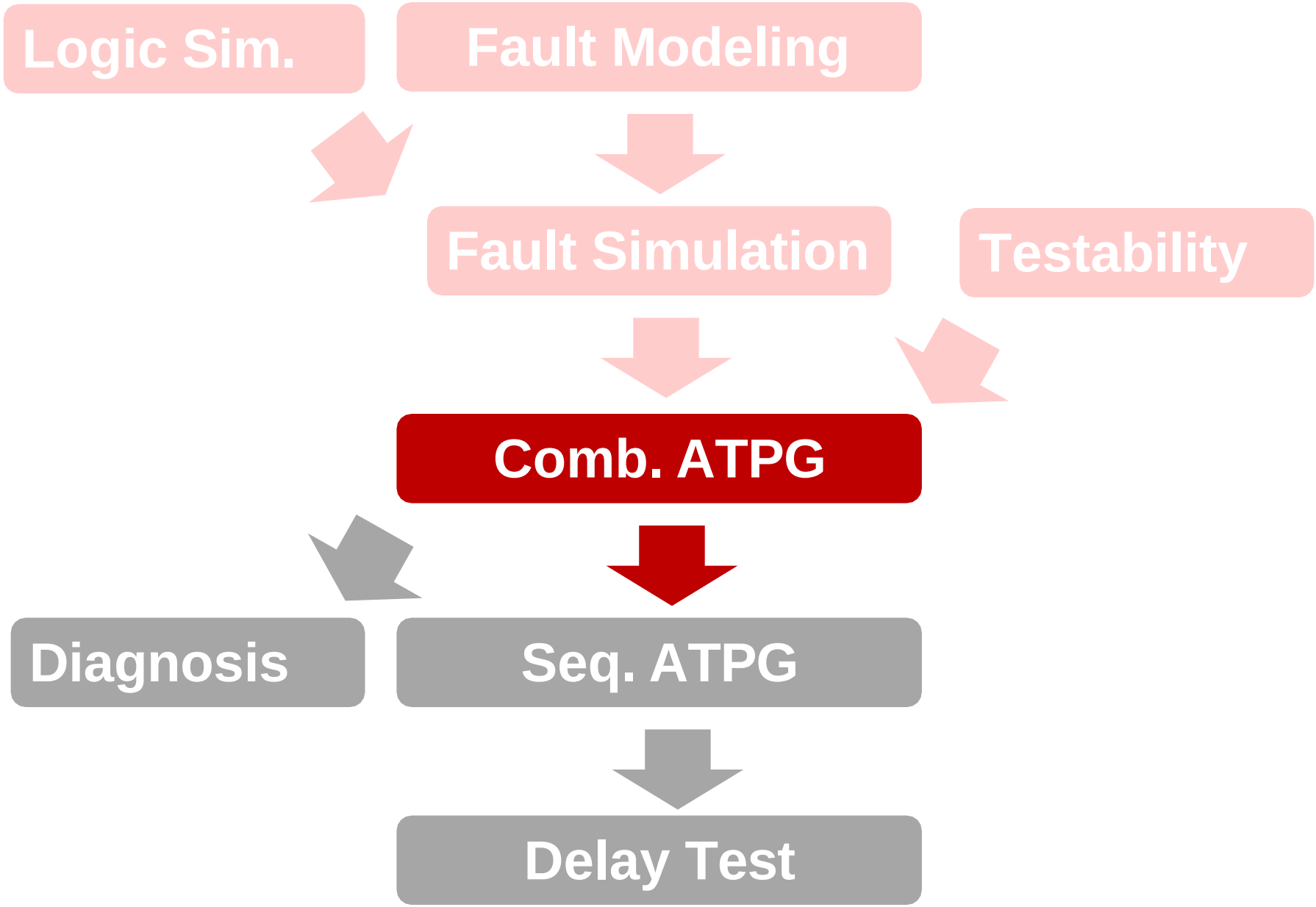
Combinational ATPG

Dr. Bharat Garg
Assistant Professor,
DECE, TIET, Patiala

Course Roadmap (EDA Topics)



Why Am I Learning This?

- **Automatic Test Pattern Generation (ATPG)**
  - **1. Generate high quality test patterns**
  - **2. Reduce Human efforts**

"Testing a product is a learning process."

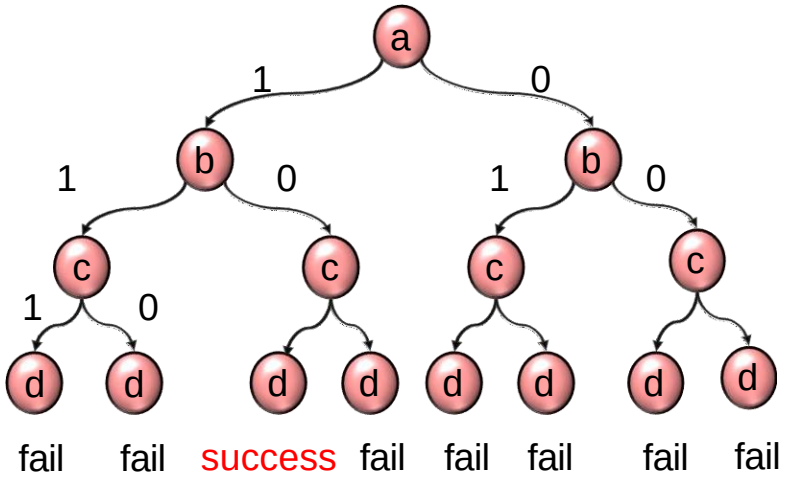
(Brian Marick)

Test Generation

Fault Models	Combinational Circuits (seq. ckt. w/ scan)	Sequential Circuits
No fault model	PET	Checking experiment
Single Stuck-at Fault Model	D PODEM FAN	Extended D 9-valued
Delay Fault Model	Path delay Transition delay	Launch on capture Launch on shift

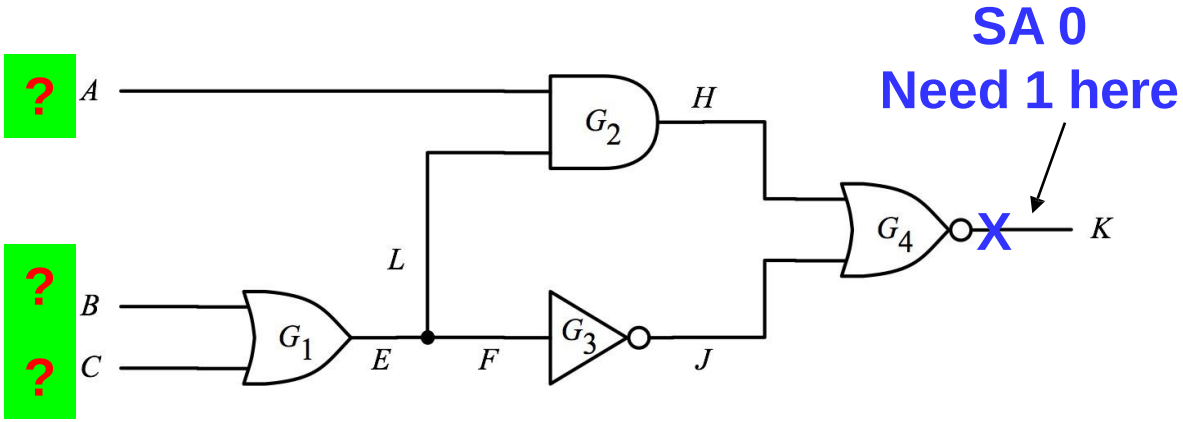
Combinational ATPG

- **Introduction**
- **Deterministic Test Pattern Generation**
- **Acceleration Techniques**
- **Concluding Remarks**



Motivating Problem

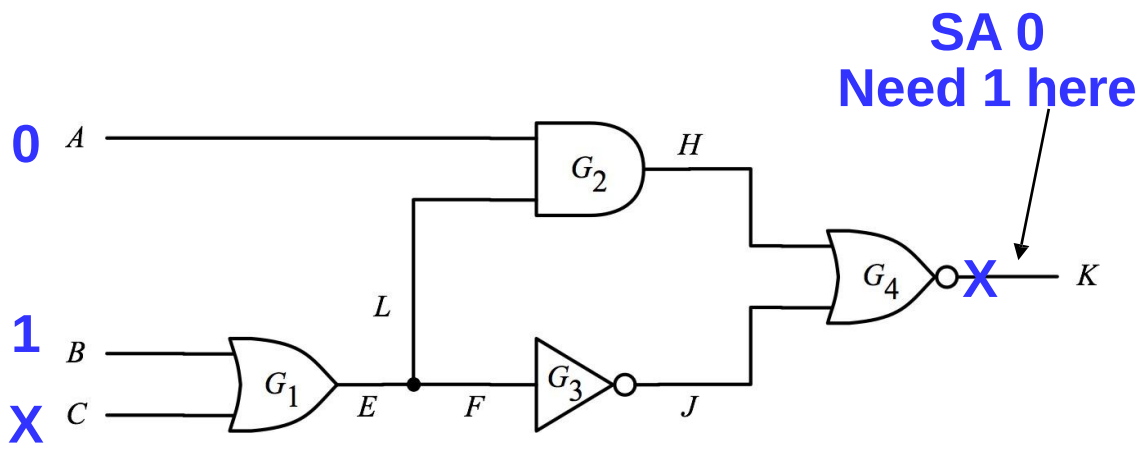
- **Your manger asks you to generate a test pattern for output stuck-at zero fault. What is your answer? (Maybe more than one answer)**



- **Simulation is like *substitution*, easy**
  - ***f(x)=x⁵+3x⁴+2*, x=3.2 ⇒ *f(x)*=?**
- **Finding test pattern is like *finding roots,* very hard**
  - ***f(x)=x⁵+3x⁴+2, f(x)=0*, ⇒ *x*=?**

What is Complexity of TPG?

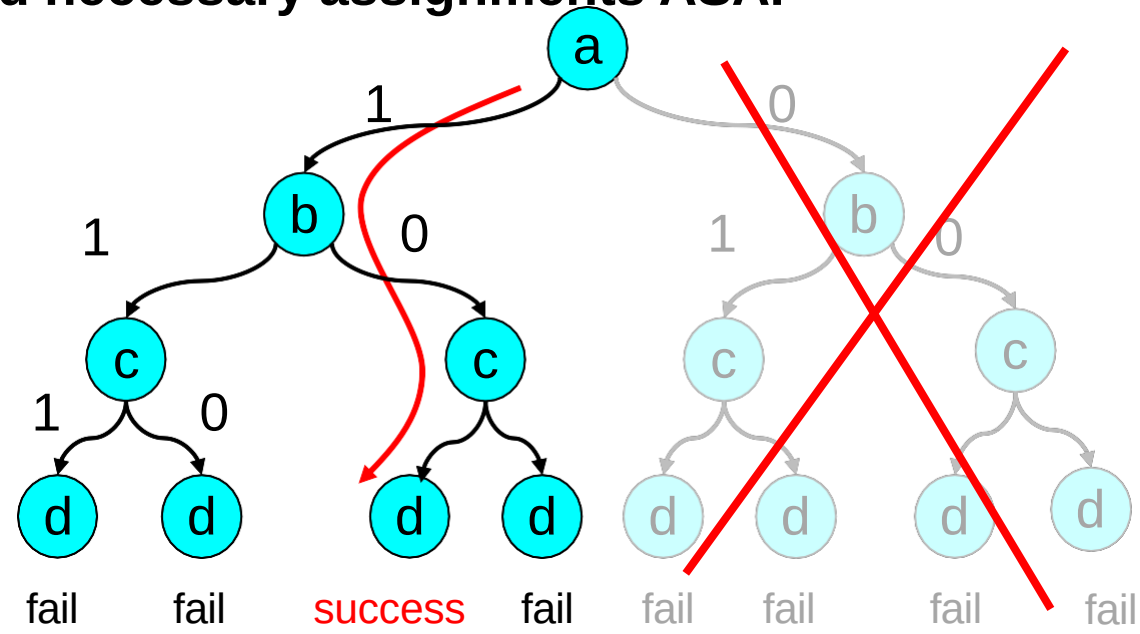
- *Test pattern generation TPG*
  - generate test patterns for a given fault model
- Generating a test pattern for a fault is *NP-complete*
  - Same as *satisfiability* problem
  - Worst case 2^{number_PI} assignments to try
- Need automatic tool, *ATPG* (*Automatic Test Pattern Generator*)



TPG is NP-complete

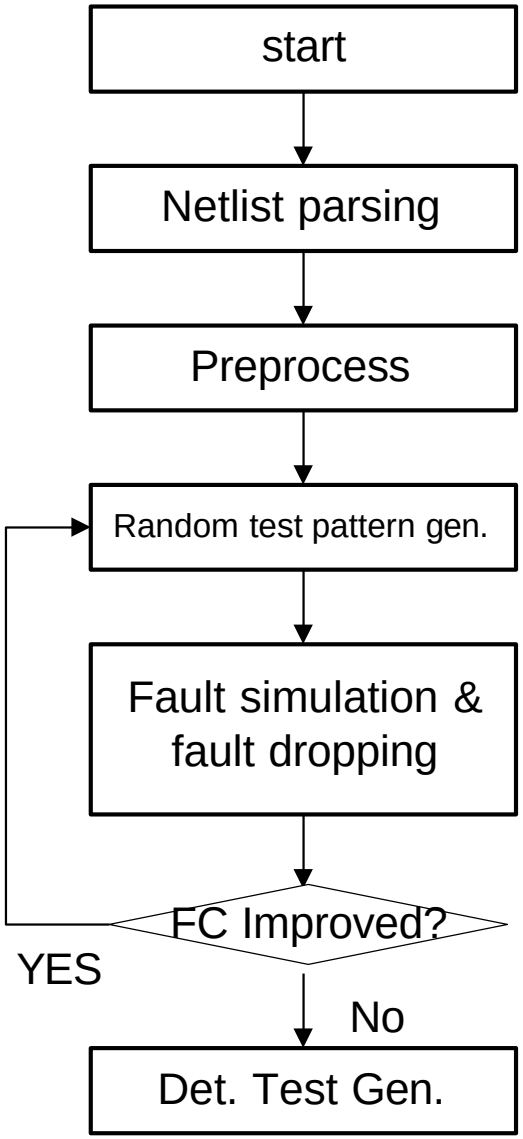
ATPG is Decision Problem

- Huge binary decision tree (exponential size!)
  - There could be *one, many, or even zero* answers
- Need smart heuristic to speed up
  - 1. Prune impossible sub-trees ASAP
  - 2. Find necessary assignments ASAP



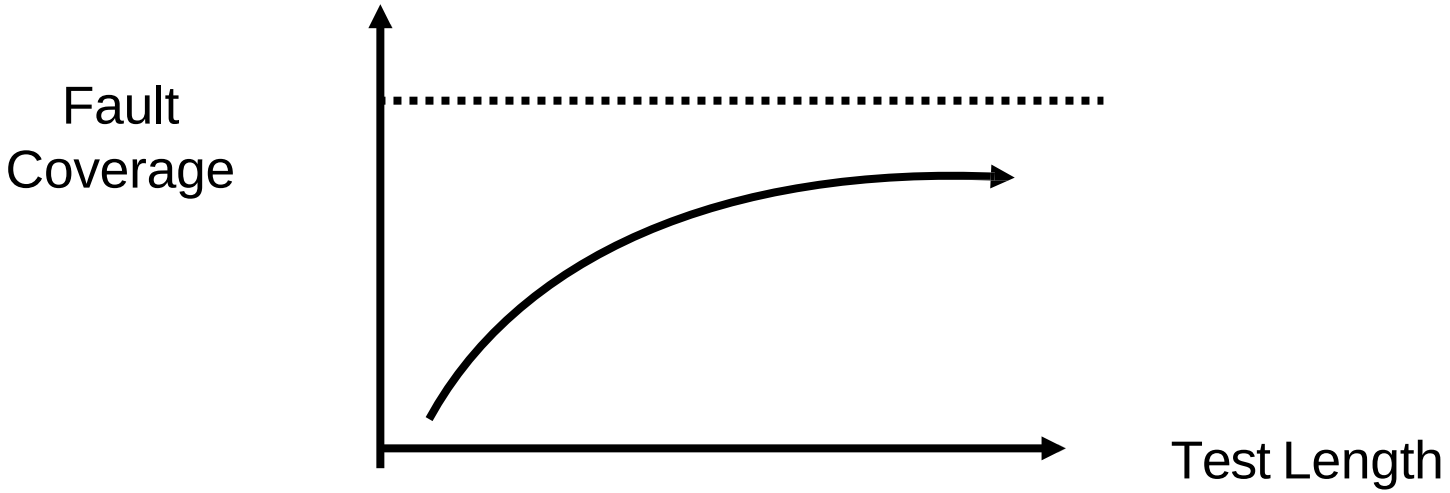
Search a Test Pattern in a Huge Tree

Random Test Pattern Gen.

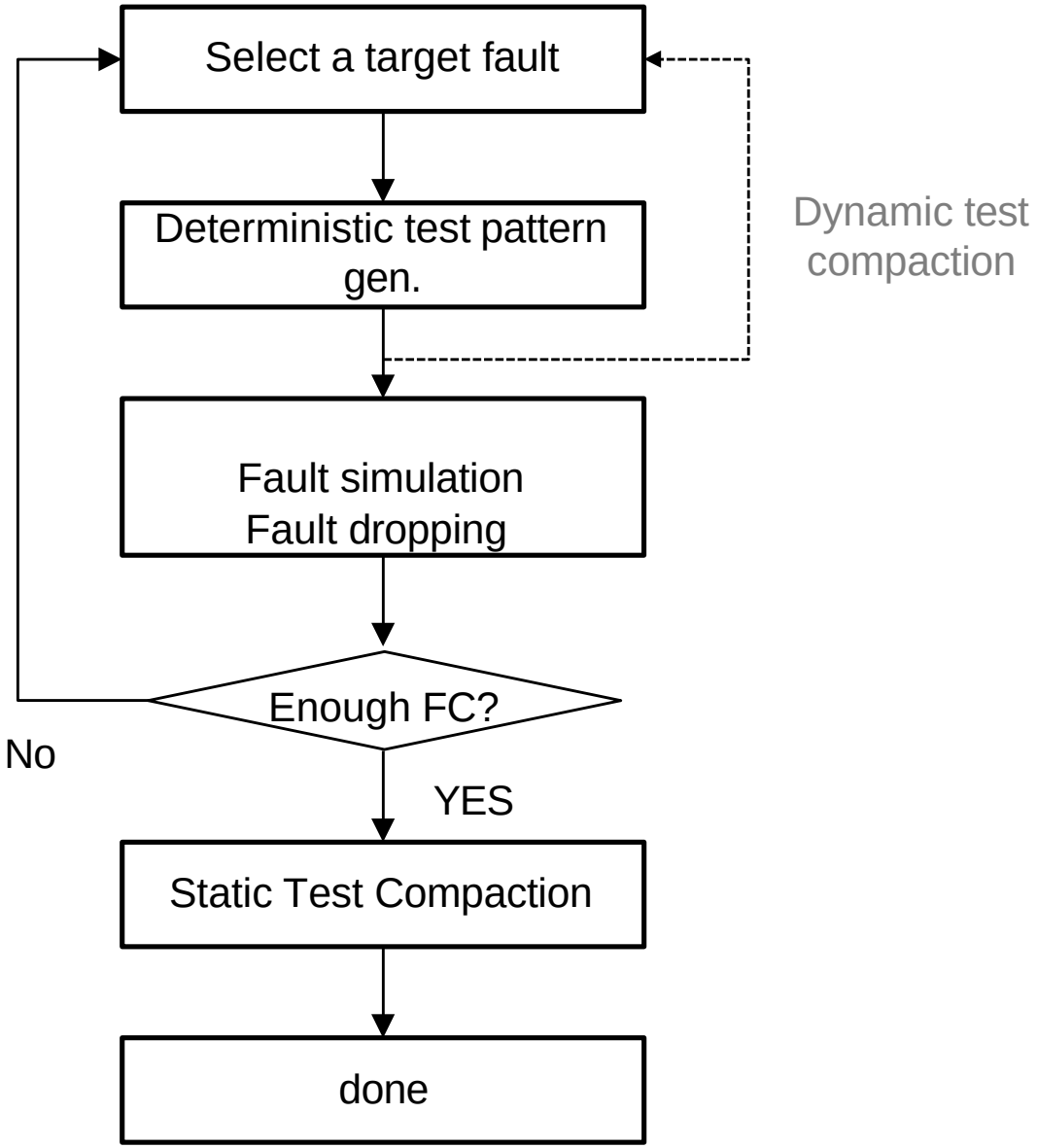


Random Test Pattern Generation

- **Idea: there are many easy-to-detect faults**
  - **First generate random patterns and**
  - **then select patterns that defect undetected faults**
- **Problem**
  - **Fault coverage often saturates after easy faults are detected**
    - ***Random pattern resistant faults*** **not easy to detect**



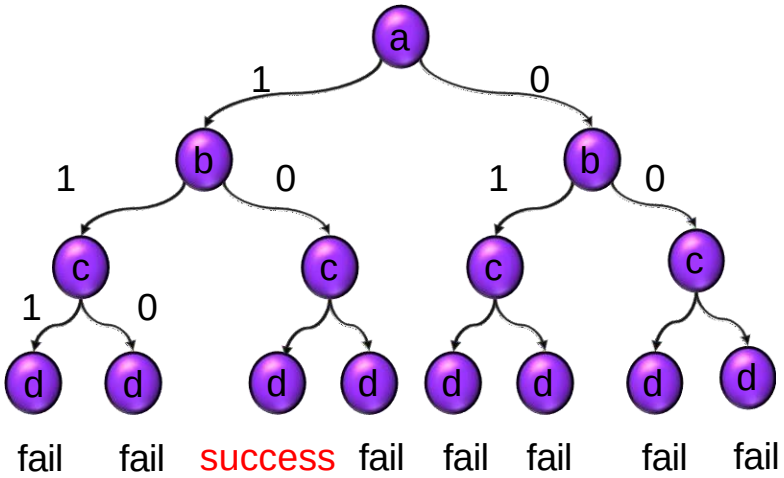
Deterministic Test Pattern Gen.



Combinational ATPG

- Introduction
- **Deterministic Test Pattern Generation**
  - **Boolean difference***
  - **Path sensitization****
  - **D-Algorithm (1965)****
  - **PODEM (1981)****
  - **FAN(1985) ****
  - **SAT-based (1992)***
- Acceleration Techniques
- Concluding Remarks

*Boolean-based methods
**path-based methods



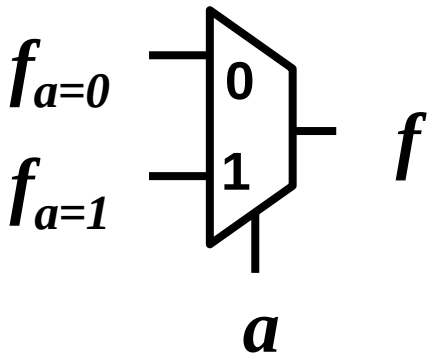
Decomposition

- **Consider a circuit that realizes the function *f(a, b, c, …)***
- **Fixed *a* to one:**
  - ***positive cofactor of f with respect to a***

$$f_{a=1} = f(a=1,\ b, c, \ldots)$$

- **Fixed *a* to zero:**
  - ***negative cofactor of f with respect to a***

$$f_{a=0} = f(a=0,\ b, c, \ldots)$$

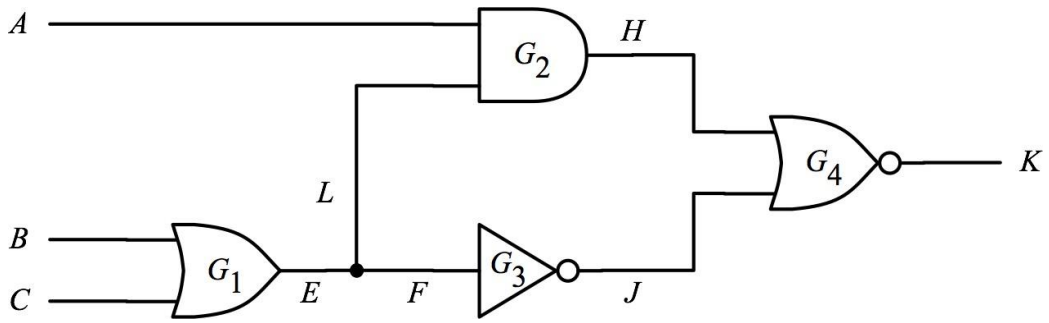


- ***Shannon's Expansion*** **w.r.t. input *a* [Shannon 1948]**

$$f = a\ f_{a=1} + a'\ f_{a=0}$$

Quiz

Q: Shannon's Expansion for output *K* w.r.t. input B
A:



Boolean Difference (1)

- To detect a stuck-at zero fault
  - Good output (f) and faulty output ($f_{a=0}$) are different: $f \oplus f_{a=0} = 1$
  - After Shannon Expansion:
$$[af_{a=1} + a' f_{a=0}] \oplus f_{a=0}$$
$$= a[f_{a=1} \oplus f_{a=0}] + a'[f_{a=0} \oplus f_{a=0}]$$
$$= a[f_{a=1} \oplus f_{a=0}] + 0$$
$$= 1$$
  - Thus: $a[f_{a=0} \oplus f_{a=1}] = 1$
- To detect a stuck-at one fault
  - Good output (f) and faulty output ($f_{a=1}$) are different: $f \oplus f_{a=1} = 1$
  - After Shannon Expansion: $[af_{a=1} + a' f_{a=0}] \oplus f_{a=1} = 1$
  - Thus: $a'[f_{a=0} \oplus f_{a=1}] = 1$
- *Boolean difference* of f w.r.t. a

$$\frac{df}{da} = [f_{a=0} \oplus f_{a=1}]$$

Boolean Difference (2)

- **Boolean Difference = 1 means *$f_{a=0}$ and $f_{a=1}$ are different***
  - **which means: *a* is sensitized to output *f***

$$\frac{df}{da} = [\, f_{a=0} \oplus f_{a=1} \,] = 1$$

- **Example: AND gate, when b=1, *f* = A**

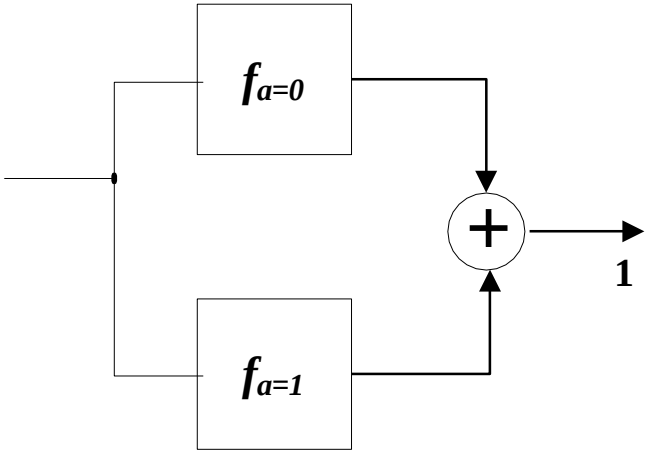
$$f = ab$$

$$\frac{df}{da} = [\, f_{a=0} \oplus f_{a=1} \,] = 0 \oplus b = b$$

- **Example: OR gate, when b=0, *f* = A**

$$f = a + b$$

$$\frac{df}{da} = [\, f_{a=0} \oplus f_{a=1} \,] = b \oplus 1 = b'$$

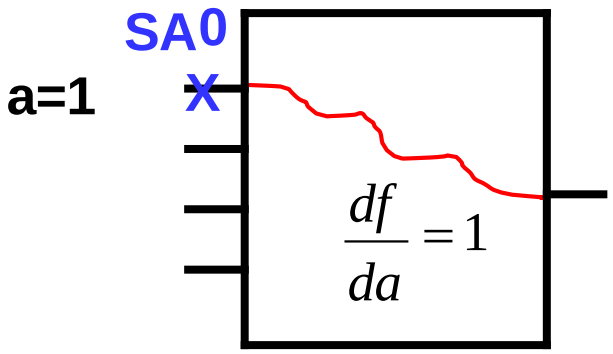


BD=1: sensitization condition

Boolean Difference (3)

- **All patterns to detect *a* stuck-at-0 fault**
  - ***a* = 1 : *fault excitation***
  - **BD = 1 : *sensitization* (aka *fault effect propagation*)**

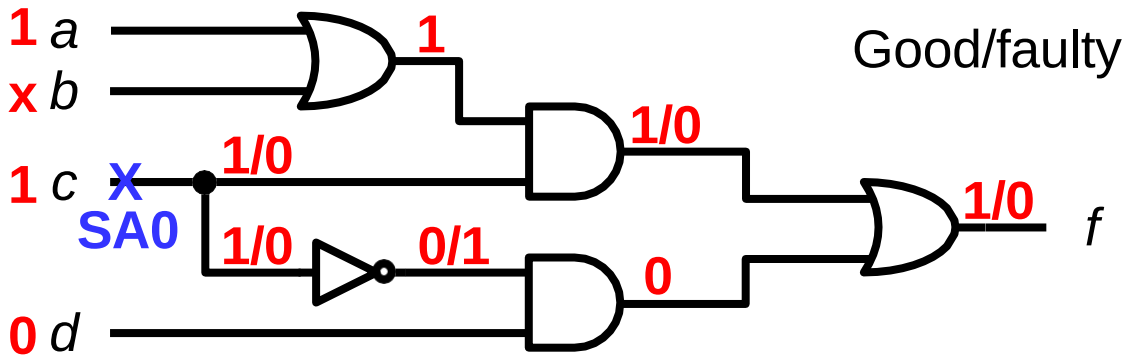
$$a\frac{df}{da}=1$$



- **All patterns to detect *a* stuck-at-1 fault**
  - ***a* = 0 : fault excitation**
  - **BD = 1 : sensitization**

$$a'\frac{df}{da}=1$$

Test Generation Example



- ***f = (a + b)c + c'd***
- **Set of all tests for *c stuck-at-0* is** $\boxed{c\frac{df}{dc}=1}$

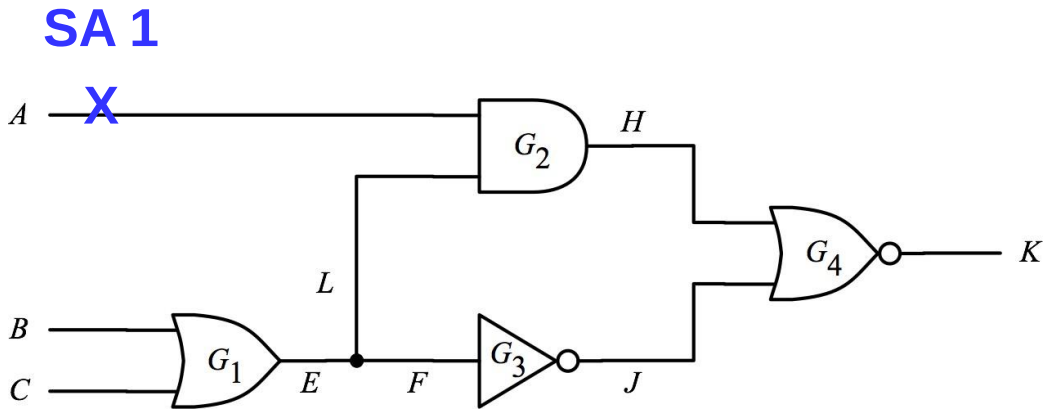
$$\frac{df}{dc} = f(a,b,0,d) \oplus f(a,b,1,d) = d \oplus (a+b) = ad'+bd'+a'b'd$$

$$c\frac{df}{dc} = acd'+bcd'+a'b'cd$$

- **Set of all tests = {1x10, x110, 0011} (x = don't care inputs)**
- **One *fully specified test pattern:* *e.g.* 1110**
- **One *partially specified test pattern (aka. test cube):* 1x10**

Quiz

(Cont'd) We already known K=A'B'C+A'B
Q1: Boolean difference dK/dA=?
Q2: Use BD to find all test patterns for A stuck-at one fault.

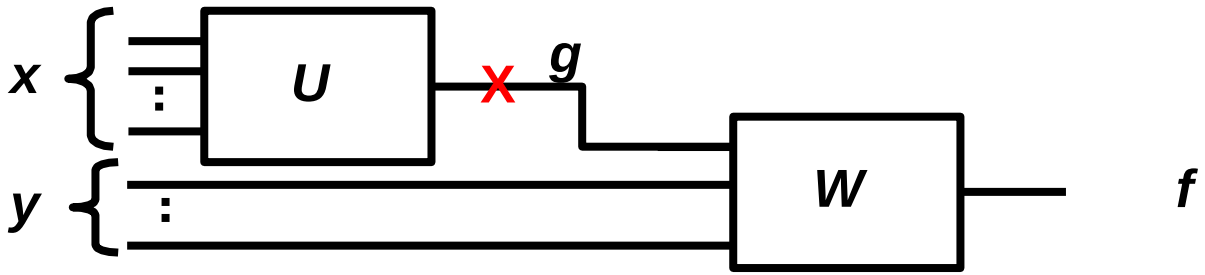


Internal Faults

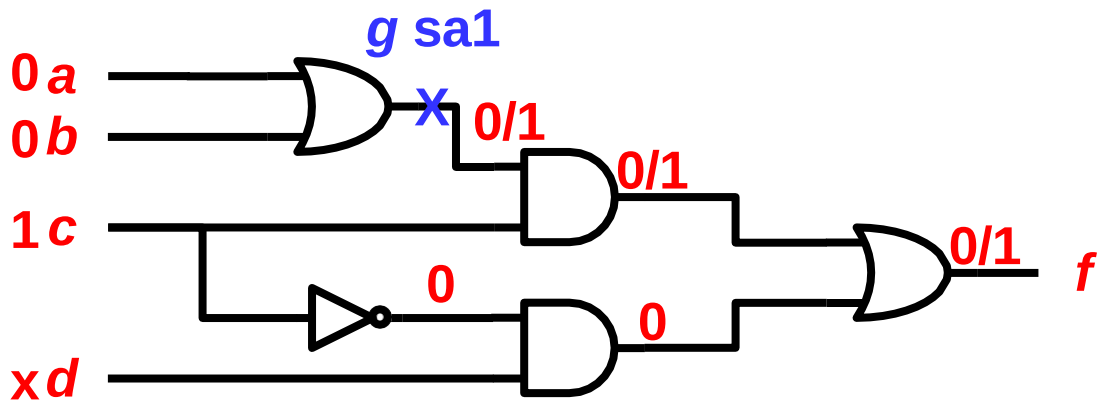
- **Same approach also used for stuck-at faults internal faults**
- **Let *g* be internal signal of Boolean function *f***

$$f(x_1, x_2, \ldots, y_1, y_2, \ldots) = W(g, y_1, y_2, \ldots)$$

- **Test sets for faults**
  - ***g* SA0:** $g\dfrac{dW}{dg} = 1$
  - ***g* SA1:** $g'\dfrac{dW}{dg} = 1$



Example



$$f = (a+b)c + c'd \qquad g = a+b$$

$$W = gc + c'd$$

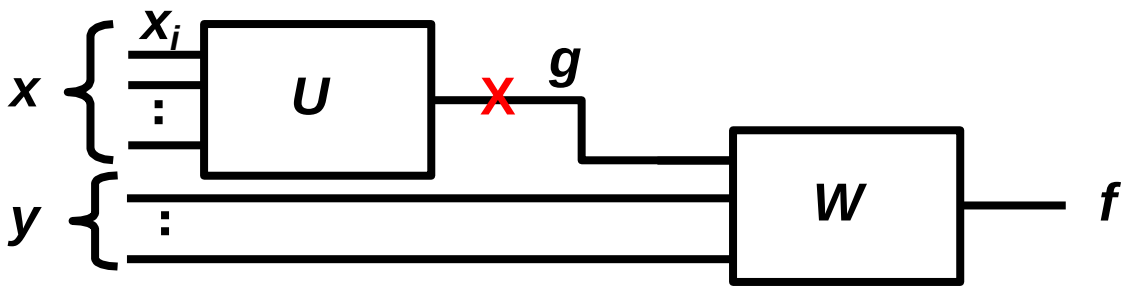
$$\frac{dW}{dg} = W_{g=0} \oplus W_{g=1} = c'd \oplus (c + c'd) = c$$

- **Tests for *g* sa0**
  (*a* + *b*)*c* = *ac* + *bc*
- **Tests for *g* sa1**
  (*a* + *b*)'*c* = *a*'*b*'*c*

Chain Rule

- **Chain rule:**

$$\frac{df}{dx_i} = \frac{df}{dg}\frac{dg}{dx_i}$$



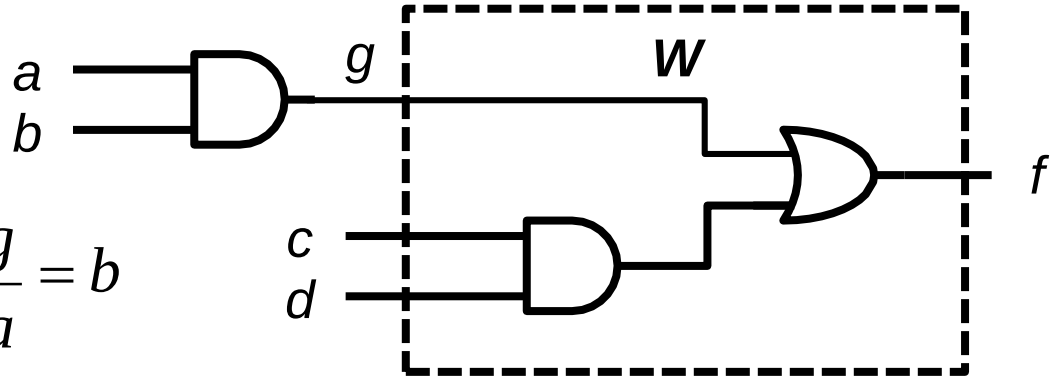
- **Example**

$$g = ab$$

$$W = g + cd$$

$$\frac{dW}{dg} = 1 \oplus cd = c'+d' \qquad \frac{dg}{da} = b$$

$$\frac{df}{da} = \frac{dW}{dg}\frac{dg}{da} = (c'+d')b$$



Summary

- **Deterministic Test Pattern Generation**
  - **BD=1 means sensitization condition**

$$\frac{df}{da} = [\, f_{a=0} \oplus f_{a=1} \,] = 1$$

  - **Test patterns to detect a stuck-at-0 fault**

$$a \frac{df}{da} = 1$$

  - **Test patterns to detect g stuck-at-0 fault**

$$g \frac{dW}{dg} = 1$$

  - **Chain rule**

$$\frac{df}{dx_i} = \frac{df}{dg} \frac{dg}{dx_i}$$

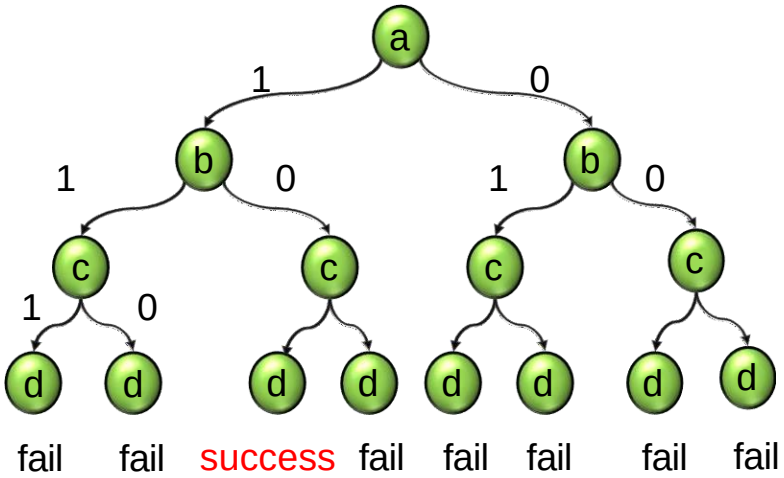
Combinational ATPG

- Introduction
- **Deterministic Test Pattern Generation**
  - Boolean difference *
  - **Path sensitization ****
  - D-Algorithm**
  - PODEM**
  - FAN**
  - SAT-based *
- Acceleration Techniques
- Concluding Remarks

Two ATPG categories:

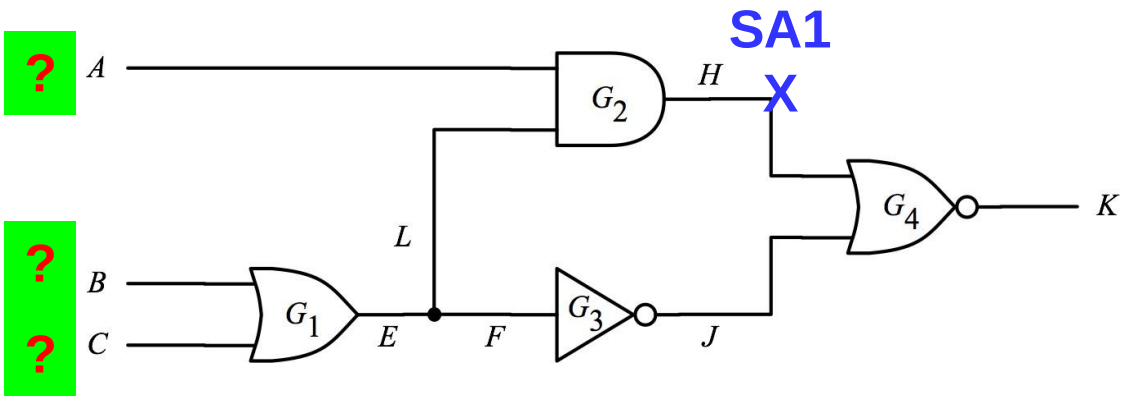
***Boolean-based methods**

****Path-based methods**



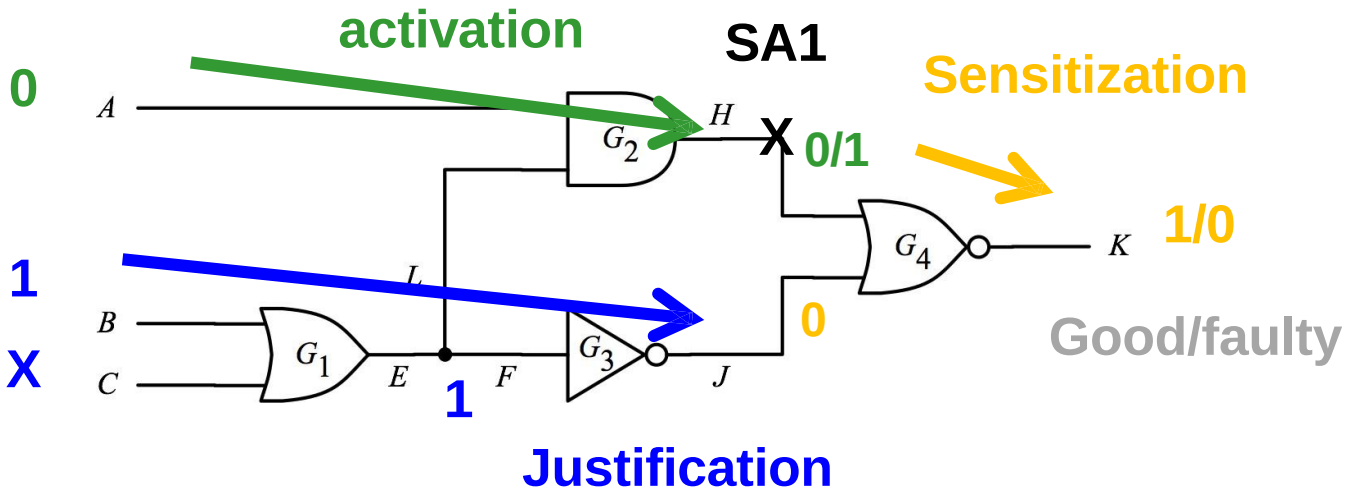
Motivating Problem

- We do not need to know Boolean expression
  - we can just find a test from circuit netlist



Let's Analyze What We Did

- ***Fault activation:*** **Assign gate inputs to generate appropriate value at fault site (*H*)**
  - ***A=0***
- ***Sensitization:*** **Assign side-inputs to non-controlling value to propagate fault effect forward**
  - ***J=0***
- ***Justification:*** **Assign primary inputs to achieve desired values**
  - ***B=1***



Single Path Sensitization

- **Single path sensitization (SPS) Algorithm:**

① ***Fault activation*** **(aka.** ***Fault excitation*****)**

Assign gate inputs to generate value at fault site

Desired value opposite to the faulty value (*e.g.* 0 for SA1)

② ***Fault effect propagation*****:**

Select one single path from fault site to an output

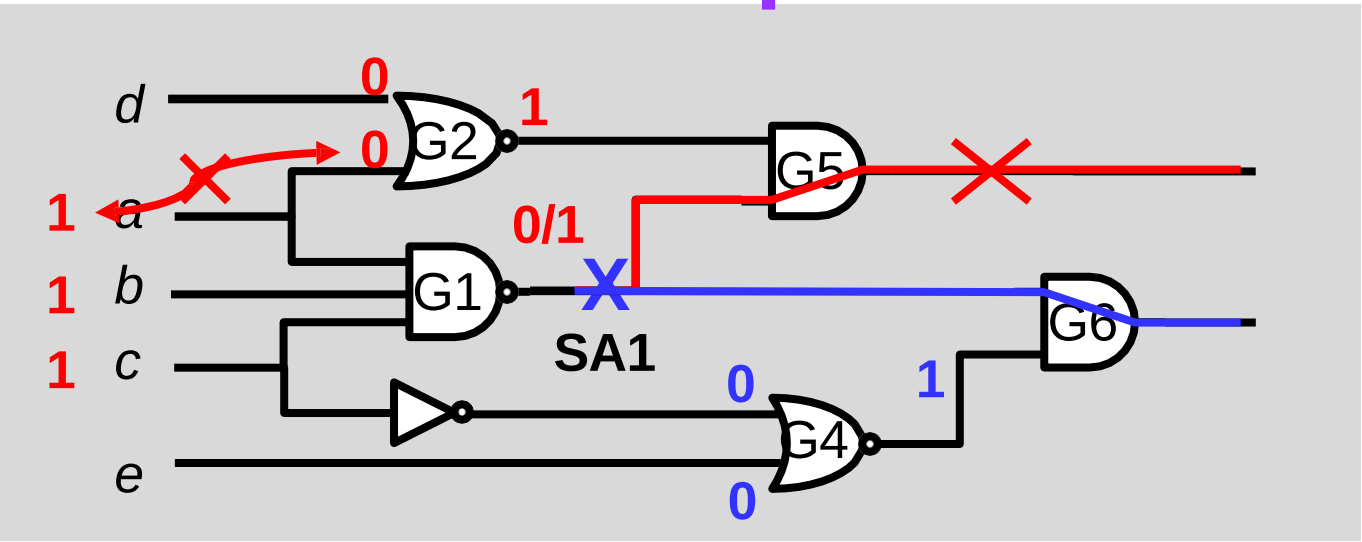
Assign side inputs to sensitize fault effect along the path

③ ***Justification*****:**

Assign primary inputs to justify desired values assigned in ①&②

If justification fails, backtrack.

Example



Consider stuck-at-1 fault

① **Fault activation**

- ***a* = *b* = *c* = 1**

② **Fault effect propagation: two propagation paths**

- **Choose path {G5}. Want G2 = 1**

③ ***a* = *d* = 0 → justification fails!**

② **Backtrack! Choose another path {G6}. Want G4 = 1**

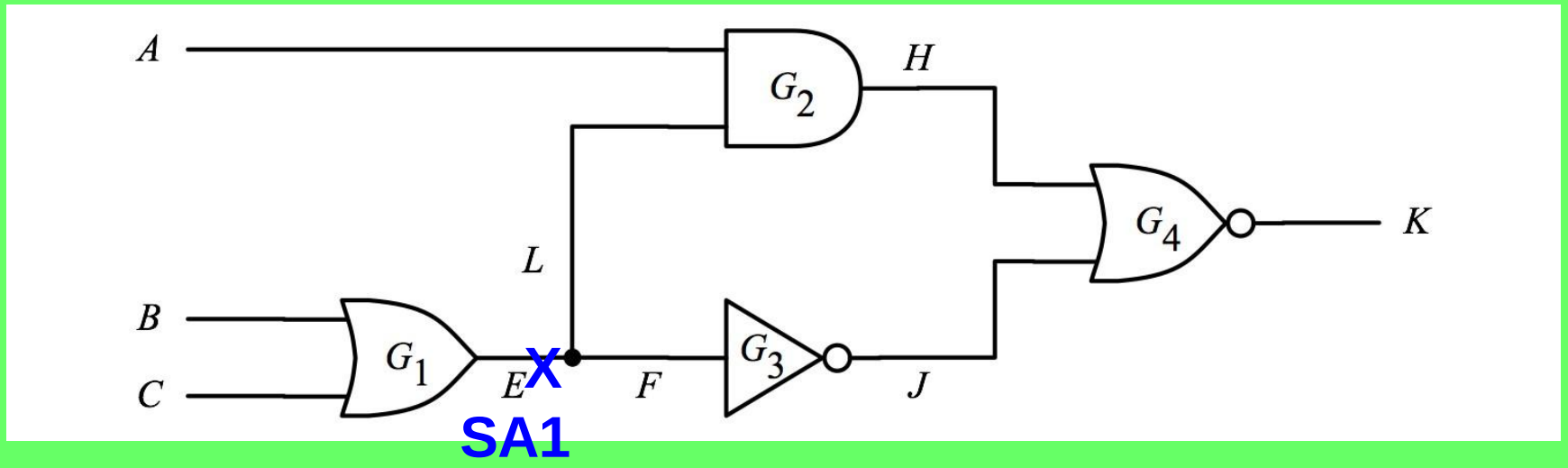
③ ***c* = 1, *e* = 0 → justification succeeds**

- **test pattern: *abce'* generated**

Quiz

Q1: Generate a test pattern for *E* SA1 fault. Choose path *ELHK*.
A:

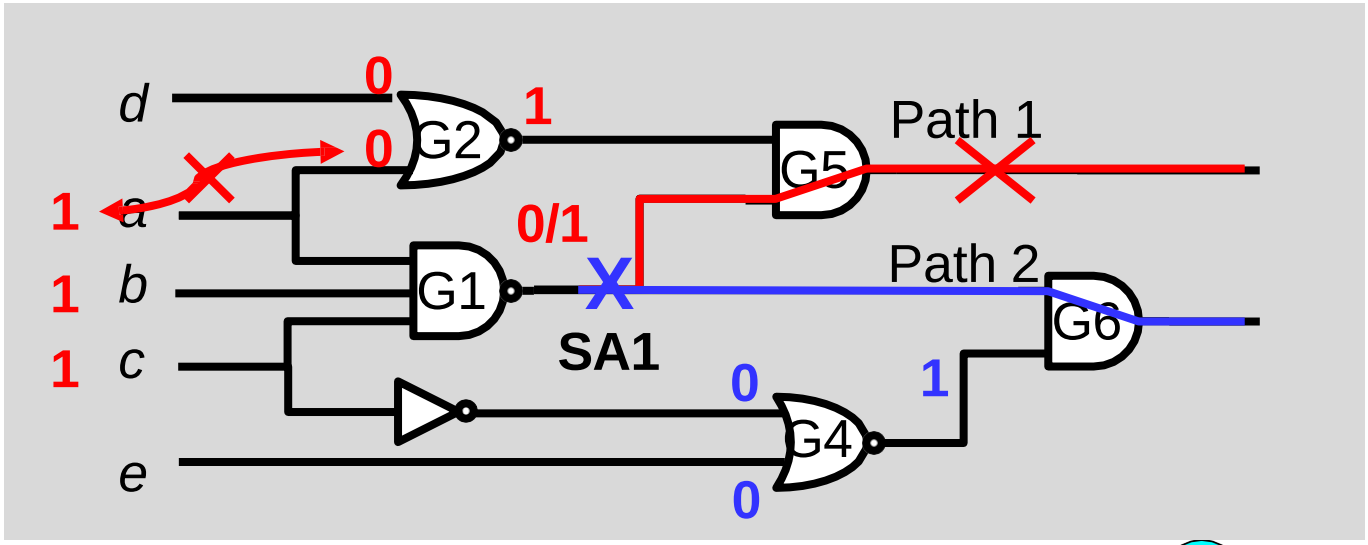
Q2: (Cont'd) Backtrack to another path *EFJK*.
A:



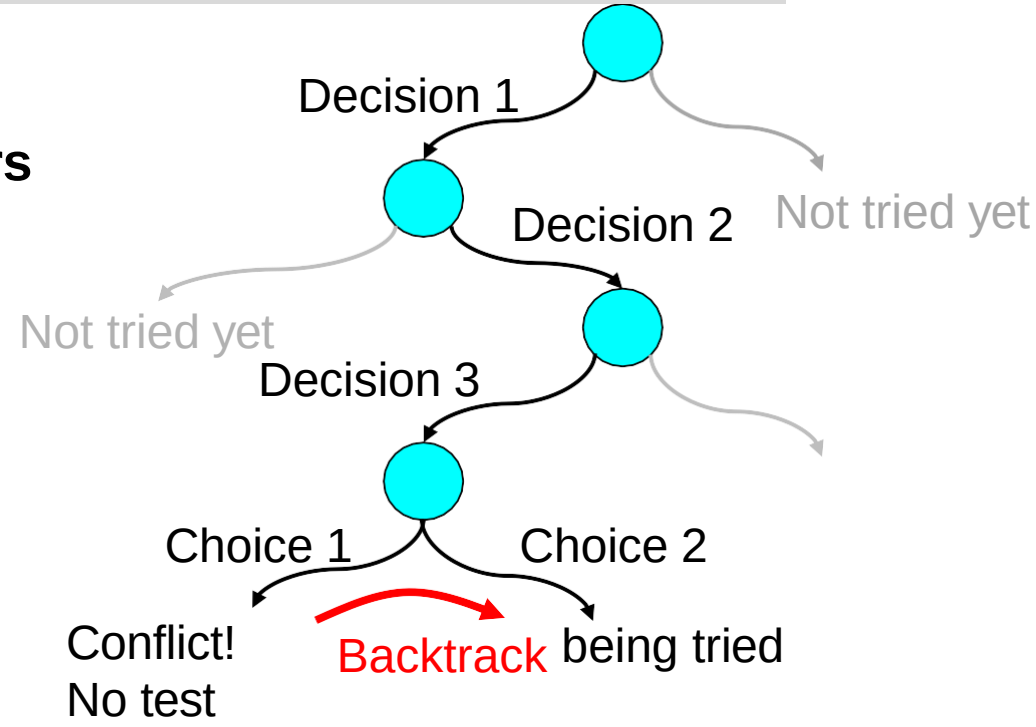
Important to Choose Correct Path

But it is difficult…

Backtrack



- **When we made a mistake in decision tree, conflict occurs**
  - **Go back to a previous decision point**
  - **Change decision**
  - **Redo the rest**



Pros and Cons

☺ **Pros**

- **Easy to implement**
- **No Boolean equation needed**

☹ **Cons**

- **Q1: Too many paths to choose, which one is correct?**
- **Q2: Single-path sensitization not enough to detect all faults**

- **Single path sensitization (SPS) Algorithm:**

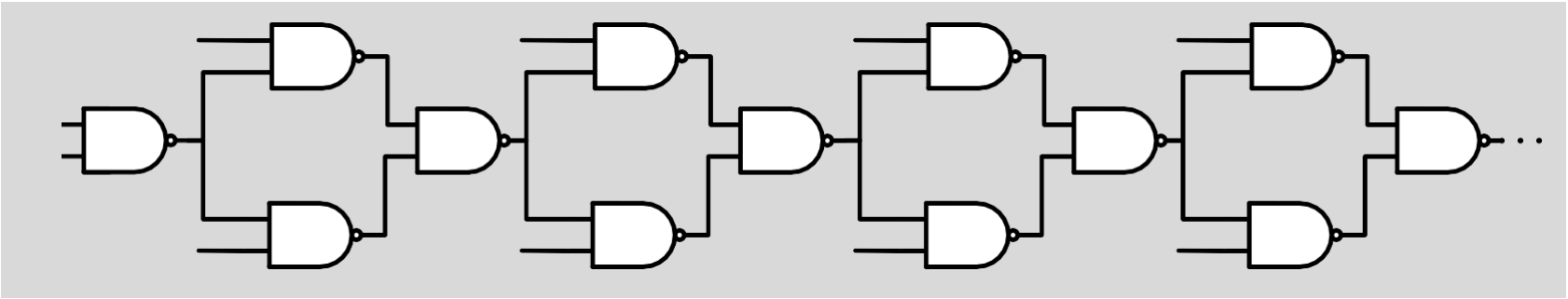
① ***Fault activation*** **(aka.** ***Fault excitation*****)**
Assign internal signals to generate value at fault site
Desired value opposite to the faulty value

② ***Fault effect propagation*****:**
Select **one single path** from fault site to an output
Assign internal signals to sensitize fault effect along the path

③ ***Justification*****:**
Assign primary inputs to justify signal values assigned in ①&②
If justification fails, **backtrack**.

Q1: Too Many Paths !

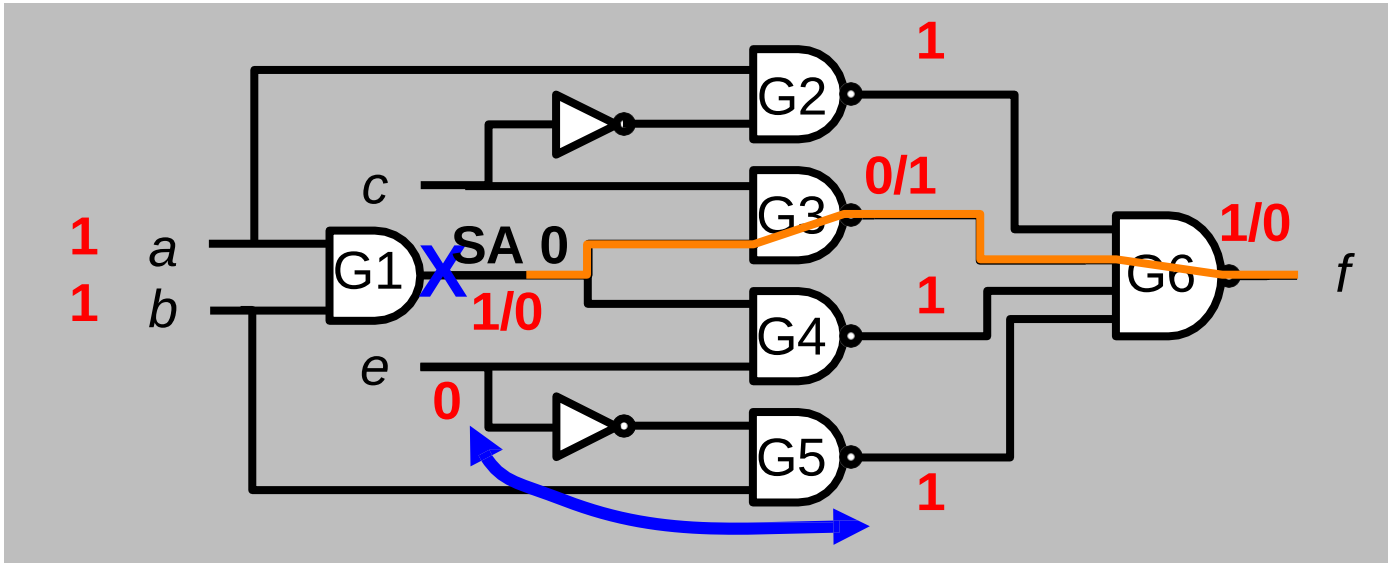
- How to choose correct path?
  - No smart algorithm
  - Simple idea: exhaustively try all paths
- How many paths in a circuit?
  - Worst case example: $3n$ gates, 2^n paths! (n = # of stages)



# of Paths is Exponential to Circuit Size!
Impossible to Try All

Q2: Single Path Not Enough

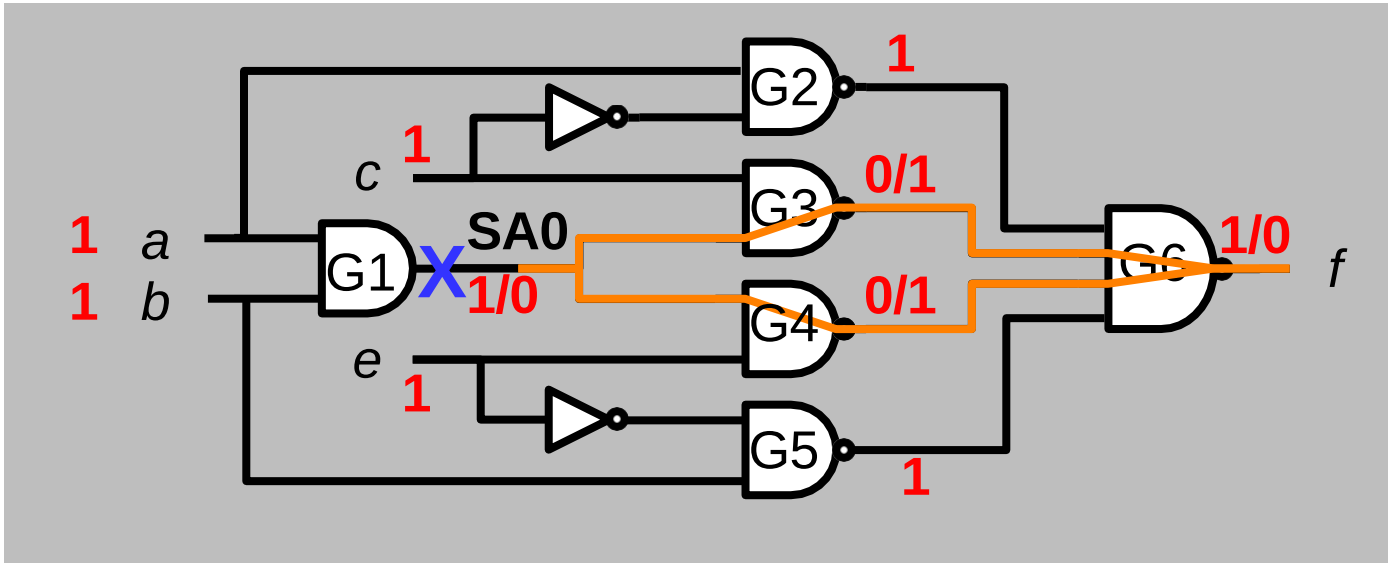
- **Example**
  - ① **Fault activation: *a* = *b* = 1**
  - ② **Fault propagation: Choose path {G3-G6}. want G2 = G4 = G5 = 1**
  - ③ **G4 = 1 → *e* = 0 → G5 = 0 → justification fails**
  - ② **Choose another path {G4-G6}. Justification also fails.**
  - ♦ **SPS algorithm fails**



This Fault Is Actually Testable. Why SPS Fail?

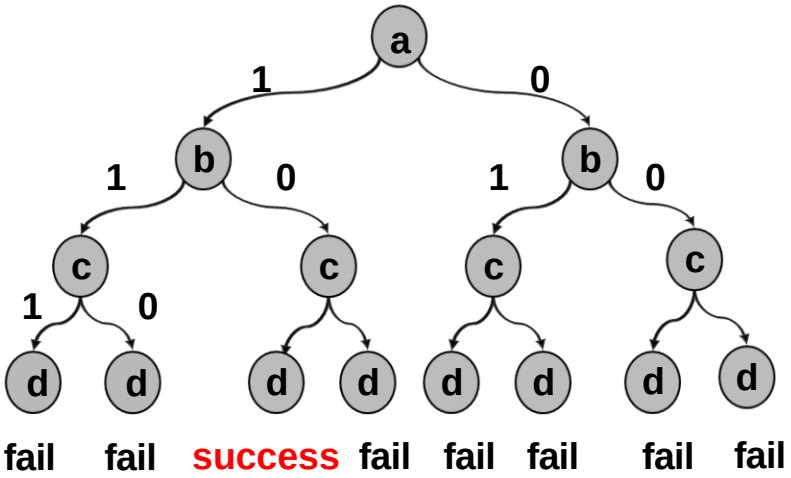
Solution: Multiple Path Sensitization

- This fault requires *multiple path sensitization*
- Both two paths {G3-G6} and {G4-G6} are sensitized
  - Error is propagated along both paths simultaneously
  - G2 = G5 = 1 , *c* = *e* = 1
  - Test generated successfully
- FFT: can we extend 1-path to 2, 3, 4, 5…path sensitization?



Complete ATPG Algorithm

- A *complete ATPG* exhausts the whole input space (2^n)
  - If a test pattern exists, complete ATPG will find it for sure
- An *incomplete ATPG* does NOT exhaust the whole input space
  - ATPG may fail even though a test pattern DOES exist



Complete ATPG Guarantees to Find Solution
if it exists

Quiz

Q: Is Single Path Sensitization a complete ATPG algorithm?

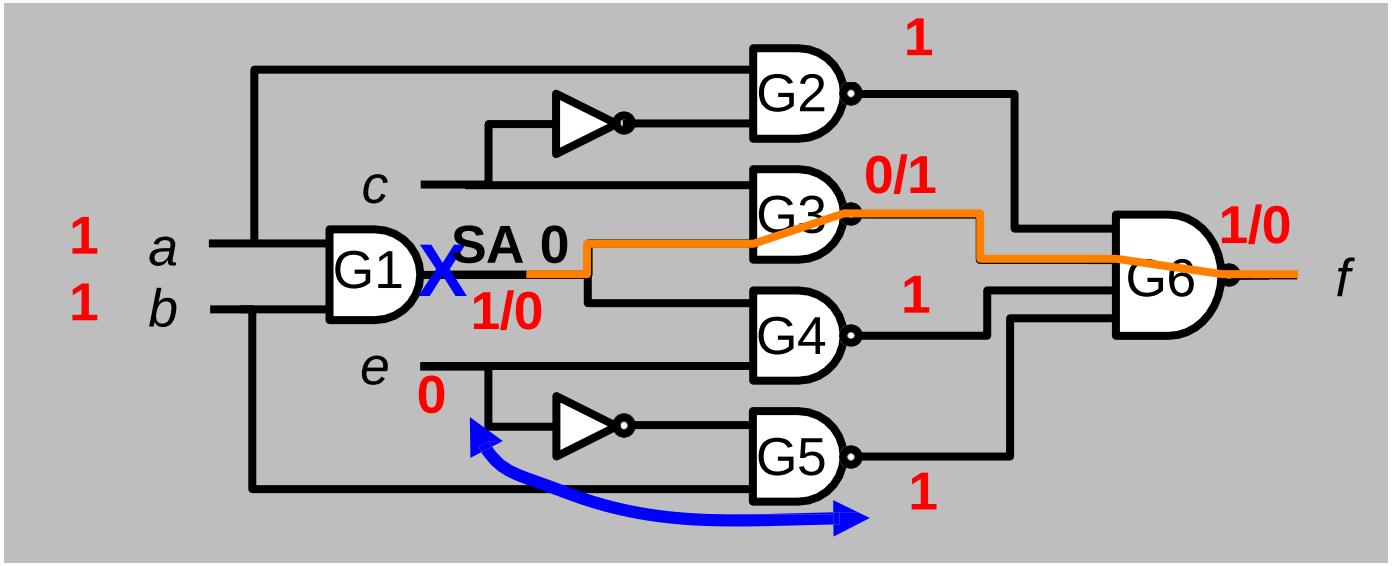
Hint: recall this example

- **Single path sensitization (SPS) Algorithm:**

① *Fault activation* (aka. *Fault excitation*)
Assign internal signals to generate value at fault site
Desired value opposite to the faulty value

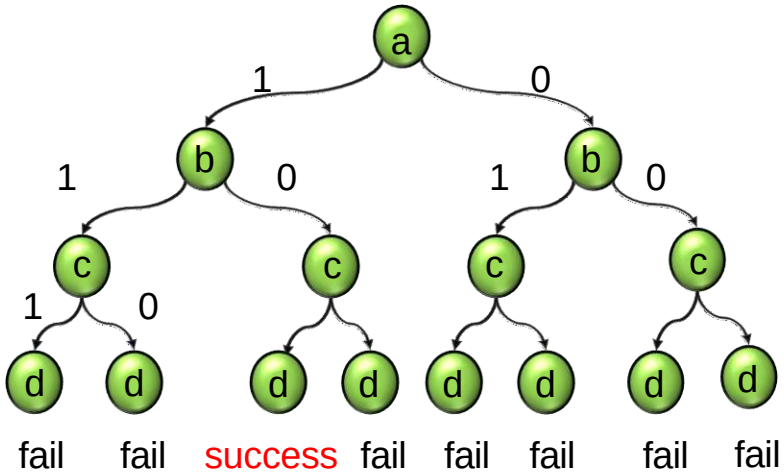
② *Fault effect propagation*:
Select one single path from fault site to an output
Assign internal signals to sensitize fault effect along the path

③ *Justification*:
Assign primary inputs to justify signal values assigned in ①&②
If justification fails, backtrack.



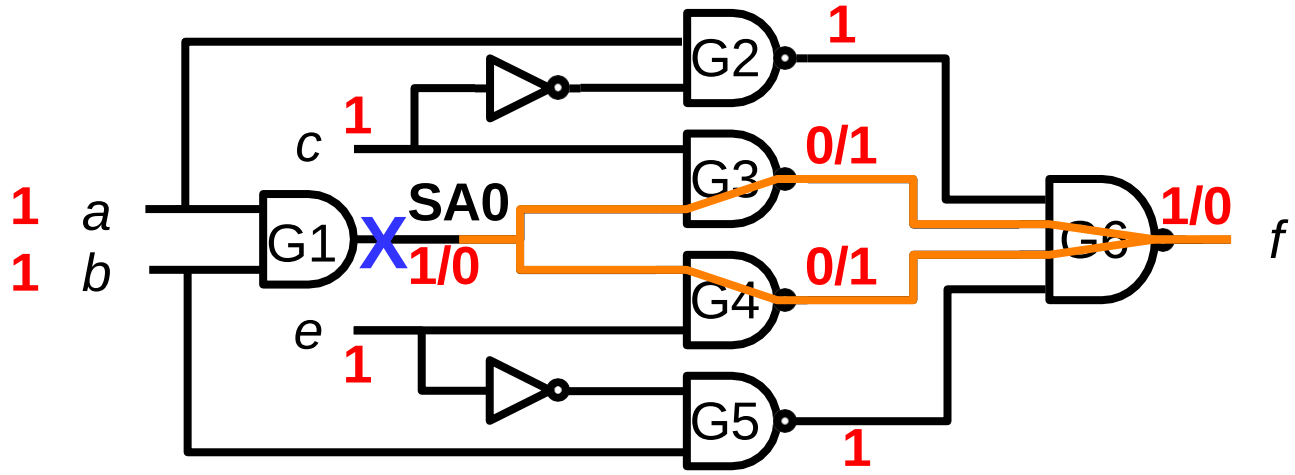
Summary

- **Single Path Sensitization Algorithm**
  - ① **Fault activation**
  - ② **Fault effect propagation**
  - ③ **Justification**
- **Complete ATPG: guarantee to find solution if one exists**
- **Disadvantages of SPS algorithm**
  - **Number of paths is exponential to circuit size. Too many!**
  - **Single path sensitization is incomplete ATPG**



FFT

- **Q1: If SPS fails, can we extend 1-path to 2, 3, 4, 5… paths?**
  - **What is the problem for multiple path sensitization algorithm?**
- **Q2: Pros and Cons of complete ATPG algorithm?**
  - **in terms of run time and fault coverage**



Combinational ATPG

- **Introduction**
- **Deterministic Test Pattern Generation**
  - **Boolean difference ***
  - **Path sensitization ****
  - **D-Algorithm [Roth 1966] ****
  - **PODEM [Goel 1981]****
  - **FAN [Fujiwara 1983]****
  - **SAT-based [Larrabee 1992]***
- **Acceleration Techniques**
- **Concluding Remarks**

*Boolean-based methods

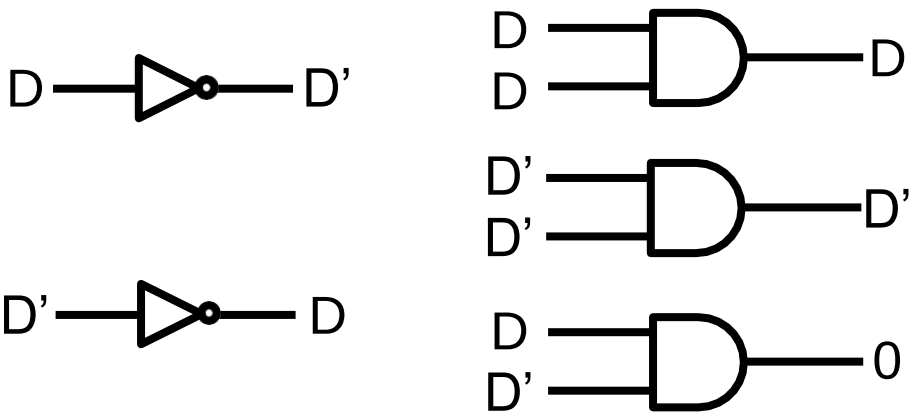
**path-based methods

D-Algorithm

- The D-algebra
- **An D-algorithm example**
- **Types of cubes**
- **Implication and Justification**
- **Flowchart of the D-algorithm**
- **Another example**
- **Problems with the D-Algorithm**

D-Algebra

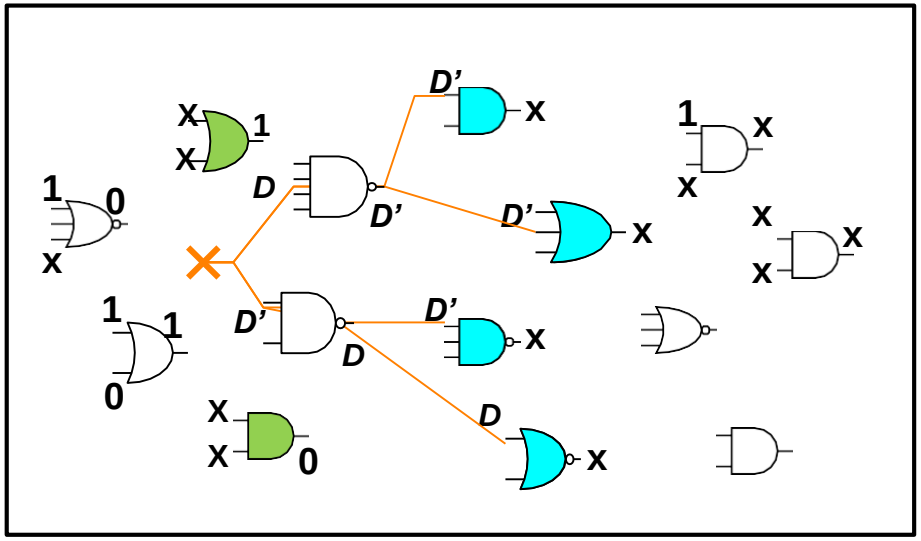
- ***Five-valued logic*: 1, 0, D, D', X**
  - **Symbol D**
  - **D = 1/0**
    - **1 in fault-free circuit and 0 in the faulty circuit**
  - **D' = $\overline{D}$ = 0/1**
    - **0 in fault-free circuit and 1 in the faulty circuit**
  - **x means " not yet specified" in ATPG**



AND	0	1	D	$\overline{D}$	X
0	0	0	0	0	0
1	0	1	D	$\overline{D}$	X
D	0	D	D	0	X
$\overline{D}$	0	$\overline{D}$	0	$\overline{D}$	X
X	0	X	X	X	X

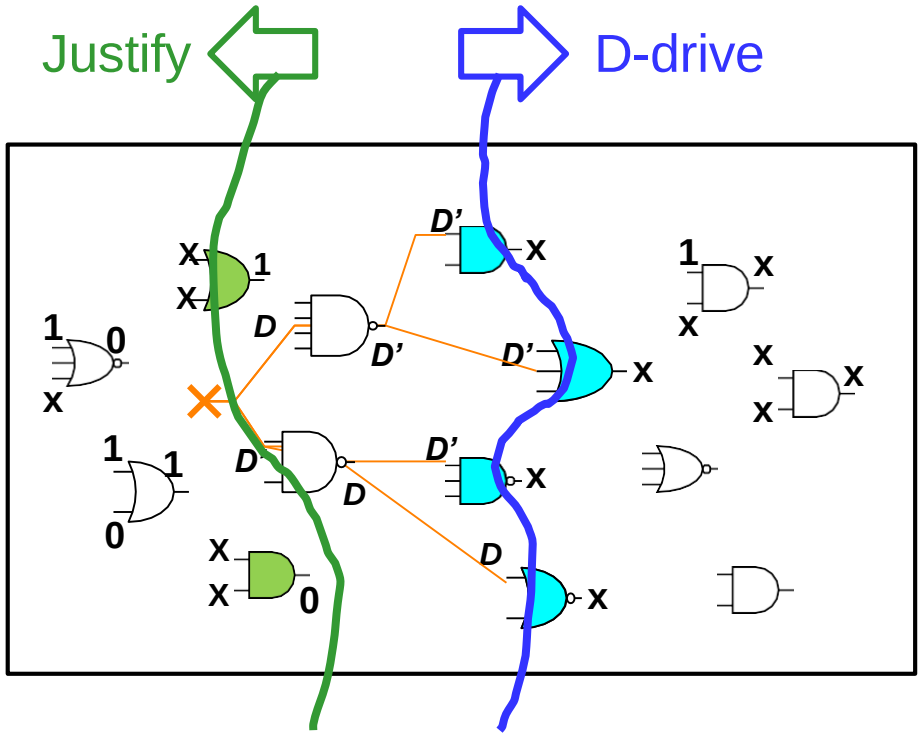
D-frontier , J-frontier

- ***D-frontier*** **: a set of gates whose output value is currently x,**
  - **but have one or more D (or D') at their inputs**
- ***J-frontier*****: a set of gates whose output value is assigned**
  - **But input values have not been decided yet**
- **Example**
  - **Blue gates are D-frontier; Green gates are J-frontier**



Idea of D-Algorithm

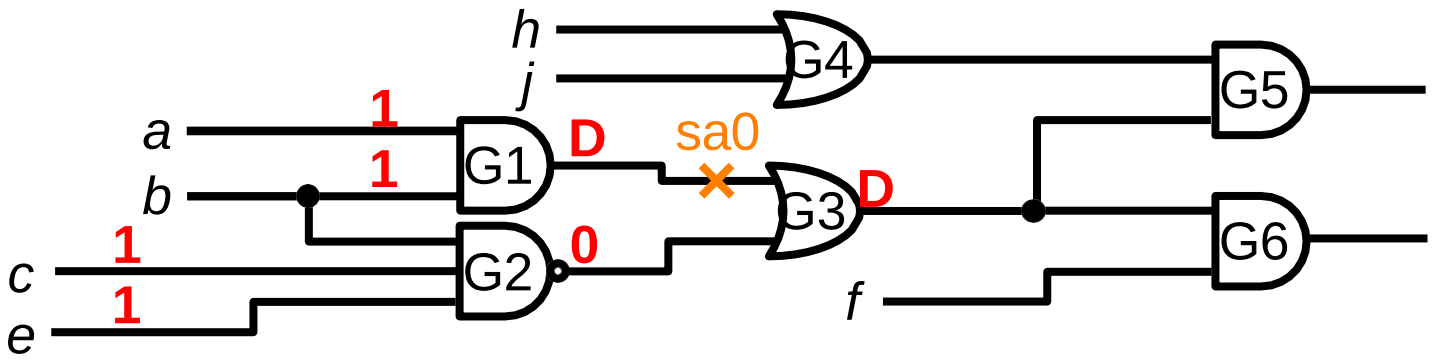
- 1. Create D-frontier (fault activation)
- 2. *Drive* D-frontier toward output (fault effect propagation)
- 3. *Justify* J-frontiers
- 4. *Backtrack* if any conflict occurs



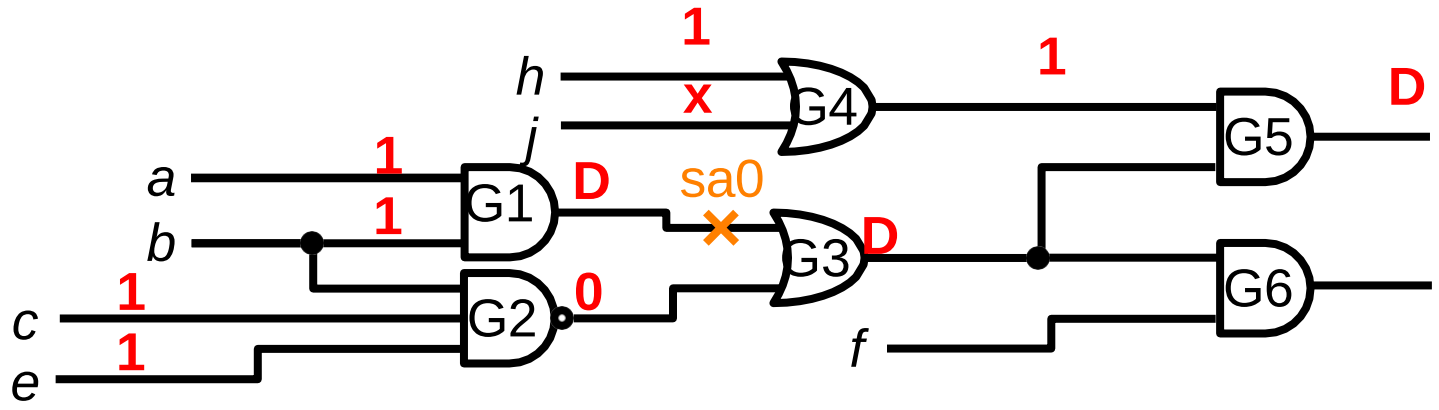
The D-Algorithm

- **The D-algebra**
- A D-algorithm example
- **Types of cubes**
- **Implication and Justification**
- **Flowchart of the D-algorithm**
- **Another example**
- **Problems with the D-Algorithm**

Example



	a	b	c	e	G_1	G_2	G_3	h	j	G_4	f	G_5	G_6	Comments
Initial test cube TC(0) = PDFC for G_1 stuck-at 0	**1**	**1**			**D**									**Fault Activation. Implication: nothing happen.**
Prop. D-cube of G_3, PD_{G3}					**D**	**0**	**D**							**D-frontier: {G3}**
$TC(1)=TC(0)\cap PD_{G3}$	**1**	**1**			**D**	**0**	**D**							**D-Drive through G3**
Singular Cover SC_{G2}		**1**	**1**	**1**		**0**								
$TC(2)=TC(1)\cap SC_{G2}$	**1**	**1**	**1**	**1**	**D**	**0**	**D**							**Backward implication**



	a	b	c	e	G_1	G_2	G_3	h	j	G_4	f	G_5	G_6	Comments
TC(2)	**1**	**1**	**1**	**1**	**D**	**0**	**D**							**D-frontier: {G_5,G_6}**
Propagation D-cube of G_5, PD_{G5}							**D**			**1**		**D**		**Choose path through G_5**
TC(3)=TC(2)∩PD_{G5}	**1**	**1**	**1**	**1**	**D**	**0**	**D**			**1**		**D**		**D-drive through G_5. D reach PO.**
Singular Cover SC_{G4}								**1**	**X**	**1**				**Justification J-frontier = {G_4}**
TC(4)=TC(3)∩SC_{G4}	**1**	**1**	**1**	**1**	**D**	**0**	**D**	**1**	**X**	**1**	**X**	**D**	**X**	**Done.**

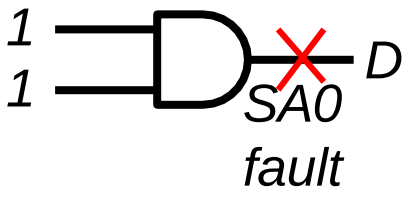
*Underlined numbers are unjustified lines

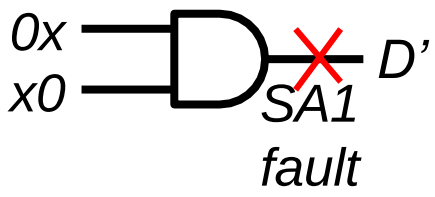
D-Algorithm

- **The D-algebra**
- **An D-algorithm example**
- Types of cubes
- **Implication and Justification**
- **Flowchart of the D-algorithm**
- **Another example**
- **Problems with the D-Algorithm**

Primitive D-Cubes for a Fault (PDCF)

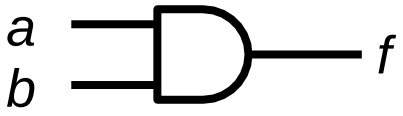
- **Specify minimal input conditions**
  - **applied to gate input to produce error at gate output**
  - **Used in *fault activation* (more on this later)**
- **Example: AND gate output stuck-at faults:**
  - **Stuck-at-0 fault: 11D**
  - **Stuck-at-1 fault: 0xD' and x0D'**





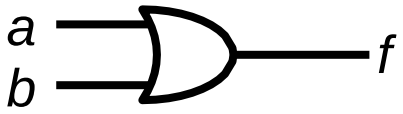
Singular Cover (SC)

- **Minimum gate input assignments for gate output =0 or =1**
  - **Used in line *justification* or *implication* (more on this later)**



Singular covers of AND

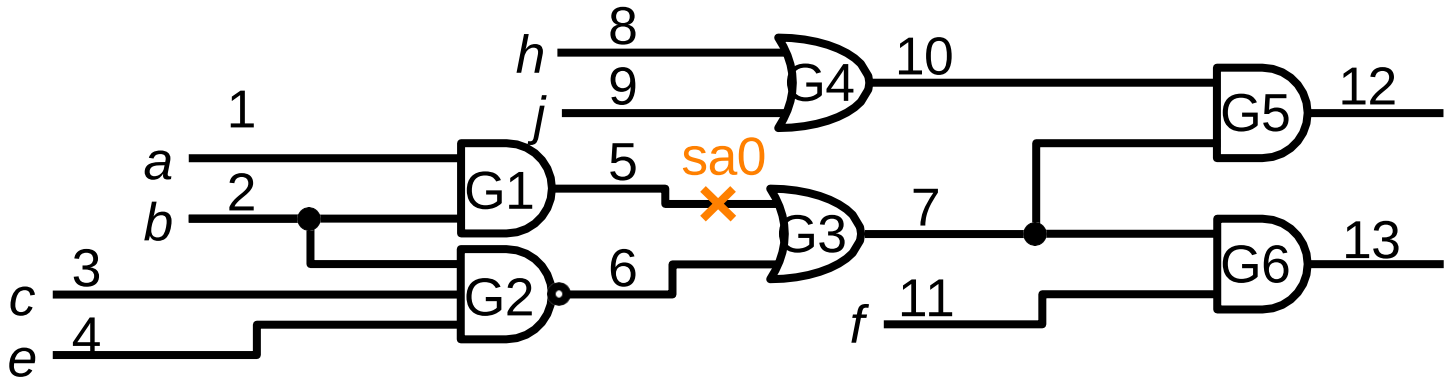
a	*b*	*f*
1	1	1
0	x	0
x	0	0



Singular covers of OR

a	*b*	*f*
0	0	0
1	X	1
X	1	1

Example



	a	b	c	e	G_1	G_2	G_3	h	j	G_4	f	G_5	G_6
Primitive D-cube for G1 sa0 fault	1	1			D								
Singular covers of G2		0 x x 1	x 0 x 1	x x 0 1		1 1 1 0							
Singular covers of G3					1 x 0	x 1 0	1 1 0						
Singular covers of G4								1 x 0	x 1 0	1 1 0			
Singular covers of G5							x 0 1			0 x 1		0 0 1	
Singular covers of G6							x 0 1				0 x 1		0 0 1

Propagation D-Cube (PDC)

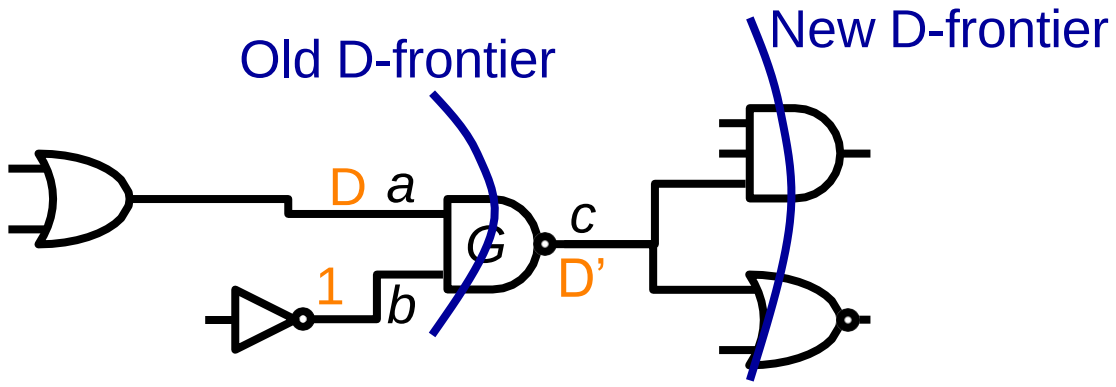
- **Minimum gate input assignments required**
  - **to propagate a D or D’ from gate input(s) to gate output**
- **Used in *D-Drive* or *implication* (more later)**

A	*B*	*AB*
D	1	D
1	D	D
D	D	D
D’	1	D’
1	D’	D’
D’	D’	D’

A	*B*	(*A*+*B*)’
0	D	D’
D	0	D’
D	D	D’
0	D’	D
D’	0	D
D’	D’	D

D-drive

- **D-drive selects an element in D-frontier**
  - **and attempts to propagate D or D’ from gate input to gate output**
  - **Using propagation D-cube**



	…	*a*	*b*	*c*	…
Test cube before D-drive	…	D	X	X	
Propagation D cube of gate *G*	…	D	1	D’	…
Test cube after D-drive	…	D	1	D’	…

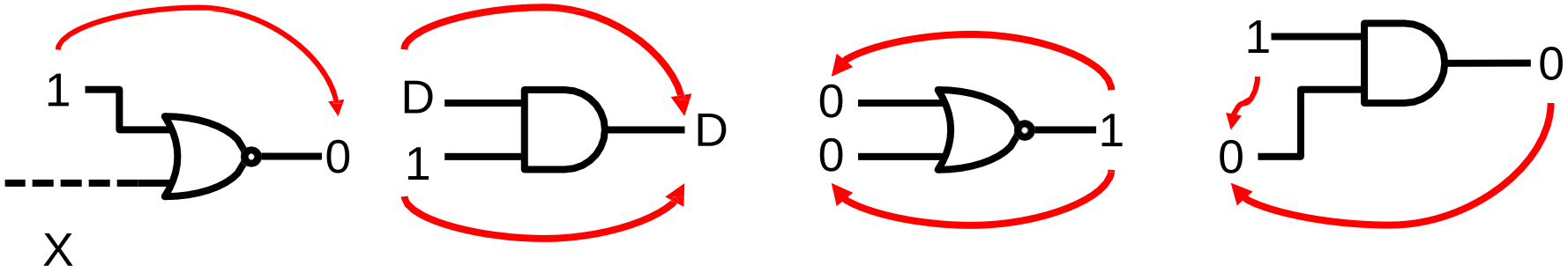
The D-Algorithm

- **The D-algebra**
- **An D-algorithm example**
- **Types of cubes**
- Implication and Justification
- **Flowchart of the D-algorithm**
- **Another example**
- **Problems with the D-Algorithm**

Implication

- ***Forward Implication***
  - **partially (or fully) specified input values *uniquely* determines the output values.**
- ***Backward Implication***
  - **knowing the output values (and some input values) can *uniquely* determine the un-specified input values.**

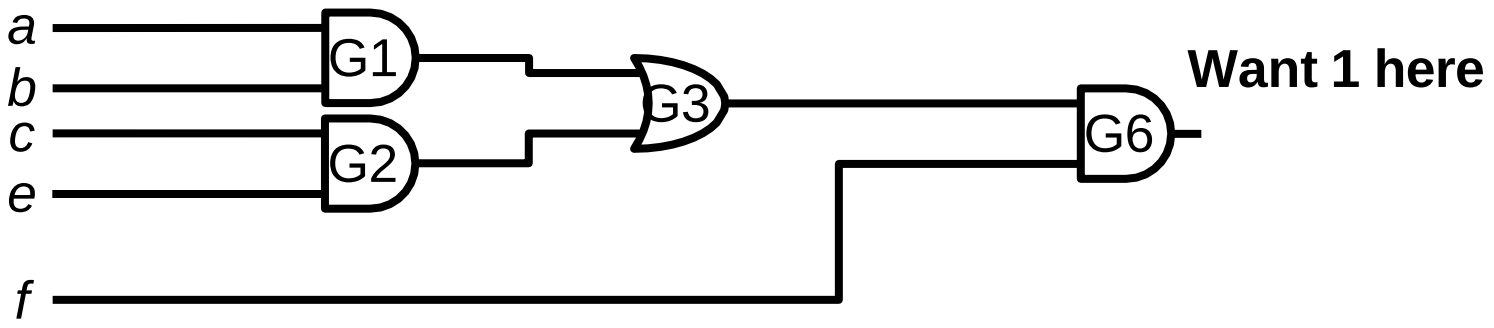
- **Examples**



- **Note 1: Implication means NO choice**
- **Note 2: Implication can be done any time a decision is made**

Justification

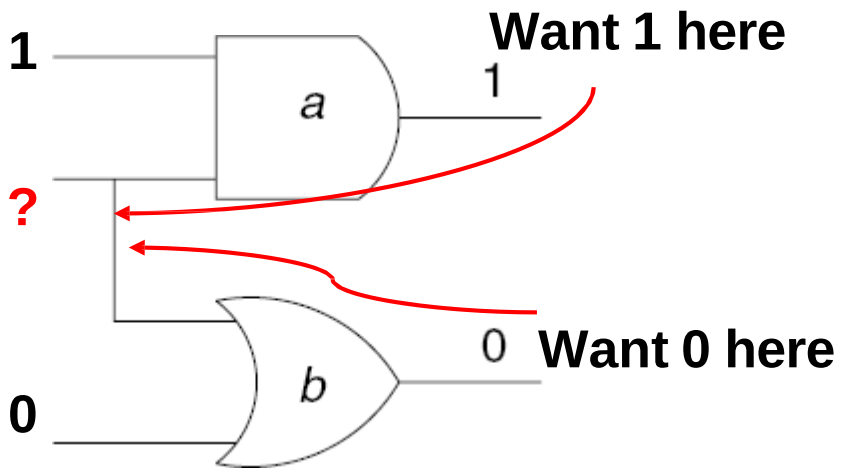
- **Definition: find a valid primary input assignment for desired values**
- **Justification is easy inside a *fanout-free* circuit**
- **No decision needed**
  - **Always finds an answer**
- **Example**



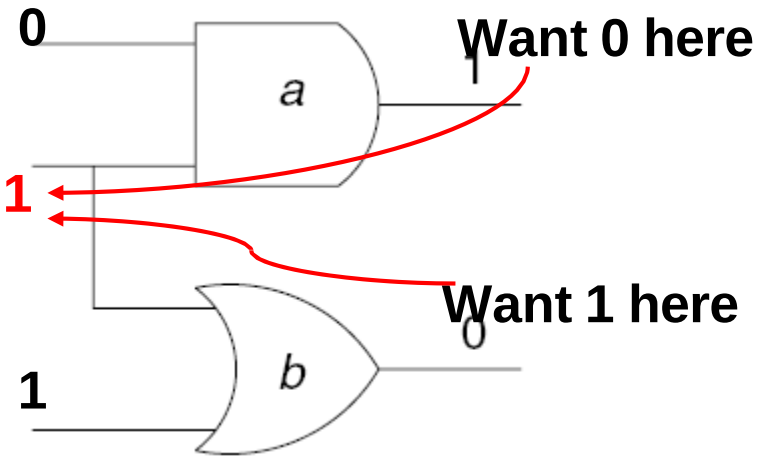
Justification (2)

- Justification may fail when there are fanout branches

Justification fails



Justification succeeds



The D-Algorithm

- **The D-algebra**
- **A D-algorithm example**
- **Types of cubes**
- **Implication and Justification**
- Flowchart of the D-algorithm
- **Another example**
- **Problems with the D-Algorithm**

Test Cube (TC)

- **A *Test cube* is a partially specified Boolean values for testing a fault**
- **In D algorithm, a test cube contains**
  - **not only primary inputs, but also internal nodes**
- **Notation**
  - **TC(*n*) = test cube at ATPG step *n***

- **NOTE: In PODEM, a test cube contains primary inputs only**
  - **More details later**

Intersection of Test Cubes

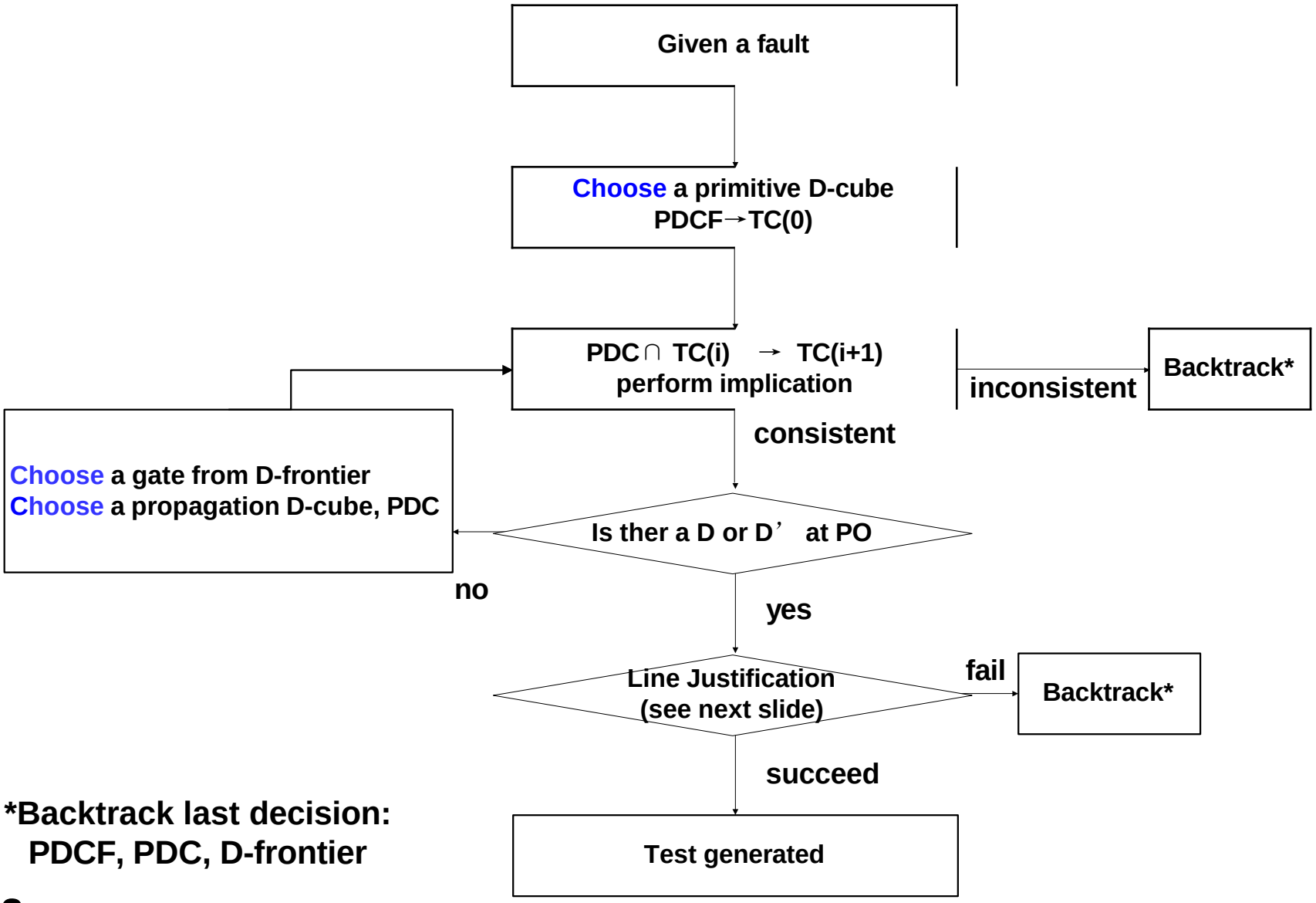
- Two bits has ***intersection*** if their logic values are not conflicting
- Two test cubes has intersection if there is no confliction in any bit
- Example :
  - TC(1) ∩ TC(2) = X0X1 ∩ 1XXX = 10X1
  - TC(1) ∩ TC(2) = 10X1 ∩ 0XXX = no intersection

Bit 1∩ Bit 2

bit2 \ bit1	0	1	X	D	D’
0	0		0		
1		1	1		
X	0	1	X	D	D’
D			D	D	
D’			D’		D’

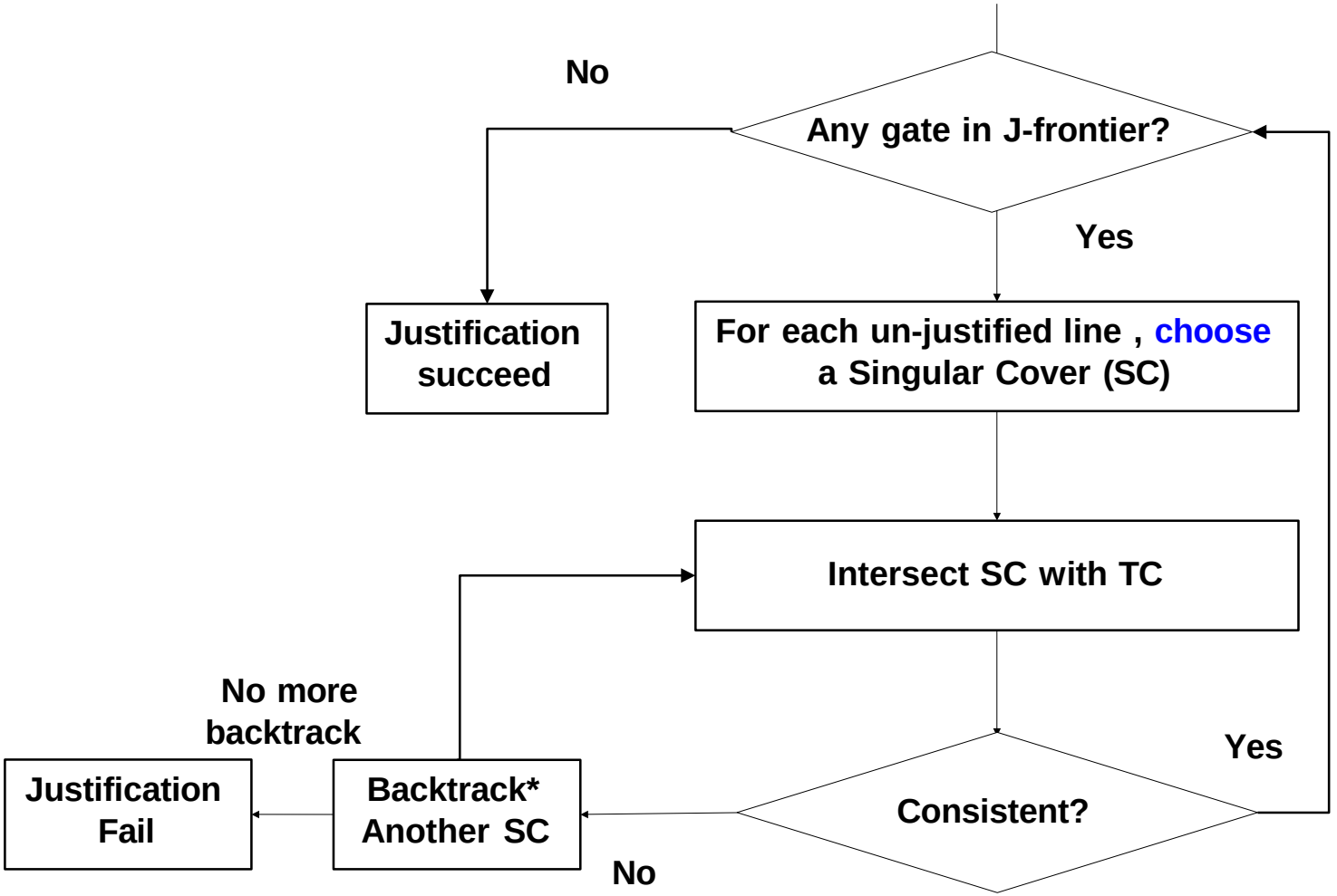
empty box = confliction = no intersection

Flowchart of the D-Algorithm



*Backtrack last decision:
PDCF, PDC, D-frontier

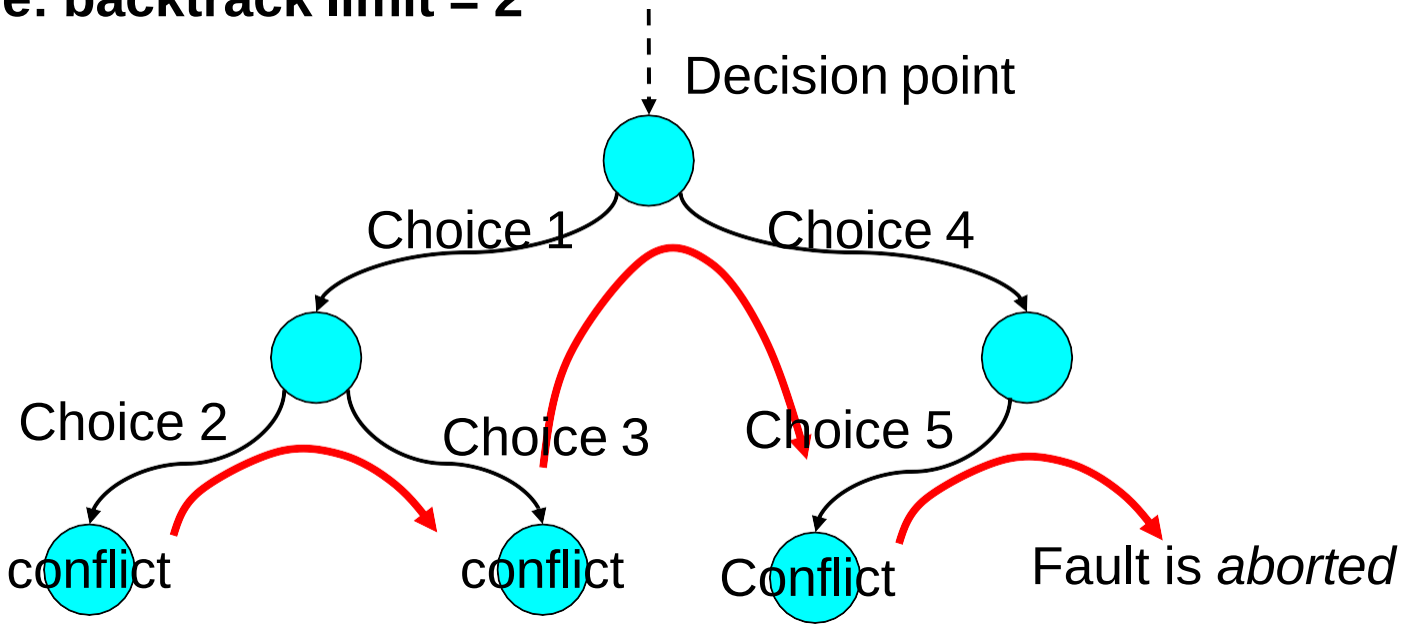
Line Justification



*backtrack last decision: SC

Backtrack

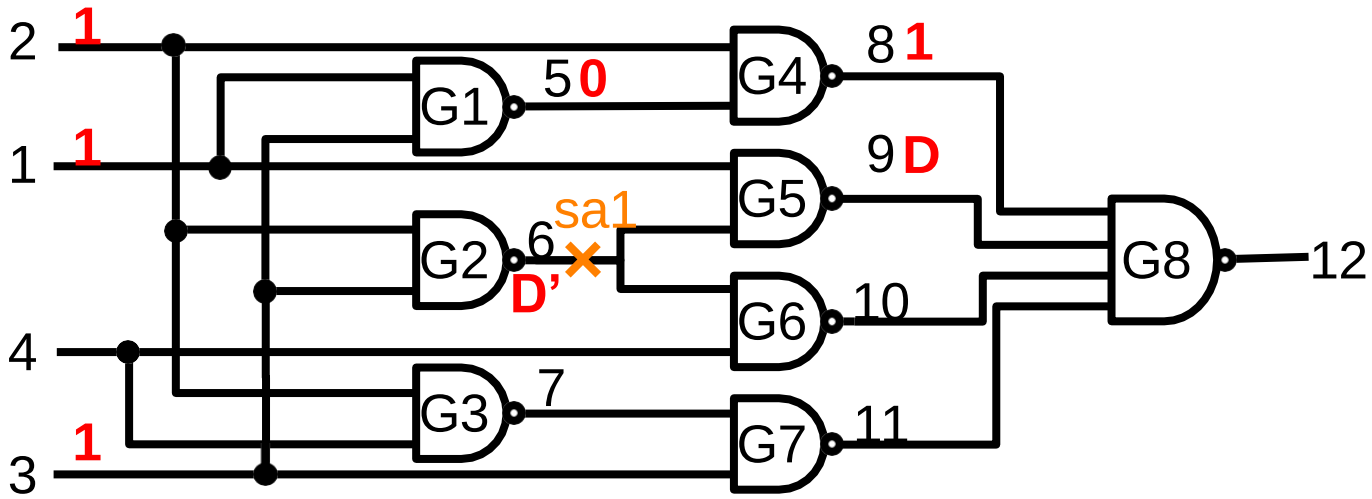
- When conflict, ***backtrack*** to last decision point and change choice
  - Choice can be: PDCF, PDC, SC, D-drive gate
- To avoid spending too much time on a fault
  - Use specify a ***backtrack limit***
    - maximum number of backtracks allowed for a single fault
  - Fault is ***aborted*** if backtrack limit is reached
- Example: backtrack limit = 2



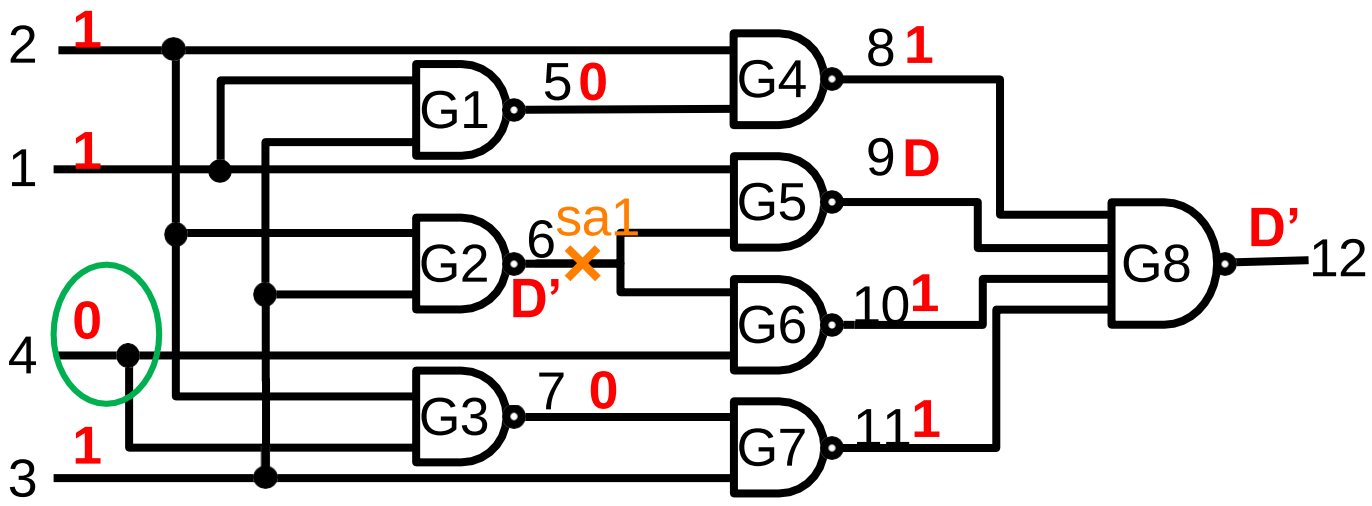
The D-Algorithm

- **The D-algebra**
- **A D-algorithm example**
- **Types of cubes**
- **Components of the D-algorithm**
- **Flowchart of the D-algorithm**
- Another example
- **Problems with the D-Algorithm**

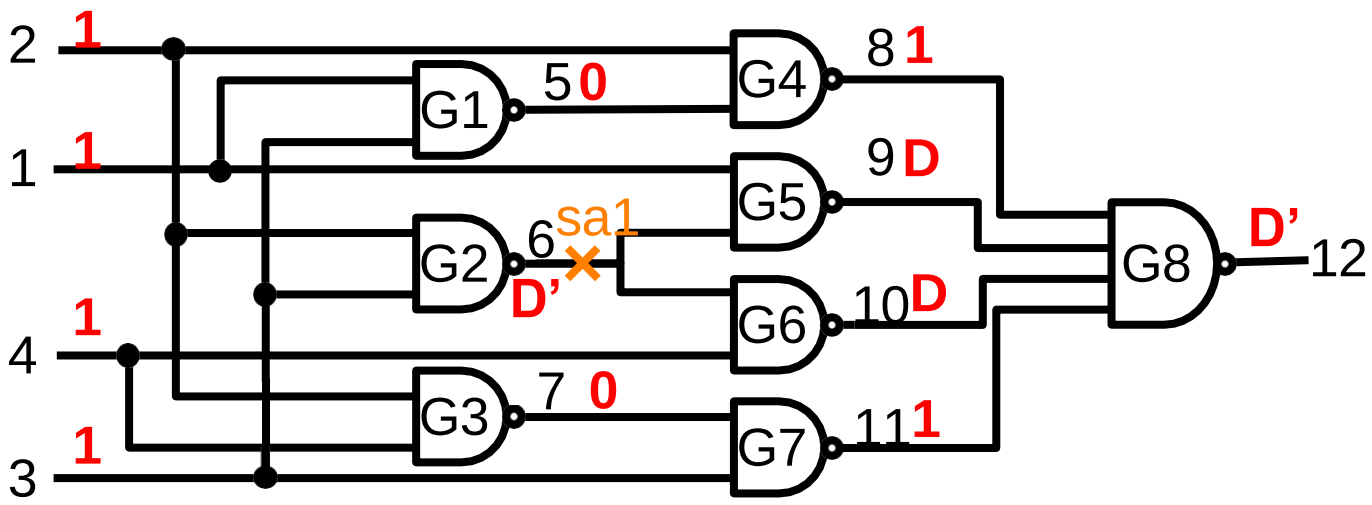
Another D-Algorithm Example



		1	2	3	4	5	6	7	8	9	10	11	12
1	PDCF:TC(0) = PDFC		1	1			D'						
2	Implication: nothing happens		1	1			D'						
3	choose D-frontier G5: PDC TC(1)=TC(0)∩ PDC	1 1	 1	 1			D' D'			D D			
4	Forward Implication: SC_{G1} TC(2)	1 1	 1	1 1		0 0	 D'			 D			
5	Forward Implication: SC_{G4} TC(3)	 1	X 1	 1		0 0	 D'		1 1	 D			



		1	2	3	4	5	6	7	8	9	10	11	12
	TC(3)	**1**	**1**	**1**		**0**	**D'**		**1**	**D**			
6	**choose D-frontier G8: PDC**								**1**	**D**	**1**	**1**	**D'**
	TC(4)	**1**	**1**	**1**		**0**	**D'**		**1**	**D**	**1**	**1**	**D'**
7	**Backward Implication G_7**			**1**				**0**				**1**	
	TC(5)	**1**	**1**	**1**		**0**	**D'**	**0**	**1**	**D**	**1**	**1**	**D'**
8	**Backward Implication G_6: SC_{G6}**				**0**		**D'**				**1**		
	TC(6)	**1**	**1**	**1**	**0**	**0**	**D'**	**0**	**1**	**D**	**1**	**1**	**D'**
9	**Backward Implication G_3: SC_{G3}**		**1**		**1**			**0**					
	Fail. Backtrack step to 6												



		1	2	3	4	5	6	7	8	9	10	11	12
	Backtrack to step 6. TC(3)	**1**	**1**	**1**		**0**	**D'**		**1**	**D**			
10	**Choose D-frontier G6: PDC**				**1**		**D'**				**D**		
	TC(4)	**1**	**1**	**1**	**1**	**0**	**D'**		**1**	**D**	**D**		
11	**Forward Implication: SC_{G3}**		**1**		**1**			**0**					
	TC(5)	**1**	**1**	**1**	**1**	**0**	**D'**	**0**	**1**	**D**	**D**		
12	**Forward Implication: SC_{G7}**			**1**				**0**				**1**	
	TC(6)	**1**	**1**	**1**	**1**	**0**	**D'**	**0**	**1**	**D**	**D**	**1**	
13	**Forward Implication: SC_{G8}**								**1**	**D**	**D**	**1**	**D'**
	TC(7)	**1**	**1**	**1**	**1**	**0**	**D'**	**0**	**1**	**D**	**D**	**1**	**D'**
14	**No justification needed. Test generated.**	**1**	**1**	**1**	**1**	**0**	**D'**	**0**	**1**	**D**	**D**	**1**	**D'**

The D-Algorithm

- **The D-algebra**
- **A D-algorithm example**
- **Types of cubes**
- **Components of the D-algorithm**
- **Flowchart of the D-algorithm**
- **Another example**
- Plus & Minus of D-Algorithm

Plus/Minus of D-Algorithm

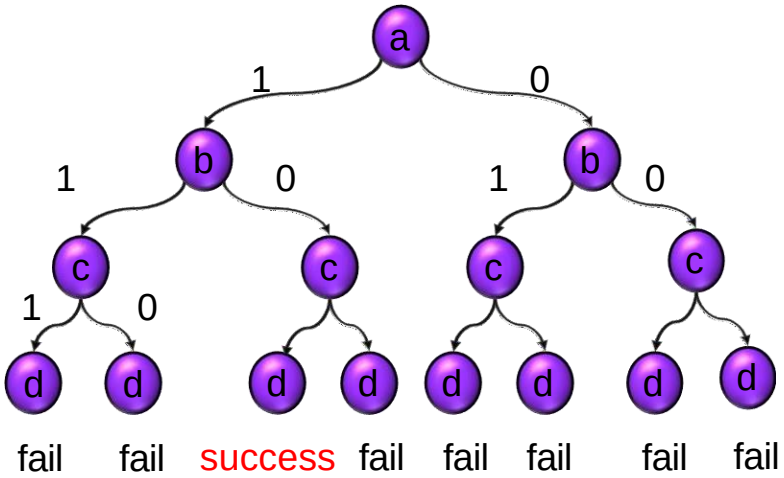
+ **D algorithm is *complete ATPG***
  - **Guarantee to generate a pattern for a testable fault**
- **Large search space**
  - **Assignment of values is allowed for internal signals**
  - **Backtracking could occur at each gate**
  - **Very large search space**

FFT

- **Q1. Why we do justification at the end of D-algorithm**
  - **Why not immediately after implication?**
- **Q2. Please explain why D-algorithm is complete ATPG**

Combinational ATPG

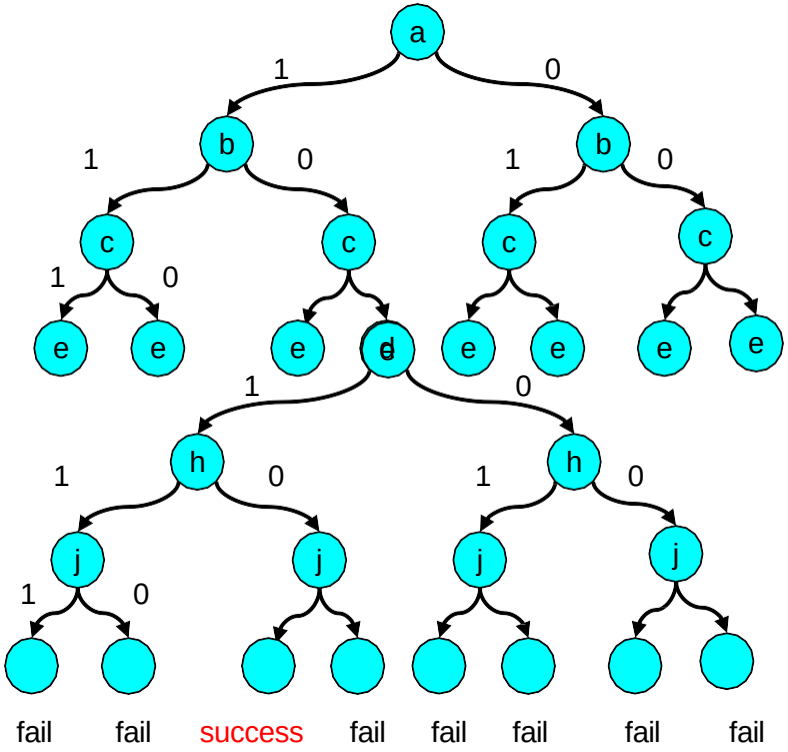
- **Deterministic Test Pattern Generation**
  - Boolean difference approach*
  - Path sensitization method**
  - D-Algorithm [Roth 1966] **
  - **PODEM [Goel 1981]****
    - **Idea**
    - **Heuristics**
    - **Algorithms**
    - **Summary**
  - FAN [Fujiwara 1983]**
  - SAT-based [Larrabee 1992]*

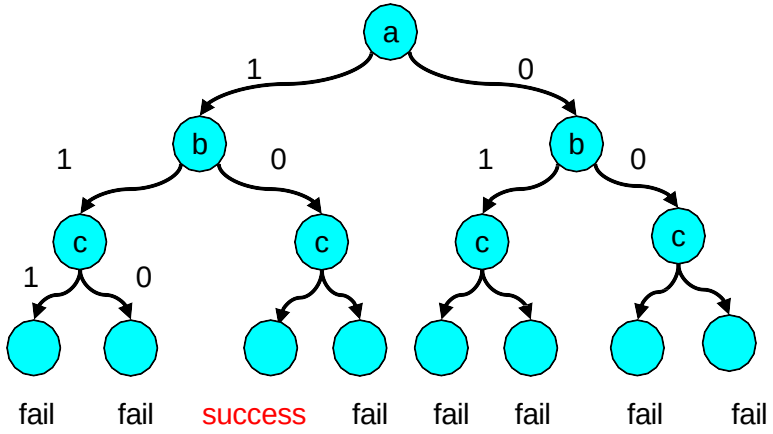


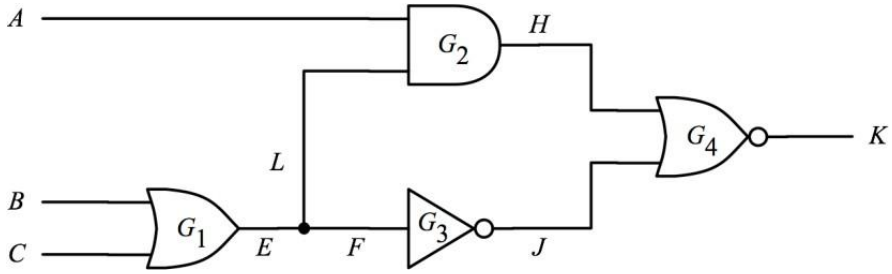
*Boolean-based methods
**path-based methods

$2^6 \rightarrow 2^3$

- **D-alg. makes decision at intern nodes**
  - **6 internal nodes**
  - **2^6=64 decisions! large**
- **PODEM makes decision at PI**
  - **3 PI**
  - **2^3=8 decisions, smaller**





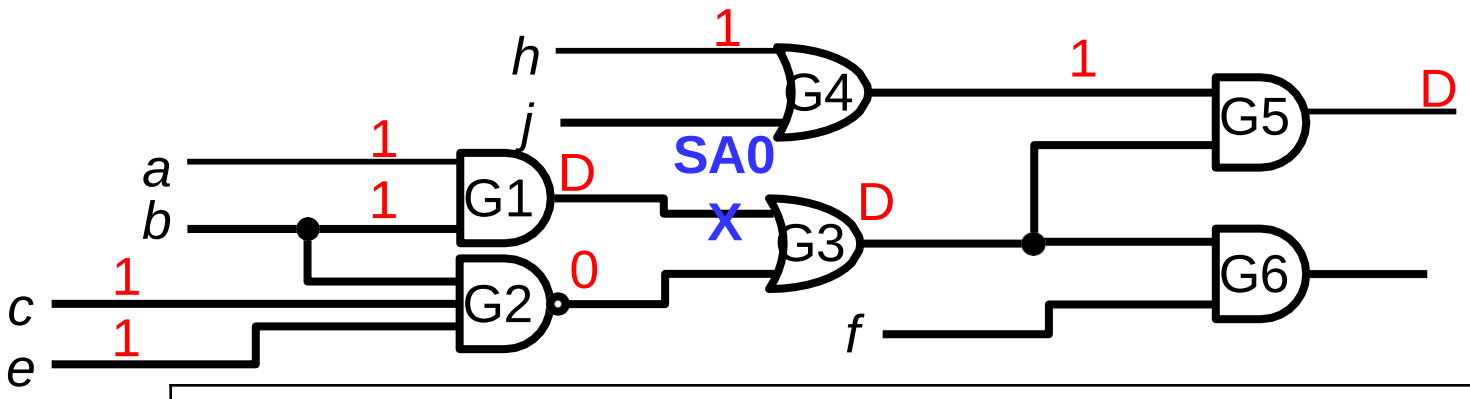


PODEM [Goel 1981]

- ***Path Oriented Decision Making*** **(PODEM)**

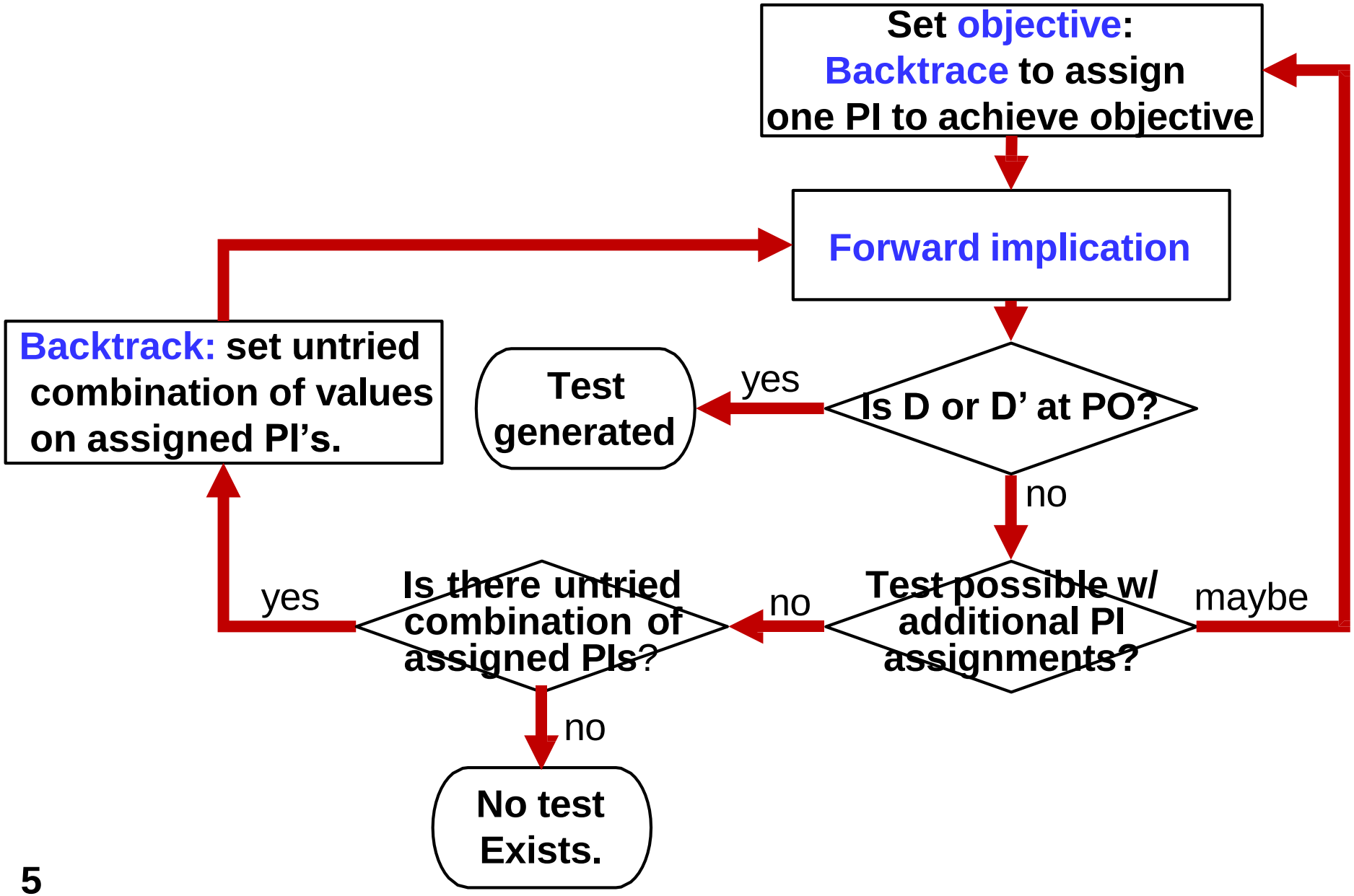
- **IDEAS:**
- **1. Only allow assignments to *PI only***
  - **Doesn't assign internal nodes**
  - **Greatly reduces search tree**
- **2. Assigned PI are then forward implication**
  - **No justification needed (Why ? FFT)**
- **3. Flip last PI assignment when two conditions:**
  - **A. Fault not activated**
  - **B. No propagation path to any output**

PODEM Example



Initial objective: G_1 =1
Backtrace to PI: *b* = 1. simulation, objective not achieved
Backtrace to PI: *a* = 1. simulation, objective achieved
Objective: G2 = 0 (propagate through G3)
Backtrace to PI: *C* = 1. simulation, objective not achieved
Backtrace to PI: *e* = 1. simulation, objective achieved
Objective: G4 = 1 (we choose to propagate through G5)
Backtrace to PI: h= 1. objective achieved.
Test Generated: abcehjf=11111XX

Flowchart of PODEM



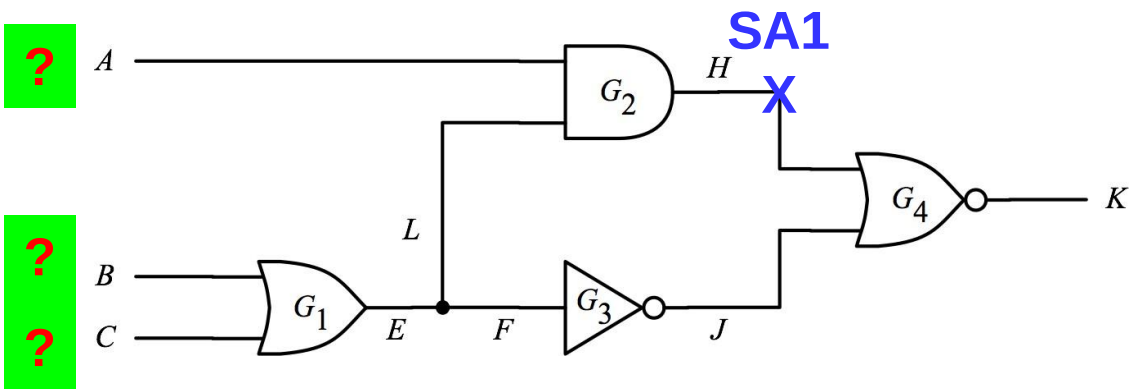
Components of PODEM

- Determine an *objective*
  - If D/D' has not appeared at the fault site,
    - set objective to activate fault
  - If D/D' has appeared
    - set objective to propagate fault effect
- Given objective, determine PI value
  - *Backtrace* to PI
  - *Backtrack* if conflict occurs

- NOTE: *Backtrace* is different from *backtrack*
  - Backtrace goes back to primary inputs of a certain signal
    - In netlist
  - Backtrack goes back to last decision
    - in decision tree

Quiz

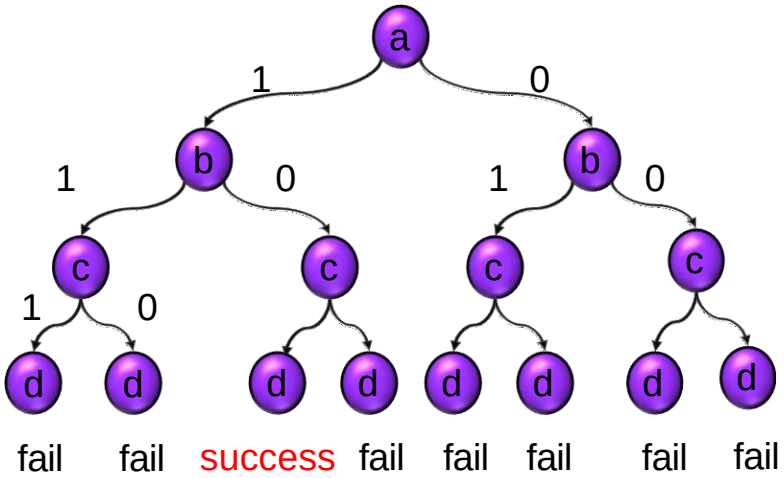
Q: Use PODEM to generate a test for H SA 1 fault please mark your
1. objective,
2. backtrace



Totally 2 objectives, 2 backtraces

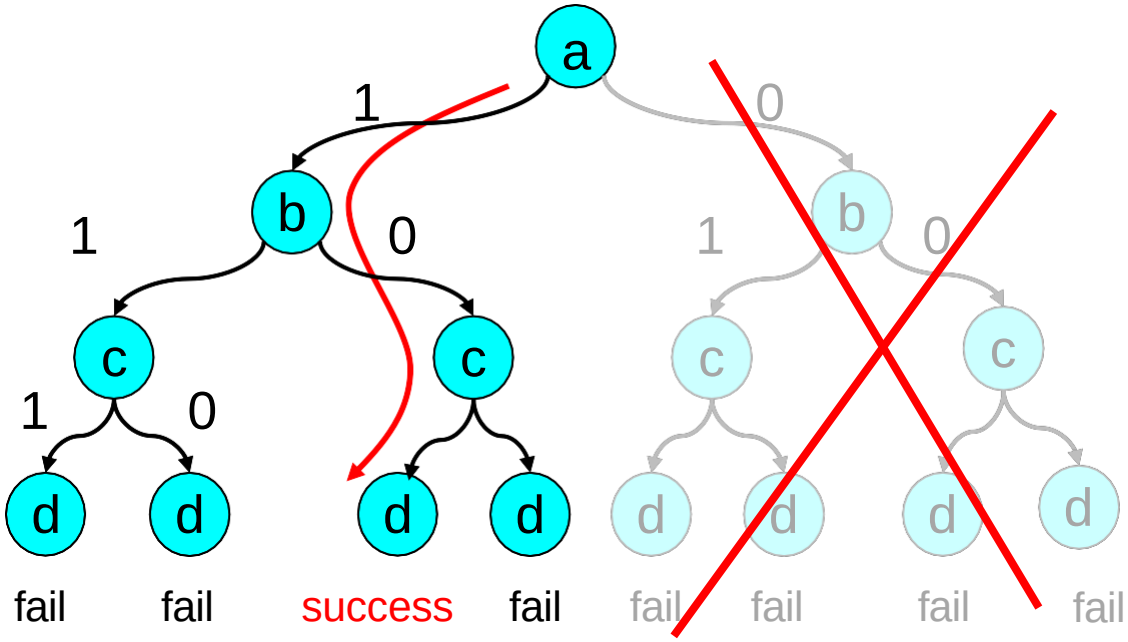
PODEM

- **PODEM [Goel 1981]**
  - **Idea**
  - **Heuristics**
  - **Algorithms**
  - **Summary**



ATPG Decision Tree

- Need smart heuristics to speed up
  - 1. Prune impossible sub-trees ASAP
  - 2. Find good assignments ASAP
- Heuristics are experience-based rules that help correct decision
  - Note: Heuristics Do NOT guaranteed to be correct all the time



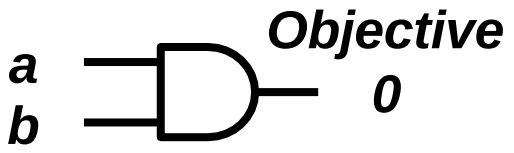
Questions to Be Answered

- **PODEM proposed three heuristics to answer three questions:**
  - **Q1: What path to *backtrace*?**
  - **Q2: What input value to assign?**
  - **Q3: What path to propagate D (D') to PO?**

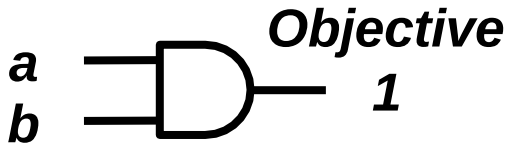
Good Heuristic = Simple and Effective (most of the time)

Q1: What Path to Backtrace?

- ***Decision Gate***:
  - Only one input can control gate output to objective value
    - OR/NAND with output objective =1
    - AND/NOR with output objective =0
  - choose ***easiest*** gate input



- ***Imply Gate***:
  - One input can't control gate output to objective value
    - OR/NAND with output objective =0
    - AND/NOR with output objective =1
  - Choose ***hardest*** gate input

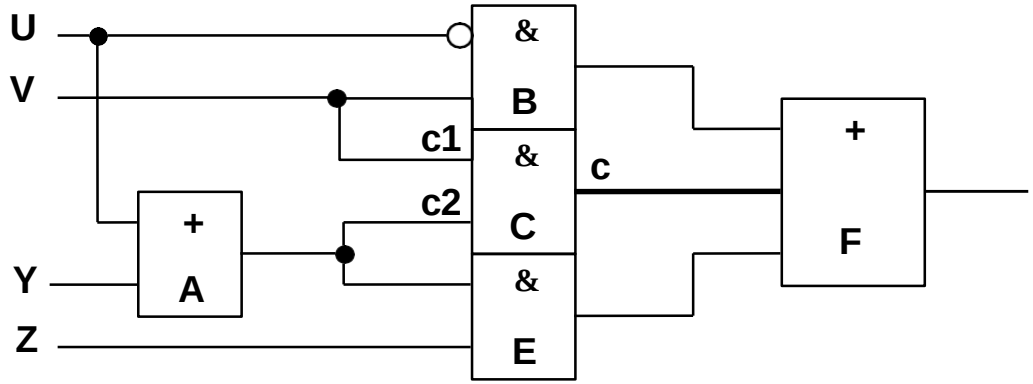


- Why? find out if test exists or not as soon as possible

Heuristic#1: Make Correct Backtrace ASAP

Example

- If objective is c=0; backtrace c1, then V=0 (decision gate)
- If objective is c=1; backtrace c2, then U=1 or Y=1 (imply gate)

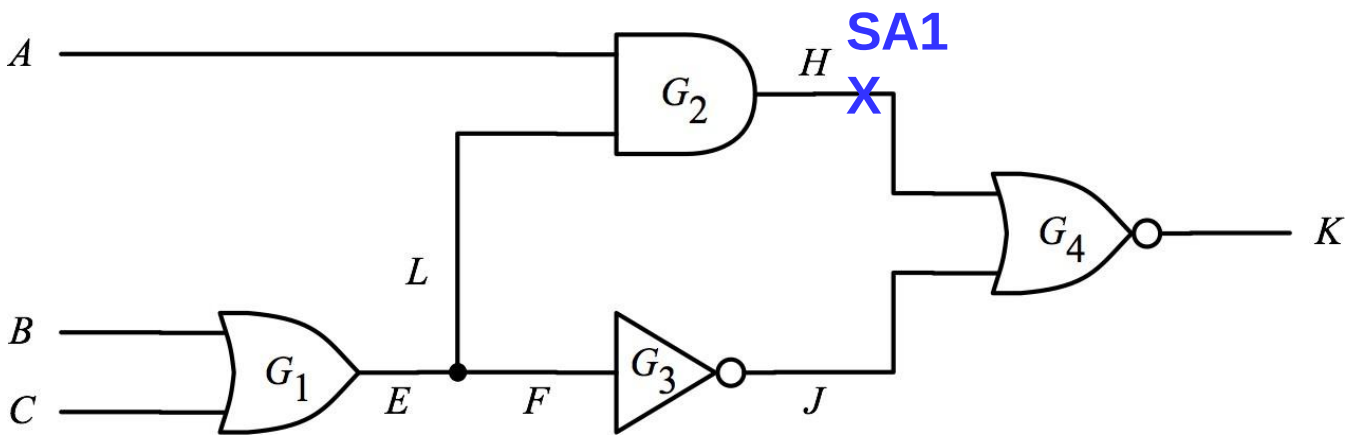


- Q: how do you know c1 is easy and c2 is hard?
  - Level of c1 smaller than level of c2
  - (Use other controllability measure, like SCOAP, also fine)

Quiz

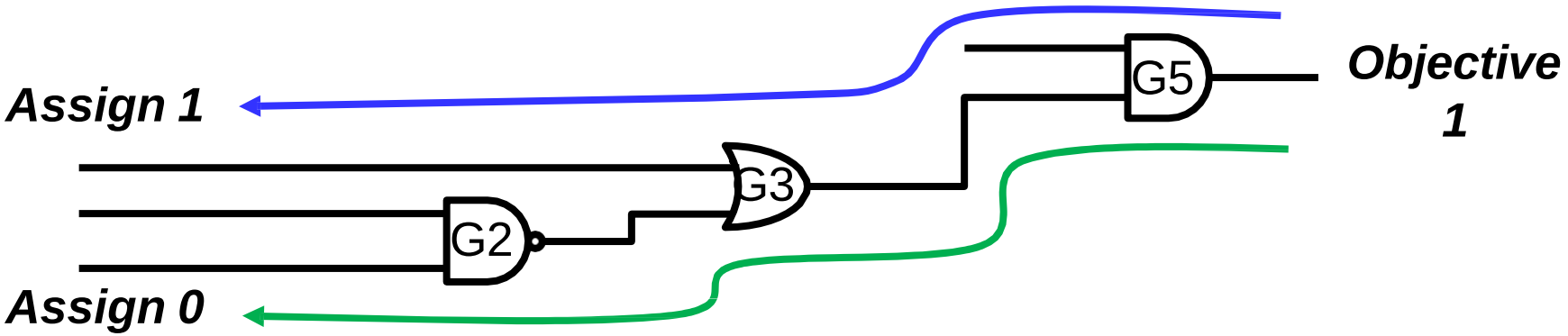
Q: Given SCOAP, generate a test for H SA1 fault.
A1: Follow heuristic
A2: Do not follow heuristic

	A	B	C	E	F	L	H	J	K
CC^0	1	1	1	3	3	3	2	3	5
CC^1	1	1	1	2	2	2	4	4	6
CO	7	6	6	4	4	6	4	3	0



Q2: What Input Values to Assign?

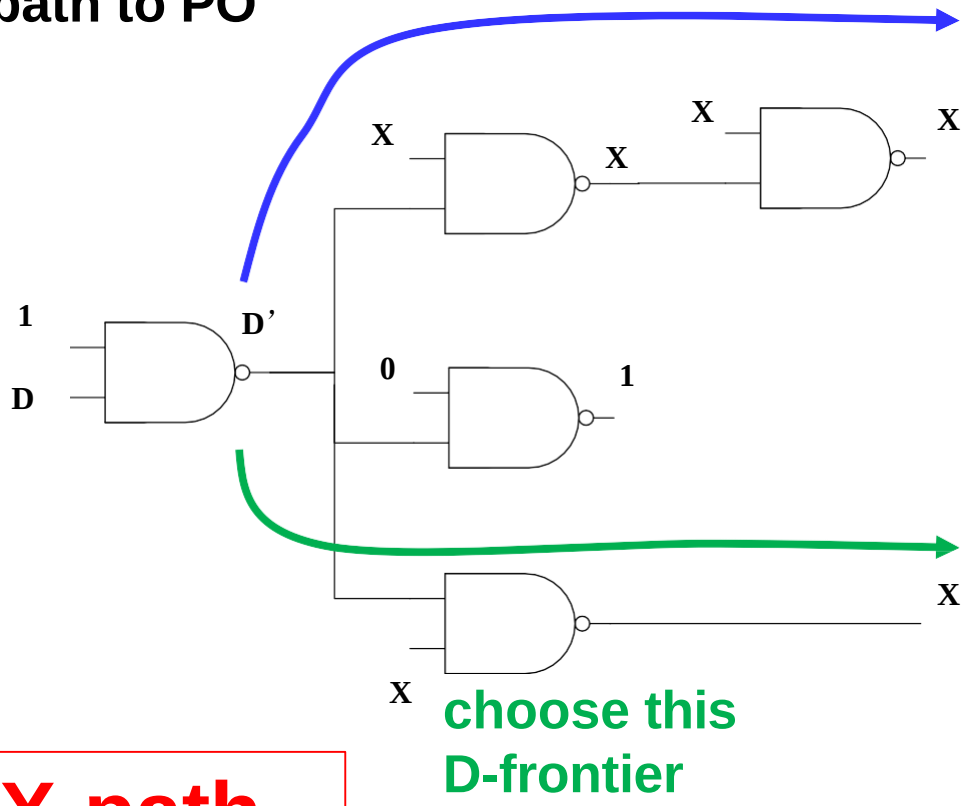
- If even inversion parity path
  - Assign same value as objective
- If odd inversion parity path
  - Assign opposite value to objective
- Why? This assignment is most likely to be correct



Heuristic#2: Inversion Parity for Assignment

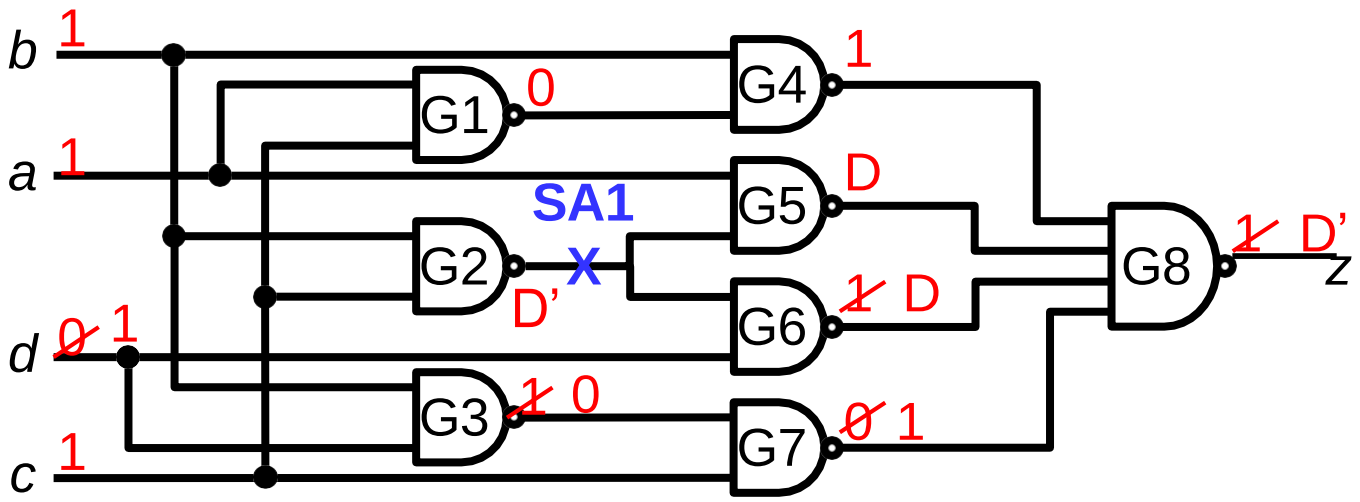
Q3: What Path to Propagate ?

- All gate output of the chosen path must have X values
  - Called *X-PATH*
- If more than one X-path to choose,
  - chose shortest X-path to PO
- If X-path disappear,
  - *backtrack*



Heuristic#3: X-path

PODEM Example w/ Backtrack

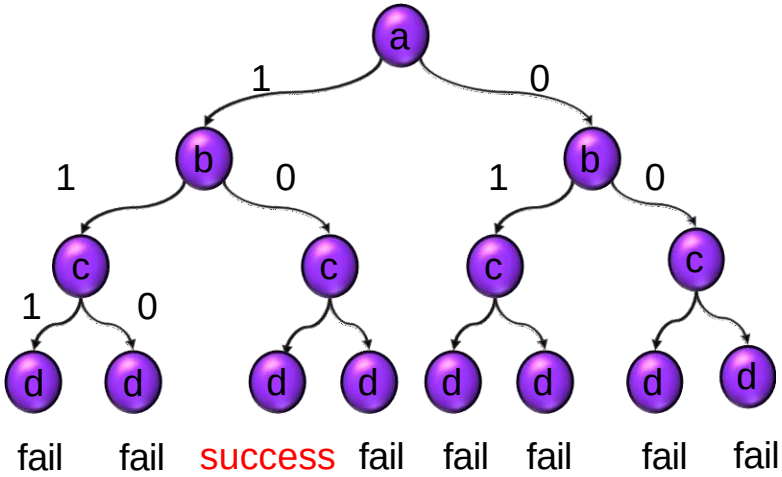


Initial objective: (G2, 0)
Backtrace to PI: *b* = 1
Initial objective: (G2, 0)
Backtrace to PI: *c* = 1
Implication:G2 = D'
Choose shortest X-path {G5} Objective a = 1 Assign a=1
Implication: G1 = 0,G4 = 1,G5 = D

Try propagate through G8.
objective: (G6,1)
Backtrace to PI: *d* = 0
Implication:G3 = 1,G7 = 0,G8 = 1
X-path disappear! Backtrack to most recent PI assignment: *d* = 0 → *d* = 1
Implication:G3 = 0,G6 = D,G8 = D'
Test generated!

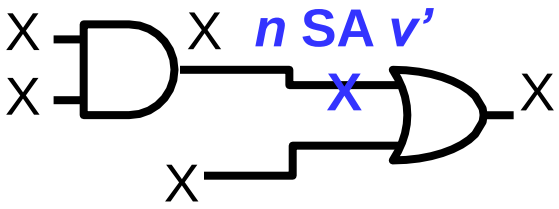
PODEM

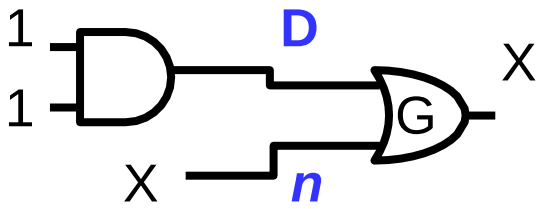
- **PODEM [Goel 1981]**
  - **Idea**
  - **Heuristics**
  - **Algorithms**
  - **Summary**



Objective

- **Pick an objective: (1) fault activation (2) propagate fault effect**





Objective (*n, v*) // target fault: net *n stuck-at v'*
// if fault has not been activated
1. **if (net *n* is unknown)**
2. **return (*n, v*);**

// else, propagate fault effect
3. **select a gate *G* from D-frontier on shortest X-path**
4. **select an unassigned input *n* of *G***
5. **if (gate *G* has non-controlling value)**
6. **v = non-controlling value of *G*; // AND v=1; OR, v=0**
7. **return (*n, v*); // *n* is objective net; *v* is objective value**

Backtrace

- Translates objective to PI assignment
- *Depth-first search*: recursively calls itself until hits PI

```
Backtrace (n, v_s)  /* n is objective net; v_s is objective y; */
1.  v = v_s;
2.  while (n is gate output)
3.      if (n is NAND or INVERTER or NOR) v = v';   // inversion parity
4.      if (objective requires setting all inputs)  // imply gate
5.          a = hardest gate input that is still X;
6.      else // decision gate
7.          a = easiest gate input that is still X;
8.      n = a;
9.      (n, v) = Backtrace (n, v);  // recursive call
       // out of while loop, n is now PI
10.  return (n, v)  // assign PI n to value v
```

PODEM

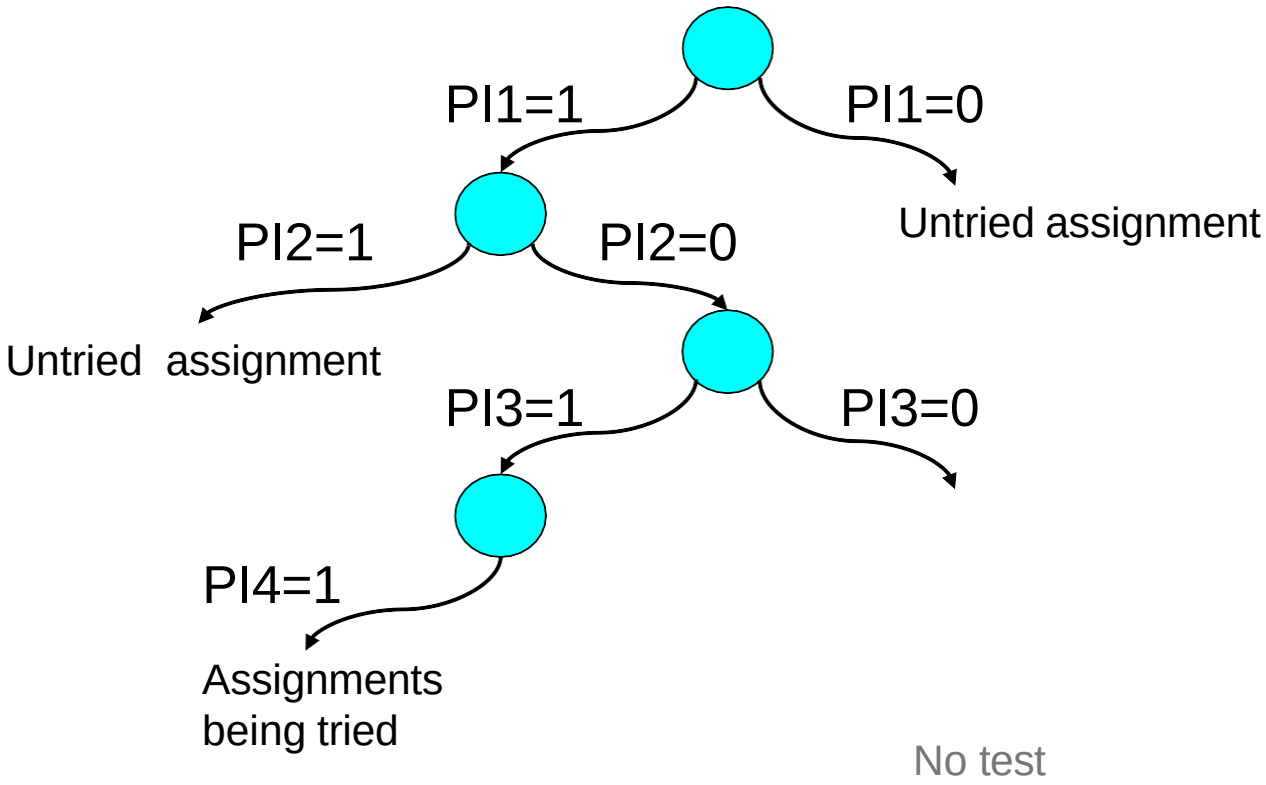
- ***Branch and bound*** **search algorithm**

```
PODEM (fault , v_fault)
1.   if (D or D' at PO) return (SUCCESS)
2.   if (test impossible) return (FAILURE)
3.   else  (n, v_s) = Objective (fault, v_fault);
4.         (pi, v) = Backtrace (n, v_s);
5.         Imply (pi, v); // assign pi, forward implication
6.         if (PODEM (fault, v_fault) == SUCCESS) return (SUCCESS);
           // backtrack
7.         Imply (pi, v');
8.         if (PODEM (fault, v_fault) ==  SUCCESS) return (SUCCESS);
9.         Imply (pi, "X");  // release PI as unknown
10.        return (FAILURE); // this node is pruned
11. end;
```

test impossible for 2 reasons:
(1) target fault cannot be activated
(2) X-path disappear

Decision Tree of PODEM (1)

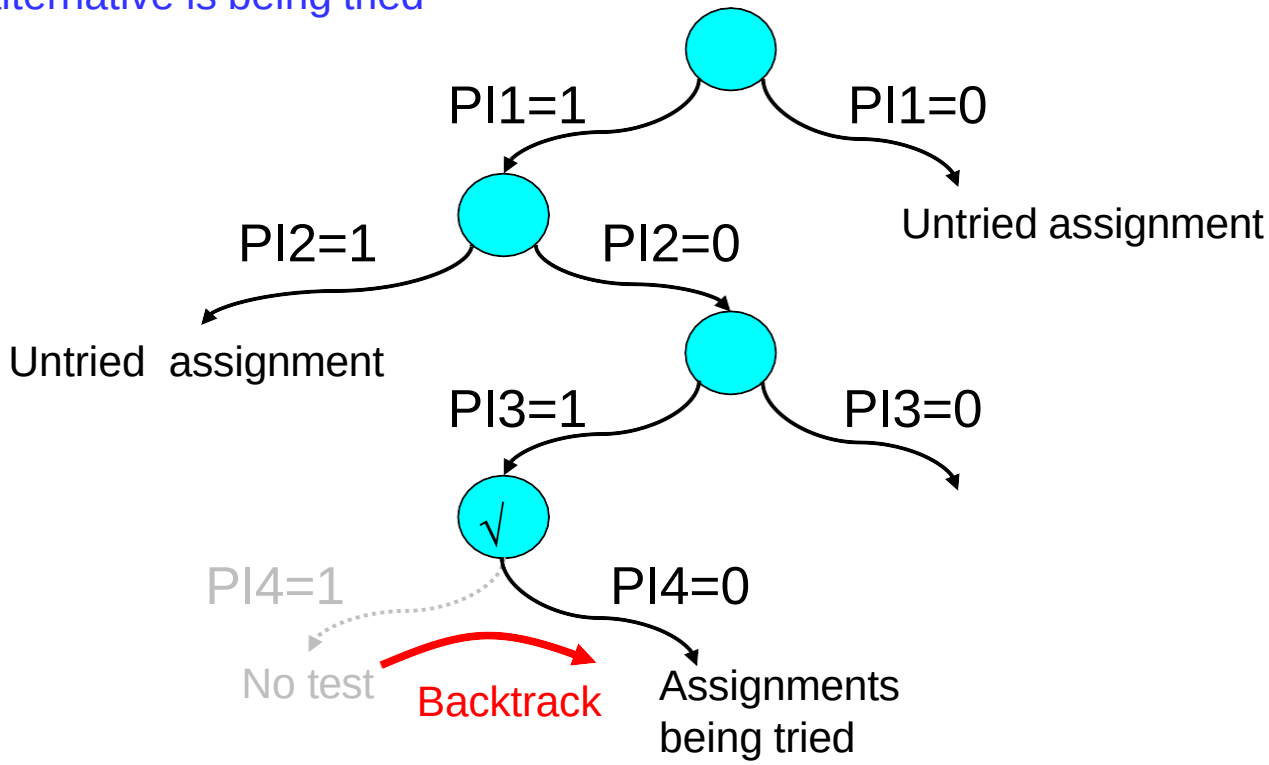
- ***Branch and bound*** **search algorithm**
  - **Each decision is a PI**



Decision Tree of PODEM (2)

- ***Branch and bound*** **search algorithm**
- ***Chronological backtrak:*** **[DPLL satisfiability 1962]**
  - **Flip *last* PI that has not been tried**

√ **:** initial assignment rejected
alternative is being tried

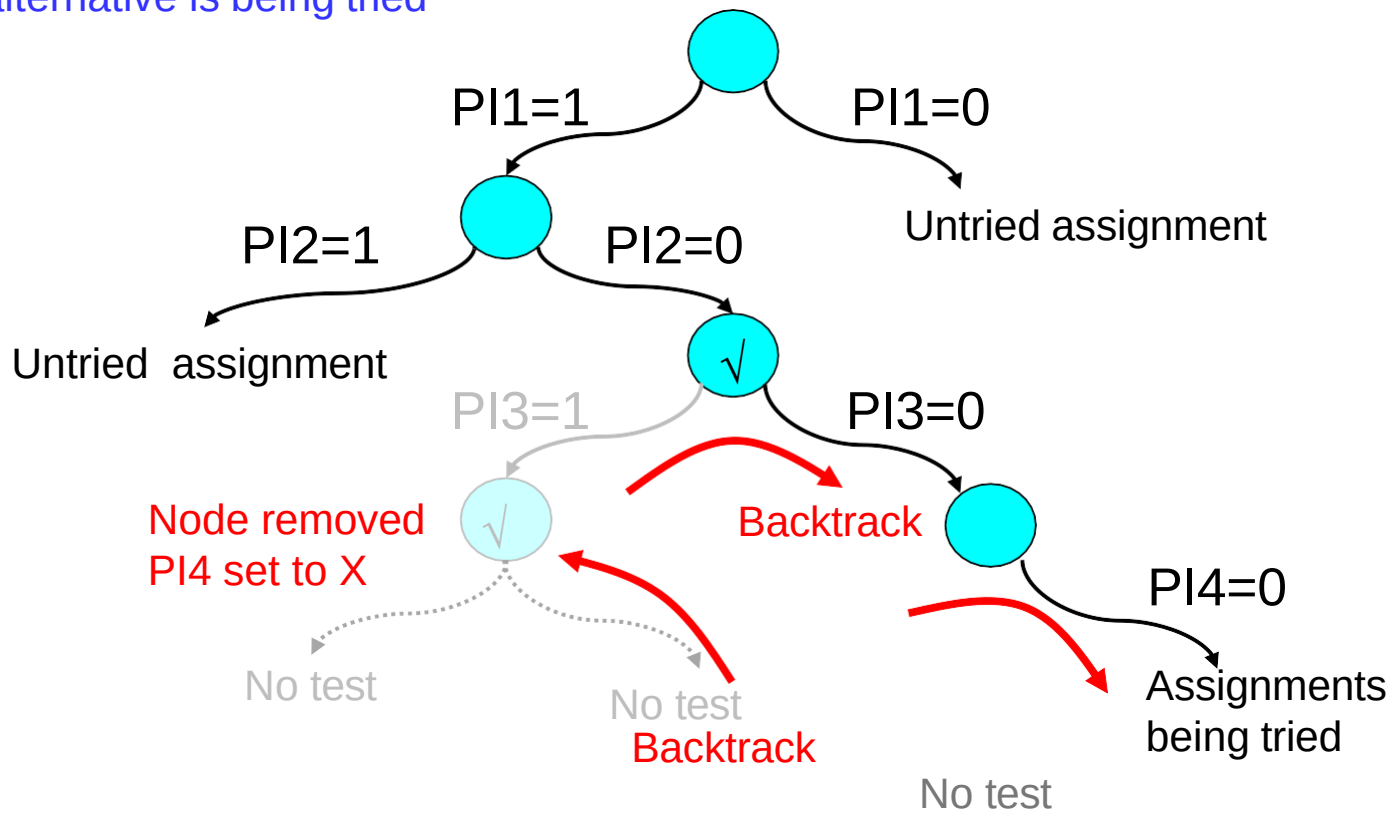


No test

Decision Tree of PODEM (3)

- ***Branch and bound*** **search algorithm**
  - **A node is removed when both alternatives have been tried**
  - **Prune PI3=1 subtree**

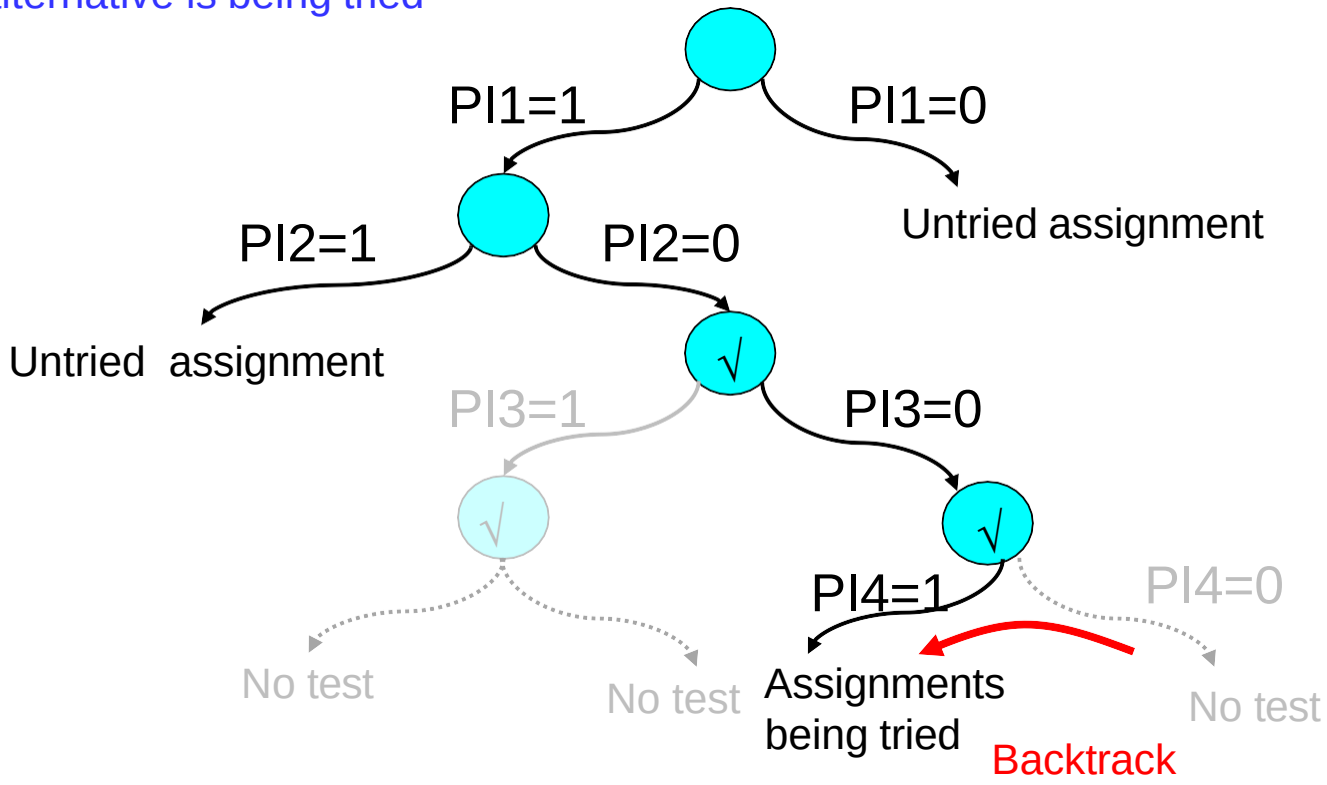
√ **:** initial assignment rejected
alternative is being tried



Decision Tree of PODEM (4)

- ***Branch and bound*** **search algorithm**

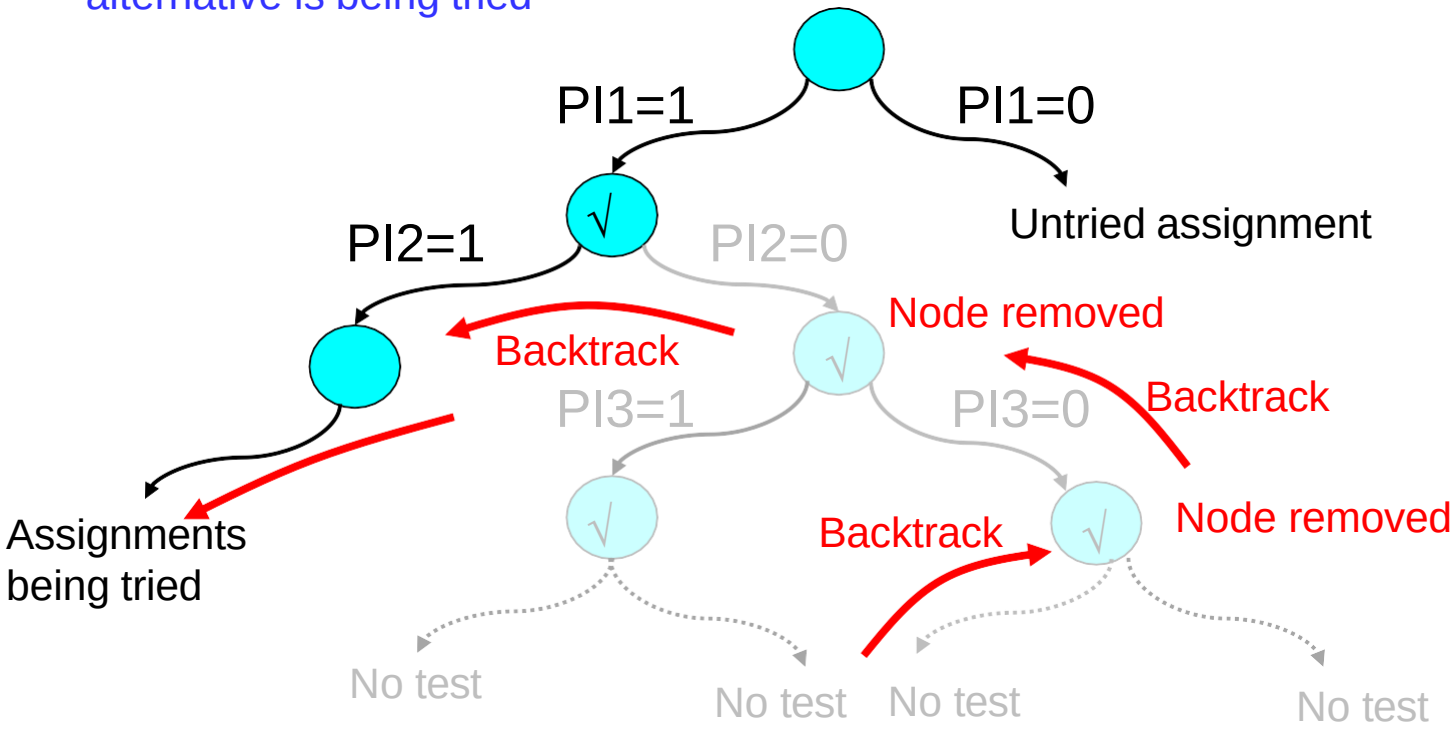
√ **:** initial assignment rejected
alternative is being tried



Decision Tree of PODEM (5)

- ***Branch and bound*** **search algorithm**
  - **Prune PI3=0 subtree**
  - **Prune PI2=0 subtree**

√ **:** initial assignment rejected
alternative is being tried



PODEM: Summary

- **PODEM [Goel 1981]**
  - **Idea:**
    - **assign PI, not internal nodes**
    - **forward implication only, no justification**
  - **Heuristics**
    - **(1) backtrace easy/hard input for decision/imply gate**
    - **(2) keep path inversion parity**
    - **(3) propagate along X-path**
  - **Algorithms**
    - **branch and bound search**

FFT

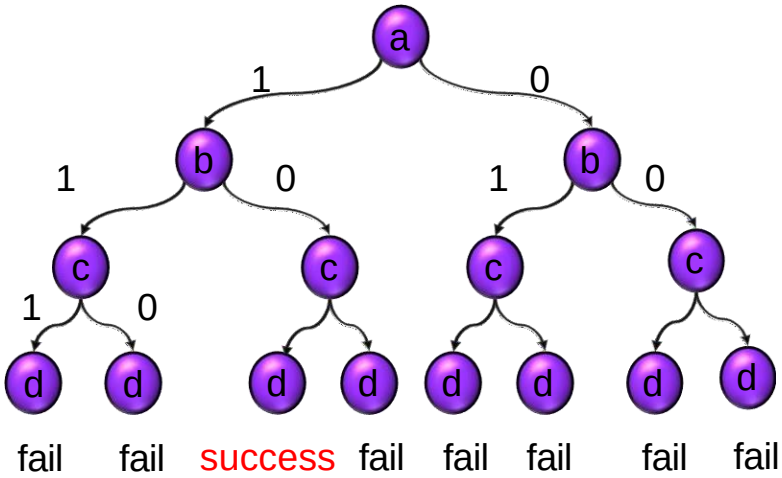
- **Q1: Is PODEM a complete ATPG algorithm?**
- **Q2: Why does PODEM have no justification?**
- **Q3: Please give examples where heuristic #1/#2 gives wrong guess**

Combinational ATPG

- Introduction
- **Deterministic Test Pattern Generation**
  - Boolean difference approach*
  - Path sensitization method**
  - D-Algorithm [Roth 1966] **
  - PODEM [Goel 1981] **
  - **FAN [Fujiwara 1983]****
  - SAT-based [Larrabee 1992] *
- Acceleration Techniques
- Concluding Remarks

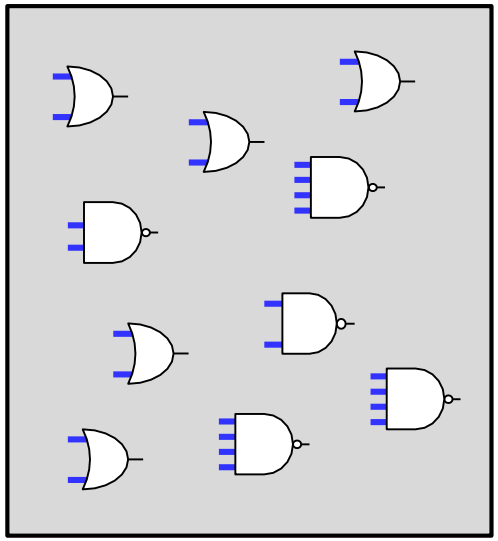
*Boolean-based methods

**path-based methods

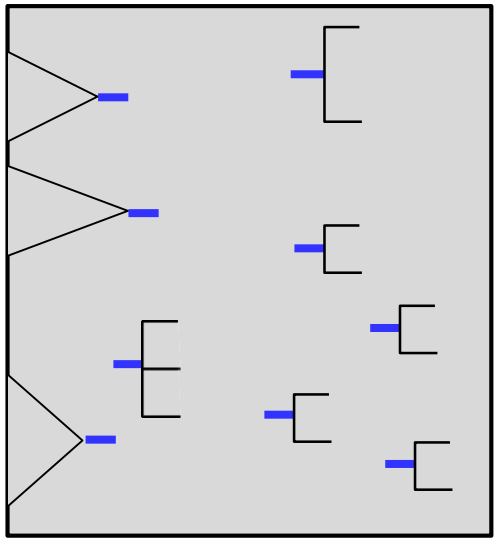


D→PODEM →FAN

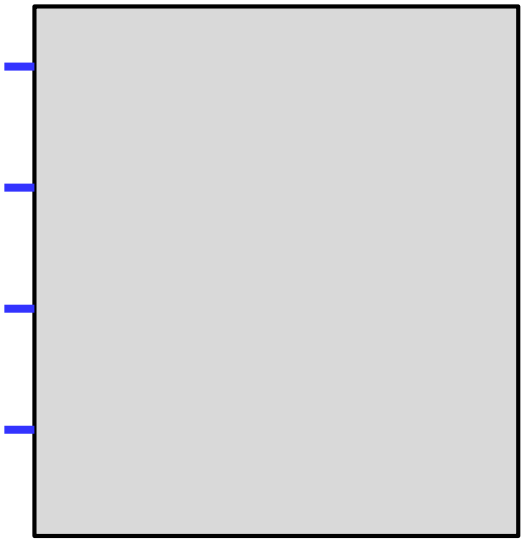
- D-algorithm decision at internal nodes → too many decisions, slow!
- PODEM decision at PI → too little information, mistake-prone!
- FAN decision at head lines and fanout stems → good trade-off



D-alg.
too many decisions



FAN
good trade-off



PODEM
too little information

— decision points

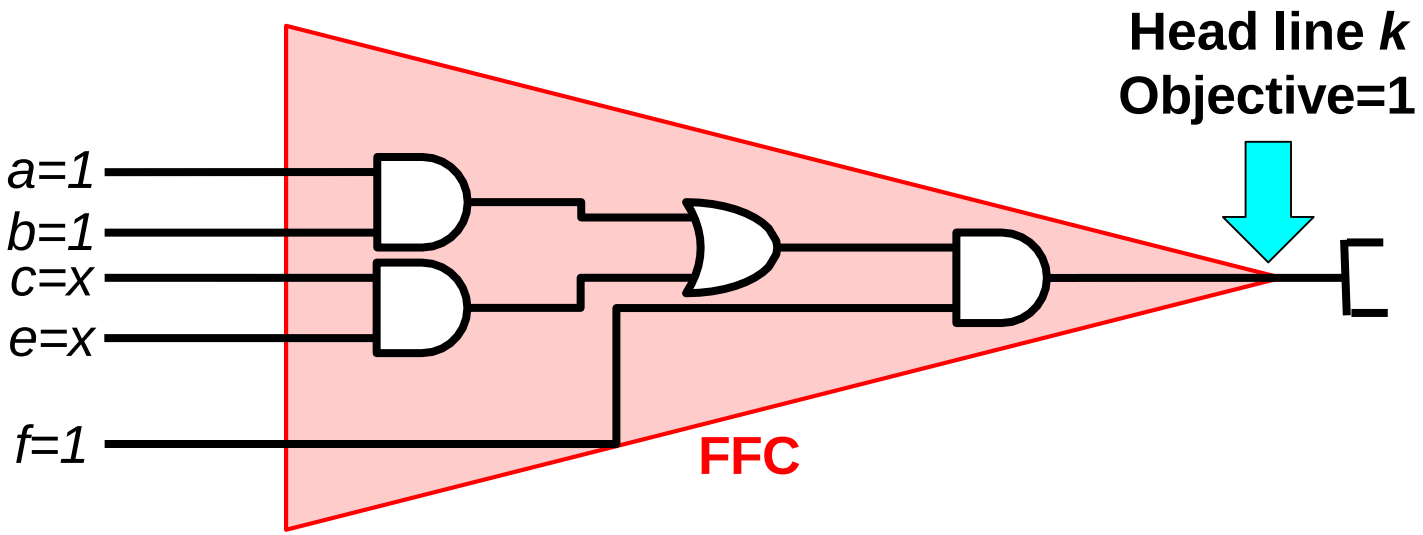
FAN [Fujiwara 1983]

- *FANout-oriented test generation*
- Four improvements over PODEM
  - #1. Make decision at head lines or fanout stem
  - #2. Forward/backward Implications
  - #3. Unique sensitization
  - #4. Multiple backtraces

https://insights.ubuntu.com

Justify Head Line Is Easy

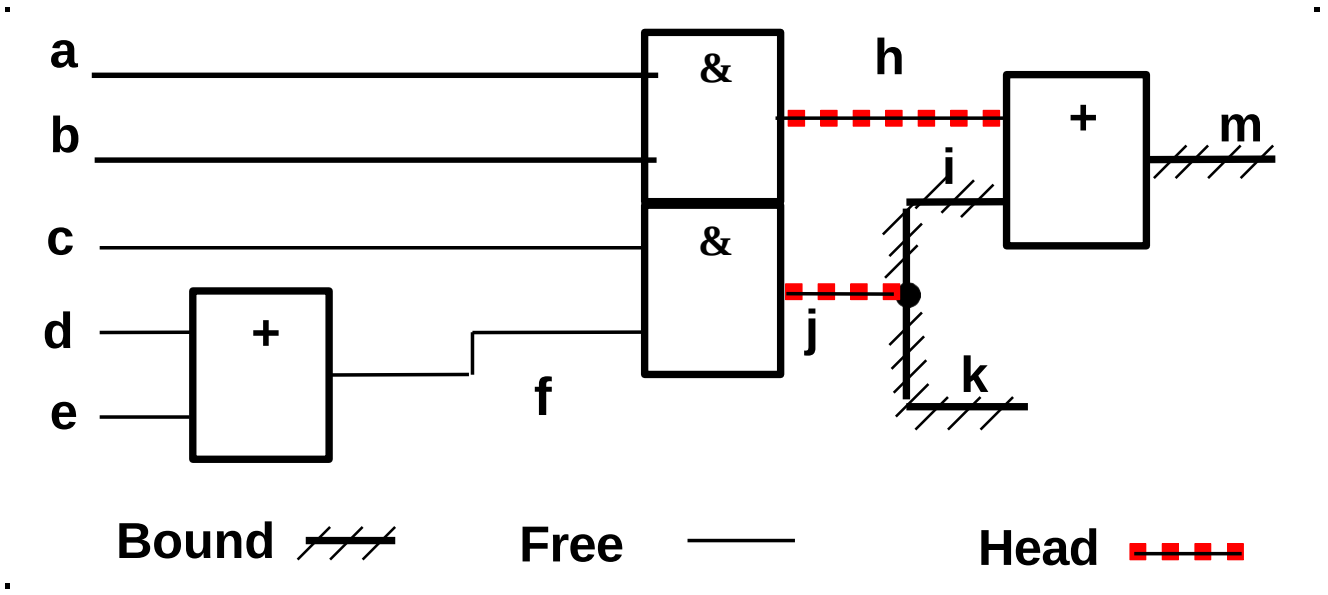
- **Justification inside FFC (*fanout-free cone*) is linear time**
  - **Guaranteed to find an answer**
- **Example: *a,b,c,e,f* are PI, *k* is *head line***
  - **objective *k* =1**



Can Make Decision at Head Line
Instead of PI

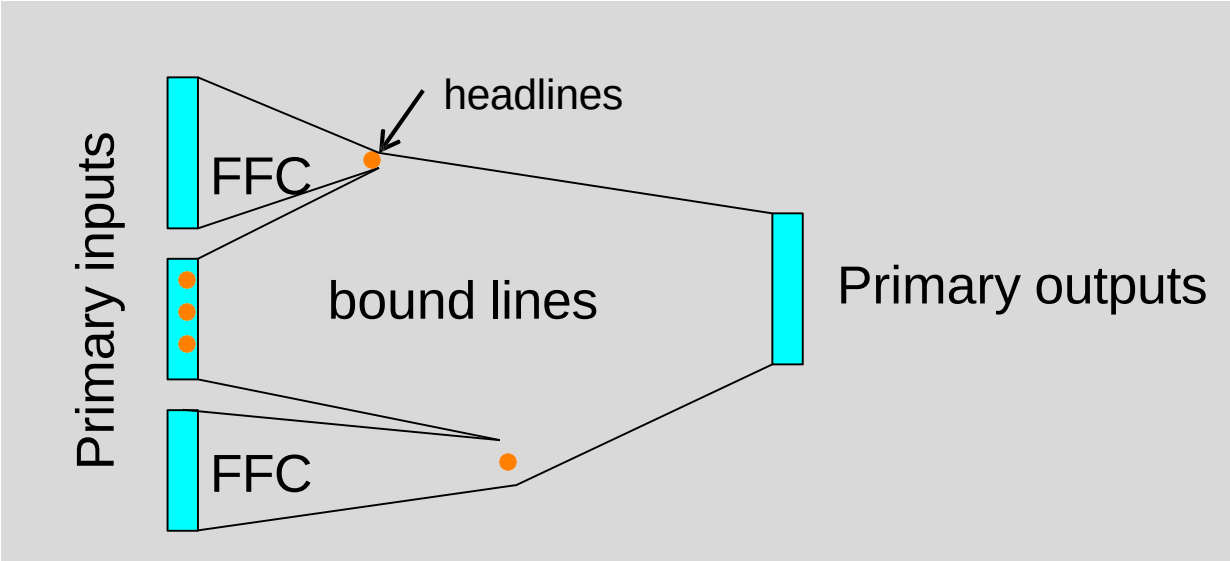
Head Line

- ***Bound line*: line fed directly (*i, k*) or indirectly (*m*) by fanout stem**
- ***Free Line*: line that is not Bound (*a~f, h, j*)**
- ***Head Line*: free line that is either**
  - **Fanout stem (*j*), or**
  - **Input to a gate with bound output (*h*)**



#1. Make Decision at Head Lines

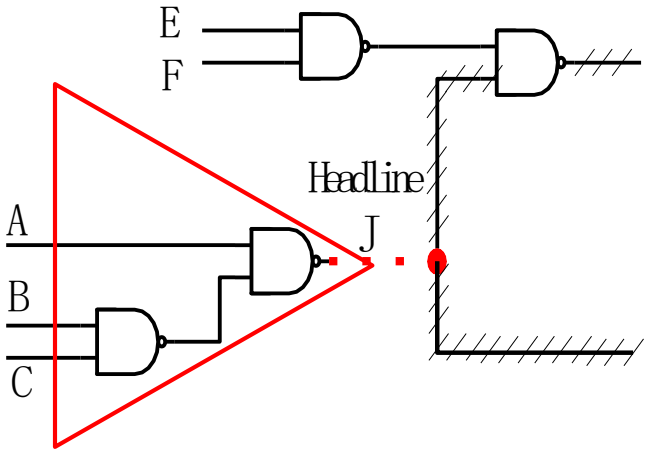
- **FFC can be isolated from rest of circuit by cutting head lines**
- **Assignment of PI's that feed head lines are**
  - **deferred until other objectives have been achieved**
    - **Reduce search space**



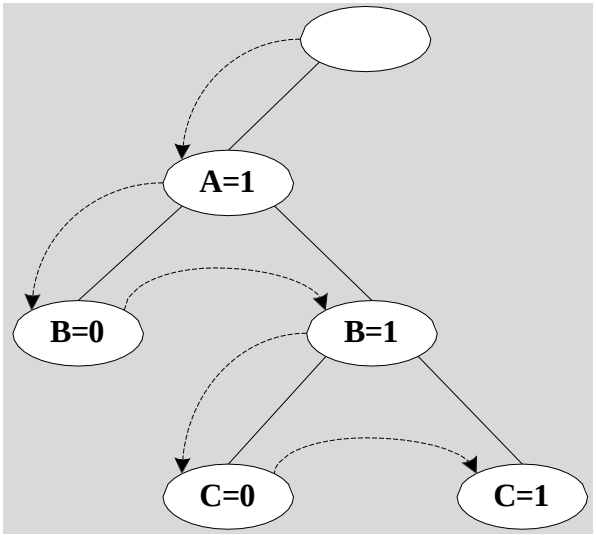
Head Lines Reduces Search Space

Example

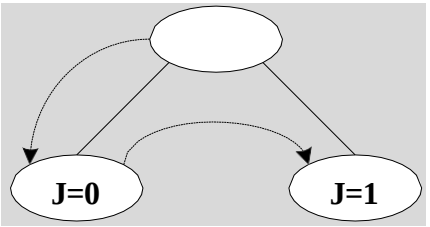
- PODEM decision tree is big
  - A=1,B=0
    - No test, backtrack
  - B=1, C=0
    - No test, backtrack
  - C=1



- FAN decision tree is small
  - J=0
    - No test, backtrack
  - J=1



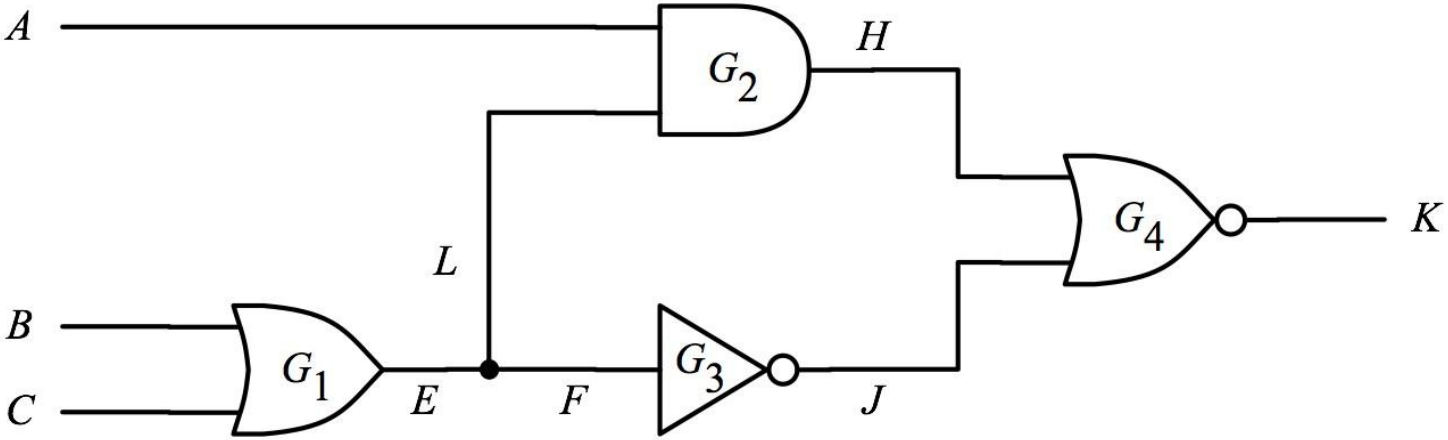
PODEM



FAN

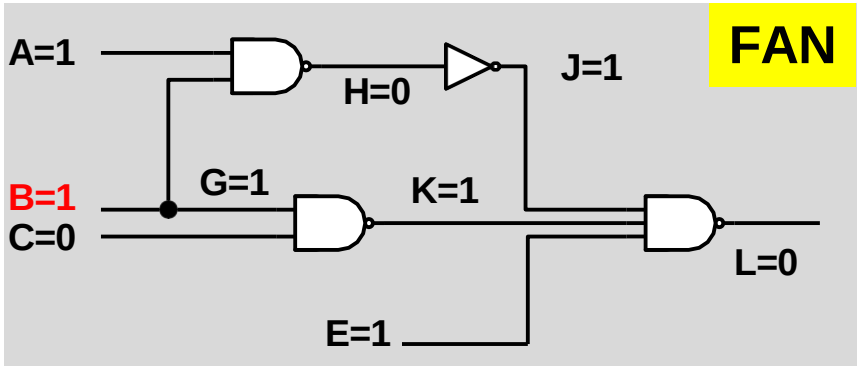
Quiz

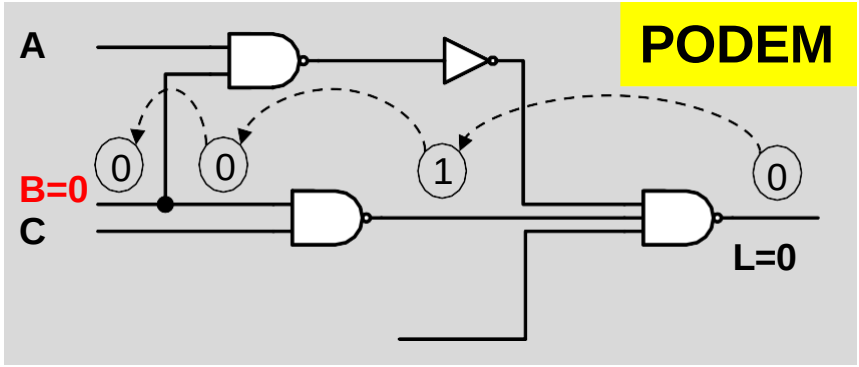
Q: Which are bound lines? free lines? head lines?
A:



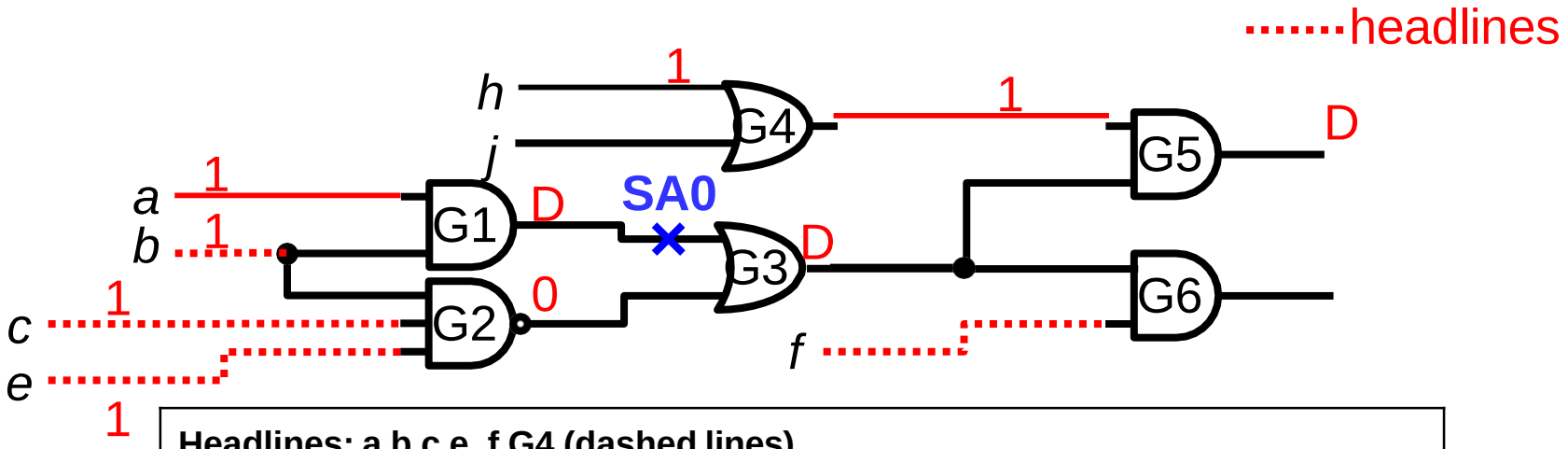
#2. Forward/backward Implication

- PODEM does not assign internal values
  - Only forward implication, no backward implication
- FAN assigns internal values when they are uniquely implied
  - Both forward and backward implication
- Example: *L* SA1 fault
- FAN
  - *Bwd: JKE*=1, *H*=0, *A*=1, *B*=1
  - *Fwd*: *G*=1
  - *Bwd: K=1, C*=0
  - no backtrack needed
- PODEM
  - Backtrace to *B*=0
  - Forward implication
  - Wrong! backtrack





FAN Example



Headlines: a b c e f G4 (dashed lines)
Initial objective: G1 output =1
Implication: assign *a* = 1; *b*=1
Objective: propagate through G3, objective G2=0
Implication: assign *c* = 1; e=1
Propagate through G5, objective G4=1
Assign headline G4 = 1
G5=D, Objective achieved.
Justify head line G4 = 1 → h=1
Test generated abcehjf = 11111xx

Implication

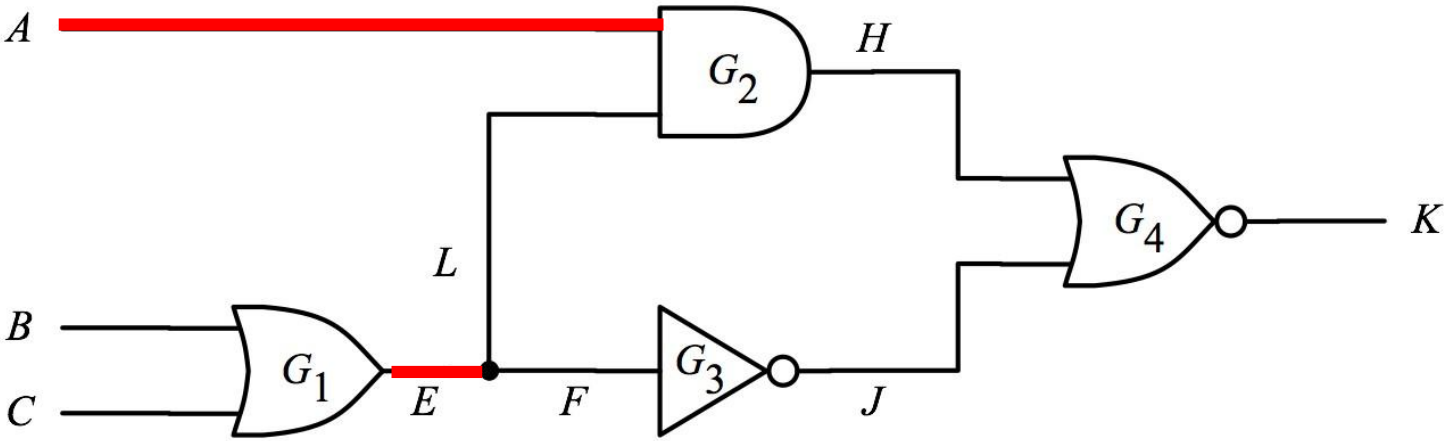
Make decision at head line

Justify head lines at end

Quiz

Q: If we want *K*=1, apply implication to determine head lines *A*=? *E*=?

ANS:

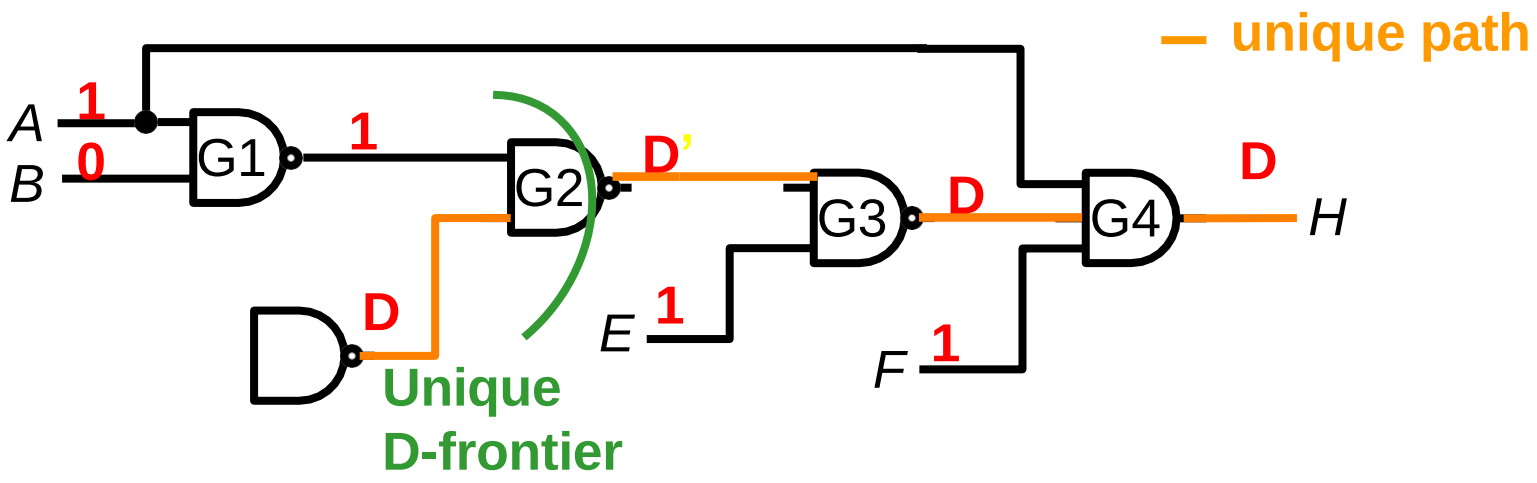


FAN

- **Four improvements over PODEM**
  - #1. Make decision at head lines or fanout stem
  - #2. Forward/backward Implications
  - **#3. Unique sensitization**
  - **#4. Multiple backtraces**

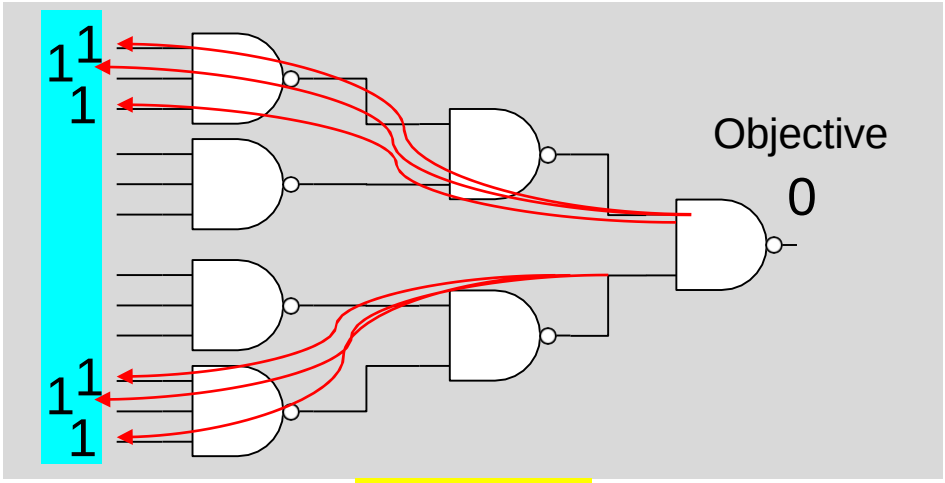
#3. Unique Sensitization

- When there is only one gate in D-frontier
  - if unique path exists, set side inputs to non-controlling values
- Example:
  - FAN
    - *G2* is unique D-frontier, only one path to *H*
    - *G1* = 1, *E* = 1, *F* = 1, *A* = 1 → *B* = 0 → success!
  - PODEM
    - Initial objective: *G1* = 1 →
    - backtrace to *A* = 0 → X-path disappear!

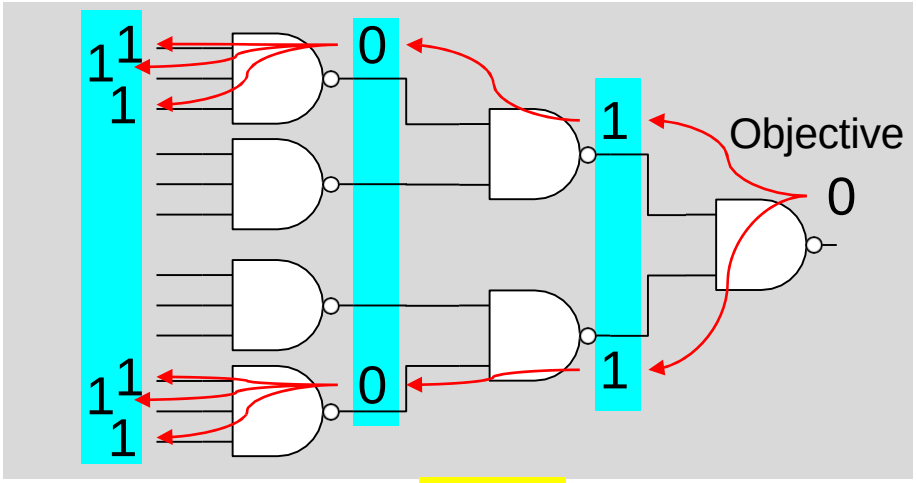


#4. Multiple Backtraces

- PODEM uses depth-first search (DFS)
  - One single backtrace at a time
- FAN uses breadth first search (BFS)
  - Multiple parallel search at a time
- Example
  - PODEM needs 6 backtraces
  - FAN needs only 1 multiple backtrace



PODEM

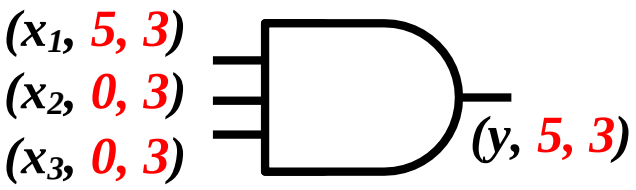


FAN

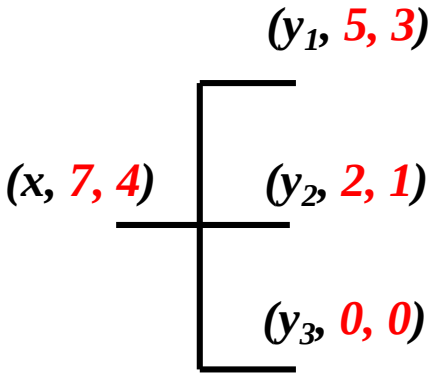
Rules for Multiple Backtraces

- **Objective: (x, n_0, n_1)**
  - **number of backtraced zeros (n_0) and ones (n_1) on signal x**

- **For AND gate**
  - **Easiest unspecified input x_1**
    - $(x_1, n_0, n_1) = (y, n_0, n_1)$
  - **Other inputs x_2, x_3**
    - $(x_2, n_0) = 0$
    - $(x_2, n_1) = (y, n_1)$

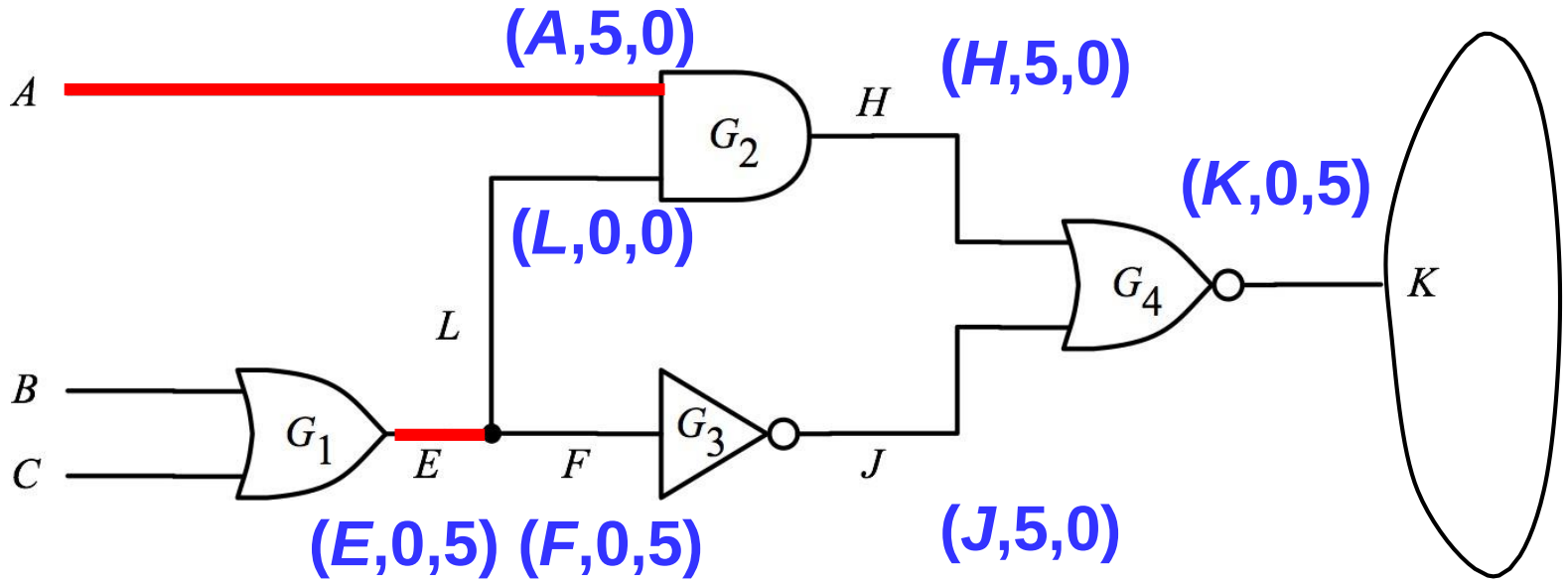


- **For fanout Stem**
  - (x, n_0) = sum of (y_i, n_0)
  - (x, n_1) = sum of (y_i, n_1)



Example

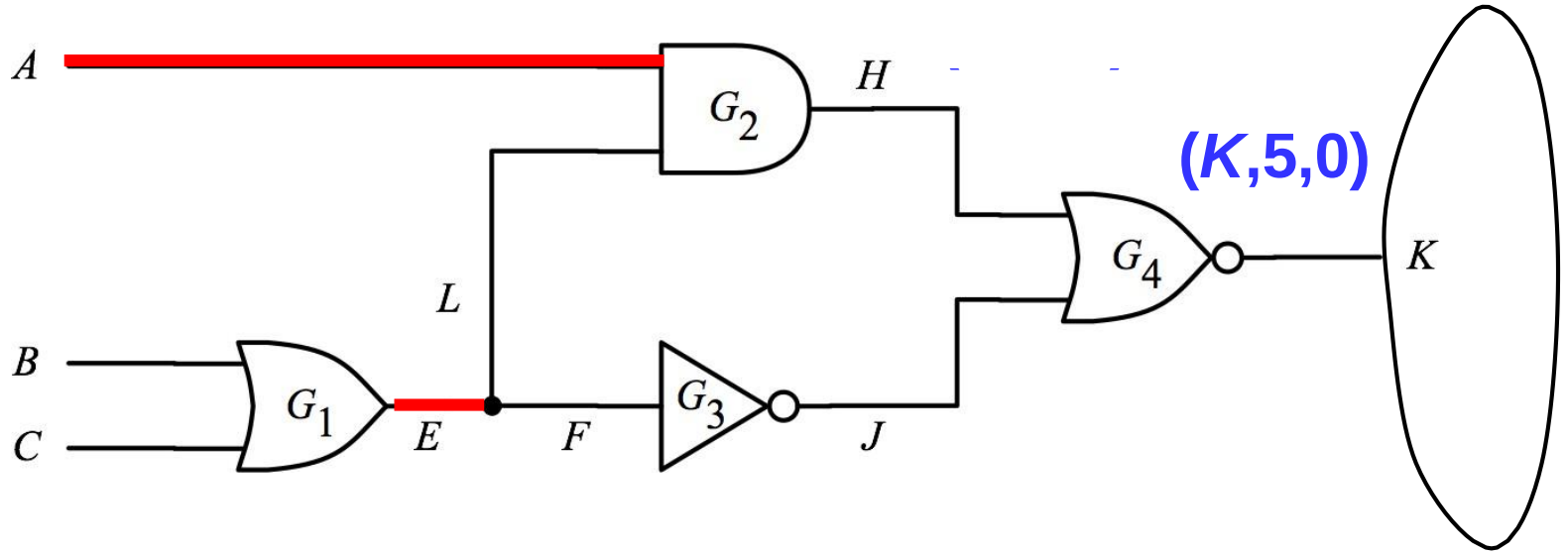
- Starting from (*K*,0,5), multiple backtrace to head lines *A* and *E*
  - So we get two assignments *A*=0, *E*=1



Quiz

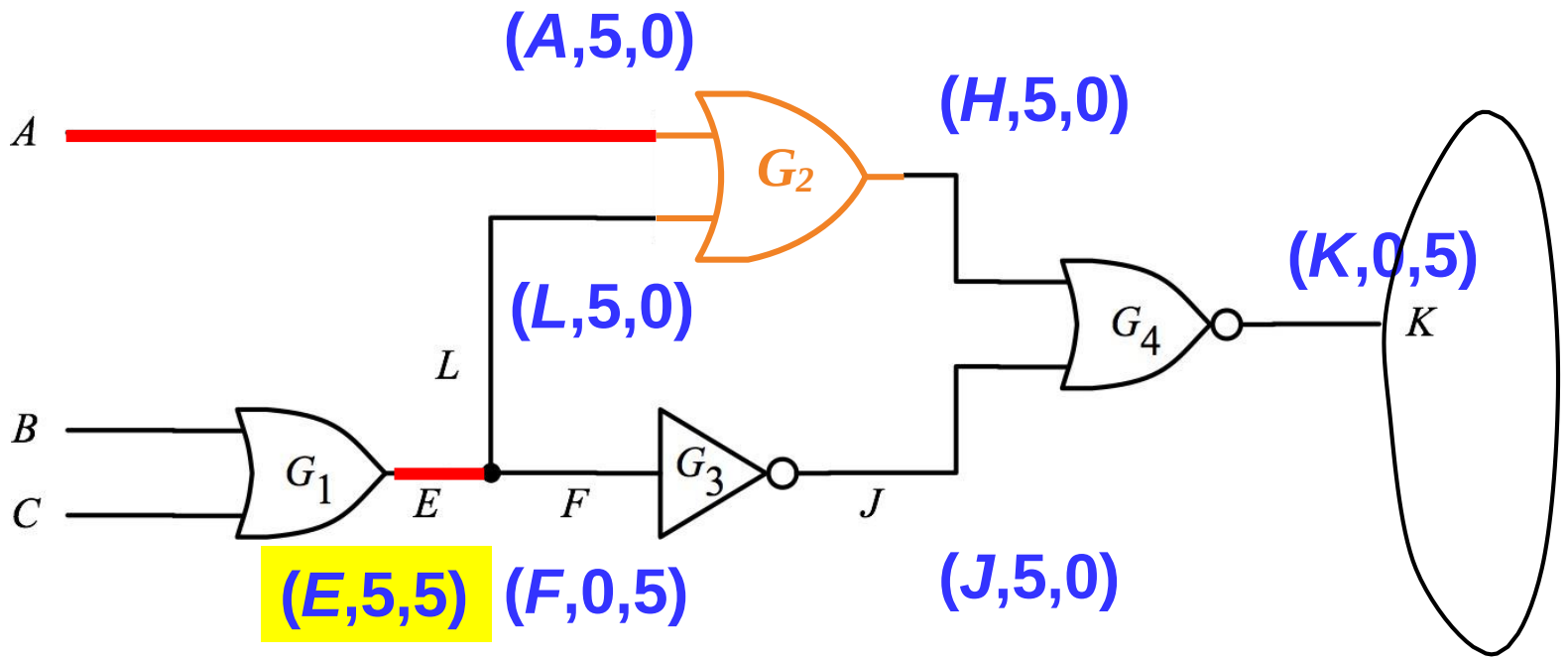
Q: Starting from (*K*, 5, 0), multiple backtrace to head lines
(*E*, n_0 , n_1)=? (*A*, n_0 , n_1)=? Suppose *H* is chosen over *J*

A:



Multiple Backtrace Conflict

- What if change G_2 to OR gate?
- Want *K*=1, perform multiple backtrace to headlines *A* and *E*
  - conflicting values at *E* !



- How to handle a conflict ?
  - assign a value most requested, then start next backtrace
  - will backtrack if it is wrong

Multiple Backtrace (1/2)

```
MultipleBacktrace (Initial_objectives, Fanout_objectives) {
  Current_objectives = Initial_objectives
  while (Current_objectives ≠ ϕ  or Fanout_objectives ≠ ϕ) {
     dequeue entry (k, v_k) from Current or Fanout_objectives
     switch (type of entry) {    //  (k, v_k ) = want v_k on signal k
       1. HEAD_LINE:
          add (k, v_k) to Headline_objectives
       2. FANOUT_BRANCH:
          j = stem(k);
          increment n_0 or n_1 at j for v_k;   //sum of n_0, n_1
          add j to Fanout_objectives
       3. GATE:                                  //page 15
          i = inversion of k; c = controlling value of k;
          if ((v_k ⊕ i) == c) {
               select easiest input  j with unknown value
               add( j, c) to Current_objectives;}
          else {
               for every input  j of k with value
               add(j, c') to Current_objectives; }
     }  // switch
```

Multiple Backtrace (2/2)

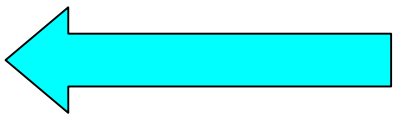
(cont'd from previous page)

*simplified from Fig. 8 of FAN paper

```
    if(Fanout_objectives≠ ϕ ) {
       dequeue highest-level stem (k) from Fanout_Objectives
       v_k = 0 or 1, depends on which of (n_0 , n_1) is larger
       if there is no conflict on k {
          add (k, v_k) to Current_objectives}  // continue backtrace
       else { return (k, v_k) as the Final_objective} } // stop backtrace
    else {   // no fanout objective
       dequeue (k, v_k) from Headline_objectives
       return (k, v_k) as the Final_objective }
  }  // while
}  // MultipleBacktrace
```

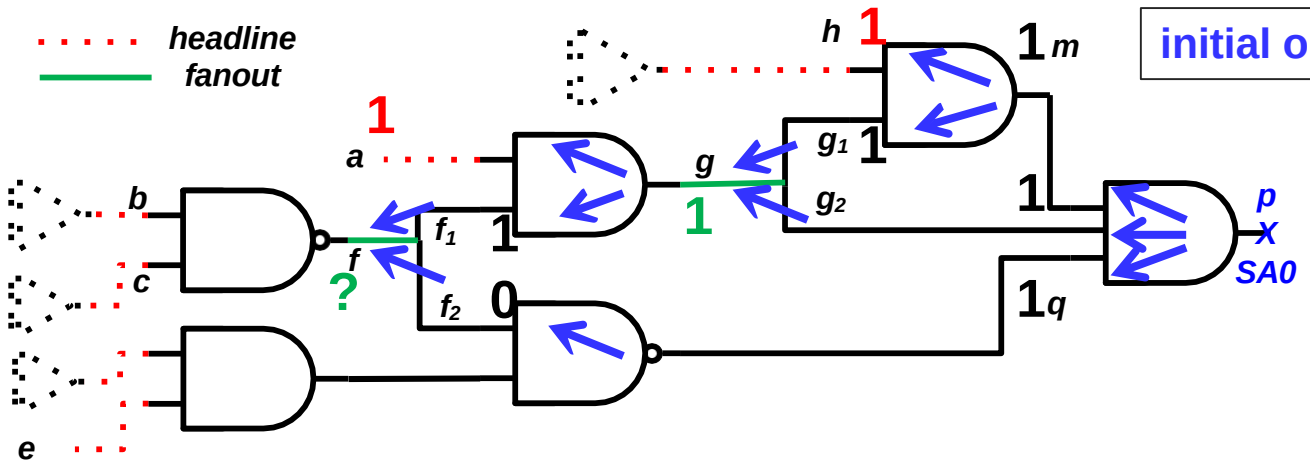
MultipleBacktrace

final_objective
(either a fanout or a headline)



initial_objectives
fanout_objectives

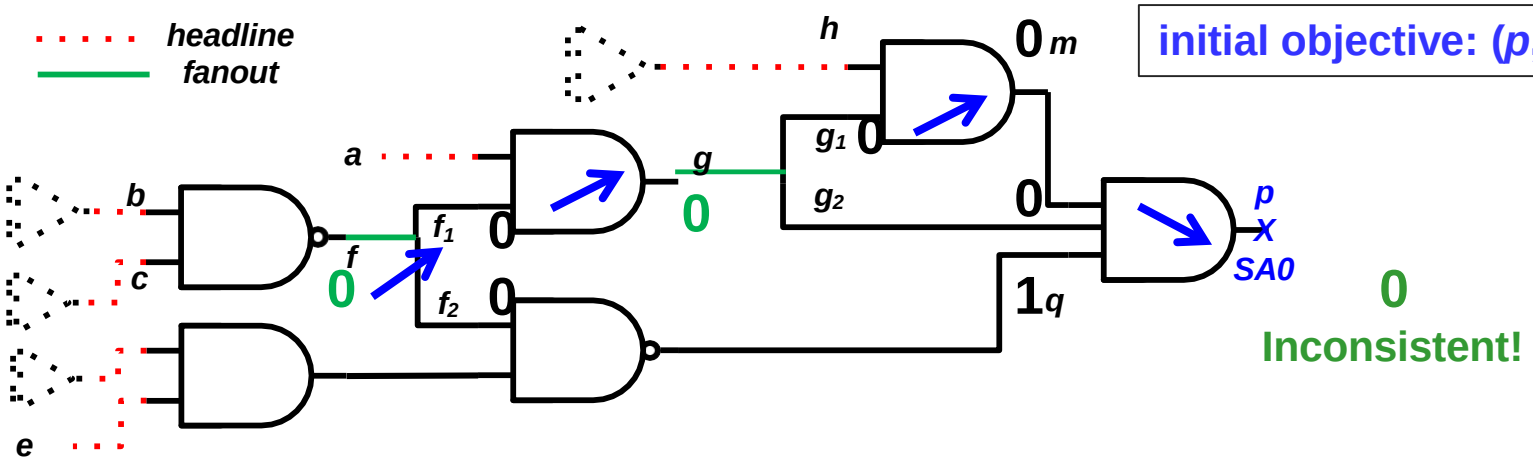
Multiple Backtrace Example (1/3)



Current Obj.	processed entry	Fanout Obj.	Headline Obj.
$(m,1)(g_2,1)(q,1)$	$(m,1)$	-	$(h,1)$
$(g_2,1)(q,1)(g_1,1)$	$(g_2,1)$	$(g,n_1=1)$	$(h,1)$
$(q,1)(g_1,1)$	$(q,1)$	$(g,n_1=1)$	$(h,1)$
$(g_1,1)(f_2,0)$	$(g_1,1)$	$(g,n_1=2)$	$(h,1)$
$(f_2,0)$	$(f_2,0)$	$(g,n_1=2)\ (f,n_0=1)$	$(h,1)$
-	$(g,1)$ consistent	$(f,n_0=1)$	$(h,1)$
$(g,1)$	$(g,1)$	$(f,n_0=1)$	$(h,1)(a,1)$
$(f_1,1)$	$(f_1,1)$	$(f,n_0=1\ n_1=1\ )$	$(h,1)(a,1)$
-	f conflict!	-	$(h,1)(a,1)$

Final Objective: *f=?*

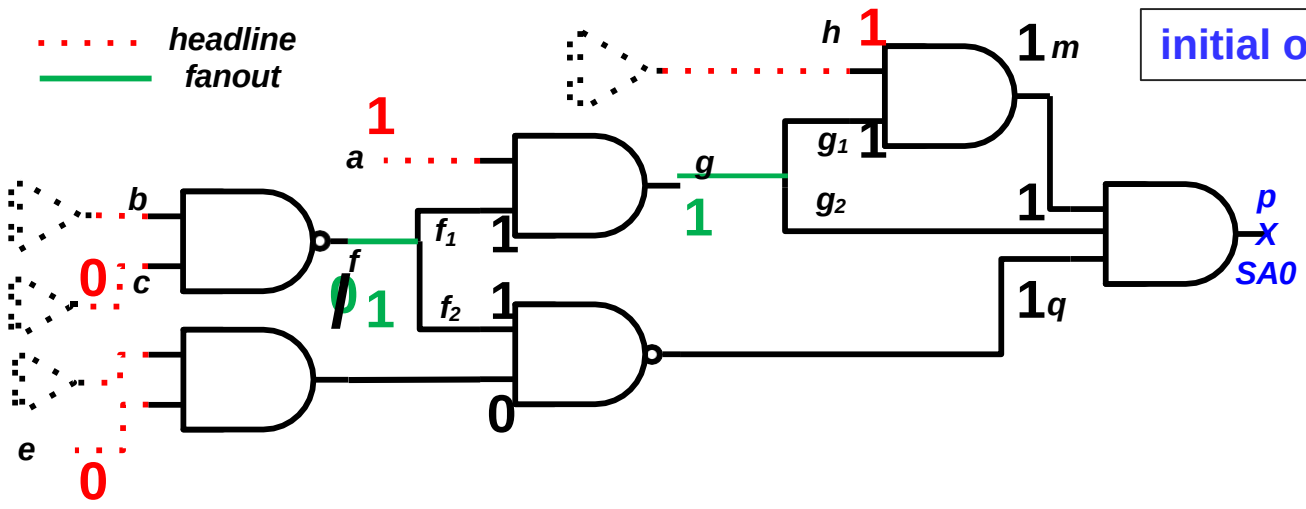
Multiple Backtrace Example (2/3)



because $n_0=n_1=1$, just choose randomly
suppose we assign *f* = 0
forward implication
inconsistent with initial objectives!
backtrack to *f*=1

Decision at Fanout Stem
Detects Inconsistency Earlier

Multiple Backtrace Example (3/3)



Assign *f*=1

Forward implication, consistent.

Multiple_Backtrace again

This time, headline objectives: *h=1, a=1, e=0, c=0*

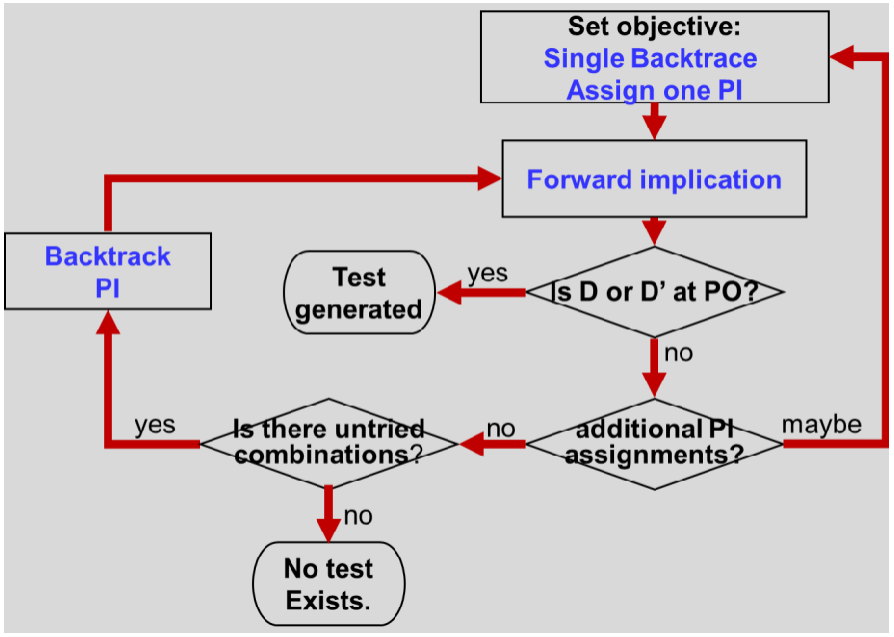
Forward implication

Initial objective achieved!

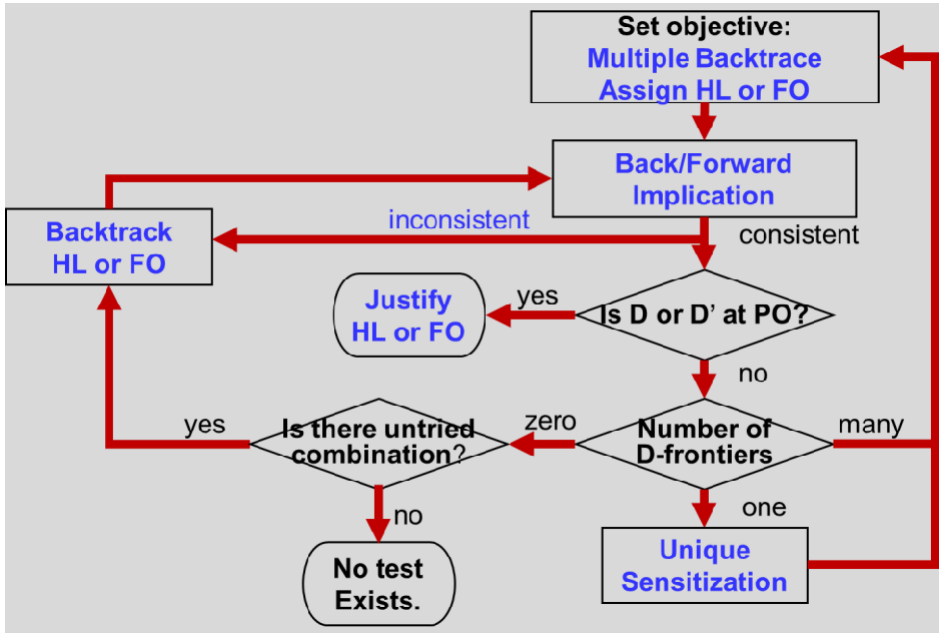
Multiple Backtrace is Fast

PODEM v.s. FAN

PODEM



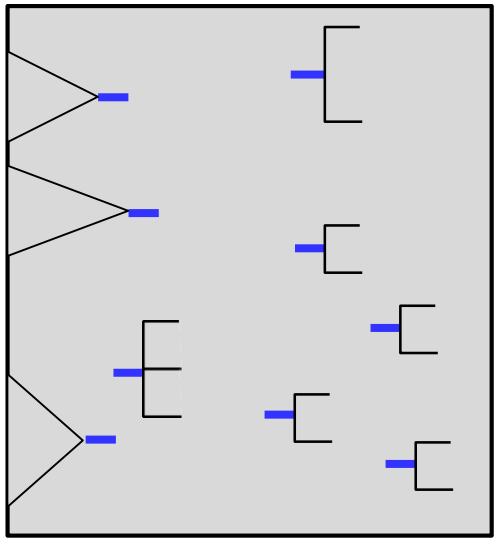
FAN



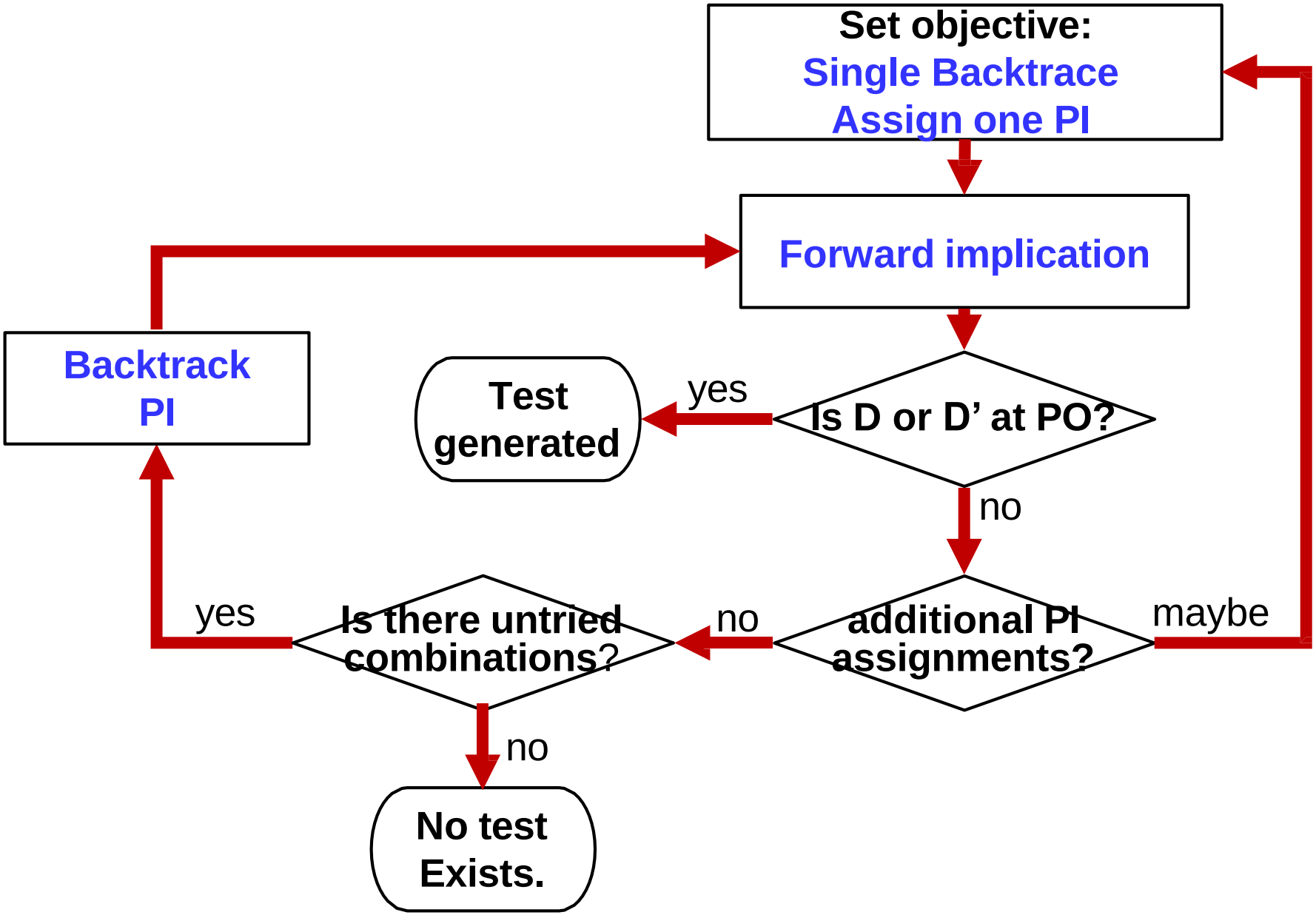
Difference highlighted in Blue

Summary

- 1. Make decision at head lines and fanout stem
  - Reduce search space
- 2. Forward/backward Implications
  - More information to make correct decision
- 3. Unique sensitization
  - Unique path to output
- 4. Multiple backtraces
  - BFS to search many paths together

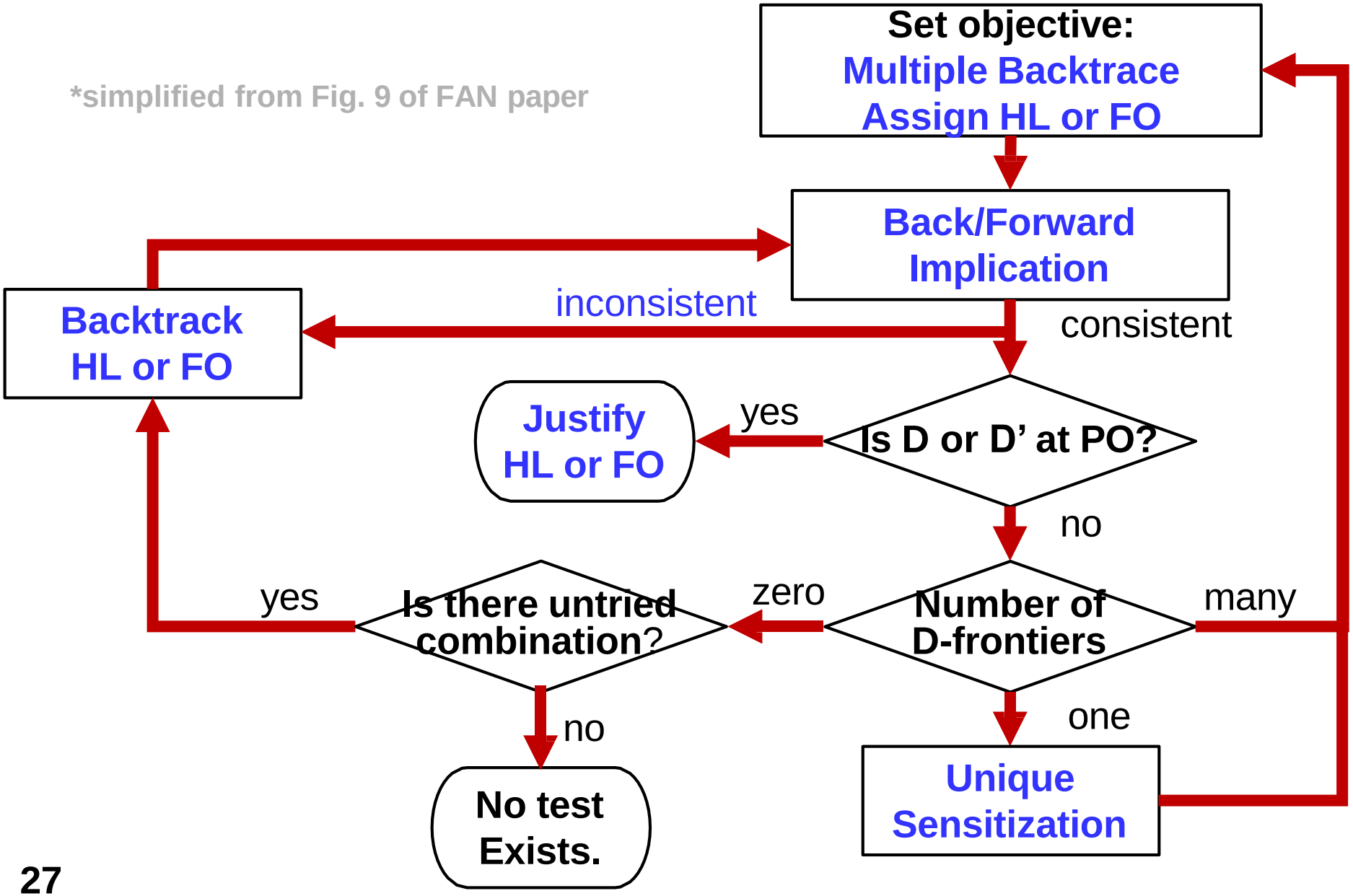


Flowchart of PODEM



Flowchart of FAN

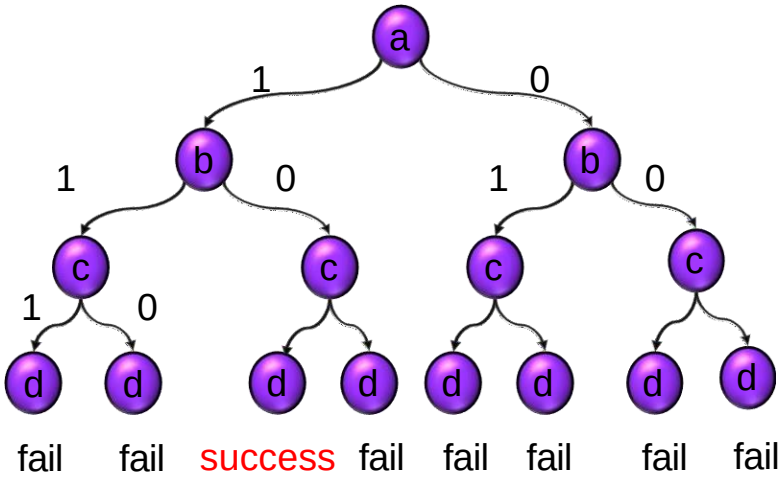
*simplified from Fig. 9 of FAN paper



Combinational ATPG

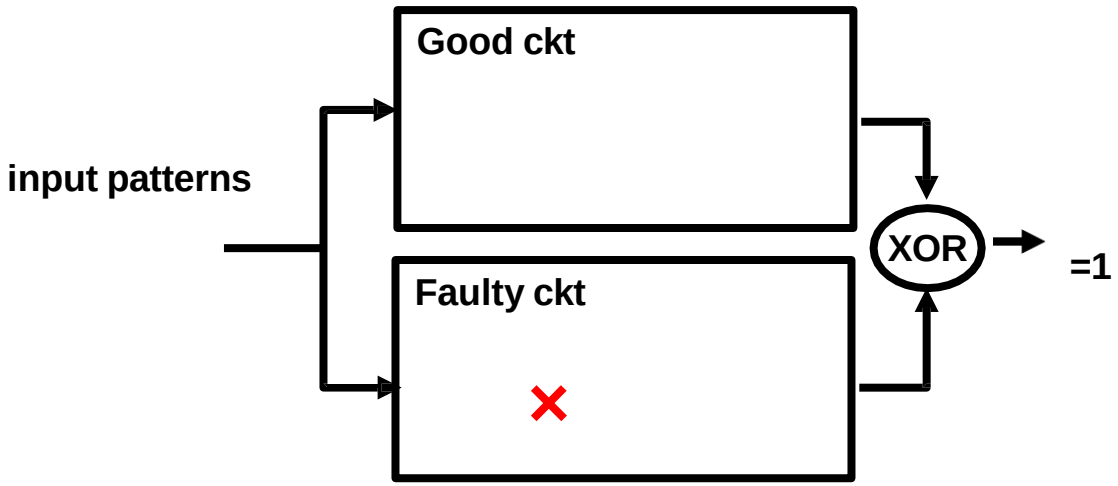
- Introduction
- **Deterministic Test Pattern Generation**
  - Boolean difference *
  - Path sensitization **
  - D-Algorithm [Roth 1966]**
  - PODEM [Goel 1981]**
  - FAN [Fujuwara 1985]**
  - **SAT-based [Larrabee 92] ***
- Acceleration techniques
- Concluding Remarks

*Boolean-based methods
**path-based methods



SAT-based ATPG

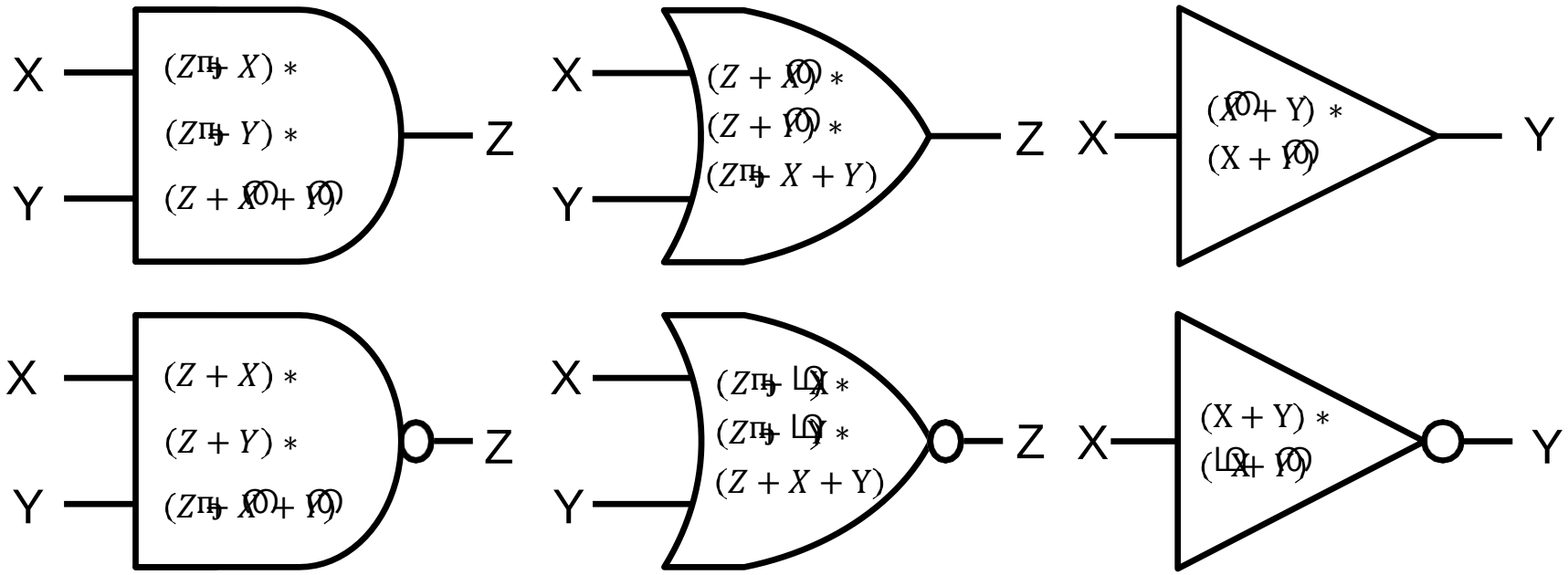
- **Idea: convert test generation problem into *satisfiability problem***
  - **Solve for a set of inputs such that**
  - **(good output) XOR (faulty output) = 1**



this is called a ***miter***

Tseitin Transformation [Tseitin 66]

- **Converts circuit into *Conjunctive Normal Form (CNF)***
  - **linear time, linear number of clauses and literals**

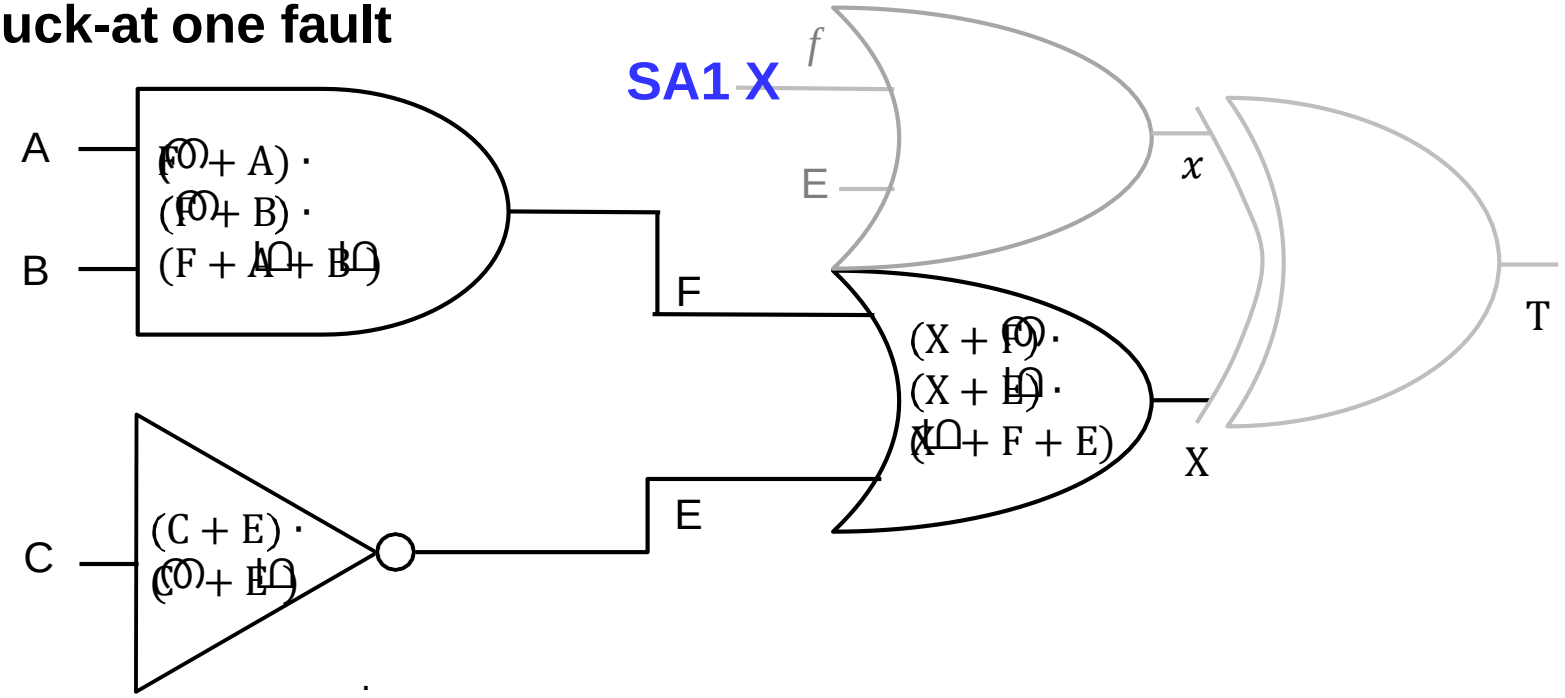


CNF for basic gates

Example (cont'd)

F good
f faulty

- **F stuck-at one fault**



faulty

$$(x + f) \cdot (x + \bar{E}) \cdot (x + f + E) \cdot (f)$$

good

$$\cdot (C + E) \cdot (\bar{C} + \bar{E})$$
$$\cdot (X + F) \cdot (X + \bar{E}) \cdot (\bar{X} + F + E)$$
$$\cdot (\bar{F} + A) \cdot (\bar{F} + B) \cdot (F + \bar{A} + \bar{B})$$

XOR

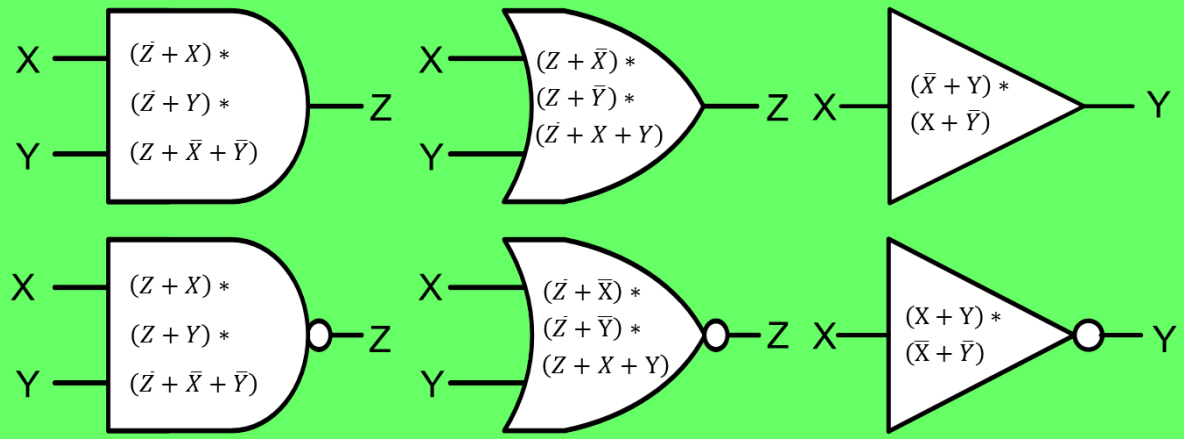
$$\cdot (\bar{X} + x + T) \cdot (X + x + T)$$
$$\cdot (\bar{X} + x + \bar{T}) \cdot (X + x + \bar{T}) \cdot T$$

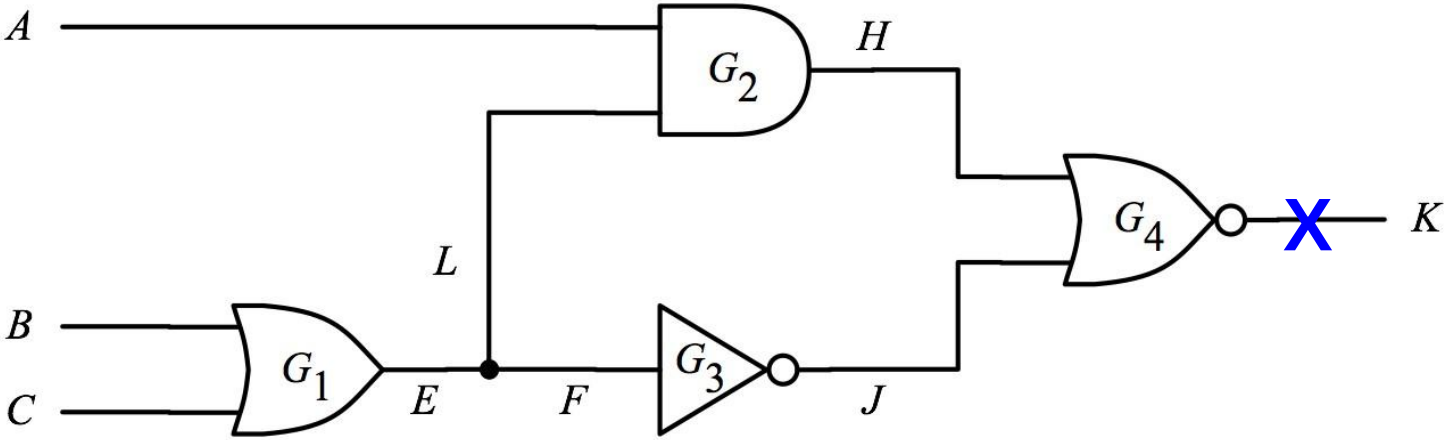
=1

Quiz

Q: Please write CNF to generate a test for K stuck-at one fault

A:





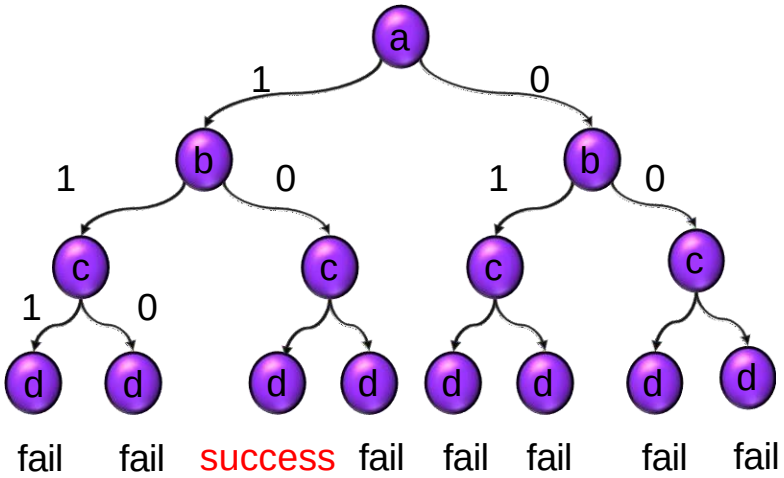
Pros and Cons

- **SAT-based ATPG**
  - + **SAT engine is making big progress recently**
  - + **proves untestable faults if CNF is unsatisfiable**
  - – **does not allow don't cares in input**
  - – **needs to redo CNF every time a target fault is injected**
  - – **does not preserve circuit structure information**
  - – **difficult for multi-valued logic (such as high impedance)**

SAT-based ATPG Not Used in Commercial Tool

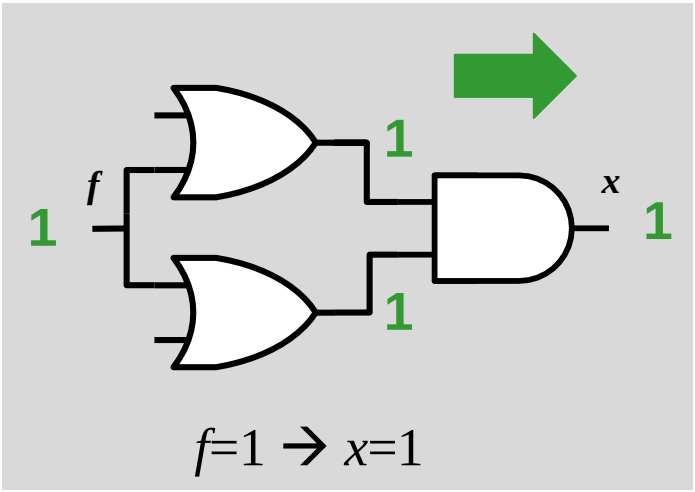
Combinational ATPG

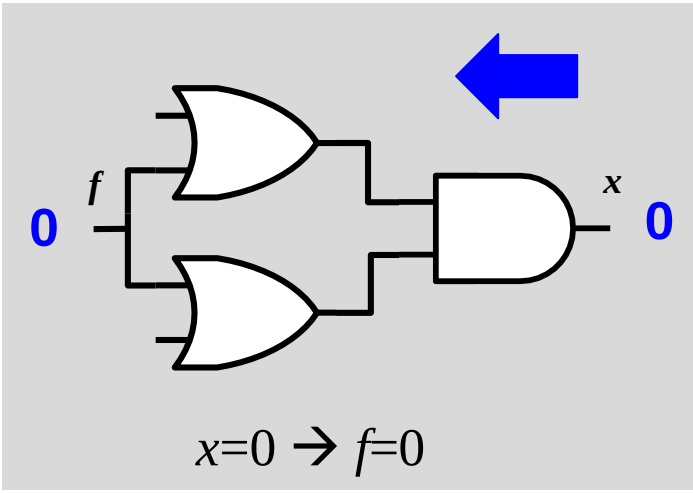
- Introduction
- Deterministic Test Pattern Generation
- **Acceleration techniques**
  - **Learning [Schulz 1988]**
  - **Redundant fault identification [Iyer 1996]**
- **Concluding Remarks**



Learning

- ***Learning*** **memorizes circuit information to speed up test generation**
  - ***Static learning:*** **performed in preprocess. no test pattern required.**
  - ***Dynamic learning*****: performed in test generation (not in lecture)**
- **Static learning example: WWW Fig 4.22**
  - **Set *f=1*, implies *x=1***
  - **So we learn *x=0*, implies *f=0***



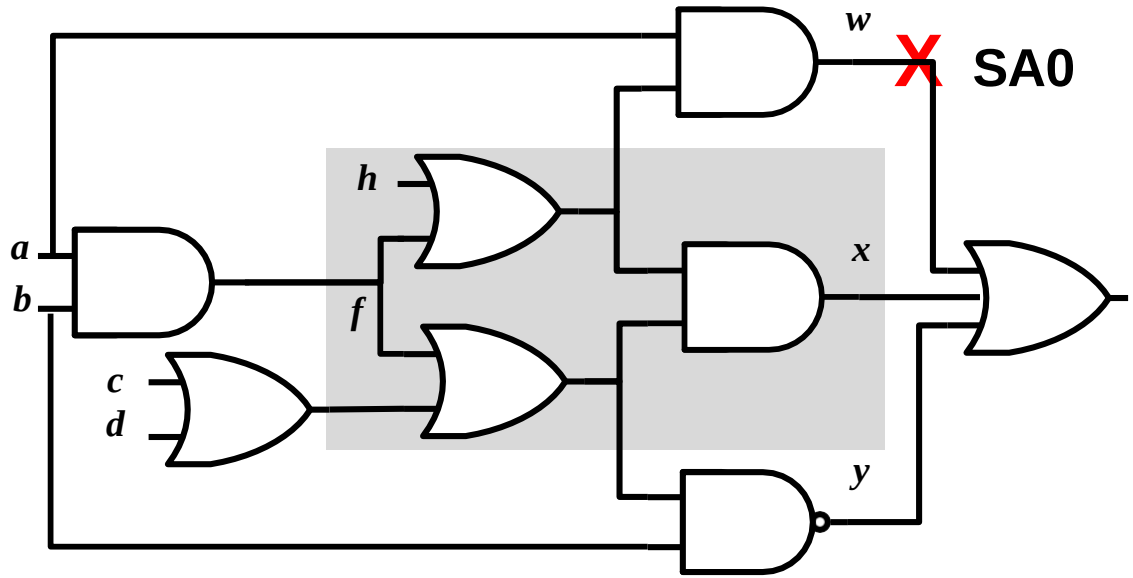


Contrapositive Law: *If p→q then q'→p'*

Learning Reduces Backtracks (1)

without learning

- Obj : w=1. assign h=1, a=1
- Obj : x=0. assign c=0, d=0, b=0
  - y=1 conflict!
- backtrack b=1
  - x=1 conflict!
- backtrack d=1, conflict!
- backtrack c=1, conflict!
- backtrack a=0, conflict!
- backtrack h=0 …. NO TEST!



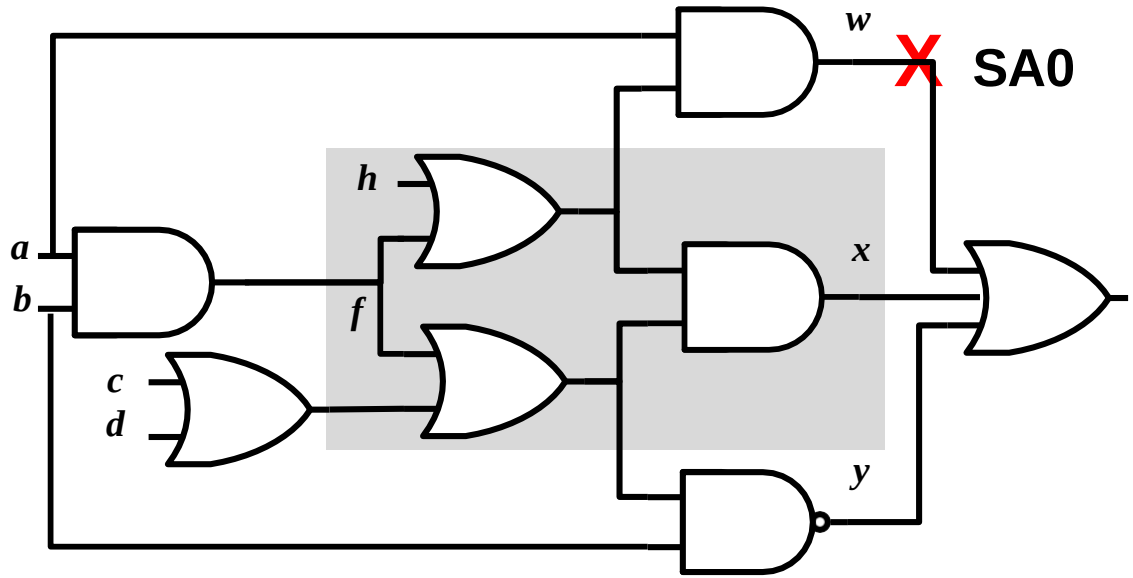
Learning Reduces Backtracks (2)

without learning

- Obj : w=1. assign h=1, a=1
- Obj : x=0. assign c=0, d=0, b=0
  - y=1 conflict!
- backtrack b=1
  - x=1 conflict!
- backtrack d=1, conflict!
- backtrack c=1, conflict!
- backtrack a=0, conflict!
- backtrack h=0 …. NO TEST!

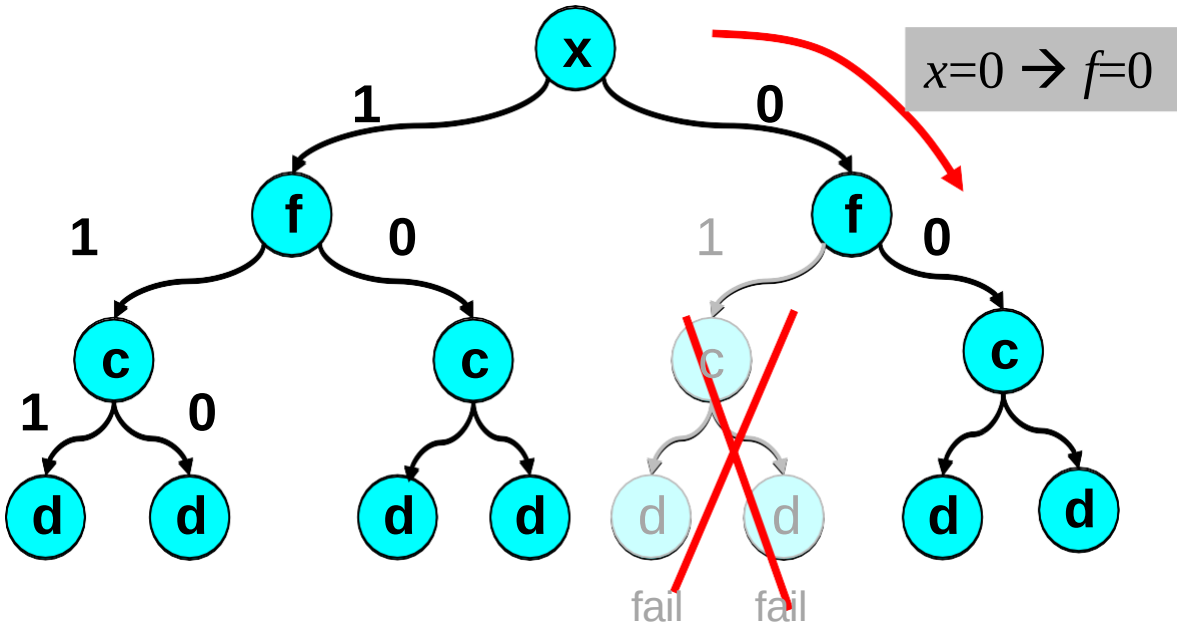
with learning

- Obj : w=1. assign h=1, a=1
- Obj : x=0 → f=0. assign b=0
  - y=1 conflict!
- backtrack b=1
  - x=1 conflict!
- backtrack a=0, conflict!
- backtrack h=0 … NO TEST!



Learning Speedup Decision

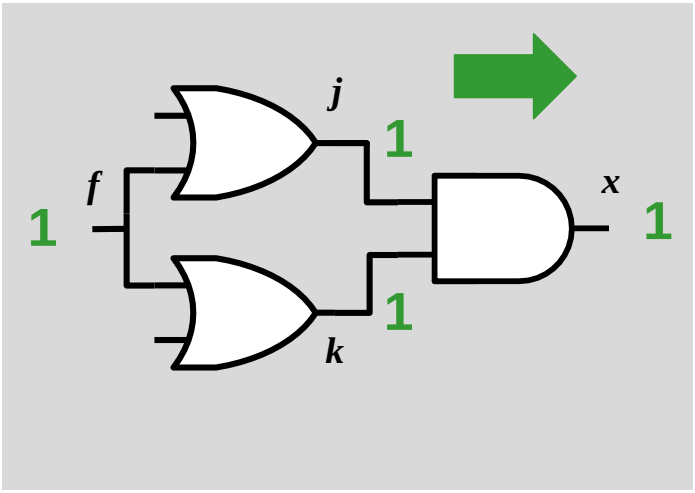
- Learning helps to
  - 1. Prune impossible sub-trees ASAP
  - 2. Find necessary assignments ASAP

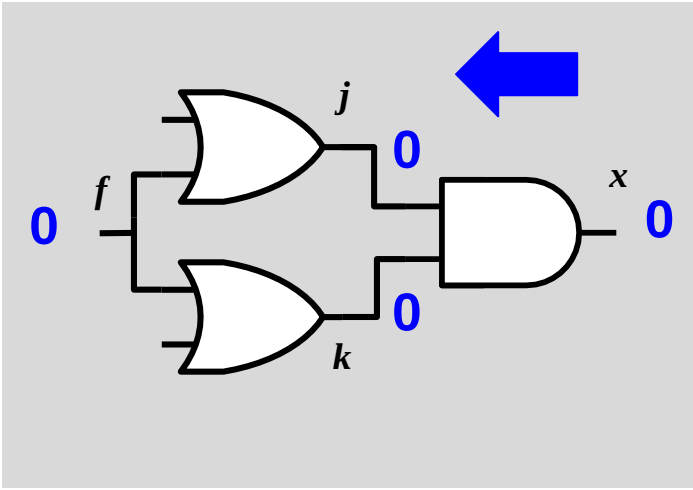


Learning Trade-off Memory for Run Time

But … Too Many to Learn!

- f=1 → j=1 so j=0 → f=0
  - Only local implication. not worth learning
- f=1 →k=1 so k=0 → f=0
  - Only local implication. not worth learning
- f=1 →x=1 so x=0 → f=0
  - global implication. worth learning!





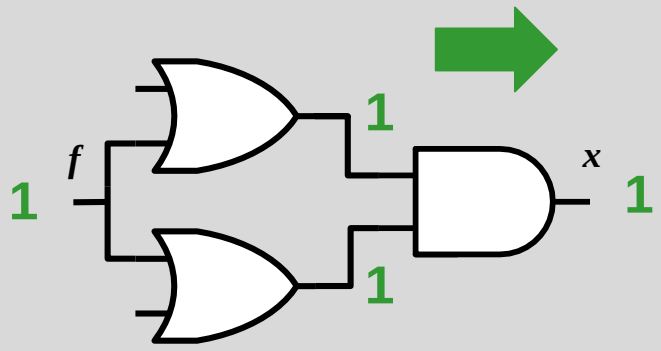
Which Are Worth Learning?

SOCRATES Algorithm [Schulz 1988]

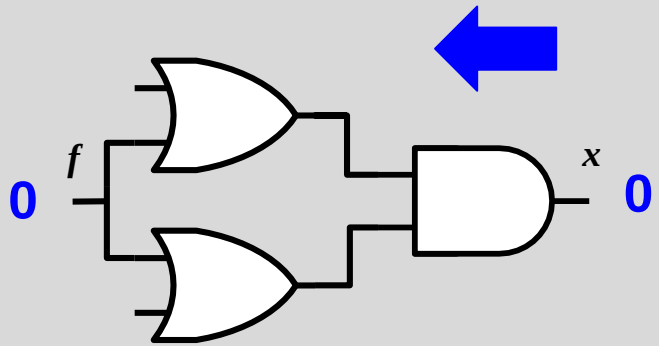
- **In preprocess phase, sets all signals to 0 and 1**
  - **Discovers what other signals are implies**
  - **Two criteria to select useful learning**

```
static_learning()
   for every signal f
      assign_value (f, 0)
      implication()
      analyze_results(f)
      assign_value (f,1)
      implication()
      analyze_results(f)
```

```
analyze_results (f)
   for every signal x whose value ≠ unknown
      if (all gate inputs to x are non-controlling)
      & (there is forward path from f to x)
            save learning result (x=v_x' → f = v_f')
```



f=1 → x=1

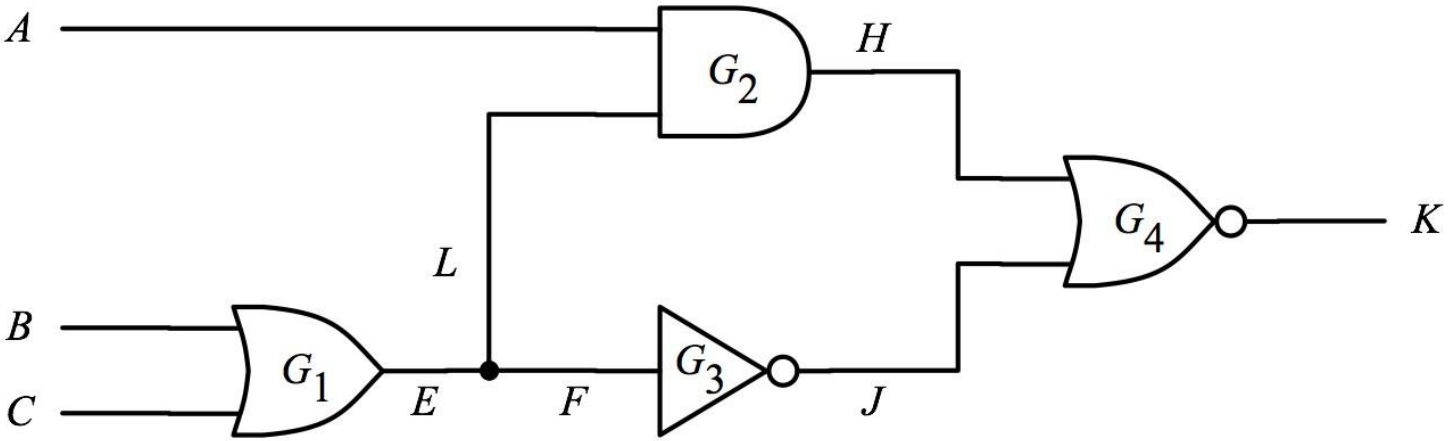


x=0 → f=0

Quiz

Q1: Set *E* = 0. What can you learn about *K* using contrapositive law?
A:

Q2: (Cont'd) If we want to detect *K* SA 0 fault, what is value of *E*?
A:

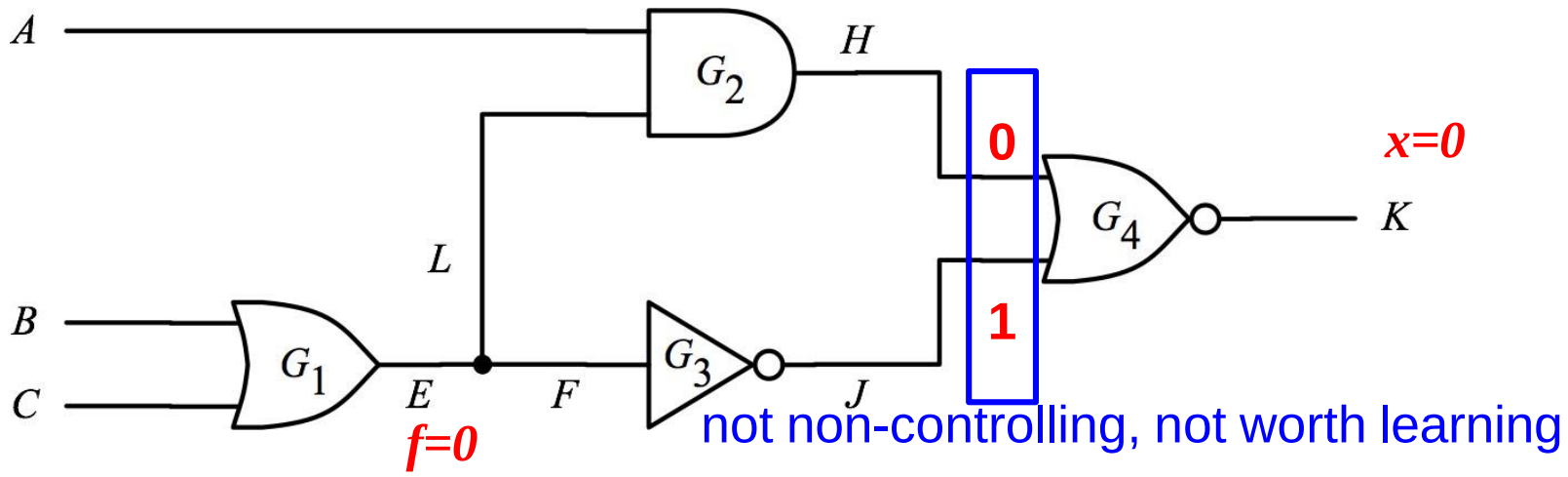


FFT

- **Q1: Why SOCRATES requires both inputs are non-controlling?**
  - **Hint: use following circuit as example**
- **Q2: Why forward path from *f* to *x*?**

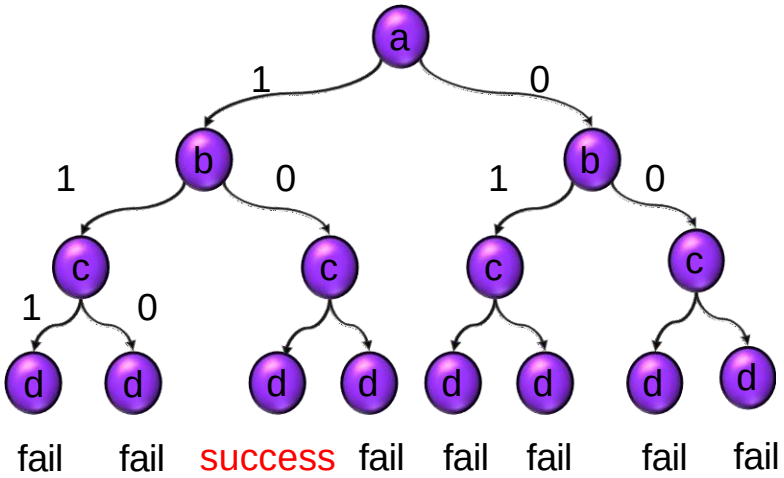
```
analyze_results (f)
  for every signal x whose value ≠ unknown
    if (all gate inputs to x are non-controlling)
    & (there is forward path from f to x)
        save learning result (x=v_x' → f = v_f')
```

```
static_learning()
  for every signal f
    assign_value (f, 0)
    implication()
    analyze_results(f)
    assign_value (f,1)
    implication()
    analyze_results(f)
```



Combinational ATPG

- Introduction
- Deterministic Test Pattern Generation
- **Acceleration techniques**
  - **Learning [Schulz 1988]**
  - **Redundant fault identification [Iyer 1996]**
- **Concluding Remarks**

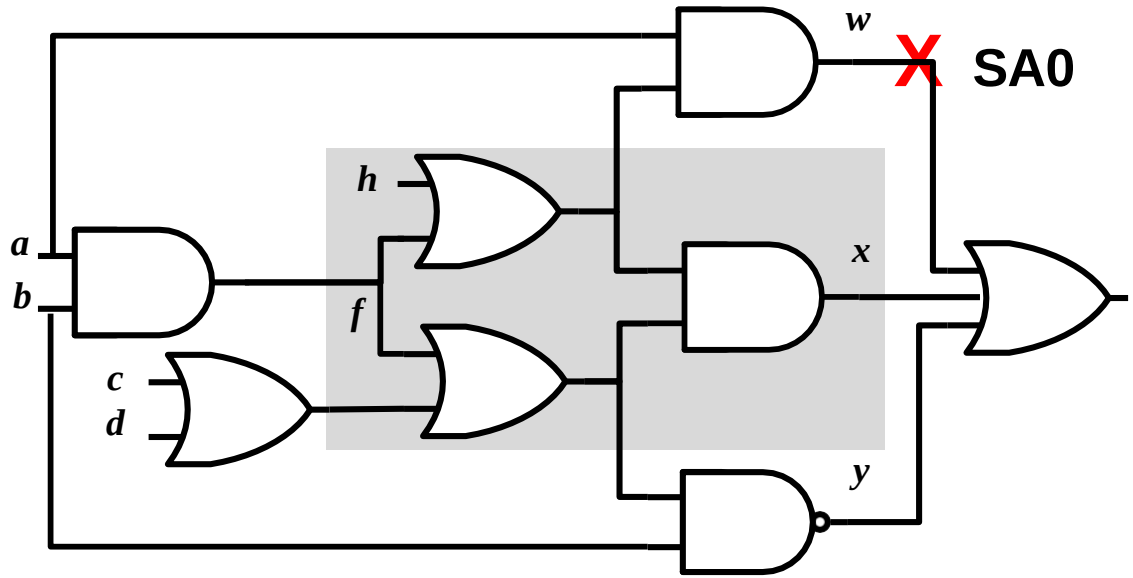


Proving Redundant Fault is Difficult

without learning

- Obj : w=1. assign h=1, a=1
- Obj : x=0. assign c=0, d=0, b=0
  - y=1 conflict!
- backtrack b=1
  - x=1 conflict!
- backtrack c=1, conflict!
- backtrack d=1, conflict!
- backtrack a=0, conflict!
- backtrack h=0 …. NO TEST!

Can We Find Redundant Fault Faster?



Redundant Fault Identification

- *Untestable faults* (aka. *redundant faults*)
  - faults that cannot be excited, or
  - faults that cannot be propagated, or
  - faults that cannot be simultaneously excited and propagated
- Why redundant fault identification?
  - ① Speed up ATPG
    - ATPG spend long time on untestable faults
  - ② Reduce area
    - Redundant logic can be removed
- To prove redundant fault is NP-complete
  - Quickly identify many redundant fault (not all) is good enough

Redundant Faults are Trouble for ATPG

FIRE [Iyer 1996]

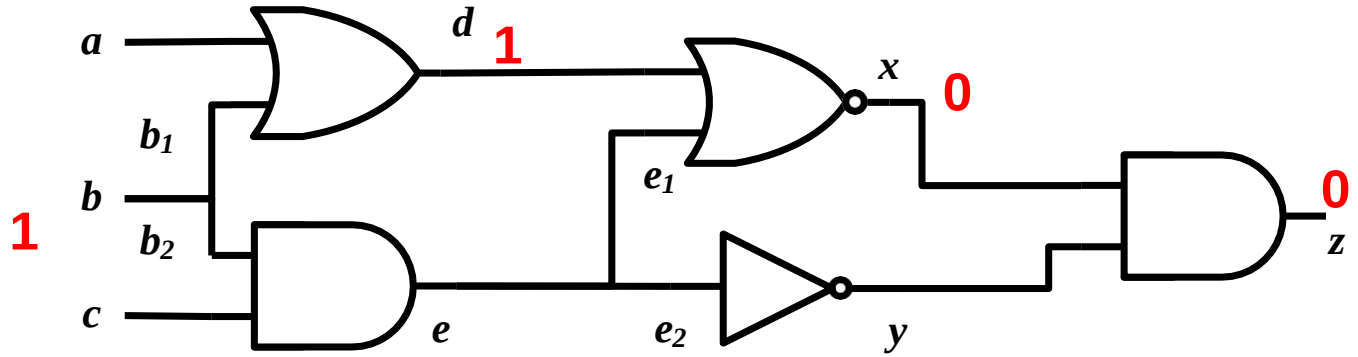
- Fault Independent Redundant Identification (FIRE)
  - based on single-line conflict analysis
- Idea
  - S0 = set of faults untestable when signal s=0
  - S1 = set of faults untestable when signal s=1
  - intersection $S0 \cap S1$ are untestable faults
- To find unexcitable faults
  - use forward implication
- To find unobservable faults
  - use backward tracing

- Advantage: close to linear time complexity (for one signal)
  - FFT: can we solve NPC problem in linear time?

FIRE Example

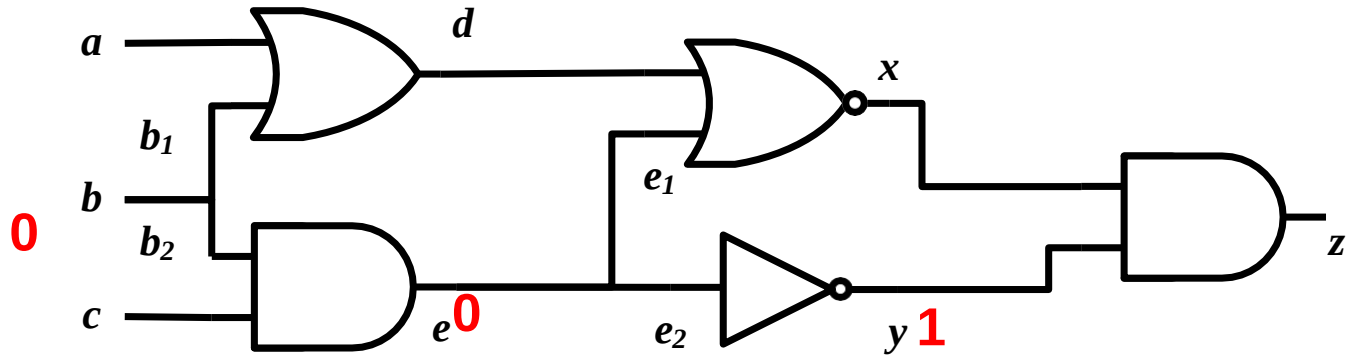
- set b=1 implies → {b=1, b_1=1, b_2=1, d=1, x=0, z=0}
  - Faults unexcitable when b=1: {b/1, b_1/1, b_2/1, d/1, x/0, z/0}
  - Faults unobservable when b=1: {a/0, a/1, e_1/0, e_1/1, y/0, y/1, e_2/0, e_2/1, e/0, e/1, b_2/0, b_2/1, c/0, c/1 }
    - Q: why b/0 not in the list?
  - Faults untestable when b=1: union of above two sets
    - S1={a/0, a/1, b/1, b_1/1, b_2/1, d/1,e1/0, e_1/1, e_2/0, e_2/1, e/1, e/1, x/0, y/0, y/1, z/0, b_2/0, c/0, c/1 }

b/1 = b SA1 fault



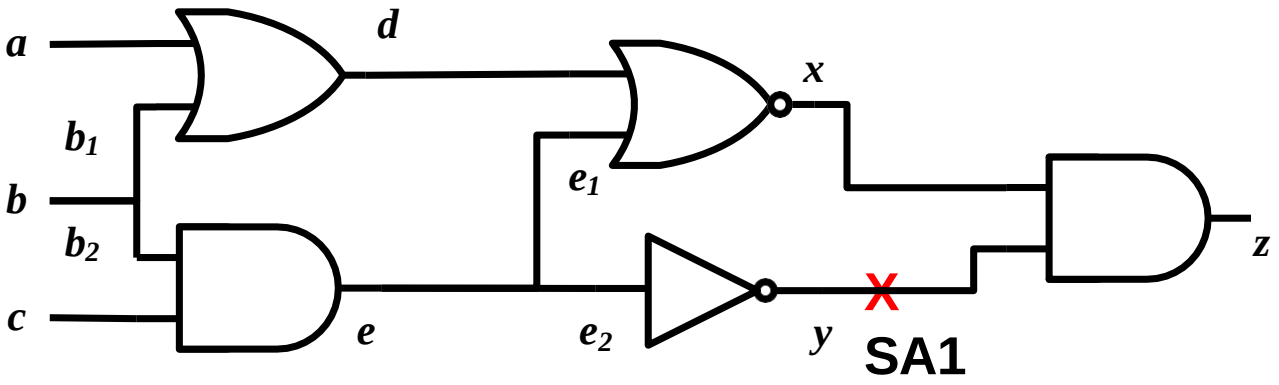
FIRE Example (cont'd)

- set b=0 implies → {b=0, b_1=0, b_2=0, e=0, e_1=0, e_2=0, y=1}
  - Faults unexcitable when b=0: {b/0, b_1/0, b_2/0, e/0, e_1/0, e_2/0, y/1}
  - Faults unobservable when b=0: {c/0, c/1}
  - Faults untestable when b=0: union of above two sets
    - S0= {b/0, b_1/0, b_2/0, c/0, c/1, e/0, e_1/0, e_2/0, y/1}
    - S1={a/0, a/1, b/1, b_1/1, b_2/1, d/1,e1/0, e_1/1, e_2/0, e_2/1, e/1, e/1, x/0, y/0, y/1, z/0, b_2/0, c/0, c/1 }
- Intersection of S1 and S0 are untestable faults
  - S1 ∩ S0 = {b_2/0, c/0, c/1, e/0, e_1/0, e_2/0, y/1}



FFT

- **Q1: Use PODAM to find a test for y/1 fault. Prove y/1 is untestable.**
- **Q2: Can we use linear time algorithm to solve NPC problem?**



Quiz

Q1: Set B=0. Find set of faults undetectable.

A: faults unexcitable = { }

faults unobservable= { }

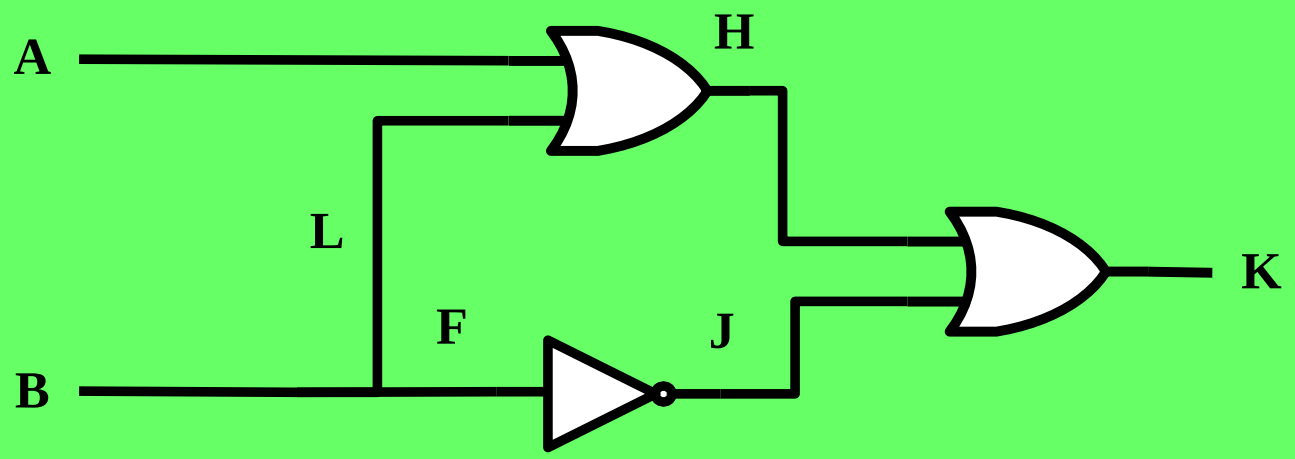
Q2: Set B=1. Find set of faults undetectable.

A: faults unexcitable = { }

faults unobservable= { }

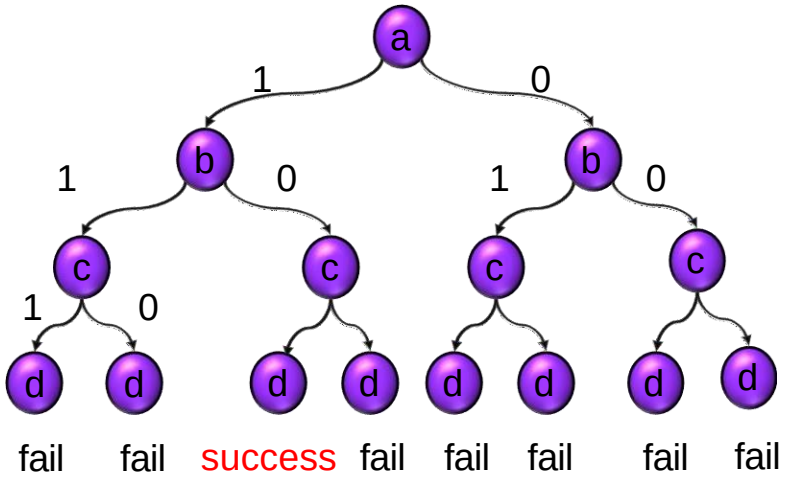
Q3: (cont'd) Which faults are redundant ?

A: Q1∩ Q2={ }



Combinational ATPG

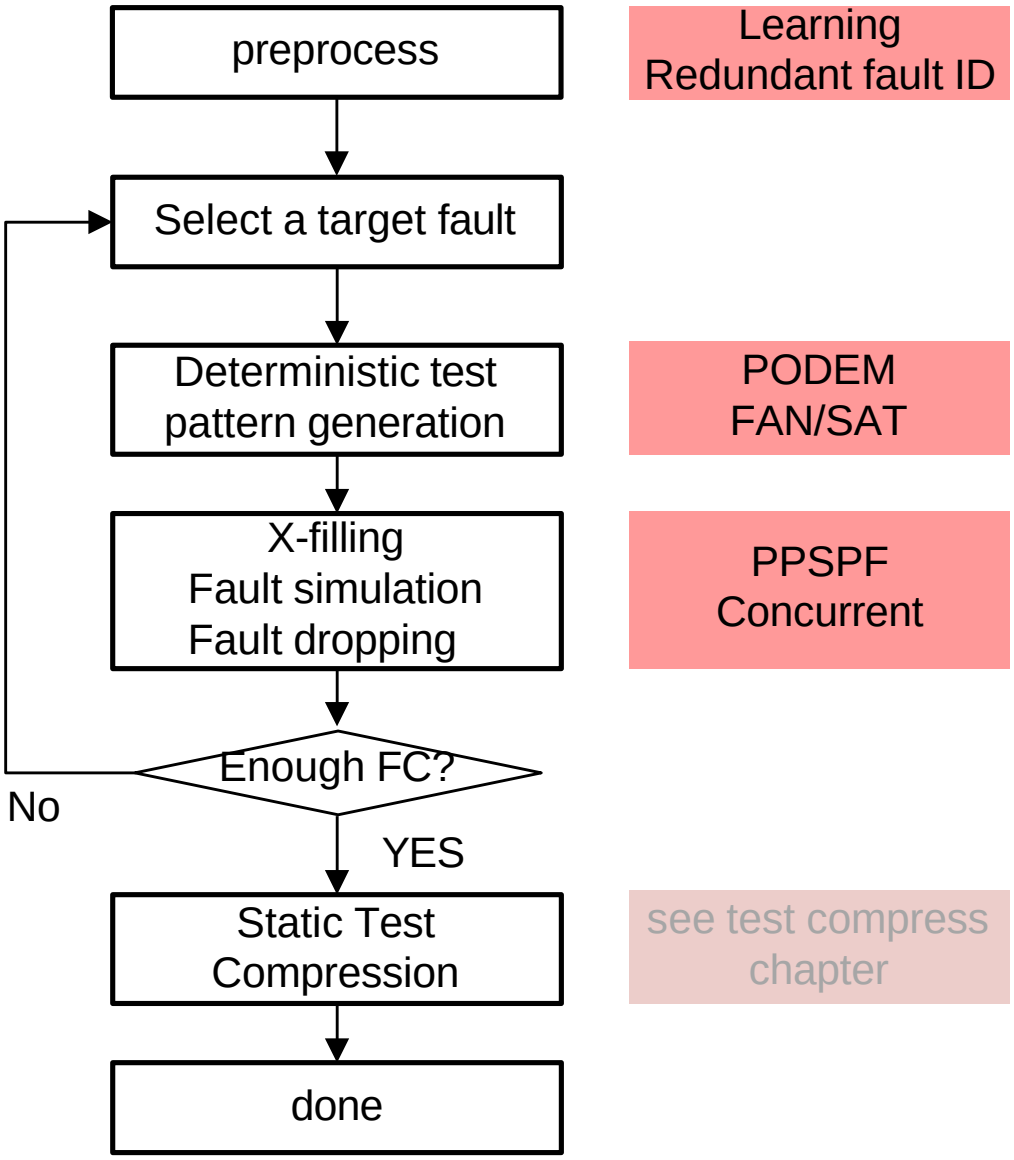
- Introduction
- Deterministic Test Pattern Generation
- Acceleration Techniques
- Concluding Remarks



Summary

- ATPG is NPC. Many acceleration techniques needed
- Static learning (SOCRATES) trade off memory for run time
  - Setting all signals to 0 and 1. imply other signals
  - Apply contrapositive law: if $p \rightarrow q$ then $q' \rightarrow p'$
- Redundant fault identification (FIRE)
  - Setting a signal to 0 and 1
    - unexcitable faults $\cup$ unobservable faults
  - Redundant fault = S1 $\cap$ S0
- Many other acceleration techniques
  - Dominator (TOPS) [Kirkland 87]
  - Recursive learning [Kunz 92]
  - Transitive Closure Graph (NNATPG) [Chakradhar 93]
  - …

ATPG Review


preprocess
Learning
Redundant fault ID
Select a target fault
Deterministic test
pattern generation
PODEM
FAN/SAT
X-filling
Fault simulation
Fault dropping
PPSPF
Concurrent
Enough FC?
No
YES
Static Test
Compression
see test compress
chapter
done

How to Read ATPG Report?

	Uncollapsed	Collapsed
Total Faults	1234	800
Detected faults	1000	700
Redundant faults	230	98
Aborted faults	4	2
Fault coverage	1000/1234 %	700/800 %
ATPG effectiveness	1230/1234 %	798/800 %
Test Length	328 patterns	
Run Time	10:57	

proven redundant by ATPG

undetected. not sure redundant or not.

$$\frac{detected\ \ faults}{total\ \ faults} \times 100\%$$

$$\frac{detected + redundant\ \ faults}{total\ \ faults} \times 100\%$$

less are better

References

- [Fujiwara 83] H. Fujiwara and T. Shimono, “On the Acceleration of Test Generation Algorithms, “ Proc. Int’l Fault-Tolerance Computing Symp., pp.98-105, 1983.
- [Goel 81] P. Goel, “An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits,” IEEE Trans. On Computers, Vol. C-30, No. 3, pp.215-222. Mar. 1981.
- [Roth 66] J. P. Roth, “Diagnosis of Automata Failures: A Calculus and a Method,” IBM Journal of Research and Development, vol. 10, no. 4, pp278-291, 1966.
- [Larrabee 92] T. Larrabee, “Test pattern generation using Boolean satisfiability,” IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, Volume 11, Issue 1, Jan 1992, pp. 4-15.
- [Iyer 96] Iyer, M.A. Abramovici, M. “FIRE: a fault-independent combinational redundancy identification algorithm,” Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, Jun 1996, Volume: 4, Issue: 2 page(s): 295-301
- [Schulz 88] M.H Schulz, et al “SOCRATES: A highly efficient automatic test pattern generation system IEEE TCAD, vol.7 no.1 pp.126, 1988.
- [Marques 94] J. Marques, S. Karen, A. Sakallah, “Dynamic search-space Pruning Techniques in Path Sensitization’ DAC 1994.

Commercial Tools

- Mentor Graphics
  - Fastscan
- Synoposys
  - Tetramax
- Syntest
  - Turboscan
- Cadence
  - Encounter Test