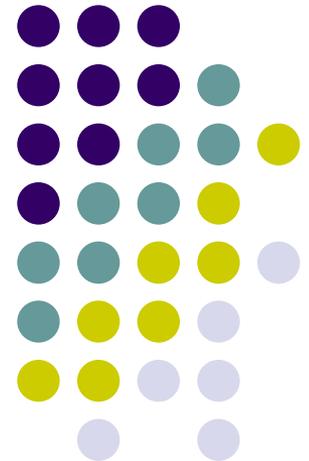


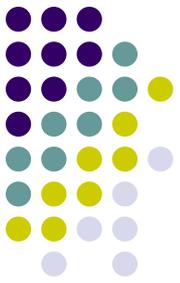
# Chapter 5

## Combinational ATPG

組合電路自動輸入樣本產生器

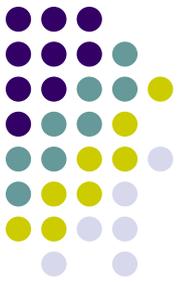


# Outline

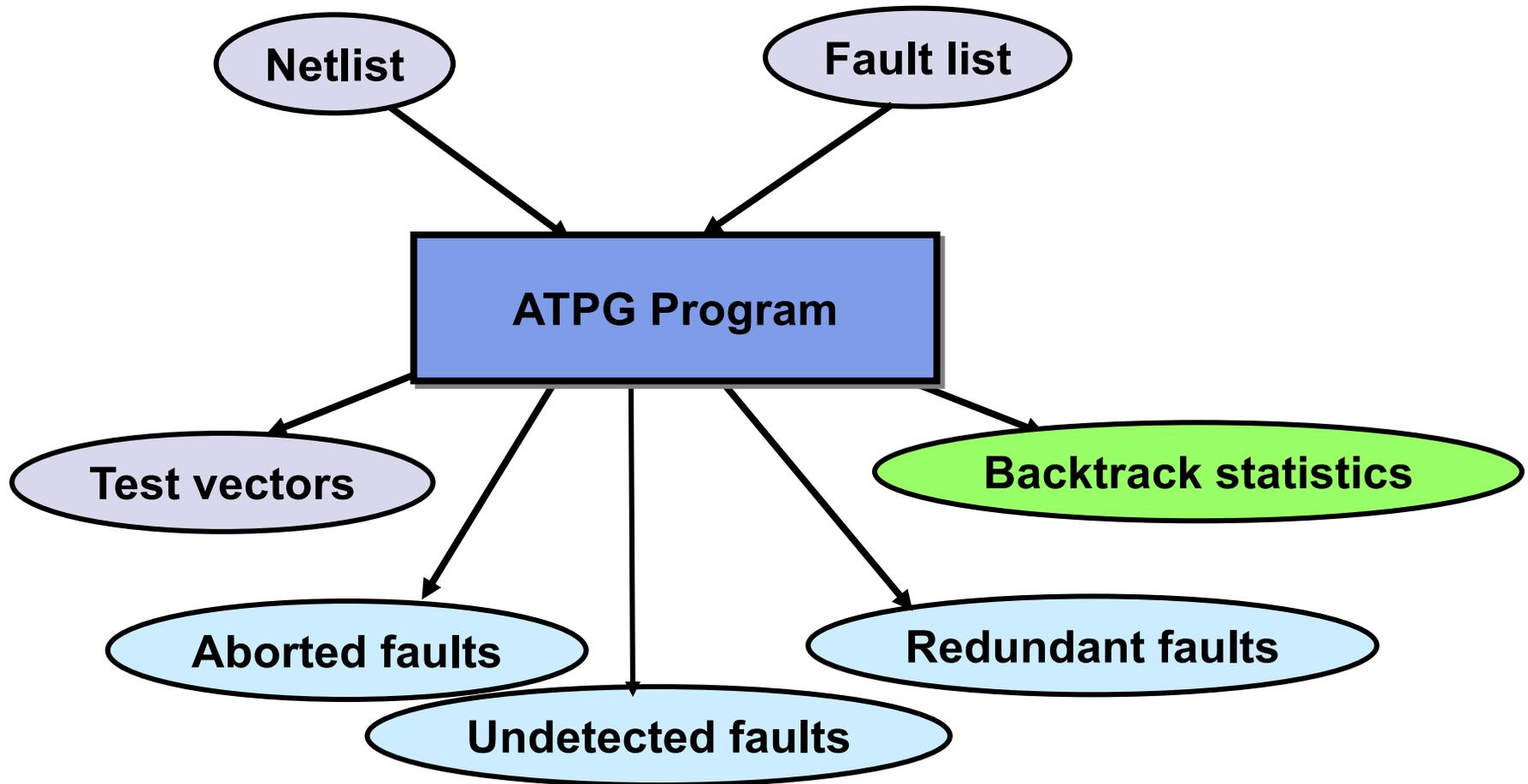


- Introduction to ATPG
- ATPG for Combinational Circuits
- Advanced ATPG Techniques

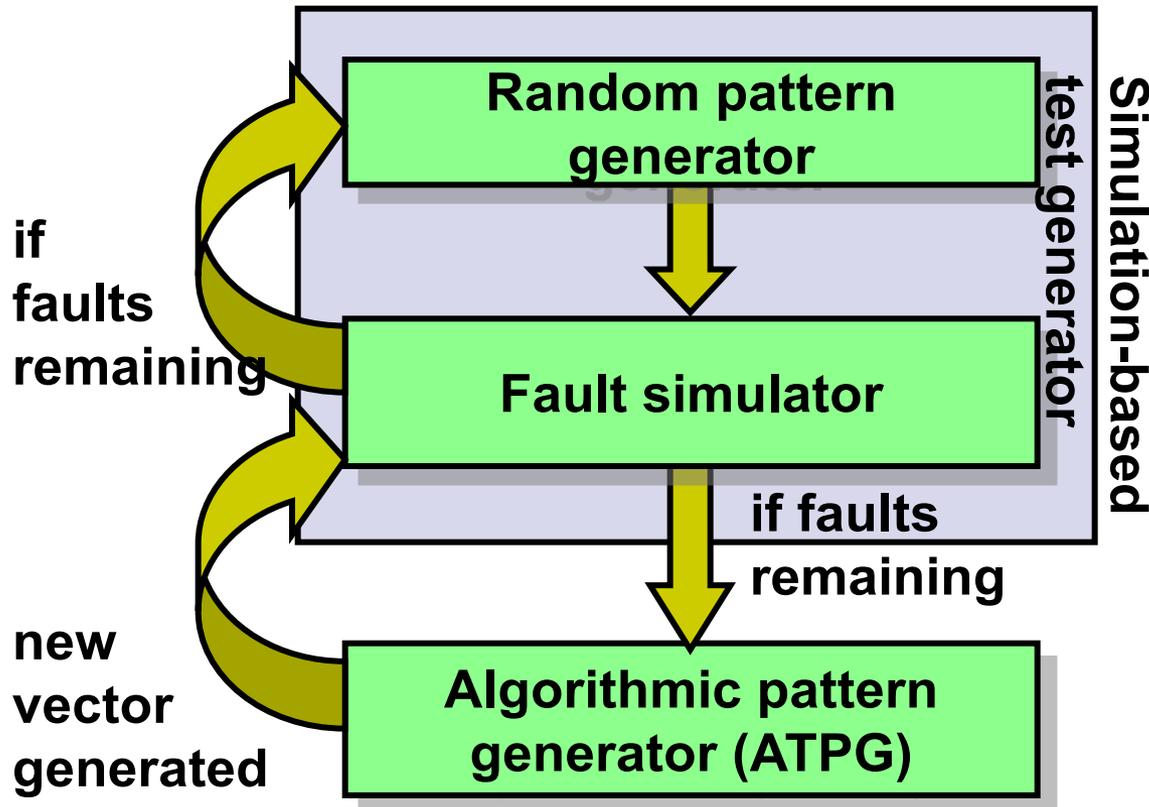
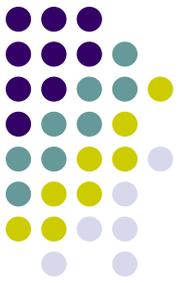
# Input and Output of an ATPG



- ATPG (Automatic Test Pattern Generation)
  - Generate a set of vectors for a set of target faults



# Components of An ATPG System

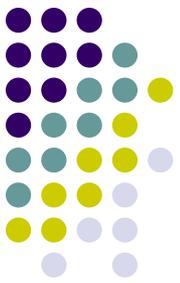


To detect easily detectable faults

Find all faults captured by current vectors

Generate vectors for undetected faults or prove them to be undetectable

# Revisiting Fault Coverage

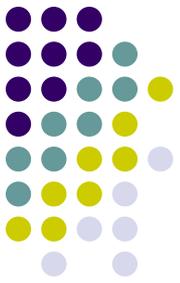


- Fault efficiency (ATPG efficiency)
  - Percentage of faults being successfully processed by a ATPG
  - Proving a fault undetectable  $\neq$  testing the fault
    - Undetectable faults can cause performance, power, reliability problems, etc.
    - Faults may be masked by undetectable faults.

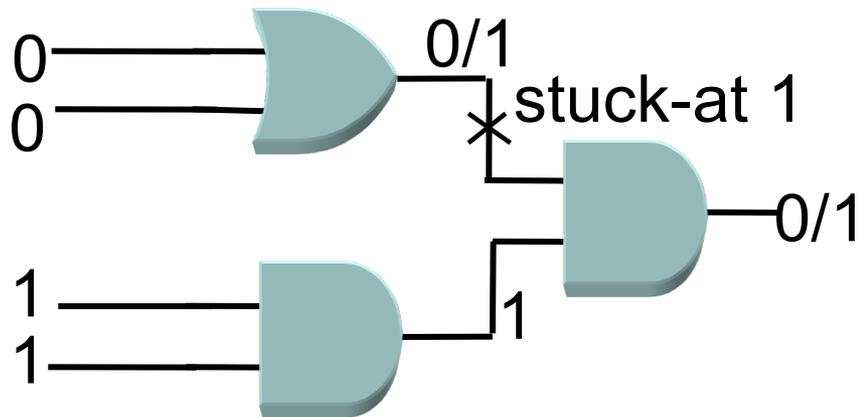
$$\text{Fault coverage} = \frac{\text{\# of detected faults}}{\text{Total \# of faults}}$$

$$\text{Fault efficiency} = \frac{\text{\# of detected faults}}{\text{Total \# of faults} - \text{\# of undetectable faults}}$$

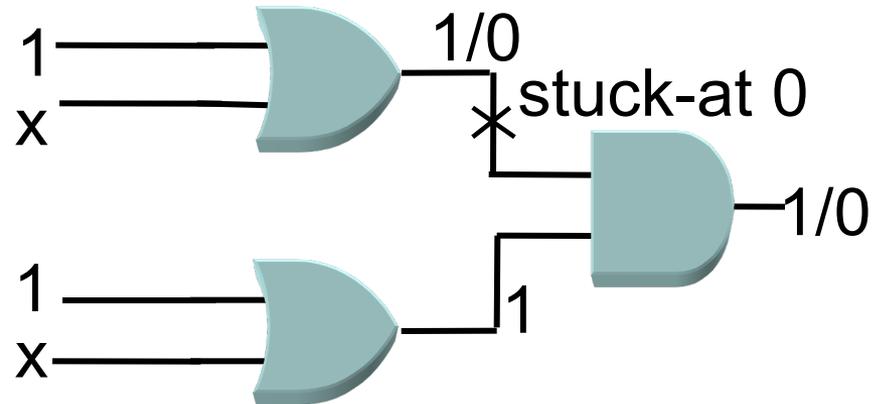
# Fully V.S. Partially Specified Patterns



A fully specified test pattern  
(every PI is either 0 or 1)

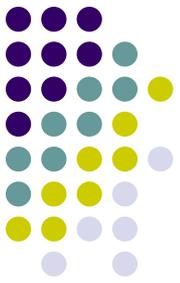


A partially specified test pattern  
(certain PI's could be unknown)



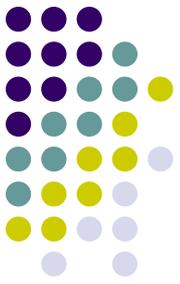
- Applications of partially specified test patterns
  - Test compaction
  - Finding pattern of best test power, delay, etc.

# Test Compaction



- To reduce the number of test vectors as long as we keep the same detectable fault coverage.
- Test compaction is important for randomly generated patterns.
  - Even for vectors generated by an ATPG, since the order of processing faults will decide which vectors are discovered first.

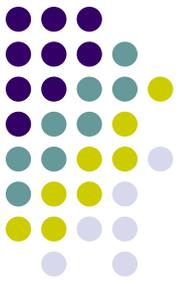
# An Example of Different Vectors Detecting the Same Faults



- Assume we know the vectors and their detectable faults
  - If we start from f1, we find v1 to detect (f1, f3, f5).
  - And then continue with f2. We find v2 to detect it.
  - And for the remaining f4, we find v3 for detection.
  - In total, we need 3 vectors.

	f1	f2	f3	f4	f5
v1	x		x		x
v2	x	x	x		
v3			x	x	x
v4	x	x			x

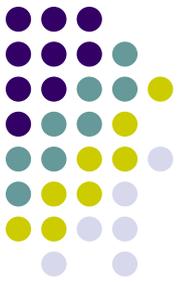
# An Example of Different Vectors Detecting the Same Faults (cont.)



- With the same assumption
  - If we order the faults as f4, f2, f5, f3, f1.
  - For f4, we find v3 to cover (f3, f4, f5)
  - And for f2, we find v2 or v4 to cover (f1, f2)
  - In total, we only need 2 vectors.

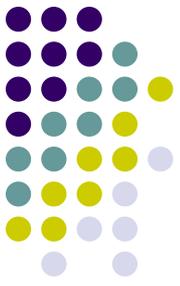
	f1	f2	f3	f4	f5
v1	x		x		x
v2	x	x	x		
v3			x	x	x
v4	x	x			x

# Test Compaction Methods



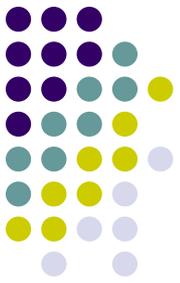
- Use fault dictionary
  - Find essential vectors
    - No other vector can test faults covered only by essential vectors.
  - Find minimum vectors to cover all rest faults
    - A NP-complete problem.
- Try different orders of input vectors in a fault simulator.

# Other Test Compaction Methods



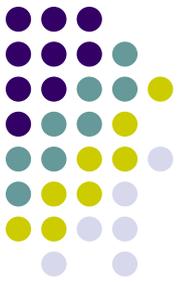
- Static compaction
  - After a set of vectors has been generated
  - Use D-intersection
  - For example,  $t_1=01X$   $t_2=0X1$   $t_3=0X0$   $t_4=X01$
  - after compaction:  $t_{13}=010$   $t_{24}=001$
- Dynamic compaction
  - Process generated vectors on-the-fly
  - After generate a test for a fault, choose a secondary target fault to be tested and try to test the second fault by don't cares (X) not assigned by the first fault.

# Combinational ATPG



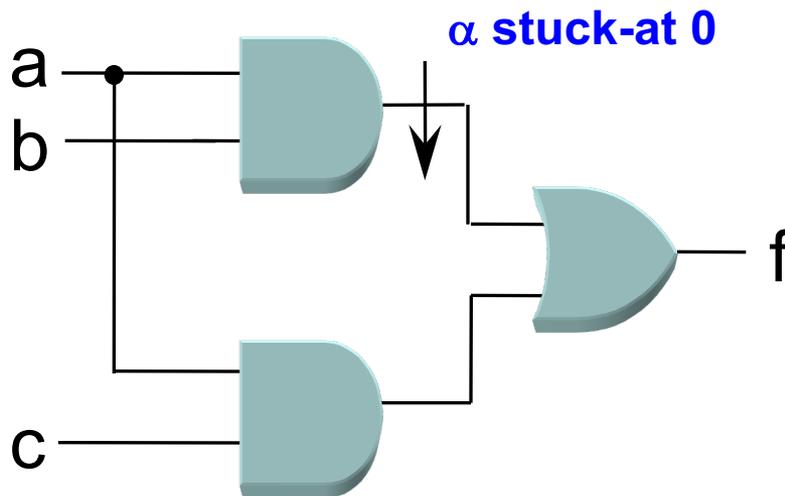
- Test Generation (TG) Methods
  - Exhaustive methods
  - Boolean Equation
  - Structural Analysis
  - Implication Graph
- Milestone Structural ATPG Algorithms
  - D-algorithm [Roth 1967]
  - 9-Valued D-algorithm [Cha 1978]
  - PODEM [Goel 1981]
  - FAN [Fujiwara 1983]
  - Other advanced techniques

# Exhaustive Test Generation



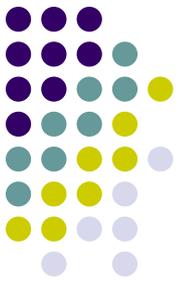
- Explore all possible input combinations to find input patterns detecting faults
- Complexity  $\sim O(n \times 2^{\text{no\_pi}})$

**Generate tests for the stuck-at 0 fault,  $\alpha$ .**



Inputs	f	$f\alpha$
000	0	0
001	0	0
010	0	0
011	0	0
100	0	0
101	1	1
110	1	0
111	1	1

# Boolean Equation Method



$$f = ab+ac, f_{\alpha} = ac$$

$T_{\alpha}$  = the set of all tests for fault  $\alpha$

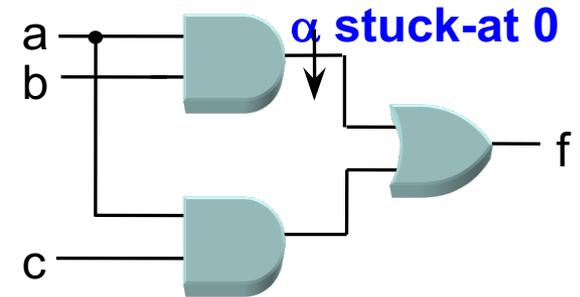
$$= \{(a,b,c) \mid f \oplus f_{\alpha} = 1\}$$

$$= \{(a,b,c) \mid f * f_{\alpha}' + f' * f_{\alpha} = 1\}$$

$$= \{(a,b,c) \mid (ab+ac)(ac)' + (ab+ac)'(ac) = 1\}$$

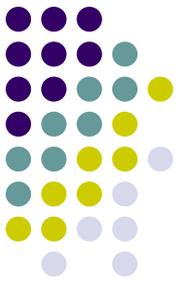
$$= \{(a,b,c) \mid abc' = 1\}$$

$$= \{(110)\}.$$



- The complexity is high with the computation of faulty function.

# Boolean Difference Method



- $f \oplus f_{\alpha=1}$

==

$(\alpha=0) (f(\alpha=0) \oplus f(\alpha=1))$  for  $\alpha$  stuck-at 1, or

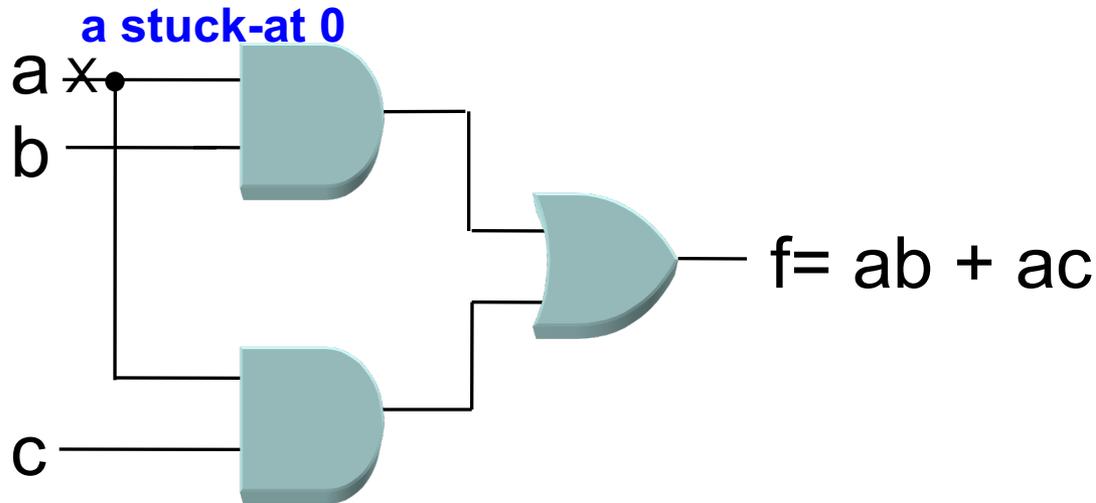
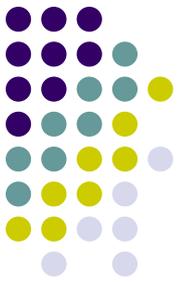
$(\alpha=1) (f(\alpha=0) \oplus f(\alpha=1))$  for  $\alpha$  stuck-at 0

- Define Boolean difference

$$df/d\alpha = f(\alpha=0) \oplus f(\alpha=1)$$

- Meaning any change at  $\alpha$  can be observed at the outputs of  $f()$ .

# Example of Boolean Difference with Fault at PIs



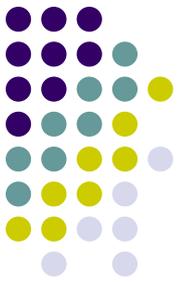
$$df/da = f(a=0) \oplus f(a=1) = 0 \oplus (b+c) = (b+c)$$

$$\text{Test-set for } a \text{ s-a-0} = \{(a,b,c) \mid \underline{a} \bullet \underline{(b+c)} = 1\} = \{(11x), (1x1)\}.$$

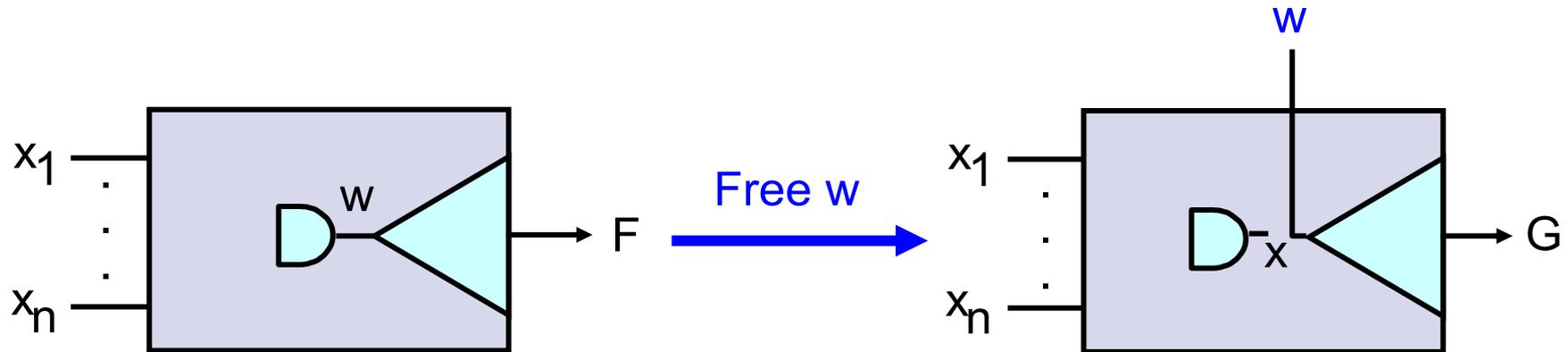
**Fault activation  
requirement**

**Fault sensitization  
requirement**

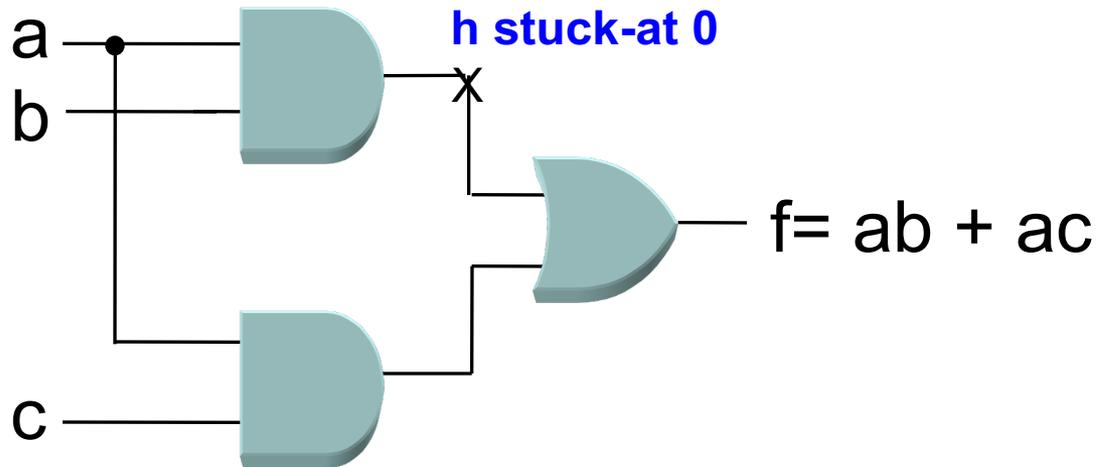
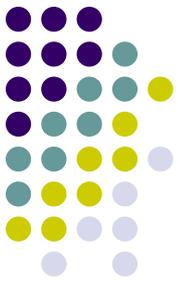
# Boolean Difference for Internal Signals



- Calculation
  - Step 1: convert the function  $F$  into a new one  $G$  that takes the signal  $w$  as an extra primary input
  - Step 2:  $dF(x_1, \dots, x_n)/dw = dG(x_1, \dots, x_n, w)/dw$



# Example of Boolean Difference with Internal Fault



$G$  (i.e.,  $F$  with  $h$  floating) =  $h + ac$

$dG/dh = G(h=0) \oplus G(h=1) = (ac \oplus 1) = (a'+c')$

Test-set for  $h$  s-a-1 is

$$\{(a,b,c) \mid h' \cdot (a'+c')=1\} = \{(a,b,c) \mid (a'+b') \cdot (a'+c')=1\} = \{(0xx), (x00)\}.$$

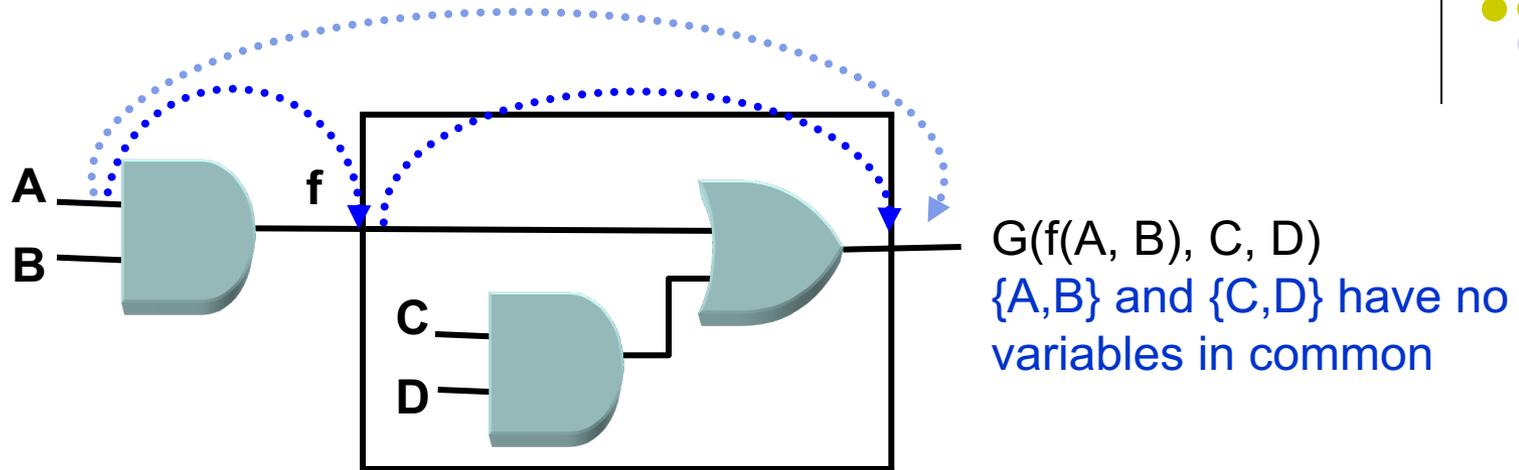
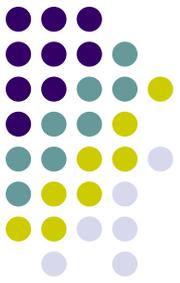
Test-set for  $h$  s-a-0 is

$$\{(a,b,c) \mid \underline{h} \cdot \underline{(a'+c')}=1\} = \{(110)\}.$$

For fault activation

For fault sensitization

# Chain Rule



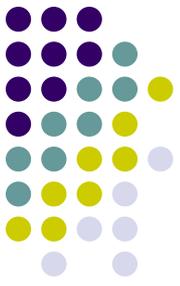
$$f = AB \quad \text{and} \quad G = f + CD$$

$$\rightarrow \frac{dG}{df} = (C' + D') \quad \text{and} \quad \frac{df}{dA} = B$$

$$\rightarrow \frac{dG}{dA} = \left(\frac{dG}{df}\right) \cdot \left(\frac{df}{dA}\right) = (C' + D') \cdot B$$

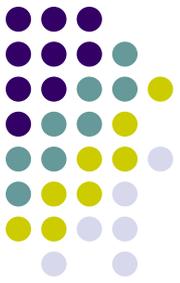
An Input vector  $v$  sensitizes a fault effect from  $A$  to  $G$   
iff  $v$  sensitizes the effect from  $A$  to  $f$  and from  $f$  to  $G$

# Test Generation Based on Structural Analysis

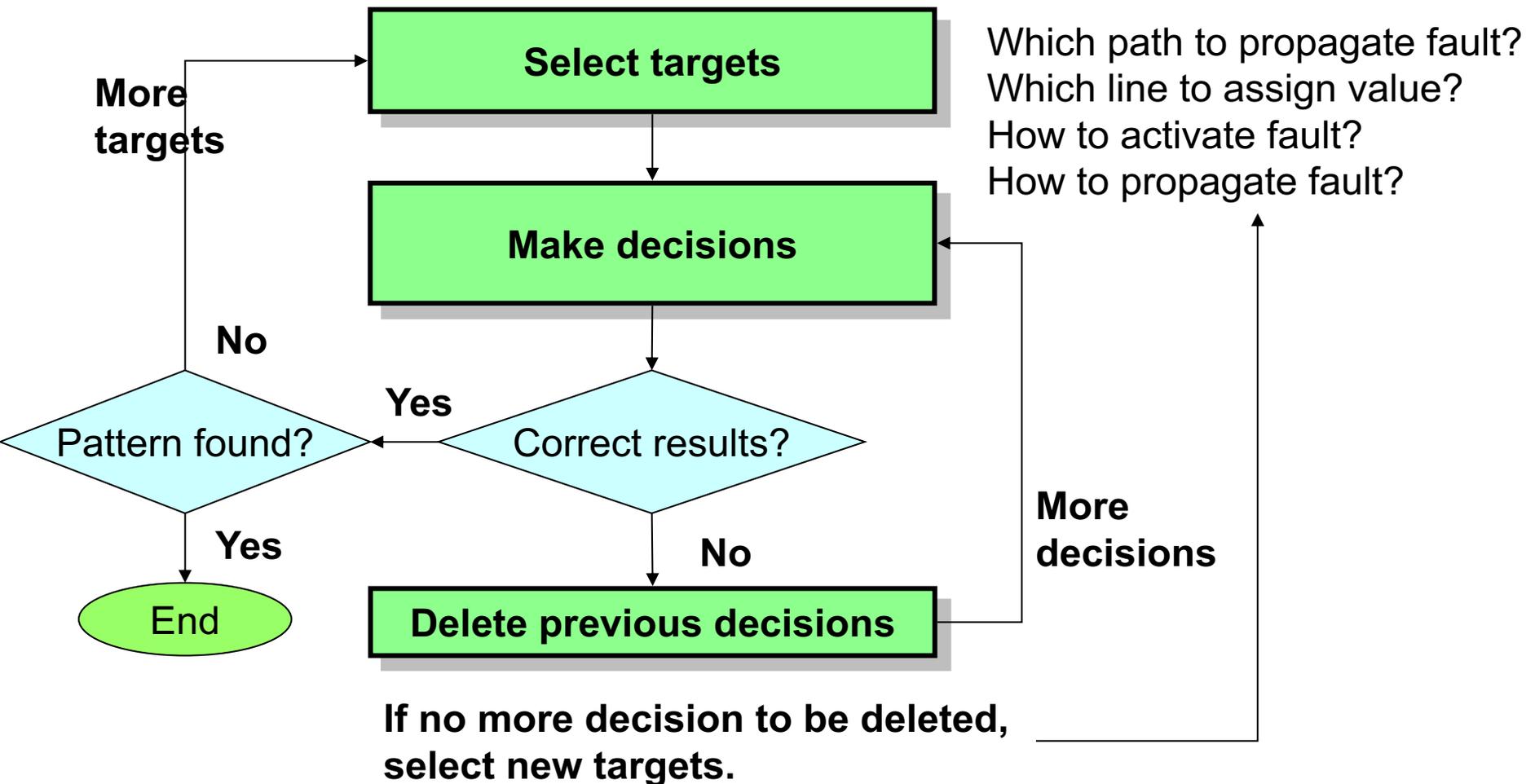


- Important algorithms
  - D-algorithm [Roth 1967]
  - 9-Valued D-algorithm [Cha 1978]
  - PODEM [Goel 1981]
  - FAN [Fujiwara 1983]
- Key techniques
  - Find inputs to (1) **activate**, and (2) **propagate** the fault through sensitized paths to POs
  - Branch and bounds

# Structural Test Generation



- ATPG traverse circuit structures by the following process:

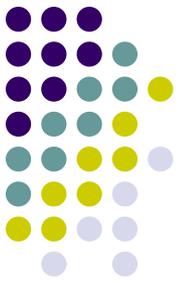


# Common Concepts for Structural Test Generation

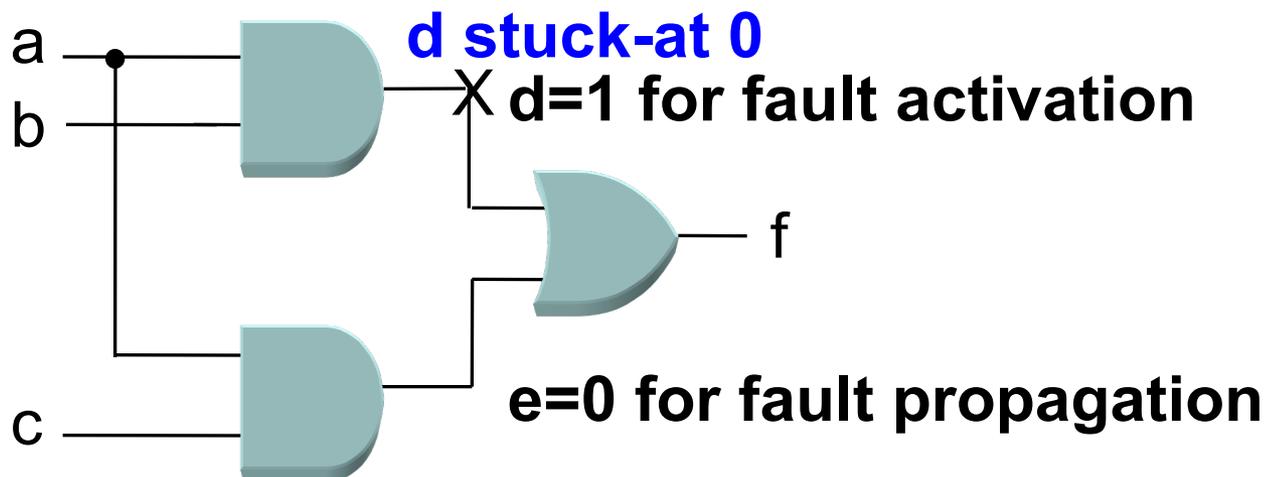


- Fault activation
  - Set the faulty signal to either 0 or 1
  - A line justification problem
- Fault propagation
  - Select propagation paths to POs
  - Inputs to the gates on the propagation paths are set to non-controlling values if the inputs are not on the path.
    - Side inputs are set to non-controlling values.
  - Multiple line justification problems
- Line justification
  - Find inputs that force certain signals to be 0 or 1.

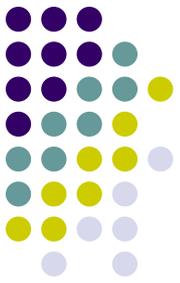
# Examples of Fault Activation and Propagation for Stuck-at Faults



- Fault activated by inverting the signal value:  $d=1$ .
- Fault propagation
  - Fault propagated to  $f$  by  $e=0$ .
- Line justification
  - To assign  $d=1$ , we need  $(a\ b)=(1\ 1)$ .
  - To assign  $e=0$ , we need  $c=0$ .

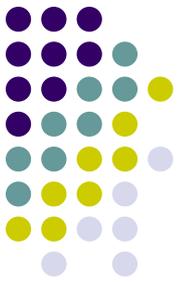


# What are the Decision Points?



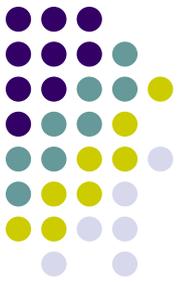
- On fault propagation
  - We **select** propagation paths to POs.
  - Involve decisions on which paths to choose.
- On line justification
  - Given a signal to justify, we need to make a decision on which inputs to set.
  - For example, to set the output of an OR gate to 1, we need to choose which input to set to 1.
- Backtrack
  - If we make a wrong decision (guess), we return and erase the decision (, and make another one).
  - All decisions are recorded.

# Branch-and-Bound Search

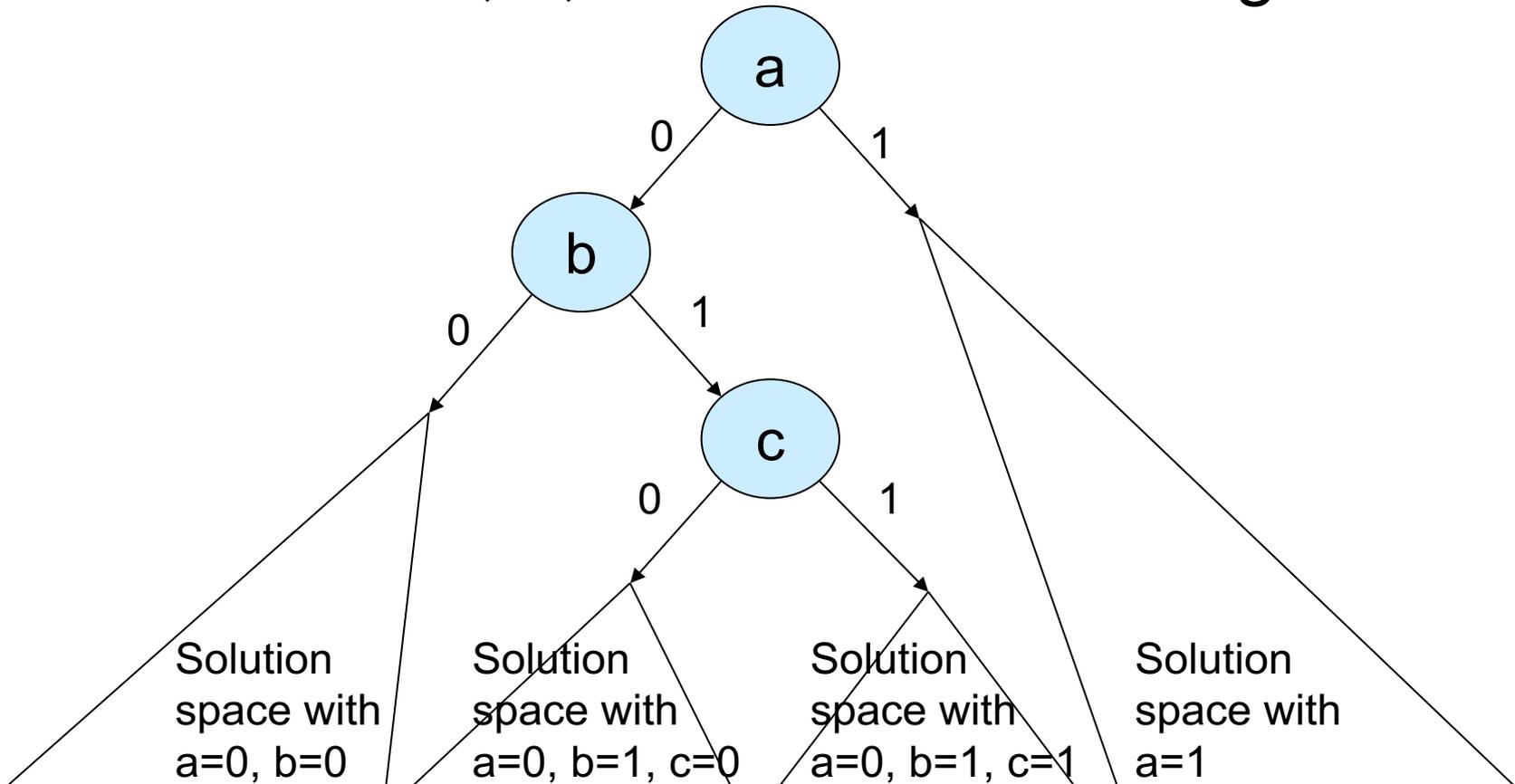


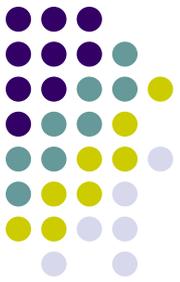
- Test Generation is a branch-and-bound search
  - Every decision point is a **branching** point
  - If a set of decisions lead to a **conflict** (or **bound**), a **backtrack** is taken to explore other decisions
- A test is found when
  - (1) fault effect is propagated to a PO
  - (2) all internal lines are justified
- Since the search is **exhaustive**, it will find a test if one exists
- No test is found after all possible decisions are tried
  - Target fault is **undetectable**

# An Illustration of Branch and Bounds

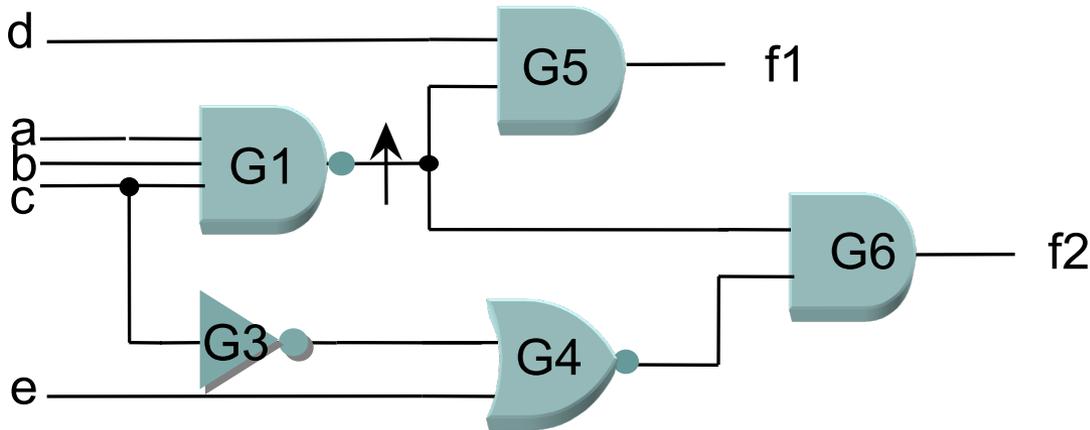


- Assume  $a$ ,  $b$ , and  $c$  are internal signals

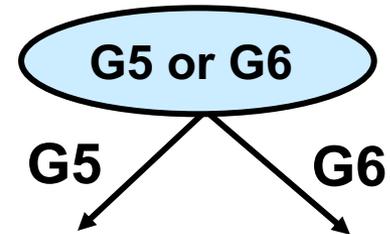




# Decision on Fault Propagation Path

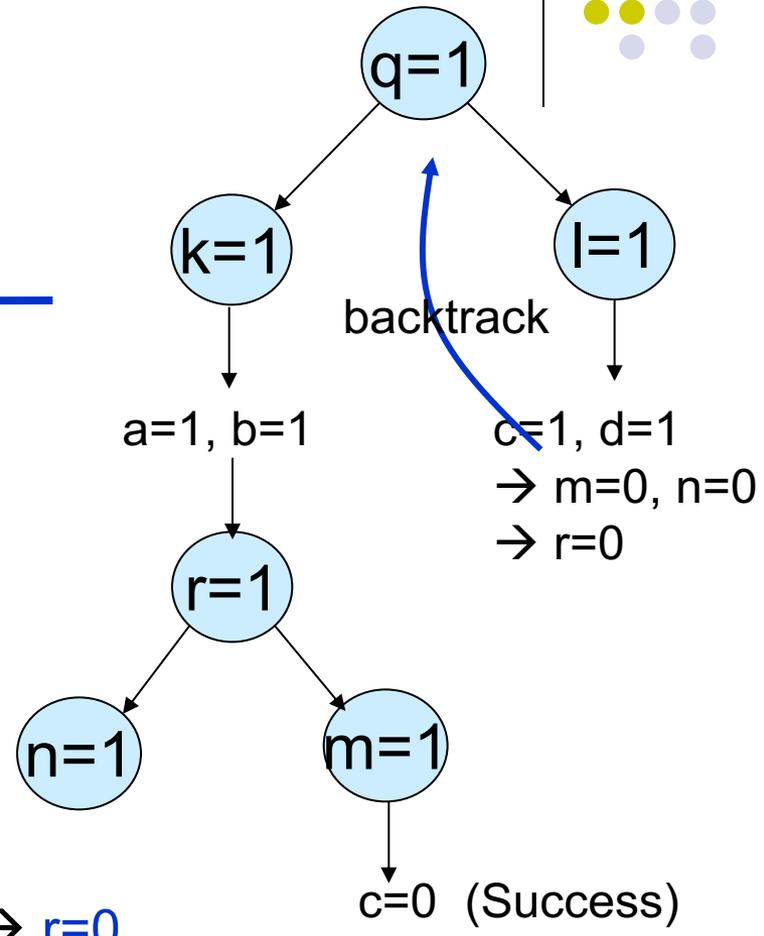
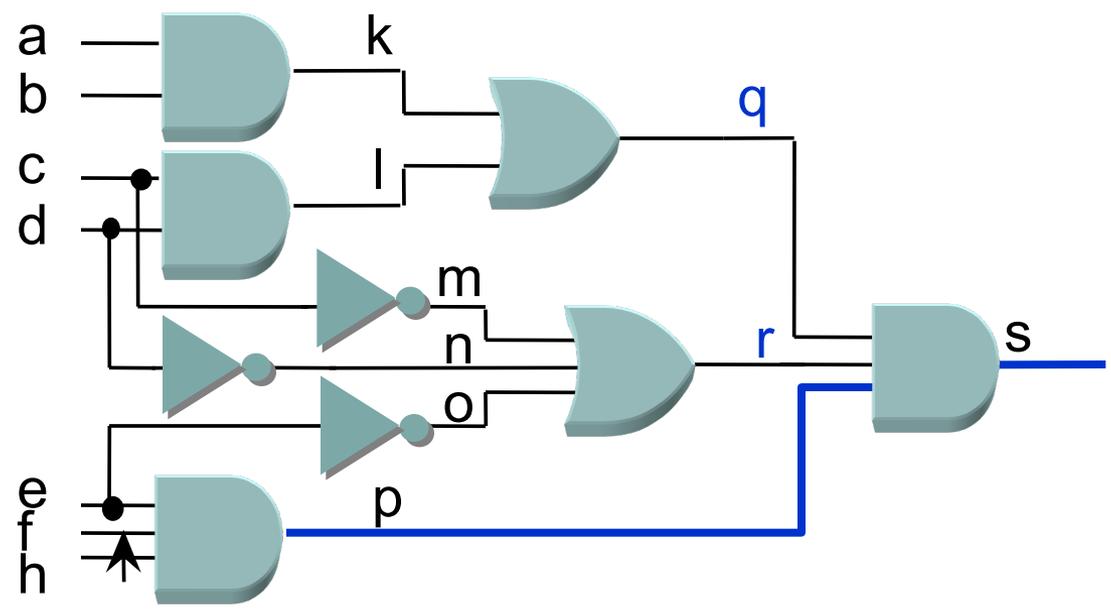


- Decision tree before we start:
- Fault activation
  - $G1=0 \rightarrow \{a=1, b=1, c=1\} \rightarrow G3=0$
- Fault propagation through G5 or G6
- Decision through G5:
  - $\rightarrow d=1 \rightarrow$  The resulting test is (1111X)
- Decision through G6:
  - $\rightarrow G4=1 \rightarrow e=0 \rightarrow$  The resulting test is (111x0)



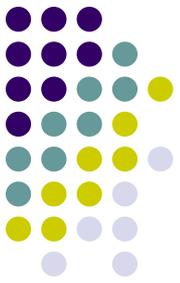


# Decisions On Line Justification

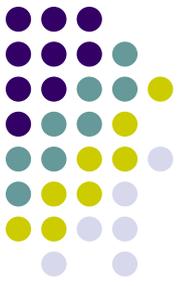


- Fault activation  $\rightarrow$  set h to 0
- Fault propagation  $\rightarrow e=1, f=1 \rightarrow o=0$
- Fault propagation  $\rightarrow q=1, r=1$
- To justify  $q=1 \rightarrow l=1$  or  $k=1$  Decision point
- Decision:  $l=1 \rightarrow c=1, d=1 \rightarrow m=0, n=0 \rightarrow r=0$   
 $\rightarrow$  inconsistency at r  $\rightarrow$  backtrack !
- Decision:  $k=1 \rightarrow a=1, b=1$
- To justify  $r=1 \rightarrow m=1$  or  $n=1$  ( $\rightarrow c=0$  or  $d=0$ )  $\rightarrow$  Done !

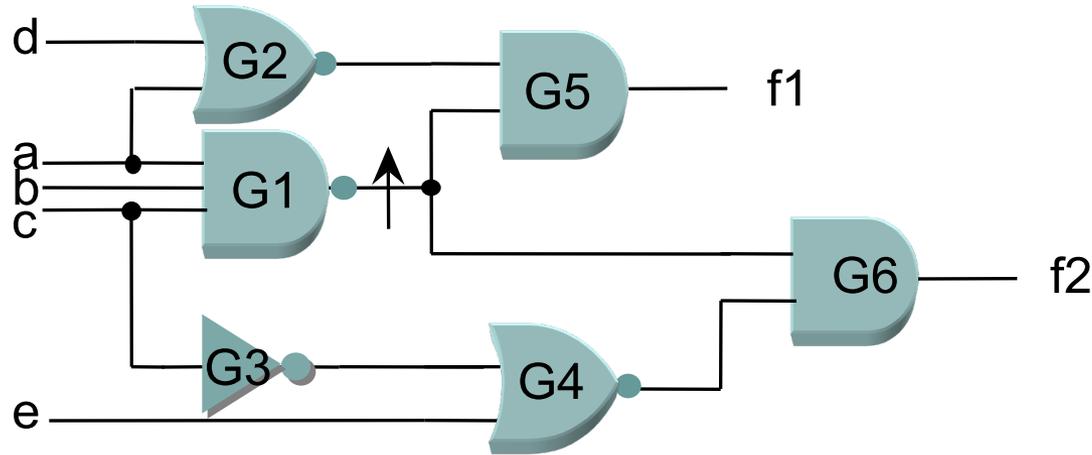
# Implications



- Implications
  - Computation of the values that can be uniquely determined
    - **Local implication**: propagation of values from one line to its immediate successors or predecessors
    - **Global implication**: the propagation involving a larger area of the circuit and re-convergent fanout
- Maximum Implication Principle
  - Perform as many implications as possible
  - It helps to either reduce the number of problems that need decisions or to **reach an inconsistency sooner**

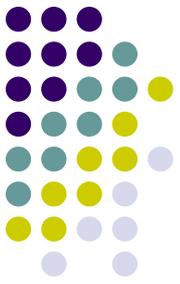


# An Example of Implication to Influence Decisions



- Fault activation
  - $G1=0 \rightarrow \{a=1, b=1, c=1\} \rightarrow \{G3=0, G2=0\} \rightarrow G5=0$
- Fault propagation through G6 only
- Decision through G6:
  - $\rightarrow G4=1 \rightarrow e=0 \rightarrow$  test pattern is (111x0)

# Milestone Structural Test Generation Methods

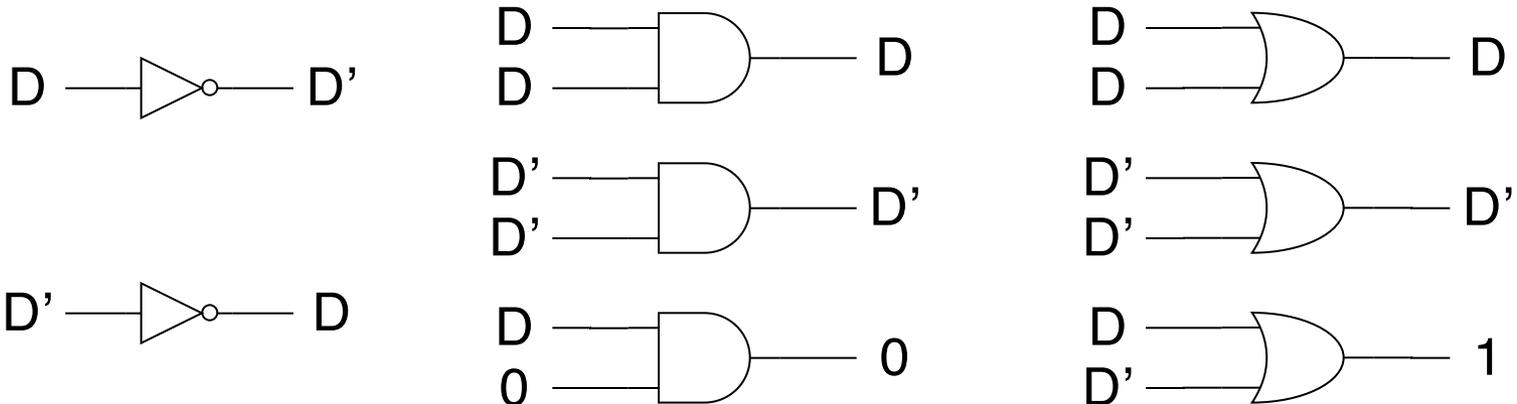


- D-algorithm [Roth 1967]
- 9-Valued D-algorithm [Cha 1978]
- PODEM [Goel 1981]
- FAN [Fujiwara 1983]
- Other advanced techniques

# The D-Algebra

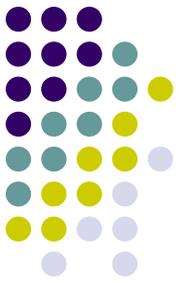


- Allow the representation of the “good” and “faulty” behavior at the same time.
- Formally define/generate decision points in fault activation and propagation
- Introduce the symbol  $D$  in addition to  $0, 1, x$   
 $D = 1/0$  ( $D' \equiv 0/1$ ) represents a signal which has value  $1$  in fault-free circuit and  $0$  in the faulty circuit.



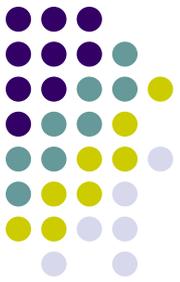
**Forward implication conditions for NOT, AND, OR in D-algebra**

# A Quick Overview of D-Algorithm



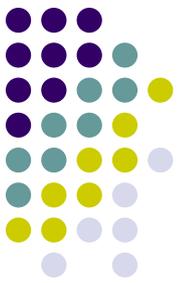
- A structural algorithm
- Use 5-value logic (0, 1, X, D, D')
- Try to activate and propagate fault to outputs first with internal justification.
  - Then apply branch and bounds on line justification problems.

# Fault Activation



- **Specify the minimal input conditions which must be applied to a logic element to produce an error signal at its output.**
- **Stuck-at faults at an AND output:**
  - Stuck-at-0 fault: 11D
  - Stuck-at-1 fault: 0xD' and x0D'
- **More complex fault model, e.g., bridging faults, gate type errors, can also be modeled.**

# Fault Activation of Complex Fault Models



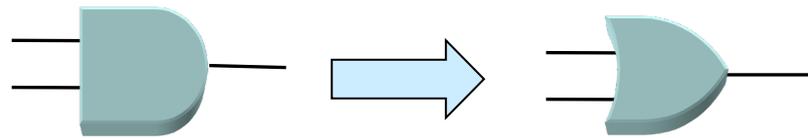
- Apply the concept of Boolean equation method
  - List the set of input patterns for function  $F$ .
  - $F$  is the (local) representation function of the fault site.
  - (1) fault-free output=0  $\{F=0\}$
  - (2) fault-free output=1  $\{F=1\}$
  - (3) faulty output=0  $\{F_f=0\}$
  - (4) faulty output=1  $\{F_f=1\}$
- $\{F=0\} \cap \{F_f=1\}$  or  $\{F=1\} \cap \{F_f=0\}$
- Example: 2-input AND gate output stuck-at 1
  - $\{F=0\}=\{00, 01, 10\}$ ,  $\{F_f=1\}=\{00, 01, 10, 11\}$ ,  $\{F=1\}=\{11\}$ ,  
 $\{F_f=0\}=\{\}$
  - $\{F=0\} \cap \{F_f=1\} = \{00, 01, 10\}$
  - $\{F=1\} \cap \{F_f=0\} = \{\}$

# Example of Complex Fault Models



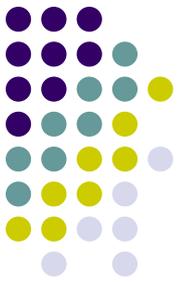
- **Assume the fault is to change an AND gate to an OR gate**

- $\{F=0\}=\{00, 01, 10\}$ ,  $\{F_f=1\}=\{01, 10, 11\}$ ,  $\{F=1\}=\{11\}$ ,  $\{F_f=0\}=\{00\}$
- $\{F=0\} \cap \{F_f=1\} = \{01, 10\}$
- $\{F=1\} \cap \{F_f=0\} = \{\}$



Apply 01, we produce a 0/1 (D') at the output  
Apply 10, we also produce a (D') 0/1.

# D-frontiers and J-frontiers

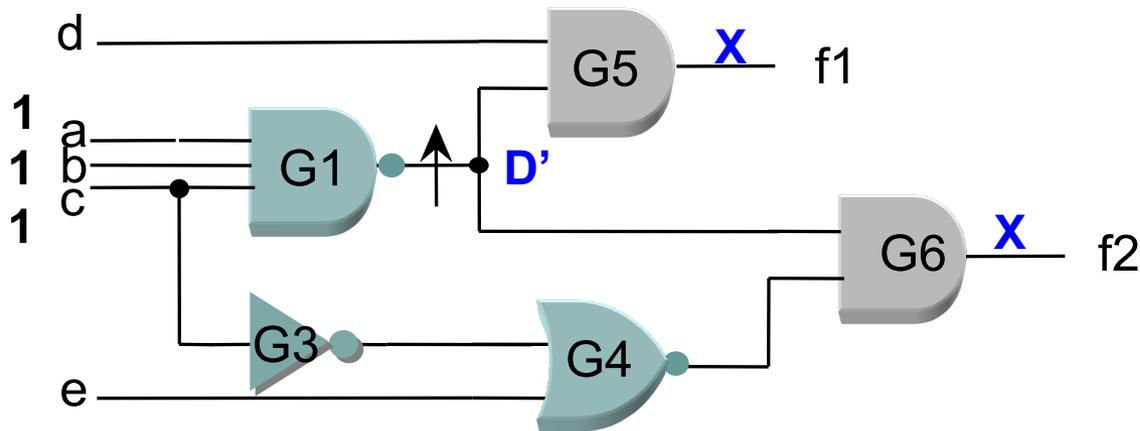


- Two most important data structures in the D-algorithm.
  - Both store a list of gates.
- D-frontiers
  - Where to choose the propagation paths.
  - D-frontiers should not be empty during the search of test patterns.
- J-frontiers
  - Where to perform line justifications.
  - J-frontiers will be empty after a test pattern is found.

# D-frontiers

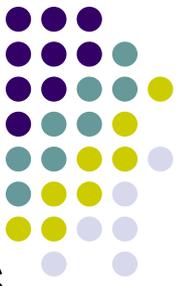


- **D-frontiers** are the gates whose output value is X, while one or more inputs are D or D'.

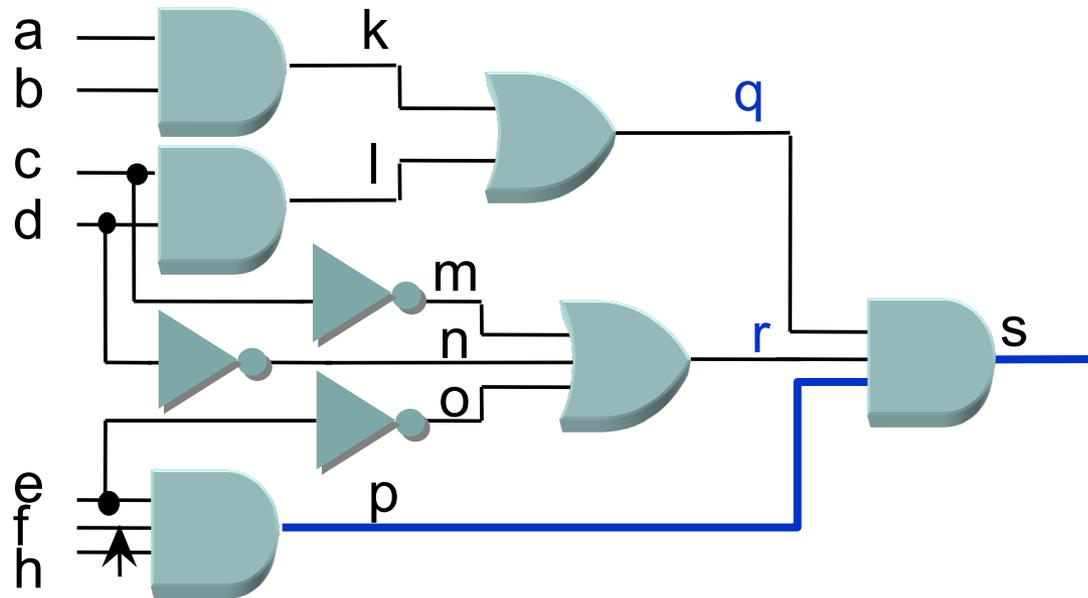


- **Fault activation**
  - $G1=0 \rightarrow \{ a=1, b=1, c=1 \} \rightarrow \{ G3=0 \}$
- **Fault propagation** can be done through either G5 or G6
  - $D\text{-frontiers}=\{G5, G6\}$

# J-frontiers

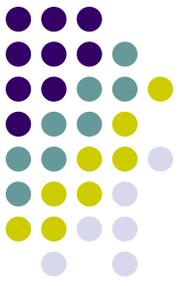


- **J-frontier**: is the set of gates whose output value is known (i.e., 0 or 1), but is not implied by its input values.

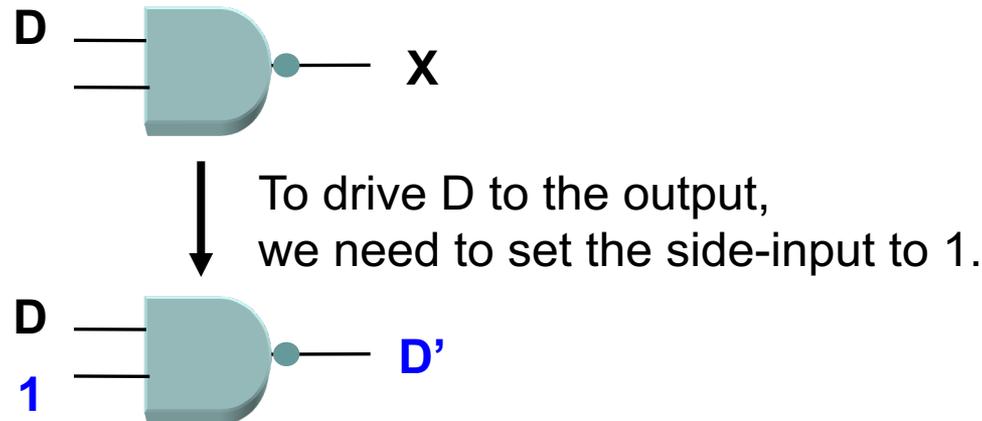


- **Fault activation** → set h to 0
- **Fault propagation** →  $e=1, f=1 \rightarrow o=0$
- **Fault propagation** →  $q=1, r=1$
- We need to justify both  $q=1$  and  $r=1$ : **J-frontiers**={q, r}.

# D-drive Function

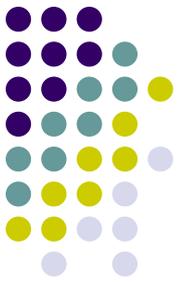


- The D-drive selects a gate in the D-frontier and attempts to propagate the D and/or D' from its input(s) to its output by setting side-inputs to non-controlling values.



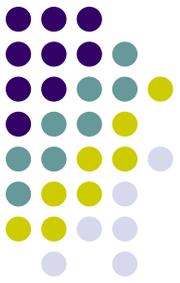
- For more complex functions, we can use the truth table of 5-value logics.

# Implication in D-Algorithm

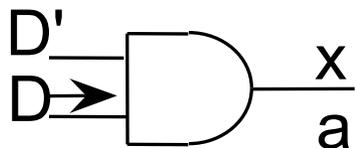
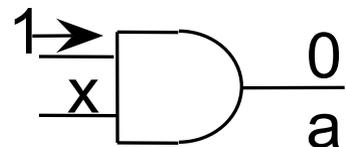
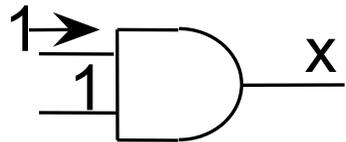
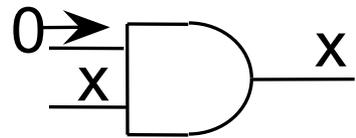


- Involve  $D$  and  $D'$  in the implication.
- Basic principles are guided by the truth table of each logic gate (complex cell).
- Examples are given in the following pages.

# Local Implications considering D/D' (Forward)



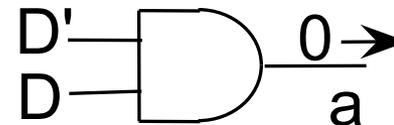
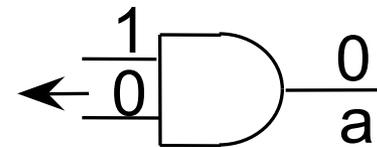
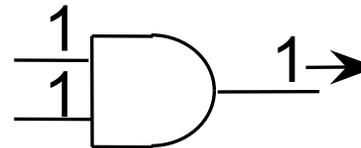
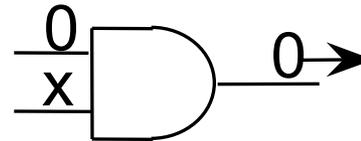
Before



J-frontier={ ...,a }

D-frontier={ ...,a }

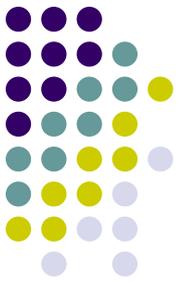
After



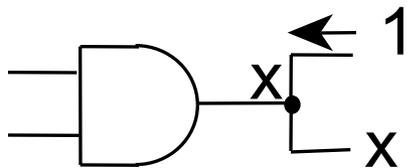
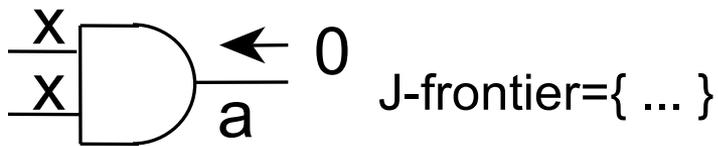
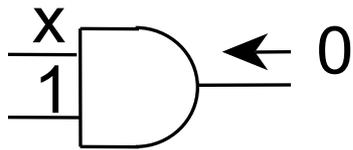
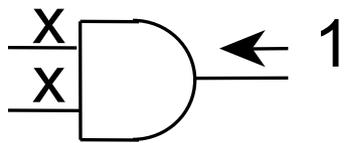
J-frontier={ ... }

D-frontier={ ... }

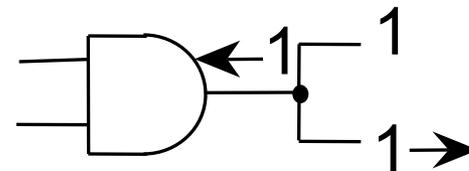
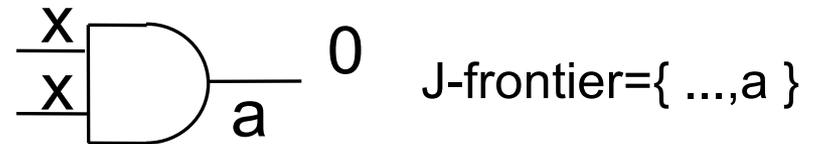
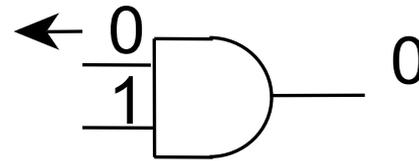
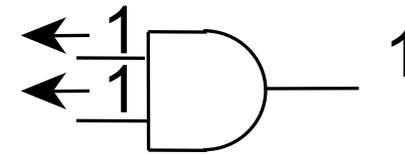
# Local Implications considering D/D' (Backward)



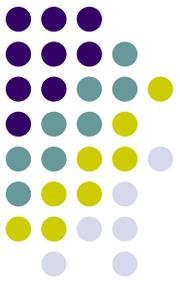
Before



After

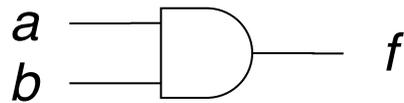


# Backward local Implication by Prime Cubes



- **Prime Cubes**

- A simplified form of truth table.
- Circle groups of 0 or 1 in the Karnaugh map.

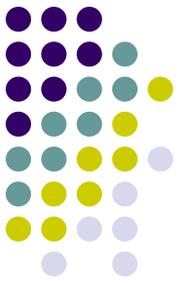


	<i>a</i>	0	1
<i>b</i>	0	0	0
1	0	1	

Prime cubes of AND

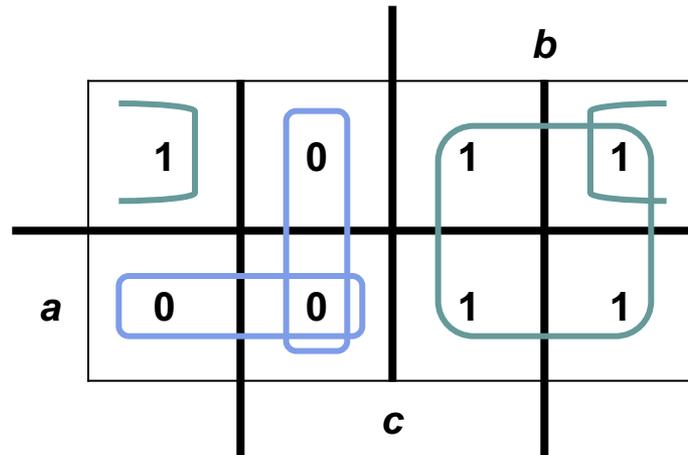
<i>a</i>	<i>b</i>	<i>f</i>
1	1	1
0	x	0
x	0	0

# Backward Implication for Complex Cells



Find the prime cubes of the function

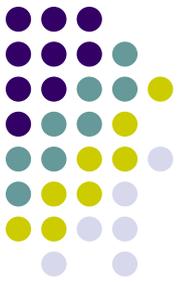
$$f(a,b,c) = a'c' + b$$



<i>a</i>	<i>b</i>	<i>c</i>	<i>f</i>
0	x	0	1
x	1	x	1
1	0	x	0
x	0	1	0

In this example, if we want to justify  $f=0$  and we know  $c=1$ , then we also obtain the backward implication of  $ab=X0$

# Checking Consistency



- We might assign different requirements from different decisions.
- We have to make sure every signal have compatible assignment in the circuit.

Consistency check for 5-V logic

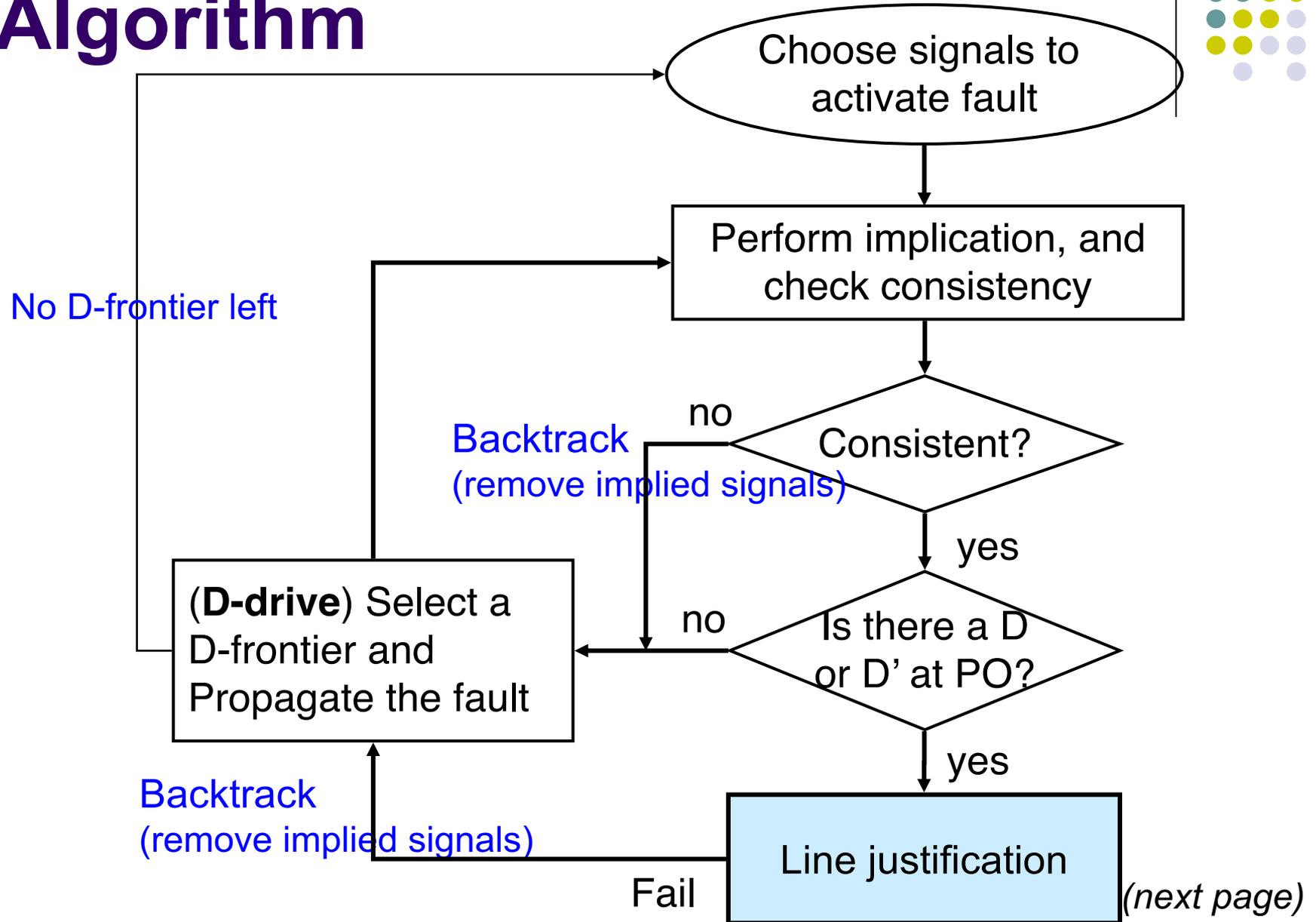
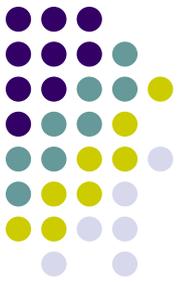
	0	1	X	D	D'
0	0	$\phi$	0	$\Phi$	$\Phi$
1	$\phi$	1	1	$\Phi$	$\Phi$
X	0	1	X	D	D'
D	$\Phi$	$\Phi$	D	D	$\Phi$
D'	$\Phi$	$\Phi$	D'	$\Phi$	D'

# Terminating Conditions for D-Algorithm

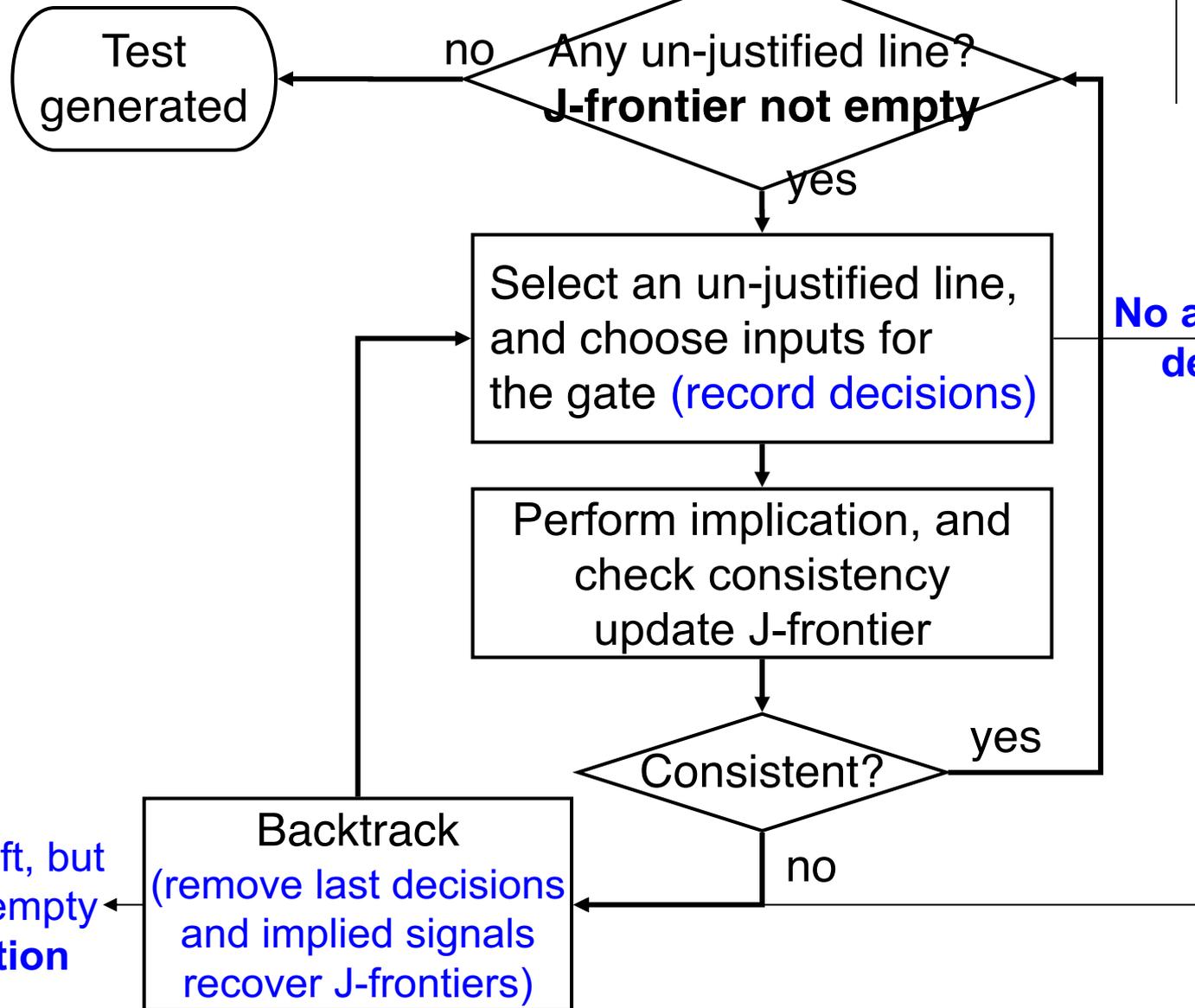
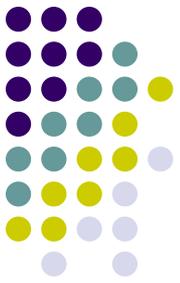


- Success:
  - (1) Fault effect at an output (D-frontier may not be empty)
  - (2) J-frontier is empty
- Failure:
  - (1) D-frontier is empty and fault is not at POs
  - (2) No decision left but J-frontier is not empty

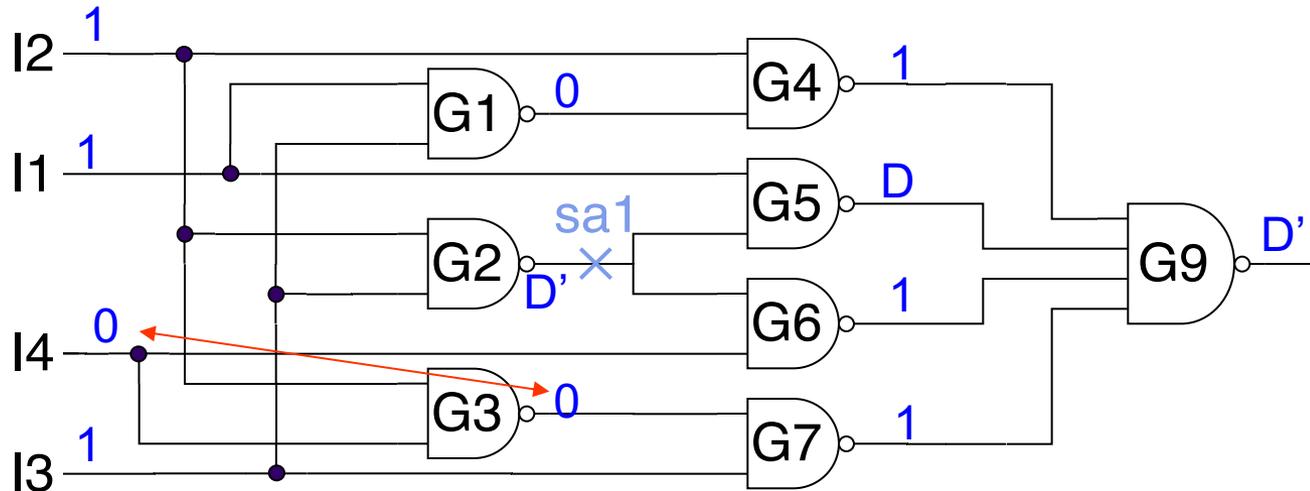
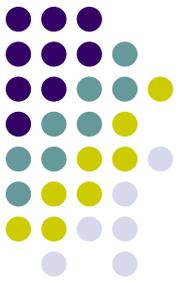
# Propagation Phase of the D-Algorithm



# Line Justification Phase

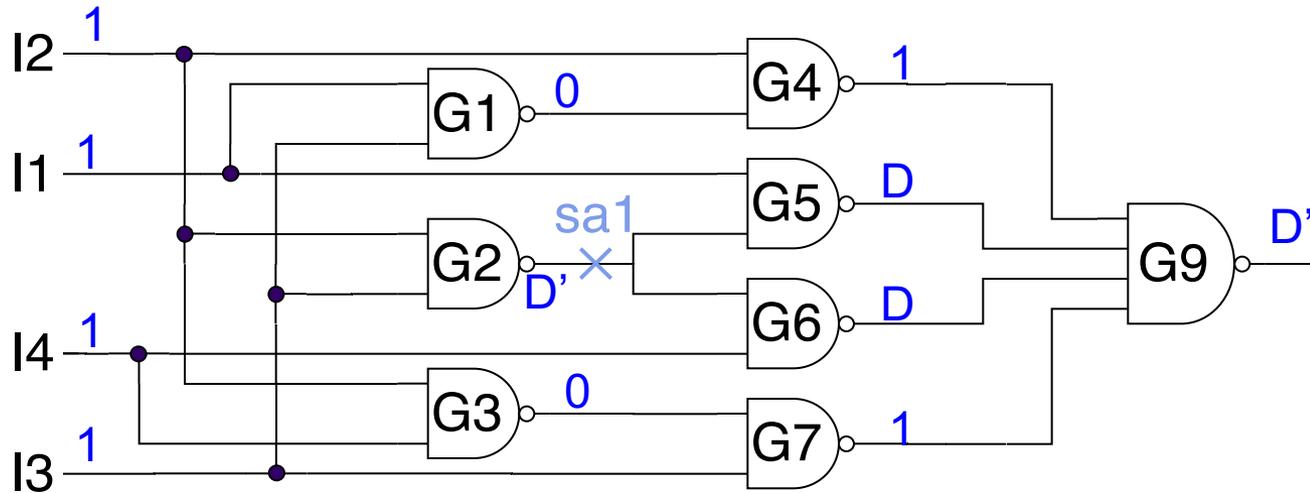
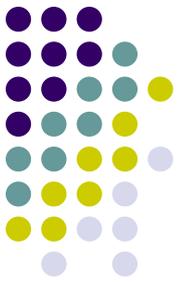


# D-Algorithm Example



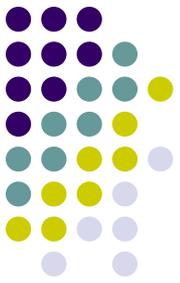
- **Fault activation** → set G2 to 0 →  $I2=1, I3=1$ , D-Frontier={G5, G6}, J-Frontier={}
- **Fault propagation** → select G5 →  $I1=1 \rightarrow G1=0 \rightarrow G4=1$ , D-Frontier={G6, G9}, J-Frontier={}
- **Fault propagation** → select G9 →  $G6=1, G7=1 \rightarrow I4=0, G3=0 \rightarrow I4=1$  (Contradictory) → propagation fails, backtrack to select another D-frontier from {G6}.

# D-Algorithm Example (Cont.)



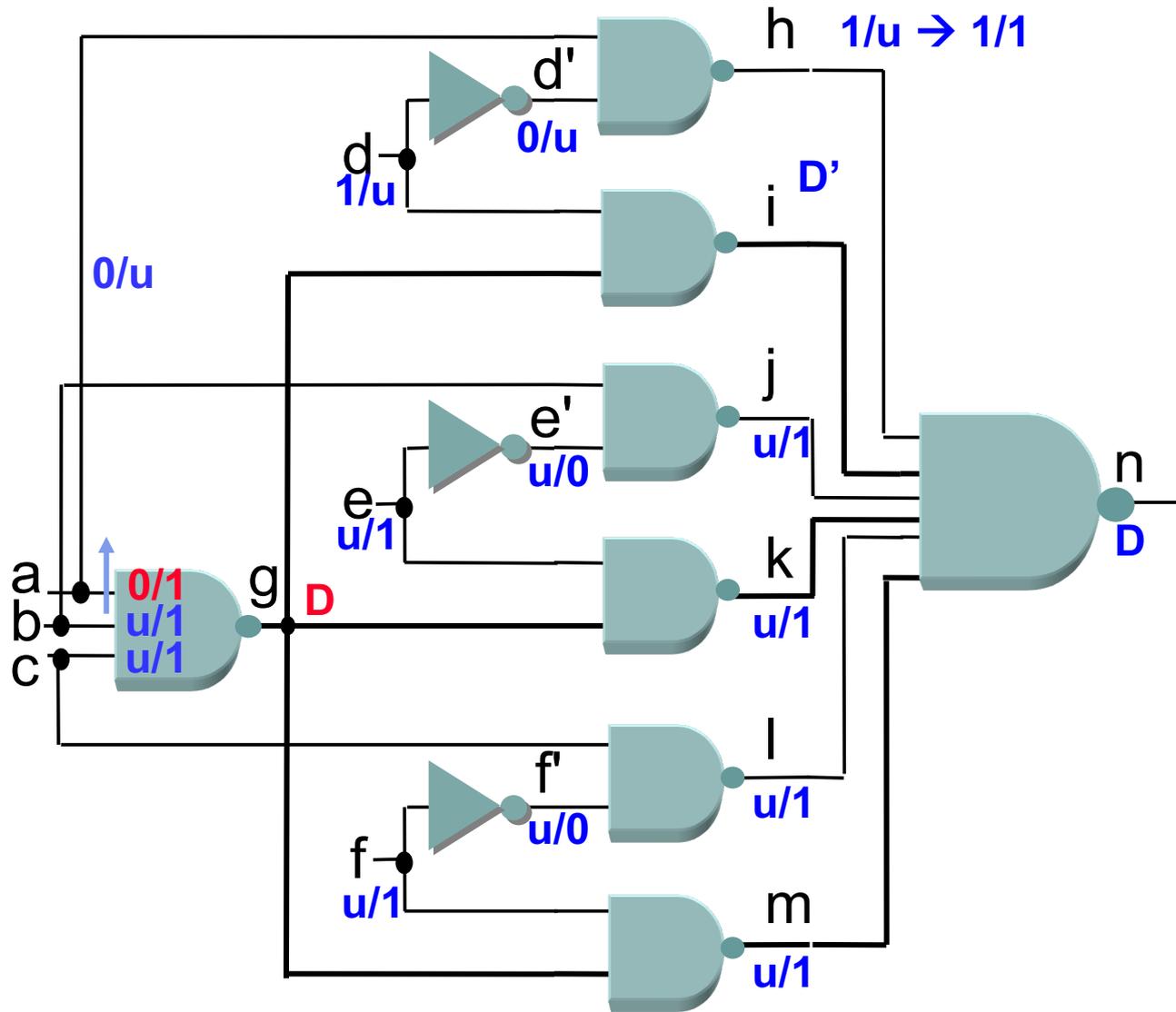
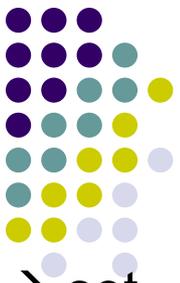
- Fault propagation  $\rightarrow$  select G6  $\rightarrow$  I4=1, G3=0, G7=1

# 9-Value D-Algorithm

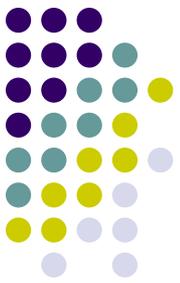


- Logic values (fault-free / faulty)
  - $\{0/0, 0/1, 0/u, 1/0, 1/1, 1/u, u/0, u/1, u/u\}$ ,
  - where  $0/u=\{0,D'\}$ ,  $1/u=\{D,1\}$ ,  $u/0=\{0,D\}$ ,  
 $u/1=\{D',1\}$ ,  $u/u=\{0,1,D,D'\}$ .
- Advantage:
  - Automatically considers **multiple-path sensitization**, thus reducing the amount of search in D-algorithm
  - The speed-up is **NOT** very significant in practice because most faults are detected through single-path sensitization

# Example: 9-Value D-Algorithm



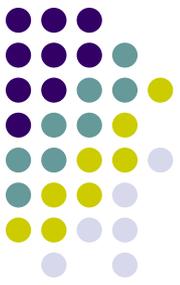
- **Fault activation** → set a to  $0/u \rightarrow h=1/u$ , **D-Frontier**={g}
- **Fault propagation** → select g →  $b=u/1$ ,  $c=u/1 \rightarrow$  **D-frontier**={i, k, m}
- **Fault propagation** → select i →  $d=1/u \rightarrow$  **D-frontier**={k, m, n}
- **Fault propagation** → select n →  $h=1$ ;  $j, k, l, m=u/1 \rightarrow$  **J-frontier**={j, k, l, m}
- **Justify j** →  $e'=u/0 \rightarrow e=u/1$  (consistent)
- Similarly  $f=u/1$



# Selecting Inputs for 9-Value D-Algorithm

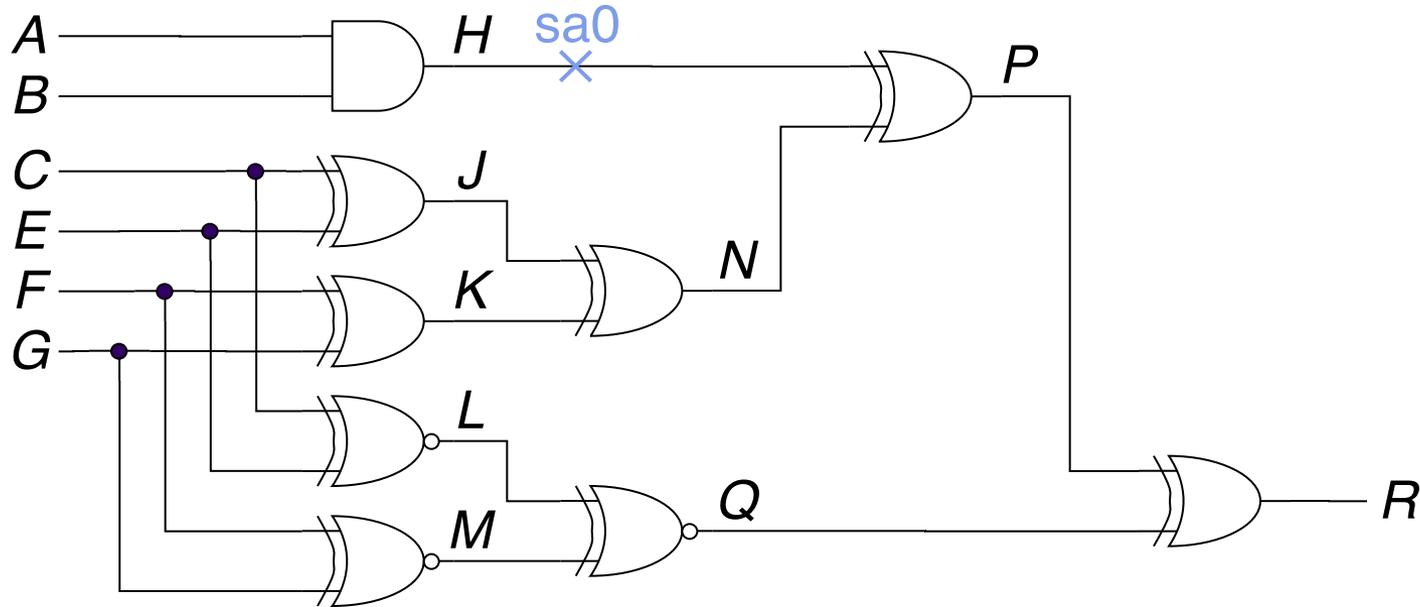
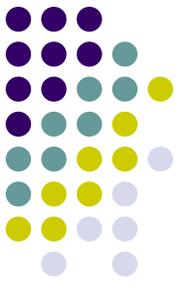
- To derive the input test vector, we choose the values consistent with both fault-free and faulty circuits.
  - $A = (0/u) = \{0, D'\} \rightarrow 0$
  - $B = (1/u) = \{1, D\} \rightarrow 1$
  - $C = (1/u) = \{1, D\} \rightarrow 1$
  - $D = (u/1) = \{1, D'\} \rightarrow 1$
  - $E = (u/1) = \{1, D'\} \rightarrow 1$
  - $F = (u/1) = \{1, D'\} \rightarrow 1$
- The final vector
  - $(A, B, C, D, E, F) = (0, 1, 1, 1, 1, 1)$

# Problems with the D-Algorithm



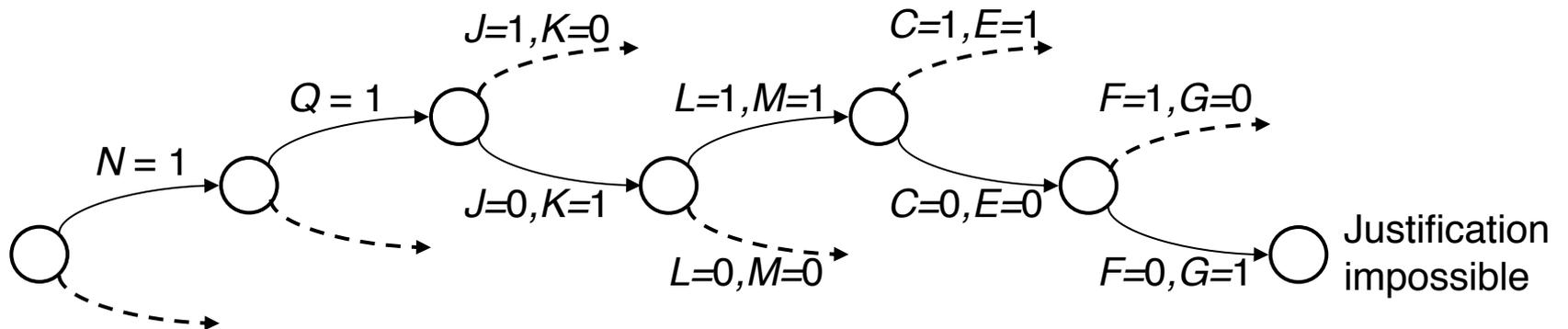
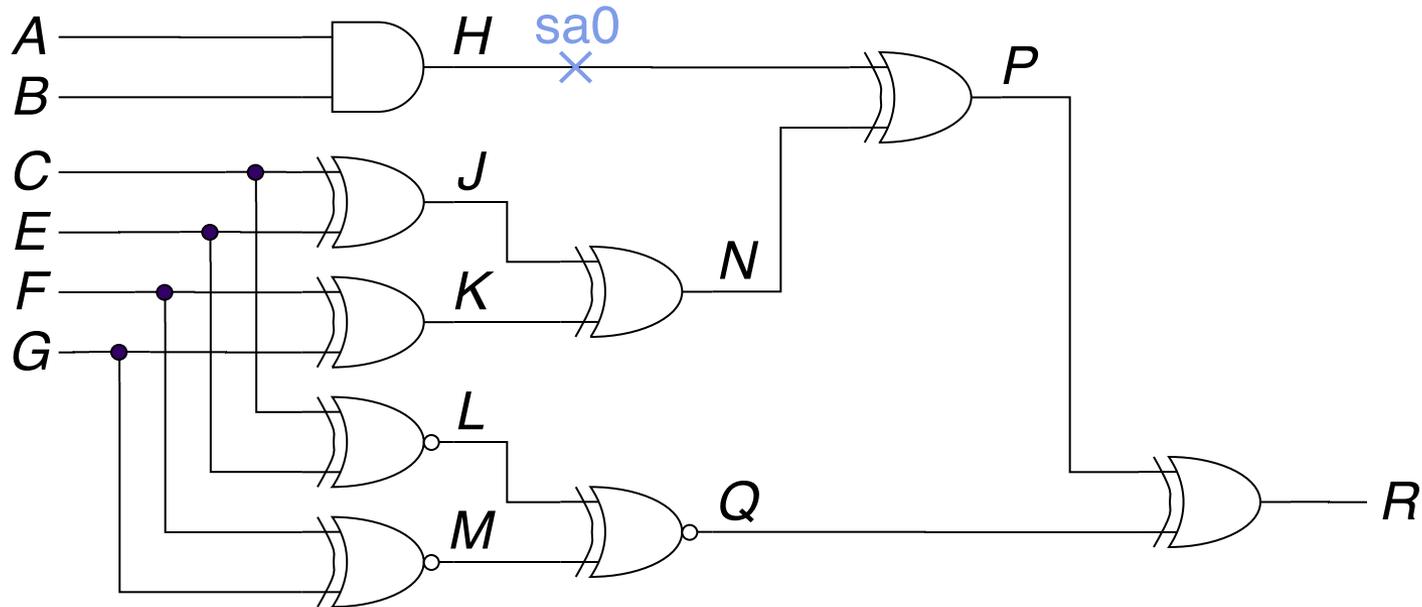
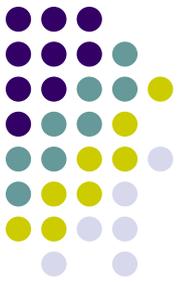
- Assignment of values is allowed for internal signals
  - Large search space
  - Backtracking could occur at each gate
  - TG is done through indirect signal assignment for FA, FP, and LJ, that eventually maps into assignments at PI's
  - The decision points are at **internal lines**
  - The worst-case number of backtracks is exponential in terms of the number of decision points (e.g., at least  $2^k$  for  $k$  decision nodes)
  - D-algorithm will continue even when D-frontier is empty
- Inefficient for large circuits and some special classes of circuits
  - Example: ECAT (error-correction-and-translation) circuits.

# ECAT Circuits

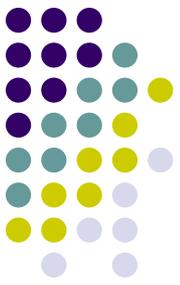


- Primitive D-cube:  $A = B = 1, H = D$
- To propagate through  $P$  to output  $R$ 
  - One possible choice is  $N = Q = 1$ , but this is impossible.
- *D-algorithm will exhaustively enumerate all internal signals to confirm that  $N = Q = 1$  is impossible.*

# The Decision Tree

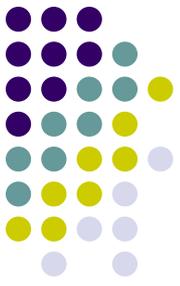


# PODEM: Path-Oriented DEcision Making

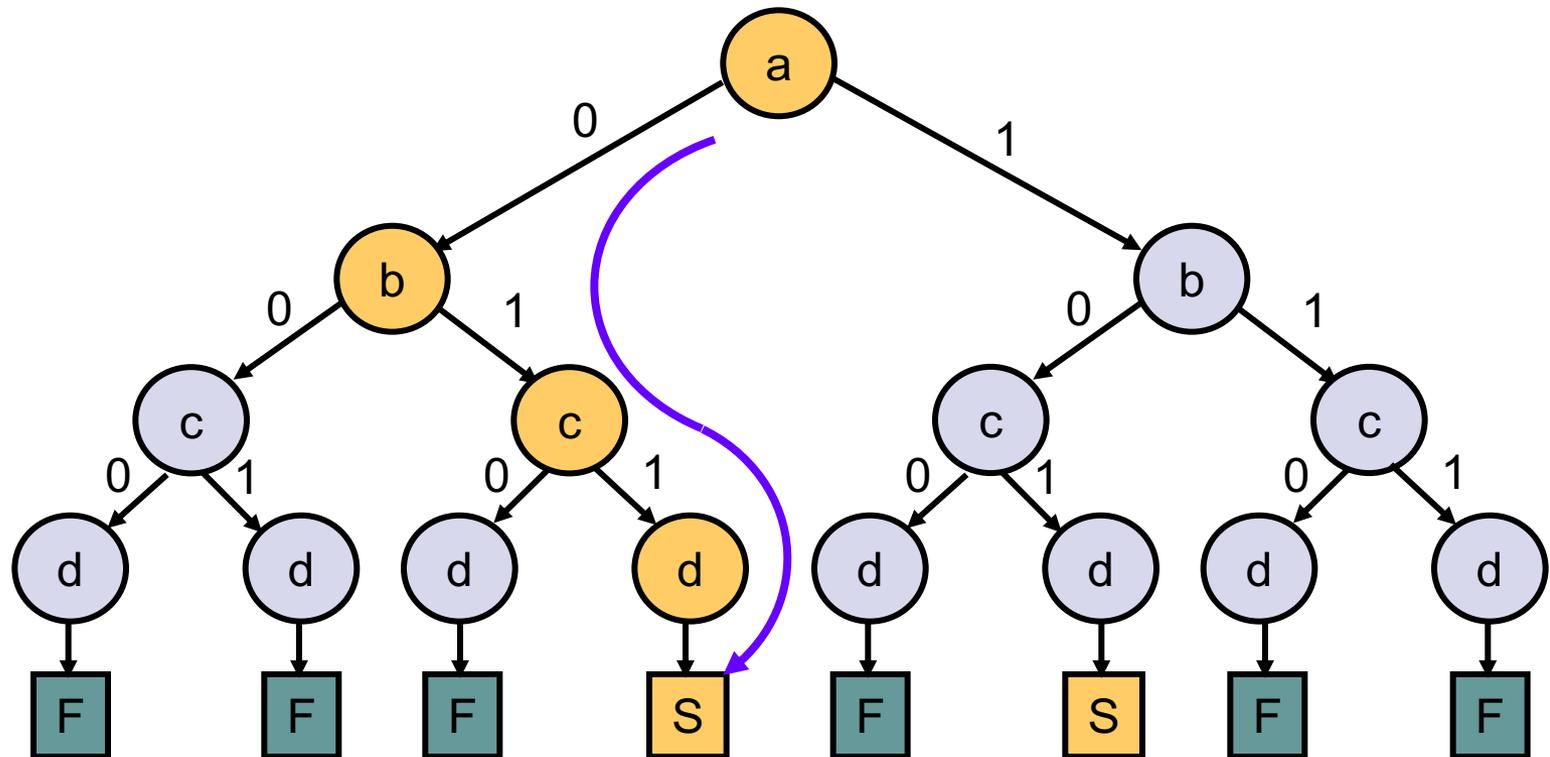


- The test generation is done through a sequence of **direct assignments at PI's**
- Decision points are at PIs, thus the number of backtracking might be fewer
- Also a structural algorithm
- Also use 5-value logic and D-frontiers.
- Also a branch and bounds algorithm.
- In PODEM, to activate and propagate faults, a series of **objective** line justifications are selected (similar to D-algorithm), but then each objective is **backtraced** to PIs and a forward simulation is used to perform implications and confirm consistency .

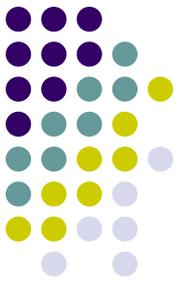
# Search Space of PODEM



- Complete Search Space
  - A binary tree with  $2^n$  leaf nodes, where  $n$  is the number of PI's
  - For example, {a, b, c, d} are PIs in the following binary tree



# PODEM: Recursive Algorithm



PODEM ()

begin

If(error at PO) return(SUCCESS);

If(D-Frontier is empty) return(FAILURE); /\* terminating conditions\*/

(k,  $v_k$ ) = **Objective()**; /\* choose a line, k, to be justified \*/

(j,  $v_j$ ) = **Backtrace(k,  $v_k$ )**; /\* choose the PI, j, to be assigned \*/

**Imply** (j,  $v_j$ ); /\* make a decision \*/

If ( **PODEM()** == SUCCESS ) return (SUCCESS);

**Imply** (j,  $v_j'$ ); /\* reverse  $v_j$  \*/

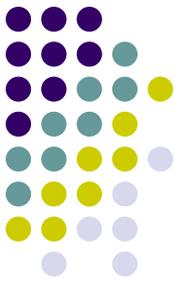
If ( **PODEM()** == SUCCESS ) return(SUCCESS);

**Imply** (j, x); /\* we have wrong objectives \*/

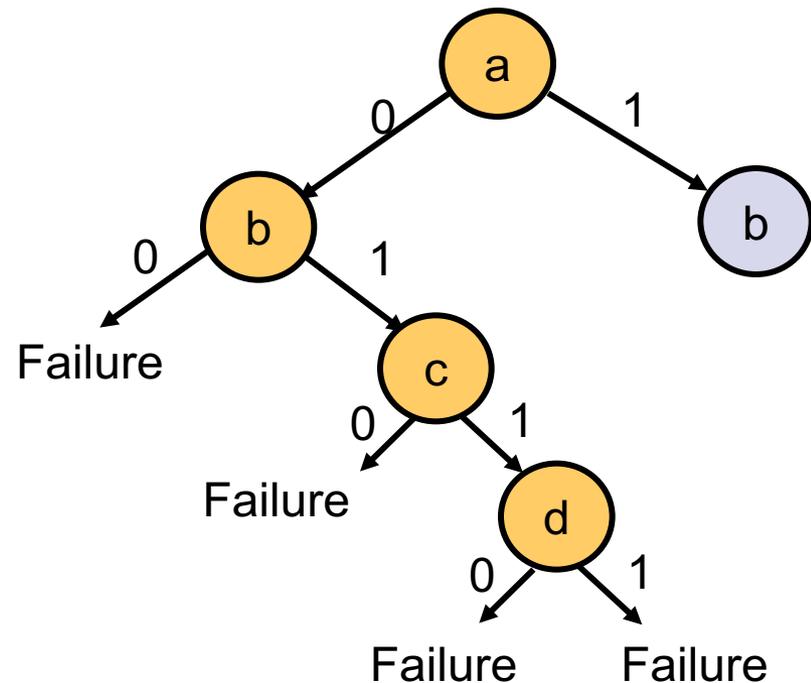
return (FAILURE);

end

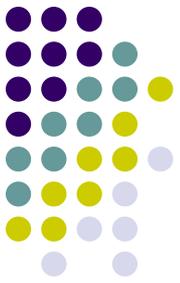
# An Example PI Assignments in PODEM



- Assume we have four PIs: a, b, c, d.
- Decision: a=0
- Decision: b=0 → fails
- Reverse decision: b=1
- Decision: c=0 → fails
- Reverse decision: c=1
- Decision: d=0 → fails
- Decision: d=1 → fails
- Backtrack: d=x
- Backtrack: c=x
- Backtrack: b=x
- Reverse decision: a=1
- ...

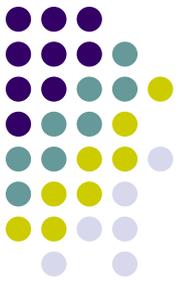


# Objective() and Backtrace()



- Objective()
  - Guide decisions to **activate and propagate** faults.
  - Lead to sets of line justification problems.
  - A signal-value pair  $(w, v_w)$
- Backtrace()
  - Guide decisions to **line justification**.
  - Backtrace maps a objective into a PI assignment that is likely to contribute to the achievement of the objective
    - Traverses the circuit back from the objective signal to PI's
    - Involves finding an all-x path from objective site to a PI, i.e., every signal in this path has value x
    - A PI signal-value pair  $(j, v_j)$
  - No signal value is actually assigned during backtrace !

# Objective() Routine



```
Objective() {
```

```
    /* The target fault is w s-a-v */
```

```
    if (the value of w is x) obj = (w, v');
```

```
    else {
```

```
        select a gate (G) from the D-frontier;
```

```
        select an input (j) of G with value x;
```

```
        c = controlling value of G;
```

```
        obj = (j, c');
```

```
    }
```

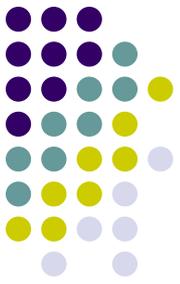
```
    return (obj);
```

```
}
```

fault activation

fault propagation

# Backtrace() Routine



```
Backtrace(w, vw) {
```

```
/* Maps objective into a PI assignment */
```

```
G = w;
```

```
v = vw;
```

```
while (G is a gate output) { /* not reached PI yet */
```

```
    inv = inversion of G; /* inv=1 for INV, NAND, NOR*/
```

```
    select an input (j) of G with value x;
```

```
    G = j; /* new objective node */
```

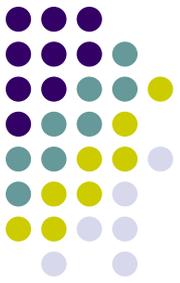
```
    v = v ⊕ inv; /* new objective value */
```

```
}
```

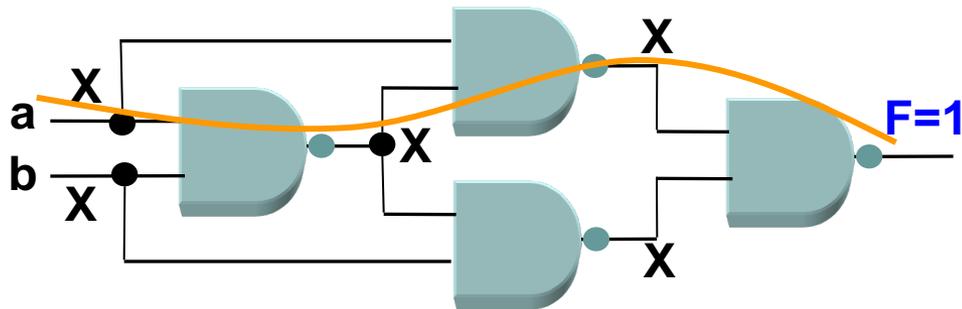
```
return (G, v); /* G is a PI */
```

```
}
```

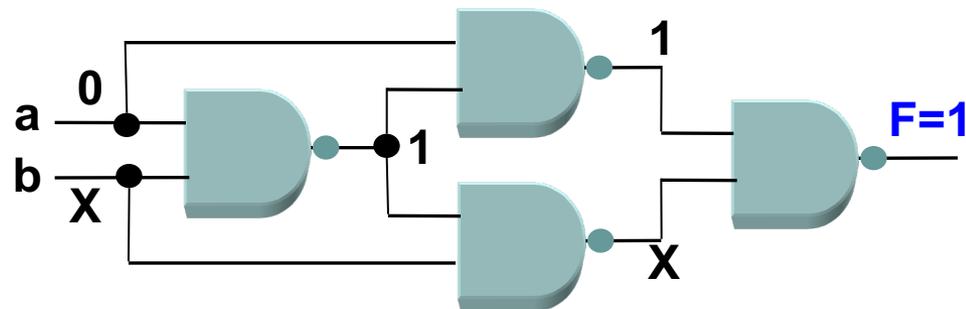
# Backtrace Example



- Objective to achieved: (F, 1)

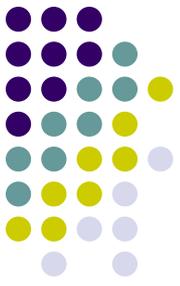


- Backtrace find  $a=0$ ; Forward simulate  $a=0$

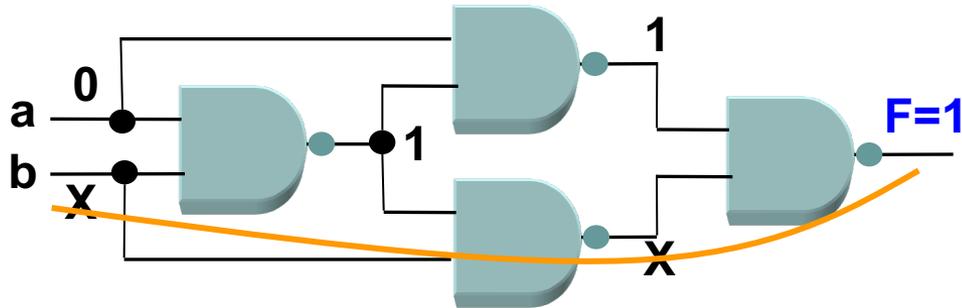


- F is still x after we set  $a=0$

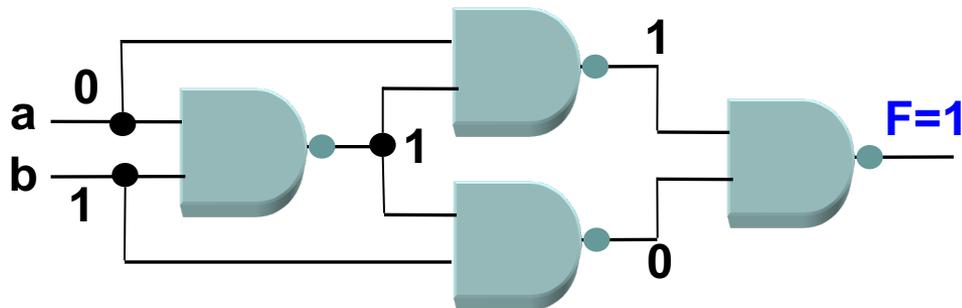
# Backtrace Example (Cont.)



- Objective to achieved: (F, 1)

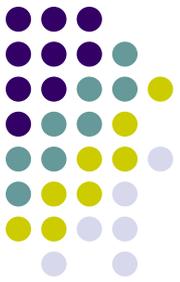


- Backtrace find  $b=1$ ; Forward simulate  $b=1$



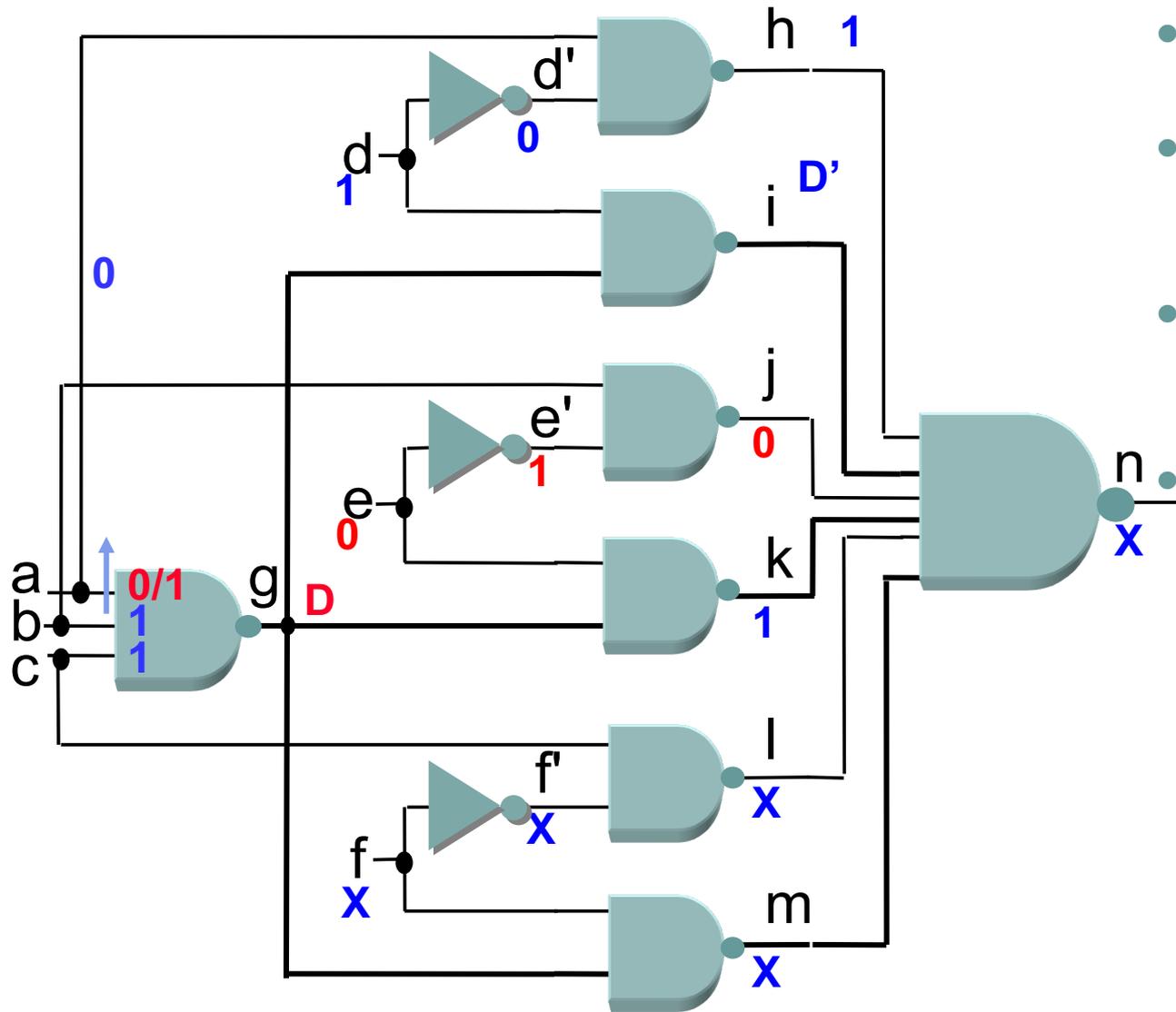
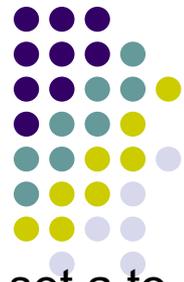
- $F=1$

# Terminating Conditions



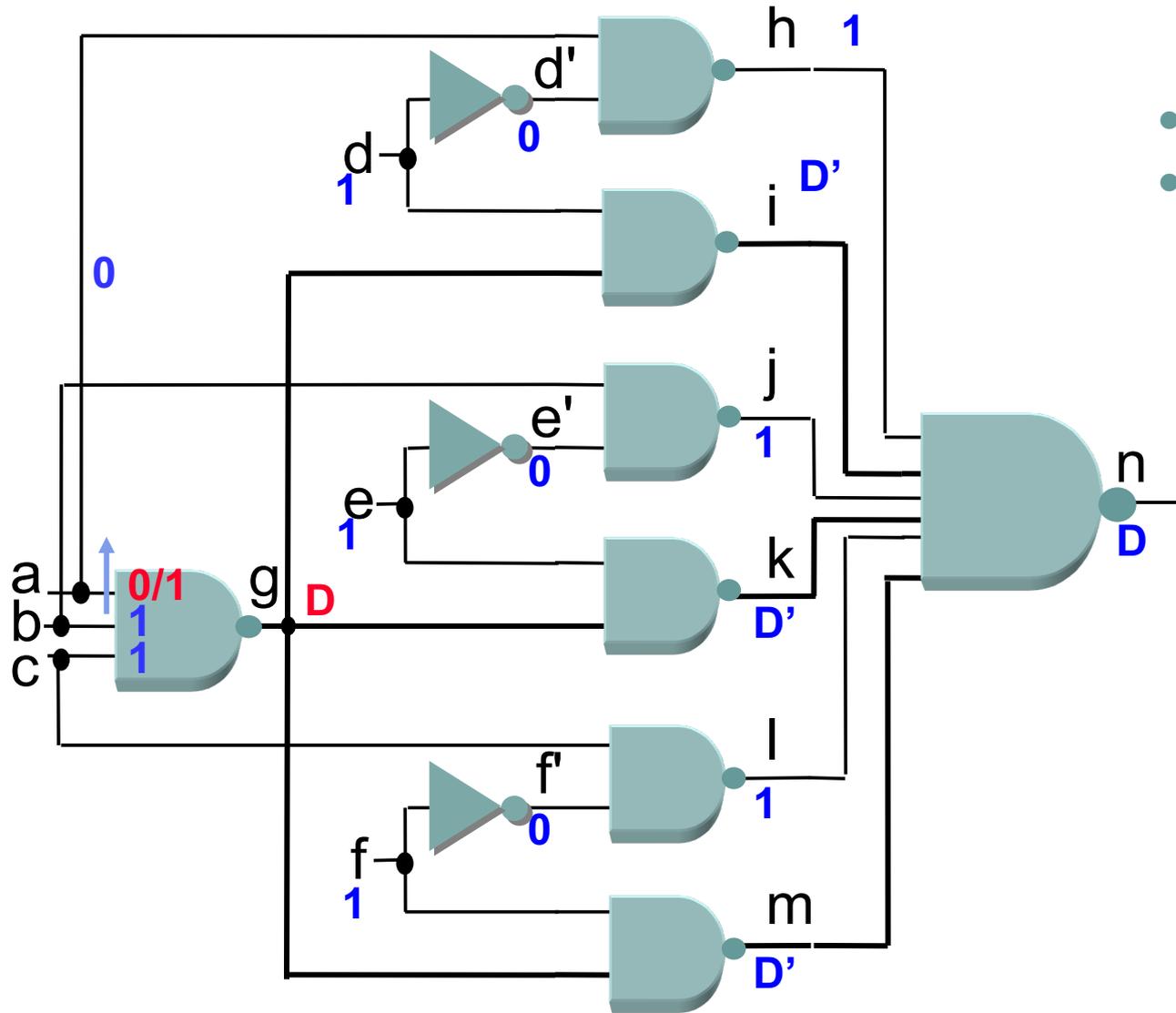
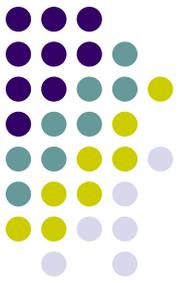
- Success:
  - Fault effect seen at an output.
- Failure:
  - D-frontier is empty and fault is not at any POs.

# Example: PODEM



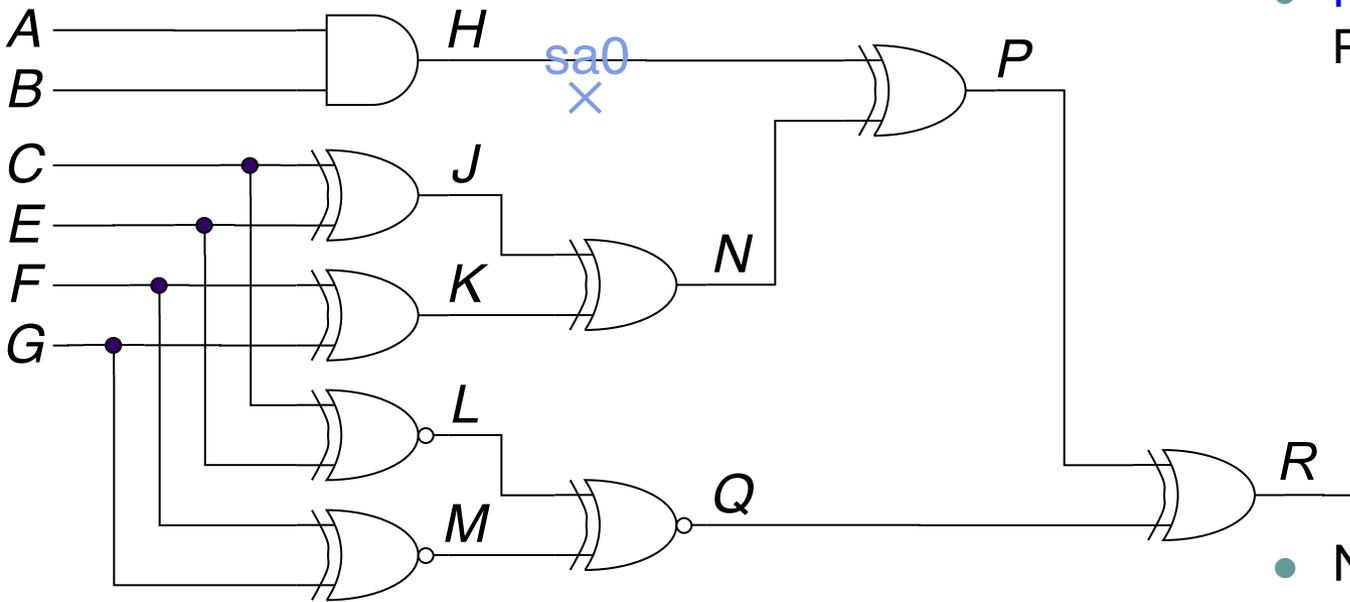
- **Fault activation** → set a to 0 → h=1, **D-Frontier**={g}
- **Fault propagation** → select g → b=1, c=1 → **D-frontier**={i, k, m}
- **Fault propagation** → select i → d=1 → **D-frontier**={k, m, n}
- **Fault propagation** → select n → objective =(k, 1)
  - **Backtrace** → e=0 → e'=1 → j=0
  - Empty D-frontier and fault not at PO
  - **Backtrack** to e=1 → e'=0 → j=1, k=D'

# Example: PODEM (Cont.)



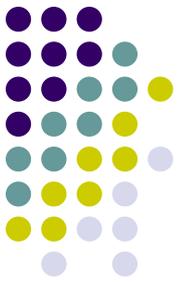
- **D-frontier**={ $m, n$ }
- **Fault propagation**  $\rightarrow$  select  $n \rightarrow$  objective =( $m, 1$ )
  - **Backtrace**  $\rightarrow f=0 \rightarrow f'=1 \rightarrow l=0$
  - Empty D-frontier and fault not at PO
  - Backtrack to  $f=1 \rightarrow f'=0 \rightarrow l=1, m=D' \rightarrow n=D$ , success!!

# Another PODEM Example: ECAT



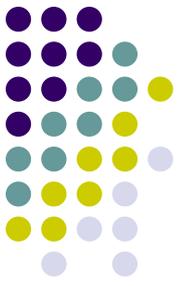
- **Fault activation** → set H to 1 → A=1, B=1, **D-Frontier**={P}
- **Fault propagation** → select P → objective =(N, 1)
  - **Backtrace** → C=1
  - **Backtrace** → E=1 → J=0, L=1
  - **Backtrace** → F=1
  - **Backtrace** → G=1 → K=0, M=1 → N=0, Q=1 → P=D → R=D'
- Note that there is no backtracking in this example.

# Characteristics of PODEM



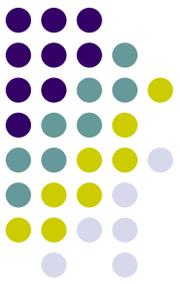
- A complete algorithm
  - like D-algorithm
  - Will find the test pattern if it exists
- Use of backtrace() and forward simulation
  - No J-frontier, since there are no values that require justification
  - No consistency check, as conflicts can never occur
  - No backward implication
  - Backtracking is implicitly done by simulation rather than by an explicit and time-consuming save/restore process
- Experimental results show that PODEM is generally faster than the D-algorithm

# The Selection Strategy in PODEM

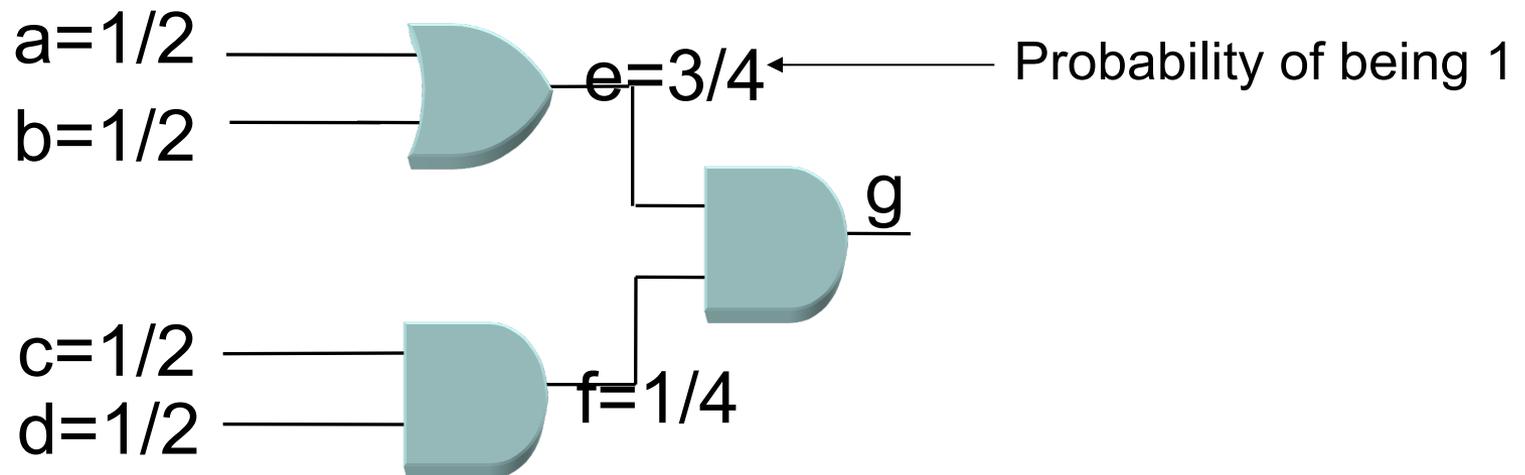


- In Objective() and Backtrace()
  - Selections are done arbitrarily in original PODEM
  - The algorithm will be more efficient if certain guidance used in the selection of objectives and backtrace paths
- Selection Principle
  - Principle 1: among several unsolved problems, attack the hardest one
    - Ex: to justify a '1' at an AND-gate output
  - Principle 2: among several solutions for solving a problem, try the easiest one
    - Ex: to justify a '1' at OR-gate output
- The key is to quantitatively define “hard” and “easy”.

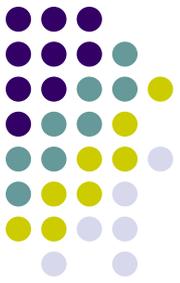
# Controllability As Guidance of Backtrace



- We usually use “controllability” to guide the selection.
  - More details will be provided in Ch 6. Testability Analysis.
- Objective (g, 1)  $\rightarrow$  choose path  $g \rightarrow f$  for backtracing
  - Two unsolved problems  $e=1$  and  $f=1$
  - Choose the harder one:  $f=1$  (lower probability of being 1).
- Objective (g, 0)  $\rightarrow$  choose path  $g \rightarrow f$  for backtracing
  - Two possible solutions:  $e=0$  or  $f=0$
  - Choose the easier one:  $f=0$  (higher probability of being 0).

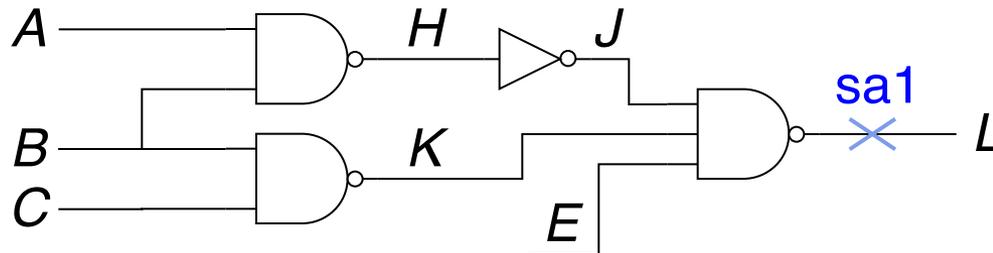
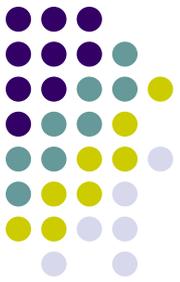


# FAN – Fanout-Oriented Test Generation



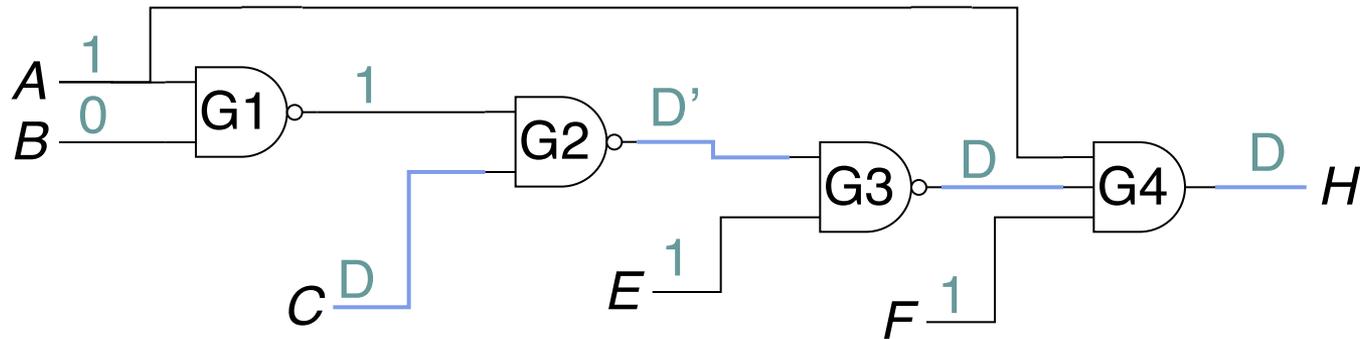
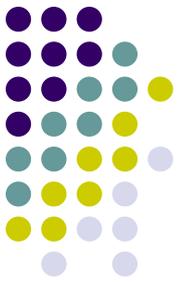
- Techniques to accelerate the test generation process
  - Immediate Implications
  - Unique sensitization
  - Headlines
  - Multiple backtrace

# Immediate Implications



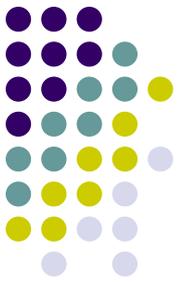
PODEM	FAN
Initial objective $L = 0$	Initial objective $L = 0$
Backtrace to PI: $B = 0$	Set $J = K = E = 1$
Implication: $H = 1, K = 1, J = 0, L = 1$	$J = 1 \rightarrow H = 0 \rightarrow A = B = 1$ $K = 1 \rightarrow C = 0$
Fail $\rightarrow$ backtrack.	

# Unique Sensitization

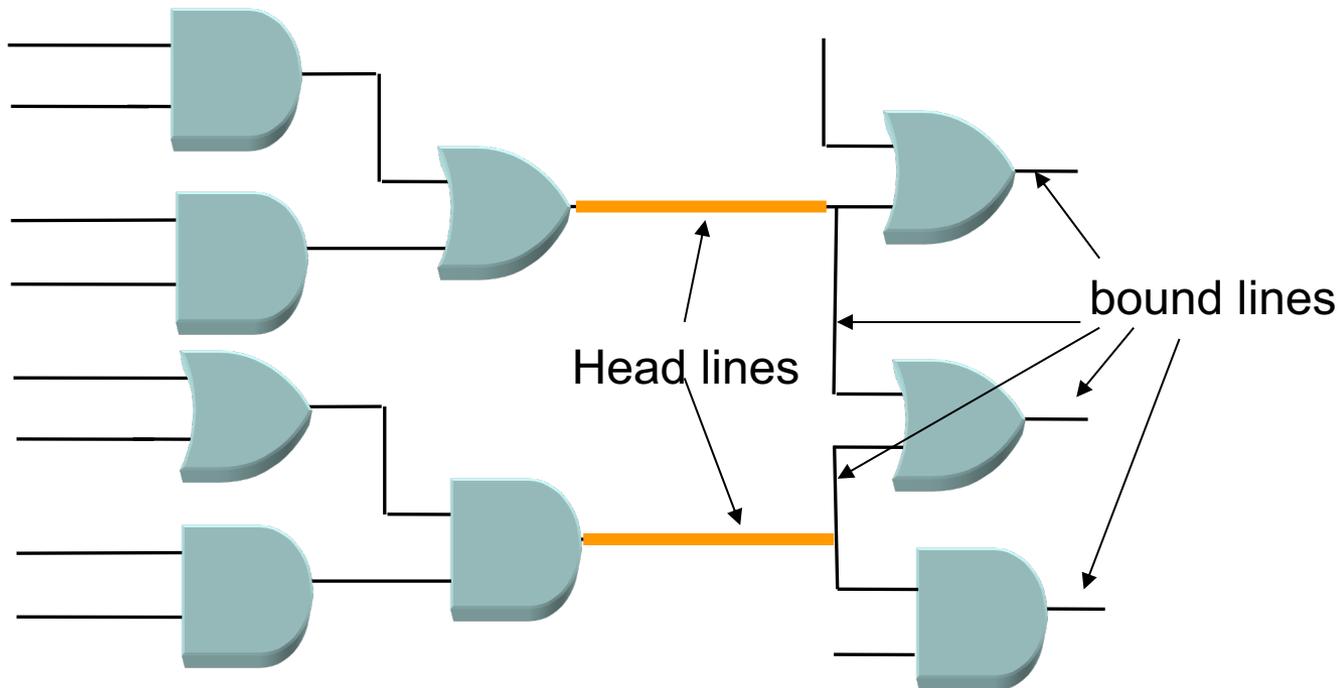


- Only one path to propagate  $D$  on signal  $C$  to output  $H$ .
- Set all the off-path inputs to non-controlling values.
  - $G1 = 1, E = 1, F = 1, A = 1 \rightarrow B = 0$
- PODEM
  - Initial objective: set  $G1$  to 1  
Backtrace to PI:  $A = 0$   
 $\rightarrow$  assigning  $A = 0$  will later block the propagation of  $D$

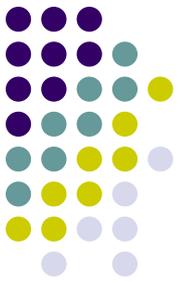
# Head lines



- Output of fanout-free regions with PIs as inputs.
  - Backtrace in FAN **stop at headlines.**
  - Reduce search space.

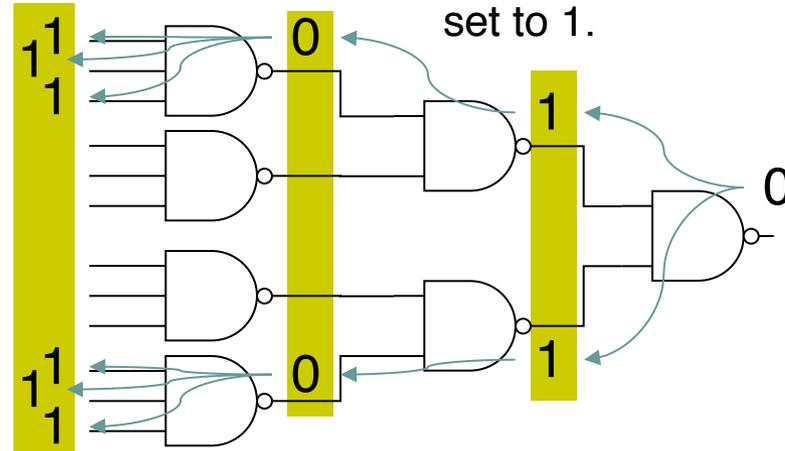


A line that is reachable from at least one stem is said to be bound



# Breadth-First Multiple Backtrace

All must be set to 1.

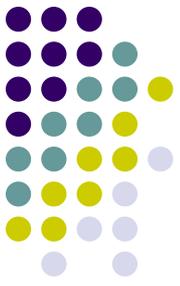


Both must be set to 1.

Set to 0 the input w/  
easiest-to-control  
to 0 input.

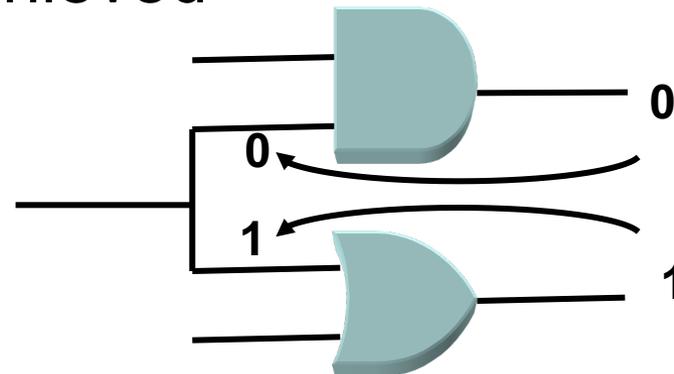
- PODEM's depth-first search is sometimes inefficient.
- Breadth-first multiple backtrace identifies possible signal conflicts earlier.
- Attempts to satisfy a set of objectives simultaneously

# Why Multiple Backtrace ?

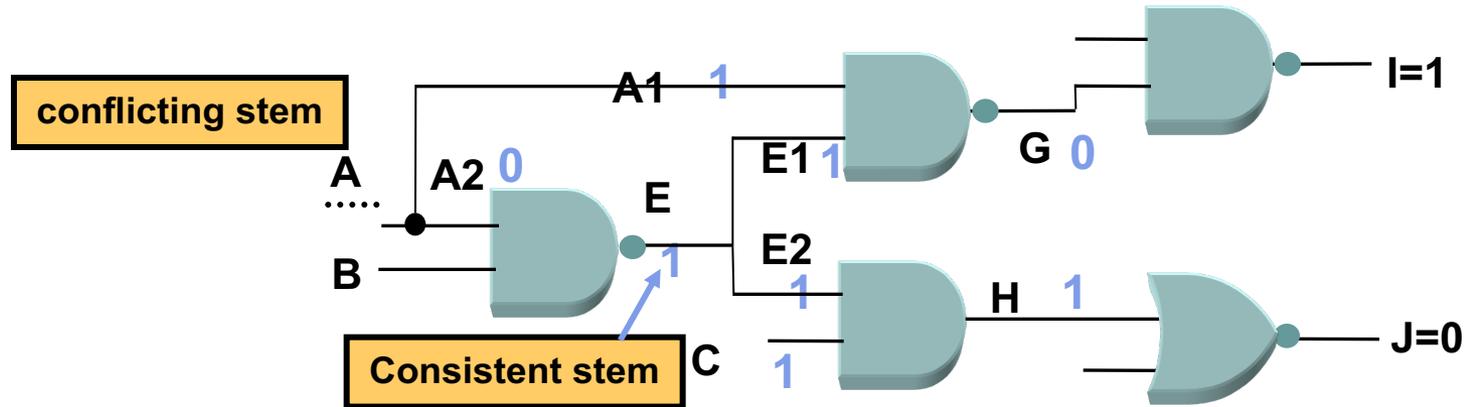
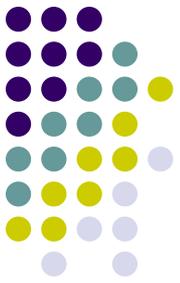


- Drawback of Single Backtrace
  - A PI assignment satisfying one objective → may preclude achieving another one, and this leads to backtracking
- Multiple Backtrace
  - Starts from a set of objectives (Current\_objectives)
  - Maps these multiple objectives into head-line assignments that is likely to
    - Contribute to the set of objectives
    - Or show that objectives cannot be simultaneously achieved

**Multiple objectives may have conflicting requirements at a stem**

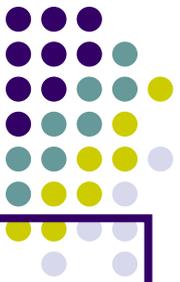


# Example: Multiple Backtrace



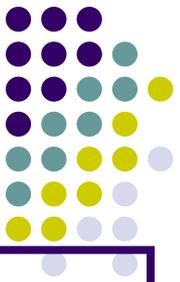
Current_objectives	Processed entry	Stem_objectives	Head_objectives
(I,1), (J,0)	(I,1)		
(J,0), (G,0)	(J,0)		
(G,0), (H,1)	(G,0)		
(H,1), (A1,1), (E1,1)	(H,1)		
(A1,1), (E1,1), (E2,1), (C,1)	(A1,1)	A	
(E1,1), (E2,1), (C,1)	(E1,1)	A,E	
(E2,1), (C,1)	(E2,1)	A,E	
(C,1)	(C,1)	A,E	C
Empty → restart from (E,1)		A	C
(E,1)	(E,1)	A	C
(A2,0)	(A2,0)	A	C
empty		A	C

# Multiple Backtrace Algorithm



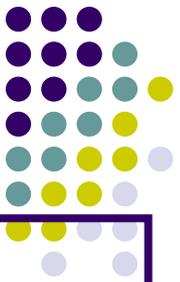
```
Mbacktrace (Current_objectives) {  
  while (Current_objectives  $\neq \phi$ ) {  
    remove one entry (k,  $v_k$ ) from Current_objectives;  
    switch (type of entry) {  
      1. HEAD_LINE:      add (k,  $v_k$ ) to Head_objectives;  
      2. FANOUT_BRANCH:  
          j = stem(k);  
          increment no. of requests at j for  $v_k$ ; /* count 0s and 1s */  
          add j to Stem_objectives;  
      3. OTHERS:  
          inv = inversion of k; c = controlling value of k;  
          select an input (j) of k with value x;  
          if ( $(v_k \oplus \text{inv}) == c$ ) add(j, c) to Current_objectives;  
          else {      for every input (j) of k with value x  
                      add(j, c') to Current_objectives; }  
    }  
  }  
} (to next page)
```

# Multiple Backtrace (con't)



```
if(Stem_objectives≠ϕ ) {  
    remove the highest-level stem (k) from Stem_Objectives;  
    vk = most requested value of k;  
    if (k has contradictory requirements and  
        k is not reachable from target faults)  
        return (k, vk)  
    /* recursive call here */  
    add (k, vk) to Current_objectives;  
    return (Mbacktrace(Current_objectives));  
}  
else {  
    remove one objective (k, vk) from Head_objectives;  
    return (k, vk);  
}  
}
```

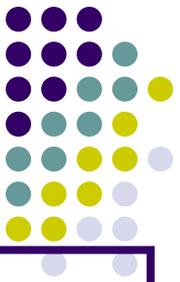
# FAN Algorithm



```
FAN() // the target fault is  $f$  s.a.v
begin
  if ( $f = x$ ) then assign ( $f, v'$ )
  if Imply_and_check() = FAILURE then return FAILURE
  if (error at PO and all bound lines are justified) then
    begin
      justify all unjustified head lines
      return SUCCESS
    end
  if (error not at PO and  $D\_frontier = \emptyset$ ) then return FAILURE
  /* initialize objectives */
  add every unjustified bound line to Current_objectives
  select one gate ( $G$ ) from the D-frontier
   $c$  = controlling value of  $G$ 
```

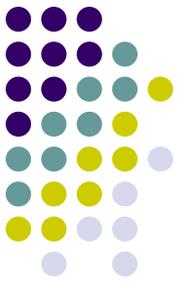
(to next page)

# FAN algorithm (con't)



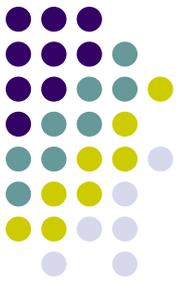
```
for every input ( $j$ ) of  $G$  with value  $x$ 
    add ( $j, c'$ ) to Current_objectives
/* multiple backtrace */
( $i, v_i$ ) = Mbacktrace(Current_objectives)
Assign( $i, v_i$ )
if FAN() = SUCCESS then return SUCCESS
Assign( $i, v_i'$ ) /* reverse decision */
if FAN() = SUCCESS then return SUCCESS
Assign( $i, x$ )
return FAILURE
end
```

# Advanced Concepts of ATPG

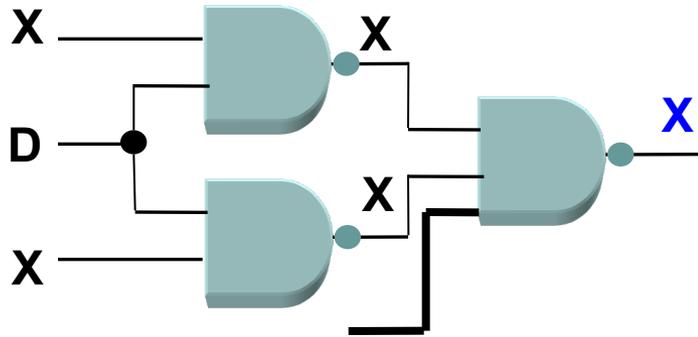


- Dominator ATPG
- Learning algorithms
- State hashing
- Satisfiability-based methods and Implication graphs

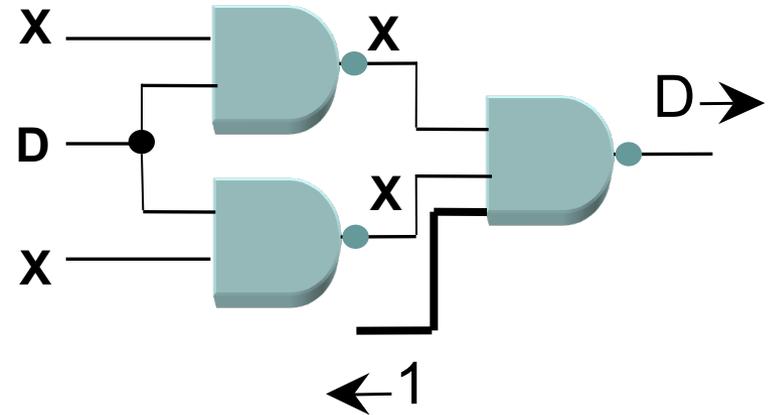
# Dominators for Fault Propagation



Before



After unique D-drive analysis



- Unique D-Drive Implication

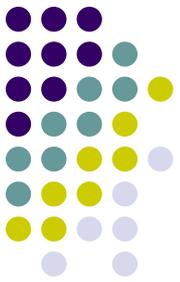
- If every gates in D-frontier pass through a dominator (**unique D-drive**), we can set non-controlling values at other side-inputs, and propagate D/D' further.

# Learning Methods for ATPG



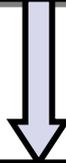
- Learning process systematically sets all signals to 0 and 1 and discover what other signal values are implied.
  - Used extensively in [implication](#).
  - Ex.  $a=1 \rightarrow b=0$ , even a and b are not implied immediately by gate relationship.
- Static learning
  - Start the learning process before ATPG
- Dynamic learning
  - Start the learning process in between ATPG steps when some signal values are known
  - Costly process

# Modus Tollens for Learning



Given

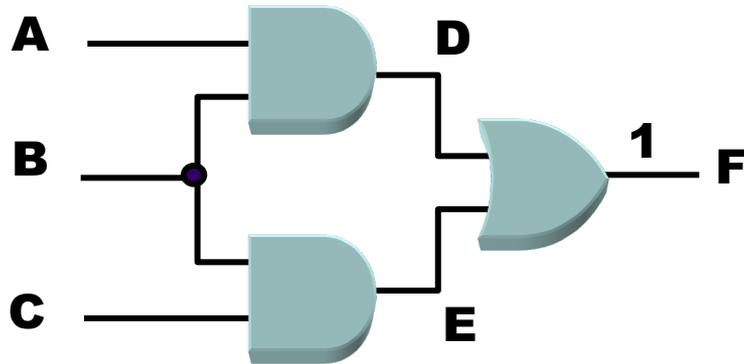
$P \rightarrow Q$  is true



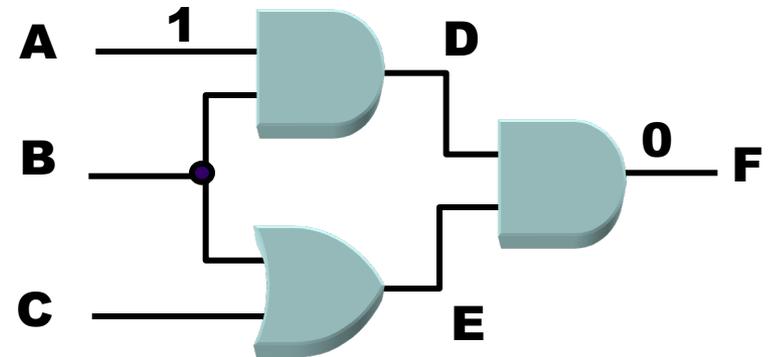
We can derive

$\sim Q \rightarrow \sim P$  is true

- Example:



P and Q are Boolean statements  
 $\sim Q$  means that Q is not true

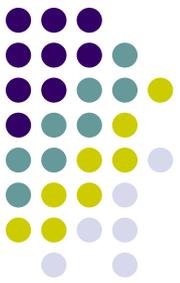


$(B=0) \rightarrow (F=0)$   
 $\Rightarrow \sim(F=0) \rightarrow \sim(B=0) \Rightarrow (F=1) \rightarrow (B=1)$

**(Static Learning)**

When  $A=1$   
 $(B=1) \rightarrow (F=1) \Rightarrow (F=0) \rightarrow (B=0)$

**(Dynamic Learning)**



# Constructive Dilemma for Learning

**Given**

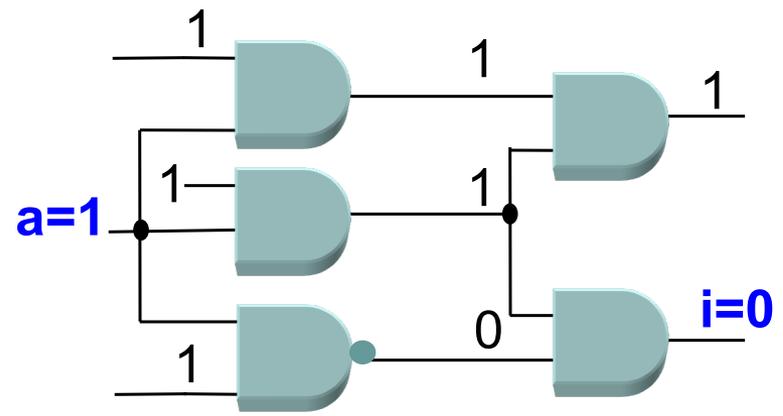
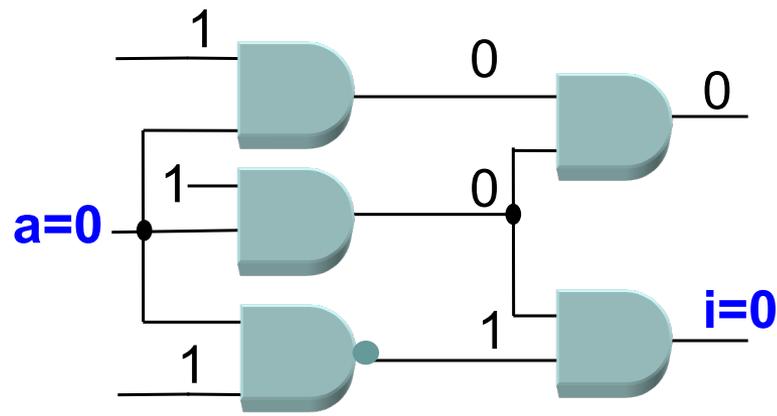
(1)  $P \Rightarrow Q$  and  $R \Rightarrow S$ , and  
(2) Either P or R is true

**We can derive**

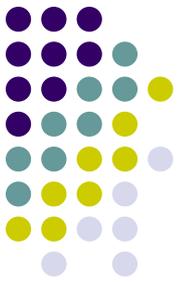
Either Q or S is true.

## • Example:

- Both  $[(a=0) \Rightarrow (i=0)]$  and  $[(a=1) \Rightarrow (i=0)]$  are true.
- Either  $(a=0)$  or  $(a=1)$  holds  $\Rightarrow (i=0)$

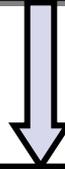


# Modus Ponens for Learning



Given

F and  $F \Rightarrow G$  are true

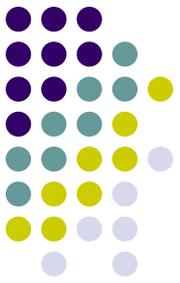


We can derive

G is true.

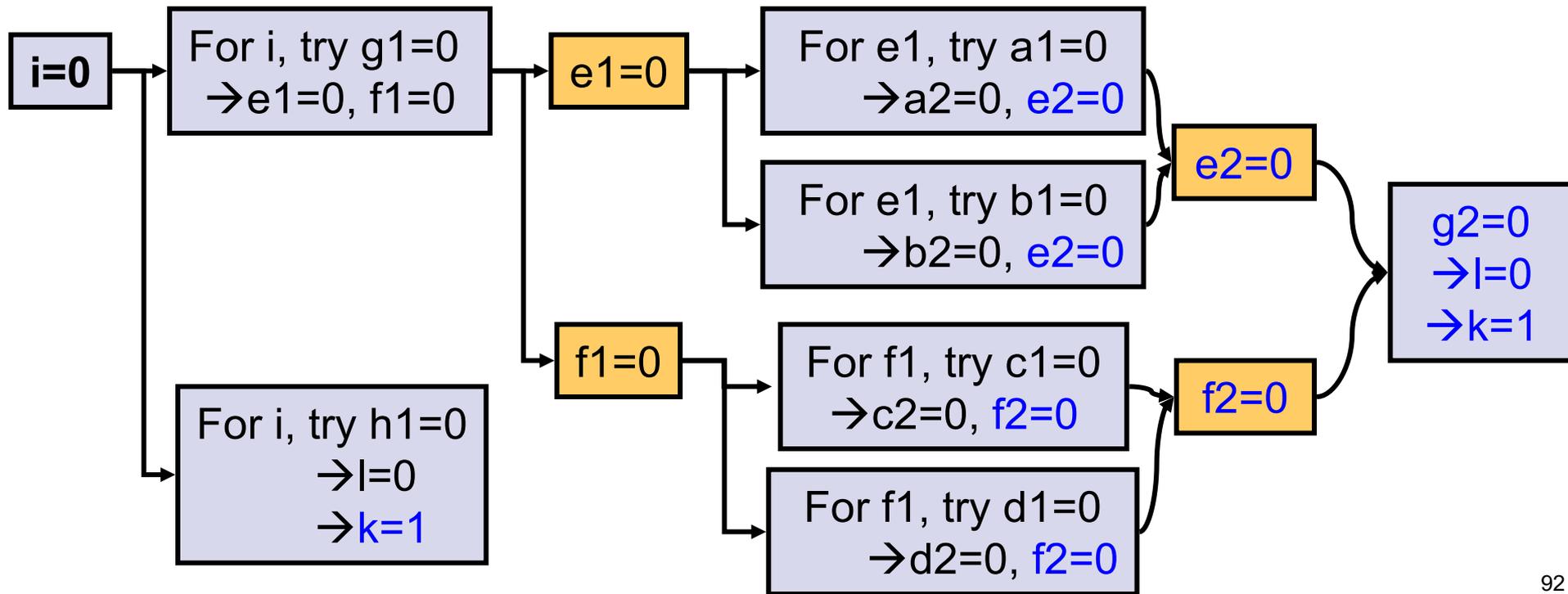
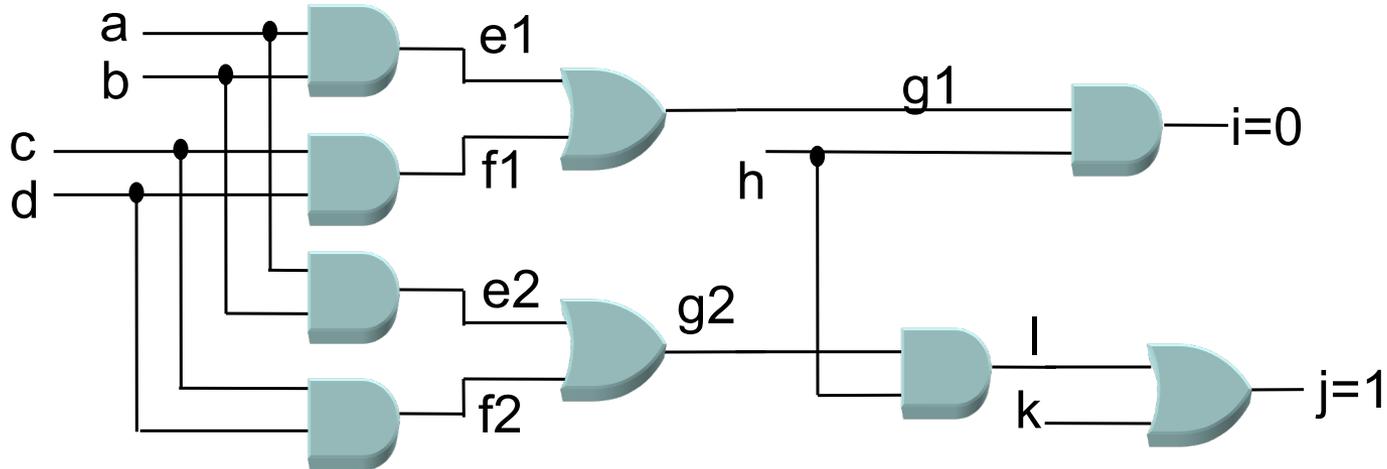
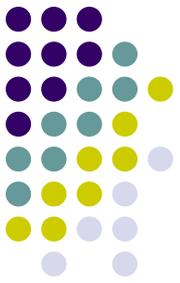
- Example:
  - $[a=0]$  and  $[(a=0) \Rightarrow (f=0)]$
  - $\Rightarrow (f=0)$

# Recursive Learning

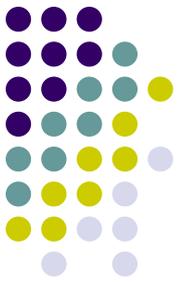


- From known signal values, try all possible decisions (recursively).
- If a certain signals have the same values among all decisions, they are implied values.

# Example of Recursive Learning

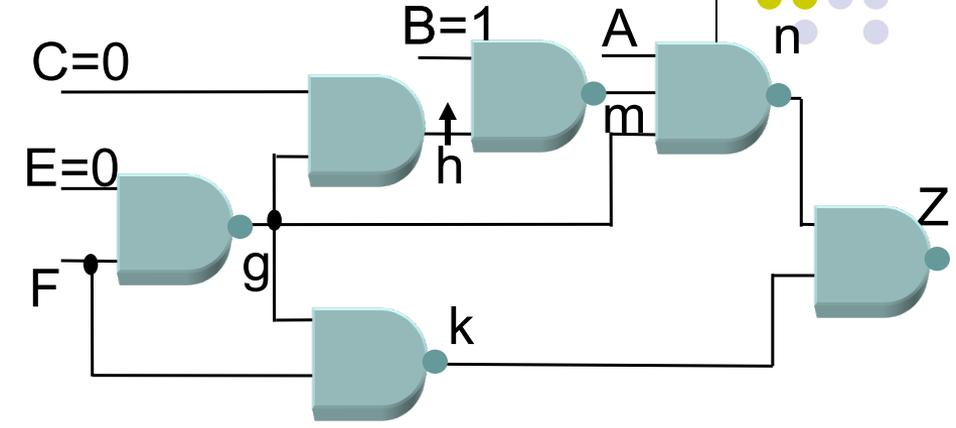
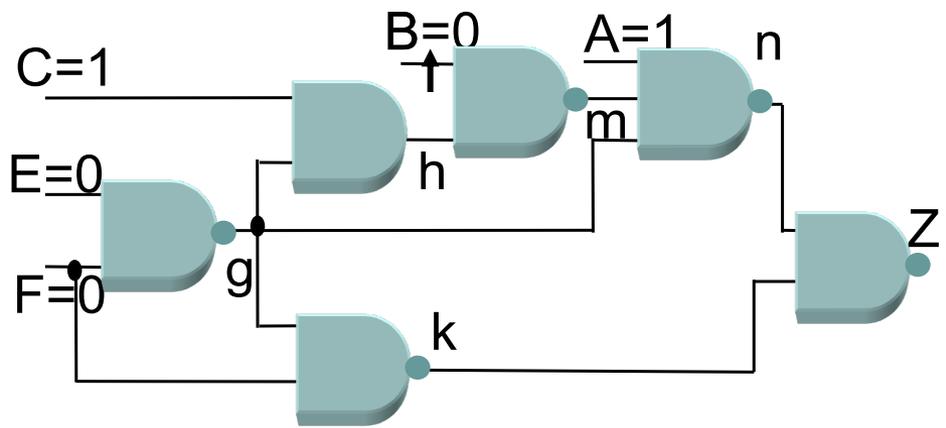


# State Hashing (EST)



- Records and compare E-frontiers
  - E-frontiers are the cut between known signals and unknown parts.
- Given identical E-frontiers
  - Previous implications can be used
  - Or inconsistency can be deduced from the previous run.

# State Hashing Example



Step	Input Choice	E-frontier
1	B=0	{B=D'}
2	C=1	{C=1, B=D'}
3	E=0	{g=1, m=D}
4	A=1	{g=1, n=D'}
5	F=0	{Z=D}

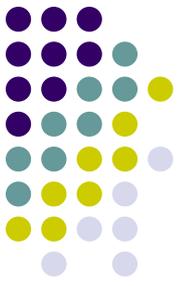
Step	Input Choice	E-frontier
1	C=0	{h=D'}
2	B=1	{m=D}
3	E=0	{g=1, m=D}

**Match found at step 3**

**Process for detecting B s-a-1**  
**Test pattern = (CEFBA)=(10001)**

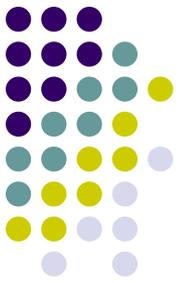
**Process for detecting h s-a-1**  
**Test pattern = (CEFBA)=(00011)**

# Implication Graph ATPG

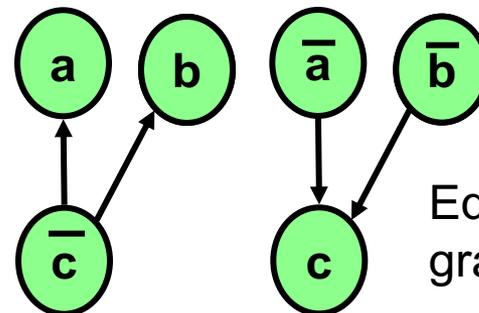
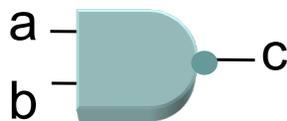


- Translate circuits into implication graphs
- Use transitive closure graph algorithm to perform ATPG
  - Some decisions and implications are better assigned with graph algorithms.
  - But experiences show that circuit structure is a very important information for ATPG.
- Satisfiability-based algorithms are currently the fastest implementation for justification and commonly applied to verification problems.
  - We have a separate set of slides about SAT.

# Translating Gate to its Equivalent Implication Graph

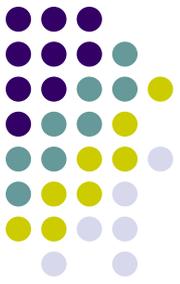


- Use NAND as an example
- A two-input NAND can be described completely by the following 3 equations.
  - $a' \rightarrow c$
  - $b' \rightarrow c$
  - $c' \rightarrow ab$
  - Note that they cover all cubes.
- Implication graph
  - Each node represents a signal variable.
  - If there is a directed edge from one node ( $v1$ ) to another ( $v2$ ),  $v1$  imply  $v2$ .

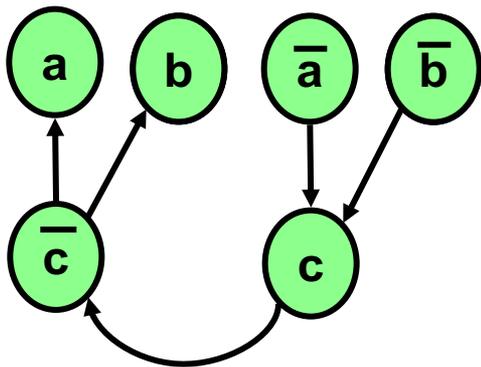


Equivalent implication graph for NAND.

# Perform Implication on the Graph



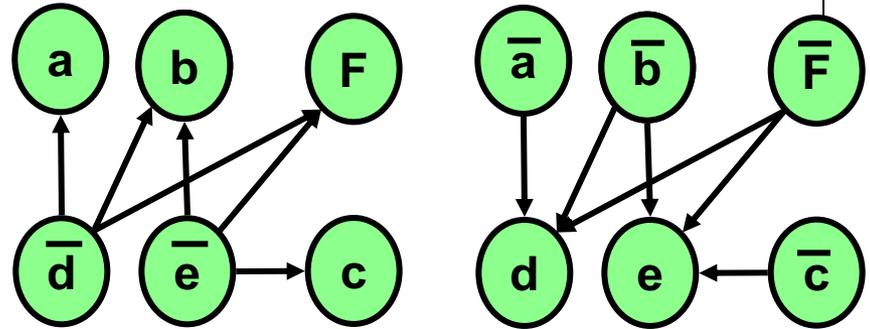
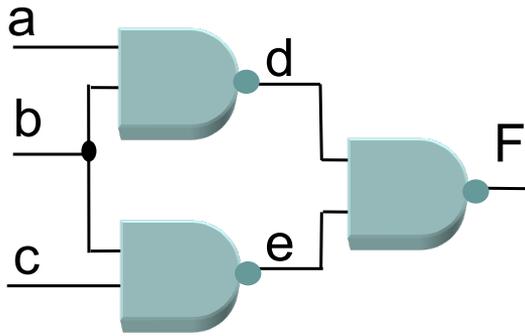
- Assume we have  $c=0$ , we add an edge for  $c \rightarrow c'$ 
  - $c \rightarrow c' == c' + c' == c'$
- Extra implication are found by tracing linked nodes.
- In this example,  $a' \rightarrow a$  and  $b' \rightarrow b$ , hence  $a=1, b=1$ .



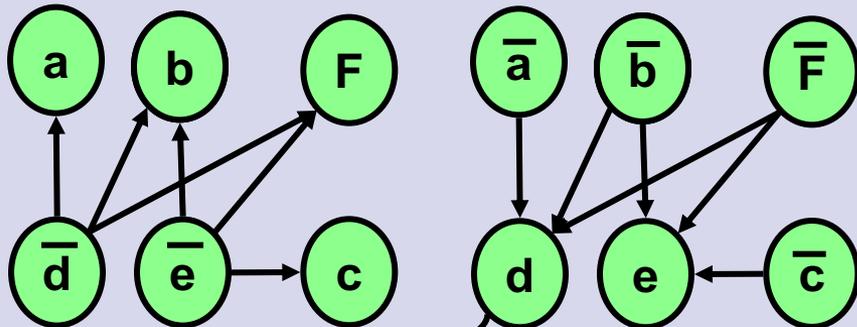
a	b	$\rightarrow$
0	0	1
0	1	1
1	0	0
1	1	1

Truth table for implication  
 $a \rightarrow b == a' + b$

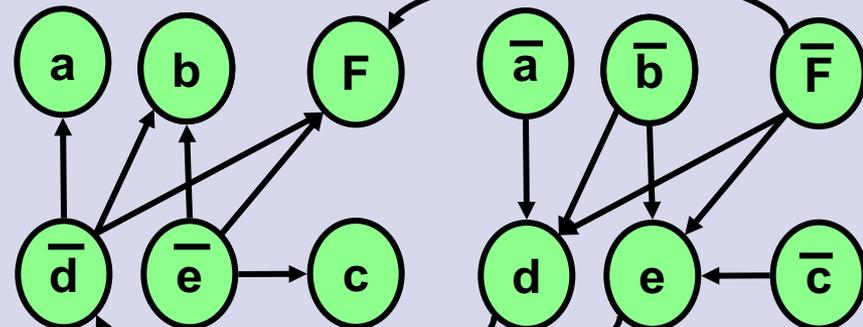
# Implication Graph Examples



Equivalent Circuit Implication Graph



**Implication Graph  $d=0$**   
 New paths:  $a'-d-d'-a$ ,  $F'-d-d'-F$ ,  $b'-d-d'-b$   
 New conditions:  $a' \rightarrow a$ ,  $F' \rightarrow F$ ,  $b' \rightarrow b$   
 Implied logic values:  $a=1$ ,  $F=1$ ,  $b=1$



**Implication Graph  $F=1$  (i.e.,  $de=0$ )**  
 New paths:  $b'-d-e-b$ ,  $b'-e-d-b$   
 New conditions:  $b' \rightarrow b$   
 Implied logic values:  $b=1$