2. Verification by Equivalence Checking

Combinational Circuits Verification

- Consist of an interconnection of logic gates — AND, OR, NOT, NAND, NOR, XOR, XNOR, and blocks implementing more complex logic (Boolean) functions.
- No logical loops, i.e., topologically there may be loops, but they are not sensitizable under any (valid) input combination, even such loops may be prohibited / not produced by automated analysis / synthesis tools

Goal

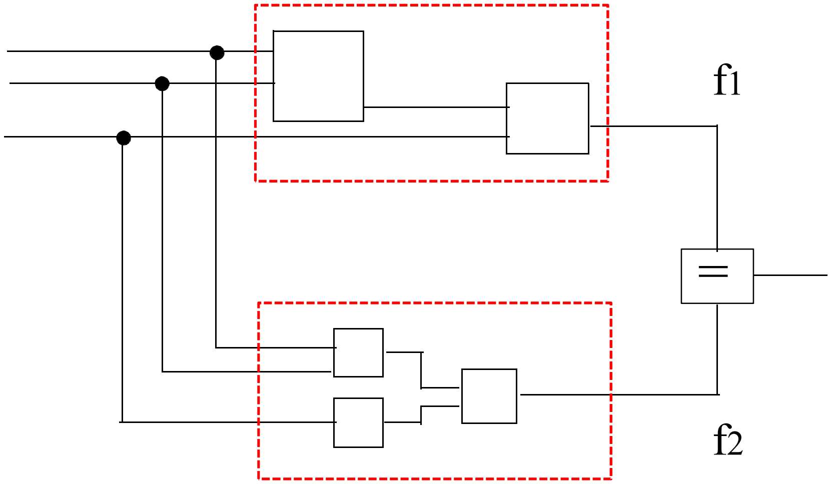
Given two Boolean netlists, check if the corresponding outputs of the two circuits are equal for all possible inputs

- Two circuits are equivalent iff the Boolean function representing the outputs of the networks are logically equivalent
- Identify equivalence points and implications between the two circuits to simplify equivalence checking
- Since a typical design proceeds by a series of local changes, in most cases there are many implications / equivalent subcircuits in the two circuits to be compared
- Various tautology/satisfiability checking algorithms based on heuristics (problem is NP-complete, but many work well on “real” applications ...)
- In this course we consider three main combinational equivalence checking methods:

  - **Propositional resolution method** (tautology/satisfiability checking)
  - **Stålmarck’s method** (recent patented algorithm, very efficient and popular)
  - **ROBDD-based method** (Boolean function converted into ROBDD’s representation)

Combinational Equivalence Checking

Explicit Proof



flag: T/F

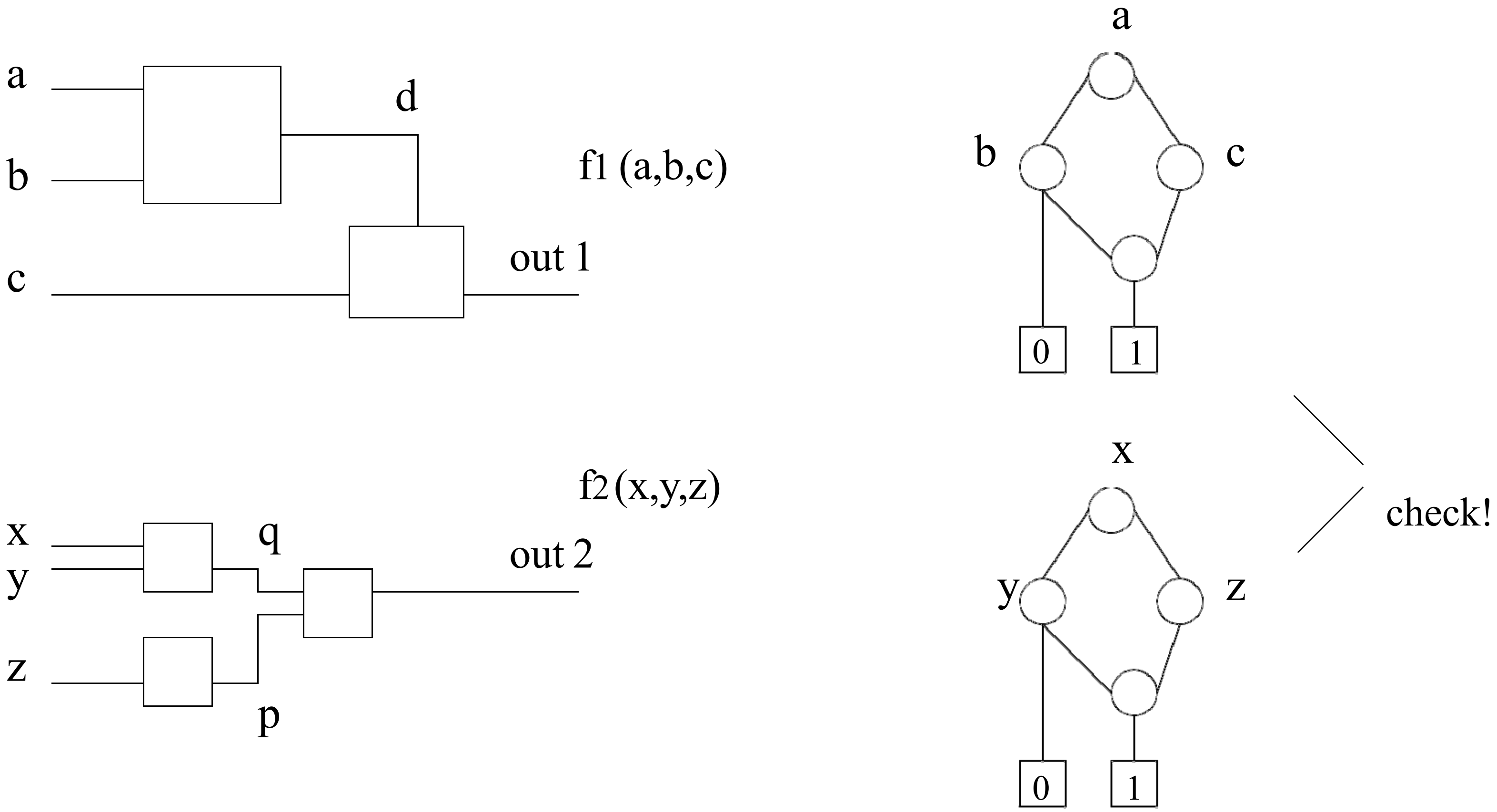
Tautology Check

$(f_1 = f_2) = T$

- Propositional resolution
- Stålmarck's procedure
- ROBDDs

Combinational Equivalence Checking (con’t)

Implicit Proof



- ROBDDs

Propositional Logic (Calculus)

Syntax

P, Q, R,... — propositional symbols (atomic propositions)

t: true; f: false — constants

$\neg$P: not P   P $\wedge$ Q: P and Q   P $\vee$ Q: P or Q;

P $\rightarrow$ Q: if P then Q (proposition equivalent to $\neg$P$\vee$Q)

P $\leftrightarrow$ Q: P if and only if Q, i.e., P equivalent to Q
(proposition equivalent to (P$\wedge$Q)$\vee$($\neg$P$\wedge$$\neg$Q))

Semantics

Given through the Truth Table:

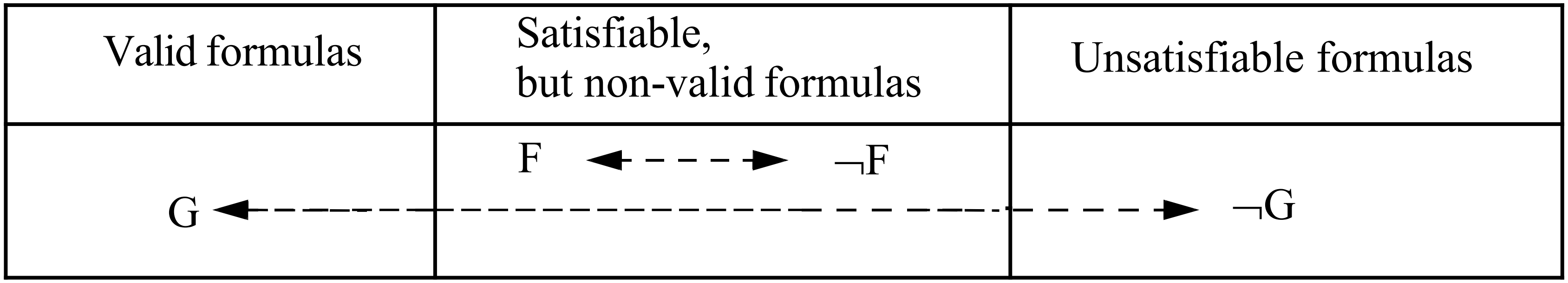
P	Q	$\neg$P	P$\wedge$Q	P$\vee$Q	P$\rightarrow$Q	P$\leftrightarrow$Q
t	t	f	t	t	t	t
t	f	f	f	t	f	f
f	t	t	f	t	t	f
f	f	t	f	f	t	t

An **interpretation** is a function from the propositional symbols to {t, f}

Propositional Logic (cont'd)

- Formula F is **satisfiable** (consistent) iff it is **true** under **at least one** interpretation
- Formula F is **unsatisfiable** (inconsistent) iff it is **false** under **all** interpretations
- Formula F is **valid** iff it is **true** (consistent) under **all** interpretations
- *Interpretation I satisfies a formula F* (*I is a **model** of F*) iff F is true under I.
  Notation: $I \models F$
- **Theorem:** A formula F is valid (a ***tautology***) iff ¬F is unsatisfiable. Notation: $\models F$
- The relationship between F to ¬F can be visualized by "mirror principle":

All formulas in propositional logic



Valid formulas	Satisfiable, but non-valid formulas	Unsatisfiable formulas
G	F ↔ ¬F	¬G

- To determine if F is satisfiable or valid, test finite number (2^n) of interpretations of the n atomic propositions occurring in F
  ... but it is an exponential method... **satisfiability is an NP-complete problem**

Propositional Logic (cont'd)

Proofs

- A proof of a proposition is derived using axioms, theorems, and inference rules (an inference rule permits deducing conclusions based on the truth of certain premises)
- A logic formula F is deducible from the set S of statements if there is a finite proof of F starting from elements of S. Notation: $S \vdash F$

Example: A simple proof system

- Axioms: K: $A \to (B \to A)$

  S: $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

  DN: $\neg\neg A \to A$
- Inference rule (Modus Ponens): $\{A \to B, A\} \vdash B$
- A proof of $A \to A$

(1) $\vdash (A\to((D\to A)\to A))\to((A\to(D\to A))\to(A\to A))$ by S ([B\D→A], [C\A])

(2) $\vdash A\to((D\to A)\to A)$ by K ([B\D→A])

(3) $\vdash (A\to(D\to A))\to(A\to A)$ by MP, (1), (2)

(4) $\vdash A\to(D\to A)$ by K

(5) $\vdash A\to A$ by MP, (3), (4).

Propositional Logic (cont'd)

Relation between syntax and semantics

- Truth tables provide a means of deciding truth
- Propositional logic is:
  - **complete**: everything that is true may be proven, i.e., if $S \vdash A$ then $S \models A$
  - **consistent** (**sound**): nothing that is false may be proven. i.e., if $S \models A$ then $S \vdash A$
  - **decidable**: there is an algorithm for deciding the truth of any proposition, i.e., test a finite (exponential) number of truth assignments

False Negative & False Positive

Let P be a proposition (a property) and A a verification method (algorithm).

- **False Negative**: (similar to incompleteness)

  A(P) reports true ⇒ $\forall$ interpretation ψ, $\psi(P) = \text{true}$

  A(P) reports false ⇒ $\neg(\forall$ interpretation ψ, $\psi(P) = \text{true})$ **!** ($\exists\ \psi$, $\psi(P) = \text{false}$)

- **False Positive**: (similar to inconsistency, unsoundness)

  A(P) reports false ⇒ $\forall$ interpretation ψ, $\psi(P) = \text{false}$

  A(P) reports true ⇒ $\neg(\forall$ interpretation ψ, $\psi(P) = \text{false})$ **!** ($\exists\ \psi$, $\psi(P) = \text{true}$)

Combinational Equivalence Checking

- Determine if two expressions f1 and f2 denote the same truth table

- Application: Determine if two combinational logic circuit designs C1 and C2 implement the same truth table (logic (Boolean) function)
  - Extract representation of logic expressions f1 and f2
  - Verify if

    (f1 ↔ f2) is a valid formula, i.e., ¬(f1 ↔ f2) is unsatisfiable
    using **satisfiability** algorithms (**Propositional Resolution** methods), or

    (f1 → f2) and (f2 → f1) hold (where f1 and f2 are transformed to
    implication form using **Stålmarck's procedure**), or

    f1 and f2 have the same *canonical form*
    using, e.g., **Reduced Binary Decision Diagrams**

Propositional Resolution

- A **Literal** L is an atomic proposition A or its negation $\neg A$
- A **Clause** C is a finite set of disjunctive literals ($C = L_1 \vee L_2 \vee L_3 \vee \ldots$)

  C is true iff one of its elements is true. The empty clause $\{\square\}$ is always false.

Let $A_1, A_2, \ldots$ be atomic propositions and $L_{i,j}$ literals

- **Conjunctive Normal Form** (CNF): a conjunction of disjunctions of literals

$$F=(\bigwedge_{i=1}^{n}(\bigvee_{j=1}^{m_i} L_{i,j})), \text{ where } L_{i,j} \in \{A_1,A_2, \ldots\} \cup \{\neg A_1, \neg A_2, \ldots\}$$

- **Disjunctive Normal Form** (DNF): a disjunction of conjunctions of literals

$$F=(\bigvee_{i=1}^{n}(\bigwedge_{j=1}^{m_i} L_{i,j})), \text{ where } L_{i,j} \in \{A_1,A_2,\ldots\} \cup \{\neg A_1, \neg A_2,\ldots\}$$

Each $L_{i,j}$ $\in \{A1,A2, \ldots\} \cup \{\neg A1, \neg A2, \ldots\}$ appears in each disjunct (conjunct) **at most once!**

Theorem: For every logic formula F, there is an equivalent CNF and an equivalent DNF

- **Canonical Conjunctive Form** (CCF): CNF in which each L appears exactly once
- **Canonical Disjunctive Form** (DCF): DNF in which each L appears exactly once

Propositional Resolution (cont'd)

- **Resolution** is a proof method underlying some automatic theorem provers based on simple syntactic transformation and *refutation.*
- **Refutation** is a procedure to show that a given formula is unsatisfiable

Resolution procedure:

- To prove F, we translate ¬F into a set of clauses, each a disjunction of atomic formulae or their negations.

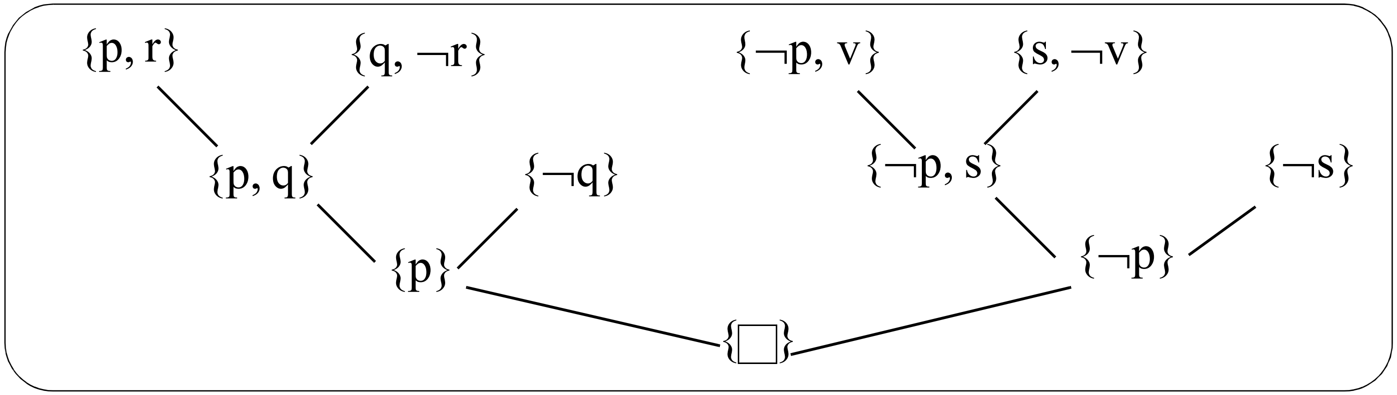
- Each resolution step takes two clauses and yields a new one.

- The method succeeds if it produces the empty clause (a contradiction), thus refuting ¬F.

Propositional Resolution (cont’d)

- Let $F=(L_{1,1}\vee\ldots\vee L_{1,n1})\wedge\ldots\wedge(L_{k,1}\vee\ldots\vee L_{k,nk})$ where literals $L_{i,\,j} \in \{A_1,A_2,\ldots\}\cup\{\neg A_1,\neg A_2,\ldots\}$

  F can be viewed as a set of clauses: $F=\{\{L_{1,1},\ldots, L_{1,n1}\},\ldots, \{L_{k,1},\ldots, L_{k,nk}\}\}$, where
  - Comma separating two literals within a clause corresponds to $\vee$
  - Comma separating two clauses corresponds to $\wedge$
- Let L be a literal in clause C_1 ($L\in C_1$) and its complement $\overline{L}$ in clause C_2 ($\overline{L} \in C_2$), Clause R is a **resolvent** of C_1 and C_2 if: $R = (C_1 - \{L\}) \cup (C_2 - \{\overline{L}\})$
- Example: $F = \{\{p, r\}, \{q, \neg r\}, \{\neg q\}, \{\neg p, v\}, \{\neg s\}, \{s, \neg v\}\}$.



Propositional Resolution (cont’d)

- A *(resolution) deduction* of C from F is a finite sequence $C_1, C_2, ..., C_n$ of clauses such that each C_i is either in F or a resolvent of C_j, C_k , (j, k < i)
- $Res(F) = F \cup R$ where R is a resolvent of two clauses in F

Lemma. F and $F \cup R$ are equivalent

- Define

$$Res^0(F) = F,$$

$$Res^{n+1}(F) = Res(\ Res^n(F)\ ),\ n \geq 0$$

- Let $Res^*(F) = \bigcup_{n \geq 0} Res^n(F)$

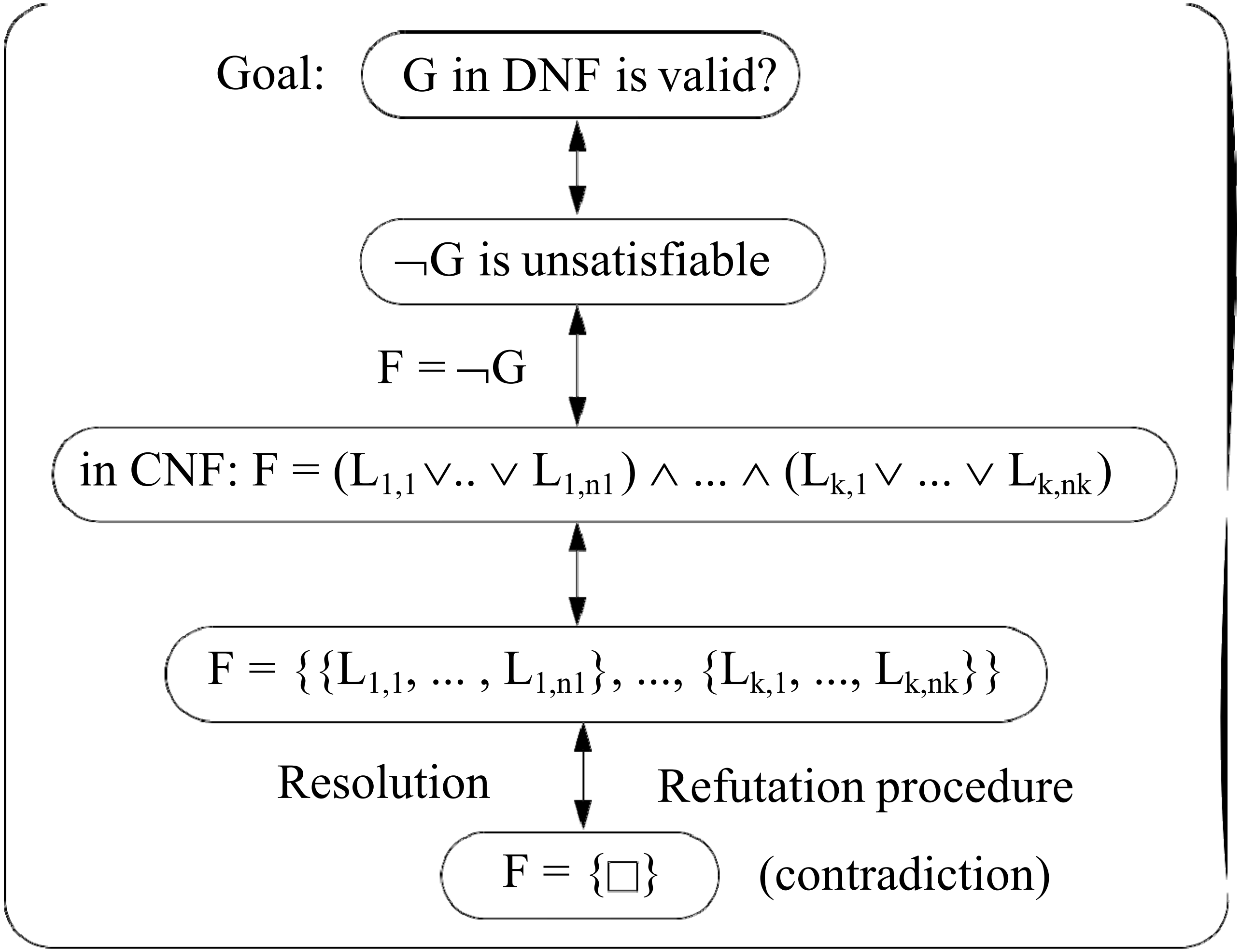
Theorem. F is unsatisfiable iff $\square \in Res^*(F)$

- Algorithm: to decide satisfiability of formula F in CNF (clause set):

```
repeat
    G:=F;
    F:=Res(F)
until (($\square$ $\in$ F) or (F = G);
if $\square$ $\in$ F then “F is unsatisfiable” else “F is satisfiable”.
```

Propositional Resolution (cont'd)

Summary of basic idea:



Goal: G in DNF is valid?

$\neg$G is unsatisfiable

F = $\neg$G

in CNF: $F = (L_{1,1} \vee .. \vee L_{1,n1}) \wedge ... \wedge (L_{k,1} \vee ... \vee L_{k,nk})$

$F = \{\{L_{1,1}, ... , L_{1,n1}\}, ..., \{L_{k,1}, ..., L_{k,nk}\}\}$

Resolution

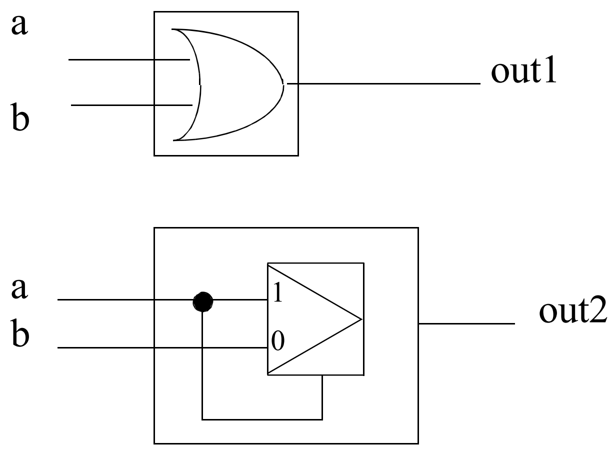
Refutation procedure

$F = \{\square\}$

(contradiction)

Propositinal Resolution - Example

Two circuits C1 and C2



C1

= ?

C2

Propositional Resolution

C1: $out1 = a \vee b$

C2: $out2 = (\neg a \wedge b) \vee (a \wedge a)$

(Mux: $out2 = (\neg s \wedge b) \vee (s \wedge a)$)

$G = (out1 \Leftrightarrow out2)$

$G = (\text{out } 1 \wedge \text{out } 2) \vee (\neg\text{out1} \wedge \neg\text{out } 2)$ **(DNF)**

$= \text{true?}$

$F = \neg\, G = \neg\, ((\text{out } 1 \wedge \text{out } 2) \vee (\neg\text{out1} \wedge \neg\text{out } 2))$

$= \text{False?}$ (unsatifiable!)

CNF

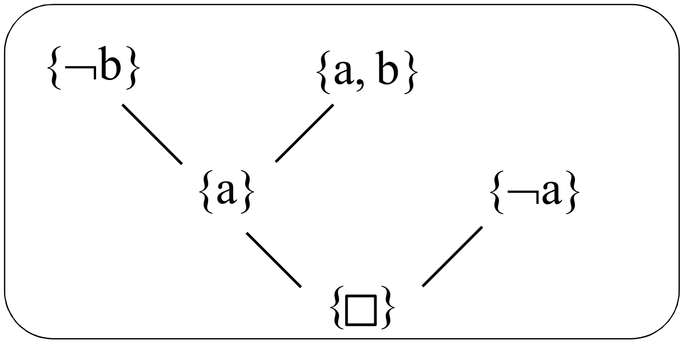
$F = (\neg\text{out } 1 \vee \neg\text{out } 2) \wedge (\text{out1} \vee \text{out } 2)$

$= (\neg(a \vee b) \vee \neg\, [(\neg\, a \wedge b) \vee (a \wedge a)]) \wedge ((a \vee b) \vee [(\neg\, a \wedge b) \vee (a \wedge a)])$

$= .....$

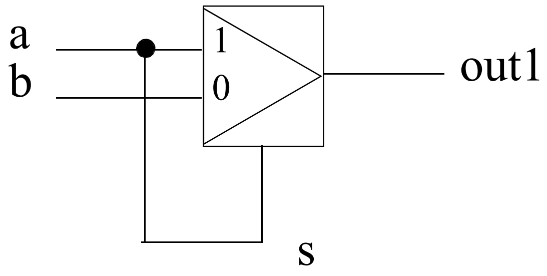
$= (\neg a) \wedge (\neg b) \wedge (a \vee b)$

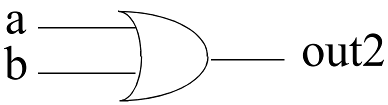
Literals: $\{\{\neg a\}, \{\neg b\}, \{a, b\}\}$



$\Rightarrow$ derive **empty clause** $\{\square\}$

Theorem Proving





$\text{out1} = (\neg\, s \wedge b) \vee (s \wedge a)$

$= (\neg\, a \wedge b) \vee (a \wedge a)$

$= (\neg\, a \wedge b) \vee \quad a$

$= (\neg\, a \vee a) \wedge (b \vee a)$

$= \quad 1 \quad \wedge (b \vee a)$

$= \quad b \vee a \quad = \quad a \vee b$

$\Rightarrow \quad \text{out2} = \text{out1}$

$\text{out2} = a \vee b$

Stålmarck's Procedure

- Transform propositional formula G (in linear time) in a nested implication form, e.g.: $G = (p \rightarrow (q \rightarrow r)) \rightarrow s$
- G is now represented using a set of triplets $\{b_i, x, y\}$, meaning "$b_i \leftrightarrow (x \rightarrow y)$", e.g.: $(p \rightarrow (q \rightarrow r)) \rightarrow s$ becomes $\{(b_1, q, r), (b_2, p, b_1), (b_3, b_2, s)\}$; $G = b_3$
- To prove a formula valid, assume that it is *false* and try to find a contradiction (use 0 for *false* and 1 for *true*, as in switching (Boolean) algebra)
- Derivation rules: (a/b means "replace a by b")

r1	$(0, y, z) \Rightarrow y/1, z/0$	meaning	*false* $\leftrightarrow (y \rightarrow z)$ implies y = *true* and z = *false*
r2	$(x, y, 1) \Rightarrow x/1$	meaning	*x* $\leftrightarrow$ (y $\rightarrow$ *true*) implies x = *true*
r3	$(x, 0, z) \Rightarrow x/1$	meaning	*x* $\leftrightarrow$ (*false* $\rightarrow$ z) implies x = *true*
r4	$(x, 1, z) \Rightarrow x/z$	meaning	*x* $\leftrightarrow$ (*true* $\rightarrow$ z) implies x = z
r5	$(x, y, 0) \Rightarrow x/\neg y$	meaning	$x \leftrightarrow (y \rightarrow 0)$ implies $x = \neg y$
r6	$(x, x, z) \Rightarrow x/1, z/1$	meaning	*x* $\leftrightarrow$ (x $\rightarrow$ z) implies x = *true* and z = *true*
r7	$(x, y, y) \Rightarrow x/1$	meaning	*x* $\leftrightarrow$ (y $\rightarrow$ y) implies x = *true*

Example: $G = (p \rightarrow (q \rightarrow p))$: $\{(b_1, q, p), (b_2, p, b_1)\}$, assume $G = b_2 = 0$, i.e., $(0, p, b_1)$

By r1 : $p = 1$ and $b_1 = 0$, substitute for b_1 and get $(0, q, 1)$ (which is a terminal triplet)

Again by r1 this is a contradiction since 1/0 is derived for z in r1, hence $b_2 = G = 1$ (true)

Stålmarck's Procedure (cont'd)

- Not all formulas can be proved with these rules, need a form of branching: **Dilemma rule** T = a set of triplets, D_i, i = 1, 2, are derivations, results $U[S_1]$ and $V[S_2]$, conclusion $T[S]$

$$\begin{array}{c} T \\ \hline \begin{array}{cc} T[x/1] & T[x/0] \\ D_1 & D_2 \\ U[S_1] & V[S_2] \end{array} \\ \hline T[S] \end{array}$$

Assume x = 0 derive a result, then assume x = 1 and also derive a result.

- If either derivation gives a contradiction, the result is the other derivation
- If both are contradictions, then T contains a contradiction
- Otherwise return the intersection of the result of the two derivations, since any information gained from x = 0 and x = 1 must be independent of that value

Example: $T = \{ (1, \neg p, p), (1, p, \neg p) \}$ cannot be resolved using r1 - r7
$T[p/1] = \{(1, 0, 1), (1, 1, 0)\}$ where (1, 1, 0) is a contradiction
$T[p/0] = \{(1, 1, 0), (1, 0, 1)\}$ where (1, 1, 0) is again a contradiction
Hence T[S] results in a contradiction.

Stålmarck’s Procedure (cont’d)

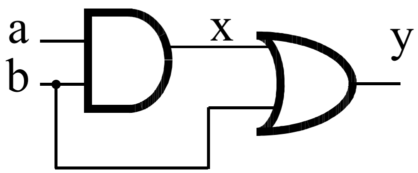
Transformation from and-or-not logic to implication form:

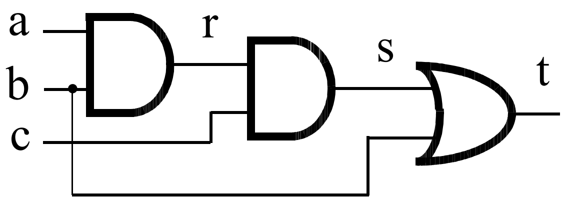
not: $G = \neg A \Leftrightarrow A \rightarrow 0 \Leftrightarrow \{(x, A, 0)\}$, $G = x$

or: $G = A \vee B \Leftrightarrow \neg A \rightarrow B \Leftrightarrow \{(x, y, B), (y, A, 0)\}$, $G = x$

and: $G = A \wedge B \Leftrightarrow \neg(A \rightarrow \neg B) \Leftrightarrow \{(x, y, 0), (y, A, z), (z, B, 0)\}$, $G = x$

Example of equivalence checking:





C1 = {(y, e, x), (e, b, 0), (x, f, 0), (f, a, g), (g, b, 0)}

C2 = {(t, h, s), (h, b, 0), (s, u, 0), (u, r, v), (v, c, 0), (r, w, 0), (w, a, p), (p, b, 0)}

Check $y \rightarrow t$ and $t \rightarrow y$

$y \rightarrow t$: Form $C1 \cup C2 \cup \{(0, y, t)\}$ which by r1 yields [y/1, t/0] and after substitution

{**(1, e, x)**, (e, b, 0), (x, f, 0), (f, a, g), (g, b, 0), **(0, h, s)**, (h, b, 0), (s, u, 0), (u, r, v), (v, c, 0), (r, w, 0), (w, a, p), (p, b, 0)} giving by r1 again [h/1, s/0] and...

Stålmarck's Procedure (cont'd)

Example of equivalence checking (cont'd):

{**(1, e, x)**, (e, b, 0), (x, f, 0), (f, a, g), (g, b, 0), **(1, b, 0)**, **(0, u, 0)**, (u, r, v), (v, c, 0), (r, w, 0), (w, a, p), (p, b, 0)} apply r1 and r5 and get [u/1, e/¬b, x/¬f, g/¬b, v/¬c, r/¬w, p/¬b] which yields

{(1, ¬b, ¬f), (f, a, ¬b), (¬b, b, 0), (1, ¬w, ¬c), (w, a, ¬b)}

Application of Dilemma rule to, say, "b" yields:

b = 0: {(1, 1, ¬f), (f, a, 1), (1, 0, 0), (1, ¬w, ¬c), (w, a, 1)} apply r2 and get [f/1, w/1] yields {**(1,1, 0)**, (1, a, 1), (1, 0, ¬c), (1, a, 1)}, where **(1,1, 0) is a contradiction**

b = 1: {(1, 0, ¬f), (f, a, 0), (0, 1, 0), (1, ¬w, ¬c), (w, a, 0)} yields [f/¬a, w/¬a] by r5, thus {**(1, 0, a), (¬a, a, 0),** (0, 1, 0), **(1, a, ¬c), (¬a, a, 0)**}, then applying the Dilema rule again on "a" leads to **a contradiction again**

Conclusion: y → t holds.
Similarly for t → y

***The two circuits are equivalent*.**

Binary Decision Diagrams (BDDs)

Classical representation of logic functions: Truth Table, Karnaugh Maps, Sum-of-Products, critical complexes, etc.

- Critical drawbacks:

- May not be a canonical form or is too large (exponential) for “useful” functions,

⇒ Equivalence and tautology checking is hard

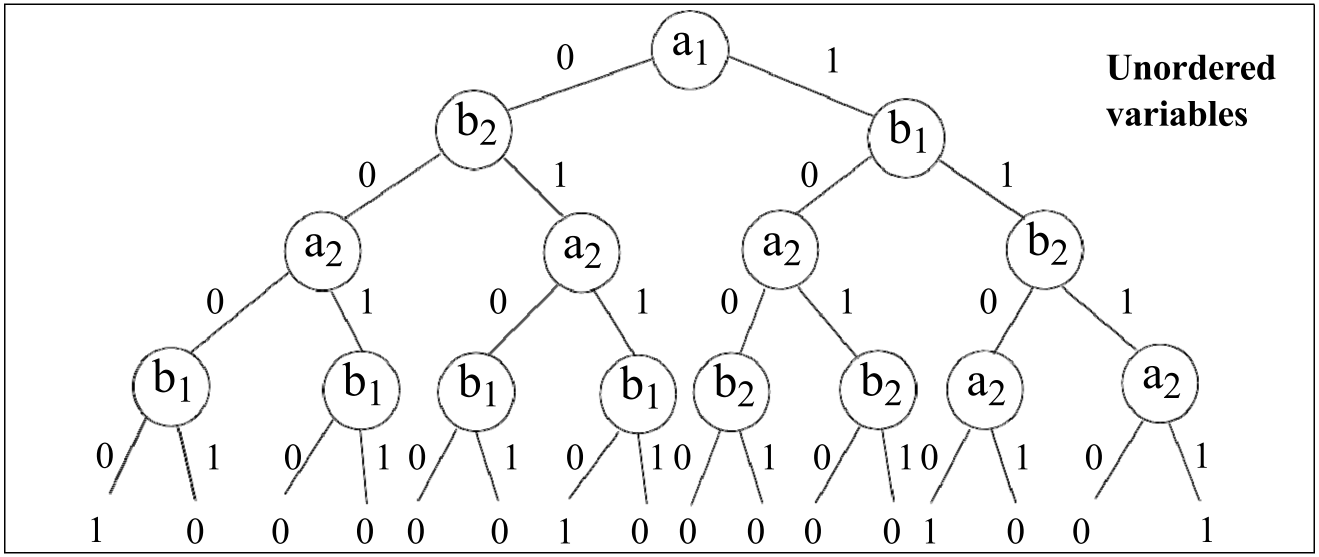
- Operations like complementation may yield a representation of exponential size

Reduced Ordered Binary Decision Diagrams (ROBDDs)

- A canonical form for Boolean functions
- Often substantially more compact than traditional normal forms
- Can be efficiently manipulated
- Introduced mainly by R. E. Bryant (1986).
- Various extensions exist that can be adapted to the situation at hand (e.g., the type of circuit to be verified)

Binary Decision Trees

- A Binary decision Tree (BDT) is a *rooted*, *directed graph* with *terminal* and *nonterminal vertices*
- Each nonterminal vertex *v* is labeled by a variable *var(v)* and has two successors:
  - *low(v)* corresponds to the case where the variable *v* is assigned 0
  - *high(v)* corresponds to the case where the variable *v* is assigned 1
- Each terminal vertex *v* is labeled by *value(v)* $\in \{0, 1\}$
- **Example**: BDT for a two-bit comparator, $f(a_1,a_2,b_1,b_2) = (a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2)$



Unordered variables

Binary Decision Trees (cont'd)

- We can decide if a truth assignment $\underline{x} = (x_1, ..., x_n)$ satisfies a formula in BDT in linear time in the number of variables by traversing the tree from the root to a terminal vertex:
  - If *var(v)* $\in \underline{x}$ is 0, the next vertex on the path is *low(v)*
  - If *var(v)* $\in \underline{x}$ is 1, the next vertex on the path is *high(v)*
  - If *v* is a terminal vertex then $f(\underline{x}) = f_v(x_1, ..., x_n) =$ *value(v)*
  - If *v* is a nonterminal vertex with *var(v)*$=x_i$, then the structure of the tree is obtained by Shanon's expansion

$$f_v(x_1, ..., x_n) = [\neg x_i \wedge f_{low(v)}(x_1, ..., x_n)] \vee [x_i \wedge f_{high(v)}(x_1, ..., x_n)]$$

- For the comparator, $(a_1 \leftarrow 1, a_2 \leftarrow 0, b_1 \leftarrow 1, b_2 \leftarrow 1)$ leads to a terminal vertex labeled by 0, i.e., $f(1, 0, 1, 1) = 0$
- Binary decision trees are redundant:
  - In the comparator, there are 6 subtrees with roots labeled by b_2, but not all are distinct
- Merge isomorphic subtrees:
  - Results in a directed acyclic graph (DAG), a ***binary decision diagram (BDD)***

Reduced Ordered BDD

Canonical Form property

- A *canonical* representation for Boolean functions is desirable:

  two Boolean functions are logically equivalent iff they have isomorphic representations
- This simplifies checking equivalence of two formulas and deciding if a formula is satisfiable
- Two BDDs are **isomorphic** if there exists a bijection *h* between the graphs such that

  - Terminals are mapped to terminals and nonterminals are mapped to nonterminals

  - For every terminal vertex *v*, *value(v)* = *value(h(v))*, and

  - For every nonterminal vertex *v*:

  $$var(v) = var(h(v)), \quad h(low(v)) = low(h(v)), \text{ and } \quad h(high(v)) = high(h(v))$$
- **Bryant (1986) showed that BDDs are a canonical representation** for Boolean functions under two restrictions:

  (1) the variables appear **in the same order along each path** from the root to a terminal

  (2) there are **no isomorphic subtrees or redundant vertices**

  ⇒ **Reduced Ordered Binary Decision Diagrams (ROBDDs)**

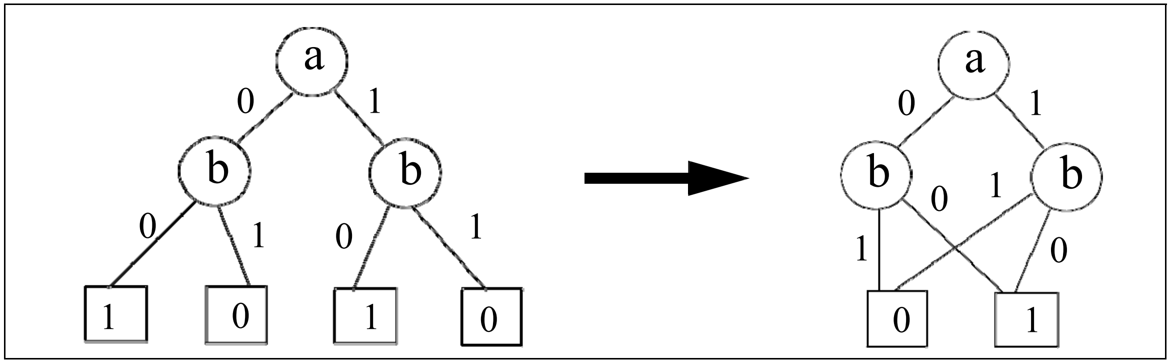
Canonical Form Property

- Requirement (1): Impose total order "<" on the variables in the formula:

  if vertex *u* has a nonterminal successor *v*, then *var(u) < var(v)*

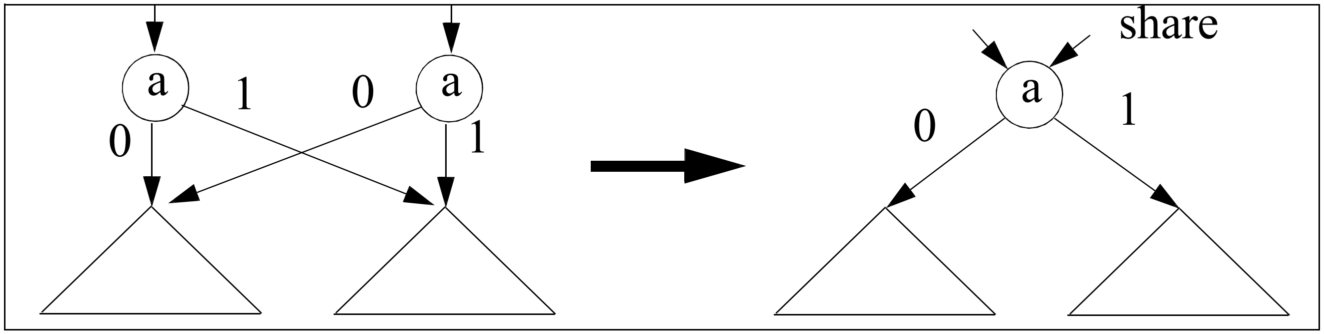
- Requirement (2): repeatedly apply three transformation rules (or implicitly in operations such as disjunction or conjunction)

1.Remove duplicate terminals: eliminate all but one terminal vertex with a given label and redirect all arcs to the eliminated vertices to the remaining one

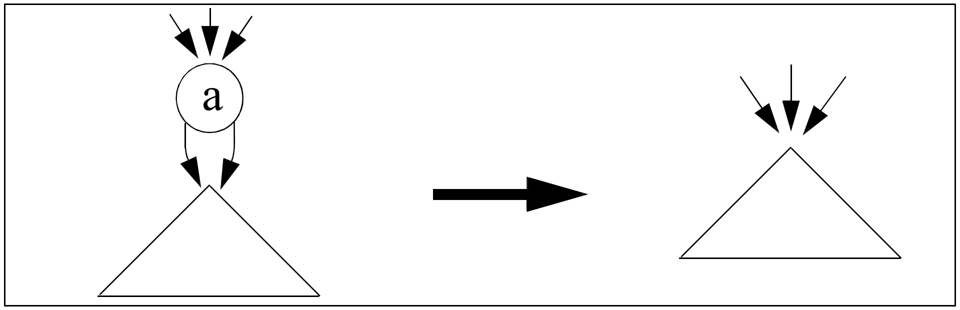


Canonical Form Property (cont'd)

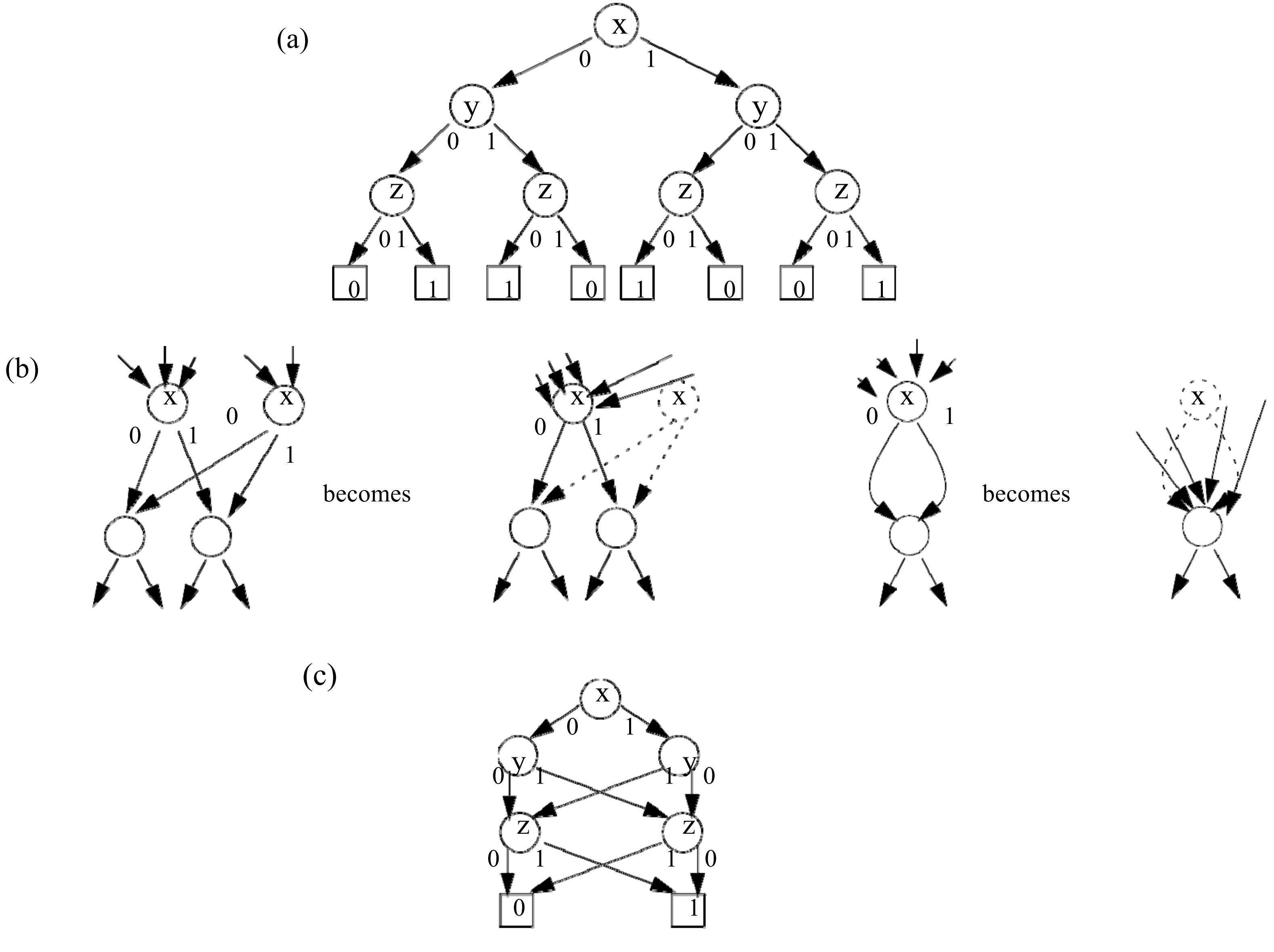
2. Remove duplicate nonterminals: if nonterminals *u* and *v* have *var(u) = var(v)*, *low(u) = low(v)* and *high(u) = high(v)*, eliminate one of the two vertices and redirect all incoming arcs to the other vertex



3. Remove redundant tests: if nonterminal vertex *v* has *low(v) = high(v)*, eliminate *v* and redirect all incoming arcs to *low(v)*



Creating the ROBDD for $(x \oplus y \oplus z)$



Canonical Form Property (cont'd)

- A canonical form is obtained by applying the transformation rules until no further application is possible
- Bryant showed how this can be done by a procedure called ***Reduce*** in linear time
- Applications:

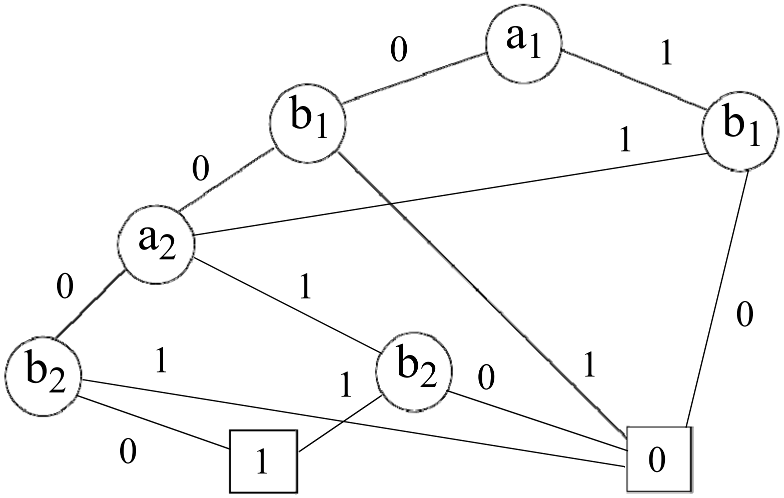
  - *checking equivalence*: verify isomorphism between ROBDDs

  - *non-satisfiability*: verify if ROBDD has only one terminal node, labeled by 0

  - *tautology*: verify if ROBDD has only one terminal node, labeled by 1

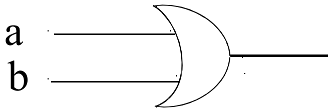
Example:

ROBDD of 2-bit comparator $f(a_1,a_2,b_1,b_2) = (a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2)$ with variable order $\mathbf{a_1 < b_1 < a_2 < b_2}$:



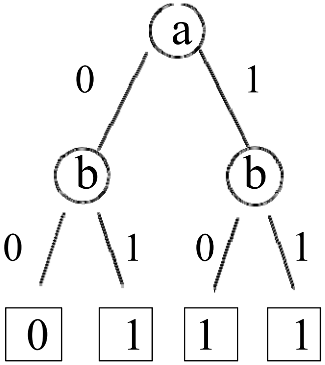
ROBDD Examples

OR

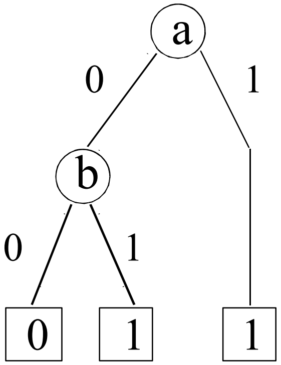


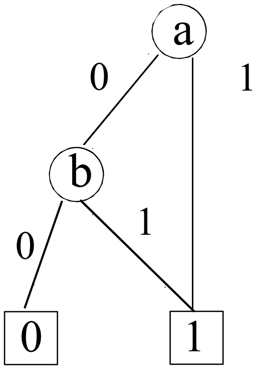
out = f (a,b) = a ∨ b

a	b	out
0	0	0
0	1	1
1	0	1
1	1	1



BDD

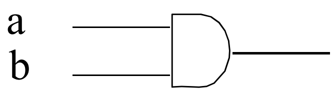




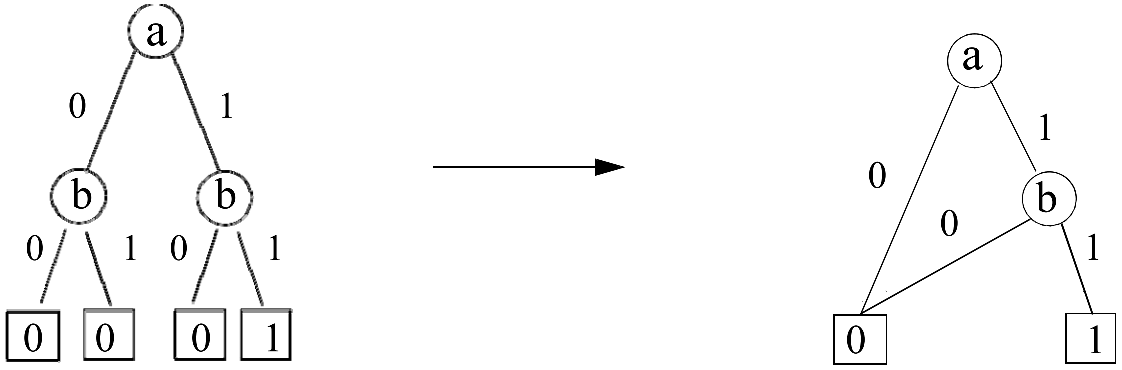
ROBDD

ROBDD Examples (con’t)

AND



out = f (a,b) = $a \wedge b$

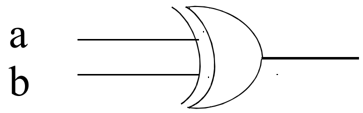


BDD

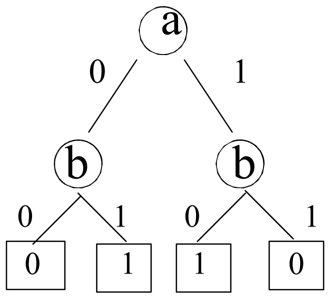
ROBDD

ROBDD Examples (con’t)

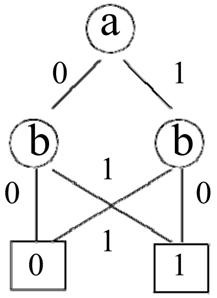
XOR



out = f(a,b) = a ⊕b



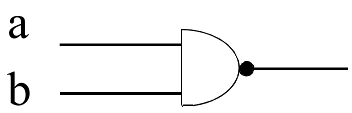
BDD



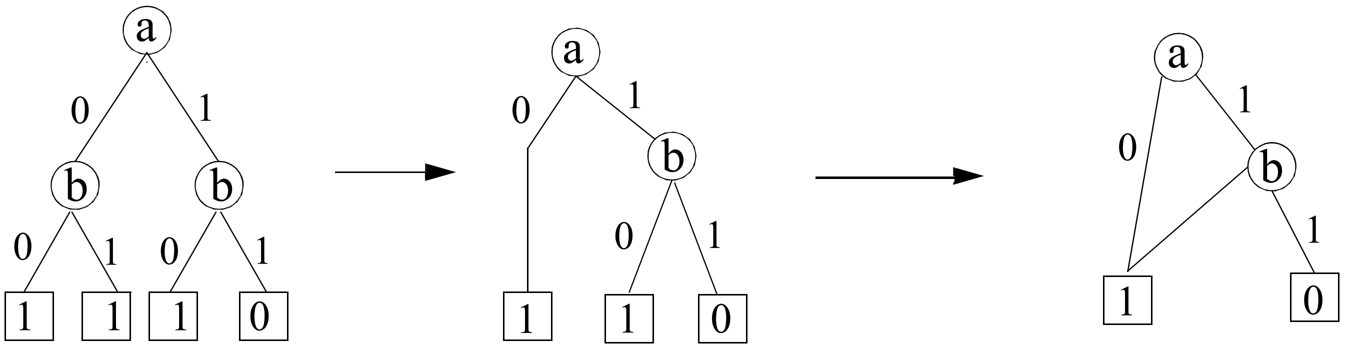
ROBDD

ROBDD Examples (con’t)

NAND



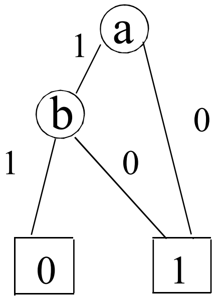
out = f(a,b) = ¬ (a ∧ b)



BDD

ROBDD

a	b	out
0	0	1
0	1	1
1	0	1
1	1	0

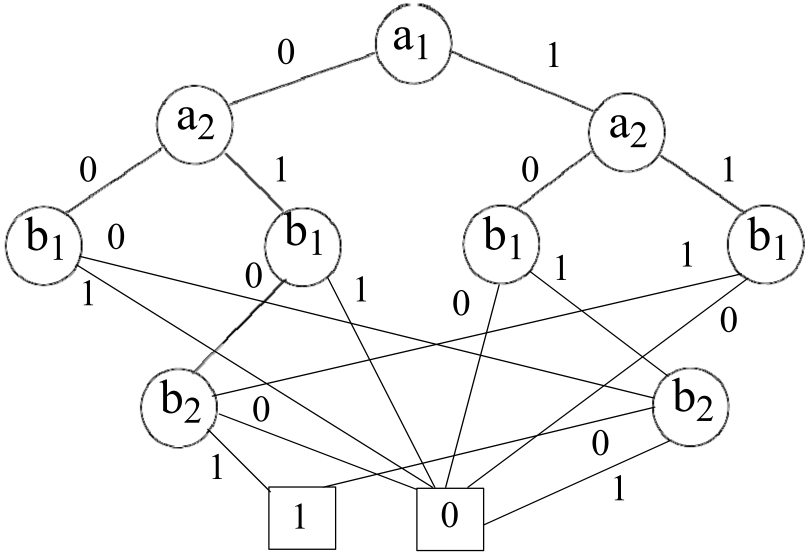


red
T.T.

a	b	out
0	-	1
-	0	1
1	1	0

Variable Ordering Problem

- The size of an ROBDD depends critically on the variable order
- For order $\mathbf{a_1 < a_2 < b_1 < b_2}$, the 2-bit comparator $f(a_1,a_2,b_1,b_2) = (a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2)$ ROBDD becomes:



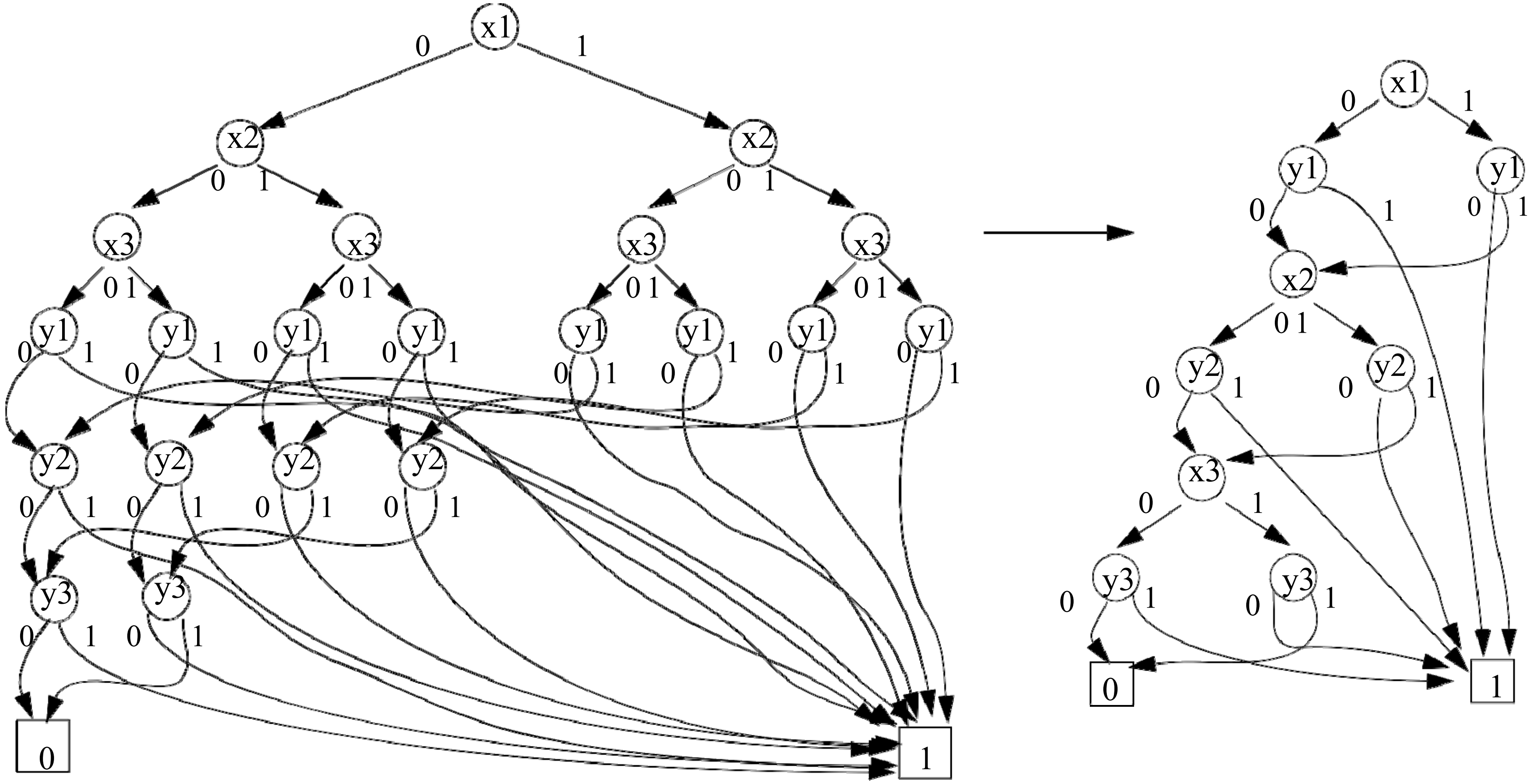
- For an n-bit comparator:

  $a_1 < b_1 < ... < a_n < b_n$ gives 3n+2 vertices (linear complexity)

  $a_1 < ... < a_n < b_1 ... < b_n$, gives $3\text{x}2^n - 1$ vertices (exponential complexity!)

Variable Ordering Problem - Example

$(x1 \oplus y1) \vee (x2 \oplus y2) \vee (x3 \oplus y3)$



Variable Ordering Problem (cont’d)

- The problem of finding the *optimal* variable order is NP-complete
- Some Boolean functions have exponential size ROBDDs for any order (e.g., multiplier)

Heuristics for Variable Ordering

- Heuristics developed for finding a *good* variable order (if it exists)
- Intuition for these heuristics comes from the observation that ROBDDs tend to be smaller when related variables are close together in the order (e.g., ripple-carry adder)
- Variables appearing in a subcircuit are related: they determine the subcircuit’s output

⇒ should usually be close together in the order

Dynamic Variable Ordering

- Useful if no obvious static ordering heuristic applies
- During verification operations (e.g., reachability analysis) functions change, hence initial order is not good later on
- Good ROBDD packages periodically internally reorder variables to reduce ROBDD size
- Basic approach based on neighboring variable exchange ... < a < b < ... ⇒ ...< b < a < ...
  Among a number of trials the best is taken, and the exchange is repeated

Logic Operations on ROBDDs

- Residual function (cofactor): $b \in \{0, 1\}$

$$f\big|_{x_i \leftarrow b}(x_1,...,x_n) = f(x_1, ...,x_{i-1}, b, x_{i+1}, ..., x_n)$$

- ROBDD of $f\big|_{x_i \leftarrow b}$ computed by a depth-first traversal of the ROBDD of **f**:

  For any vertex v which has a pointer to a vertex w such that *var(w)* = x_i, replace the pointer by *low(w)* if b is 0 and by *high(w)* if b is 1.

  If not in canonical form, apply *Reduce* to obtain ROBDD of $f\big|_{x_i \leftarrow b}$.

- All 16 two-argument logic operations on Boolean function implemented efficiently on ROBDDs in linear time in the size of the argument ROBDDs.

Logic Operations on ROBDDs (cont'd)

- Based on Shannon's expansion

$$f = [\neg x \wedge f|_{x \leftarrow 0}] \vee [x \wedge f|_{x \leftarrow 1}]$$

- Bryant (1986) gave a uniform algorithm, ***Apply***, for computing all 16 operations:

  $f * f'$: an arbitrary logic operation on Boolean functions f and f'

  v and v': the roots of the ROBDDs for f and f', $x = var(v)$ and $x' = var(v')$

- Consider several cases depending on v and v'

  (1) v and *v'* are both *terminal* vertices: $f * f' = value(v) * value(v')$

  (2) $x = x'$: use Shannon's expansion

$$f * f' = [\neg x \wedge (f|_{x \leftarrow 0} * f'|_{x \leftarrow 0})] \vee [x \wedge (f|_{x \leftarrow 1} * f'|_{x \leftarrow 1})]$$

  to break the problem into two subproblems, each is solved recursively

  The root is *v* with $var(v) = x$

  Low(v) is $(f|_{x \leftarrow 0} * f'|_{x \leftarrow 0})$

  High(v) is $(f|_{x \leftarrow 1} * f'|_{x \leftarrow 1})$

Logic Operations on ROBDDs (cont'd)

(3) $x < x'$: $f'|_{x \leftarrow 0} = f'|_{x \leftarrow 1} = f'$ since f' does not depend on x

In this case the Shannon's expansion simplifies to

$$f * f' = [\neg x \wedge (f|_{x \leftarrow 0} * f')] \vee [x \wedge (f|_{x \leftarrow 1} * f')]$$, similar to (2)

and compute subproblems recursively,

(4) $x' < x$: similar to the case above

Improvement using the *if-then-else* (ITE) operator:

$\text{ITE}(F, G, H) = F . G + F' . H$ where F, G and H are functions

Recursive algorithm based on the following, *v* is the top variable (lowest index):

$$\begin{aligned} \text{ITE}(F, G, H) &= v.(F.G + F'.H)_v + v'.(F.G + F'.H)_{v'} \\ &= v.(F_v.G_v + F'_v.H_v) + v'.(F_{v'}.G_{v'} + F'_{v'}.H_{v'}) \\ &= (v, \text{ITE}(F_v, G_v, H_v), \text{ITE}(F_{v'}, G_{v'}, H_{v'})) \end{aligned}$$

With terminal cases being: F = ITE(1, F, G) = ITE(0, G, F) = ITE(F, 1, 0) = ITE(G, F, F)

we define NOT(F) = ITE(F, 0, 1) AND(F, G) = ITE(F, G, 0)

OR(F, G) = ITE(F, 1, G) XOR(F, G) = ITE(F, ¬G, G)

LEQ(F, G) = ITE(F, G, 1) etc.

Logic Operations on ROBDDs (cont'd)

- By using ***dynamic programming***, it is possible to make the ITE algorithm polynomial:

  (1) The result must be reduced to ensure that it is in canonical form;

  - record constructed nodes (*unique table*);
  - before creating a new node, check if it already exists in this *unique* hash table

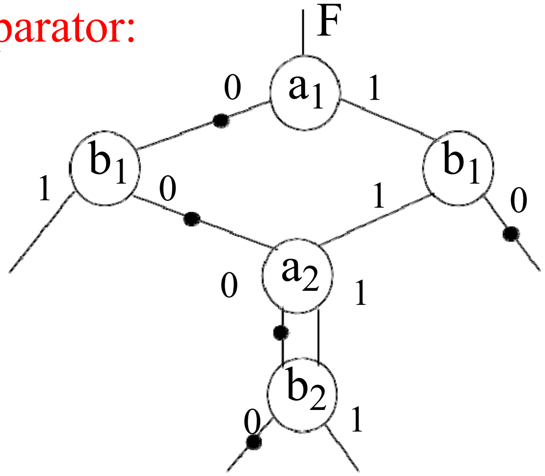
  (2) Record all previously computed functions in a hash table (*computed table*);

  - must be implemented efficiently as it may grow very quickly in size;
  - before computing any function, check table for solution already obtained

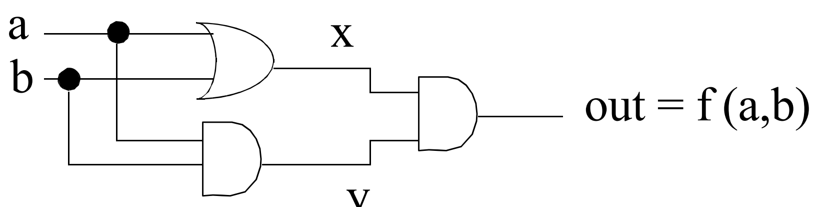
- **Complement edges** can reduce the size of an ROBDD by a factor of 2
  - Only one terminal node is labeled 1
  - Edges have an attribute (dot) to indicate if they are inverting or not
  - To maintain canonicity, a dot can appear only on *low(v)* edges
  - Complementation achieved in O(1) time by placing a dot on the function edge
  - F and F' can share entry in *computed table*
  - Adaptation of ITE easy

- Test for F ≤ G can be computed by a specialized ITE_CONSTANT algorithm

Comparator:



BDD Operators - Example

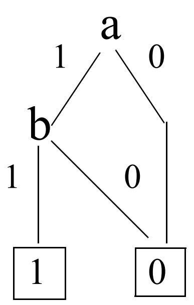


Task: compute ROBDD for f (a,b)

1) $f = x \wedge y = (a \vee b) \wedge (a \wedge b)$ order a,b.

a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

a	b	out
0	-	0
-	0	0
1	1	1



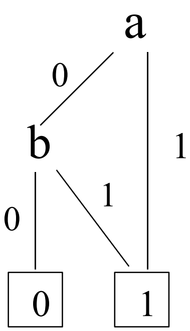
BDD Operators - Examples (con’t)

2) $f = x \wedge y$

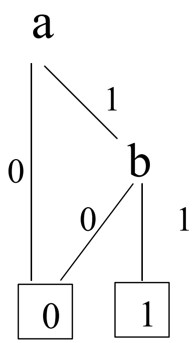
$\mathrm{BDD}_f =$ “$\mathrm{BDD}_x \wedge \mathrm{BDD}_y$”

$= \mathrm{Conj}\ (\mathrm{BDD}_x, \mathrm{BDD}_y)$

BDD_x



BDD_y



$x \wedge 0 = 0$

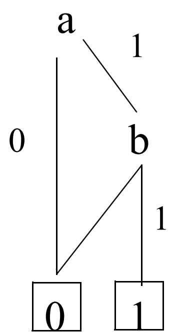
$x \wedge 1 = x$

$a = 1: \quad 1 \quad \wedge \quad b \quad = \quad b$

$a = 0: \quad b \quad \wedge \quad 0 \quad = \quad 0$

$b = 1: \quad 1 \quad \wedge \quad 1 \quad = \quad 1$

$b = 0: \quad 0 \quad \wedge \quad 0 \quad = \quad 0$



Other Decision Diagrams

- Multiterminal BDD (MTBDD): Pseudo-Boolean functions $B^n \rightarrow N$, terminal nodes are integers
- Binary Moment Diagrams (BMD): for representing and verifying arithmetic operations, word-level representation
- Ordered Kronecker Functional BDDs (OKFBDD): Based on XOR operations and OBDD
- Free BDDs (FBDD): Different variable order along different paths in the graph
- Zero suppressed BDDs (ZBDD)
- Combination of various forms of DDs integrated in DD software packages: Drechsler *et al* (U. Freiburg, Germany), Clarke *et al* (Carnegie Mellon U., USA)
- Extension to represent systems of linear and Boolean constraints (DTU)
- Multiway Decision Diagrams (MDG): Representation for a subset of equational first-order logic for modeling state machines with abstract and concrete data (U. of Montreal)

Well known ROBDD packages:

- CMU (as used in SMV from Carnegie Mellon U.)
- CUDD, U. of Colorado at Boulder (as used in VIS from UC at Berkeley)
- Industrial packages: Intel, Lucent, Cadence, Synopsys, Bull Systems, etc.

Applications of ROBDDs

ROBDD:

- Construction DD from circuit description:
  - Depth-first vs. breadth-first construction (keep only few levels in memory, rest on disk; problem with dynamic reordering)
  - Partitioning of Boolean space, each partition represented by a separate graph
  - Bottom-up vs. top-down, introducing decomposition points
- Internal correspondences in the two circuits — equivalent functions, or complex relations

ATPG-based:

- Combine circuits with an XOR gate on the outputs, show inexistence of test for a fault s-a-0 on the output (i.e., the output would have to be driven to 1 meaning that there is a difference in the two circuits)
- Use ATPG and learning to determine equivalent circuit nodes

Fast random simulation:

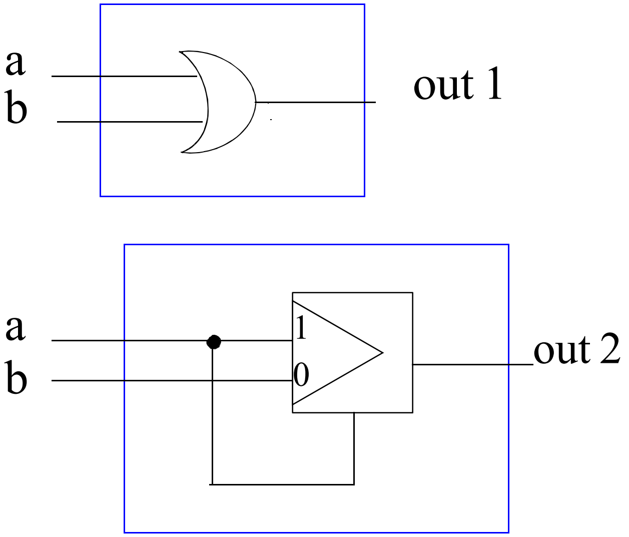
- Detect quickly easy differences

Real tools:

- Use a combination of techniques, fast and less powerful first, slow but exact later

Combinational Equivalence Chequing - Example

Two circuits C1 and C2

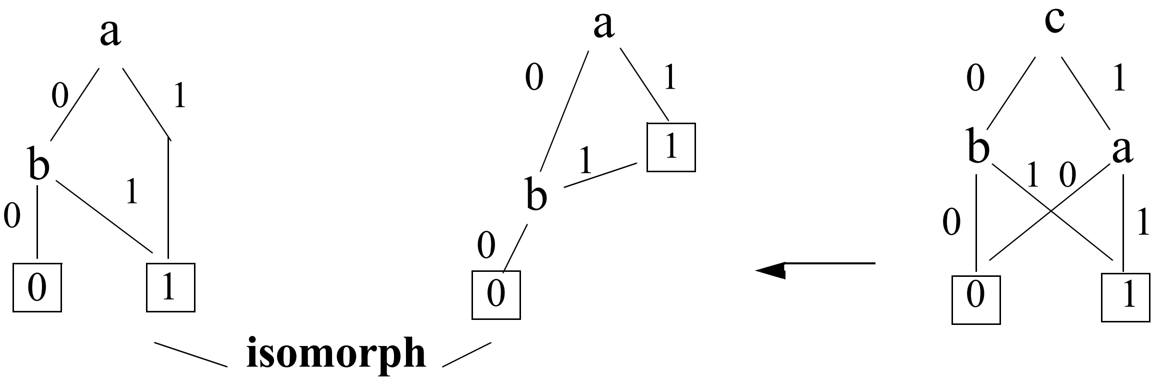


C1

= ?

C2

C1: a ∨ b C2: if a then a else b MUX: if c then a else b



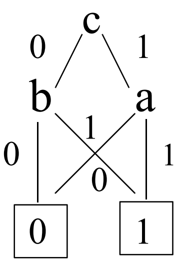
Combinational Equivalence Checking – Multiplexor Example

Specification: if c = 1 then out = a
else out = b

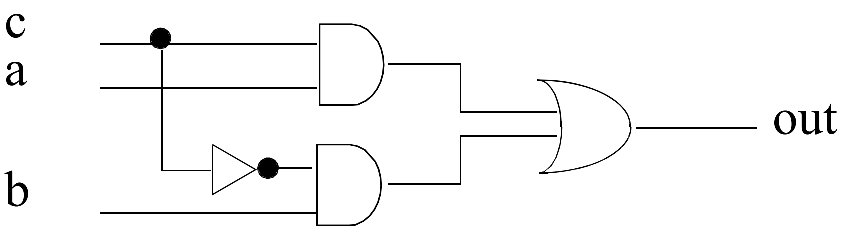
Build ROBDD for Spec:

c	a	b	out
1	1	-	1
1	0	-	0
0	-	1	1
0	-	0	0

ROBDD1:
order: c, a, b



Implementation:

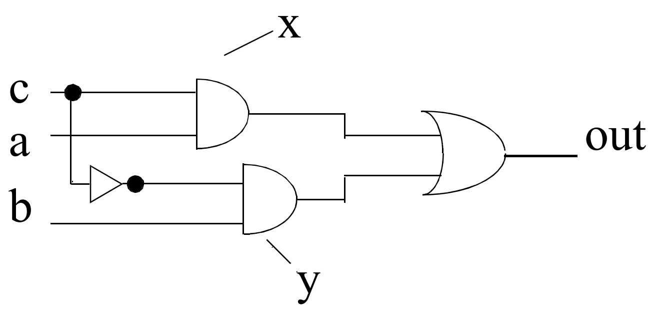


c	a	b	out
1	1	0	1
1	1	1	1
1	0	0	0
1	0	1	0
0	1	0	0
0	1	1	1
0	0	0	0
0	0	1	1

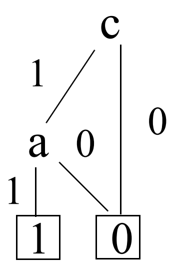
$\text{out} = (c \wedge a) \vee (\neg c \wedge b)$

Multiplexor Example (con’t)

Build ROBDD for Imp:

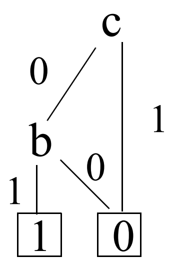


x: c ∧ a

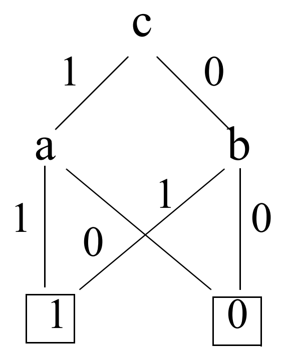


disjunction (or)

y: ¬c ∧ b



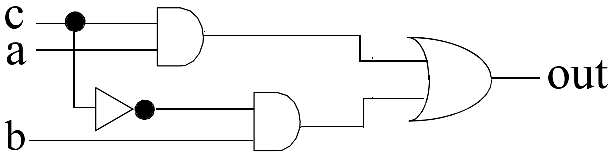
ROBDD2
order: c, a, b



isomorph to ROBDD1 !

Multiplexor Example (con't)

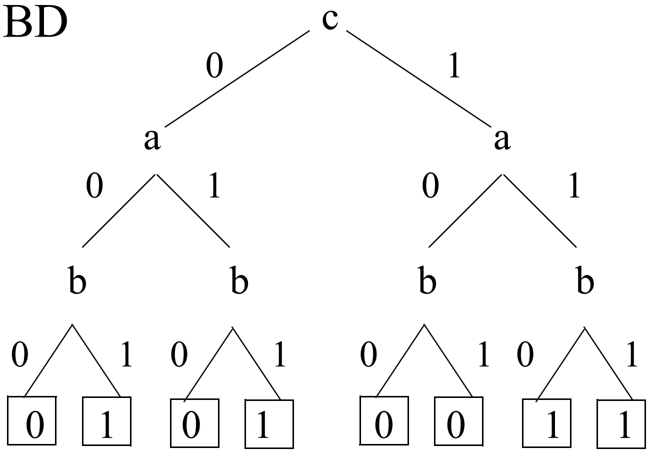
Alternative way to build ROBDD2:



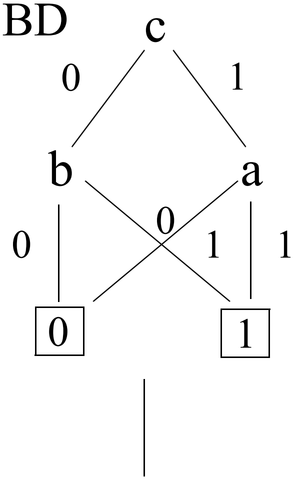
$\text{out} = (c \wedge a) \vee (\neg c \wedge b)$

order: c, a, b

BBD

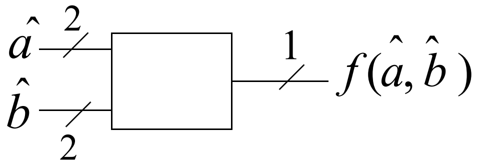


ROBBD

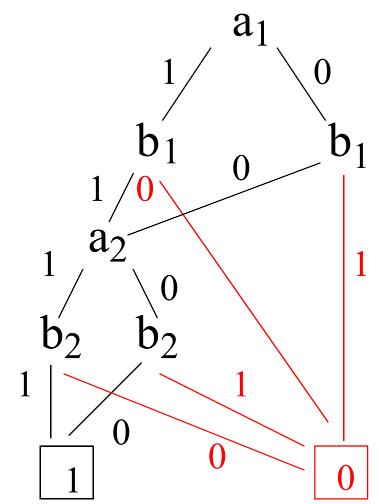


isomorph to ROBDD1

Comparator Example



Spec: $f(\hat{a}, \hat{b}) = 1$ if $\hat{a} = \hat{b}$

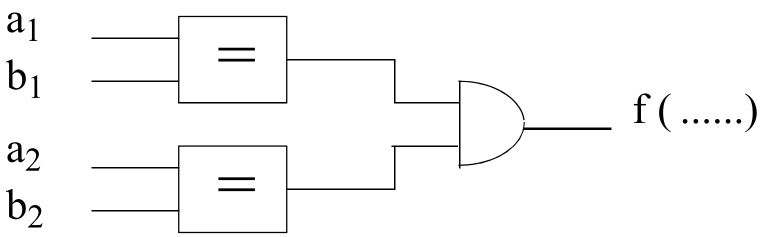


<u>Refinement:</u> $f(a_1, a_2, b_1, b_2) = 1$ if $(a_1 = b_1) \wedge (a_2 = b_2)$

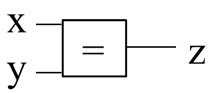
$$\hat{a} = a_1 a_2$$

$$\hat{b} = b_1 b_2$$

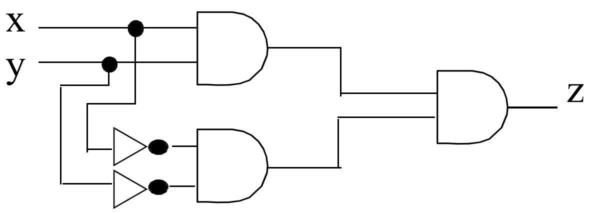
<u>Implicit:</u>

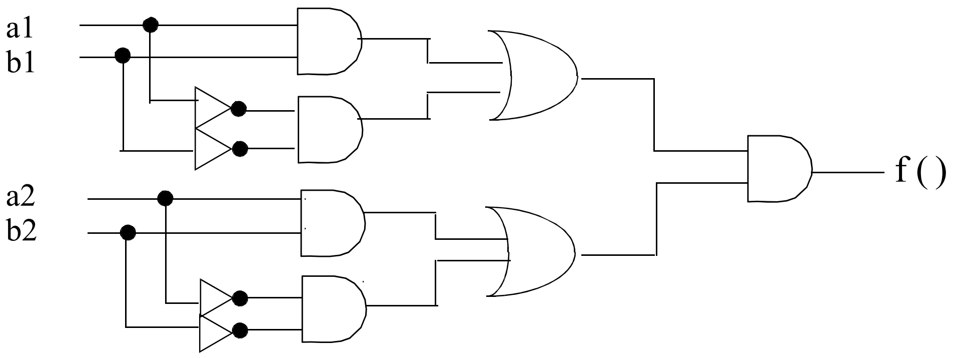


Comparator Example (cont’d)



: $z = (x \wedge y) \vee (\neg x \wedge \neg y)$





T1 **T2**

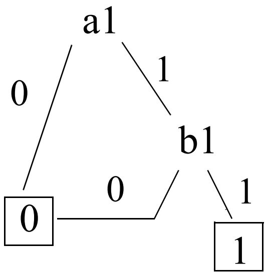
$$f = \quad [(a1 \wedge b1) \vee (\neg a1 \wedge \neg b1)] \wedge$$

$$[(a2 \wedge b2) \vee (\neg a2 \wedge \neg b2)]$$

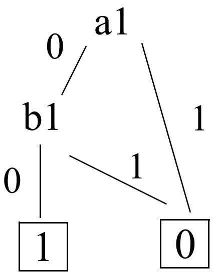
T4 **T3**

Comparator Example (cont’d)

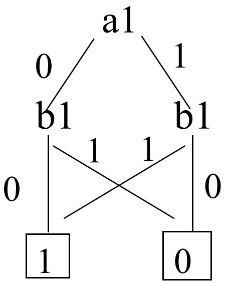
T1:



T2:



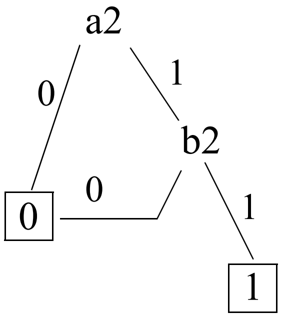
<u>disj:</u> T1, T2, T12



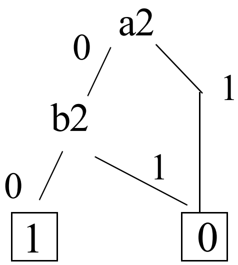
$x \vee 0 = x$

$x \vee 1 = 1$

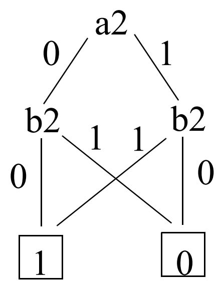
T3:



T4:



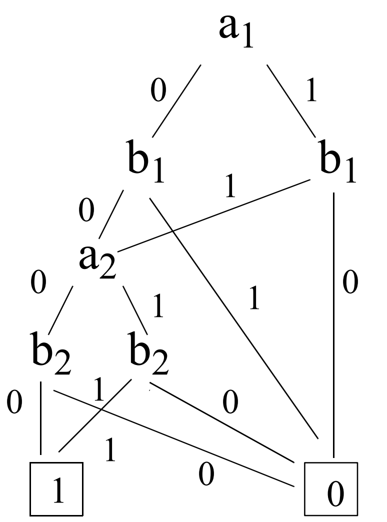
<u>disj:</u> T3, T4, T34



Comparator Example (cont'd)

conj. T_{12}, T_{34}, a_1, a_2 b_1, b_2) independent

order: a_1, b_1, a_2, b_2



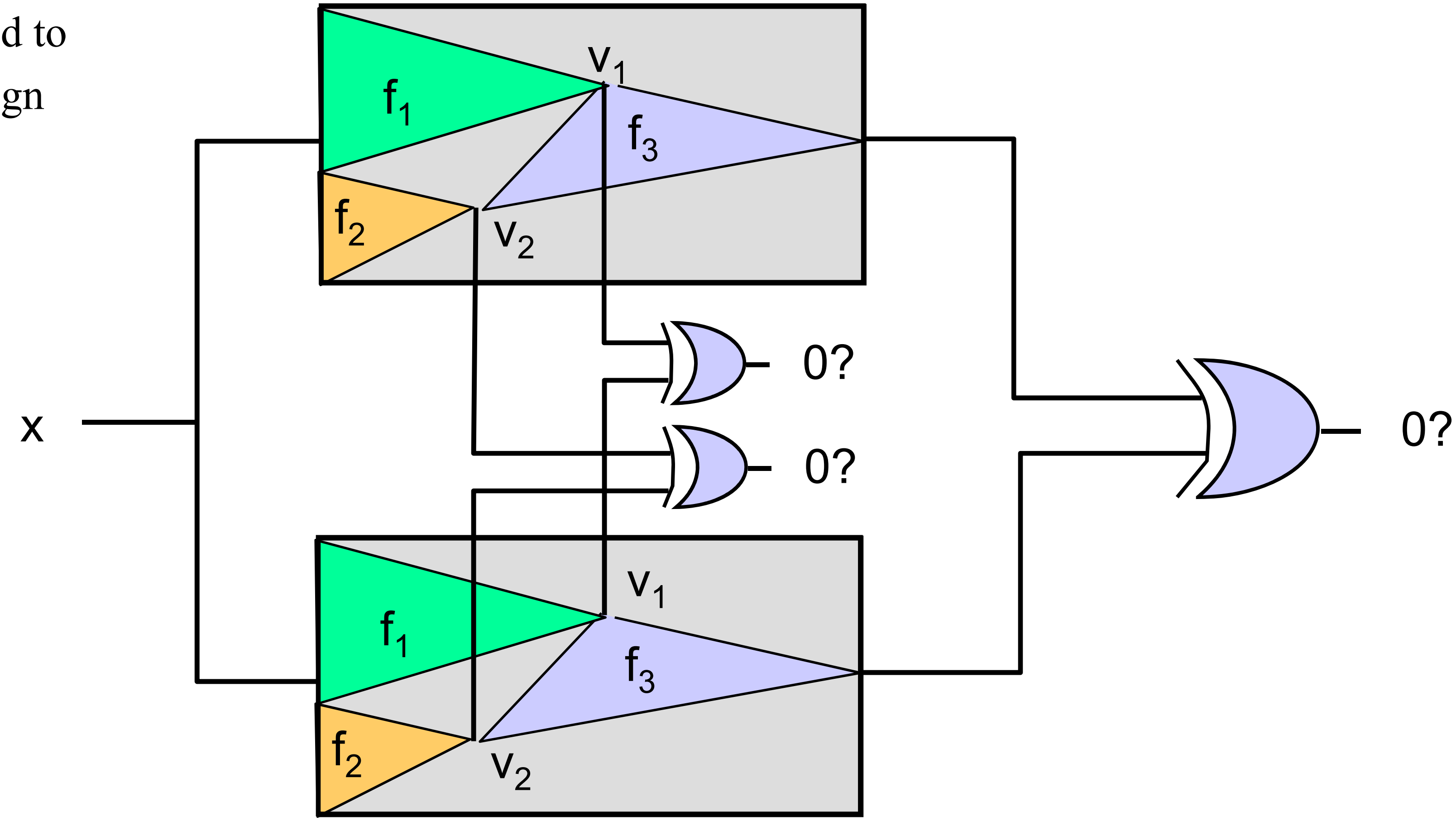
Isomorph to the spec

Equivalence Checking in Practice

- Usually, combinational circuits implement arithmetic and logic operations, and next-state and output functions of finite-state machines (sequential circuits)
- Verifying the behavior of the gate-level implementation against the RTL design of digital systems can often be reduced to verifying the combinational circuits
  - Equivalence comparison between the next-state and output functions (combinational circuits)
  - Requires that both have the same state space (and of course inputs and outputs), knowing the mapping between states helps...
  - Can also be used to verify gate-level implementation against gate-level model extracted from layout
  - This kind of verification is useful for confirming the correctness of manual changes or synthesis tools
- If the state space is not the same, sequential (behavioral equivalence) of FSM must be considered ...

Cutpoint-based Equivalence Checking

Cutpoints are used to partition the Design



Cutpoint guessing:

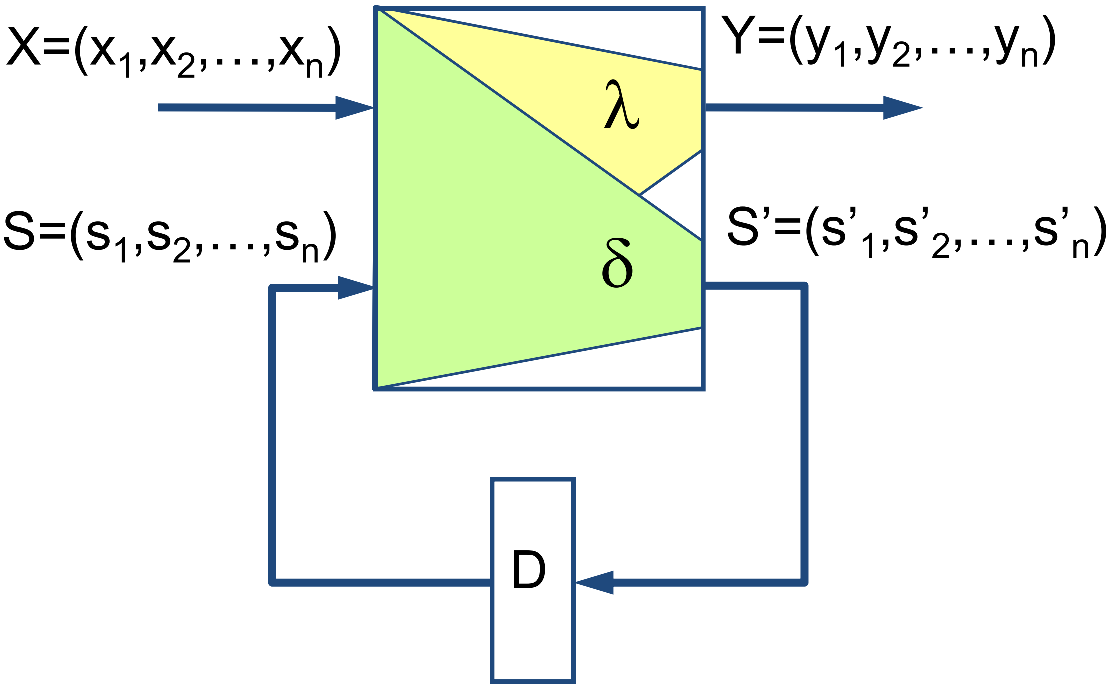
- Compute net signature with random simulator
- Sort signatures + select cutpoints
- Iteratively verify and refine cutpoints
- Verify outputs

Sequential Equivalence Checking

- If combinational verification paradigm fails (e.g. we have no name matching)

- Two options:
  - Try to **match registers** automatically
    - functional register correspondence
    - structural register correspondence
  - Run **full sequential verification** based on state traversal
    - very expensive but most general

Basic Model Finite State Machines



$M(X,Y,S,S_0,\delta,\lambda)$:

X: Inputs

Y: Outputs

S: Current State

S_0: Initial State(s)

δ: $X \times S \rightarrow S$ (next state function)

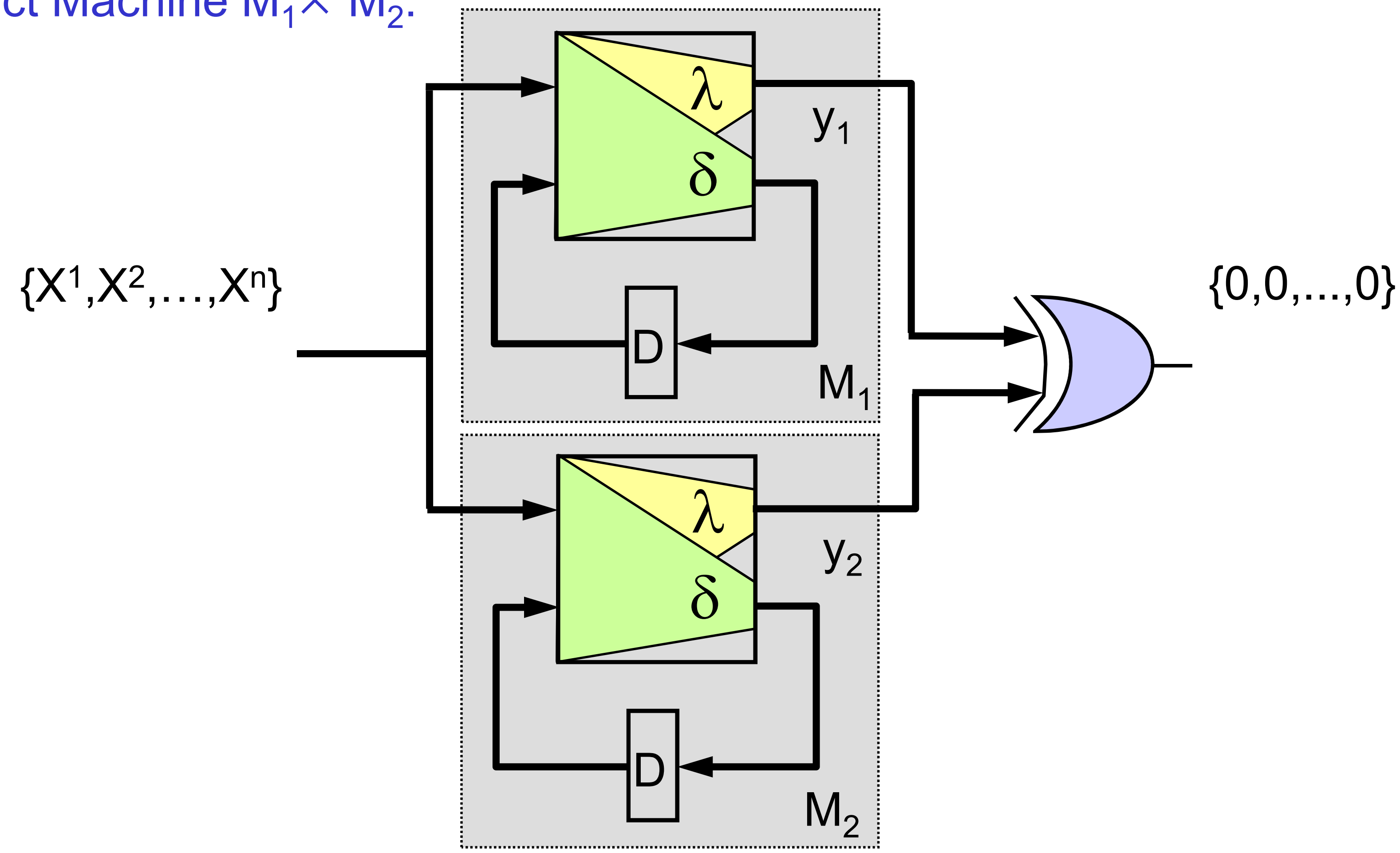
λ: $X \times S \rightarrow Y$ (output function)

Delay element:

- Clocked: synchronous
  - single-phase clock, multiple-phase clocks
- Unclocked: asynchronous

Finite State Machines Equivalence

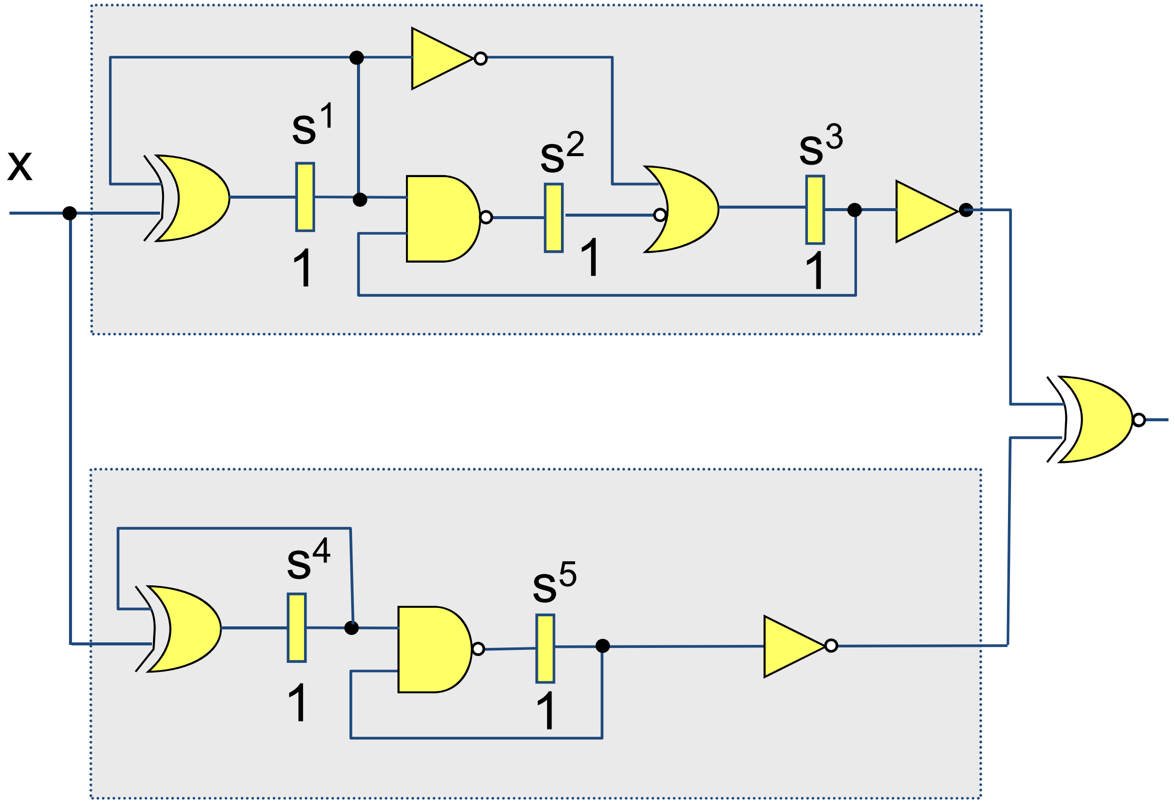
Build Product Machine $M_1 \times M_2$:



Definition:

M_1 and M_2 are functionally equivalent iff the product machine $M_1 \times M_2$ produces a constant 0 for all valid input sequences $\{X_1,\ldots,X_n\}$.

Illustrative Example



Product Machine:

$\{s^1, s^2, s^3\}$ U $\{s^4, s^5\}$

Transition Relations:

$(s^1)' = s^1 \oplus x$

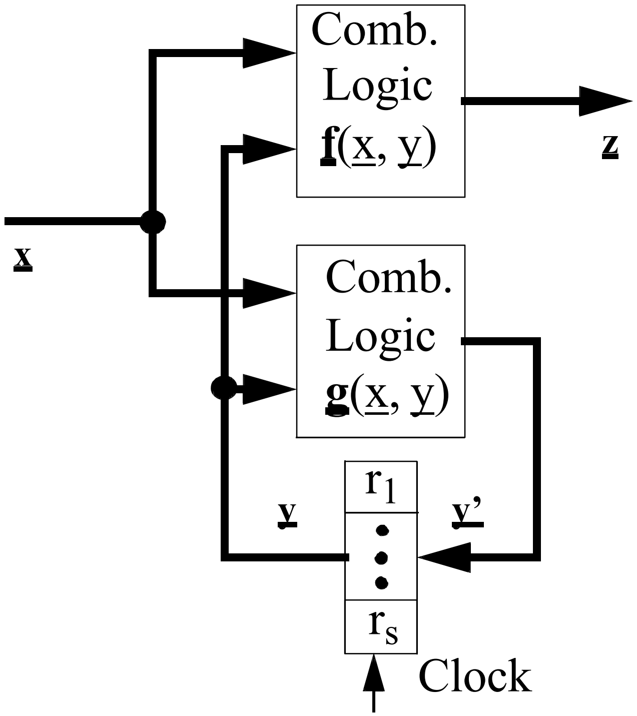
$(s^2)' = \neg\ (s^1 \wedge\ s^3)$

$(s^3)' = \neg s^1 \vee \neg s^2$

$(s^4)' = s^4 \oplus x$

$(s^5)' = \neg\ (s^4 \wedge s^5)$

Sequential Circuits and Finite State Machines


Comb.
Logic
$\underline{\mathbf{f}}(\underline{x}, \underline{y})$
$\underline{\mathbf{z}}$
$\underline{\mathbf{x}}$
Comb.
Logic
$\underline{\mathbf{g}}(\underline{x}, \underline{y})$
r_1
$\underline{\mathbf{y}}$
$\underline{\mathbf{y'}}$
r_s
Clock

$\underline{\mathbf{r}} = (r_1, ..., r_s)$ a vector of memory bits

— state variables, memorize encoded states

$\underline{\mathbf{y}} = (y_1, ..., y_s)$ a vector of present state values

$\underline{\mathbf{y'}} = (y'_1, ..., y'_s)$ a vector of next state values

$\underline{\mathbf{x}} = (x_1, ..., x_m)$ a vector of input bits

— encode input symbols

$\underline{\mathbf{z}} = (z_1, ..., z_n)$ a vector of output bits

— encode output symbols

$\underline{\mathbf{f}}$ = output function, $\underline{\mathbf{f}}(\underline{x}, \underline{y})$ = Mealy, $\underline{\mathbf{f}}(\underline{y})$ = Moore

$\underline{\mathbf{g}}$ = next-state function

Here we consider FSM synchronized on clock transitions — synchronous sequential circuits

- To verify the behavior of such circuits we need efficient representation for the manipulation of next-state and output functions and sets of states
- Using characteristic functions of relations and sets

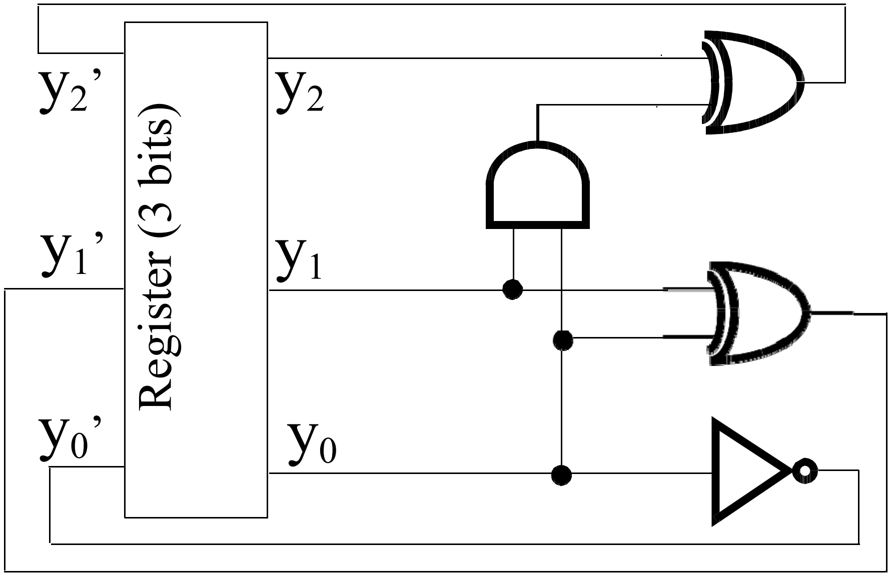
Relational Representation of FSM

Representation of Relations and Sets

- If R is n-ary relation over {0,1} then R can be represented by (the ROBDD of) its *characteristic function:* $f_R(v_1,\ldots,v_n) = 1$ iff $(v_1,\ldots,v_n) \in R$
  - Same technique can be used to represent sets of states
- Transition relation N of a sequential circuit is represented by its Boolean characteristic function over inputs and state variables:

$$N(\underline{\mathbf{x}}, y_1, \ldots, y_s, y_1', \ldots, y_s')$$

- **Example**: synchronous modulo 8 counter, $N(\underline{\mathbf{y}}, \underline{\mathbf{y}}') = N_0(\underline{\mathbf{y}}, y_0') \wedge N_1(\underline{\mathbf{y}}, y_1') \wedge N_2(\underline{\mathbf{y}}, y_2')$



Next state $\underline{\mathbf{y}}'$:

$y_0' = \neg y_0$

$y_1' = y_0 \oplus y_1$

$y_2' = (y_0 \wedge y_1) \oplus y_2$

Transition relation N(y,y'):

$N_0(y,y') = (y_0' \Leftrightarrow \neg y_0)$

$N_1(y,y') = (y_1' \Leftrightarrow y_0 \oplus y_1)$

$N_2(y,y') = (y_2' \Leftrightarrow (y_0 \wedge y_1) \oplus y_2)$

Relational Representation of FSM (cont'd)

Quantified Boolean Formulas (QBF)

- Needed to construct complex relations and manipulate FSMs
- $V=\{v_1,v_2,..., v_n\}$ = set of Boolean (propositional) variables
- QBF(V) is the smallest set of formulas such that
  - every variable in V is a formula
  - if f and g are formulas, then $\neg f$, $f \wedge g$, $f \vee g$ are formulas
  - if f is a formula and $v \in V$, then $\forall v.f$ and $\exists v.f$ are formulas
- A truth assignment for QBF(V) is a function $\sigma: V \rightarrow \{0,1\}$

  If $a \in \{0,1\}$, then $\sigma[v \leftarrow a]$ represents

  $$\sigma[v \leftarrow a](w) = a \text{ if } v = w$$

  $$\sigma[v \leftarrow a](w) = \sigma(w) \text{ if } v \neq w$$

- f is a formula in QBF(V) and σ is a truth assignment: $\sigma \models f$ if f is true under σ.

Relational Representation of FSM (cont’d)

Quantified Boolean Formulas (cont’d)

- QBF formulas have the same expressive power as ordinary propositional formulas; however, they may be more concise
- QBF Semantics: relation $\models$ is defined recursively:

  o $\models$ v iff $\sigma(v)=1$;

  o $\models$ $\neg f$ iff σ $|\neq f$;

  o $\models$ $f \vee g$ iff $\sigma \models f$ or $\sigma \models g$;

  o $\models$ $f \wedge g$ iff $\sigma \models f$ and $\sigma \models g$;

  $\sigma \models \exists v.f$ iff $\sigma[v \leftarrow 0] \models f$ or $\sigma[v \leftarrow 1] \models f$;

  $\sigma \models \forall v.f$ iff $\sigma[v \leftarrow 0] \models f$ and $\sigma[v \leftarrow 1] \models f$.

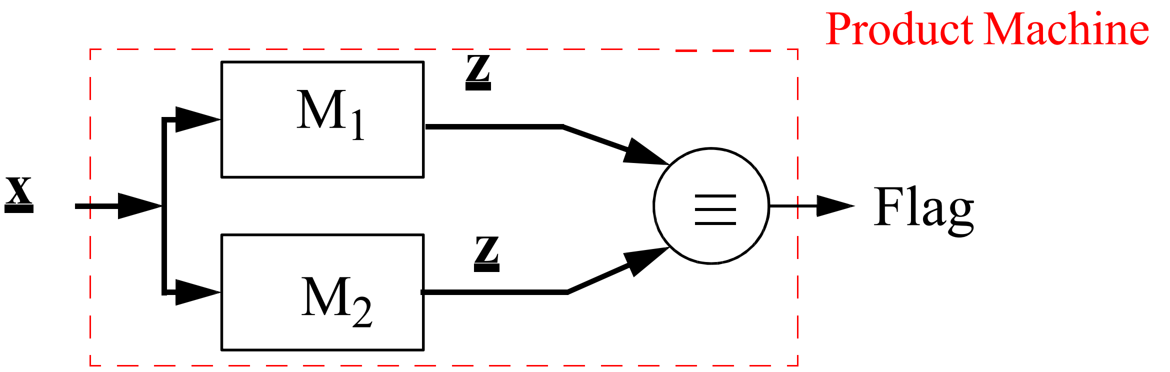
- Every QBF formula can represent an n-ary Boolean relation consisting of those truth assignments for the variables in V that makes the formula true: Boolean characteristic function of the relation
- $\exists x.\ f = f|_{x \leftarrow 0} \vee f|_{x \leftarrow 1}$, $\forall x.\ f = f|_{x \leftarrow 0} \wedge f|_{x \leftarrow 1}$

  In practice, special algorithms needed to handle quantifiers efficiently (e.g., on ROBDD)

Sequential Equivalence Checking

Basic Idea:

To prove the equivalence of two FSMs M_1 and M_2 (with the same input and output alphabet), a *product machine* is formed which tests the equality of outputs of the two individual machines in every state



M_1 and M_2 are equivalent iff the product machine produces Flag = *true* output in every state reachable from the initial state

- Coudert *et al.* were first to recognize the advantage of representing set of states with ROBDD's: Symbolic Breadth-First Search of the transition graph of the product machine
- Their technique was initially applied to checking machine equivalence and later extended by McMillan, et al. to symbolic model checking of temporal logic formulas (in CTL)

Relational Product of FSMs

Relational Products — implementation using ROBDD

- A typical task in verification: compute relational products with abstraction of variables:

  $\exists$ v.[f(v) $\wedge$ g(v)]

- Algorithm *RelProd* computes it in one pass over ROBDDs f(v) and g(v), instead of constructing f(v)$\wedge$g(v)
- *RelProd* uses a *computed table* (result cache), and is based on Shannon's expansion
- Entries in the cache have the form (f, g, E, h), where E is a set of variables that are existentially qualified out and f, g and h are (pointers to) ROBDDs
- If an entry indexed by f, g and E is in the cache, then a previous call to RelProd (f, g, E) has returned h, it is not recomputed
- Algorithm works well in practice, even if it has theoretical exponential complexity

Relational Representation of FSMs (cont'd)

Relational Product Algorithm

```
RelProd (f, g: ROBDD, E: set of variables)
if f=false $\vee$ g=false then return false
  else if f=true $\wedge$ g=true
          then return true
          else if (f, g, E, h) is cached
                  then return h
                  else    let x and y be the top variables of f and g, respectively
                          let z be the topmost of x and y,
                          h0:=RelProd($f|_{z=0}$, $g|_{z=0}$, E)
                          h1:=RelProd($f|_{z=1}$, $g|_{z=1}$, E)
                          if z $\in \Delta$E
                             then h:=Or(h0, h1) {ROBDD: h0$\vee$h1}
                             else h:=IFThenElse(z, h1, h0)
                  endif
insert (f, g, E, h) in cache
return h
endif
```

Reachability Analysis on FSMs

Computing Set of Reachable States

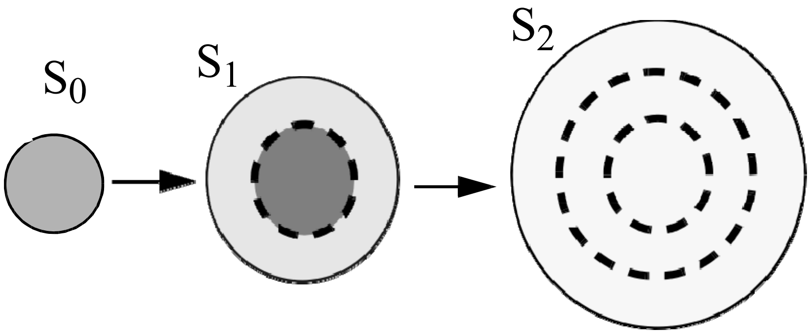
- Reachable state computation (state enumeration) is needed for FSM equivalence and model checking
- S_0 = a set of states, represented by the ROBDD $S_0(V)$

Find those states S_1 reachable in at most one transition from S_0:

$$S_1 = S_0 \cup \{\, s' \mid \exists s\, [s \in S_0 \wedge (s, s') \in N]\}$$

ROBDD's $S_0(\underline{\mathbf{y}})$ **and** $N(\underline{\mathbf{y}}, \underline{\mathbf{y}}')$, compute an ROBDD representing $S{:}_1$

$$S_1(\underline{\mathbf{y}}') = S_0(\underline{\mathbf{y}}') \vee \underset{yi \,\in\, \underline{\mathbf{y}}}{\exists y_i} [S_0(\underline{\mathbf{y}}) \wedge N(\underline{\mathbf{y}}, \underline{\mathbf{y}}')]$$



$$S_2 = S_0 \cup \{s' \mid \exists s\, [s \in S_1 \wedge (s, s') \in N\,]\}$$

$$S_2(\underline{\mathbf{y}}') = S_0(\underline{\mathbf{y}}') \vee \underset{yi \,\in\, \underline{\mathbf{y}}}{\exists y_i} [S_1(\underline{\mathbf{y}}) \wedge N(\underline{\mathbf{y}}, \underline{\mathbf{y}}')]$$

Reachability Analysis on FSMs (cont'd)

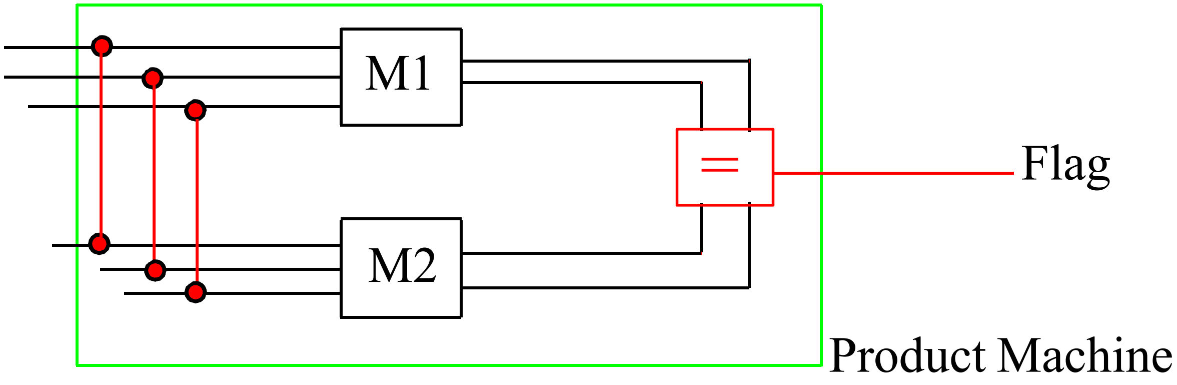
Reachability Analysis (cont'd)

- In general, the states reachable in at most k+1 steps are represented by:

$$\boxed{S_{k+1}(\underline{\mathbf{y}}') = S_0(\underline{\mathbf{y}}') \vee \underset{yi \,\in\, \underline{\mathbf{y}}}{\exists y_i} [\, S_k(\underline{\mathbf{y}}) \wedge N(\underline{\mathbf{y}}.\underline{\mathbf{y}}')]}$$

- As each set of states is a superset of the previous one, and the total number of states is finite, at some point, we must have $S_{k+1} = S_k$, $k \leq 2^s$ the number of states
- Reachability computation can be viewed as finding "least fixpoint"
- What about inputs $\underline{\mathbf{x}}$? Existentially quantify them out in the relational product (equivalent to closing the system with a non-deterministic source of values for $\underline{\mathbf{x}}$)

BDD Encoding



Basic idea:

1) connect both machines to equality check of outputs

2) compute set of reachable states

2a) representing set of states using ROBDD

2b) computing “images” of BDDs of all next states (using transition relations)

2c) reachability iteration (using images starting from one initial state until sequence converges)

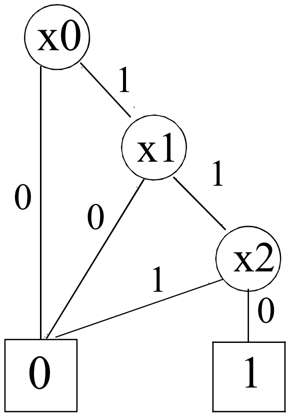
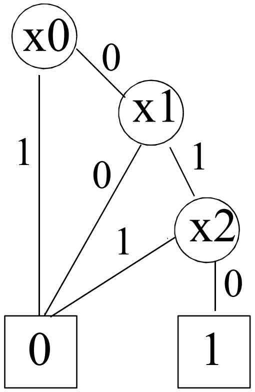
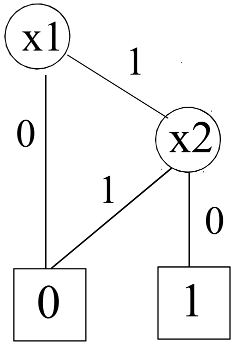
$$R_0 = \mathit{initial\ BDD}$$

.....

$$R_{i+1} = R_i \vee \mathit{Image}(R_i) \rightarrow \mathit{convergence};$$

ROBDD Encoding (cont'd)

Representing set of states using ROBDDs

Set	$\{110\}$	$\{010\}$	$\{110, 010\}$
Formula	$x0 \wedge x1 \wedge \neg x2$	$\neg x0 \wedge x1 \wedge \neg x2$	$(x0 \wedge x1 \wedge \neg x2) \vee (\neg x0 \wedge x1 \wedge \neg x2)$ $= \; x1 \wedge \neg x2$
ROBDD			

ROBDD Encoding (cont'd)

Representing set of states using ROBDDs

Set

$\{100, 101, 110, 111, 010, 011\}$

↓

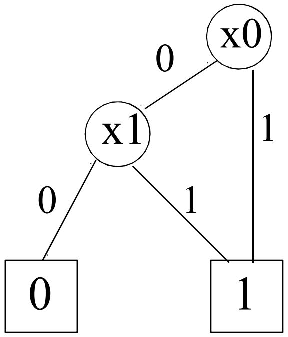
Formula

$$(x0 \wedge \neg x1 \wedge \neg x2) \vee (x0 \wedge \neg x1 \wedge x2) \vee (x0 \wedge x1 \wedge \neg x2)$$
$$\vee (x0 \wedge x1 \wedge x2) \vee (\neg x0 \wedge x1 \wedge \neg x2) \vee (\neg x0 \wedge x1 \wedge x2)$$

$$= x0 \vee x1$$

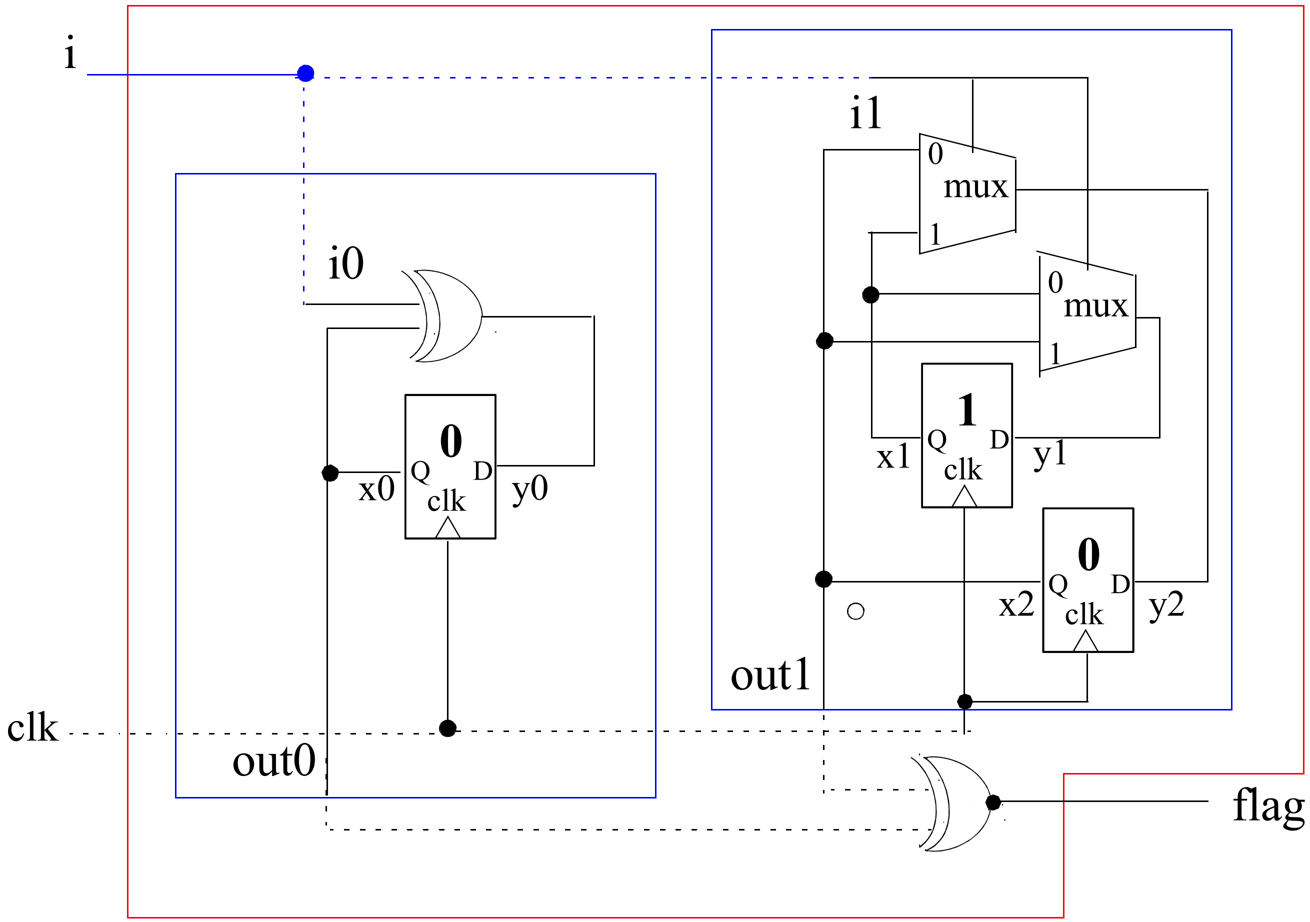
↓

ROBDD



Sequential Equivalence Checking Example

1) Connect both machines to equality check of outputs



Sequential Equivalence Checking Example (con't)

2a) Representing set of states using ROBDD

Initial State: $x0 = 0$
$x1 = 1$
$x2 = 0$

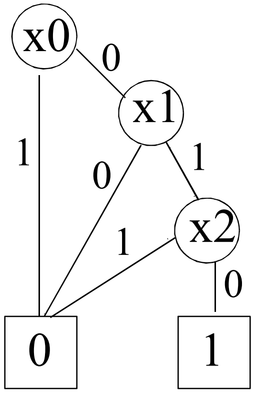
$\underline{\{0\ 1\ 0\}}$

set

$\underline{\neg x0 \wedge x1 \wedge \neg x2}$

formula

ROBDD



Sequential Equivalence Checking Example (cont’d)

2b) Compute images of set {0 1 0}

Tansition Relation

$y0 = x0 \oplus i$ —— $(x0{=}0)$ $= 0$ (i = 0) $= 1$ (i = 1)

$y1 = (\neg i \wedge x1) \vee (i \wedge x2)$ —— $\binom{x1=1}{x2=0}$ $= 1$ (i = 0) $= 0$ (i = 1)

$y2 = (\neg i \wedge x2) \vee (i \wedge x1)$ —— $\binom{x1=1}{x2=0}$ $= 0$ (i = 0) $= 1$ (i = 1)

	i = 0		i = 1	
x0: 0		0		1
x1: 1		1		0
x2: 0		0		1

{010 , 101}

↓

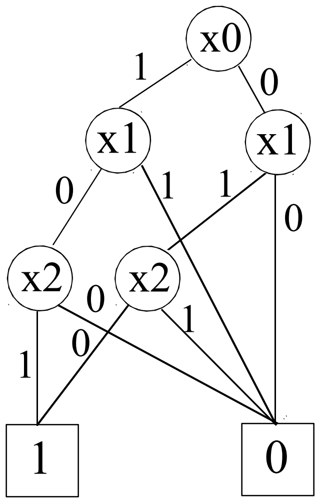
ROBDD

image: $(\neg x0 \wedge x1 \wedge \neg x2) \vee (x0 \wedge \neg x1 \wedge x2)$

BDD1 i = 0 i = 1 BDD2

$(BDD_1 \vee BDD_0)$

disj. (BDD1, BDD2)



Example (cont'd)

2c) Reachability iteration

$$R_0 = \neg x0 \wedge x1 \wedge \neg x2$$

$$R_1 = (\neg x0 \wedge x1 \wedge \neg x2) \vee (x0 \wedge \neg x1 \wedge x2) \vee R_0 = 1$$

$$R_2 = 1 \vee R_0 = 1$$

$$\rightarrow R_2 = R_1$$

In terms of sets:

R0 = {010}

R1 = {010, 101}

R2 = {010,101}

→ **R2** = **R1**

⇒ Converged

⇒ all states reached!

Equivalence Checking Tools

Commercial tools:

- Chrysalis: Design Verifier
- **Synopsys: Formality**
- **Cadence: Conformal**
- Verysys: Tornado
- **AHL: ChekOff-E**

Application:

- Used to prove equivalence of two sequential circuits that have the same state variables (or at least the same state space and a known mapping between states) by verifying that they have the same next-state and output functions
- Used in place of gate vs. RTL verification by simulation

Recommendations:

- Use modular design, relatively small modules, 10k - 20k gates
- Maintain hierarchy during synthesis (not flattening) and before layout: equivalence can be proven hierarchically much faster, especially for arithmetic circuits

Equivalence Checking Tools (cont'd)

CheckOff-E

- Commercial product by Abstract Hardware Ltd. (UK) and Siemens AG (Germany)
- Performs behavioral comparison of two Finite State Machines
- Input EDIF netlist + library or **VHDL**
- VHDL subset (superset of synthesizable synchronous VHDL)
  - no real time clauses (*after, wait for*), no conditional loop statements
- Interprets VHDL simulation semantics to build a *Micro FSM*
- Converts to *Macro FSM* by merging transition until stabilization at each time t
- Macro FSM is starting point for any verification; representation in ROBDD
- **Product discontinued!**

References

1. V. Sperschneider, G. Antoniou. *Logic: A Foundation for Computer Science*. Addison-Wesley, 1991.
2. S. Reeves, M. Clarke. *Logic for Computer Science*. Addison-Wesley, 1991.
3. Alan J. Hu, Formal Hardware Verification with BDDs: An Introduction, *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pp.677-682, 1997.
4. J. Jain, A. Narayan, M. Fujita, and A. Sangiovanni-Vincentelli, Formal Verification of Combinational Circuits, *VLSI Design*, 1997.
5. R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8), pp. 677-691, August 1986.
6. R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3), 1992, pp. 293-318.
7. R.E. Bryant. Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. *International Conference on Computer-Aided Design*, pp. 236-243, 1995.
8. S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, 1996.
9. M. Sheeran, G. Stålmarck. *A tutorial on Stålmarck's proof procedure for propositional logic. Formal Methods in Systems Design*, Kluwer, 1999.
10. O. Coudert and J.C. Madre, A Unified Framework for the Formal Verification of Sequential Circuits, *Int. Conference on Computer-Aided Design*, pp. 126-129, 1990.
11. H. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli, Implicit State Enumeration of Finite State Machines Using BDD's, *Int. Conference on Computer-Aided Design*, pp. 130-133, 1990.