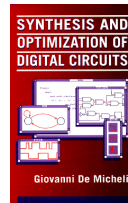


# *Boolean Methods for Multi-level Logic Synthesis*

**Giovanni De Micheli**  
***Integrated Systems Centre***  
***EPF Lausanne***



---

This presentation can be used for non-commercial purposes as long as this note and the copyright footers are not removed

© Giovanni De Micheli – All rights reserved

# Module 1

---

## ◆ Objectives

- ▲ What are Boolean methods
- ▲ How to compute *don't care* conditions
  - ▼ Controllability
  - ▼ Observability
- ▲ Boolean transformations

# Boolean methods

---

- ◆ **Exploit Boolean properties of logic functions**
- ◆ **Use *don't care* conditions**
- ◆ **More complex algorithms**
  - ▲ **Potentially better solutions**
  - ▲ **Harder to reverse the transformations**
- ◆ **Used within most synthesis tools**

# External *don't care* conditions

---

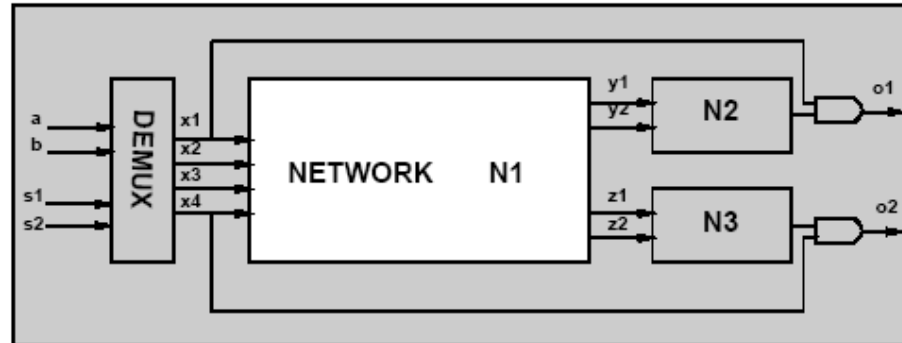
## ◆ **Controllability** *don't care* set $CDC_{in}$

- ▲ Input patterns never produced by the environment at the network's input

## ◆ **Observability** *don't care* set $ODC_{out}$

- ▲ Input patterns representing conditions when an output is not observed by the environment
- ▲ Relative to each output
- ▲ Vector notation

# Example



- Inputs driven by a de-multiplexer.
- $CDC_{in} = x'_1x'_2x'_3x'_4 + x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4$ .
- Outputs observed when  $\begin{bmatrix} x_1 \\ x_4 \end{bmatrix} = \mathbf{1}$

$$ODC_{out} = \begin{bmatrix} x'_1 \\ x'_1 \\ x'_4 \\ x'_4 \end{bmatrix}$$

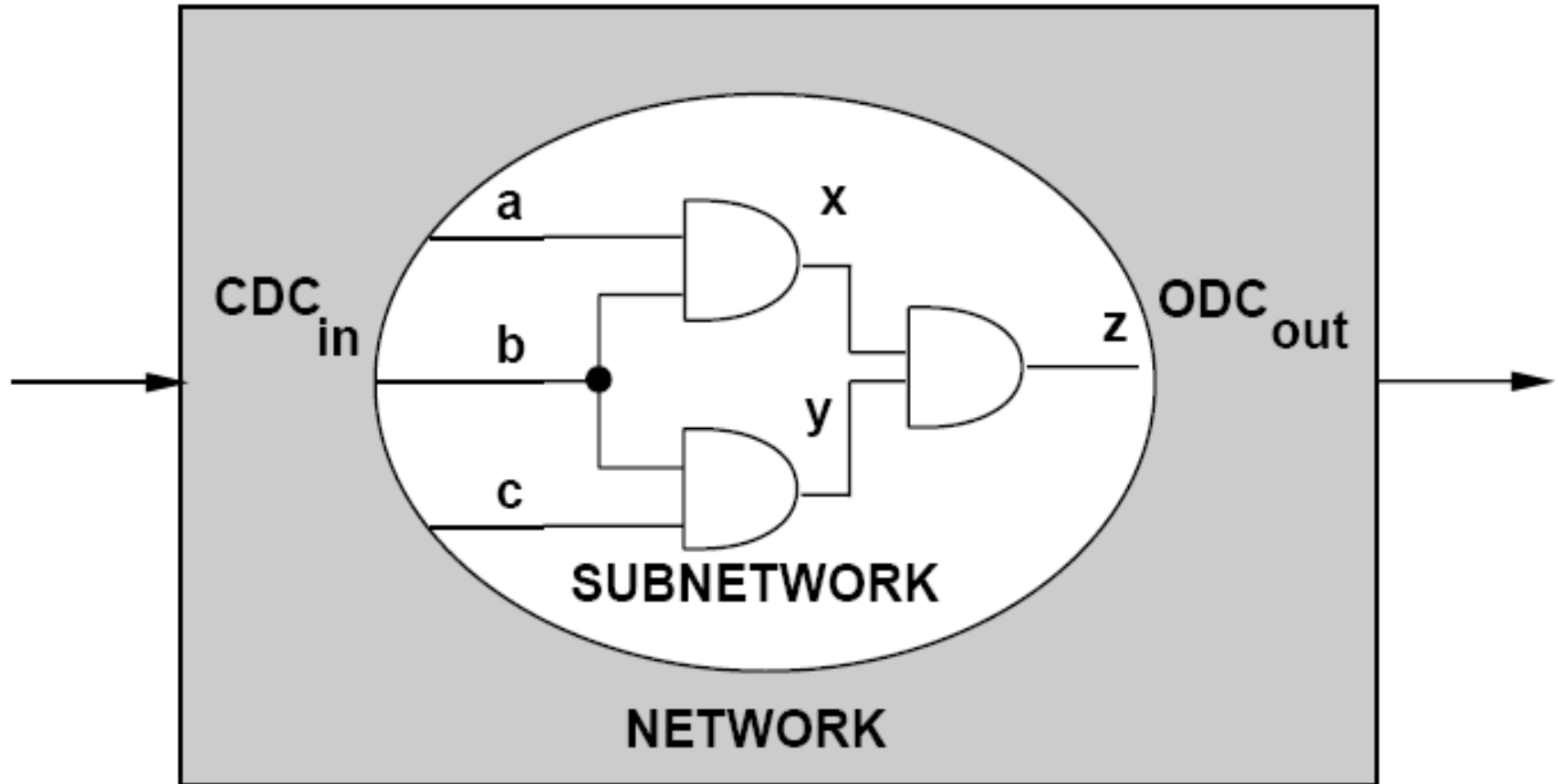
# Overall external *don't care* set

---

- ◆ Sum the controllability *don't cares* to each entry of the observability *don't care* set vector

$$\mathbf{DC}_{ext} = \mathbf{CDC}_{in} + \mathbf{ODC}_{out} = \begin{bmatrix} x'_1 + x_2 + x_3 + x_4 \\ x'_1 + x_2 + x_3 + x_4 \\ x'_4 + x_2 + x_3 + x_1 \\ x'_4 + x_2 + x_3 + x_1 \end{bmatrix}$$

# Internal *don't care* conditions



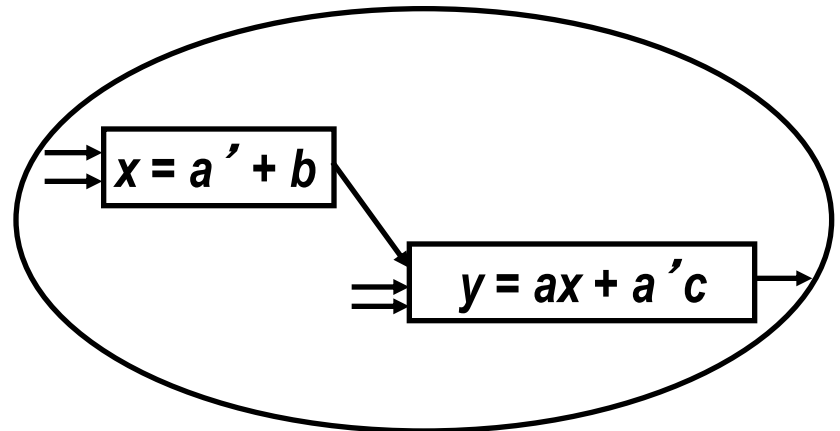
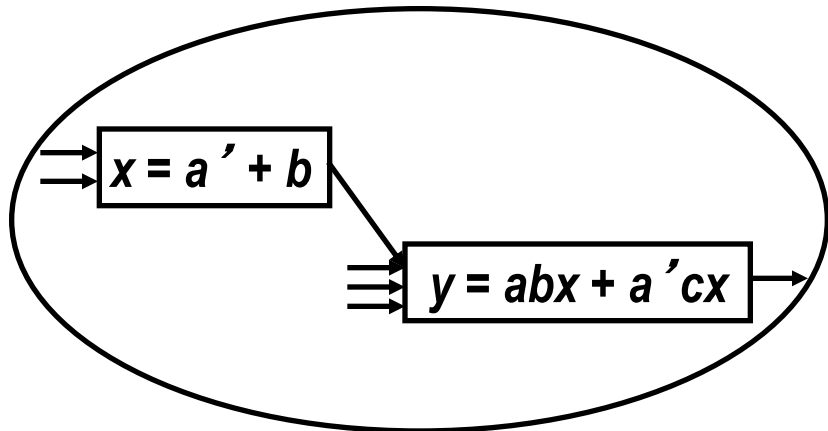
# Internal *don't care* conditions

---

- ◆ Induced by the network structure
- ◆ Controllability *don't care* conditions:
  - ▲ Patterns never produced at the inputs of a sub-network
- ◆ Observability *don't care* conditions
  - ▲ Patterns such that the outputs of a sub-network are not observed



# Example of optimization with *don't cares*



- ◆ CDC of  $y$  includes  $ab'x + a'x'$
- ◆ Minimize  $f_y$  to obtain:  $g_y = ax + a'c$

# Satisfiability *don't care* conditions

---

◆ Invariant of the network:

$$\mathbf{x} = \mathbf{f}_x \rightarrow \mathbf{x} \neq \mathbf{f}_x \subseteq \text{SDC}$$

$$\text{SDC} = \sum_{\text{all internal nodes}} \mathbf{x} \oplus \mathbf{f}_x$$

◆ Useful to compute controllability don't cares

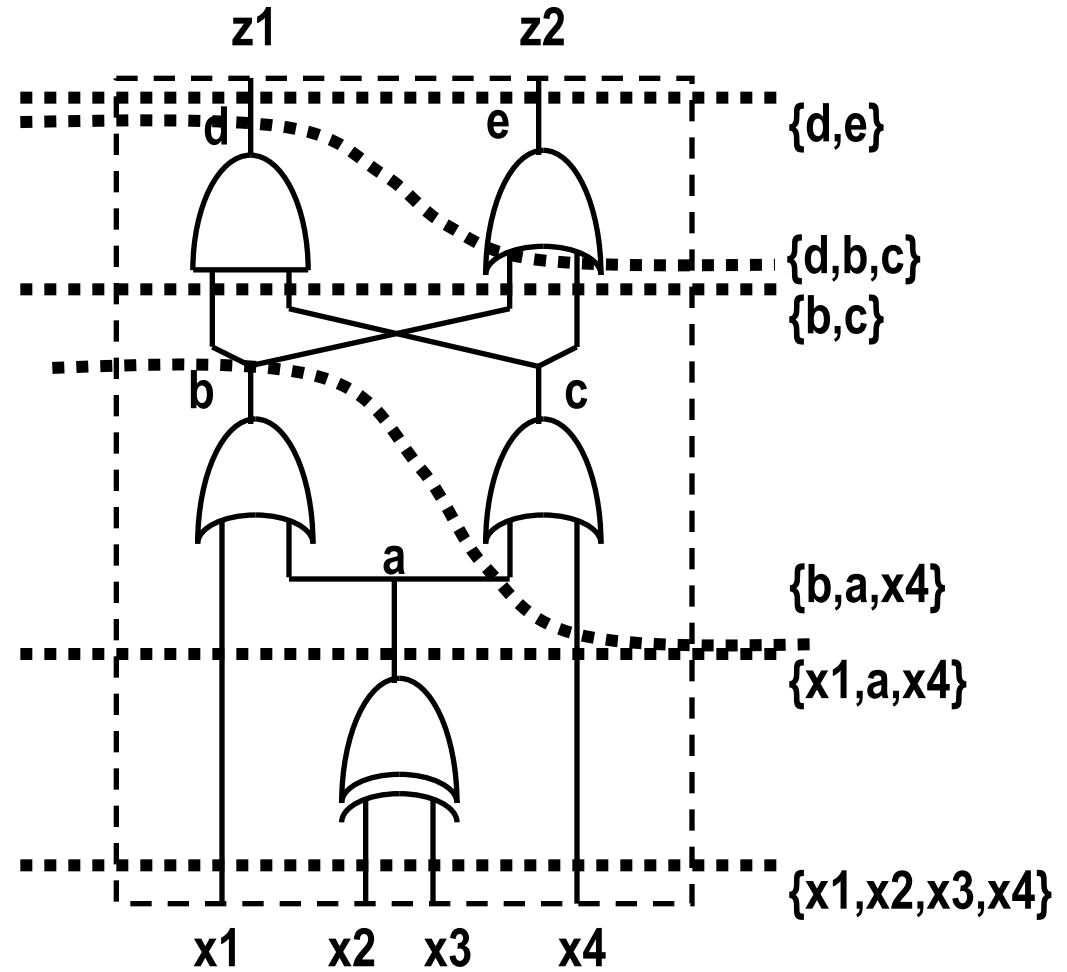
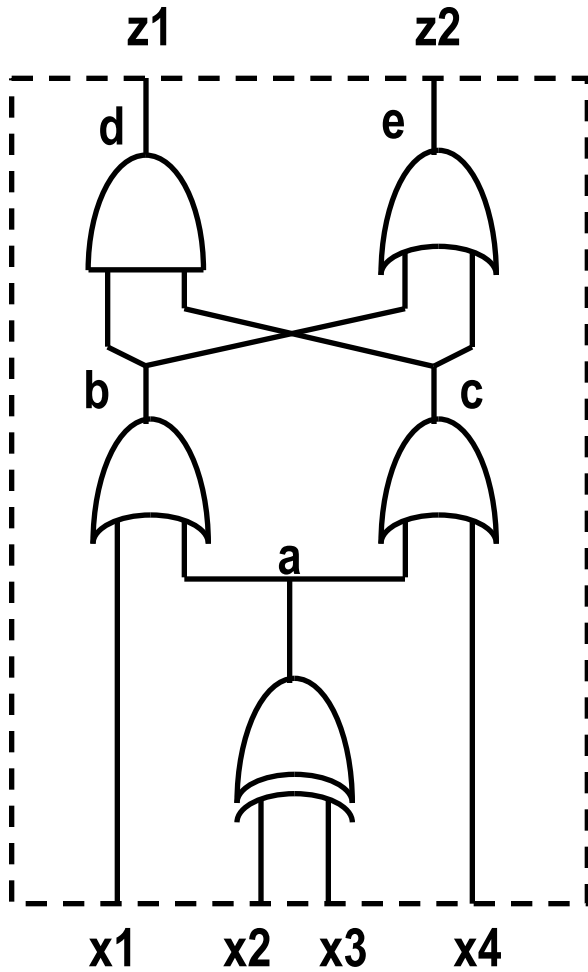
# CDC Computation

---

## ◆ Method 1: Network traversal algorithm

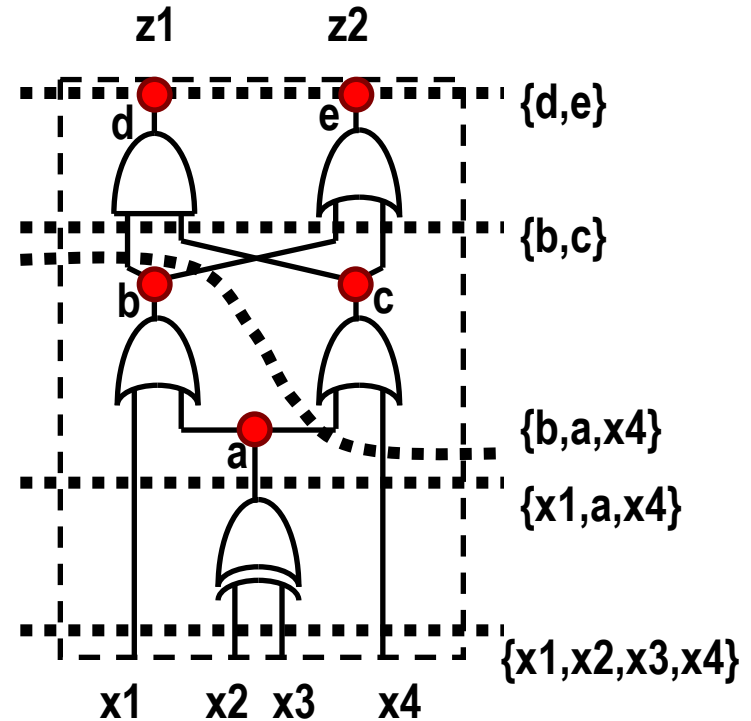
- ▲ Consider initial **CDC** = **CDC**<sub>in</sub> at the primary inputs
- ▲ Consider different cutsets moving through the network from inputs to outputs
- ▲ As the cutset moves forward
  - ▼ Consider **SDC** contribution of the newly considered block
  - ▼ Remove unneeded variables by consensus

# Example



# Example

- ◆ Assume  $CDC_{in} = x_1' x_4'$
- ◆ Select vertex  $v_a$ 
  - ▲ Contribution of  $v_a$  to  $CDC_{cut} = a \oplus (x_2 \oplus x_3)$
  - ▲ Updated  $CDC_{cut} = x_1' x_4' + a \oplus (x_2 \oplus x_3)$
  - ▲ Drop variables  $D = \{x_2, x_3\}$  by consensus:
  - ▲  $CDC_{cut} = x_1' x_4'$
- ◆ Select vertex  $v_b$ 
  - ▲ Contribution to  $CDC_{cut}$ :  $b \oplus (x_1 + a)$ .
    - ▼ Updated  $CDC_{cut} = x_1' x_4' + b \oplus (x_1 + a)$
  - ▲ Drop variables  $x_1$  by consensus:
    - ▼  $CDC_{cut} = b' x_4' + b' a$
- ◆ ...
- ◆  $CDC_{out} = e' = z_2'$



# CDC Computation

```
CONTROLLABILITY( $G_n(V,E)$ ,  $CDC_{in}$ ) {  
   $C = V$ ;  
   $CDC_{cut} = CDC_{in}$ ;  
  foreach vertex  $v_x \in V$  in topological order {  
     $C = C \cup v_x$ ;  
     $CDC_{cut} = CDC_{cut} + f_x \oplus x$ ;  
     $D = \{v \in C \text{ s.t. all direct successors of } v \text{ are in } C\}$   
    foreach vertex  $v_y \in D$   
       $CDC_{cut} = C_y(CDC_{cut})$ ;  
     $C = C - D$ ;  
  };  
   $CDC_{out} = CDC_{cut}$ ;  
}
```

# CDC Computation

---

- ◆ Method 2: **range** or **image** computation
- ◆ Consider the function **f** expressing the behavior of the cutset variables in terms of primary inputs
- ◆ **CDC<sub>cut</sub>** is the complement of the **range** of **f** when **CDC<sub>in</sub> = 0**
- ◆ **CDC<sub>cut</sub>** is the complement of the **image** of **(CDC<sub>in</sub>)'** under **f**
- ◆ The range and image can be computed recursively
  - ▲ Terminal case: scalar function
  - ▲ The range of **y = f( x )** is **y + y'** (any value)  
unless **f** ( or **f'** ) is a tautology and the range is **y** ( or **y'** )

# Example

◆  $range(f) = d \ range((b+c)|_{d=bc=1}) + d' \ range((b+c)|_{d=bc=0})$

◆ When  $d = 1$ , then  $bc = 1 \rightarrow b + c = 1$  is TAUTOLOGY

◆ If I choose 1 as top entry in output vector:

▲ the bottom entry is also 1.

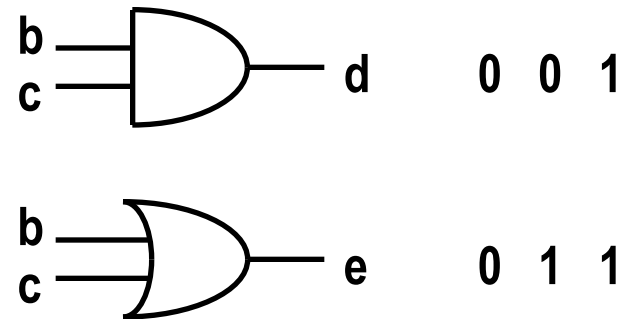
$$\begin{bmatrix} 1 \\ ? \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

◆ When  $d = 0$ , then  $bc = 0 \rightarrow b+c = \{0,1\}$

◆ If I choose 0 as top entry in output vector:

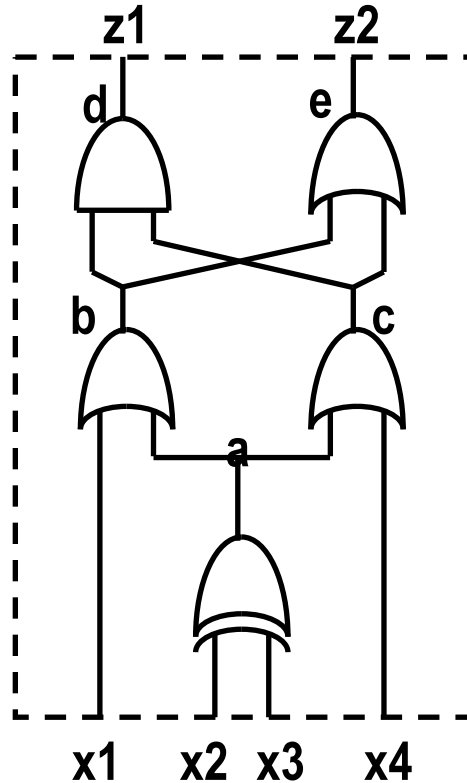
▲ The bottom entry can be either 0 or 1.

◆  $range(f) = de + d' (e + e') = de + d' = d' + e$



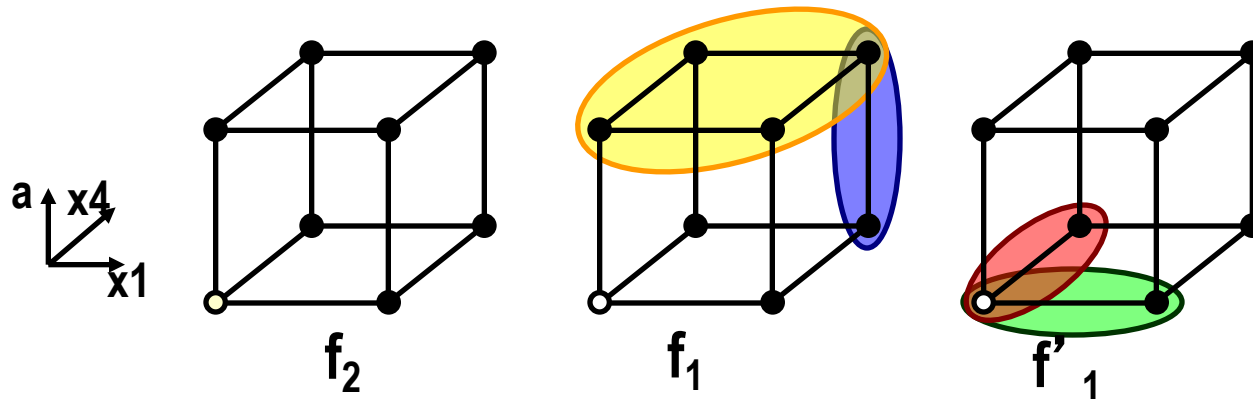


# Example



$$f = \begin{bmatrix} f^1 \\ f^2 \end{bmatrix} = \begin{bmatrix} (x_1 + a)(x_4 + a) \\ (x_1 + a) + (x_4 + a) \end{bmatrix} = \begin{bmatrix} x_1 x_4 + a \\ x_1 + x_4 + a \end{bmatrix}$$

# Example



$$\text{range}(f) = d \text{ range}(f^2|_{(x_1x_4 + a)=1}) + d' \text{ range}(f^2|_{(x_1x_4 + a)=0})$$

$$= d \text{ range}(x_1 + x_4 + a|_{(x_1x_4 + a)=1}) + d' \text{ range}(x_1 + x_4 + a|_{(x_1x_4 + a)=0})$$

$$= d \text{ range}(1) + d' \text{ range}(a' (x_1 \oplus x_4))$$

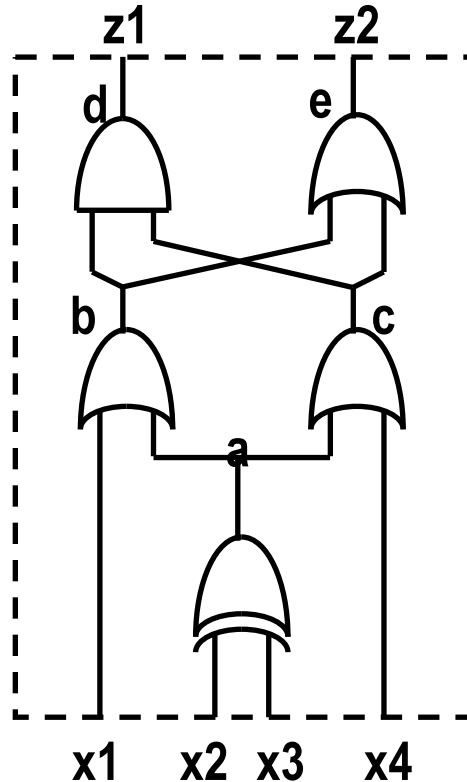
$$= de + d' (e + e')$$

$$= e + d'$$

$$\blacklozenge \text{CDC}_{\text{out}} = (e + d')' = de' = z_1 z_2'$$

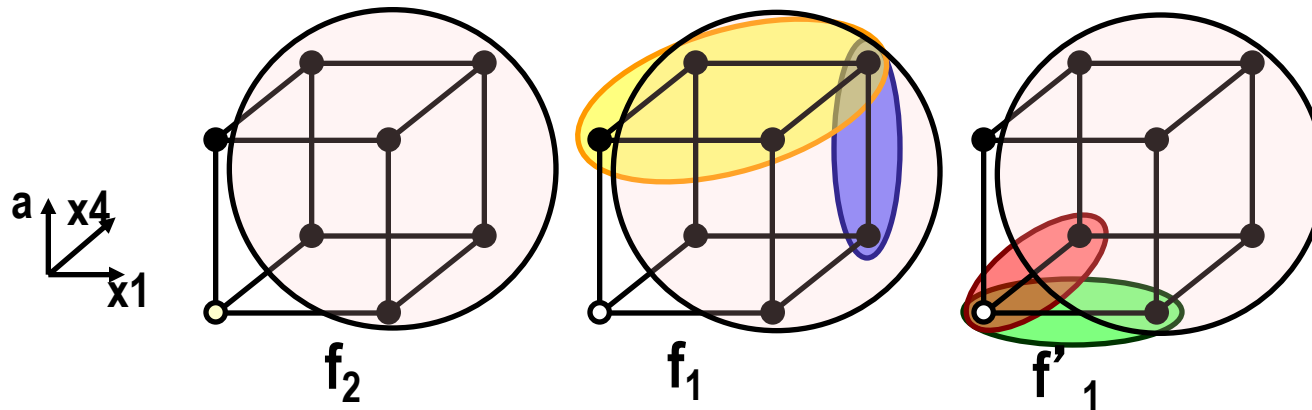
# Example

$$CDC_{in} = x'_1 x'_4$$



$$f = \begin{bmatrix} f^1 \\ f^2 \end{bmatrix} = \begin{bmatrix} (x_1 + a)(x_4 + a) \\ (x_1 + a) + (x_4 + a) \end{bmatrix} = \begin{bmatrix} x_1 x_4 + a \\ x_1 + x_4 + a \end{bmatrix}$$

# Example



$$\begin{aligned}
 \text{image}(f) &= d \text{ image}(f^2|_{(x_1x_4 + a)=1}) + d' \text{ image}(f^2|_{(x_1x_4 + a)=0}) \\
 &= d \text{ image}(x_1 + x_4 + a|_{(x_1x_4 + a)=1}) + d' \text{ image}(x_1 + x_4 + a|_{(x_1x_4 + a)=0}) \\
 &= d \text{ image}(1) + d' \text{ image}(1) \\
 &= de + d' e \\
 &= e
 \end{aligned}$$

◆  $\text{CDC}_{\text{out}} = e' = z_2'$

# Observability analysis

---

- ◆ **Complementary to controllability**
  - ▲ Analyze network from outputs to inputs
- ◆ **More complex because network has several outputs and observability depends on output**
- ◆ **Observability may be understood in terms of perturbations**
  - ▲ If you flip the polarity of a signal at net  $x$ , and there is no change in the outputs, then  $x$  is not observable

# Observability *don't care* conditions

---

- ◆ Conditions under which a change in polarity of a signal **x** is not perceived at the output
- ◆ If there is an explicit representation of the function, the **ODC** is the complement of the Boolean difference
$$\text{ODC} = (\partial f / \partial x)'$$
- ◆ Often, the terminal behavior is described implicitly
  - ▲ Applying chain rule to Boolean difference is computationally hard

# Tree-network traversal

---

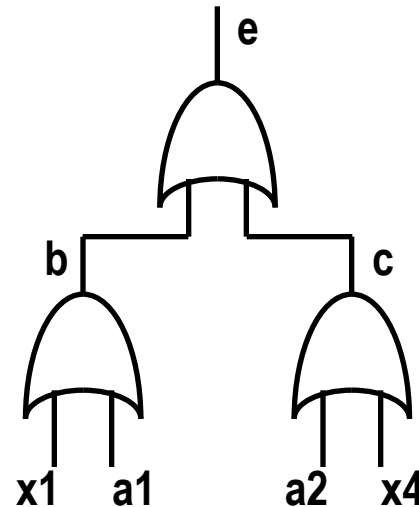
- ◆ Consider network from outputs to input
- ◆ At root
  - ▲  $ODC_{out}$  is given
  - ▲ It may be empty
- ◆ At internal nodes:
  - ▲ Local function  $y = f_y(x)$
  - ▲  $ODC_x = (\partial f_y / \partial x)' + ODC_y$
- ◆ Observability don't care set has two components:
  - ▲ Observability of the local function and observability of the network beyond the local block

# Example

$$e = b + c$$

$$b = x_1 + a_1$$

$$c = x_4 + a_2$$



◆ Assume  $ODC_{out} = ODC_e = 0$

◆  $ODC_b = (\partial f_e / \partial b)' = (b + c)|_{b=1} \oplus (b + c)|_{b=0} = c$

◆  $ODC_c = (\partial f_e / \partial c)' = b$

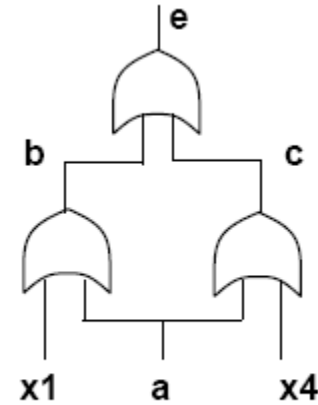
◆  $ODC_{x_1} = ODC_b + (\partial f_b / \partial x_1)' = c + a_1$



# Non-tree network traversal

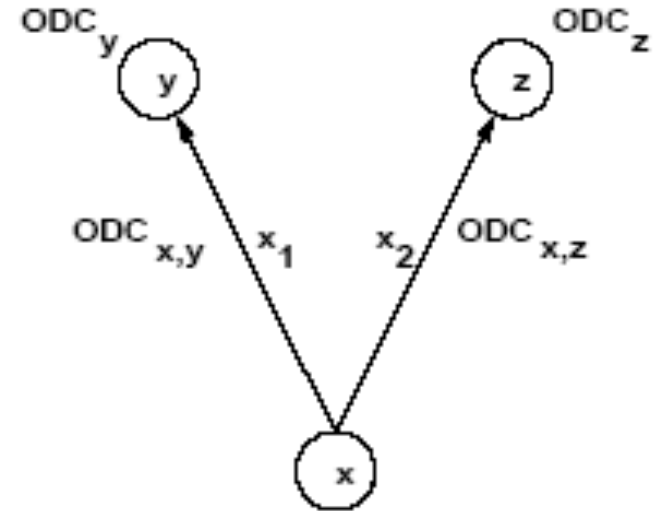
---

- ◆ General networks have forks and fanout reconvergence
- ◆ For each fork point, the contribution to the ODC depends on both paths
- ◆ Network traversal cannot be applied in a straightforward way
- ◆ More elaborate analysis is needed



# Two-way fork

- ◆ Compute **ODC** sets associated with edges
- ◆ Recombine ODCs at fork point
- ◆ Theorem:
  - ▲  $ODC_x = ODC_{x,y|x=x'} \oplus ODC_{x,z}$
  - ▲  $ODC_x = ODC_{x,z|x=x'} \oplus ODC_{x,y}$
- ◆ Multi-way forks can be reduced to a sequence of two-way forks



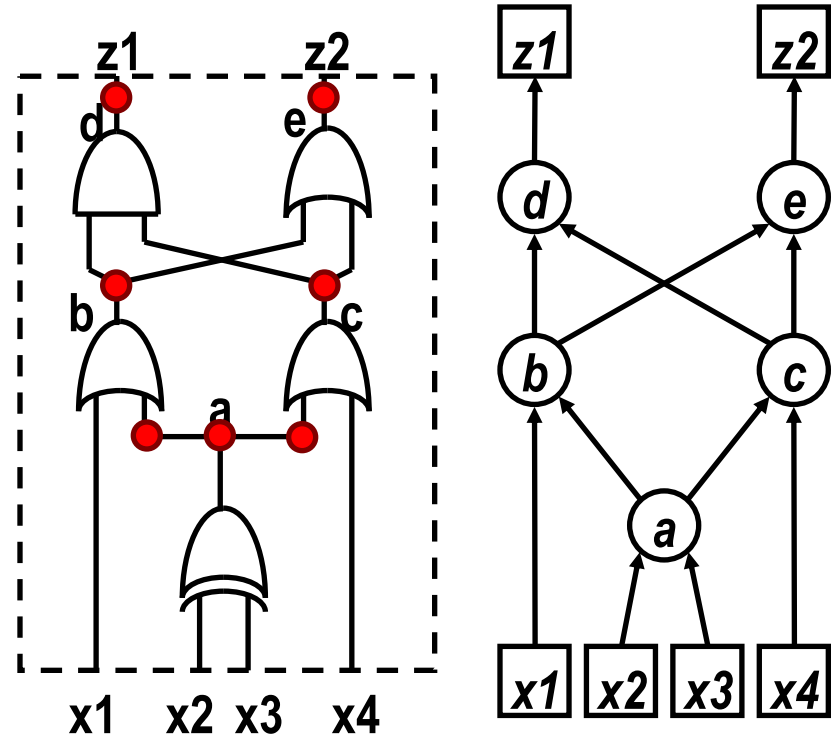
# Example

$$\text{ODC}_c = \begin{pmatrix} b' \\ b \end{pmatrix}; \quad \text{ODC}_b = \begin{pmatrix} c' \\ c \end{pmatrix};$$

$$\text{ODC}_{a,b} = \begin{pmatrix} c' + x_1 \\ c + x_1 \end{pmatrix} = \begin{pmatrix} a' x_4' + x_1 \\ a + x_4 + x_1 \end{pmatrix}$$

$$\text{ODC}_{a,c} = \begin{pmatrix} b' + x_4 \\ b + x_4 \end{pmatrix} = \begin{pmatrix} a' x_1' + x_4 \\ a + x_1 + x_4 \end{pmatrix}$$

$$\text{ODC}_a = \text{ODC}_{a,b|a=a'} \oplus \text{ODC}_{a,c} = \begin{pmatrix} a' x_4' + x_1 \\ a' + x_4 + x_1 \end{pmatrix} \oplus \begin{pmatrix} a' x_1' + x_4 \\ a + x_1 + x_4 \end{pmatrix} = \begin{pmatrix} x_1 x_4 \\ x_1 + x_4 \end{pmatrix}$$



# Don't care computation summary

---

- ◆ **Controllability *don't* cares** are derived by image computation
  - ▲ Recursive algorithms and data structure are applied
- ◆ **Observability *don't* cares** are derived by backward traversal
  - ▲ Exact and approximate computation
  - ▲ Approximate methods compute *don't* care subsets

# Transformations with don't cares

---

## ◆ Boolean simplification

- ▲ Generate local DC set for local functions
- ▲ Use heuristic minimizer (e.g., Espresso)
- ▲ Minimize the number of literals

## ◆ Boolean substitution:

- ▲ Simplify a function by adding one (ore more) inputs
- ▲ Equivalent to simplification with *global don't care* sets

# Example – Boolean substitution

---

◆ Substitute  $q = a + cd$  into  $f_h = a + bcd + e$

▲ Obtain  $f_h = a + bq + e$

◆ Method

▲ Compute SDC including  $q \oplus (a+cd) = q' a + q' cd + qa' (cd)'$

▲ Simplify  $f_h = a + bcd + e$  with DC =  $q' a + q' cd + qa' (cd)'$

▲ Obtain  $f_h = a + bq + e$

◆ Result

▲ Simplified function has one fewer literal by changing the support of  $f_h$

# Simplification operator

---

## ◆ Cycle over the network blocks

- ▲ Compute local don't care conditions
- ▲ Minimize

## ◆ Issues:

- ▲ Don't care sets change as blocks are being simplified
- ▲ Iteration may not have a fixed point
- ▲ It would be efficient to parallelize some simplifications

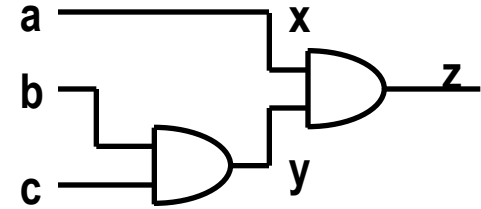
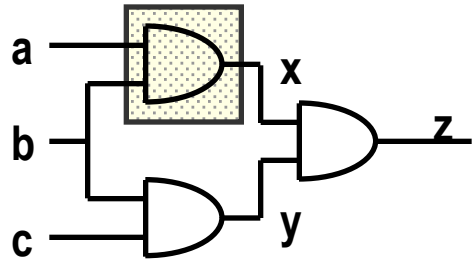
# Optimization and perturbation

---

- ◆ Minimizing a function at a block  $x$  is the replacement of a local function  $f_x$  with a new function  $g_x$
- ◆ This is equivalent to perturbing the network locally by
  - ▲  $\delta_x = f_x \oplus g_x$
- ◆ Conditions for a feasible replacement
  - ▲ Perturbation bounded by local don't care sets
  - ▲  $\delta_x$  included in  $DC_{\text{ext}} + ODC + CDC$
- ◆ Smaller, approximate *don't care* sets can be used
  - ▲ But have smaller degrees of freedom



# Example



- ◆ No external *don't care* set.
- ◆ Replace **AND** by wire:  $g_x = a$
- ◆ Analysis:
  - ▲  $\delta = f_x \oplus g_x = ab \oplus a = ab'$
  - ▲  $ODC_x = y' = b' + c'$
  - ▲  $\delta = ab' \subseteq DC_x = b' + c' \Rightarrow$  feasible!

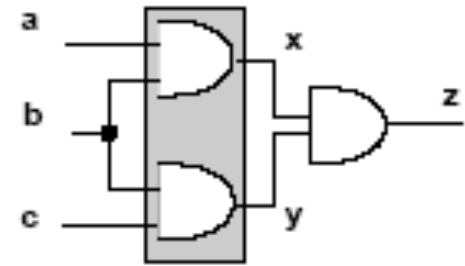
# Parallel simplification

---

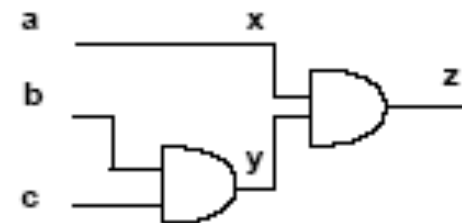
- ◆ **Parallel minimization of logic blocks is always possible when blocks are logically independent**
  - ▲ **Partitioned network**
- ◆ **Within a connected network, logic blocks affect each other**
- ◆ **Doing parallel minimization is like introducing multiple perturbations**
  - ▲ **But it is attractive for efficiency reasons**
- ◆ **Perturbation analysis shows that degrees of freedom cannot be represented by just an upper bound on the perturbation**
  - ▲ **Boolean relation model**

# Example

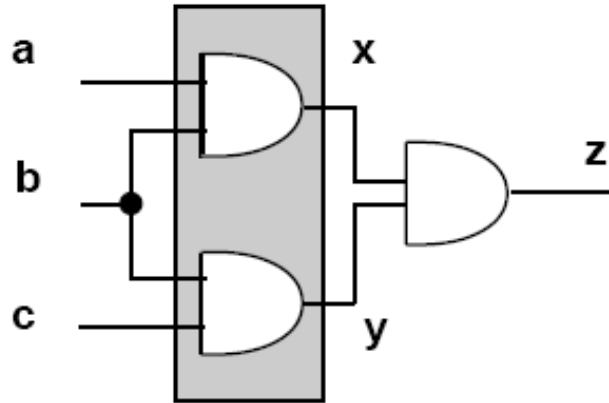
- ◆ Perturbations at **x** and **y** are related because of the reconvergent fanout at **z**
- ◆ Cannot change simultaneously
  - ▲ **ab** into **a**
  - ▲ **cb** into **c**



(a)



# Boolean relation model



<i>a</i>	<i>b</i>	<i>c</i>	<i>x, y</i>
0	0	0	{ 00, 01, 10 }
0	0	1	{ 00, 01, 10 }
0	1	0	{ 00, 01, 10 }
0	1	1	{ 00, 01, 10 }
1	0	0	{ 00, 01, 10 }
1	0	1	{ 00, 01, 10 }
1	1	0	{ 00, 01, 10 }
1	1	1	{ 11 }

<i>a</i>	<i>b</i>	<i>c</i>	<i>x, y</i>
1	*	*	10
*	1	1	01

# Boolean relation model

---

- ◆ **Boolean relation minimization is the correct approach to handle Boolean optimization at multiple vertices**
- ◆ **Necessary steps**
  - ▲ **Derive equivalence classes for Boolean relation**
  - ▲ **Use relation minimizer**
- ◆ **Practical considerations**
  - ▲ **High computational requirement to use Boolean relations**
  - ▲ **Use approximations instead**

# Parallel Boolean optimization compatible *don't care* sets

---

- ◆ Determine a subset of *don't care* sets which is safe to use in a parallel minimization
  - ▲ Remove those degrees of freedom that can lead to transformations incompatible with others effected in parallel
- ◆ Using **compatible *don't care* sets**, only upper bounds on the perturbation need to be satisfied
- ◆ Faster and efficient method

# Example

---

## ◆ Parallel optimization at two vertices

### ◆ First vertex $x$

▲ CODC equal to ODC set

$$\text{▲ } \text{CODC}_x = \text{ODC}_x$$

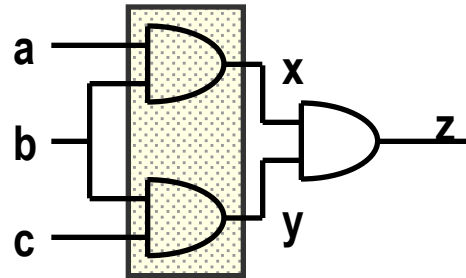
### ◆ Second vertex $y$

▲ CODC is smaller than its ODC to be safe enough to allow for transformations permitted by the first ODC

$$\text{▲ } \text{CODC}_y = C_x(\text{ODC}_y) + \text{ODC}_y \text{ ODC}'_x$$

### ◆ Order dependence

# Example



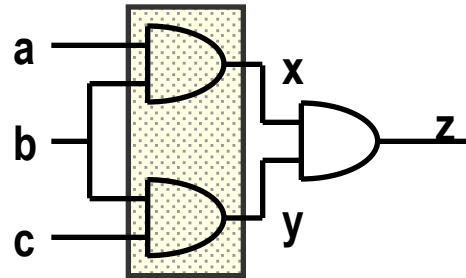
$$\blacklozenge \text{CODC}_y = \text{ODC}_y = x' = b' + a'$$

$$\blacklozenge \text{ODC}_x = y' = b' + c'$$

$$\begin{aligned} \blacklozenge \text{CODC}_x &= C_y(\text{ODC}_x) + \text{ODC}_x(\text{ODC}_y)' \\ &= C_y(y') + y' x = y' x \\ &= (b' + c')ab = abc' \end{aligned}$$



## Example (2)



### ◆ Allowed perturbation:

$$\blacktriangle f_y = bc \rightarrow g_y = c$$

$$\blacktriangle \delta_y = bc \oplus c = b'c \subseteq \text{CODC}_y = b' + a'$$

### ◆ Disallowed perturbation:

$$\blacktriangle f_x = ab \rightarrow g_x = a.$$

$$\blacktriangle \delta_x = ab \oplus a = ab' \not\subseteq \text{CODC}_x = abc'$$

# Boolean methods Summary

---

- ◆ **Boolean methods are powerful means to restructure networks**
  - ▲ **Computationally intensive**
- ◆ **Boolean methods rely heavily on *don't care* computation**
  - ▲ **Efficient methods**
  - ▲ **Possibility to subset the *don't care* sets**
- ◆ **Boolean method often change the network substantially, and it is hard to undo Boolean transformations**

# Module 2

---

## ◆ Objectives

### ▲ Testability

### ▲ Relations between testability and Boolean methods

# Testability

---

- ◆ **Generic term to mean easing the testing of a circuit**
- ◆ **Testability in logic synthesis context**
  - ▲ **Assume combinational circuit**
  - ▲ **Assume single/multiple stuck-at fault**
- ◆ **Testability is referred to as the possibility of generating test sets for all faults**
  - ▲ **Property of the circuit**
  - ▲ **Related to fault coverage**

# Test for *stuck-ats*

---

## ◆ Net **y** stuck-at 0

- ▲ Input pattern that sets **y** to TRUE
- ▲ Observe output
- ▲ Output of faulty circuit differs from correct circuit

## ◆ Net **y** stuck-at 1

- ▲ Input pattern that sets **y** to FALSE
- ▲ Observe output
- ▲ Output of faulty circuit differs from correct circuit

## ◆ Testing is based on *controllability* and *observability*

# Test sets – *don't care* interpretation

---

## ◆ Stuck-at 0 on net $y$

▲ { Input vector  $\mathbf{t}$  such that  $y(\mathbf{t}) \text{ ODC}' y(\mathbf{t}) = 1$  }

## ◆ Stuck-at 1 on net $y$

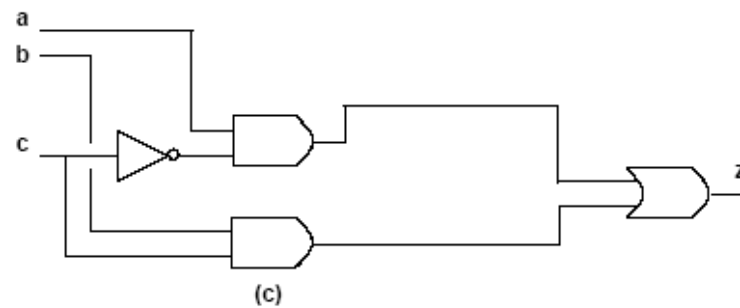
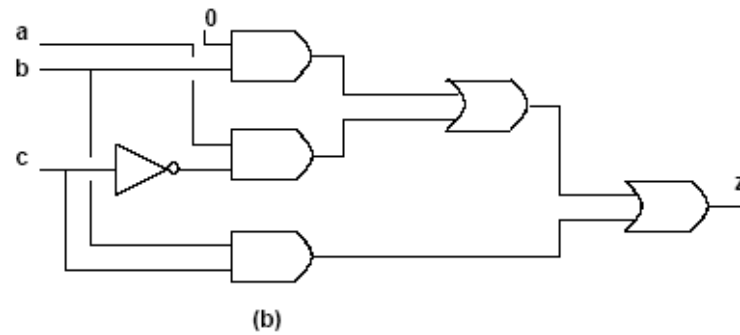
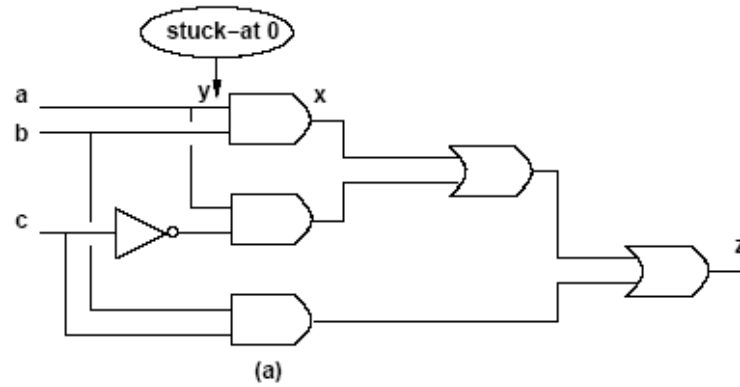
▲ { Input vector  $\mathbf{t}$  such that  $y'(\mathbf{t}) \text{ ODC}' y(\mathbf{t}) = 1$  }

# Using testing methods for synthesis

---

- ◆ Redundancy removal
  - ▲ Use ATPG to search for untestable fault
- ◆ If stuck-at 0 on net  $y$  is untestable:
  - ▲ Set  $y = 0$
  - ▲ Propagate constant
- ◆ If stuck-at 1 on net  $y$  is untestable
  - ▲ Set  $y = 1$
  - ▲ Propagate constant
- ◆ Iterate for each untestable fault

# Example





# Redundancy removal and perturbation analysis

## ◆ Stuck-at 0 on $y$

▲  $y$  set to 0. Namely  $g_x = f_x|_{y=0}$



▲ Perturbation:

$$\nabla \delta = f_x \oplus f_x|_{y=0} = y \cdot \partial f_x / \partial y$$

## ◆ Perturbation is feasible $\Leftrightarrow$ fault is untestable

▲ No input vector  $\mathbf{t}$  can make  $y(\mathbf{t}) \cdot \text{ODC}_y'(\mathbf{t})$  true

▲ No input vector  $\mathbf{t}$  can make  $y(\mathbf{t}) \cdot \text{ODC}_x'(\mathbf{t}) \cdot \partial f_x / \partial y$  true

$$\nabla \text{Because } \text{ODC}_y = \text{ODC}_x + (\partial f_x / \partial y)'$$

# Redundancy removal and perturbation analysis

---

◆ Assume untestable stuck-at 0 fault.

◆  $y \cdot \text{ODC}_x' \cdot \partial f_x / \partial y \subseteq \text{SDC}$

◆ Local don't care set:

▲  $\text{DC}_x \supseteq \text{ODC}_x + y \cdot \text{ODC}_x' \cdot \partial f_x / \partial y$

▲  $\text{DC}_x \supseteq \text{ODC}_x + y \cdot \partial f_x / \partial y$

◆ Perturbation  $\delta = y \cdot \partial f_x / \partial y$

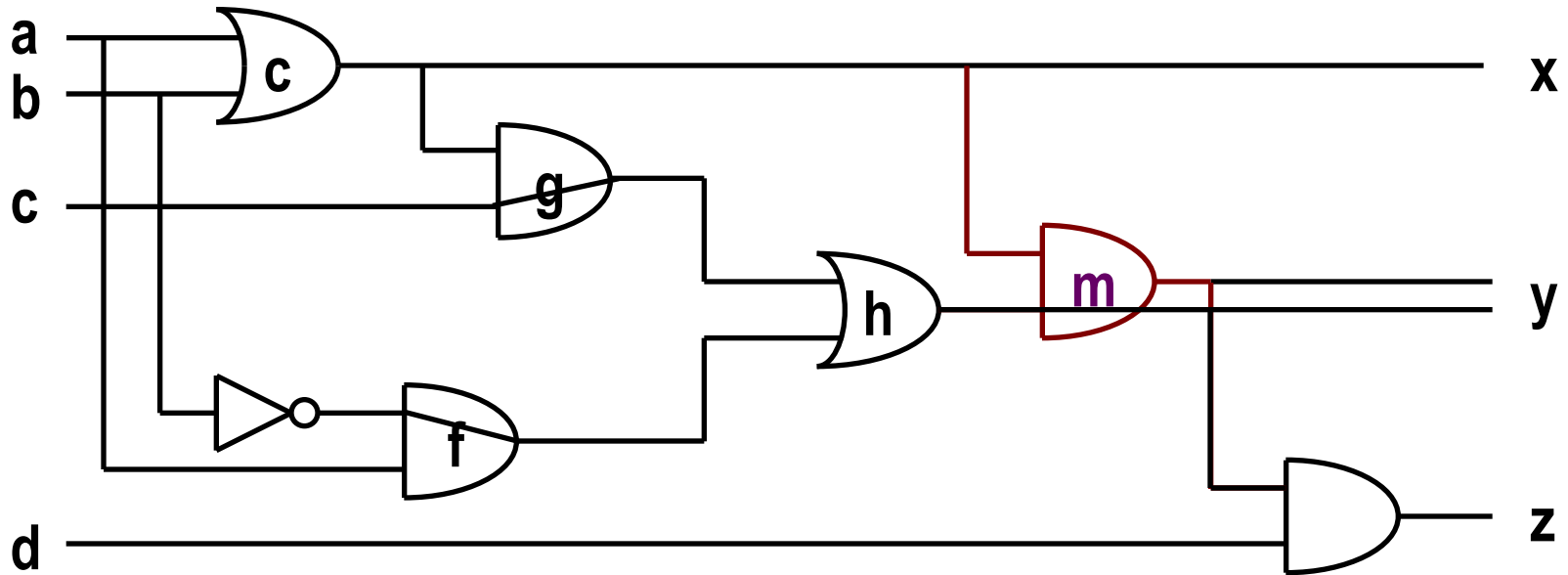
▲ Included in the local don't care set

# Rewiring

---

- ◆ **Extension to redundancy removal**
  - ▲ Add connection in a circuit
  - ▲ Create other redundant connections
  - ▲ Remove redundant connections
- ◆ **Iterate procedure to reduce network**
  - ▲ A connection corresponds to a wire
  - ▲ Rewiring modifies gates and wiring structure
  - ▲ Wires may have specific costs due to distance

# Example



# Synthesis for testability

---

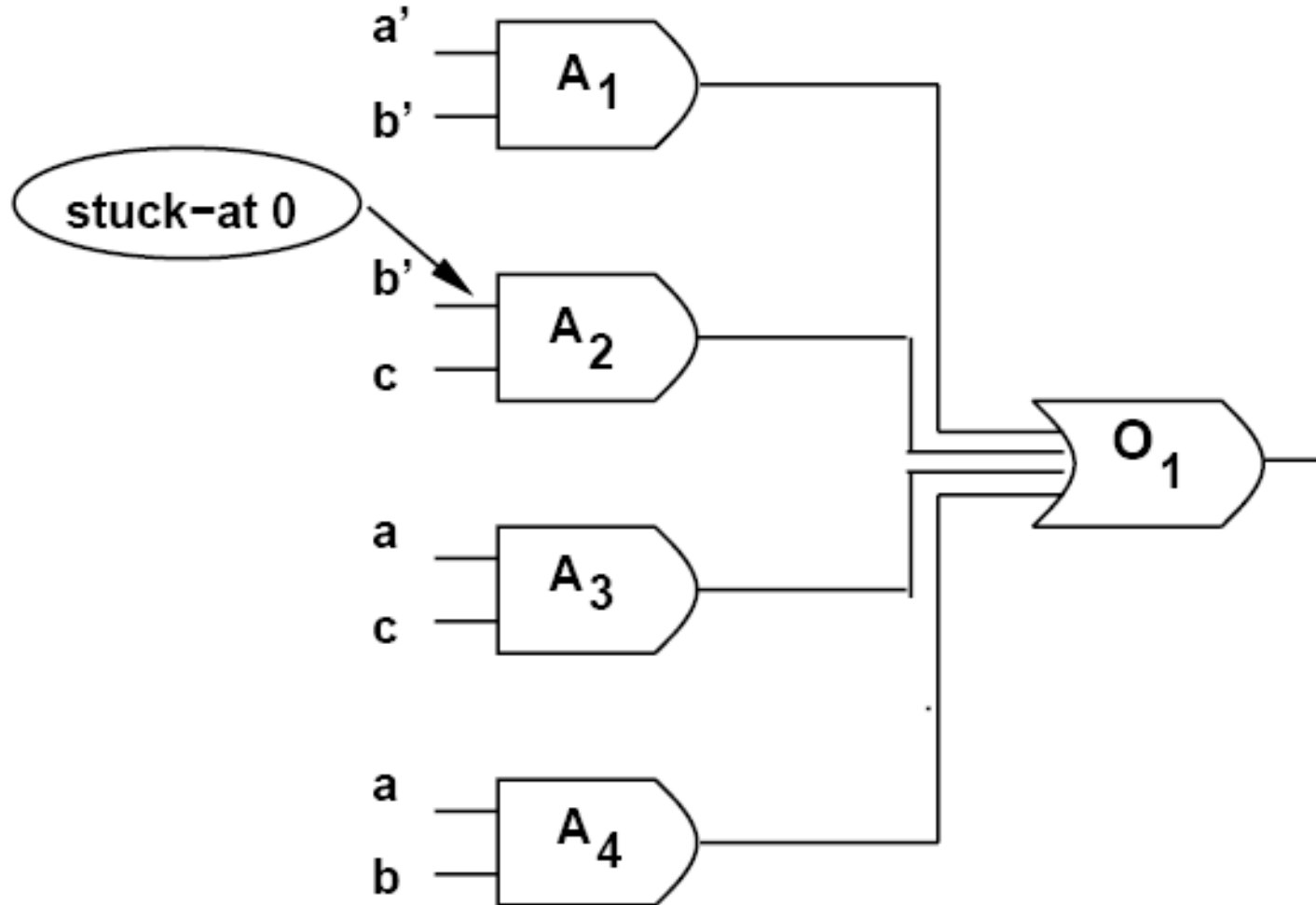
- ◆ **Synthesize fully testable circuits**
  - ▲ For single or multiple stuck-at faults
- ◆ **Realizations**
  - ▲ Two-level forms
  - ▲ Multi-level networks
- ◆ **Since synthesis can modify the network properties, testability can be addressed during synthesis**

# Two-level forms

---

- ◆ **Full testability for single stuck-at faults:**
  - ▲ **Prime and irredundant covers**
- ◆ **Full testability for multiple stuck-at faults**
  - ▲ **Prime and irredundant cover when**
    - ▼ **Single output function**
    - ▼ **No product-term sharing**
    - ▼ **Each component is prime and irredundant**

# Example $f = a' b' + b' c + ac + ab$



# Multiple-level networks

---

- ◆ Consider logic networks with local functions in *sop* form
- ◆ Prime and irredundant network
  - ▲ No literal and no implicant of any local function can be dropped
  - ▲ The AND-OR implementation is fully testable for single *stuck-at* faults
- ◆ Simultaneous prime and irredundant network
  - ▲ No subsets of literals and no subsets of implicants can be dropped
  - ▲ The AND-OR implementation is fully testable for multiple *stuck-ats*



# Synthesis for testability

---

- ◆ Heuristic logic minimization (e.g., Espresso) is sufficient to insure testability of two-level forms
- ◆ To achieve fully testable networks, simplification has to be applied to all logic blocks with full *don't care* sets
- ◆ In practice, don't care sets change as neighboring blocks are optimized
- ◆ Redundancy removal is a practical way of achieving testability properties

# Summary – Synthesis for testability

---

- ◆ **There is synergy between synthesis and testing**
  - ▲ **Don't care conditions play a major role in both fields**
- ◆ **Testable network correlate to a small area implementation**
- ◆ **Testable network do not require to slow-down the circuit**
- ◆ **Algebraic transformations preserve multi-fault testability, and are preferable under this aspect**