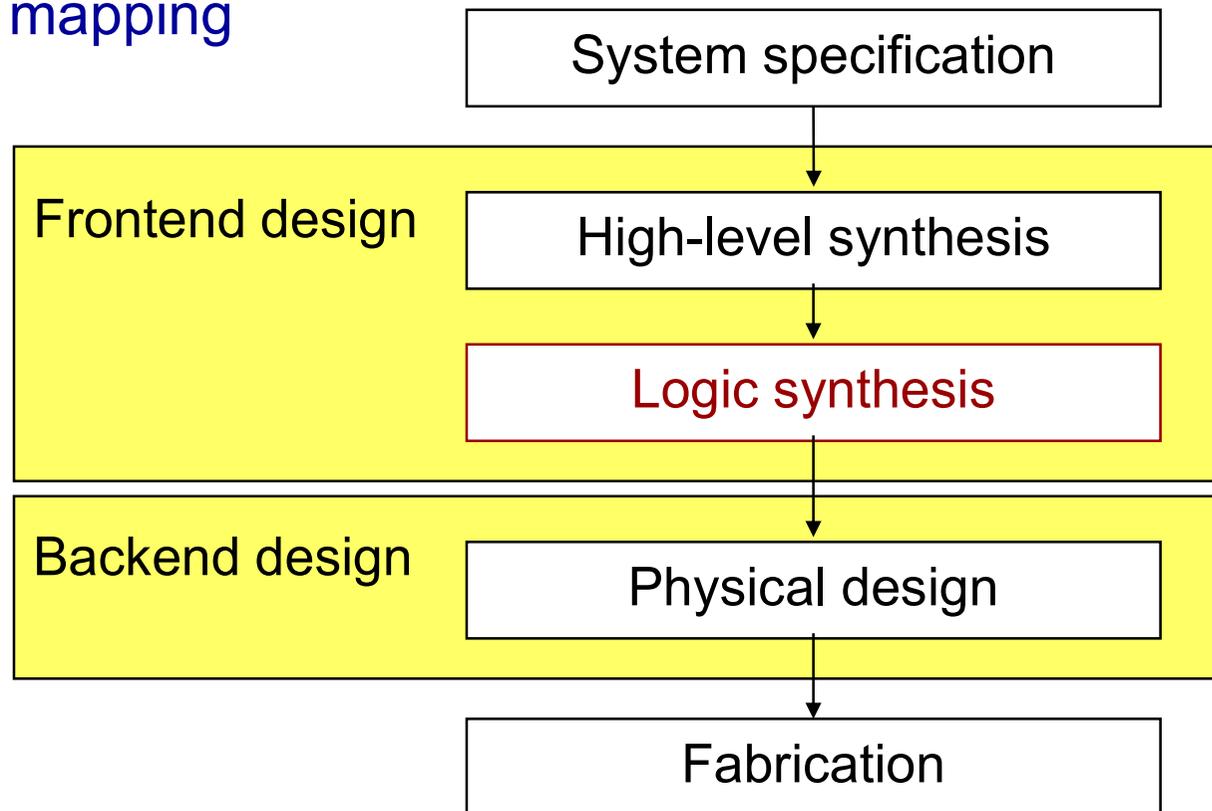
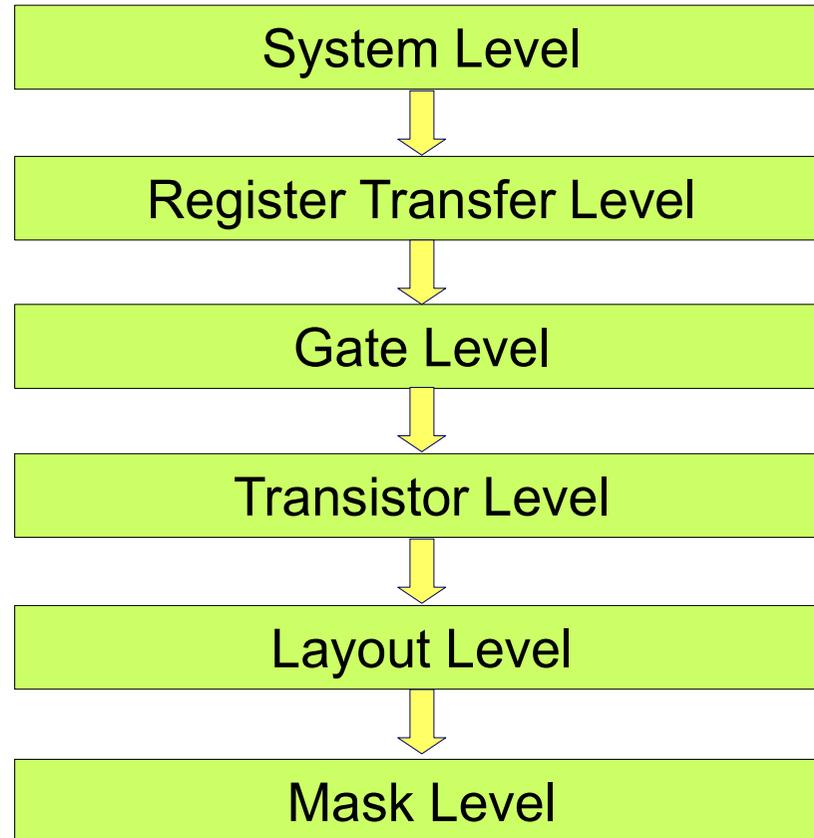


Unit 2: Logic Synthesis and Verification

- Course contents
 - Logic synthesis basics
 - Logic optimization
 - Technology mapping
 - Verification



Design of Integrated Systems



by courtesy of A. Kuehlmann

Courtesy of Prof. H.-R. Jiang

System Level

- **Abstract** algorithmic description of high-level behavior
 - e.g. C-Programming language

```
Port*
compute_optimal_route_for_packet(Packet_t *packet,
                                Channel_t *channel)
{
    static Queue_t *packet_queue;

    packet_queue = add_packet(packet_queue, packet);
    ...
}
```

- Abstract because it does not contain any implementation details for timing or data
- Efficient to get a compact execution model as first design draft
- Difficult to maintain throughout project because no link to implementation

RTL Level

- Cycle accurate model “close” to the hardware implementation
 - Bit-vector data types and operations as abstraction from bit-level implementation
 - Sequential constructs (e.g. if - then - else, while loops) to support modeling of complex control flow

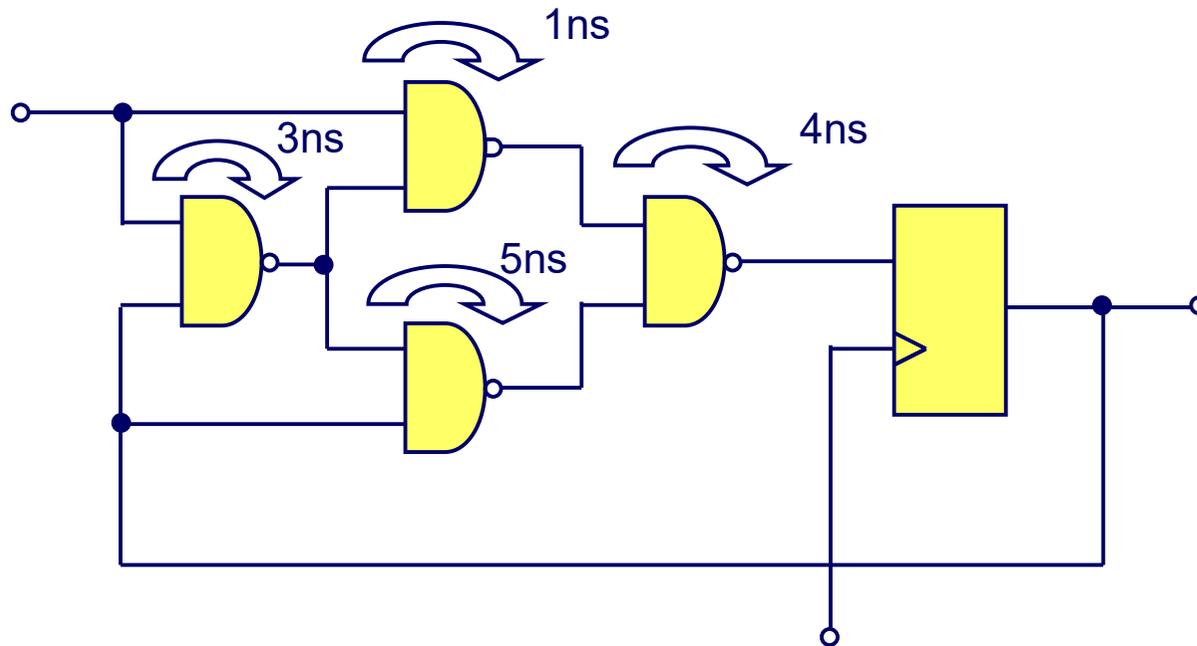
```
module mark1;
reg [31:0] m[0:8192];
reg [12:0] pc;
reg [31:0] acc;
reg[15:0] ir;

always
begin
    ir = m[pc];
    if(ir[15:13] == 3b'000)
        pc = m[ir[12:0]];
    else if (ir[15:13] == 3'b010)
        acc = -m[ir[12:0]];
    ...
end
endmodule
```

by courtesy of A. Kuehlmann

Gate Level

- Model on finite-state machine level
 - Models function in Boolean logic using registers and gates
 - Various delay models for gates and wires

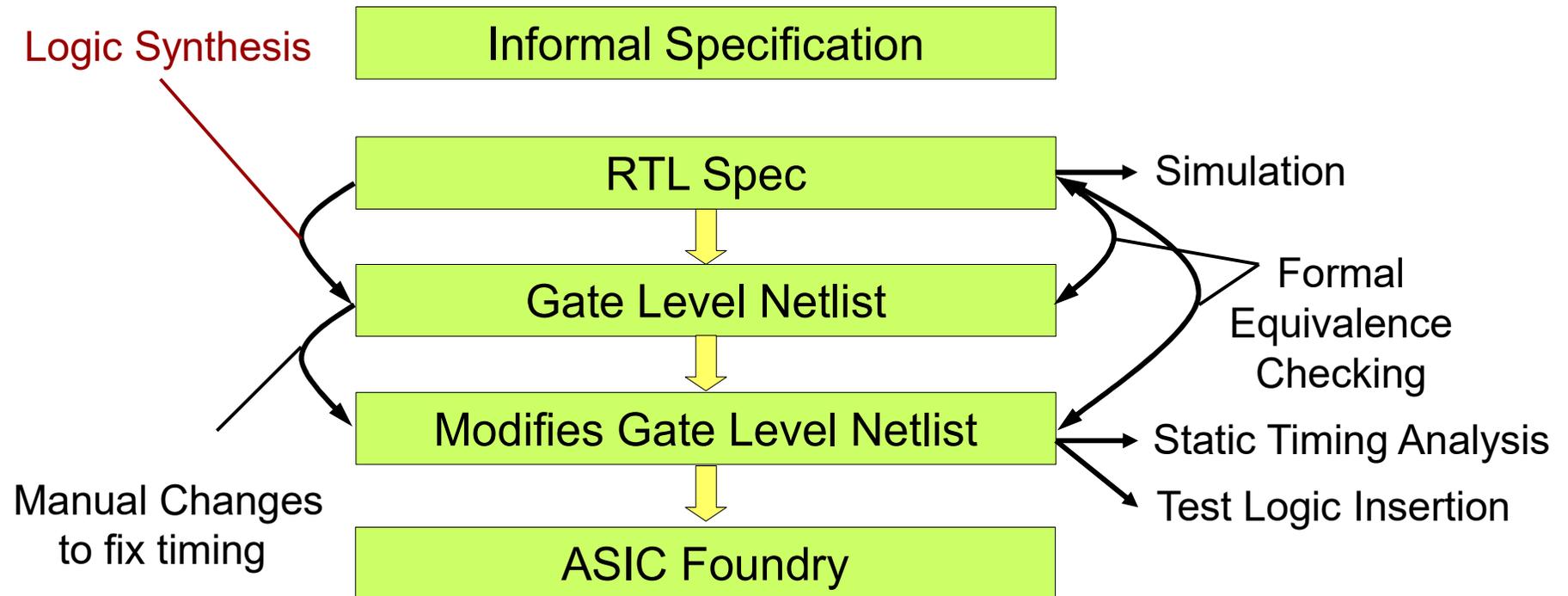


- In this lecture we will mostly deal with gate level

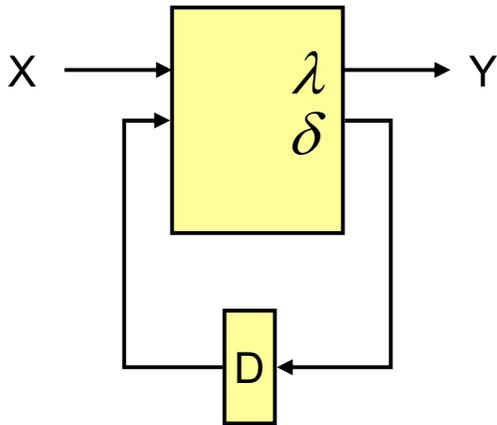
by courtesy of A. Kuehlmann

ASIC Design Flow

- Incomplete picture:



What is Logic Synthesis?



Given: Finite-State Machine $F(X, Y, Z, \lambda, \delta)$ where:

X: Input alphabet

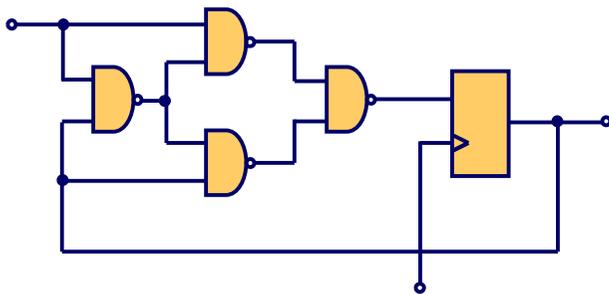
Y: Output alphabet

Z: Set of internal states

$\lambda : X \times Z \rightarrow Y$ (output function)

$\delta : X \times Z \rightarrow Z$ (next state function)

These are Boolean Functions!!



Target: Circuit $C(G, W)$ where:

G: set of circuit components $g \in \{\text{Boolean gates, flip-flops, etc}\}$

W: set of wires connecting G

so-called Netlist!

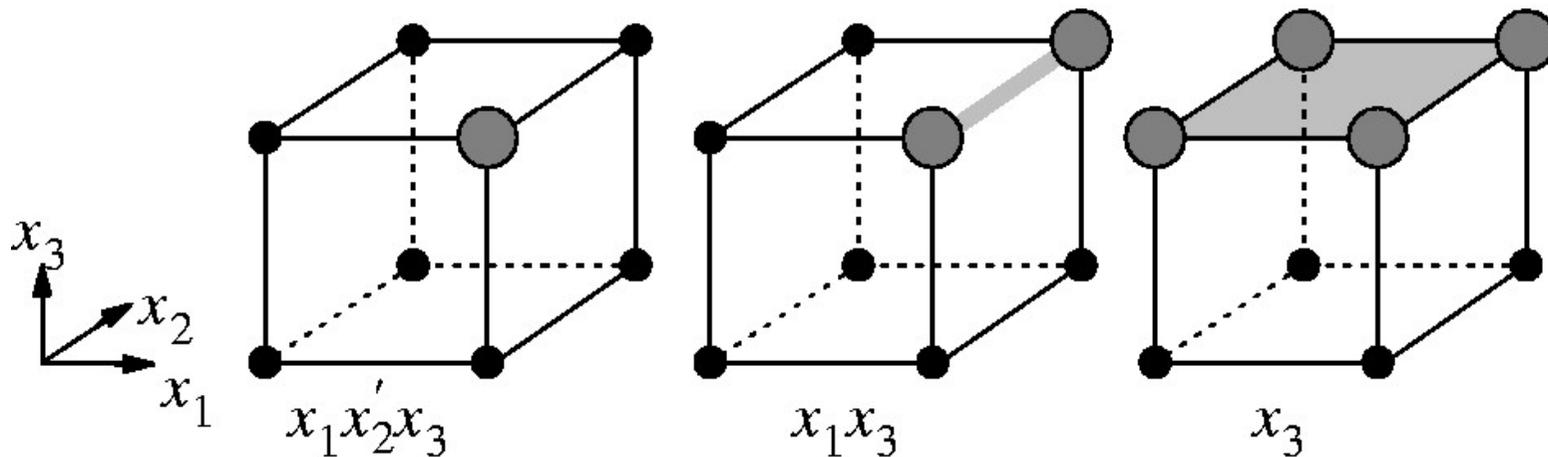
by courtesy of A. Kuehlmann

Boolean Functions

- $B = \{0, 1\}$, $Y = \{0, 1, D\}$
- A Boolean function $f: B^m \rightarrow Y^n$
 - $f = \bar{x}_1 \bar{x}_2 + \bar{x}_1 \bar{x}_3 + \bar{x}_2 x_3 + x_1 x_2 + x_2 \bar{x}_3 + x_1 x_3$
- Input variables: x_1, x_2, \dots, x_m
- The value of the output partitions B^m into three sets
 - the on-set (1)
 - the off-set (0)
 - the don't-care set (D)
 - f is an incompletely specified function if the don't care set is nonempty. Otherwise, f is a completely specified function.

Minterms and Cubes

- A **literal** is a Boolean variable x or its negation x' (or \bar{x} , $\neg x$) in a Boolean formula
- A **minterm** is a product of **all** input variables or their negations.
 - A minterm corresponds to a single point in B^m
- A **cube** is a product of the input variables or their negations (product of literals).
 - The fewer the number of variables in the product, the bigger the space covered by the cube.



Courtesy of Prof. H.-R. Jiang

Implicant and Cover

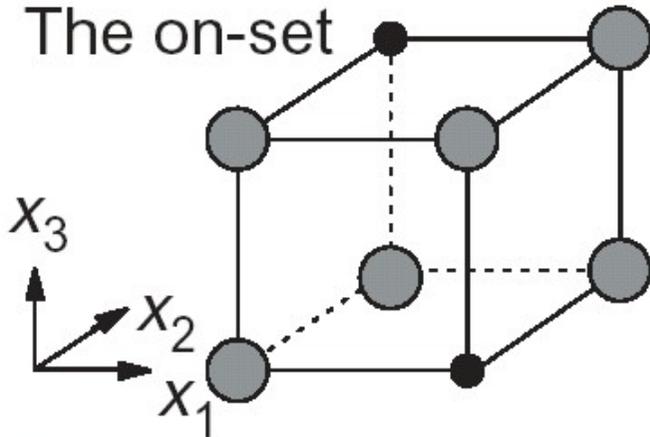
- An **implicant** is a cube whose minterms are either in the on-set or the dc-set.
- A **prime implicant** is an implicant that is not included in any other implicant.
- A set of prime implicants that together cover all minterms in the on-set (and some or all minterms of the dc-set) is called a **prime cover**.
 - A prime cover is **irredundant** when none of its prime implicants can be removed from the cover.
 - An irredundant prime cover is **minimal** when the cover has the minimal number of prime implicants.

Cover Examples

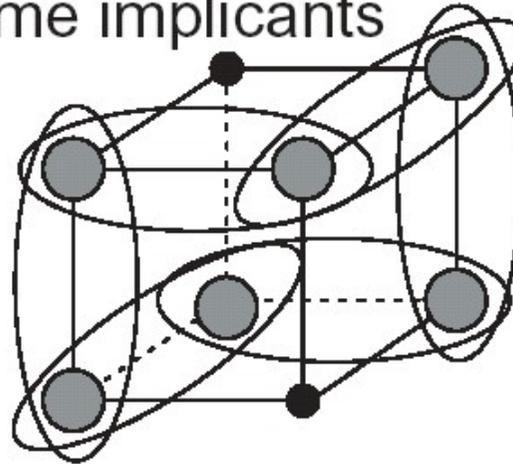
- $f = \bar{x}_1 \bar{x}_3 + \bar{x}_2 x_3 + x_1 x_2$

- $f = \bar{x}_1 \bar{x}_2 + x_2 \bar{x}_3 + x_1 x_3$

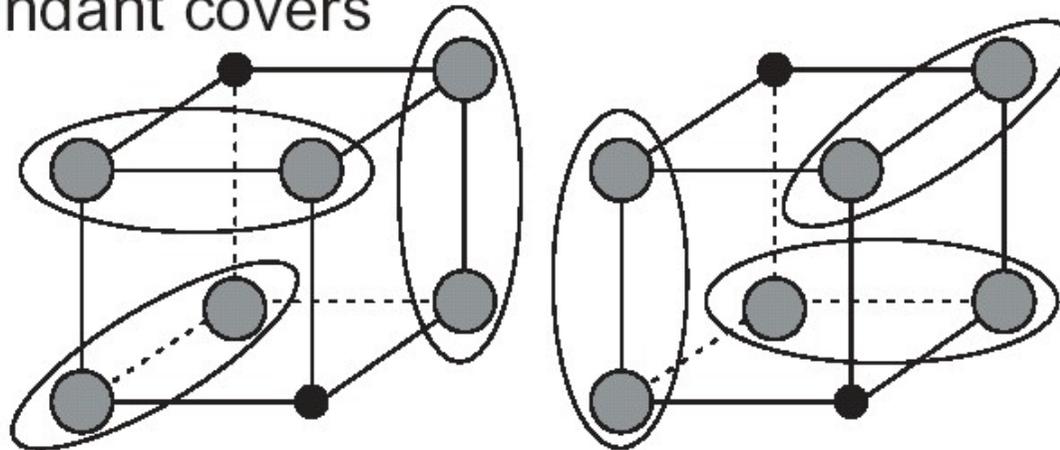
The on-set



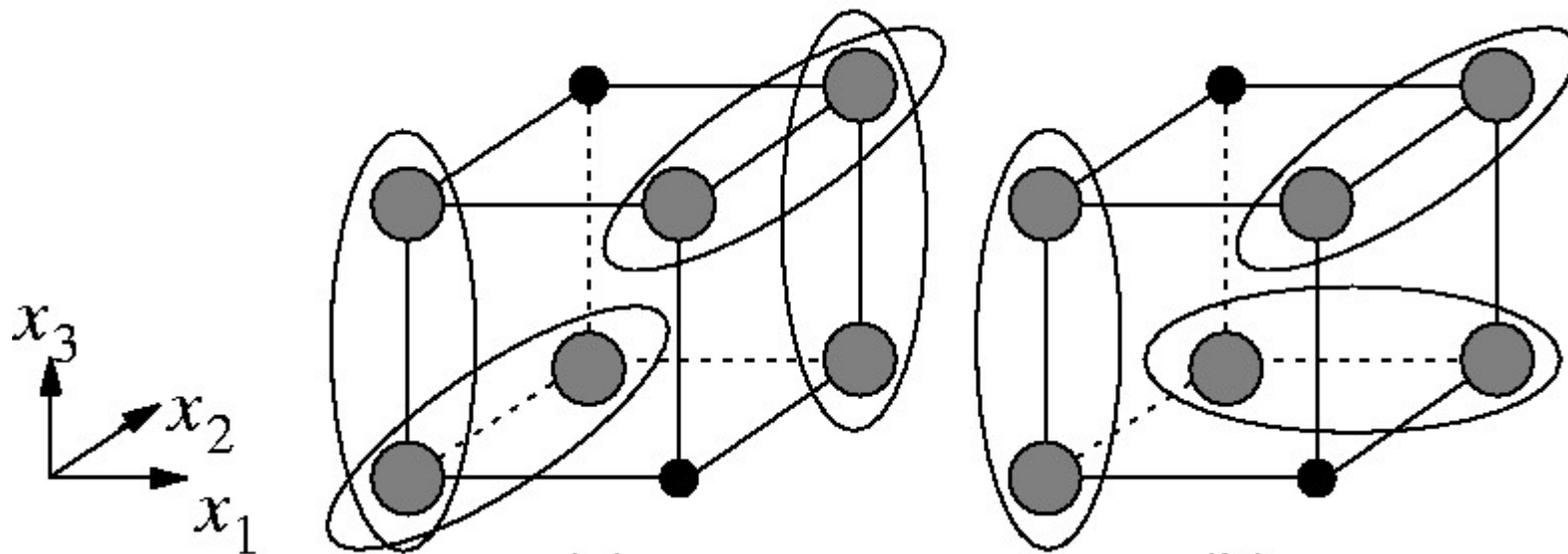
All prime implicants



Two irredundant covers



Optimality of Covers



A local and a global minimum

Representation of Boolean Functions

- Some common representations:
 - Truth table
 - SOP: sum-of-products, a.k.a. DNF (disjunctive normal form)
 - POS: product-of-sums, a.k.a. **CNF** (conjunctive normal form)
 - BDD: binary decision diagram
 - Boolean network
 - Network of PLAs
 - Network of nodes with complex Boolean functions
 - And-Inv Graph (AIG)

Truth Table

- The **truth table** (function table) of a function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ is a tabulation of its value at each of the 2^n vertices of \mathbf{B}^n .

- i.e., the truth table lists all **mintems**

- e.g.,

$$f = a'b'c'd + a'b'cd + a'bc'd + ab'c'd + ab'cd + abc'd + abcd' + abcd$$

	<u>abcd</u>	<u>f</u>		<u>abcd</u>	<u>f</u>
0	0000	0	8	1000	0
1	0001	1	9	1001	1
2	0010	0	10	1010	0
3	0011	1	11	1011	1
4	0100	0	12	1100	0
5	0101	1	13	1101	1
6	0110	0	14	1110	1
7	0111	0	15	1111	1

- The truth table representation is

- Intractable for large n

- Canonical

- A **canonical form** of a Boolean function is a **unique** representation of the function

SOP

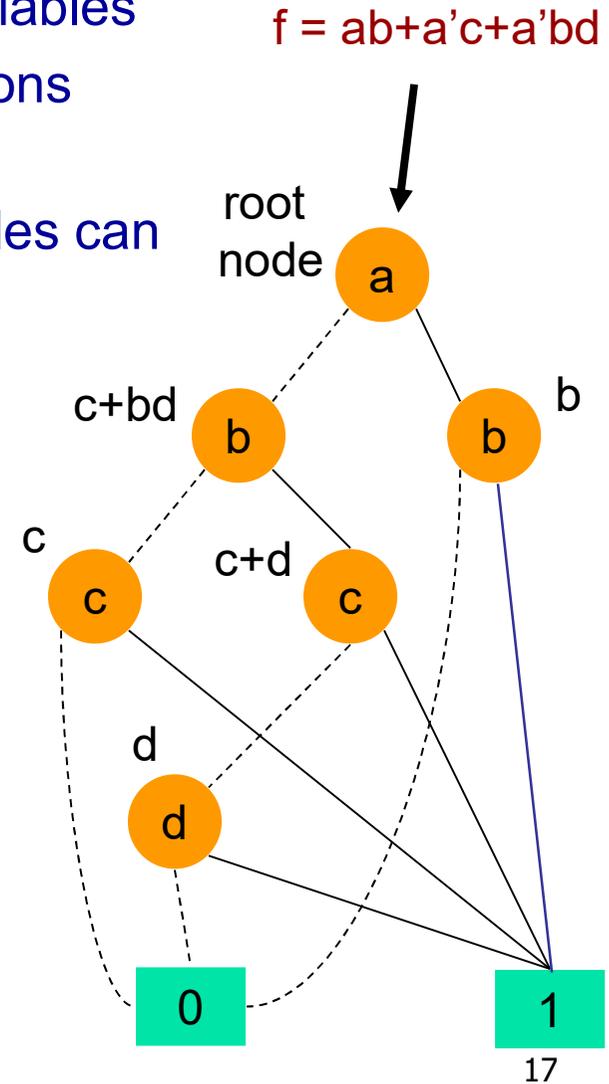
- A function can be represented by a sum of products (or sum of cubes, disjunctive normal form):
 - $f = ab + ac + bc$
- Since each **cube** is a product of literals, this is a sum of products (SOP) representation
- A SOP can be thought of as a set of cubes F
 - $F = \{ab, ac, bc\}$
- A set of cubes that represents f is called a **cover** of f
 - $F_1 = \{ab, ac, bc\}$ and $F_2 = \{abc, abc', ab'c, a'bc\}$
are covers of $f = ab + ac + bc$.

POS

- Dual to SOP representation, a function can be represented by a product of sums (or **conjunctive normal form**):
 - $f = (a+b'+c) (a'+b+c) (a+b'+c') (a+b+c)$
- A Boolean function in a POS representation can be derived from an SOP representation with De Morgan's law and the distributive law

Binary Decision Diagram (BDD)

- Graph representation of a Boolean function f
 - Vertices represent decision nodes for variables
 - Two children represent the two subfunctions
 - $f(x = 0)$ and $f(x = 1)$ (cofactors)
 - Restrictions on ordering and reduction rules can make a BDD representation canonical
 - Reduced Ordered BDD (ROBDD)



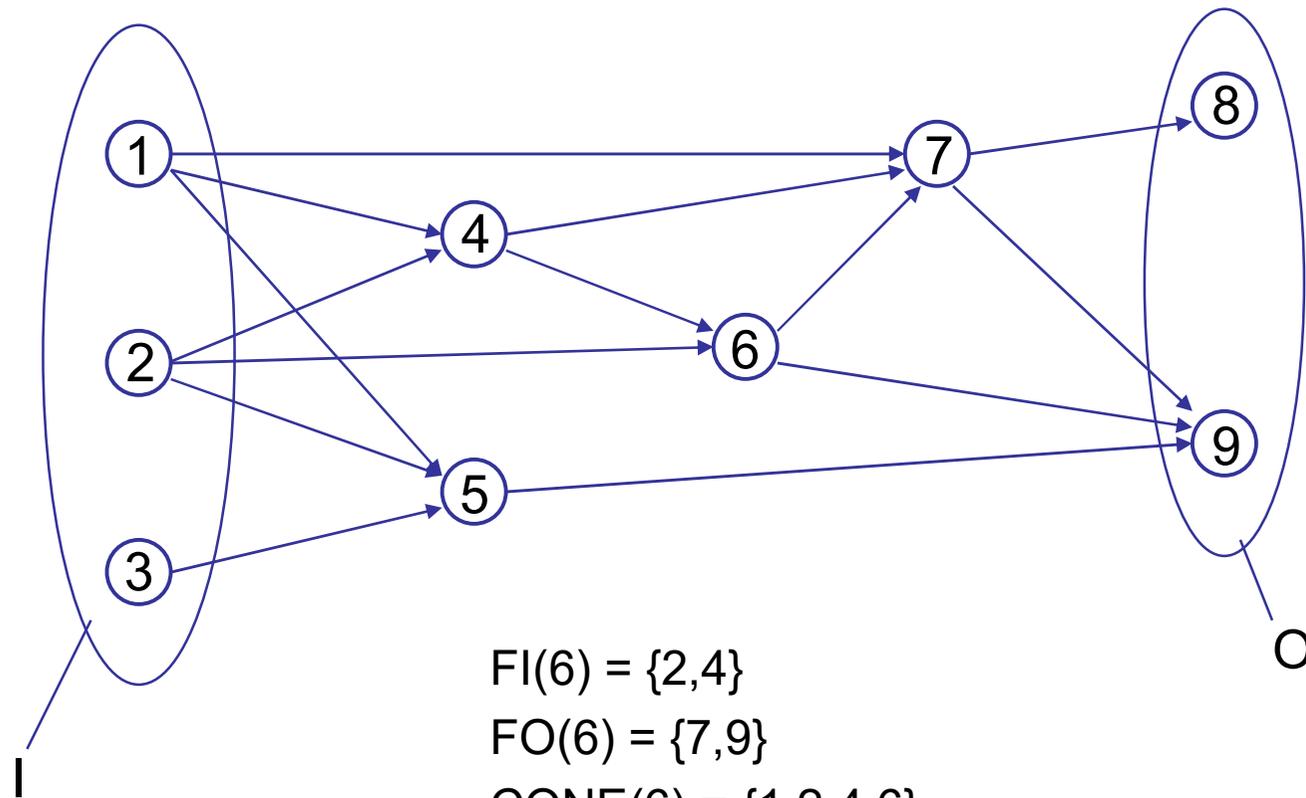
Boolean Network

- A **Boolean circuit** is a directed graph $C(G,N)$ where G are the gates and N are the directed edges (nets) connecting the gates.
- Some of the vertices are designated:
Inputs: $I \subseteq G$
Outputs: $O \subseteq G, I \cap O = \emptyset$
- Each gate g is assigned a Boolean function f_g which computes the output of the gate in terms of its inputs.

Boolean Network (cont'd)

- The **fanin** $FI(g)$ of a gate g are all predecessor vertices of g :
 - $FI(g) = \{g' \mid (g',g) \in N\}$
- The **fanout** $FO(g)$ of a gate g are all successor vertices of g :
 - $FO(g) = \{g' \mid (g,g') \in N\}$
- The **cone** $CONE(g)$ of a gate g is the transitive fanin of g and g itself.
- The **support** $SUPPORT(g)$ of a gate g are all inputs in its cone:
 - $SUPPORT(g) = CONE(g) \cap I$

Example Boolean Network



$$FI(6) = \{2,4\}$$

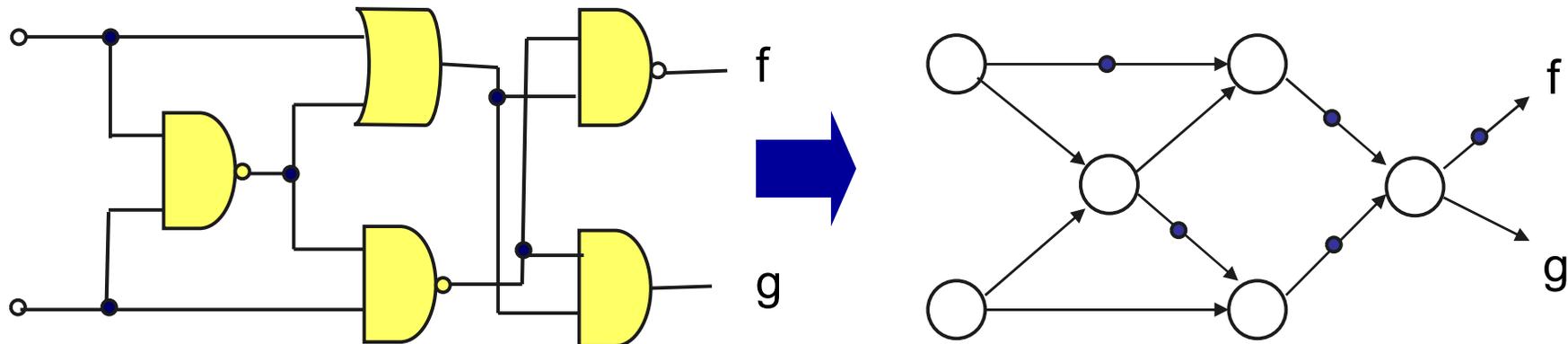
$$FO(6) = \{7,9\}$$

$$CONE(6) = \{1,2,4,6\}$$

$$SUPPORT(6) = \{1,2\}$$

And/Inverter Graph (AIG)

- Base data structure uses two-input AND function for vertices and INVERTER attributes at the edges (individual bit)
 - Use De'Morgan's law to convert OR operation etc.



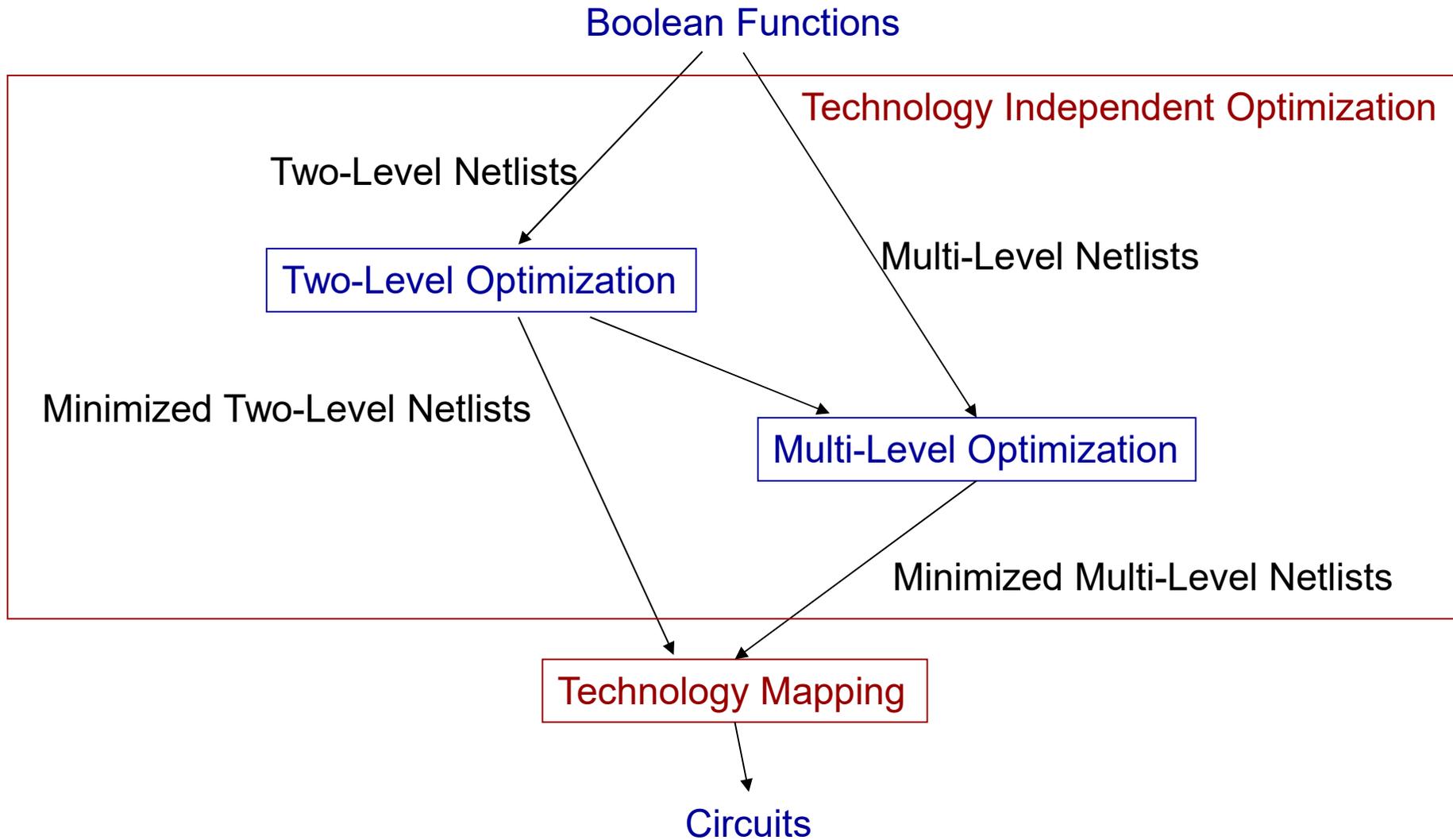
Canonical Forms

- A **canonical form** of a Boolean function is a **unique** representation of the function.
 - It can be used for verification purposes.
- The **truth table** or the **sum of minterms** are canonical forms
 - They grow exponentially with the number of input variables.
- An irredundant prime cover is not a canonical form.
- **Reduced ordered binary decision diagram (ROBDD):** a canonical form that is interesting from a practical point of view.

Comparisons of Different Representations

- Truth table
 - Canonical
 - Useful in representing small functions
- SOP
 - Useful in two-level logic optimization
- POS
 - Useful in SAT solving and Boolean reasoning
- ROBDD
 - Canonical
 - Useful in formal verification and Boolean reasoning
- Boolean network
 - Useful in multi-level logic optimization

Combinational Optimization

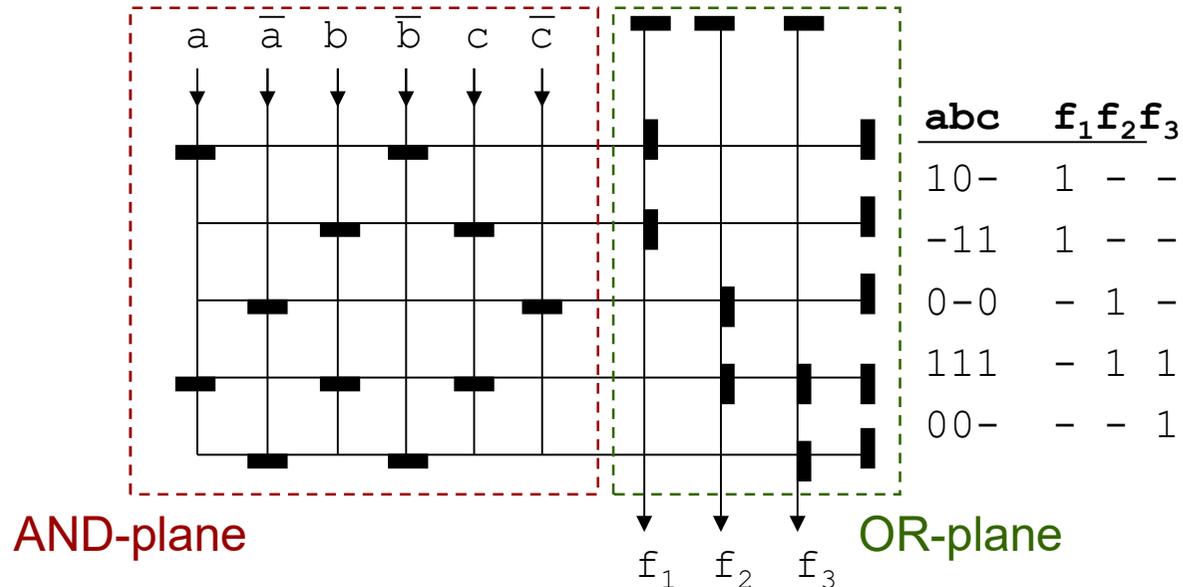


by courtesy of J.-H. Jiang
Unit 10

Courtesy of Prof. H.-R. Jiang

Two-Level Logic Minimization

- Any Boolean function can be realized using PLAs in two levels: AND-OR (**SOP**), NAND-NAND, etc.



- Classic problem solved by the *Quine-McClusky* algorithm [1956].
 - Basic idea: Boolean law $x + x' = 1$
 - Cost function: #cubes and #literals in an SOP expression
 - #cubes: #rows in PLAs
 - #literals: #transistors in PLAs

Unit 10 — Objective: to find a **minimum irredundant prime cover**

Two-Level Logic Minimization

- Exact algorithm
 - The Quine-McClusky algorithm
- Heuristic algorithm
 - Espresso

The Quine-McClusky Algorithm (Exact)

- Given G and D (covers for $\mathfrak{S} = (f, d, r)$ and d , respectively), find a minimum cover G^* of primes where:

$$f \subseteq G^* \subseteq f+d$$

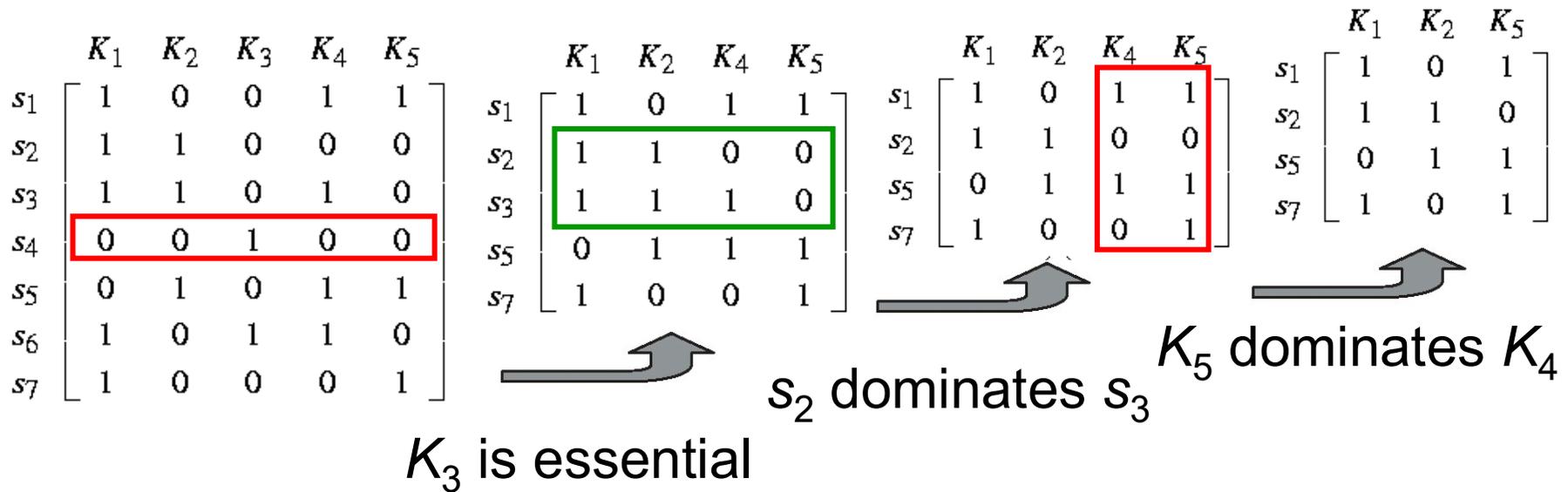
- G^* : prime cover of \mathfrak{S}
- f : onset, d : don't-care set, r : offset
- Q-M Procedure
 1. Generate all the primes of \mathfrak{S} , $\{P_j\}$ (i.e., primes of $(f+d)=G+D$)
 2. Find a minimum cover:
 - 2.1 Generate all the minterms $\{m_i\}$ of $f+d$
 - 2.2 Build the covering matrix B where
$$B_{ij} = 1 \text{ if } m_i \in P_j$$
$$= 0 \text{ otherwise}$$
 - 2.3 Solve the **minimum(-cost) column covering** problem for B

Example: The Quine-McClusky Algorithm

- $F(a, b, c, d) = \sum_m(2, 3, 7, 9, 11, 13) + \sum_d(1, 10, 15)$
- Calculate **all** prime implicants (of the union of the on-set and dc-set) by grouping minterms using $xy + xy' = x$
- Find the minimum cover of all minterms in the on-set by prime implicants.
 - Construct the covering matrix
 - Simplify the covering matrix by detecting **essential** columns, **row and column dominance**.
 - What is left is the **cyclic core** of the covering matrix – solved by a branch-and-bound algorithm

Step 1-1	Step 1-2	Step 1-3	Step 2																																			
1 0001 v	(1, 3) 00-1 v	(1, 3, 9, 11) -0-1	<table border="1"> <thead> <tr> <th>prime imp.</th> <th>2</th> <th>3</th> <th>7</th> <th>9</th> <th>11</th> <th>13</th> </tr> </thead> <tbody> <tr> <td>(1, 3, 9, 11)</td> <td></td> <td>x</td> <td></td> <td>x</td> <td>x</td> <td></td> </tr> <tr> <td>*(2, 3, 10, 11)</td> <td>x</td> <td>x</td> <td></td> <td></td> <td>x</td> <td></td> </tr> <tr> <td>*(3, 7, 11, 15)</td> <td></td> <td></td> <td>x</td> <td></td> <td>x</td> <td></td> </tr> <tr> <td>*(9, 11, 13, 15)</td> <td></td> <td></td> <td></td> <td></td> <td>x</td> <td>x</td> </tr> </tbody> </table> <p>* essential prime implicant</p> <p>$F = b'c + cd + ad$</p>	prime imp.	2	3	7	9	11	13	(1, 3, 9, 11)		x		x	x		*(2, 3, 10, 11)	x	x			x		*(3, 7, 11, 15)			x		x		*(9, 11, 13, 15)					x	x
prime imp.	2	3		7	9	11	13																															
(1, 3, 9, 11)		x			x	x																																
*(2, 3, 10, 11)	x	x				x																																
*(3, 7, 11, 15)				x		x																																
*(9, 11, 13, 15)						x	x																															
2 0010 v	(1, 9) -001 v	(2, 3, 10, 11) -01-																																				
3 0011 v	(2, 3) 001- v	(3, 7, 11, 15) -11																																				
9 1001 v	(2, 10) -010 v	(9, 11, 13, 15) 1-1																																				
10 1010 v	(3, 7) 0-11 v																																					
7 0111 v	(3, 11) -011 v																																					
11 1011 v	(9, 11) 10-1 v																																					
13 1101 v	(9, 13) 1-01 v																																					
15 1111 v	(10, 11) 101- v																																					
	(7, 15) -111 v																																					
	(11, 15) 1-11 v																																					
	(13, 15) 11-1 v																																					

Example: Simplification Rules in Covering

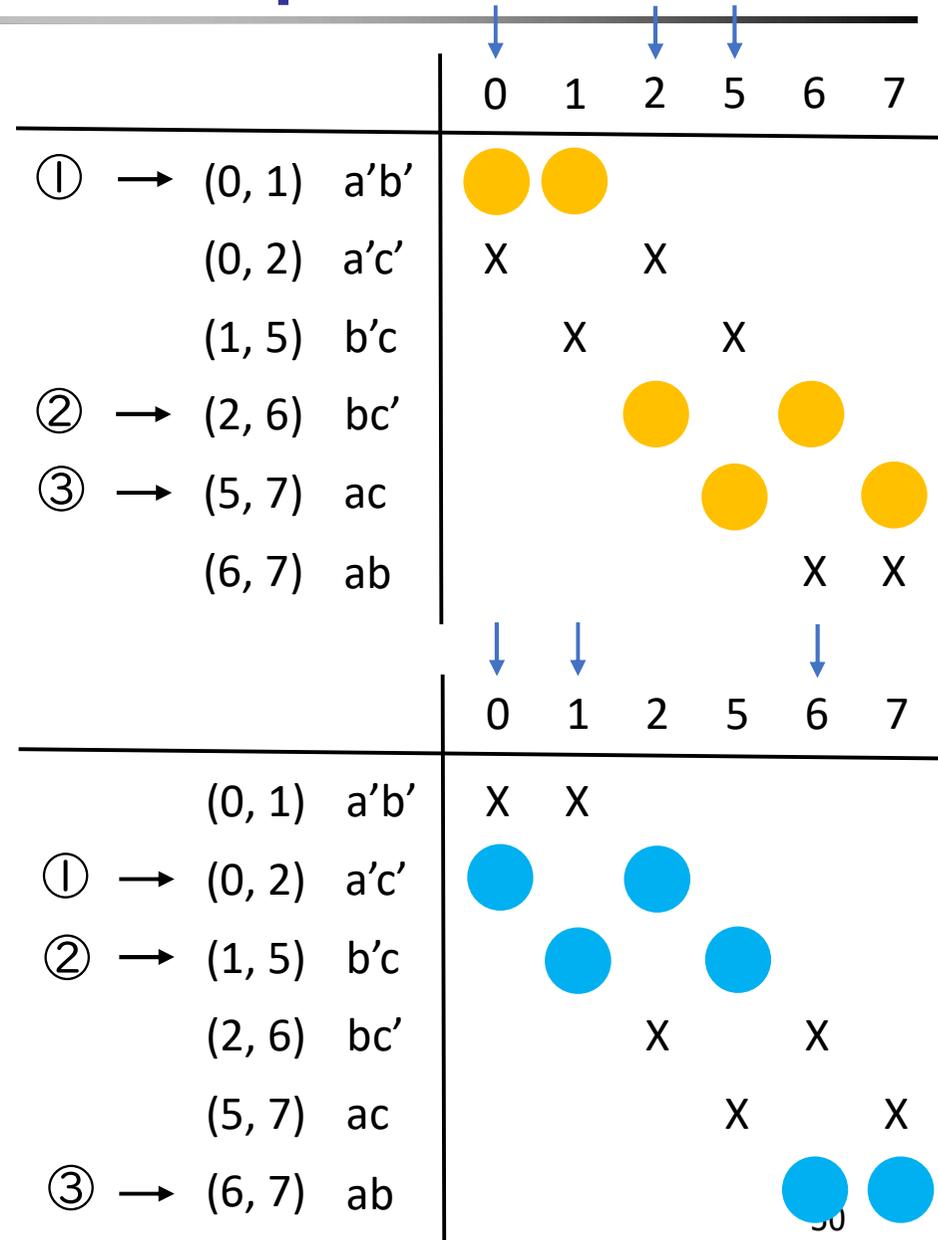


Example: Cyclic Prime Implicant Chart

$$F = \Sigma m(0, 1, 2, 5, 6, 7)$$

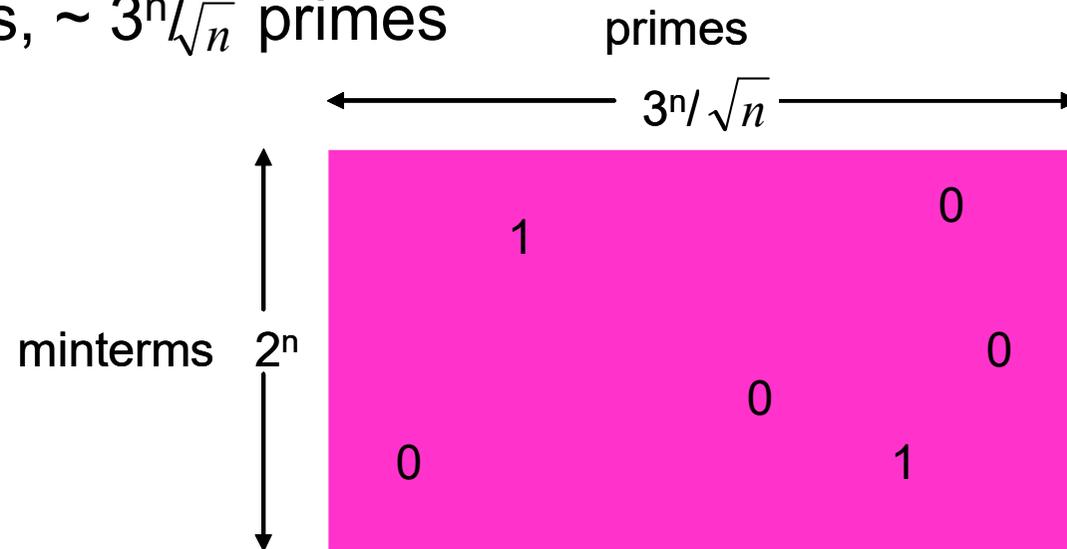
Derivation of Prime Implicants

→	<u>0 000</u>	0,1 00--
→	1 001	<u>0,2 0--0</u>
→	<u>2 010</u>	1,5 --01
→	5 101	<u>2,6 --10</u>
→	<u>6 110</u>	5,7 1--1
→	7 111	6,7 11--



Difficulty of Q-M

- $\sim 2^n$ minterms, $\sim 3^n/\sqrt{n}$ primes



- Thus $O(2^n)$ rows and $O(3^n/\sqrt{n})$ columns in covering matrix AND minimum covering problem is NP-complete.
- Q-M is **exact** but **intractable**

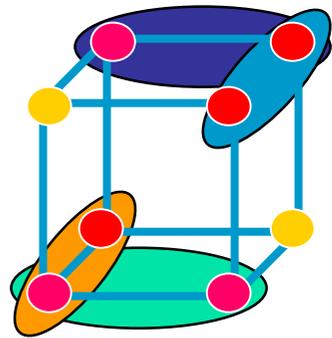
From Exact to Heuristic

- Exact: Q-M
 1. Generate **cover** of **all** primes
 2. Make G^* irredundant (in **optimum** way)
- Heuristic: Espresso
 1. Generate (**somehow**) a cover of \mathfrak{S} using **some** of the primes
 2. Make G^* irredundant (maybe **not** optimally)
 3. Keep best result – try again (i.e. go to 1)

ESPRESSO

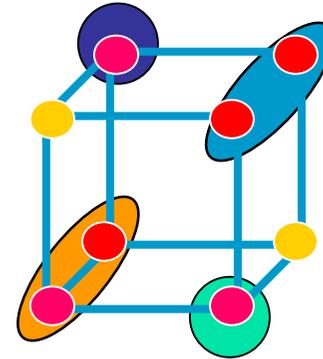
- REDUCE
 - Reduce size of each implicant while preserving cover
- EXPAND
 - Make implicants *prime*
 - Remove covered implicants
- IRREDUNDANT_COVER
 - Make cover *irredundant*
- Repeat REDUCE / EXPAND / IRREDUNDANT_COVER to find alternative covers

ESPRESSO Illustrated

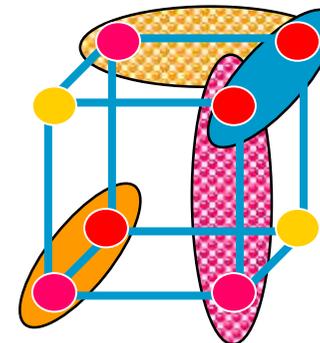
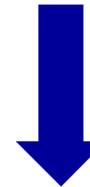


local minimum

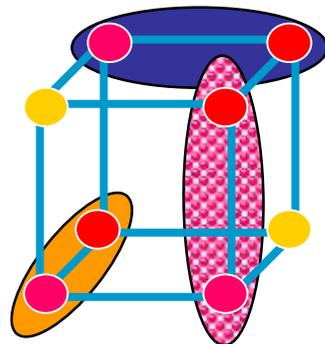
REDUCE



EXPAND



IRREDANDANT



local minimum

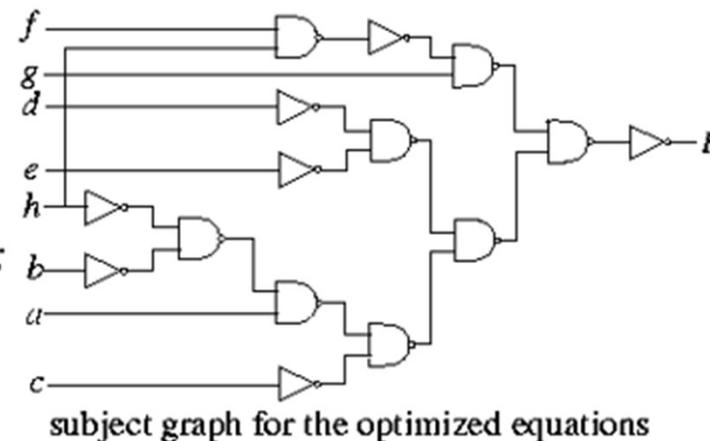
by courtesy of A. Kuehlmann
Unit 10

Courtesy of Prof. H.-R. Jiang

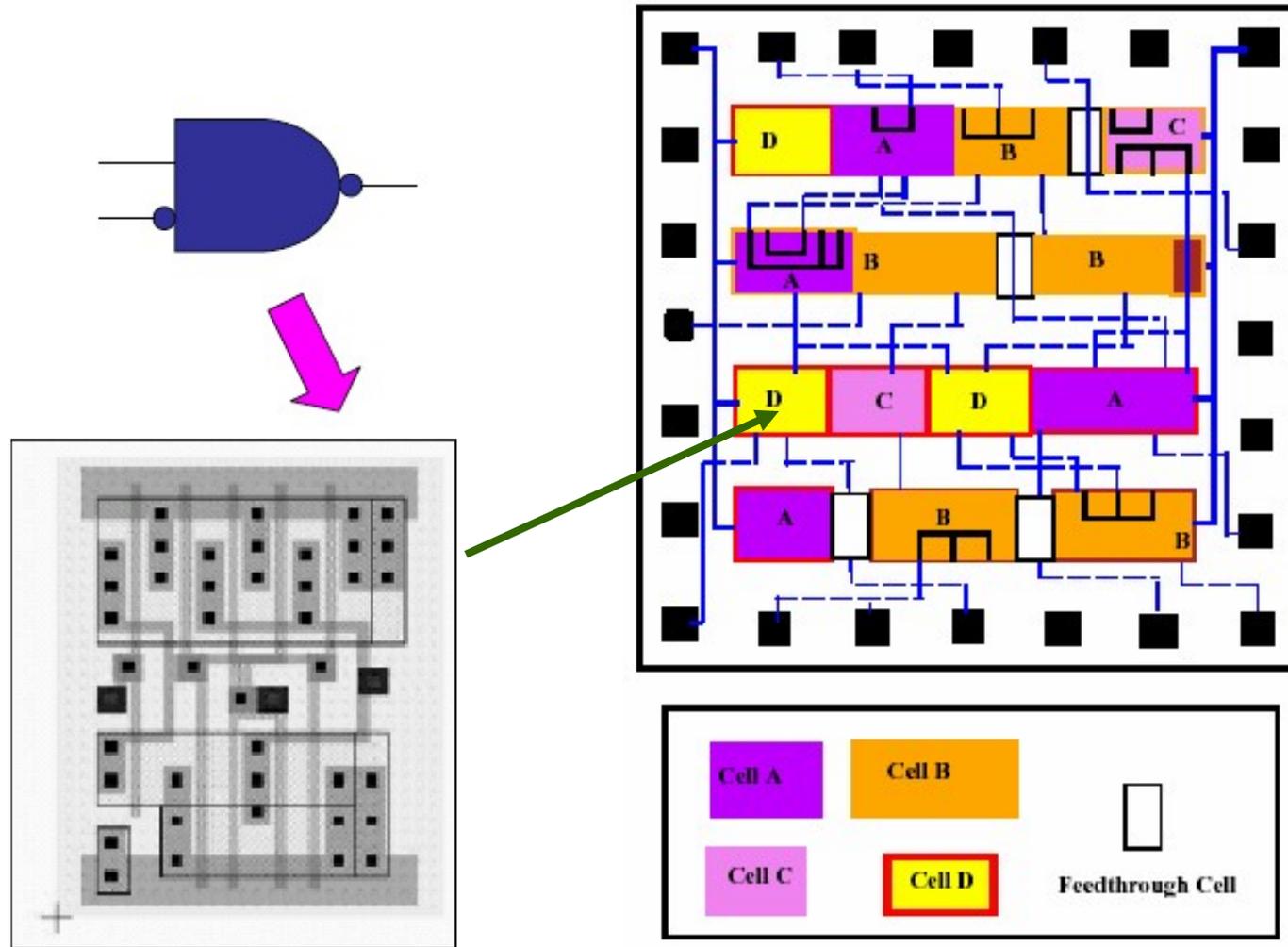
Multi-Level Logic Minimization

- Basic techniques in Boolean network manipulation:
 - Structural operations/transformations (change topology)
 - Algebraic
 - Algorithmic and rule-based
 - Node simplification/Boolean methods (change node functions)
 - Don't cares
 - Node minimization
(two-level logic minimization!)
- In commercial use for years: Synopsys, MIS

$$\begin{array}{l}
 t1 = a + b \ c; \\
 t2 = d + e; \\
 t3 = a \ b + c; \\
 t4 = t1 \ t2 + f \ g; \\
 t5 = t4 \ h + t2 \ t3; \\
 F = t5';
 \end{array}
 \xrightarrow{\text{logic optimization}}
 \begin{array}{l}
 t1 = d + e; \\
 t2 = b + h; \\
 t3 = a \ t2 + c; \\
 t4 = t1 \ t3 + f \ g \ h;
 \end{array}$$



Multi-Level Circuits Implementation: Standard Cells



by courtesy of J.-H. Jiang
Unit 10

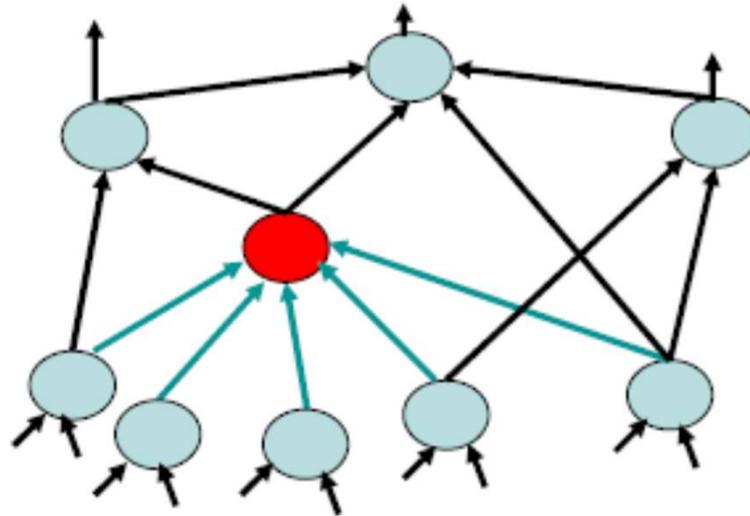
Courtesy of Prof. H.-R. Jiang

Structural Operations

- Decomposition (single function)
 - $f = abc+abd+a'c'd'+b'c'd'$
 - $f = xy+x'y' \quad x = ab \quad y = c+d$
- Extraction (multiple functions)
 - $f = (az+bz')cd+e \quad g = (az+bz')e' \quad h = cde$
 - $f = xy+e \quad g = xe' \quad h = ye \quad x = az+bz' \quad y = cd$
- Factoring (series-parallel decomposition)
 - $f = ac+ad+bc+bd+e$
 - $f = (a+b)(c+d)+e$
- Substitution
 - $g = a+b \quad f = a+bc$
 - $f = g(a+c)$
- Collapsing (also called elimination)
 - $f = ga+g'b \quad g = c+d$
 - $f = ac+ad+bc'd' \quad g = c+d$
- Note: “division” plays a key role in all of these (algebraic models)

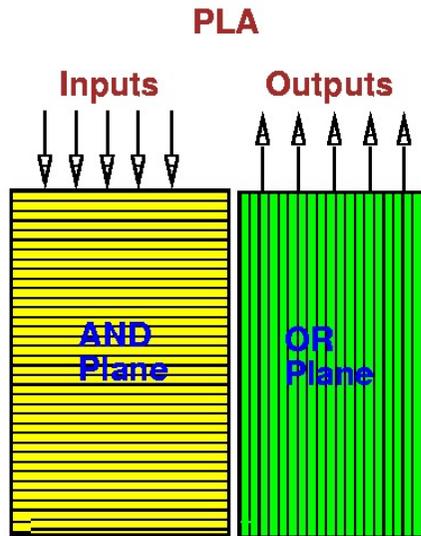
Node Simplification

- **Goal:** For any node of a given Boolean network, find a **least-cost** sum-of-products expression among the implementable functions at the node
 - Don't care computation + two-level logic minimization



Combinational Boolean network

Two-Level (PLA) vs. Multi-Level

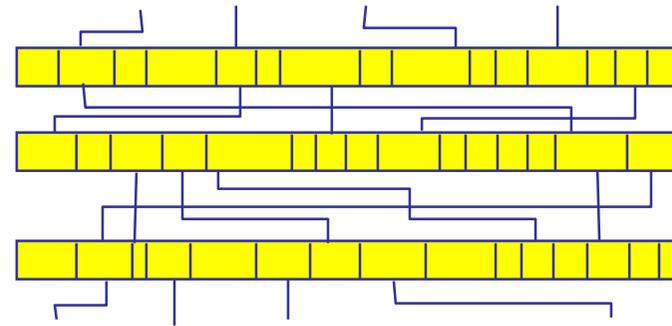


PLA

- control logic
- constrained layout
- highly automatic
- technology independent
- multi-valued logic
- input, output, state encoding

Very predictable

E.g. Standard Cell Layout

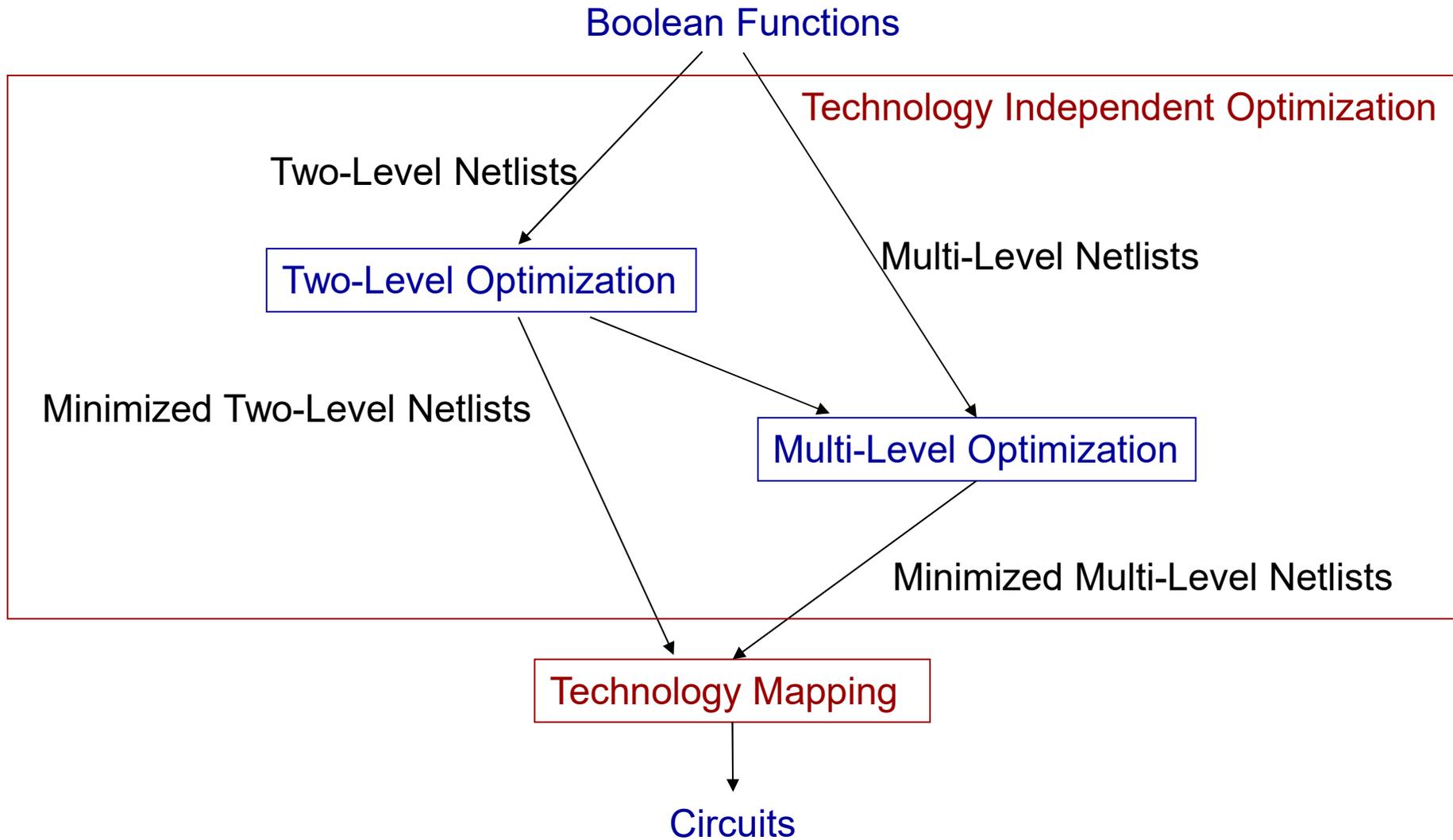


Multi-level Logic

- all logic
- general (e.g. standard cell, regular blocks,...)
- automatic
- partially technology independent
- some ideas
- part of multi-level logic

Very hard to predict

Combinational Optimization Revisited



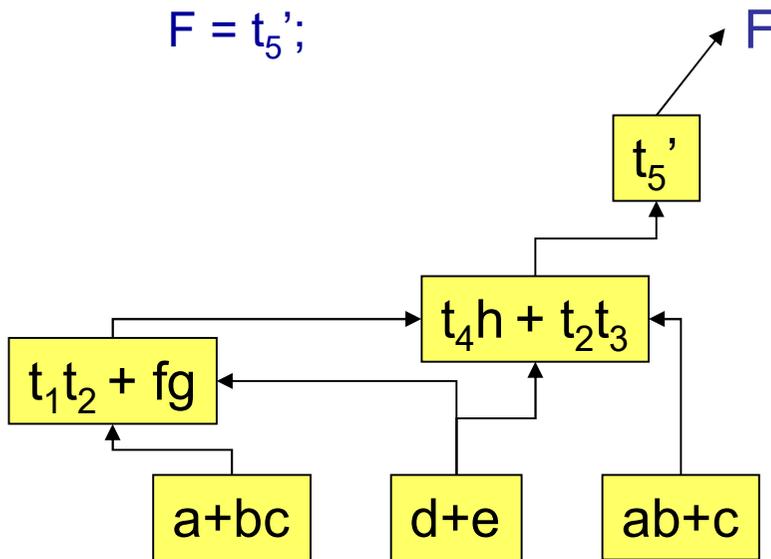
by courtesy of J.-H. Jiang
Unit 10

Courtesy of Prof. H.-R. Jiang

Example: Technology Independent Optimization

- An unoptimized set of logic equations consists of **17** literals

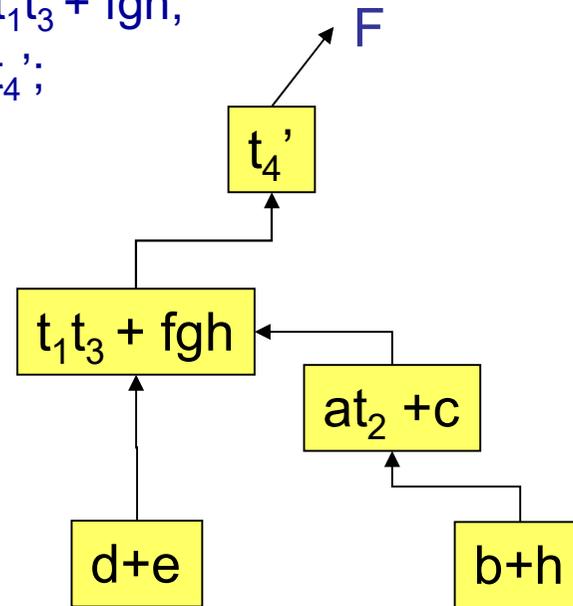
$$\begin{aligned}
 t_1 &= a + bc; \\
 t_2 &= d + e; \\
 t_3 &= ab + c; \\
 t_4 &= t_1 t_2 + fg; \\
 t_5 &= t_4 h + t_2 t_3; \\
 F &= t_5';
 \end{aligned}$$



by courtesy of A. Kuehlmann

- After technology independent optimization, these equations are optimized using only **13** literals

$$\begin{aligned}
 t_1 &= d + e; \\
 t_2 &= b + h; \\
 t_3 &= at_2 + c; \\
 t_4 &= t_1 t_3 + fgh; \\
 F &= t_4';
 \end{aligned}$$



Technology Mapping/Library Binding

- Implement the optimized network using a set of gates which form a **library**. Each gate has a **cost** (i.e. its area, delay, power, etc.)
 - Algorithmic approaches: DAGON, MISII
- Represent each function of a network using a set of **base functions**. This representation is called the **subject graph**.
 - Typically the base is 2-input NANDs and inverters [MISII].
- Each gate of the library is likewise represented using the base set. This generates **pattern graphs**
 - Represent each gate in all possible ways
- **Technology Mapping**: The optimization problem of finding a minimum cost covering of the **subject graph** by choosing from the collection of **pattern graphs** for all gates in the library.

Subject Graph

- The optimized network:

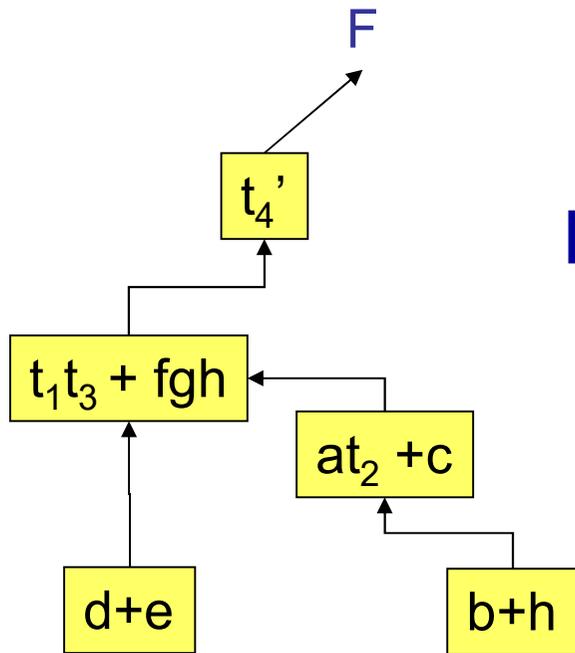
$$t_1 = d + e;$$

$$t_2 = b + h;$$

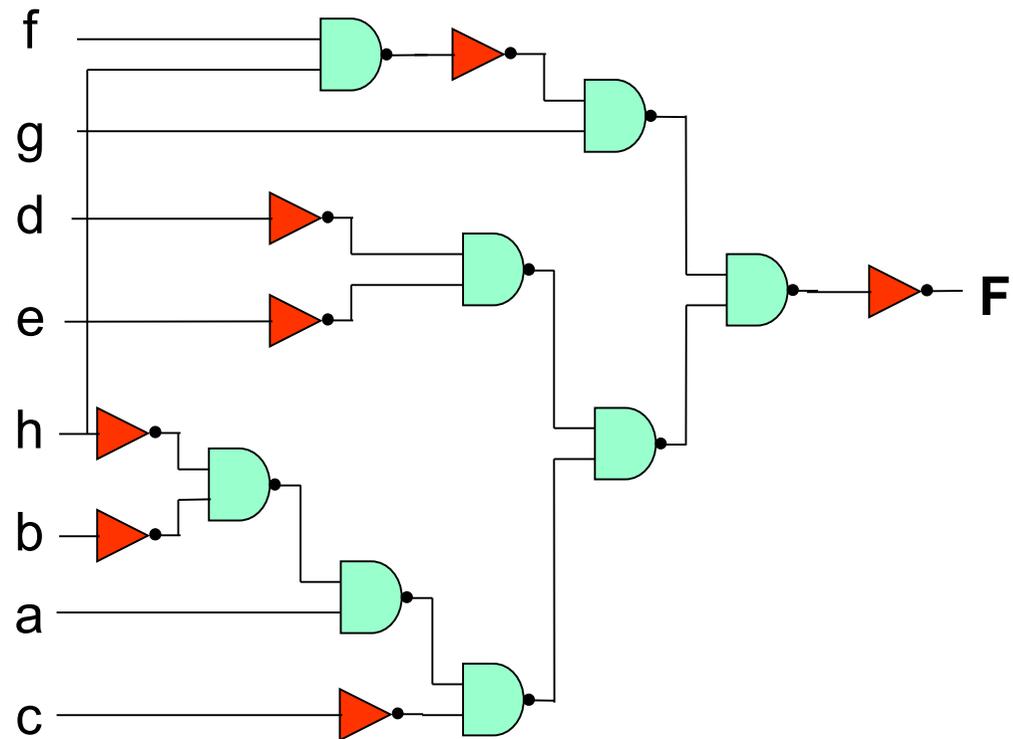
$$t_3 = at_2 + c;$$

$$t_4 = t_1t_3 + fgh;$$

$$F = t_4';$$



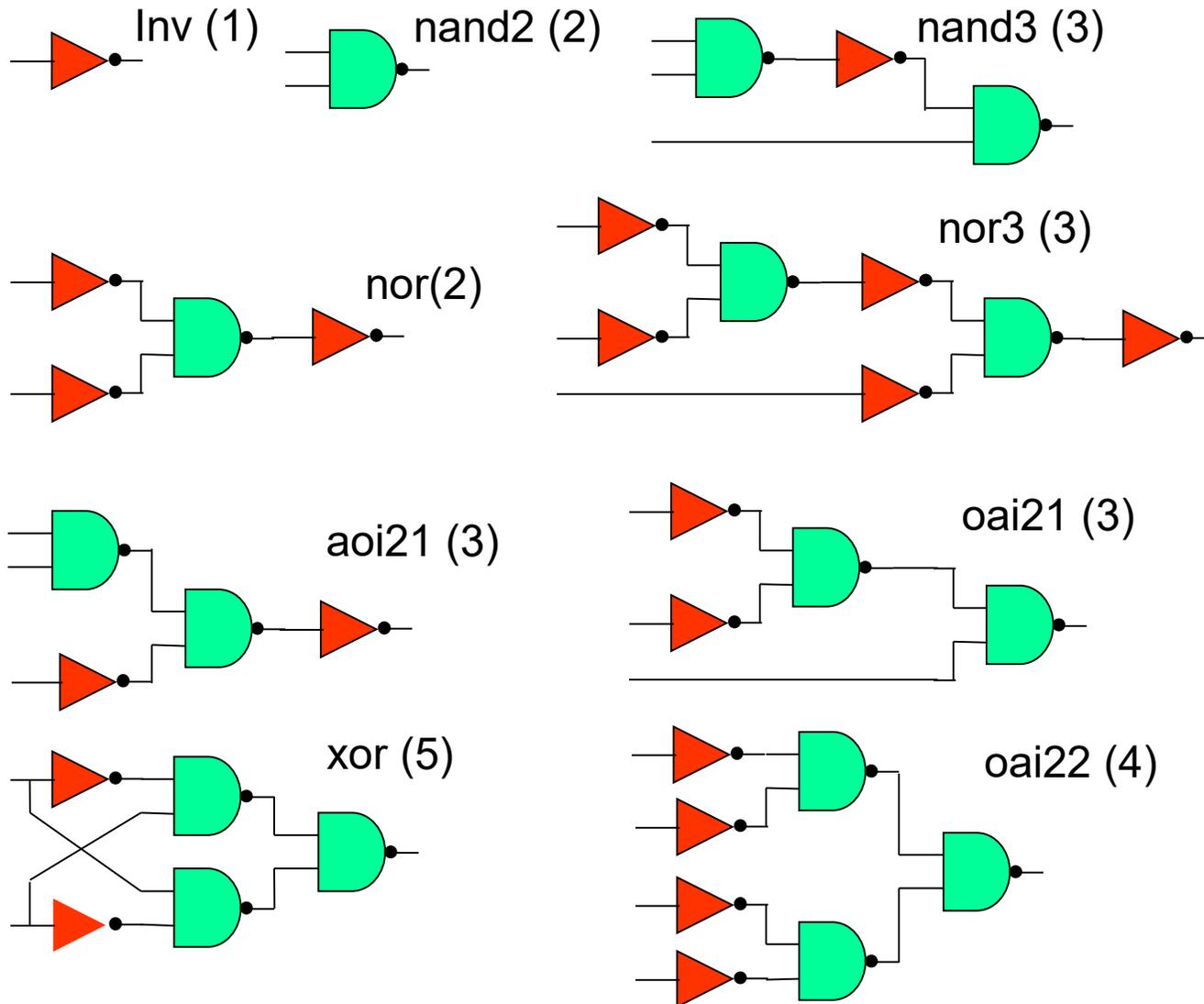
- Subject graph of 2-input NANDs and inverters:



by courtesy of A. Kuehlmann

Courtesy of Prof. H.-R. Jiang

Pattern Graphs for the IWLS Library



by courtesy of A. Kuehlmann

Courtesy of Prof. H.-R. Jiang

Trivial Covering

$$t_1 = d + e;$$

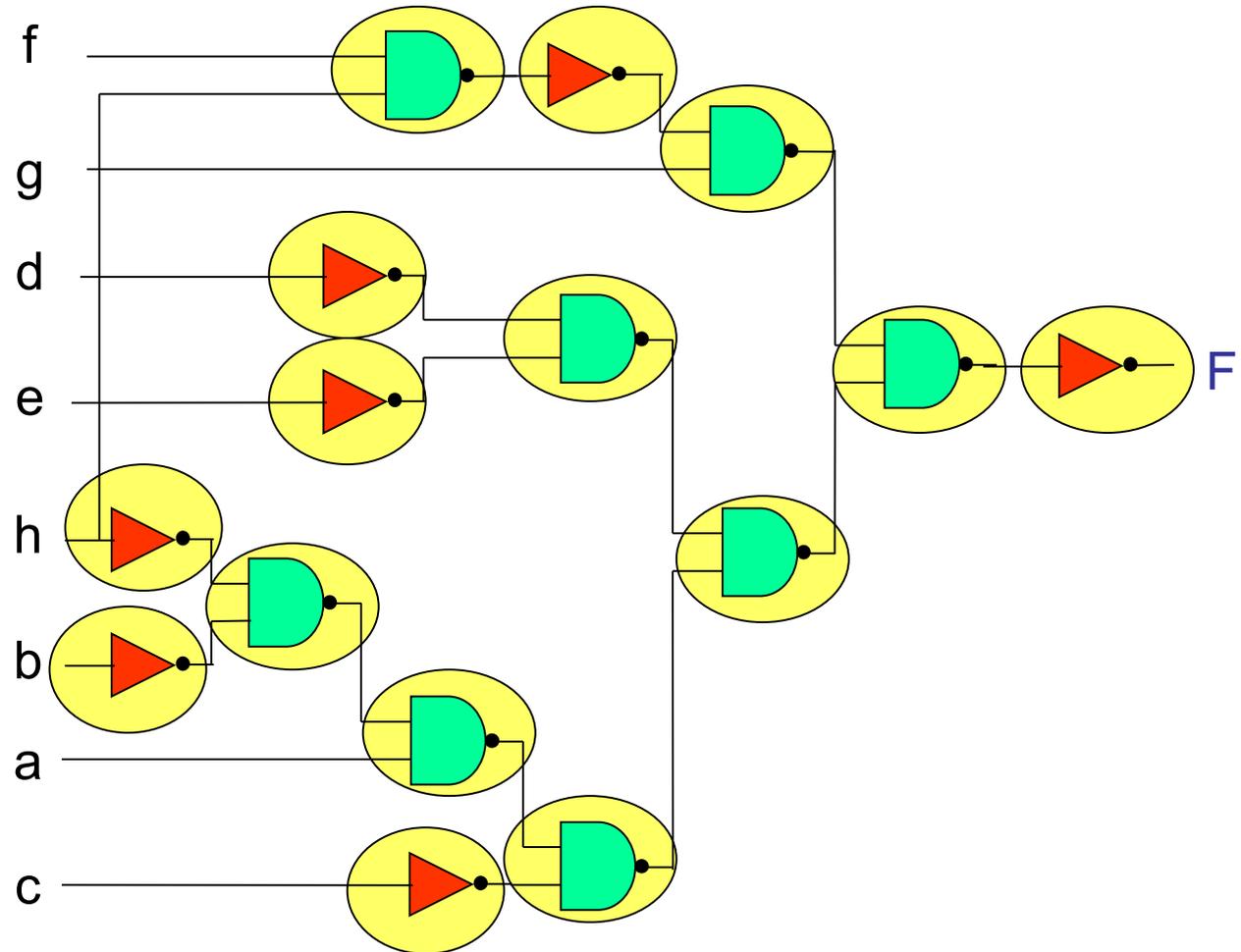
$$t_2 = b + h;$$

$$t_3 = at_2 + c;$$

$$t_4 = t_1t_3 + fgh;$$

$$F = t_4';$$

Total cost = 23



by courtesy of A. Kuehlmann

Courtesy of Prof. H.-R. Jiang

Best Covering

$$t_1 = d + e;$$

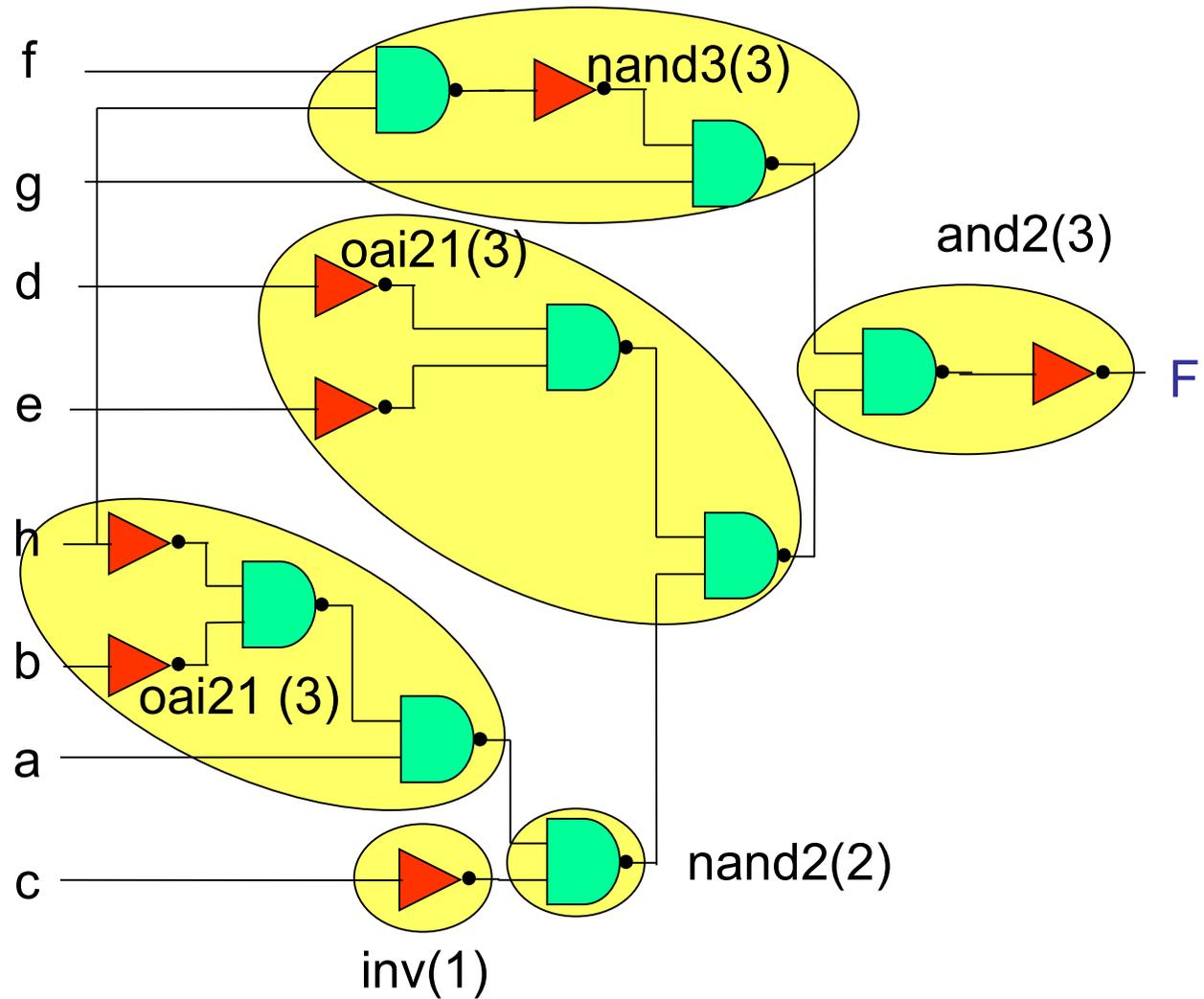
$$t_2 = b + h;$$

$$t_3 = at_2 + c;$$

$$t_4 = t_1 t_3 + fgh;$$

$$F = t_4';$$

Total cost = 15



by courtesy of A. Kuehlmann

Optimal Tree Covering by Dynamic Programming

- If the subject DAG and primitive DAG's are **trees**, then an efficient algorithm to find the best cover exists
 - Based on **dynamic programming**: optimal substructure? overlapping subproblems?
- Given
 - Subject trees (networks to be mapped)
 - Library cells
- Consider a node N of a subject tree:
 - **Recursive Assumption**: for all children of N , a **best** cost match (which implements the node) is known
 - Cost of a **leaf** of the tree is 0.
 - Compute cost of each pattern tree which matches at N ,
Cost = SUM of best costs of implementing each input of pattern plus the cost of the pattern
 - Choose least cost matching pattern for implementing N

Dynamic Programming (DP) : A Brief Review

- Typically apply to **optimization problem**.
- Generic approach
 - Calculate the solutions to **all** subproblems.
 - Proceed computation from the small subproblems to the larger subproblems.
 - Compute a subproblem based on previously computed results for smaller subproblems.
 - Store the solution to a subproblem in a table and never recompute.
- Development of a DP
 1. Characterize the structure of an optimal solution.
 2. Recursively define the value of an optimal solution.
 3. Compute the value of an optimal solution bottom-up.
 4. Construct an optimal solution from computed information (omitted if only the optimal value is required).

Matrix-Chain Multiplication

- If A is a $p \times q$ matrix and B a $q \times r$ matrix, then $C = AB$ is a $p \times r$ matrix

$$C[i, j] = \sum_{k=1}^q A[i, k] B[k, j]$$

time complexity: $O(pqr)$.

- **The matrix-chain multiplication problem:** Given a chain $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices, matrix A_i has dimension $p_{i-1} \times p_i$, parenthesize the product $A_1 A_2 \dots A_n$ to minimize the number of scalar multiplications.
- **Exp:** dimensions: $A_1: 4 \times 2$; $A_2: 2 \times 5$; $A_3: 5 \times 1$
 $(A_1 A_2) A_3$: total multiplications = $4 \times 2 \times 5 + 4 \times 5 \times 1 = 60$.
 $A_1 (A_2 A_3)$: total multiplications = $2 \times 5 \times 1 + 4 \times 2 \times 1 = 18$.
- **So the order of multiplications can make a big difference!**

Matrix-Chain Multiplication: Brute Force

- $A = A_1 A_2 \dots A_n$: How to evaluate A using the minimum number of multiplications?
- Brute force: check all possible orders?
 - $P(n)$: number of ways to multiply n matrices.

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

- $P(n) = \Omega\left(\frac{4^n}{n^{3/2}}\right)$, **exponential** in n .

- Any efficient solution? Dynamic programming!

Matrix-Chain Multiplication

- Step 1: the structure of an optimal parenthesization
 - Suppose that an optimal parenthesization of $A_i A_{i+1} \dots A_j$ splits the product between A_k and A_{k+1} . Then the parenthesization of the prefix $A_i A_{i+1} \dots A_k$ subchain within this optimal parenthesization of $A_i A_{i+1} \dots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \dots A_k$. (proof by contradiction)
- Step 2: recursive solution
 - $m[i, j]$: minimum number of multiplications to compute matrix $A_{i..j} = A_i A_{i+1} \dots A_j$, $1 \leq i \leq j \leq n$.
 - $m[1, n]$: the cheapest cost to compute $A_{1..n}$.

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

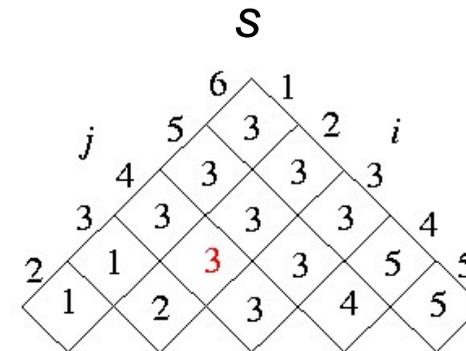
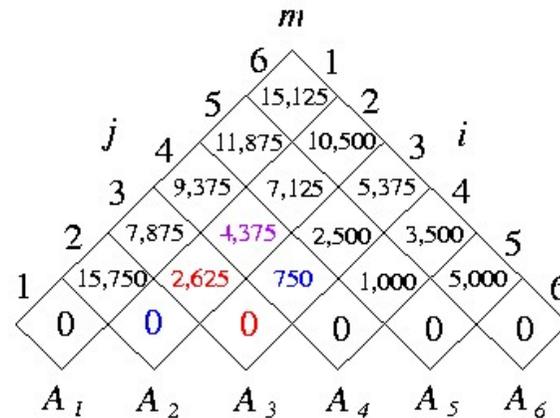
Bottom-Up DP Matrix-Chain Order

Matrix-Chain-Order(p)

1. $n \leftarrow \text{length}[p]-1$
2. **for** $i \leftarrow 1$ **to** n **do**
3. $m[i, i] \leftarrow 0$
4. **for** $l \leftarrow 2$ **to** n **do**
5. **for** $i \leftarrow 1$ **to** $n-l+1$ **do**
6. $j \leftarrow i+l-1$
7. $m[i, j] \leftarrow \infty$
8. **for** $k \leftarrow i$ **to** $j-1$ **do**
9. $q \leftarrow m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$
10. **if** $q < m[i, j]$ **then**
11. $m[i, j] \leftarrow q$
12. $s[i, j] \leftarrow k$
13. **return** m and s

$p = \langle p_0, p_1, \dots, p_n \rangle$

matrix	dimension
A_1	30 * 35
A_2	35 * 15
A_3	15 * 5
A_4	5 * 10
A_5	10 * 20
A_6	20 * 25

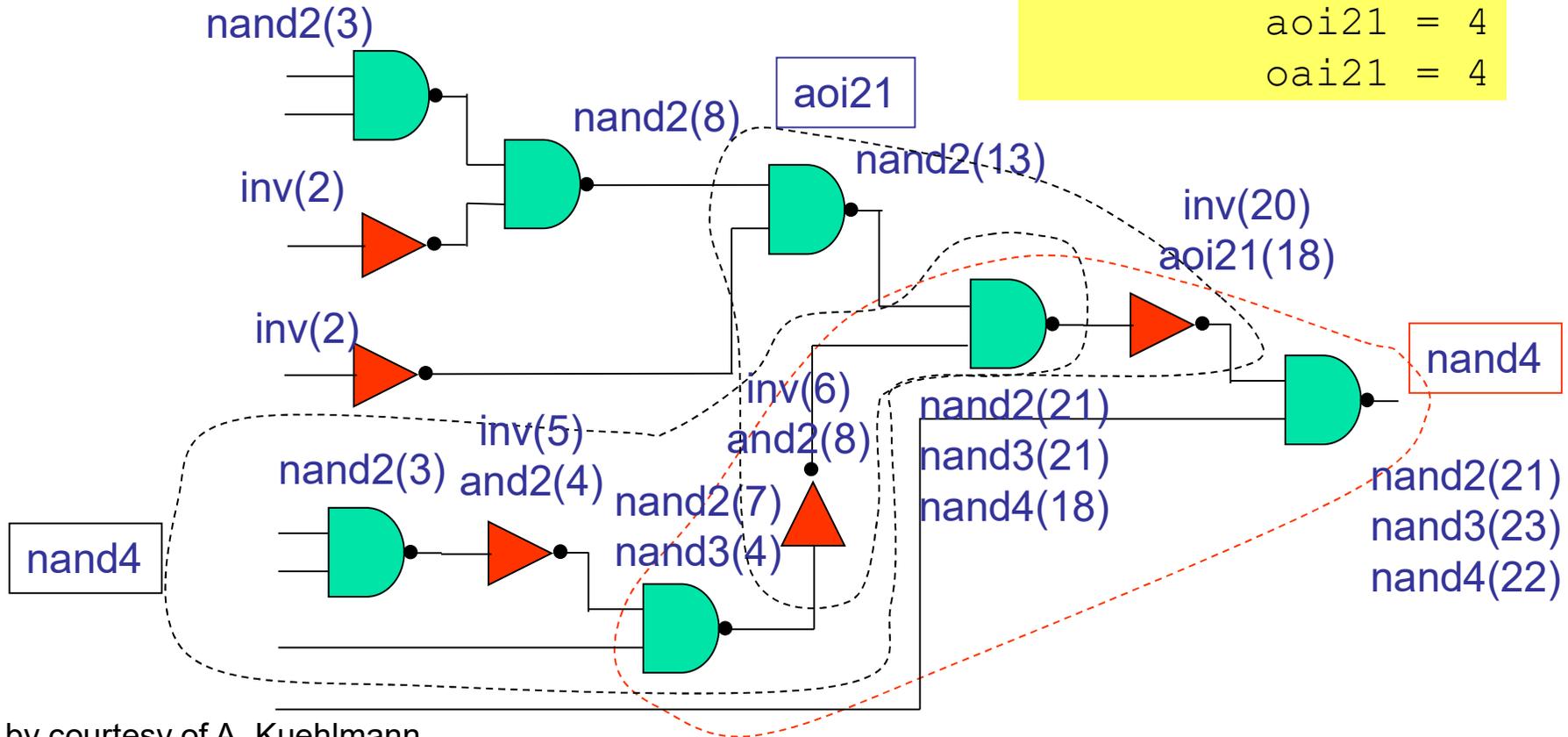


$$m[2, 4] = \min \left\{ \begin{array}{l} m[2, 2] + m[3, 4] + p_1 p_2 p_4 = 0 + 750 + 35 \times 15 \times 10 = 6000. \\ m[2, 3] + m[4, 4] + p_1 p_3 p_4 = 2625 + 0 + 35 \times 5 \times 10 = 4375. \end{array} \right.$$

Tree Covering in Action

- Step 1. Find all candidate matches
- Step 2. Find the optimal match

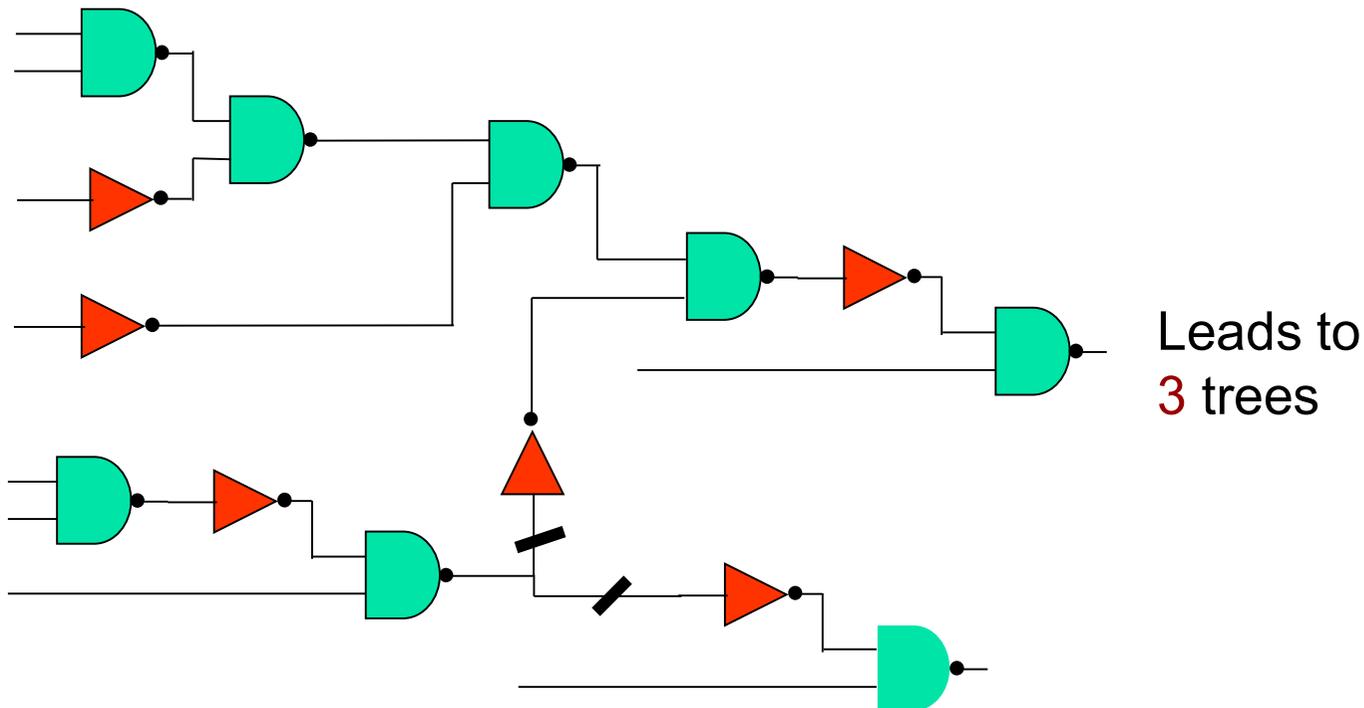
Library:	nand2	=	3
	inv	=	2
	nand3	=	4
	nand4	=	5
	and2	=	4
	aoi21	=	4
	oai21	=	4



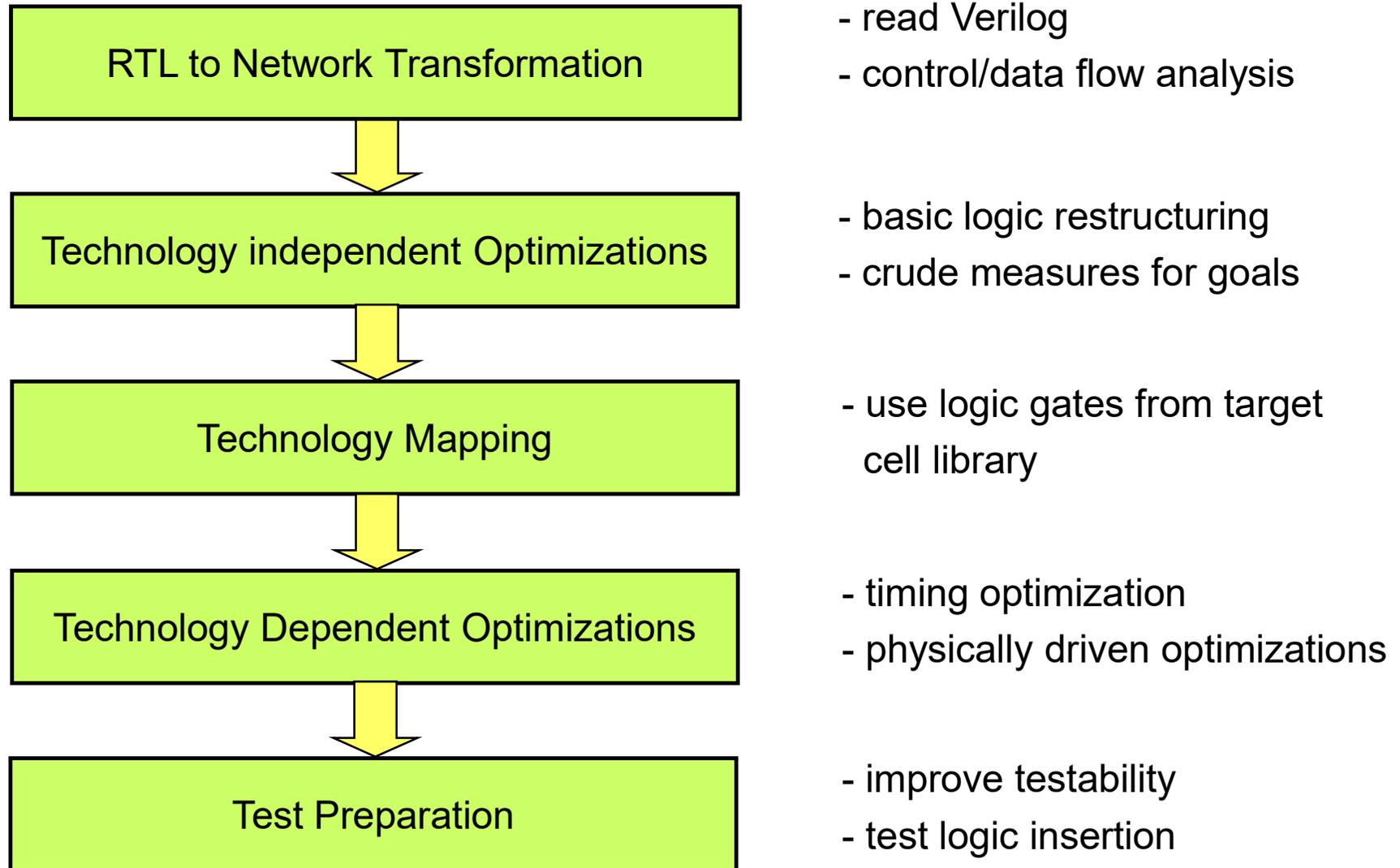
by courtesy of A. Kuehlmann
Unit 10

Tree-Covering by Dynamic Programming (DAGON)

- If the subject DAG is not a tree
 - Partition the subject graph into forest of trees
 - Cut the graph at all multiple fanout points
 - Cover each tree optimally using the dynamic programming
 - Overall solution is only an approximation



Typical Synthesis Scenario



by courtesy of A. Kuehlmann
Unit 10

Courtesy of Prof. H.-R. Jiang

Optimization Criteria for Synthesis

- The optimization criteria for multi-level logic is to minimize some function of:
 1. **Area occupied by the logic gates and interconnect** (approximated by literals = transistors in technology independent optimization)
 2. **Critical path delay** of the longest path through the logic
 3. **Degree of testability** of the circuit, measured in terms of the percentage of faults covered by a specified set of test vectors for an approximate fault model (e.g. single or multiple stuck-at faults)
 4. **Power** consumed by the logic gates
 5. **Noise Immunity**
 6. **Place-ability, Wire-ability**

while simultaneously satisfying upper or lower bound constraints placed on these physical quantities

Binary-Decision Diagram (BDD) Revisited

- BDD is a Directed Acyclic Graph (DAG) used to represent a Boolean function $f: B^m \rightarrow B^n$

- Terminal node

- Attribute

- $\text{value}(v)=0$

- $\text{value}(v)=1$

- Nonterminal node

- $\text{Index}(v)=i$

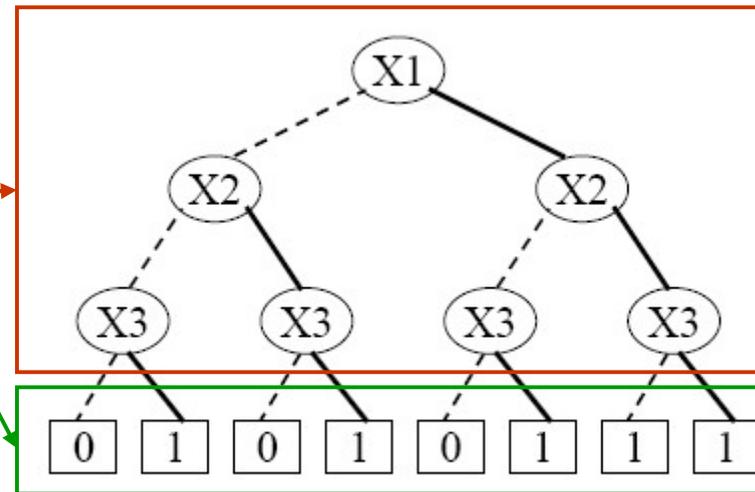
- Two children

- $\text{low}(v)$

- $\text{high}(v)$

$$f = x_i \cdot f_{x_i} + \bar{x}_i \cdot f_{\bar{x}_i}$$

$$f = x_1x_2 + x_3$$



----- : 0

————— : 1

Binary-Decision Diagram (BDD) Principles

- **Restriction** resulting in the **positive** and **negative** cofactors (two restrictions) of a Boolean function:

$$f_{x_i} = f(x_1, \dots, x_{i-1}, '1', x_{i+1}, \dots, x_m)$$

$$f_{\bar{x}_i} = f(x_1, \dots, x_{i-1}, '0', x_{i+1}, \dots, x_m)$$

– $f = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3$

$$f_{x_1} = \bar{x}_2 x_3 + x_2 \bar{x}_3 + x_2 x_3$$

$$f_{\bar{x}_1} = \bar{x}_2 \bar{x}_3 + x_2 \bar{x}_3 + \bar{x}_2 x_3$$

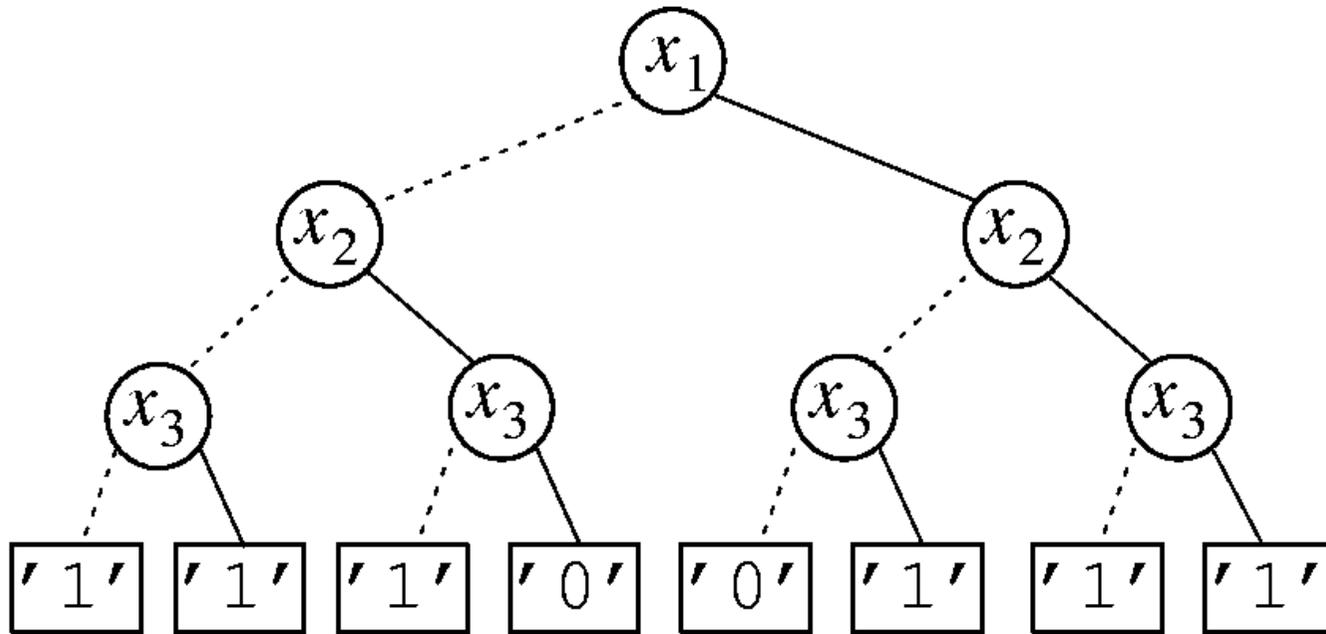
- **Shannon expansion** (already known to Boole) states:

$$f = x_i \cdot f_{x_i} + \bar{x}_i \cdot f_{\bar{x}_i}$$

- A complete expansion can be obtained by successively applying Shannon expansion on all variables of a function until either of the constant function '0' or '1' is reached.

Example: Ordered Binary-Decision Diagram (OBDD)

- The complete Shannon expansion can be visualized as a tree (solid lines correspond to the positive cofactors and dashed lines to negative cofactors).



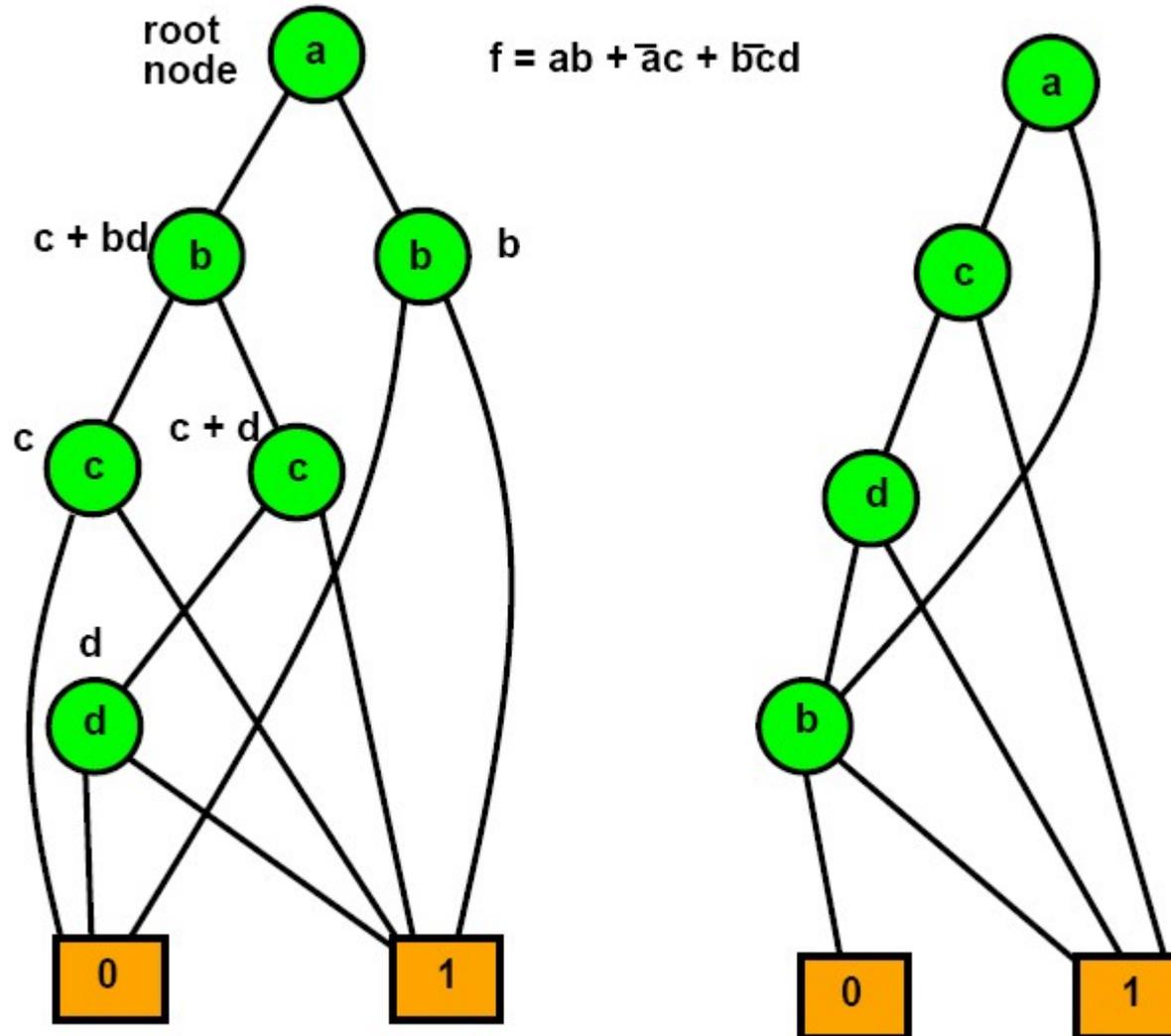
$$f = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3$$

Reduced Ordered BDD (ROBDD)

- ROBDD was proposed by R.E. Bryant in 1986
 - “Graph-Based Algorithms for Boolean Function Manipulation,” IEEE Trans. on Computers, C-35-8, pp. 677—691, Aug. 1986.
- An ROBDD is a Shannon co-factoring tree, except reduced and ordered
 - Reduced
 - **Rule 1**: Any node with identical children is removed.
 - **Rule 2**: Two nodes with isomorphic BDD’s are merged.
 - Ordered
 - Co-factoring variables (splitting variables) always follow the same order $x_{i1} < x_{i2} < \dots < x_{in}$.
 - Canonical
 - Two functions are the same iff their ROBDD’s are equivalent graphs (isomorphic) using the **same** variable ordering.
 - This feature makes ROBDD widely used in logic synthesis and verification

Example ROBDD

- Two different orderings, same function.

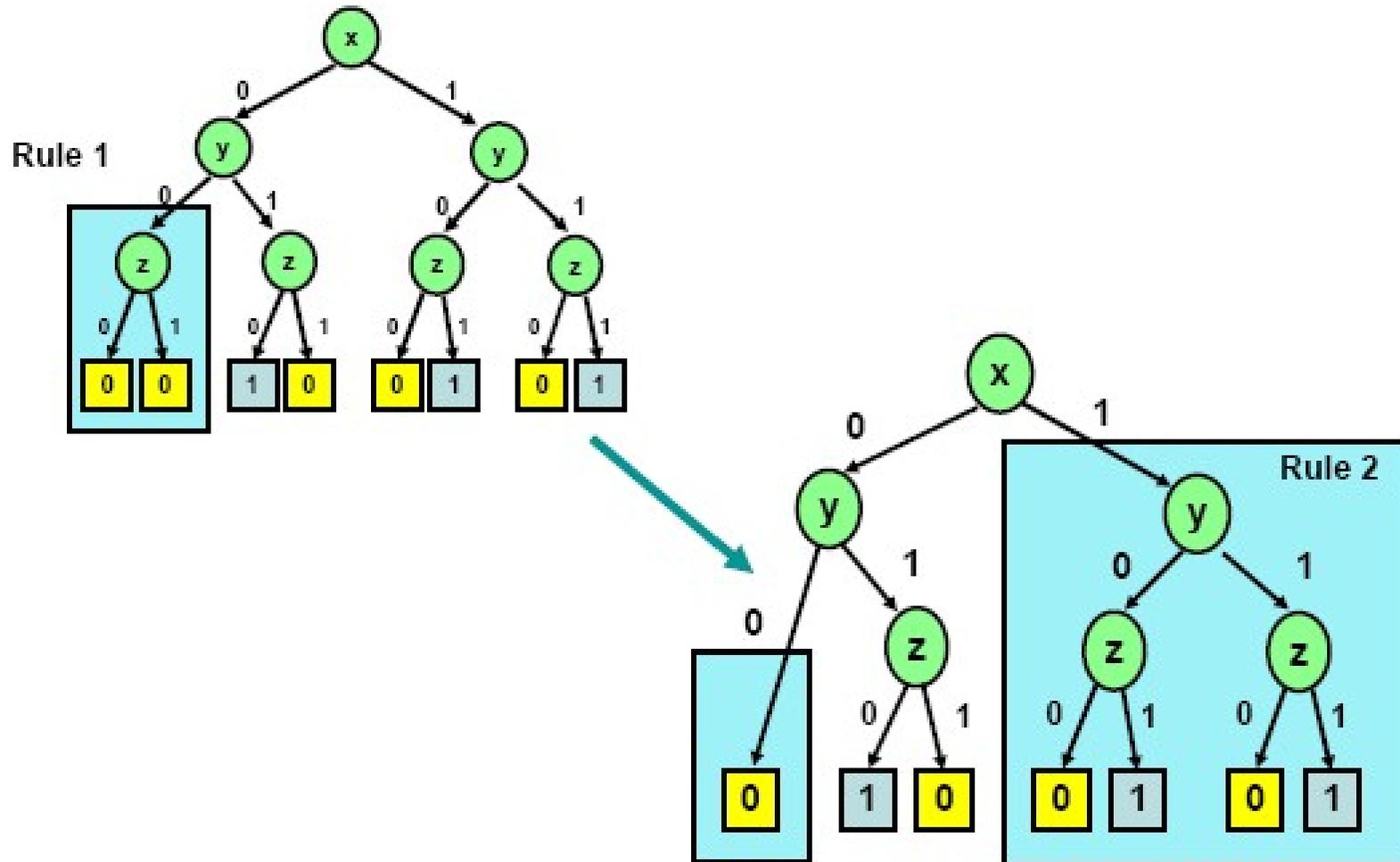


Creating a ROBDD

- An OBDD is a directed tree $G(V,E)$.
- Each vertex $v \in V$ is characterized by an associated variable $\phi(v)$, a high subtree $\eta(v)$ (**high(v)**) and a low subtree $\lambda(v)$ (**low(v)**).
- Procedure to reduce an OBDD:
 - Merge all identical leaf vertices and appropriately redirect their incoming edges; (**Rule 2**)
 - Proceed **from bottom to top**, process all vertices: if two vertices u and v are found for which $\phi(u) = \phi(v)$, $\eta(u) = \eta(v)$, and $\lambda(u) = \lambda(v)$, merge u and v and redirect incoming edges; (**Rule 2**)
 - For vertices v for which $\eta(v) = \lambda(v)$, remove v and redirect its incoming edges to $\eta(v)$. (**Rule 1**)

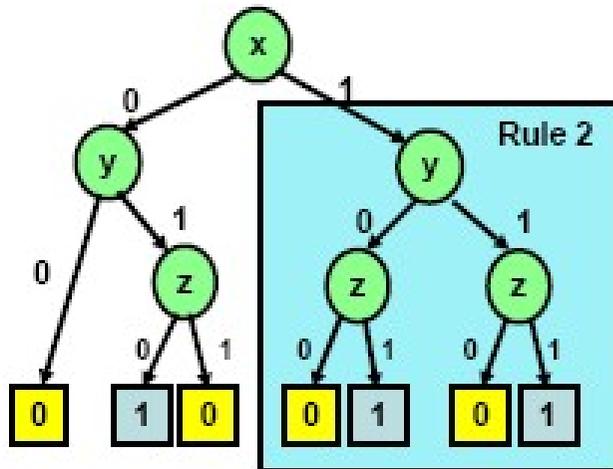
Example: Rule 1

- Rule 1: Any node with identical children is removed

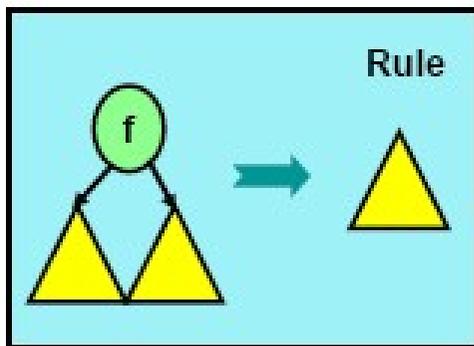
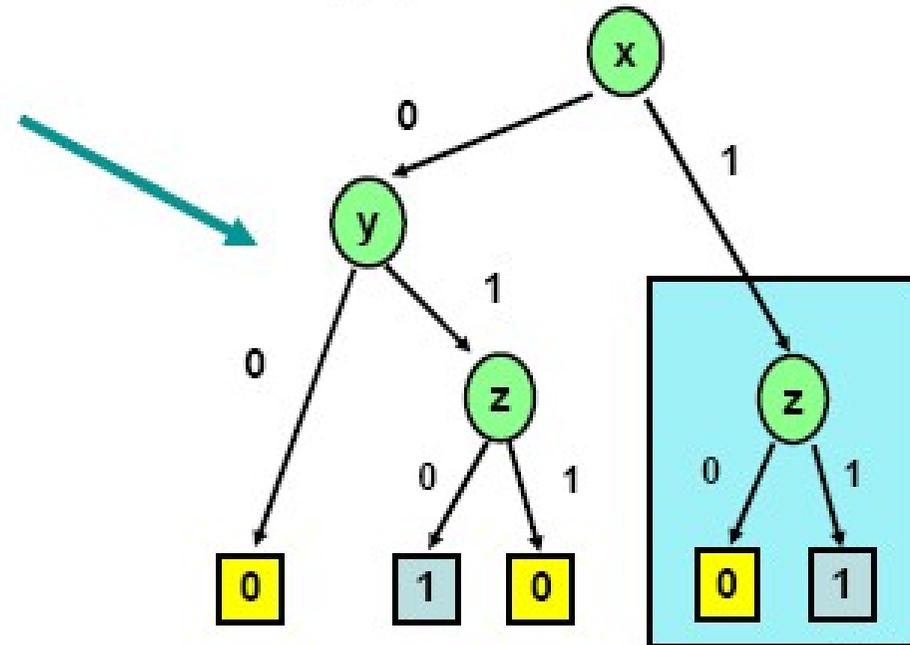


Example: Rule 2

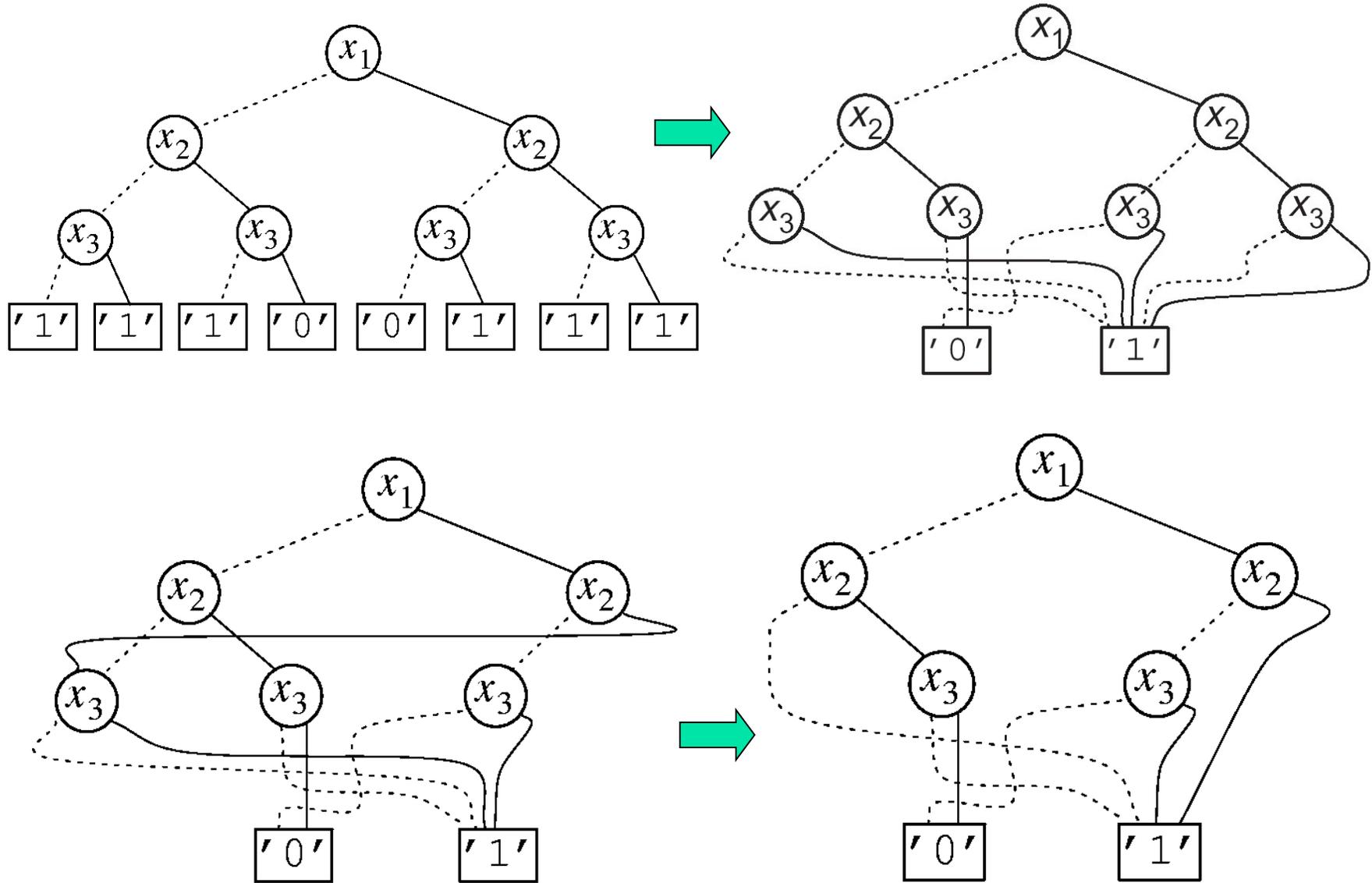
- **Rule 2:** Two nodes with isomorphic BDD's are merged.



No more rules can be applied
→ An ROBDD without isomorphic sub-graphs is achieved



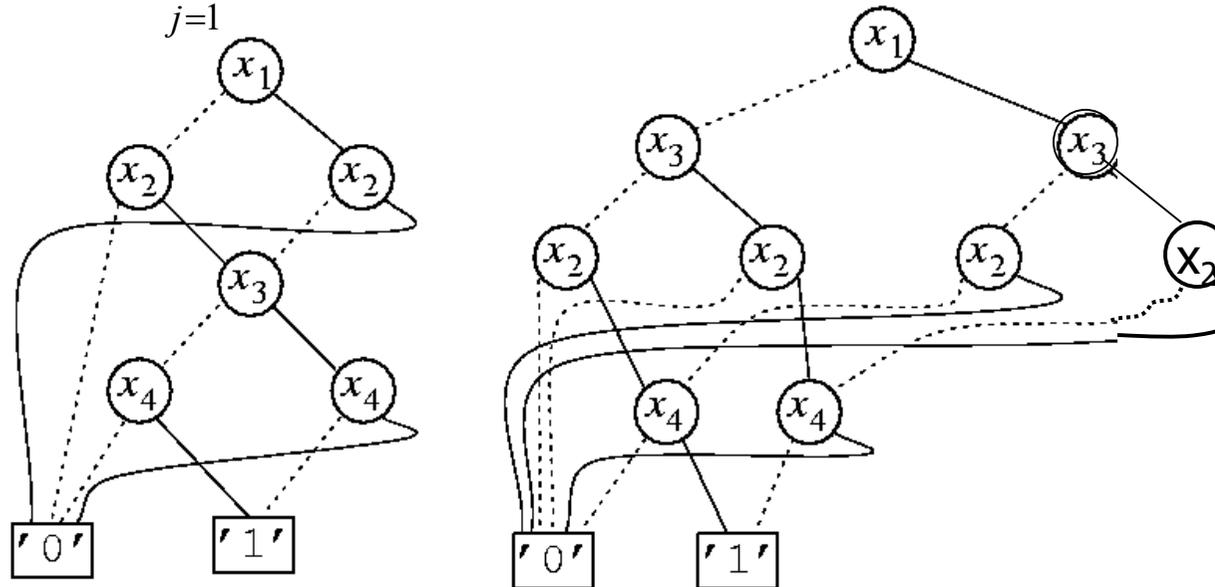
Reduction Example



ROBDD Properties

- The ROBDD is a canonical representation, **given a fixed ordering of the variables.**
- The ROBDD is a compact representation for many Boolean functions used in practice.
- **Variable ordering can greatly affect the size of an ROBDD.**

– E.g.,
$$f = \prod_{j=1}^k x_{2j-1} \oplus x_{2j}$$



A BDD Package

- A BDD package refers to a software program that can manipulate ROBDDs. It has the following properties:
 - Interaction with BDDs takes place through an abstract data type (functionality is independent from the internal representation used).
 - It supports the conversion of some external representation of a Boolean function to the internal ROBDD representation.
 - It can store multiple Boolean functions, sharing all vertices that can be shared.
 - It can **create new functions by combining existing ones** (composition) e.g., $h = f \cdot g$.
 - It can convert the internal representation back to an external one.

BDD Data Structures

- A triple (ϕ, η, λ) uniquely identifies an ROBDD vertex.

```
struct vertex {  
    char * $\phi$ ;  
    struct vertex * $\eta$ , * $\lambda$ ;  
    ...  
}
```

- A **unique table** (implemented by a **hash table**) that stores all triples already processed.

```
struct vertex *old_or_new(char * $\phi$ , struct vertex * $\eta$ , * $\lambda$ )  
{  
    if (“a vertex  $v = (\phi, \eta, \lambda)$  exists”)  
        return  $v$ ;  
    else {  
         $v \leftarrow$  “new vertex pointing at  $(\phi, \eta, \lambda)$ ”;  
        return  $v$ ;  
    }  
}
```

Building an ROBDD

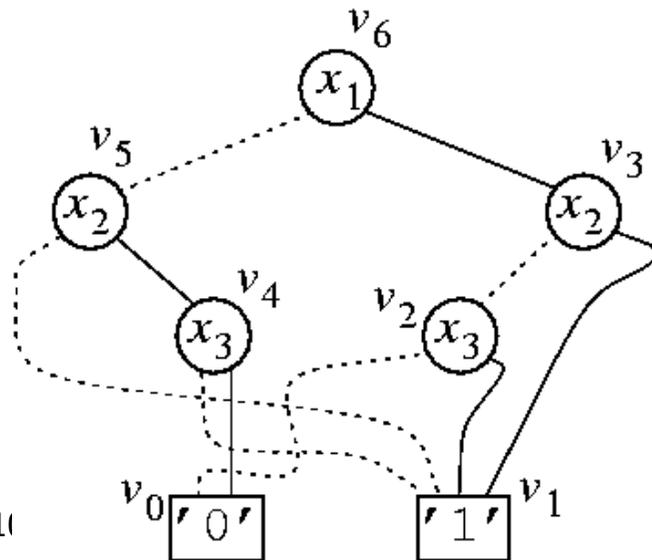
```
struct vertex *robdd_build(struct expr f, int i)
{
    struct vertex * $\eta$ , * $\lambda$ ;
    struct char * $\phi$ ;
    Terminal case
    if (equal(f, '0'))
        return  $v_0$ ;
    else if (equal(f, '1'))
        return  $v_1$ ;
    else {
         $\phi \leftarrow \pi(i)$ ;
         $\eta \leftarrow \text{robdd\_build}(f_\phi, i + 1)$ ;
         $\lambda \leftarrow \text{robdd\_build}(f_{\bar{\phi}}, i + 1)$ ;
        if ( $\eta = \lambda$ )
            return  $\eta$ ;
        else
            return old_or_new( $\phi, \eta, \lambda$ );
    }
}
```

- The procedure directly builds the compact ROBDD structure.
- A simple symbolic computation system is assumed for the derivation of the cofactors.
- $\pi(i)$ gives the i^{th} variable from the top

Example robdd_build

$$\begin{aligned}
 & \text{robdd_build}(\overline{x_1} \cdot \overline{x_3} + \overline{x_2} \cdot x_3 + x_1 \cdot x_2, 1) \\
 & \xrightarrow{\eta} \text{robdd_build}(\overline{x_2} \cdot x_3 + x_2, 2) \\
 & \quad \xrightarrow{\eta} \text{robdd_build}('1', 3) \\
 & \quad \quad v_1 \\
 & \quad \xrightarrow{\lambda} \text{robdd_build}(x_3, 3) \\
 & \quad \quad \xrightarrow{\eta} \text{robdd_build}('1', 4) \\
 & \quad \quad \quad v_1 \\
 & \quad \quad \xrightarrow{\lambda} \text{robdd_build}('0', 4) \\
 & \quad \quad \quad v_0 \\
 & \quad \quad \quad v_2 = (x_3, v_1, v_0) \\
 & \quad \quad \quad v_3 = (x_2, v_1, v_2)
 \end{aligned}$$

$$\begin{aligned}
 & \xrightarrow{\lambda} \text{robdd_build}(\overline{x_3} + \overline{x_2} \cdot x_3, 2) \\
 & \quad \xrightarrow{\eta} \text{robdd_build}(\overline{x_3}, 3) \\
 & \quad \quad \xrightarrow{\eta} \text{robdd_build}('0', 4) \\
 & \quad \quad \quad v_0 \\
 & \quad \quad \xrightarrow{\lambda} \text{robdd_build}('1', 4) \\
 & \quad \quad \quad v_1 \\
 & \quad \quad \quad v_4 = (x_3, v_0, v_1) \\
 & \quad \xrightarrow{\lambda} \text{robdd_build}(\overline{x_3} + x_3, 3) \\
 & \quad \quad \xrightarrow{\eta} \text{robdd_build}('1', 4) \\
 & \quad \quad \quad v_1 \\
 & \quad \quad \xrightarrow{\lambda} \text{robdd_build}('1', 4) \\
 & \quad \quad \quad v_1 \\
 & \quad \quad \quad v_5 = (x_2, v_4, v_1) \\
 & \quad \quad \quad v_6 = (x_1, v_3, v_5)
 \end{aligned}$$



ROBDD Manipulation

- We can create new functions from **existing** functions
- Separate algorithms could be designed for each separate operator on ROBDDs, such as AND, NOR, etc.
- However, the universal **if-then-else** operator '*ite*' is sufficient. $z = \text{ite}(f, g, h)$, z equals g when f is true and equals h otherwise: $z = \text{ite}(f, g, h) = f \cdot g + \bar{f} \cdot h$
- Examples: $z = f \cdot g = \text{ite}(f, g, '0')$
 $z = f + g = \text{ite}(f, '1', g)$
- The *ite* operator is well-suited for a recursive algorithm based on ROBDDs ($\phi(v) = x$):
 $v = \text{ite}(F, G, H) = (x, \text{ite}(F_x, G_x, H_x), \text{ite}(F_{\bar{x}}, G_{\bar{x}}, H_{\bar{x}}))$

The *ite* Algorithm

```
struct vertex *apply_ite(struct vertex *F, *G, *H, int i)
{
    char x;
    struct vertex *η, *λ;

    if (F = v1)
        return G;
    else if (F = v0)
        return H;
    else if (G = v1 && H = v0)
        return F;
    else {
        x ← π(i);
        η ← apply_ite(Fx, Gx, Hx, i + 1);
        λ ← apply_ite(F $\bar{x}$ , G $\bar{x}$ , H $\bar{x}$ , i + 1);
        if (η = λ)
            return η;
        else
            return old_or_new(x, η, λ);
    }
}
```

$$z = \text{ite}(f, g, h) = f \cdot g + \bar{f} \cdot h$$

Comments on the *ite* Algorithm

- The algorithm processes the variables in the order used in the BDD package.
 - $\pi(i)$ gives the i^{th} variable from the top; $\pi^{-1}(x)$ gives the index position of variable x from the top.

- Computation of the restrictions: suppose that F is the root vertex of the function for which F_x should be computed:

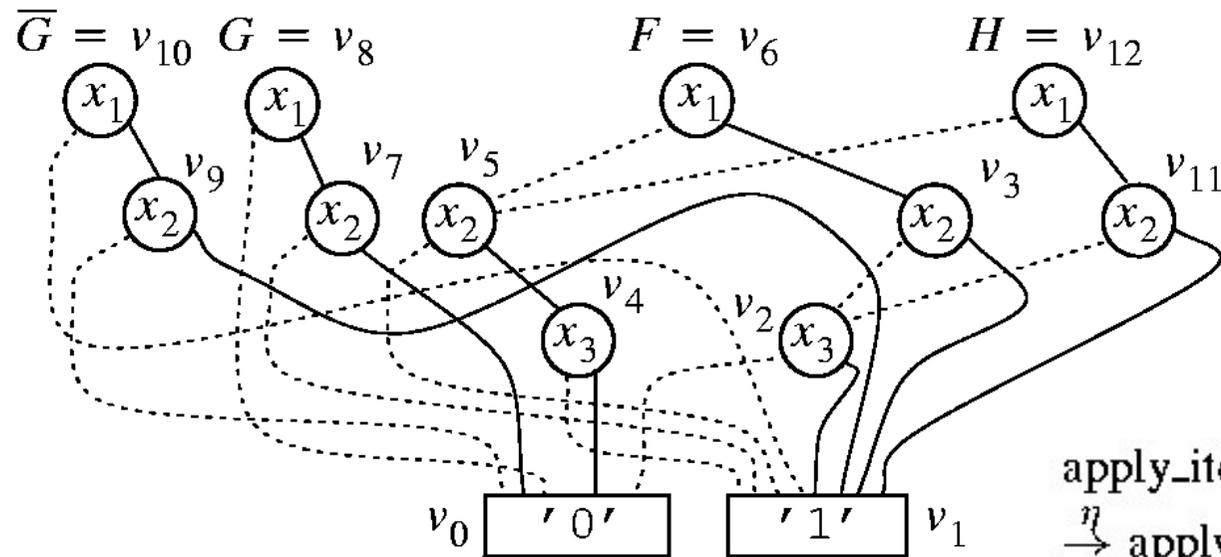
$$F_x = \eta(F) \text{ if } \pi^{-1}(\phi(F)) = i$$

- The calculation of $F_{\bar{x}}$ is done in an analogous way.
- The time complexity of the algorithm is $O(|F|^*|G|^*|H|)$.
 - Each call of *apply_ite* will create at most one ROBDD vertex

ite Operators

Operator	Equivalent <i>ite</i> form
0	0
$f \cdot g$	$ite(f, g, 0)$
$f \cdot g'$	$ite(f, g', 0)$
f	f
$f' \cdot g$	$ite(f, 0, g)$
g	g
$f \oplus g$	$ite(f, g', g)$
$f + g$	$ite(f, 1, g)$
$(f + g)'$	$ite(f, 0, g')$
$(f \oplus g)'$	$ite(f, g, g')$
g'	$ite(g, 0, 1)$
$f + g'$	$ite(f, 1, g')$
f'	$ite(f, 0, 1)$
$f' + g$	$ite(f, g, 1)$
$(f \cdot g)'$	$ite(f, g', 1)$
1	1

ROBDD Example: Computing/Obtaining \bar{G} from G



$$\bar{G} = \text{ite}(G, 0, 1)$$

```

apply_ite(v8, v0, v1, 1)
   $\xrightarrow{\eta}$  apply_ite(v7, v0, v1, 2)
     $\xrightarrow{\eta}$  apply_ite(v0, v0, v1, 3)
      v1
     $\xrightarrow{\lambda}$  apply_ite(v1, v0, v1, 3)
      v0
    v9 = (x2, v1, v0)
   $\xrightarrow{\lambda}$  apply_ite(v0, v0, v1, 2)
    v1
  v10 = (x1, v9, v1)
  
```


Composition

- The composite problem is
 - the ROBDDs of two functions f and g are known/pre-computed
 - the output of g is connected to an input of f
 - compute the ROBDD of the composed function h , where

$$h = f(x_1, \dots, x_{i-1}, g, x_{i+1}, \dots, x_n).$$

- Using **Shannon expansion**, one finds that

$$h = g \cdot f_{x_i} + \bar{g} \cdot f_{\bar{x}_i} = \text{ite}(g, f_{x_i}, f_{\bar{x}_i})$$

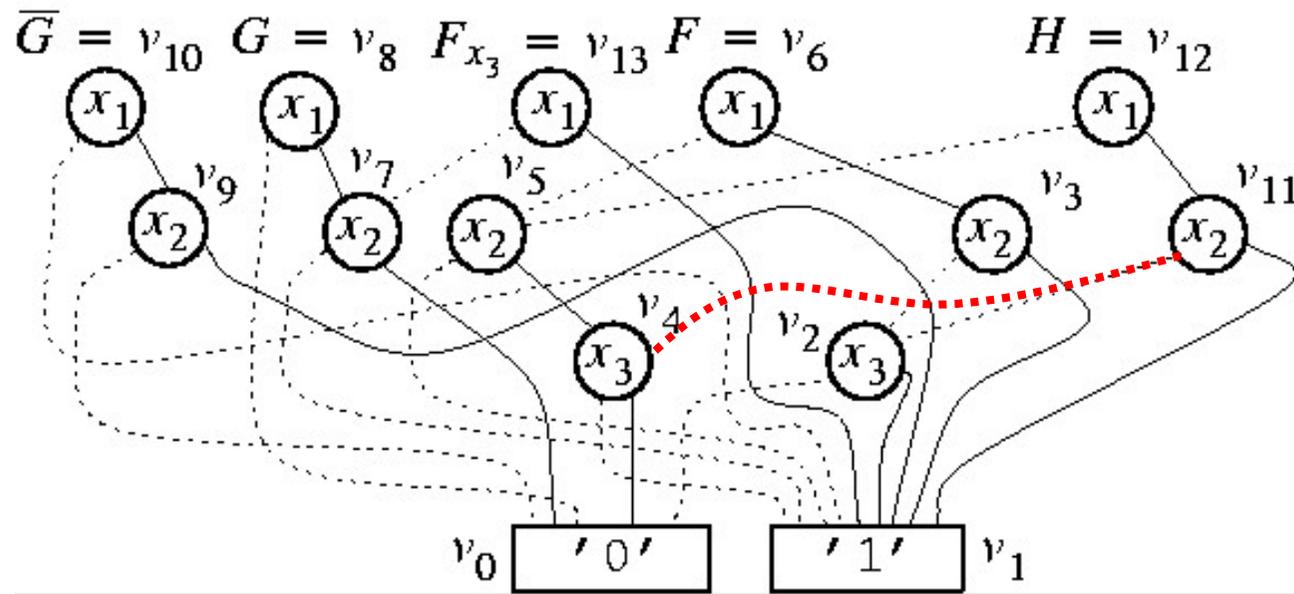
- Now, the restrictions/cofactors have to be calculated by dedicated algorithms.

Restriction: Positive Cofactor

```
struct vertex *positive_cofactor(struct vertex *F, int r, i)
{
    char x;
    struct vertex *η, *λ;

    if (F = v1)
        return v1;
    else if (F = v0)
        return v0;
    else if (r = i)
        return η(F);
    else {
        x ← π(i);
        η ← positive_cofactor(Fx, r, i + 1);
        λ ← positive_cofactor(F $\bar{x}$ , r, i + 1);
        if (η = λ)
            return η;
        else
            return old_or_new(x, η, λ);
    }
}
```

Positive Cofactor Example: Computing F_{x_3}



positive_cofactor($v_6, 3, 1$)

$\xrightarrow{\eta}$ positive_cofactor($v_3, 3, 2$)

$\xrightarrow{\eta}$ positive_cofactor($v_1, 3, 3$)

v_1

$\xrightarrow{\lambda}$ positive_cofactor($v_2, 3, 3$)

v_1

v_1

$\xrightarrow{\lambda}$ positive_cofactor($v_5, 3, 2$)

$\xrightarrow{\eta}$ positive_cofactor($v_4, 3, 3$)

v_0

$\xrightarrow{\lambda}$ positive_cofactor($v_1, 3, 3$)

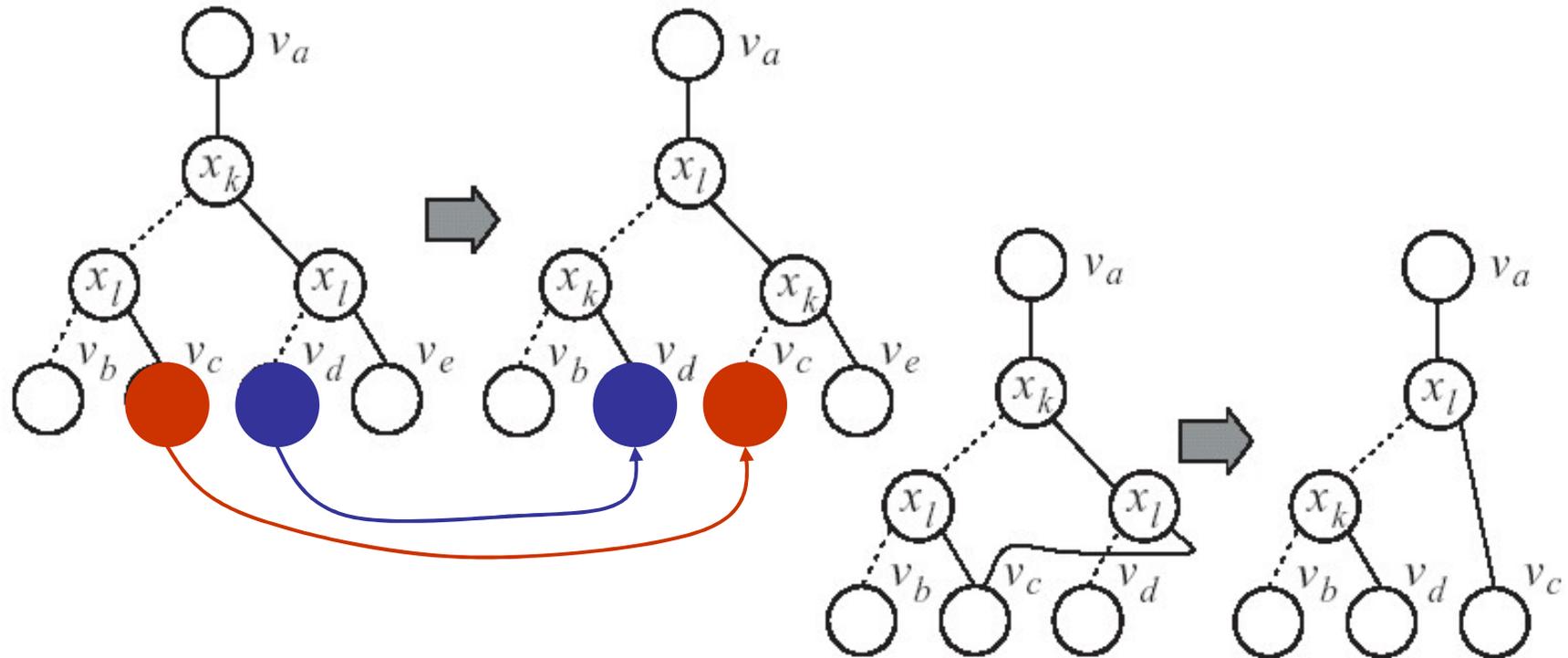
v_1

$v_7 = (x_2, v_0, v_1)$

$v_{13} = (x_1, v_1, v_7)$

Variable Ordering

- Reorder adjacent variables only has a local effect on the ROBDD.



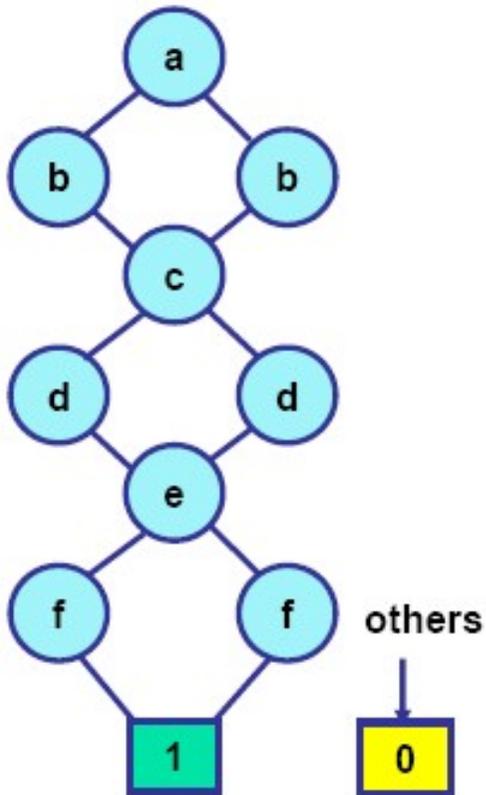
The Influence of Variable Ordering

- Size of BDD
 - Can vary from linear to exponential in the number of the variables, depending on the ordering
- Hard-to-build BDD
 - Data path components (e.g., multipliers) cannot be represented in polynomial space, regardless of the variable ordering
- Finding the ordering that minimizes the ROBDD size for some function is intractable (co-NPC)
 - The optimal ordering may change as ROBDDs are being manipulated.

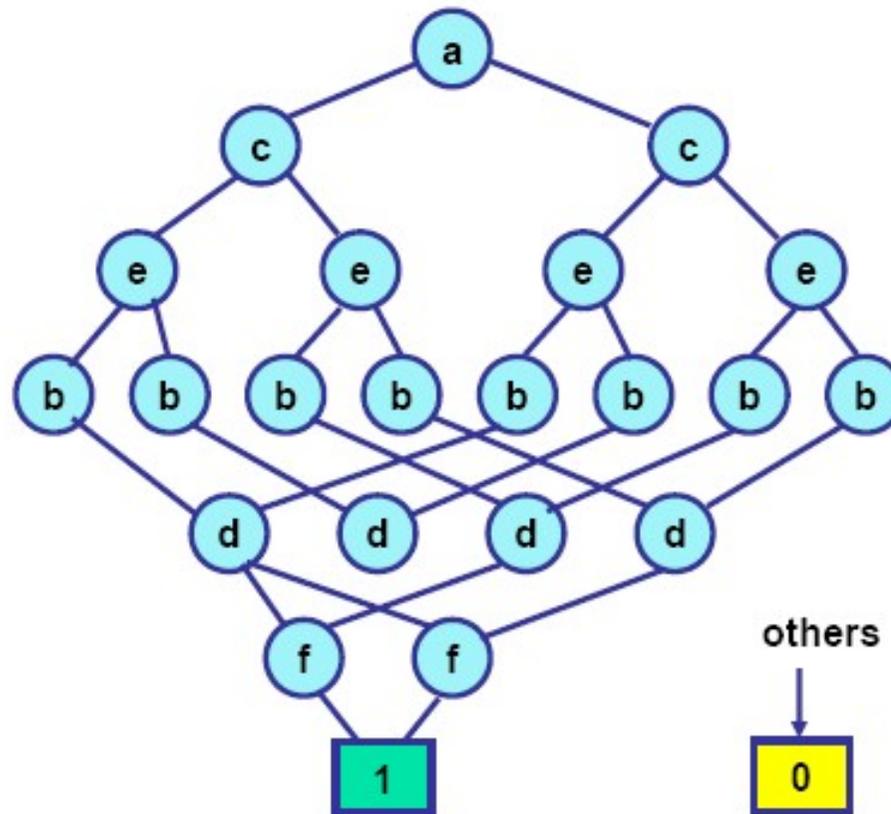
Example on Variable Ordering

$$z = (a \oplus b) \cdot (c \oplus d) \cdot (e \oplus f)$$

good order



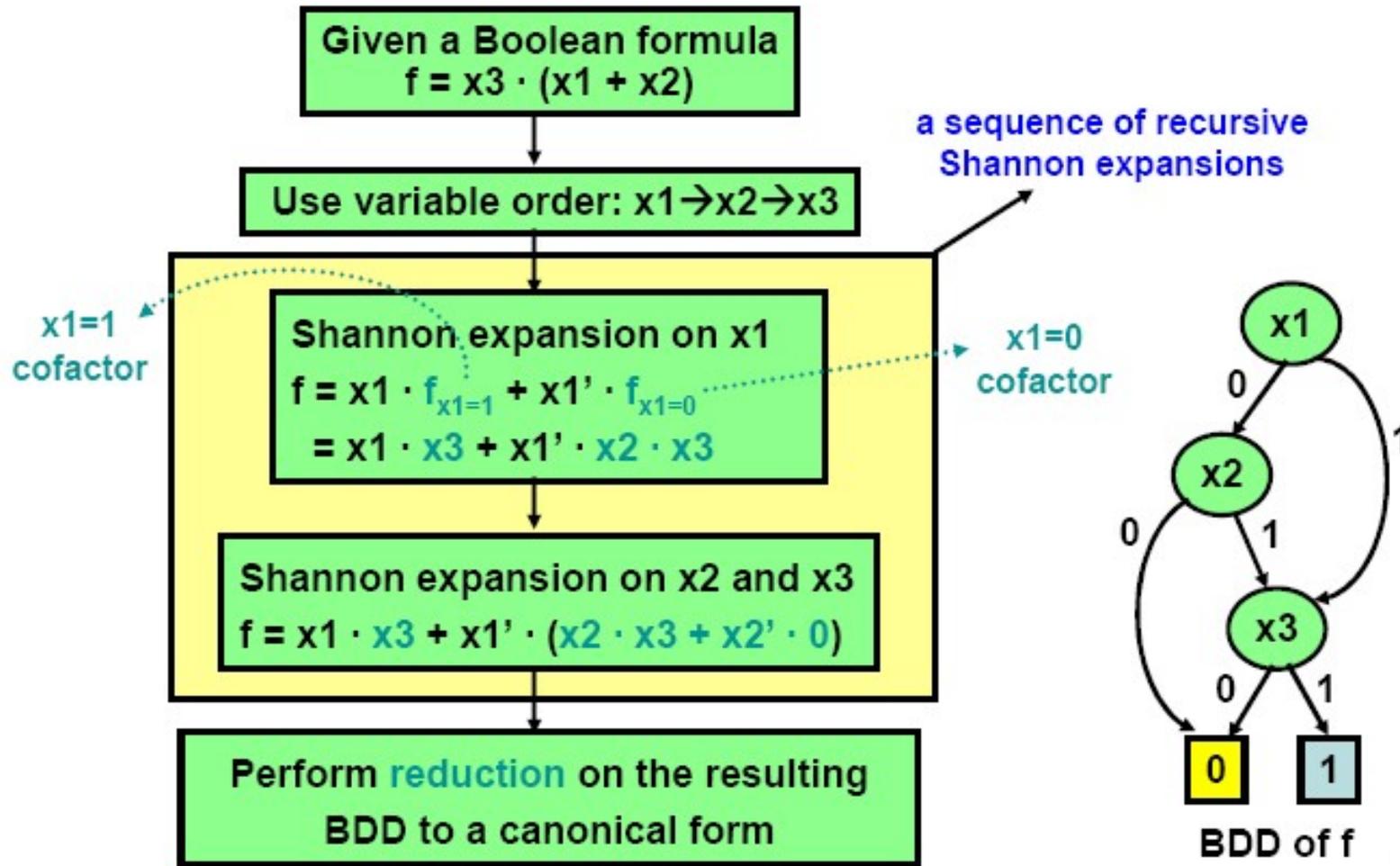
bad order



Heuristic of Variable Ordering

- Heuristics of ordering
 - Put variables that influence most on the top of BDD.
 - Minimize the distance between strongly related variables.
 - e.g., $x_1x_2 + x_2x_3 + x_3x_4$
 $x_1 > x_2 > x_3 > x_4$ is better than $x_1 > x_4 > x_2 > x_3$
- An ROBDD package will try to reorder the variables at distinct moments.
 - It could move one variable to the top and back to the bottom and remember the best position. It could then repeat the procedure for the other variables.
- Another “invisible” feature of an ROBDD package is garbage collection.
 - Automatically remove vertices that are no longer required

Summary: From Boolean Expression to BDD



BDD Features

- Strengths
 - BDD is a compact representation for Boolean functions
 - Canonical, given a fixed variable ordering
 - Polynomial time in BDD size for many Boolean operations
- Weaknesses
 - In the worst case, the size of a BDD is $O(2^n)$ for n -input Boolean functions

ROBDDs and Satisfiability

- A Boolean function is **satisfiable** if an assignment to its variables exists for which the function becomes '1'
- Any Boolean function whose ROBDD is unequal to '0' is satisfiable.
- Suppose that choosing a Boolean variable x_i to be '1' costs c_i . Then, the **minimum-cost satisfiability** problem asks to minimize:
$$\sum_{i=1}^n c_i \mu(x_i)$$

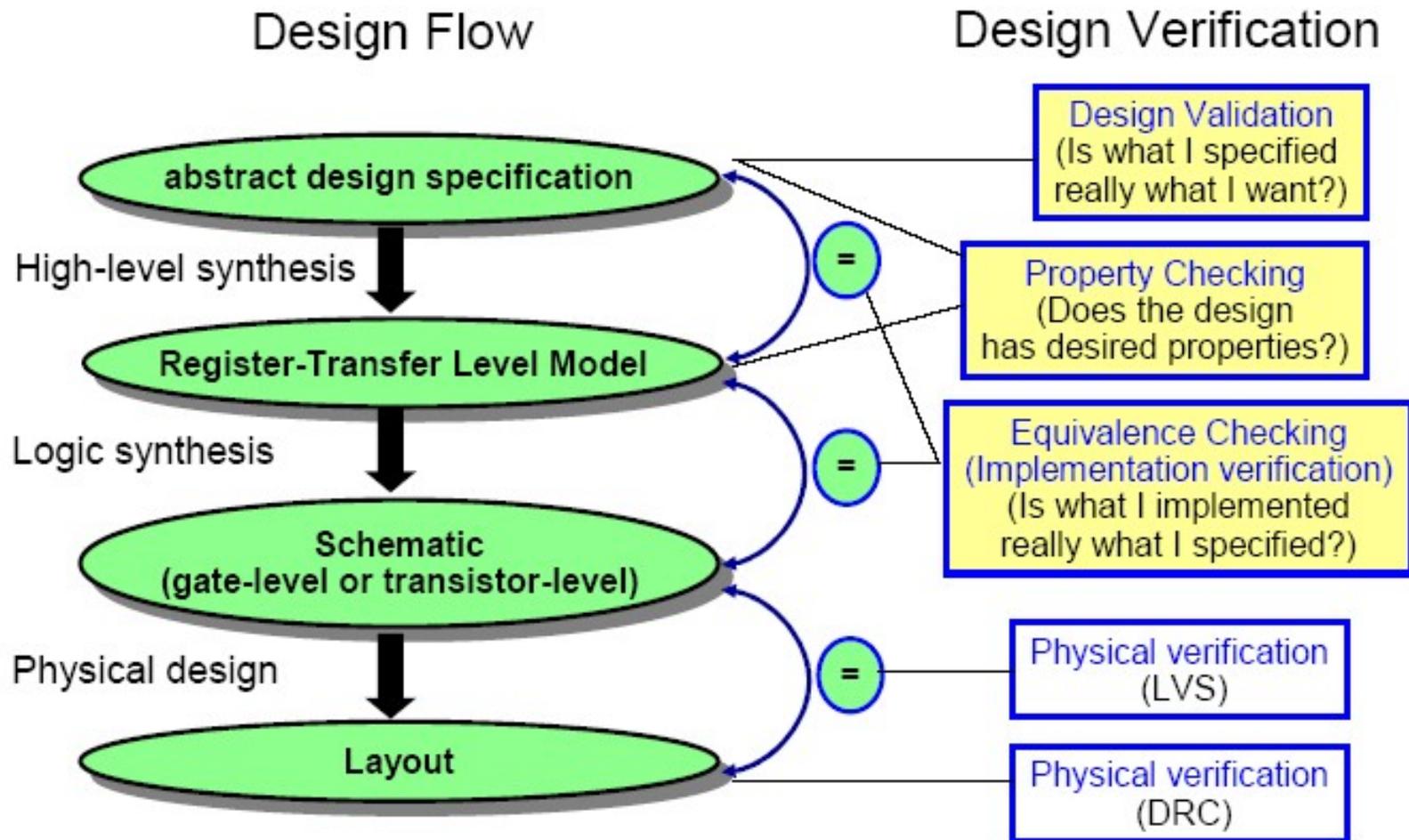
where $\mu(x_i) = 1$ when $x_i = '1'$ and $\mu(x_i) = 0$ when $x_i = '0'$.

- Solving minimum-cost satisfiability amounts to computing the shortest path in an ROBDD, which can be solved in linear time (since an ROBDD is a DAG).
 - **Weights:** $w(v, \eta(v)) = c_i$, $w(v, \lambda(v)) = 0$, variable $x_i = \phi(v)$.

ROBDD Application: Functional Verification

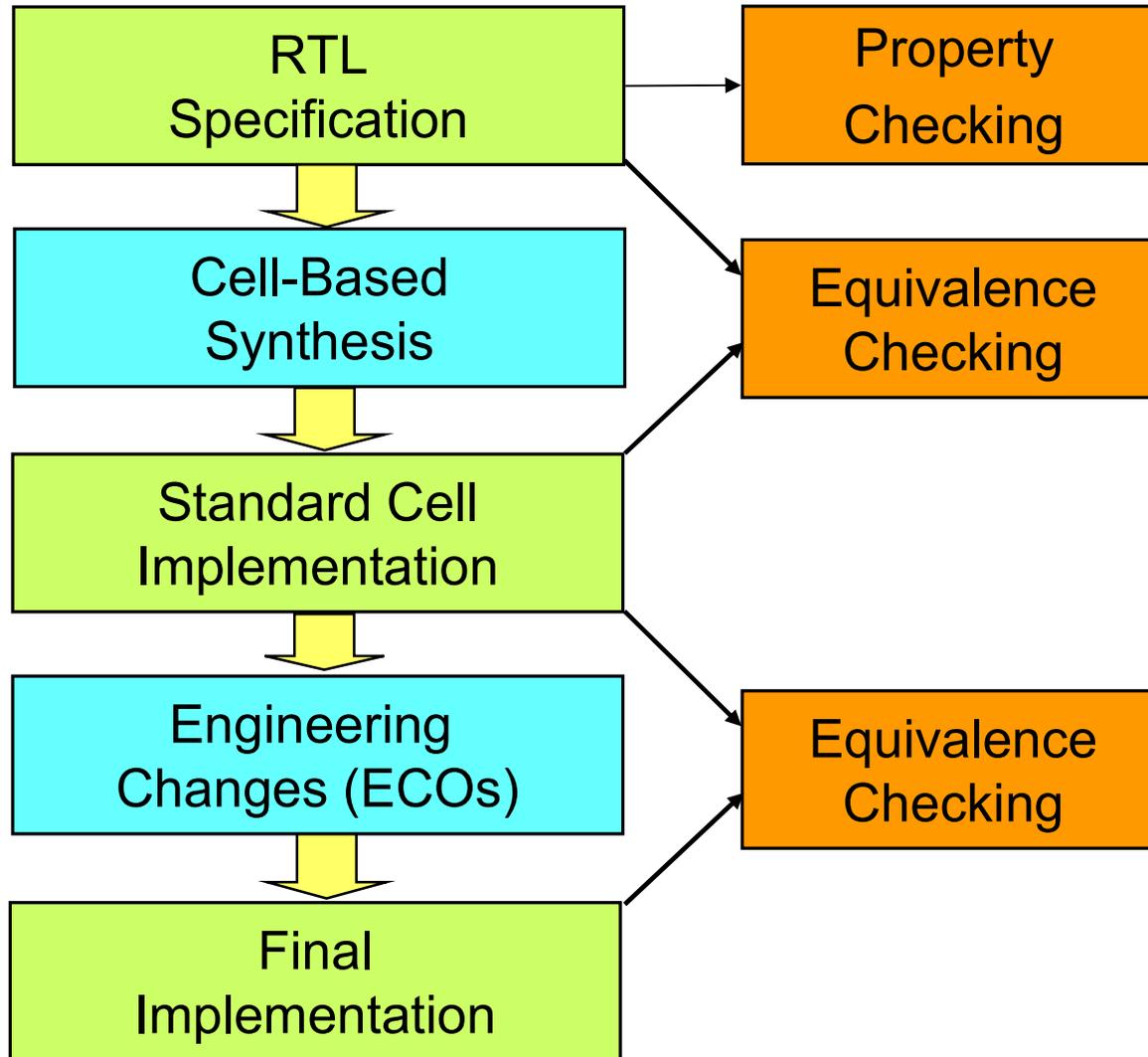
- Design creation process can be viewed as a series of transformation, from abstract **specification** all the way to **layout**.
- Functional verification is to compare a specification f to an implementation g .
 - They can both be represented by ROBDDs (F resp. G).
 - Can be performed on the same level or different levels
- In case of a fully specified function, verification is trivial (pointer comparison) because of the **strong canonicity** of the ROBDD data structure.
 - **Strong canonicity**: the representations to identical functions are the same.
- If there is a dc-set, use two functions f and d . The implementation g is correct when $d + f \cdot g + \bar{f} \cdot \bar{g}$ is a **tautology** (the expression evaluates to '1').

The Roles of Functional Verification



LVS: layout vs. schematic check, DRC: design rule check

Application of EC in ASIC Designs

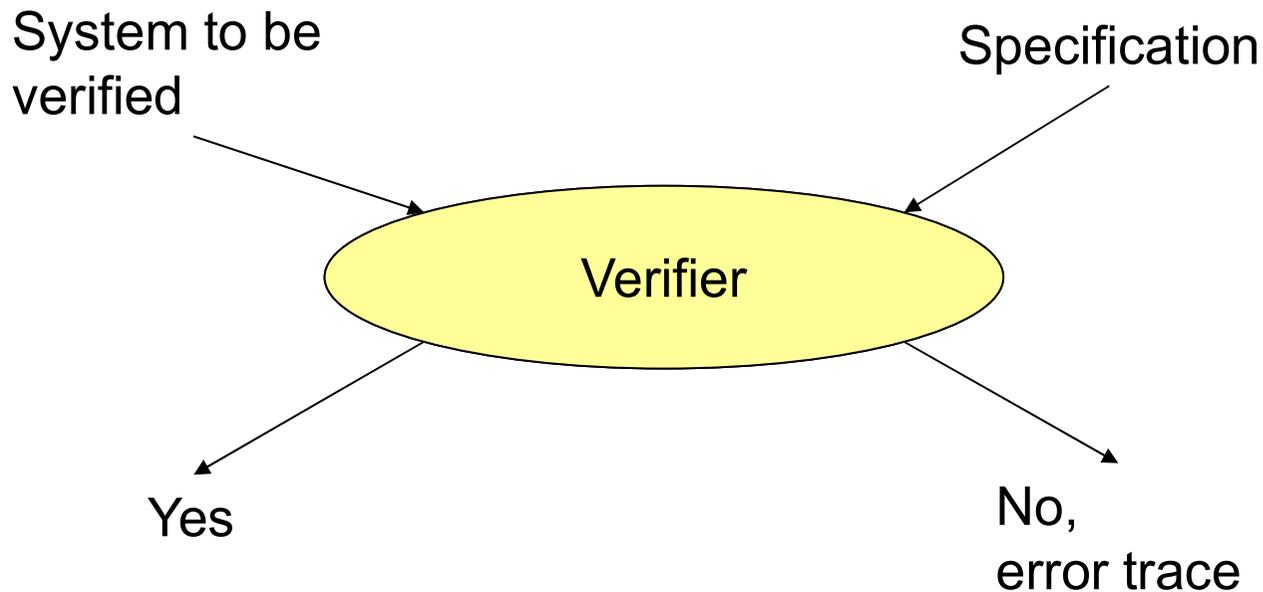


Verification

- Design Verification
 - Functional Verification
 - Property checking in system level
 - **PSPACE-complete** ($P \subseteq NP \subseteq PSPACE$)
 - The hardest problems in **PSPACE** are the **PSPACE-complete** problems
 - Equivalence checking in RTL and gate level
 - **PSPACE-complete**
 - Physical Verification
 - DRC (design rule check) and LVS (layout vs. schematic check) in layout level
 - **Tractable**
- Manufacture Verification
 - Testing
 - NP-complete
- “Verification” often refers to functional verification

Informal vs. Formal Verification

- Informal verification
 - Functional simulation aiming at locating bugs
 - Incomplete
 - Show existence of bugs, but not absence of bugs
 - Low complexity
- Formal verification
 - Mathematical proof of design correctness
 - Complete
 - Show both existence and absence of bugs
 - High complexity



Functional Verification Comparison

- Simulation (software)
 - Input stimuli derived manually or automatically by constraints solvers
 - **Incomplete** (i.e., may fail to catch bugs)
 - Very **time-consuming**, especially when at lower abstraction level such as the gate or transistor level
 - Still the most popular way for **design validation**
- Emulation (hardware)
 - e.g. FPGA-based emulation system, or emulation system based on a **massively parallel machine** (e.g., with 8 boards, each having 128 processors)
 - **2 to 3 orders** of magnitude faster than software simulation
 - **Costly** and might not be very easy-to-use
- Formal verification
 - A relatively new paradigm for **property checking** and **equivalence checking**
 - Requires **no input stimuli**
 - Performs **exhaustive proof** through rigorous **logical reasoning**