  Example

  $\varphi = (a+b'+c)(a'+b+c)(a+b'+c')(a+b+c)$

  is SAT (e.g. under a=1, b=1, c=0)
- SAT in CNF (POS) $\Leftrightarrow$ Tautology in DNF (SOP)
  - How about Tautology in CNF and SAT in DNF?

SAT Solving

- Given a circuit, suppose we would like to know if some signal is always zero. This can be formulated as a SAT problem if we can covert the circuit to an CNF.

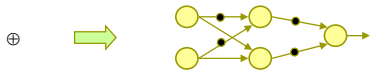


an AIG

Is output always 0 ?

Circuit to CNF

- Naive conversion of circuit to CNF:
  - Multiply out expressions of circuit until two level structure
  - Example: $y = x_1 \oplus x_2 \oplus x_2 \oplus \ldots \oplus x_n$ (Parity function)
    - circuit size is linear in the number of variables



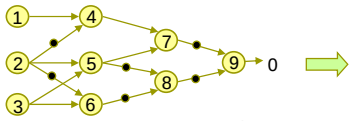
    - generated chess-board Karnaugh map
    - CNF (or DNF) formula has 2^{n-1} terms (exponential in #vars)
- Better approach:
  - Introduce one variable per circuit vertex
  - Formulate the circuit as a conjunction of constraints imposed on the vertex values by the gates
  - Uses more variables but size of formula is linear in the size of the circuit

Circuit to CNF

- Example
  - Single gate:

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

  - Circuit of connected gates:



Is output always 0 ?

Justify to "1"

$(\neg 1 + 2 + 4)(1 + \neg 4)(\neg 2 + \neg 4)$
$(\neg 2 + \neg 3 + 5)(2 + \neg 5)(3 + \neg 5)$
$(2 + \neg 3 + 6)(\neg 2 + \neg 6)(3 + \neg 6)$
$(\neg 4 + \neg 5 + 7)(4 + \neg 7)(5 + \neg 7)$
$(5 + 6 + 8)(\neg 5 + \neg 8)(\neg 6 + \neg 8)$
$(7 + 8 + 9)(\neg 7 + \neg 9)(\neg 8 + \neg 9)$
(9)

Circuit to CNF

- Circuit to CNF conversion
  - can be done in linear size (with respect to the circuit size) if intermediate variables can be introduced
  - may grow exponentially in size if no intermediate variables are allowed

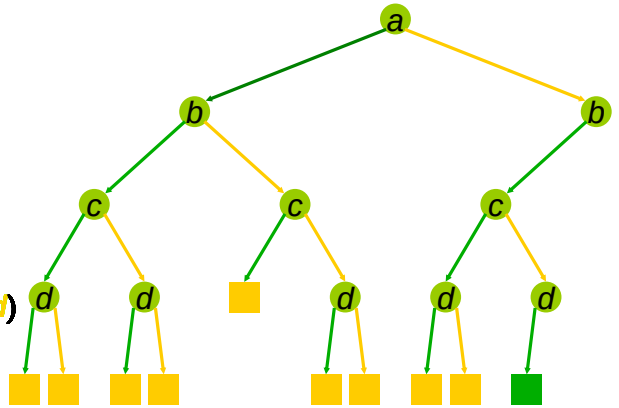
DPLL-Style SAT Solving

```
SAT(clause set S, literal v)
1. S :=  S_v   //cofactor each clause of S w.r.t. v
2. If no clauses in S, return T
3. If a clause in S is empty (FALSE), return
   F
4. If S has a unit clause with literal u,
   then return SAT(S, u) //implication
5. Choose a variable x with value not yet
   assigned
6. If SAT(S, x), return T
7. If SAT(S, ¬x), return T
8. Return F
```


SAT Solving with Case Splitting

- Example

1. $(a + b + c)$
2. $(a + b + \neg c)$
3. $(\neg a + b + \neg c)$
4. $(a + c + d)$
5. $(\neg a + c + d)$
6. $(\neg a + c + \neg d)$
7. $(\neg b + \neg c + \neg d)$
8. $(\neg b + \neg c + d)$



Source: Karem A. Sakallah, Univ. of Michigan

SAT Solving with Implication

- Implication in a CNF formula are caused by unit clauses
  - A unit clause is a clause in which all literals except one are assigned (to be false)
    - The value of the unassigned variable is implied

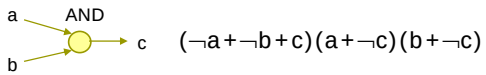
    Example

    (a+¬b+c)

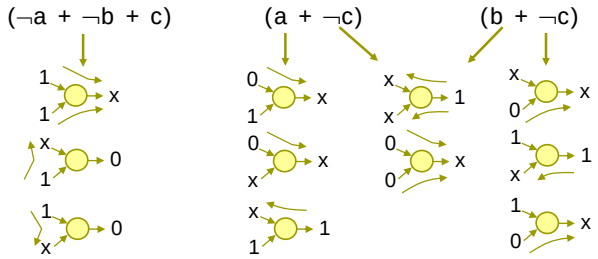
    a=0, b=1 ⇒ c=1

Implications in CNF

- Example



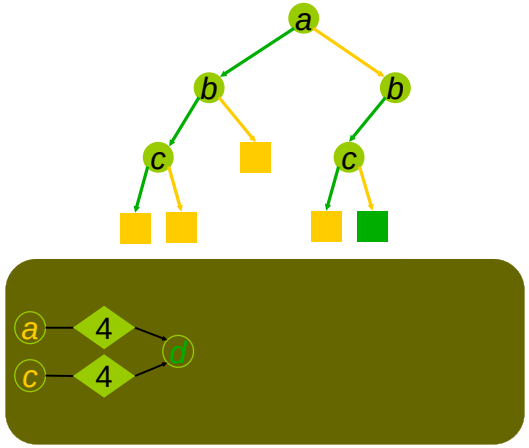
Implications:



SAT Solving with Implication

- Example

1 (a + b + c)
2 (a + b + ¬c)
3 (¬a + b + ¬c)
4 (a + c + d)
5 (¬a + c + d)
6 (¬a + c + ¬d)
7 (¬b + ¬c + ¬d)
8 (¬b + ¬c + d)

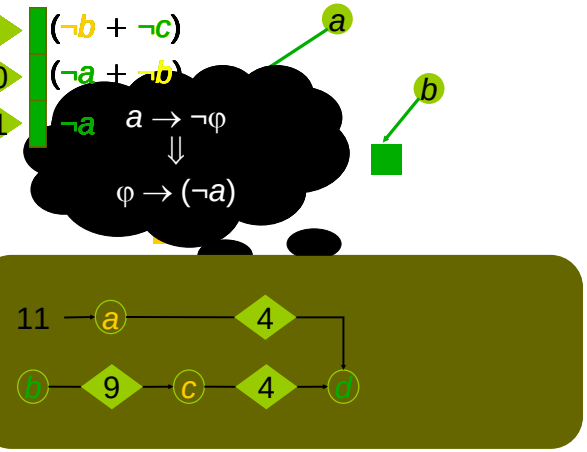


Source: Karem A. Sakallah, Univ. of Michigan

SAT Solving with Learning

- Example

1 (a + b + c)
2 (a + b + ¬c)
3 (¬a + b + ¬c)
4 (a + c + d)
5 (¬a + c + d)
6 (¬a + c + ¬d)
7 (¬b + ¬c + ¬d)
8 (¬b + ¬c + d)
9 (¬b + ¬c)
10 (¬a + ¬b)
11 ¬a



Source: Karem A. Sakallah, Univ. of Michigan

Implementation Issues

- Track sensitivity of clauses for changes (two-literal-watch scheme)
  - clause with all literals but one assigned → implication
  - clause with all literals but two assigned → sensitive to a change of either literal
  - all other clauses are insensitive and need not be observed
- Learning:
  - learned implications are added to the CNF formula as additional clauses
    - limit the size of the clause
    - limit the "lifetime" of a learned clause, will be removed after some time

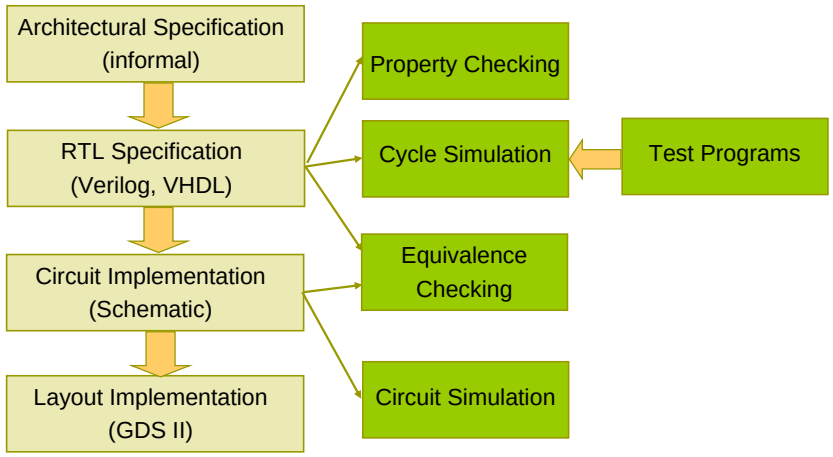
Quantification over CNF and DNF

- Recall a quantified Boolean formula (QBF) is
  $Q_1\ x_1,\ Q_2\ x_2,\ \ldots,\ Q_n\ x_n.\ \varphi$
  where Q*i* is either a existential (∃) or universal quantifier (∀), x_i is a Boolean variable, and φ is a Boolean formula.
- Existential (respectively universal) quantification over DNF (respectively CNF) is easy
  - One approach to quantifier elimination is by back-and-forth CNF-DNF conversion!
- Solving QBFs with QBF-solvers

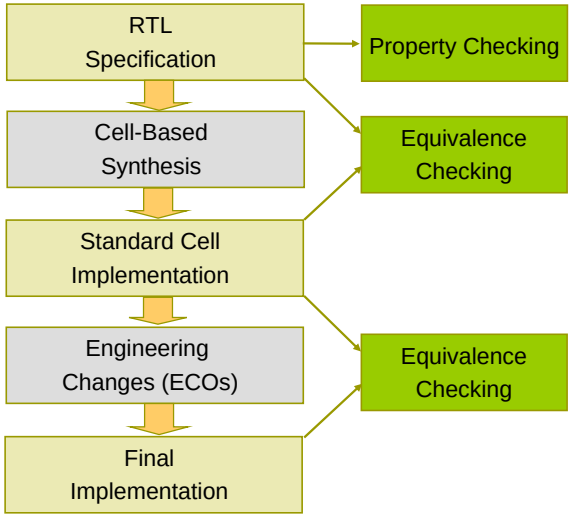
Outline

- Introduction
- Boolean reasoning engines
- Equivalence checking
- Property checking

Equivalence Checking in Microprocessor Design



Equivalence Checking in ASIC Design



Equivalence Checking

- Equivalence checking is one of the most important problem in design verification
  - It ensures logic transformation process (e.g. two-level, multi-level logic minimization, retiming and resynthesis, etc.) does not introduce errors
- Two types of equivalence checking
  - Combinational equivalence checking
    - Check if two combinational circuits are equivalent
  - Sequential equivalence checking
    - Check if two sequential circuits are equivalent

Outline

- Introduction
- Boolean reasoning engines
- Equivalence checking
  - Combinational equivalence checking
  - Sequential equivalence checking
- Property checking

History of Equivalence Checking

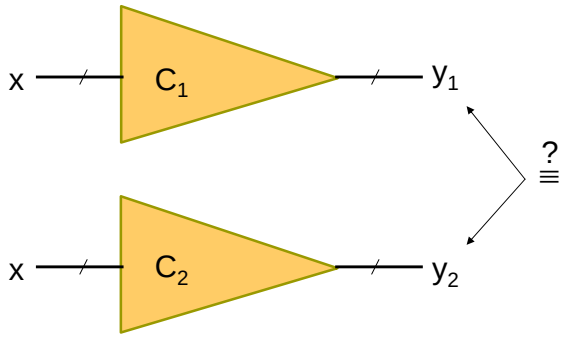
- SAS (IBM 1978 - 1994):
  - standard equivalence checking tool running on mainframes
  - based on the DBA algorithm (“BDDs in time”)
  - verified manual cell-based designs against RTL spec
  - handling of entire processor designs
    - application of “proper cutpoints”
    - application of synthesis routines to make circuits structurally similar
    - special hacks for hard problems
- Verity (IBM 1992 - today):
  - originally developed for switch-level designs
  - today IBMs standard EC tool for any combination of switch-, gate-, and RTL designs

History of Equivalence Checking

- Chrysalis (1994 - Avanti - now Synopsys):
  - based on ATPG technology and cutpoint exploitation
  - very weak if many cutpoints present
  - did not adopt BDDs for a long time
- Formality (1997 - Synopsys)
  - multi-engine technology including strong structural matching techniques
- Verplex (1998 - now Cadence)
  - strong multi-engine based tool
  - heavy SAT-based
  - very fast front-end

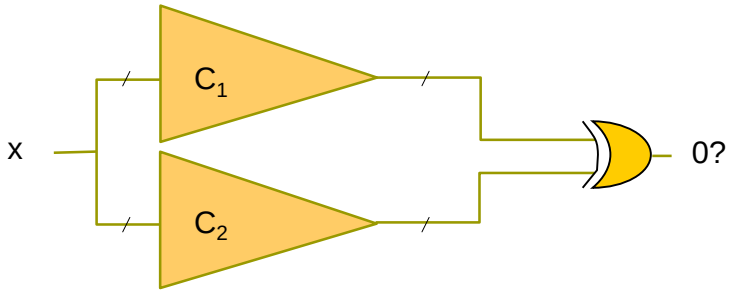
Combinational EC

- Given two combinational circuits C_1 and C_2, are their outputs equivalent under any possible input assignment?



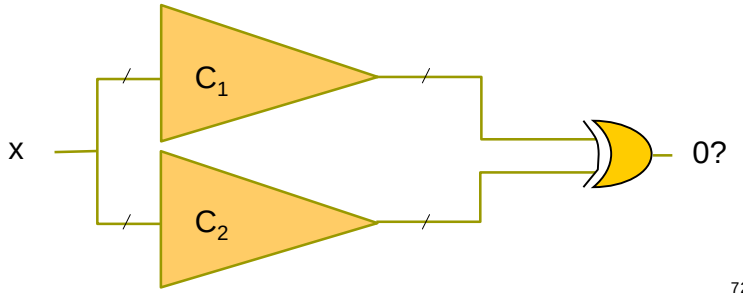
Miter for Combinational EC

- Two combinational circuits C_1 and C_2 are equivalent if and only if the output of their "miter" structure always produces constant 0



Approaches to Combinational EC

- Basic methods:
  - random simulation
    - good at identifying inequivalent signals
  - BDD-based methods
  - structural SAT-based methods



BDD-based Combinational EC

- Procedure
  1. Construct the ROBDDs F_1 and F_2 for circuits C_1 and C_2, respectively
     - Variable orderings of F_1 and F_2 should be the same
  2. Let $G = F_1 \oplus F_2$. If $G=0$, C_1 and C_2 are equivalent; otherwise, they are inequivalent
     - No false negative or false positive
       - False negative: circuits are equivalent; however, verifier fails to tell
       - False positive: circuits are inequivalent; however, verifier says otherwise

SAT-based Combinational EC

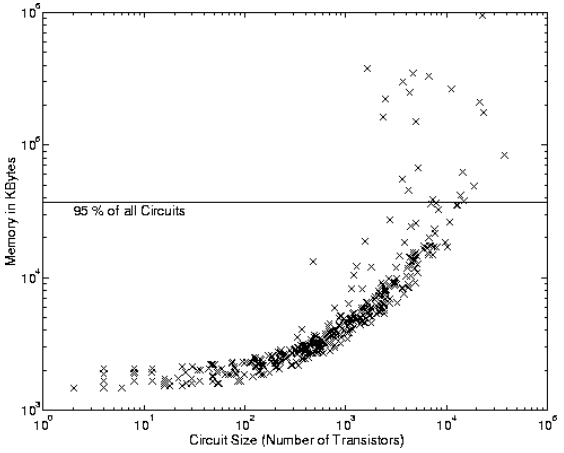
- Procedure
  1. Convert the miter structure into a CNF
  2. Perform SAT solving to verify if the output variable cannot be valuated to true under every input assignment (i.e. UNSAT)

Combinational EC

- Pure BDD and plain SAT solving cannot handle all logic cones
  - BDDs can be built for about 80% of the cones of high-speed designs and less for complex ASICs
  - plain SAT blows up in CPU time on a miter structure
- Contemporary method highly exploit structural similarities between two circuits to be compared

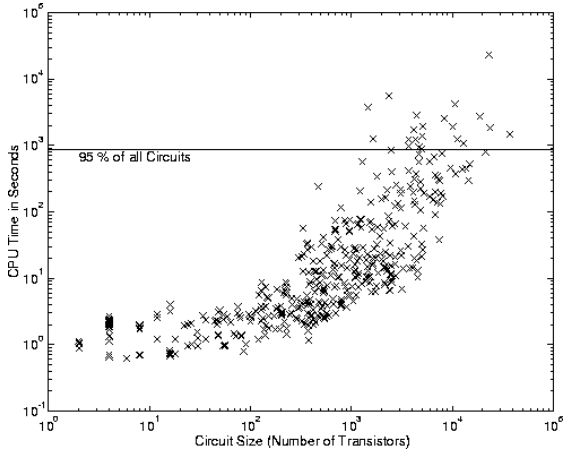
Combinational EC

- Memory statistics of BDD-based EC on a PowerPC processor design



Combinational EC

- Runtime statistics of BDD-based EC on a PowerPC processor design

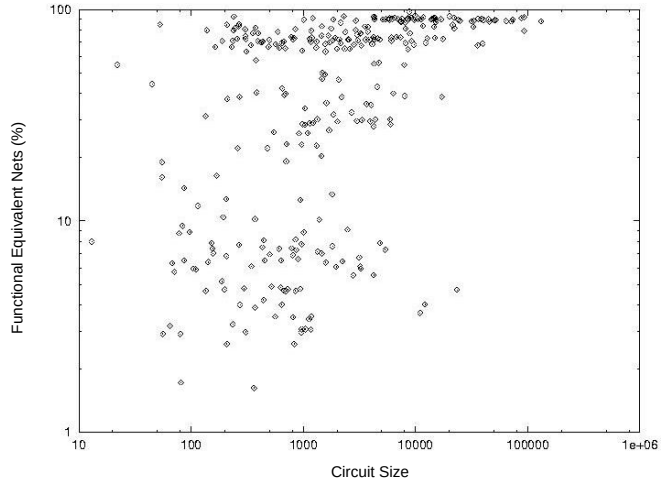


Necessity of Structure Similarity

- Pure BDDs are incapable of verifying equivalence of large circuits
  - Even more so for arithmetic circuits (e.g. BDDs blow up in representing multipliers)
- Identifying structure similarity helps simplify verification tasks
  - E.g. structure hashing in AIGs

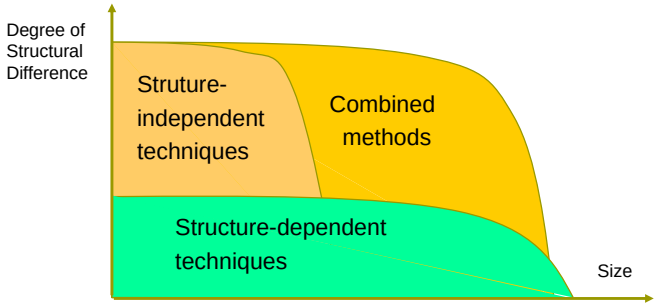
Combinational EC

- Evidence of vast existence of structure similarities



Structure and Verification

- Structure-independent techniques
  - Exhaustive simulation
  - Decision diagrams
- Structure-dependent techniques
  - Graph hashing
  - SAT based cutpoint identification



Summary

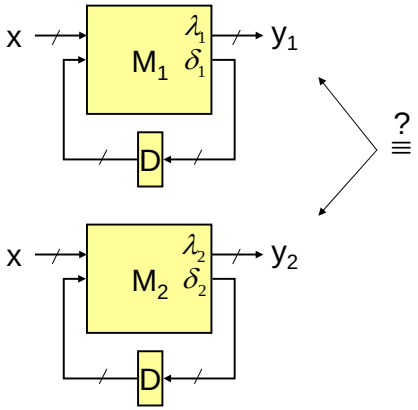
- Combinational EC is considered to be solvable in most industrial circuits (w/ multi-million gates)
  - Computational efforts scale almost linearly with the design size
  - Existence of structural similarities
    - Logic transformations preserve similarities to some extent
  - Hybrid engine of BDD, SAT, AIG, simulation, etc.
    - Cutpoint identification
- Unsolved for arithmetic circuits
  - Absence of structural similarities
    - Commutativity ruins internal similarities
  - Word- vs. bit-level verification

Outline

- Introduction
- Boolean reasoning engines
- Equivalence checking
  - Combinational equivalence checking
  - Sequential equivalence checking
- Property checking

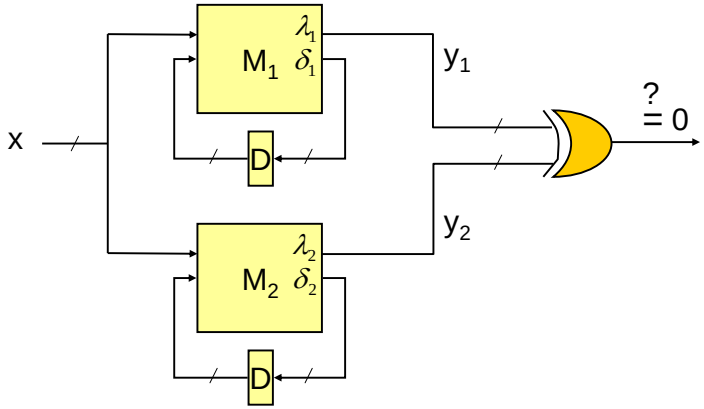
Sequential EC

- Given two sequential circuits (and thus FSMs), do they produce the same output sequence under any possible input sequence?



Miter for Sequential EC

- Two FSMs M_1 and M_2 are equivalent if and only if the output of their product machine always produces constant 0



Product Machine

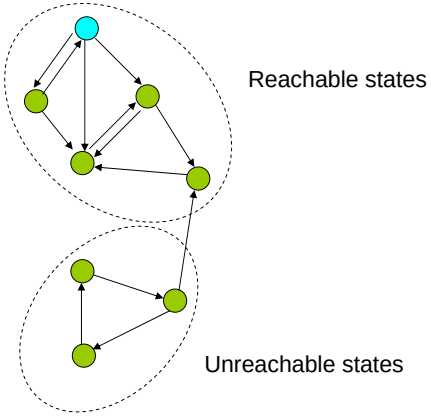
- The product FSM $M_{1\times2}$ of FSMs $M_1 = (Q_1, I_1, \Sigma, \Omega, \delta_1, \lambda_1)$ and $M_2 = (Q_2, I_2, \Sigma, \Omega, \delta_2, \lambda_2)$ is a six-tuple $(Q_{1\times2}, I_{1\times2}, \Sigma, \Omega, \delta_{1\times2}, \lambda_{1\times2})$, where
  - State space $Q_{1\times2} = Q_1 \times Q_2$
  - Initial state set $I_{1\times2} = I_1 \times I_2$
  - Input alphabet Σ
  - Output alphabet {0,1}
  - Transition function $\delta_{1\times2} = (\delta_1, \delta_2)$
  - Output function $\lambda_{1\times2} = (\lambda_1 \oplus \lambda_2)$

Sequential EC

- Approaches for combinational EC do not work for sequential EC because two equivalent FSMs need not have the same transition and output functions
  - False negatives may result from applying combinational EC on sequential circuits
- One solution to sequential EC is by reachability analysis
  - Two FSMs M_1 and M_2 are equivalent if and only if the output of their product FSM $M_{1\times2}$ is constant 0 under all input assignments and all reachable states of $M_{1\times2}$
  - Need to know the set of reachable states of $M_{1\times2}$

Reachability Analysis

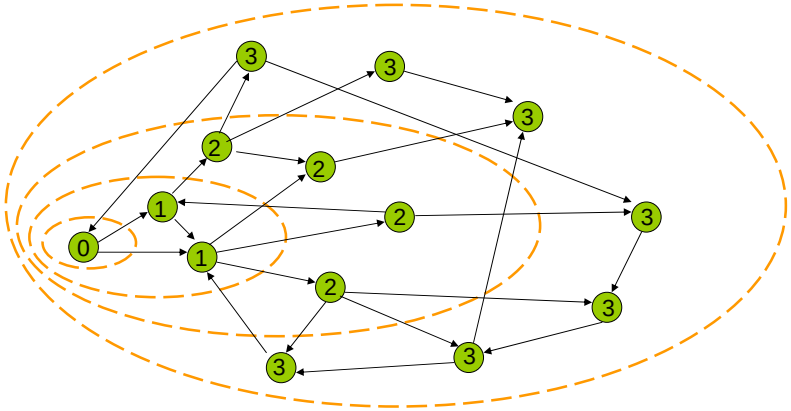
- Given an FSM $M = (Q, I, \Sigma, \Omega, \delta, \lambda)$, which states are reachable from the initial state set I ?



Symbolic Reachability Analysis

- Reachability analysis can be performed either explicitly (over a state transition graph) or implicitly (over transition functions or a transition relation)
  - Implicit reachability analysis is also called symbolic reachability analysis (often using BDDs and more recently SAT)
- Image computation is the core computation in symbolic reachability analysis

Reachability Onion Ring

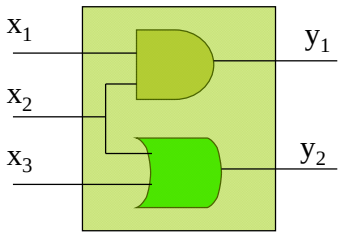


Computing Reachable States

- Input: Sequential system represented by a transition relation and an initial state (or a set of initial states)
  - Transition functions can be converted into a transition relation
- Computation: Image computation using Boolean operations on characteristic functions (representing state sets)
- Output: A characteristic function representing the set of reachable states

Relation

- Definition. Relation $R \subseteq X \times Y$ is a subset of the Cartesian product of two sets X and Y. If $(x,y) \in R$, then we alternatively write "x R y" meaning x is related to y by R.



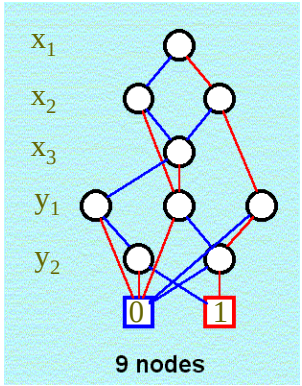
x_1	x_2	x_3	y_1	y_2
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

Courtesy of A. Mishchenko

Characteristic Function

- Relation $R \subseteq X \times Y$ can be represented by a characteristic function: a Boolean function $F_R(x,y)$ taking value 1 for those $(x,y) \in R$ and 0 otherwise.

x_1	x_2	x_3	y_1	y_2	F
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	0	1	1
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	1	1	1
1	1	1	1	1	1
other					0



Courtesy of A. Mishchenko

Transition Relation

- Definition. A transition relation T of an FSM M = (Q, I, Σ, Ω, δ, λ) is a relation $T \subseteq (\Sigma \times Q) \times Q$ such that $T(\sigma, q_1, q_2) = 1$ iff there is a transition from q_1 to q_2 under input σ.
  - δ: $(\Sigma \times Q) \rightarrow Q$
  - T: $(\Sigma \times Q) \times Q \rightarrow \{0,1\}$

Assume $\delta = (\delta_1, \ldots, \delta_\kappa)$. Then

$$T(\vec{x},\vec{s},\vec{s}') = (s_1' \equiv \delta_1(\vec{x},\vec{s})) \wedge (s_2' \equiv \delta_2(\vec{x},\vec{s})) \wedge \cdots \wedge (s_k' \equiv \delta_k(\vec{x},\vec{s}))$$
$$= \prod_i (s_i' \equiv \delta_i(\vec{x},\vec{s}))$$

where x, s, s' are primary-input, current-state, and next-state variables, respectively.

Quantified Transition Relation

- Definition

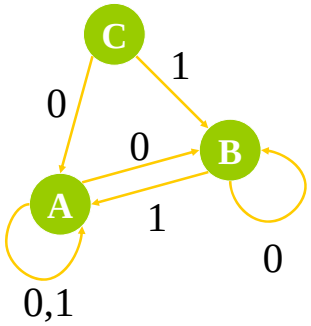
  Let M = (Q, I, Σ, Ω, δ, λ) be an FSM
  - Quantified transition relation $T_\exists$

$$T_\exists(\vec{s},\vec{s}') = \exists\vec{x}.(s_1' \equiv \delta_1(\vec{x},\vec{s})) \wedge (s_2' \equiv \delta_2(\vec{x},\vec{s})) \wedge \cdots \wedge (s_k' \equiv \delta_k(\vec{x},\vec{s}))$$
$$= \exists\vec{x}.\prod_i (s_i' \equiv \delta_i(\vec{x},\vec{s}))$$

  - (p,q) ∈ $T_\exists$ if there exists an input assignment bringing the M from state p to state q
  - only concerns about the reachability of the FSM's transition graph

Transition Relation

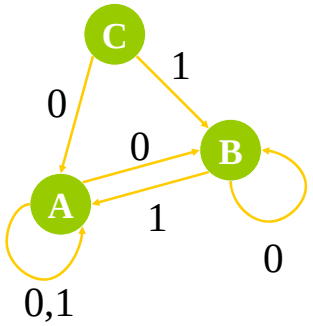
- Example



x	CS	$s_1 s_2$	NS	$s_1' s_2'$	T
0	A	00	B	10	1
0,1	A	00	A	00	1
0	B	10	B	10	1
1	B	10	A	00	1
0	C	01	B	10	1
1	C	01	A	00	1
other					0

Transition Relation

- Example



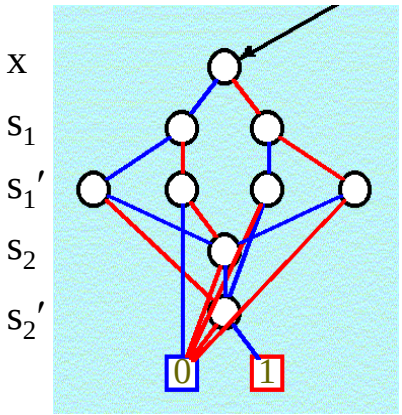


Image Computation

- Given a mapping of one Boolean space (input space) into another Boolean space (output space)
  - For a set of minterms (care set) in the input space
    - The image is the set of related minterms from the output space
  - For a set of minterms in the output space
    - The pre-image is the set of related minterms in the input space

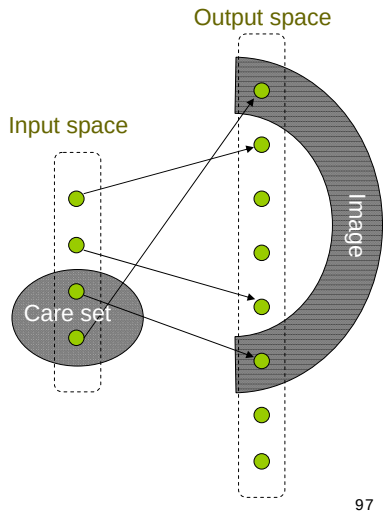
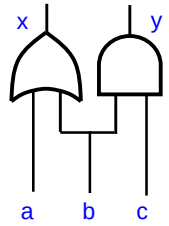


Image Computation

- Example



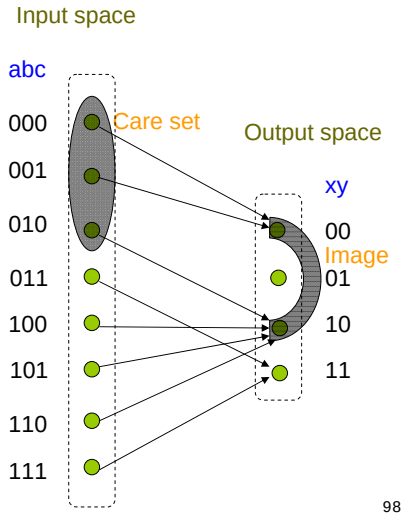


Image Computation

- Image(C(x),T(x,y)) = $\exists x\,[C(x) \wedge T(x,y)]$
- Implicit methods by far outperform explicit ones
  - Successfully computing images with more than 2^{100} minterms in the input/output spaces
- Operations $\wedge$ and $\exists$ are basic Boolean manipulations are implemented using BDDs
- To avoid large intermediate results (during and after the product computation), operation AND-EXIST is used, which performs product and quantification in one pass over the BDD

Symbolic Image Computation

- Definition. Let F: $B^m \times B^n$ be a projection and C be a set of minterms in B^m. Then the image of C is the set Img(C, F) = { w ∈ B^n | (v, w) ∈ F and v ∈ C} in B^n.
- Characteristic function
  - for reachable next-state computation

$$N_i(\vec{s}\,') = Img(R_i(\vec{s}), T_\exists(\vec{s}, \vec{s}\,'))$$
$$= \exists \vec{s}.(R_i(\vec{s}) \wedge T_\exists(\vec{s}, \vec{s}\,'))$$
$$= \exists \vec{s}.(R_i(\vec{s}) \wedge (\exists \vec{x}.\prod_i (s_i' \equiv \delta_i(\vec{x}, \vec{s}))))$$