Laboratorij za načrtovanje integriranih vezij

Univerza *v Ljubljani*
Fakulteta *za elektrotehniko*

Preizkušanje elektronskih vezij

Generacija testnih vzorcev
Test pattern generation

Overview

- Introduction
- Theoretical Background in Boolean Difference
- Designing a Stuck-at ATPG for Combinational Circuits
- Designing a Sequential ATPG
- ATPG for Non-Stuck-At Faults
- Other Issues in Test Generation

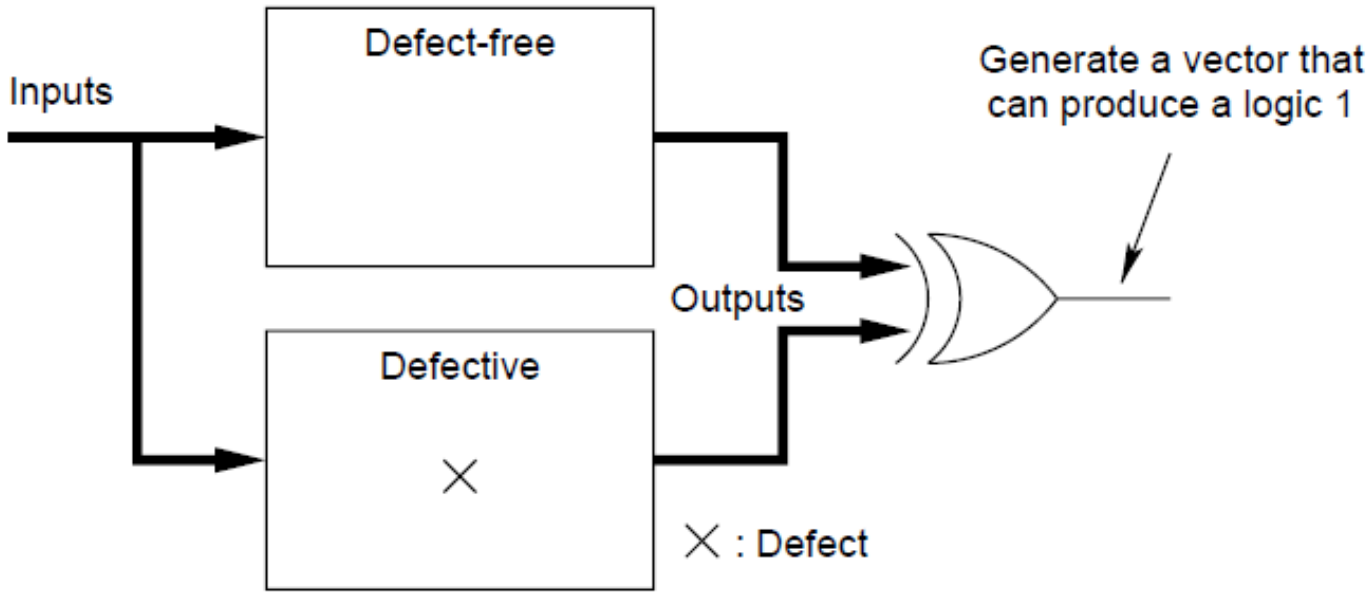
Introduction

- Test generation is the task of **searching for a test pattern** that will detect a specific fault.
- This process is called **ATPG – Automatic Test Pattern Generation**.
- Without powerful and efficient ATPG, chips will mostly depend on design for testability techniques (-> increasing area and cost).
- ATPG is one of the most important, challenging and difficult problem.

- The goal of this chapter is to present ATPG techniques for various fault models.

Introduction

- For any defective chip, which is functionally different from the defect-free chip, there must exist at least one input (test vector) that can differentiate both chips.
- The goal of ATPG is to efficiently generate that test vector.

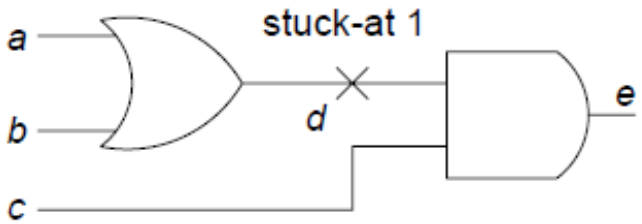


Introduction

- If ATPG is capable to deliever high-quality test paterns (high fault coverage and small test set), DFT is no longer necessary.
- As it is difficult to generate test vectors targeting all possible, ATPG operate on an abstract representation of defects, referred as faults.
- The most popular and used fault model is single stuck-at fault model.

Introduction

- Single stuck-at fault model assumes that a circuit node is tied to logic 1 or logic 0
  - single stuck-at 1, s-s-1
  - single stuck-at 0, s-s-0
- Consider the single stuck-at 1 at node d (d/1).
- First, we have to activate the fault, i.e. to find the difference between the fault-free circuit and the circuit with the fault d/1.
- Then, we have to propagate information about the fault (fault signal) to the circuit output node.



Introduction

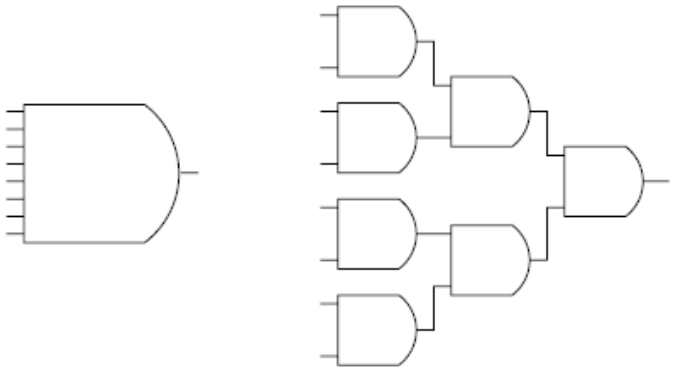
- ATPG systems attempts to generate test vectors for every possible fault in the circuit.
- In this example, other faults d/0, a/0, a/1, b/0, are targeted by ATPG.
- As some of the faults in the circuit can be logically equivalent, no test can be obtained to distinguish between (these faults form a set of equivalent faults).
- Therefore, fault collapsing is used before ATPG in order to reduce the number of faults to consider.

Random Test Generation

- Random Test generation (RTG) is one of the simplest methods for generating vectors.
- Test vectors are randomly generated and fault simulated (fault graded) on the circuit under test (CUT).
- Because no specific fault is targeted , the RTG complexity is low.
- Disadvantages of RTG are large test set size and not sufficiently high fault coverage.
- Logic values are randomly generated at the primary inputs, with equal probability of assigning a logic 1 or logic 0 to each primary input.
- Note, that pseudo-random generator is most often used -> repeated test set with the same pseudo-random generator.

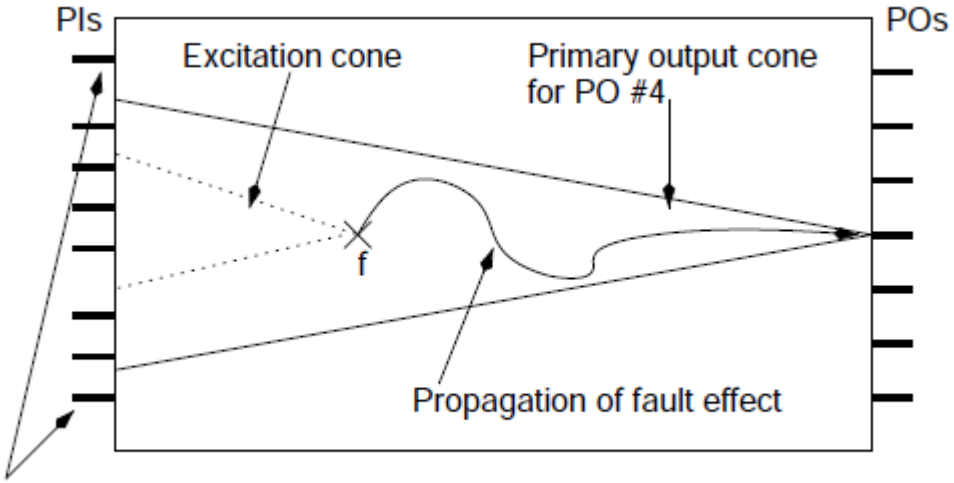
Random Test Generation

- **Level of confidence** on a random test set T can be measured as the probability that T can detect all stuck-at faults in the circuit.
- For N random vectors, the test quality t_N indicates that all detectable stuck-at faults are detected by these N random vectors.
- Some faults (**random-pattern resistant faults**) are difficult to test with RTG. Example:



Random Test Generation

- To target random-pattern resistant faults, biasing is required -> input vectors are no longer uniformly distributed.
- Determining the optimal bias values for each primary input is difficult task.
- Minimum detection probability of a detectable fault *f* can be determined by the output cone in which *f* resides.



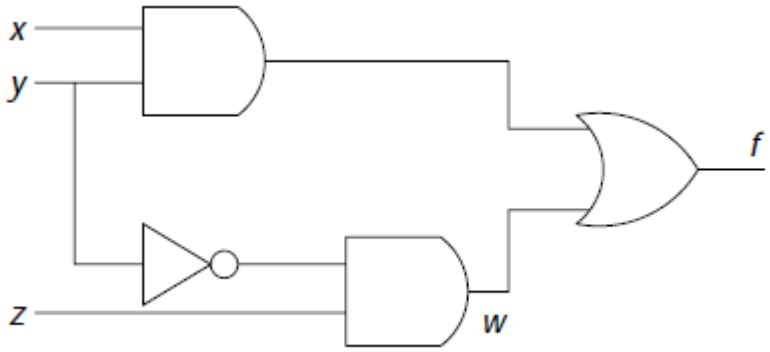
Exhaustive Test Generation

- If the combinationa circuit has few primary inputs, **exhaustive testing** (**ET**) is a viable option.
- Every possible input vector is enumerated.
- This is superior to RTG since RTG can produce duplicated vectors.
- For large circuit, ET is impractical.
- But, it may be possible to partition the circuit and only exhaust the input vectors within each cone (pseudo-exhaustive testing, **PET**).
- For the circuit with three PO, each with corresponding cone with n_1, n_2 and n_3 PI, the number of PET is at most $2^{n1} + 2^{n2} + 2^{n3}$

Theoretical Background: Boolean Difference

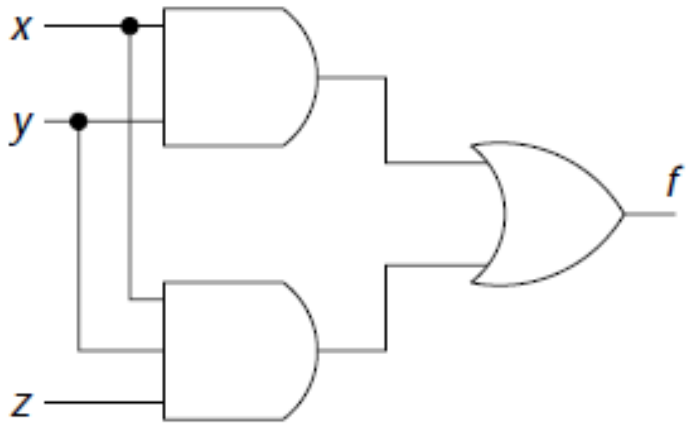
- Consider s-s-0 on primary input y (y s-s-0).
- Faulty circuit f‘ = f(y=0)
- Test vector must satisfy equation:

  f(y=1) EX-OR f(y=0) = 1
- Also, fault must be first excited:
  - y · f(y=1) EX-OR f(y=0) = 1
  - f(y=1) EX-OR f(y=0) is called **Boolean Difference** of *f* with respect to *y*



Untestable Fault

- If there exists no input vector to test a certain fault, the fault is **untestable** (or **redundant**)
- Consider the fault z/0.



Stuck-at ATPG for Combinational Circuits

- In ATPG, there are two main tasks:
  - 1) Excitation of target fault
  - 2) Propagation of the fault to the primary output
- Logic values for fault-free and faulty circuit are needed (v and v_f).
- 5-valued algebra (Roth): *0, 1, X*, *D* and *D'* is used.
  - $D = 1/0$ and $D' = 0/1$.
- Boolean operators (AND, OR, NOT, XOR) can be used on 5-valued algebra.
- The simplest way to perform Boolean operations is to represent each component value into the v/v_f form and perform Boolean operations on fault-free and faulty value separately.

Boolean operation for 5-valued algebra

AND Operation

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

OR Operation

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

NOT Operation

NOT	
0	1
1	0
D	$\overline{D}$
$\overline{D}$	D
X	X

Naive ATPG Algorithm

- Worst-case computational complexity is exponential.
- All possible input patterns may have to be tried before a vector is found or that the fault is declared as undetectable.
- Intelligence mechanism can be used to reduce the search space.

Algorithm 1 Naive ATPG (C, f)

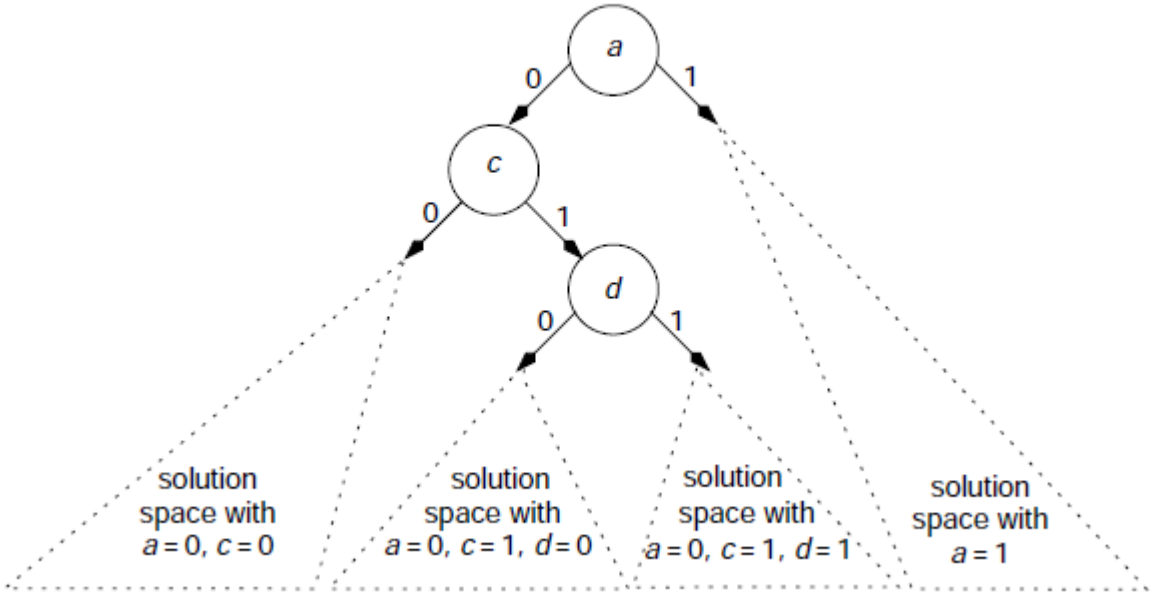
1: **while** a fault-effect of f has not propagated to a PO and all possible vector combinations have not been tried **do**
2: pick a vector, v, that has not been tried;
3: fault simulate v on the circuit C with fault f;
4: **end while**

ATPG Algorithm

- ATPG may make a wrong **decision** for specific logic value on PI.
- In this case, the decision should be altered (opposite logic value) on PI.
- The process of making decisions and and reversing decisions results in a **decision tree**.
- Each node in the decision tree represents a decision variable.
- If only two choices are possible for each decision variable, decision tree is a **binary tree**.

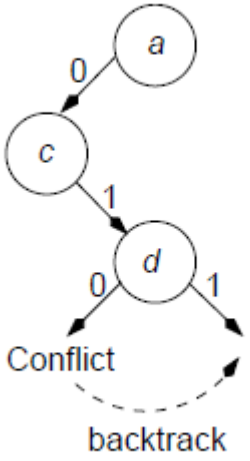
Decision Tree

- Example of decision tree.
- At each decision, the search space is halved.
- If a test vector exists, there must be a path along the decision tree that leads to the test vector.



Backtracking

- Whenever a conflict is detected, the search must return to some earlier point in decision process.
- The reversal of decision is called a **backtrack**.
- The easiest mechanism to keep track of decisions is to reverse the most recent decision made.
- When reversing any decision, the signal values implied by the assignment of the previous decision variables must be undone.



Basic ATPG Algorithm

- Given a target fault *g/v* in a fanout-free combinational circuit *C*, procedure to generate a vector for the fault (Algorithm 2).
- Functions JustifyFanoutFree() and PropagateFanoutFree() are recursive functions.

Algorithm 2 Basic Fanout Free ATPG (C, g/v)

1: initialize circuit by setting all values to X;
2: JustifyFanoutFree($C, g, \bar{v}$); /* excite the fault by justifying line g to $\bar{v}$ */
3: PropagateFanoutFree(C, g); /* propagate fault-effect from g to a PO */

Basic ATPG Algorithm

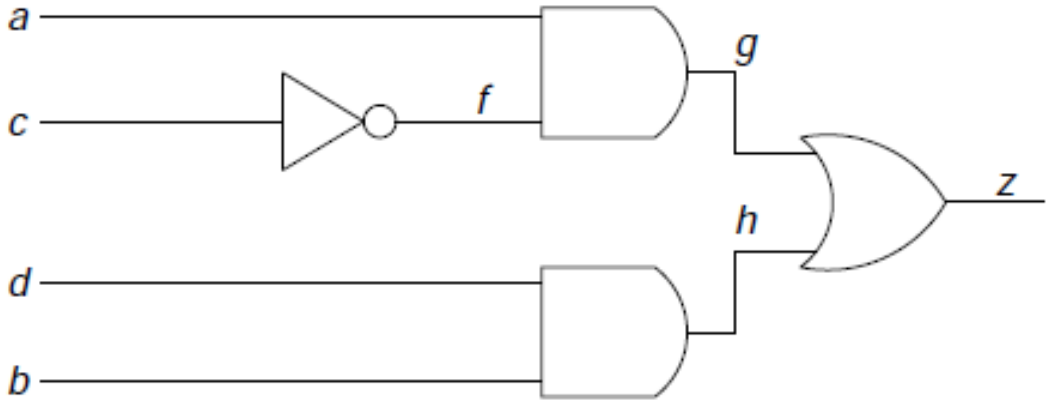
- JustifyFanoutFree(*g,v*) recursively justifies predecessor signal of *g* until all signals that should be justified are justified from the PI.

Algorithm 3 JustifyFanoutFree(C, g, v)

```
 1: g = v;
 2: if gate type of g == primary input then
 3:    return;
 4: else if gate type of g == AND gate then
 5:    if v == 1 then
 6:       for all inputs h of g do
 7:          JustifyFanoutFree(C, h, 1);
 8:       end for
 9:    else {v == 0}
10:       h = pick one input of g whose value == X;
11:       JustifyFanoutFree(C, h, 0);
12:    end if
13: else if gate type of g == OR gate then
14:    ...
15: end if
```

Justify Function

- Example circuit *C*, justify *g*=1

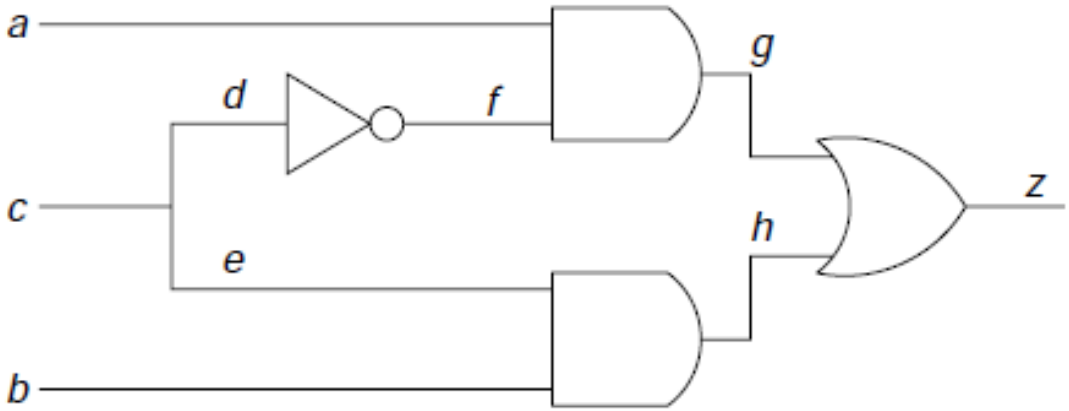


call #1: JustifyFanoutFree($C, g, 1$)
call #2: JustifyFanoutFree($C, a, 1$)
call #3: JustifyFanoutFree($C, f, 1$)
call #5: JustifyFanoutFree($C, c, 0$)

Input vector *abcd* = 1X0X justifies *g*=1

Justify Function

- Example circuit *C*, justify *g*=1

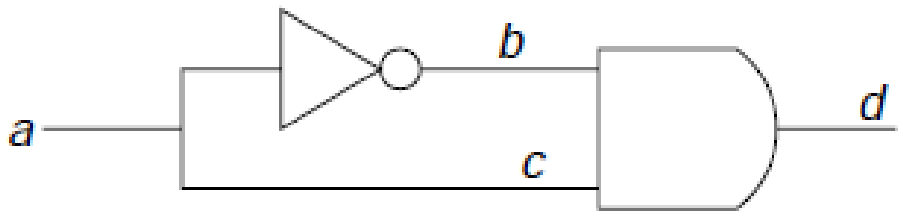


call #1: JustifyFanoutFree($C, g, 1$)
call #2: JustifyFanoutFree($C, a, 1$)
call #3: JustifyFanoutFree($C, f, 1$)
call #4: JustifyFanoutFree($C, d, 0$)
call #5: JustifyFanoutFree($C, c, 0$)

Input vector *abc* = 1X0 justifies *g*=1

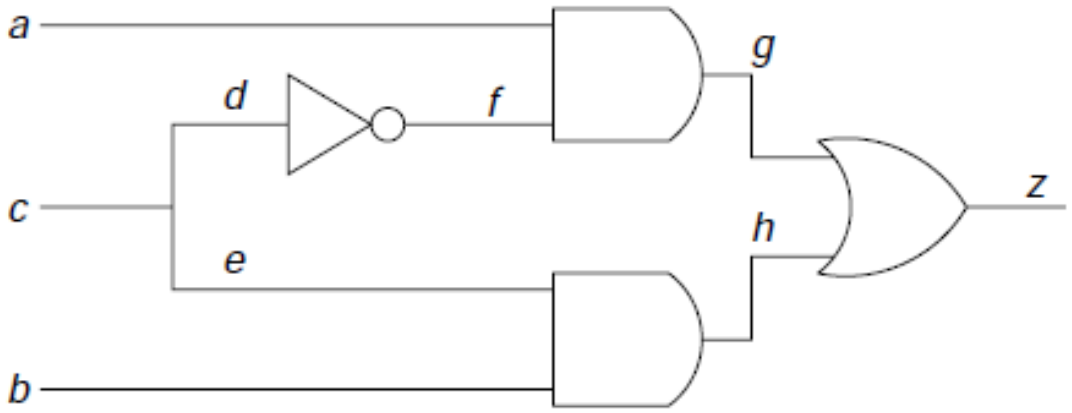
Justify Function

- In fanout-free circuit, JustifyFanoutFree() routine will always be able to set *g* to the desired value *v* and no conflict wil occur.
- This is no true for circuits with fanout branches -> two or more signals tracing back to the same fanout stem are correlated.
- For example, justifying *d*=1 is impossible (conflict on *a*)



Justify Function

- Example circuit *C*, justify *z*=0
- Due to fanout structure, choices for decisions are limited.



Propagate Function

- Once the fault is excited, the next step is to propagate the fault-effect to PO.
- PropagateFanoutFree() is also a recursive function.
- The fault-effect is propagated one gate at a time until it reaches PO.
- Example: propagate *D* at *g* to PO *z*.

call #1: PropagateFanoutFree(C, g)
call #2: JustifyFanoutFree(C, h, 0)
call #3: JustifyFanoutFree(C, b, 0)
call #4: PropagateFanoutFree(C, z)

Input vector *abc* = 100 justifies *g*=1

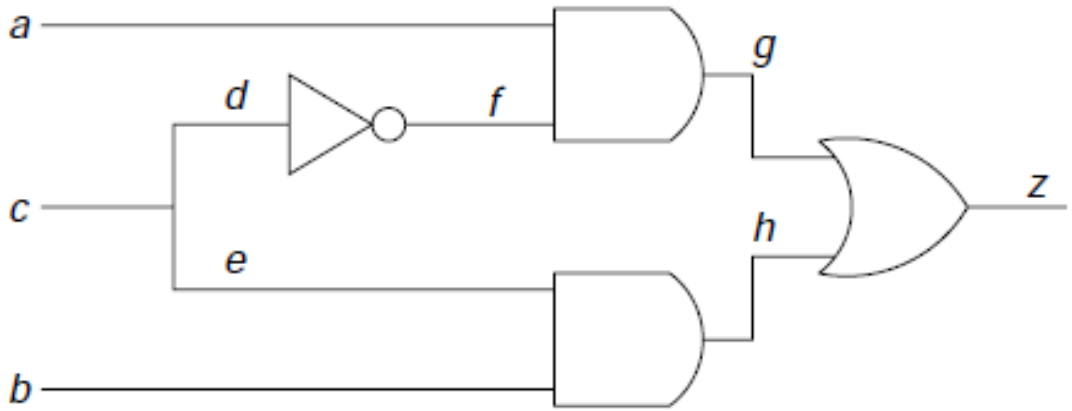
Propagate Function Algorithm

Algorithm 4 PropagateFanoutFree(C, g)

```
 1: if g has exactly one fanout then
 2:     h = fanout gate of g;
 3:     if none of the inputs of h has the value of X then
 4:         backtrack;
 5:     end if
 6: else {g has more than one fanout}
 7:     h = pick one fanout gate of g that is unjustified;
 8: end if
 9: if gate type of h == AND gate then
10:     for all inputs, j, of h, such that j ≠ g do
11:         if the value on j == X then
12:             JustifyFanoutFree(C, j, 1);
13:         end if
14:     end for
15: else if gate type of h == OR gate then
16:     for all inputs, j, of h, such that j ≠ g do
17:         if the value on j == X then
18:             JustifyFanoutFree(C, j, 0);
19:         end if
20:     end for
21: else if gate type of h == . . . gate then
22:     . . .
23: end if
24: PropagateFanoutFree(C, h);
```

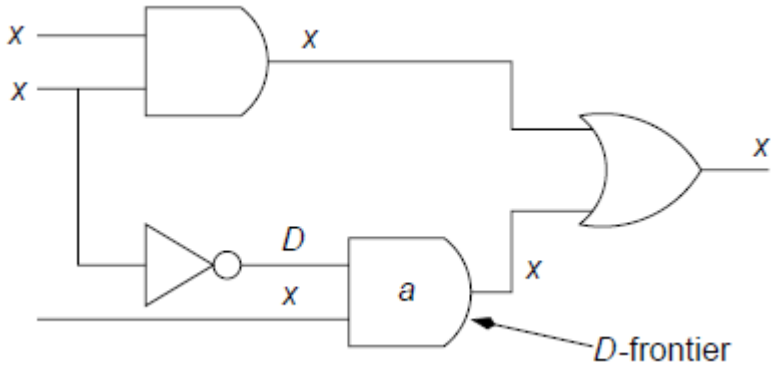
Justify Function, Decisions

- Example circuit *C*, fault *g/1*.
- For justification *g*=0, either *a* = 0 or *f* = 0 can be selected.
- ATPG should make a decision.
- **Testability measures** can be used as a guide to make good decisions (*a* = 0 should be better than *f* = 0)



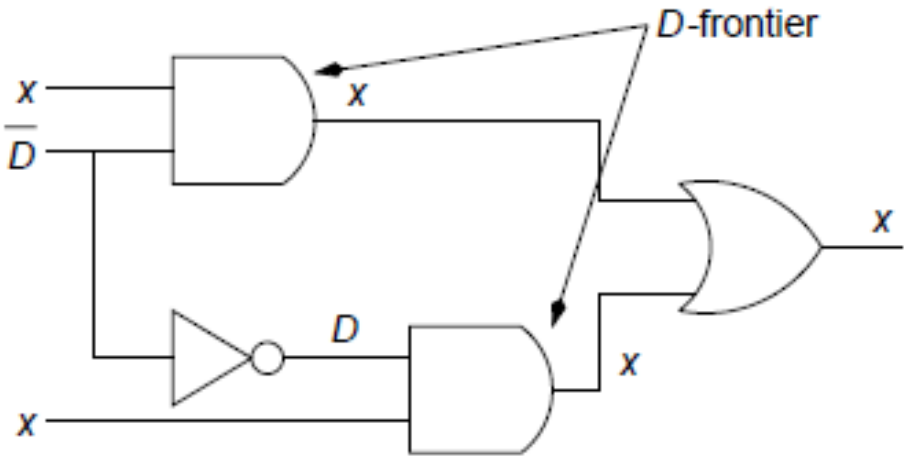
D Algorithm (Roth)

- The first complete ATPG algorithm -> if the fault is detectable, D algorithm will find a test vector.
- *D* or *D'* of the target fault is propagated to PO.
- ***D*-frontier**: all gates whose output value is *x* and fault-effect is at one or more of its inputs.
- Example of *D*-frontier with one gate:



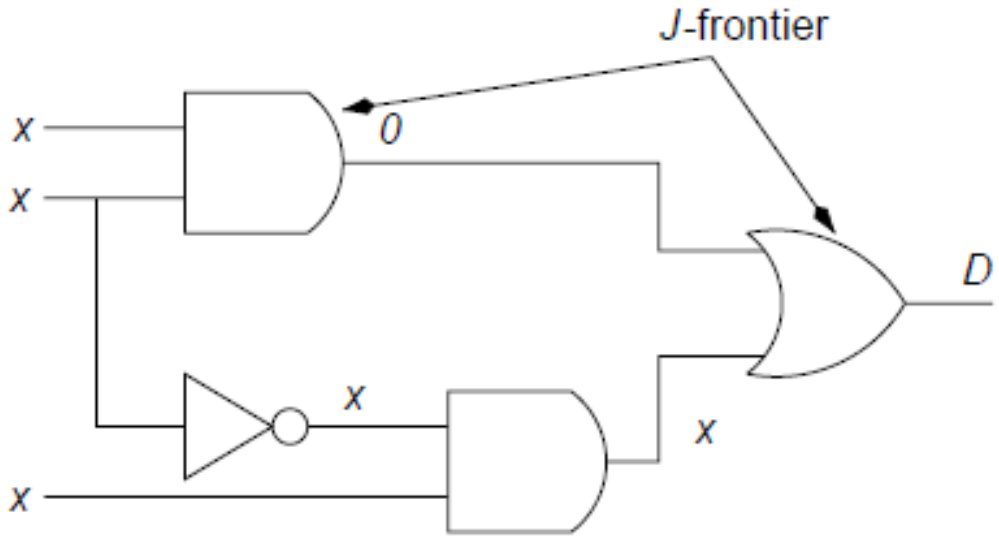
D Algorithm (Roth)

- If the *D*-frontier is empty, the fault can no longer be detected.
- Example of *D*-frontier with two gates:



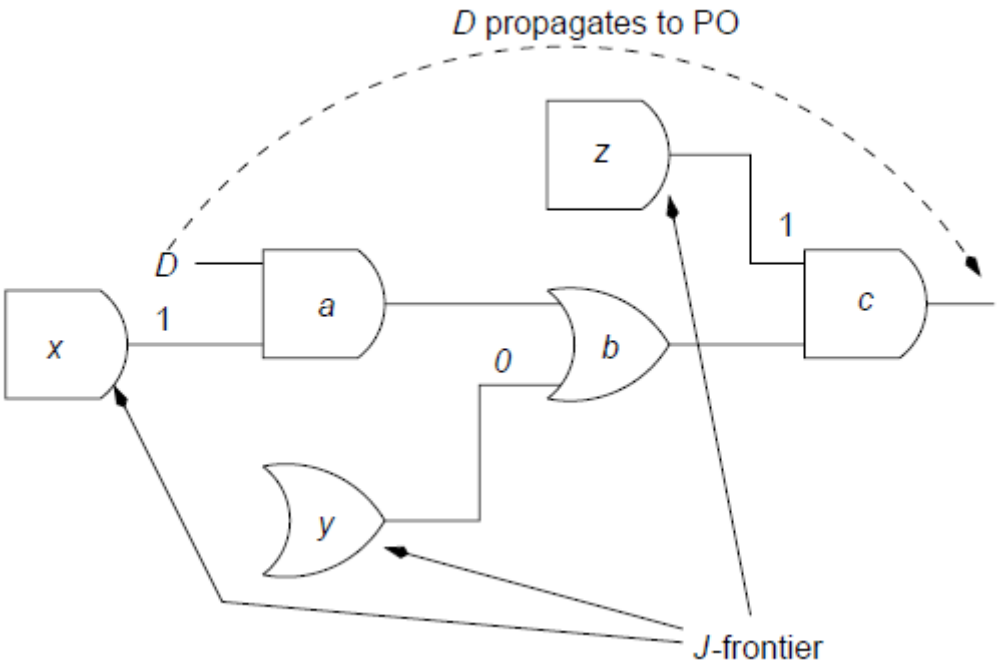
J-Frontier

- ***J*-frontier**: all gates whose output values are known (any value in 5-valued logic) but they are not yet justified by its inputs.
- Example of *J*-frontier:



D Algorithm, Example

- Propagation routine will set all side inputs of the path *a*-> *b*-> *c* to propagate the signal *D* to PO.
- These side input gates *x* , *y*, *z*, form the *J*-frontier as they are not yet justified.



D Algorithm, Pseudo-Code

- The overall procedure for the *D* algorithm is:

Algorithm 5 *D*-Algorithm(C, f)

1: initialize all gates to don't-cares;
2: set a fault-effect (D or $\overline{D}$) on line with fault f and insert it to the *D*-frontier;
3: *J*-frontier $= \phi$;
4: result = *D*-Alg-Recursion(C);
5: **if** result == success **then**
6: print out values at the primary inputs;
7: **else**
8: print fault f is untestable;
9: **end if**

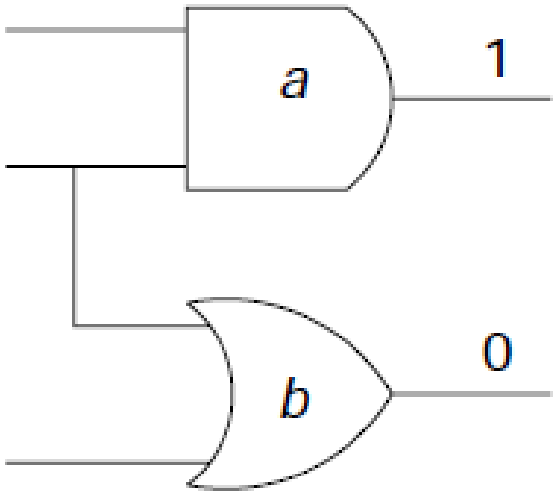
D Algorithm, *D*-Alg-Recursion

Algorithm 6 D-Alg-Recursion(*C*)

```
1: if there is a conflict in any assignment or D-frontier is ∅ then
2:    return failure;
3: end if
4: /* first propagate the fault-effect to a PO */
5: if no fault-effect has reached a PO then
6:    while not all gates in D-frontier has been tried do
7:       g = a gate in D-frontier that has not been tried;
8:       set all unassigned inputs of g to non-controlling value and add them to the J-frontier;
9:       result = D-Alg-Recursion(C);
10:      if result == success then
11:         return (success);
12:      end if
13:   end while
14:   return (failure);
15: end if {fault-effect has reached at least one PO}
16: if J-frontier is ∅ then
17:   return (success);
18: end if
19: g = a gate in J-frontier;
20: while g has not been justified do
21:   j = an unassigned input of g;
22:   set j = 1 and insert j = 1 to J-frontier;
23:   result = D-Alg-Recursion(C);
24:   if result == success then
25:      return (success);
26:   else try the other assignment
27:      set j = 0;
28:   end if
29: end while
30: return(failure);
```

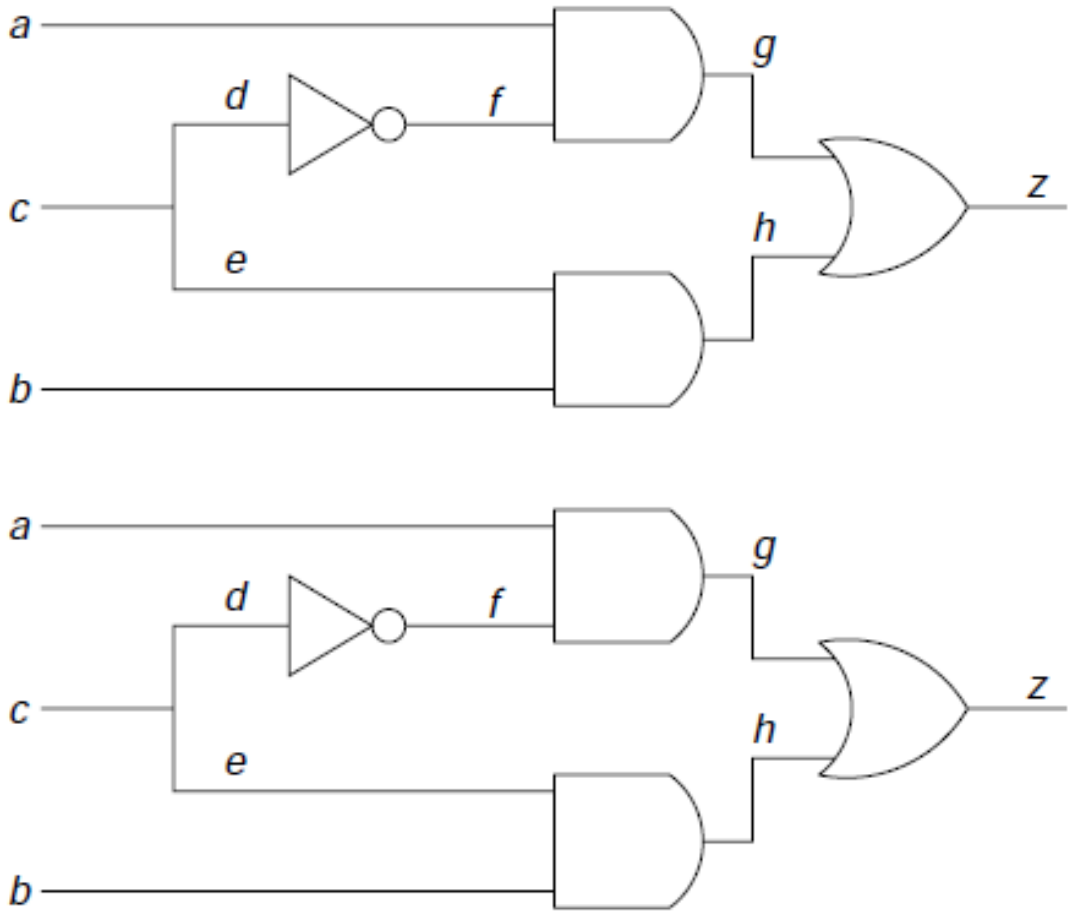
D Algorithm, Detecting Conflicts

- If there any conflicts, they should be detected asap.
- Example: justifying $a = 1$ and $b = 0$ is not possible.
- Detecting such conflicts helps to avoid future backtracks.



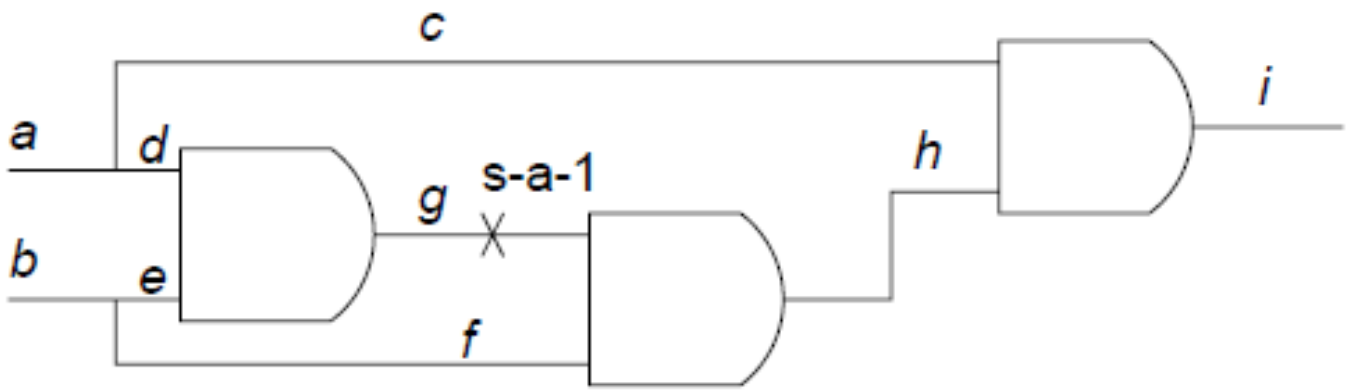
D Algorithm, Examples

- Consider faults *f*/0 and *f*/1 in the sample circuit.



D Algorithm, Examples

- Consider fault *g*/1 in the sample circuit.



PODEM

- In *D* algorithm, decisions can be made at every node
- However, the final result of every test generation step is the test vector – signals at PI.
- Number of Pis is generally much fewer than number of nodes in the circuits -> decisions are restricted only to Pis.

PODEM – Pseudo Code

- The overall procedure for the PODEM algorith is:

Algorithm 7 PODEM(C, f)

```
1: initialize all gates to don't-cares;
2: D-frontier = ∅;
3: result = PODEM-Recursion(C);
4: if result == success then
5:     print out values at the primary inputs;
6: else
7:     print fault f is untestable;
8: end if
```

PODEM – PODEM-Recursion

Algorithm 8 PODEM-Recursion(C)

```
 1: if fault-effect is observed at a PO then
 2:     return (success);
 3: end if
 4: (g, v) = getObjective(C);
 5: (pi, u) = backtrace(g, v);
 6: logicSimulate_and_imply(pi, u);
 7: result = PODEM-Recursion(C);
 8: if result == success then
 9:     return(success);
10: end if
11: /* backtrack */
12: logicSimulate_and_imply(pi, ū);
13: result = PODEM-Recursion(C);
14: if result == success then
15:     return(success);
16: end if
17: /* bad decision made at an earlier step, reset pi */
18: logicSimulate_and_imply(pi, x);
19: return(failure);
```

PODEM – getObjective

Algorithm 9 getObjective(C)

1: **if** fault is not excited **then**
2: return $(g, \bar{v})$;
3: **end if**
4: d = a gate in D-frontier;
5: g = an input of d whose value is x;
6: v = non-controlling value of d;
7: return (g, v);

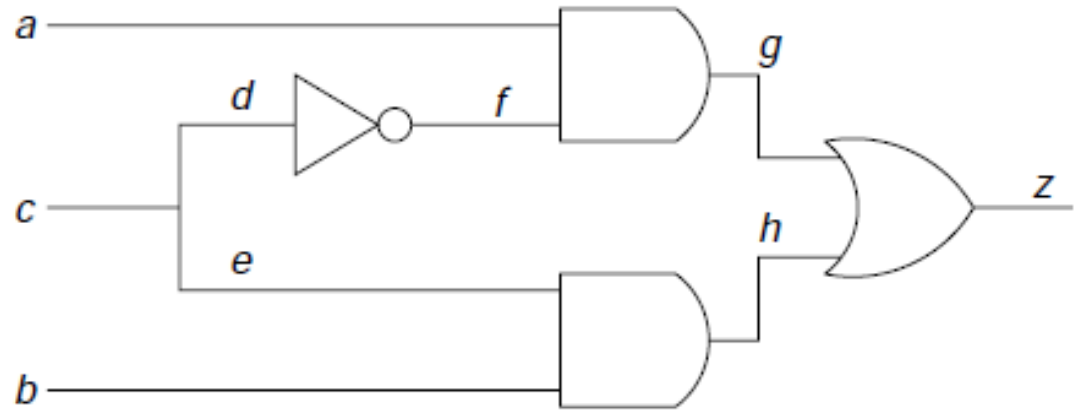
PODEM – Backtrace

Algorithm 10 backtrace(C)

```
 1: i = g;
 2: num_inversion = 0;
 3: while i ≠ primary input do
 4:    i = an input of i whose value is x;
 5:    if i is an inverted gate type then
 6:       num_inversion++;
 7:    end if
 8: end while
 9: if num_inversion == odd then
10:    v = v̄;
11: end if
12: return(i, v);
```

PODEM – Example

- Consider fault f/0:

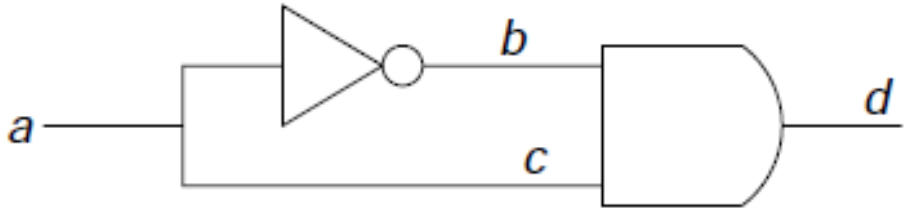


getObjective()	backtrace()	logicSim()	D-frontier
$f = 1$	$c = 0$	$d = 0, f = D,$ $e = 0, h = 0$	g
$a = 1$	$a = 1$	$g = D, z = D$	$f/0$ detected

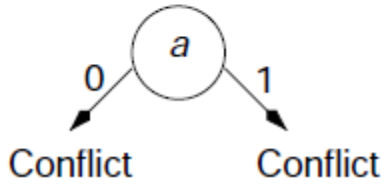
Test vector 1X0

PODEM – Example

- Consider fault b/0:



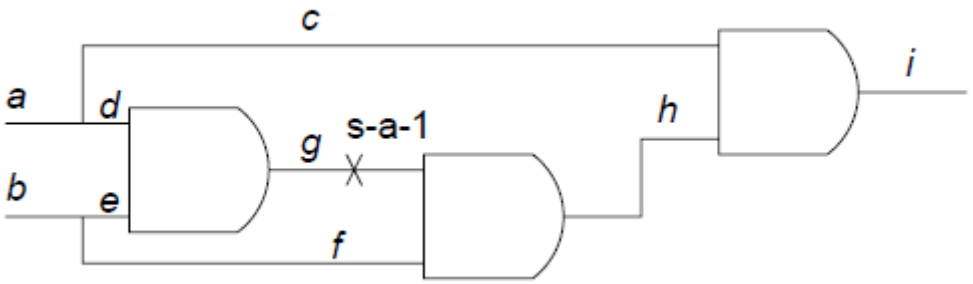
getObjective()	backtrace()	logicSim()	D-frontier
$b = 1$	$a = 0$	$b = 1, c = 0, d = 0$	$\emptyset$
$a = 1$ (reversal)	—	$b = 0, c = 1, d = 0$	$\emptyset$



Fault b/0 is undetectable.

PODEM – Example

- Consider fault g/1:



getObjective()	backtrace()	logicSim()	D-frontier
$g = 0$	$a = 0$	$g = D,\ c = 0$ $d = 0,\ i = 0$	h (but no X-path to PO)
$a = 1$ (reversal)	—	$c = 1,\ d = 1$	$\emptyset$

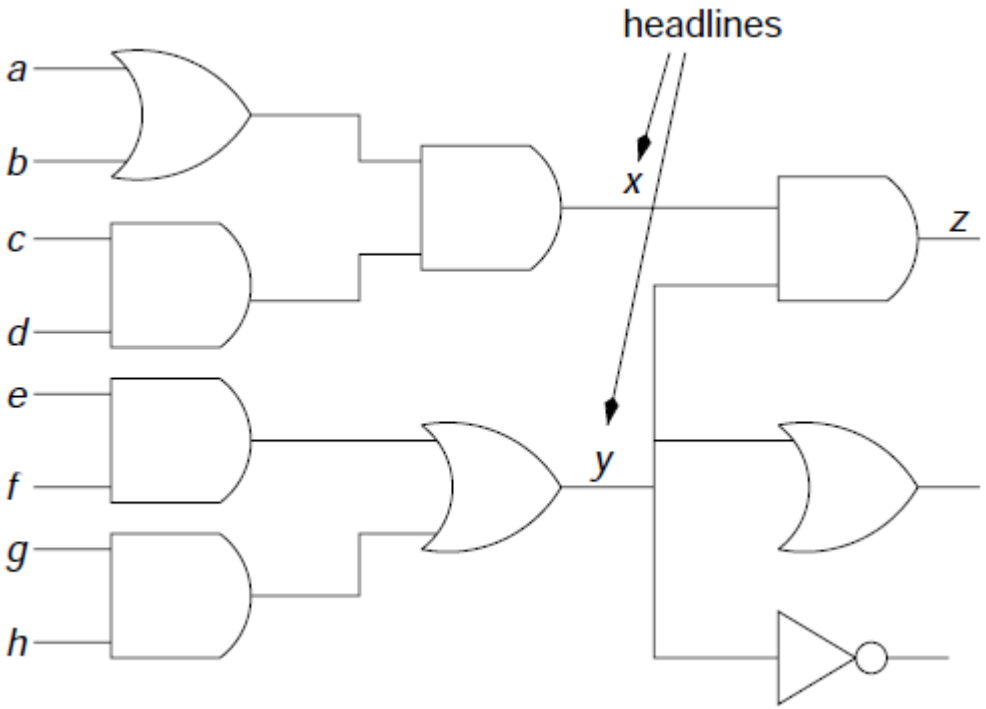
Fault g/1 is undetectable.

FAN

- PODEM can still make an excessive number of decisions.
- FAN (Fanout-Oriented TG) algorithm improves PODEM by reducing the number of decision points.
- FAN identifies headlines in the circuit, which are the output signals of fanout-free regions -> any value assignment on the headline can always be justified by its fanin cone.
- Backtrace function stops at PI (as in PODEM) or at headlines, thus reducing the number of decision points.

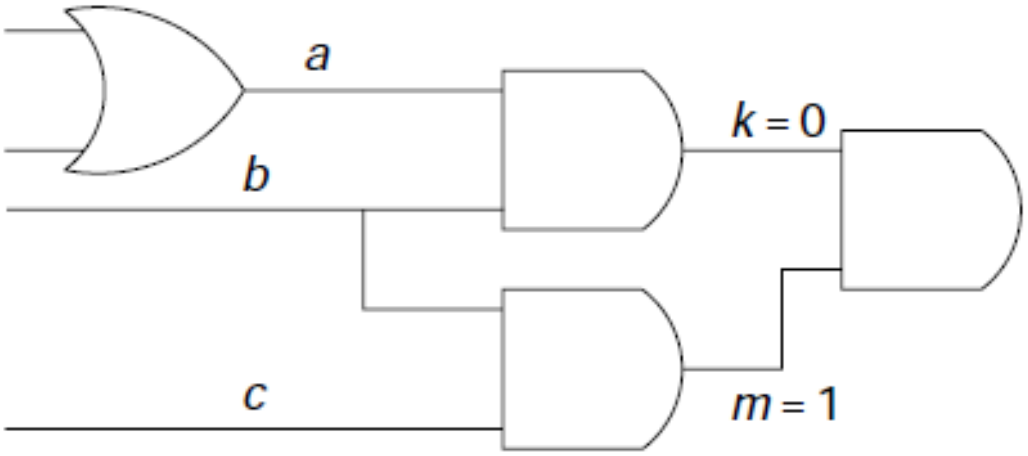
FAN - Example

- Consider objective z = 1.
- PODEM: a=1, c=1, d=1, e=1, f=1
- FAN: x=1, y=1



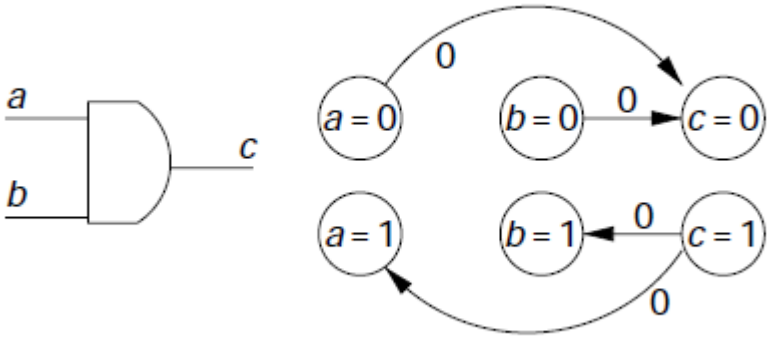
FAN – Multiple Objectives

- FAN considers simultaneously multiple objectives.
- Justify k=0
  - Selecting b=0 causes a conflict later with objective m=1



Logic Implications

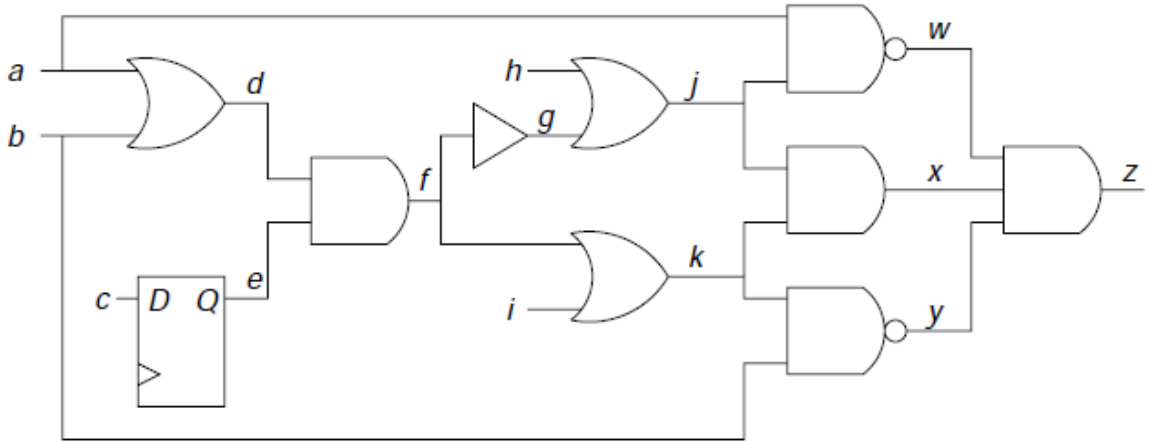
- Logic implications capture the effect of assigning logic values on other gates in order to make better decisions.
- Example:

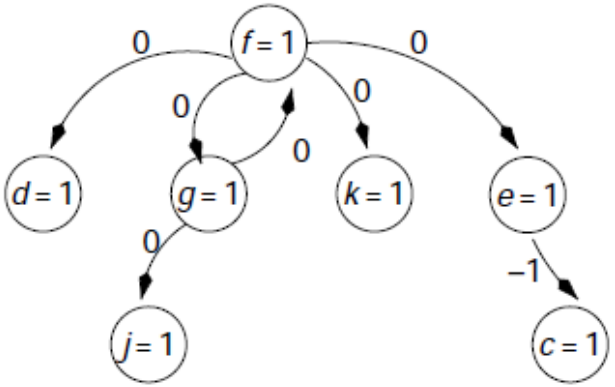


- They can be divided into
  - Static logic implications
  - Dynamic logic implications

Static Logic Implications

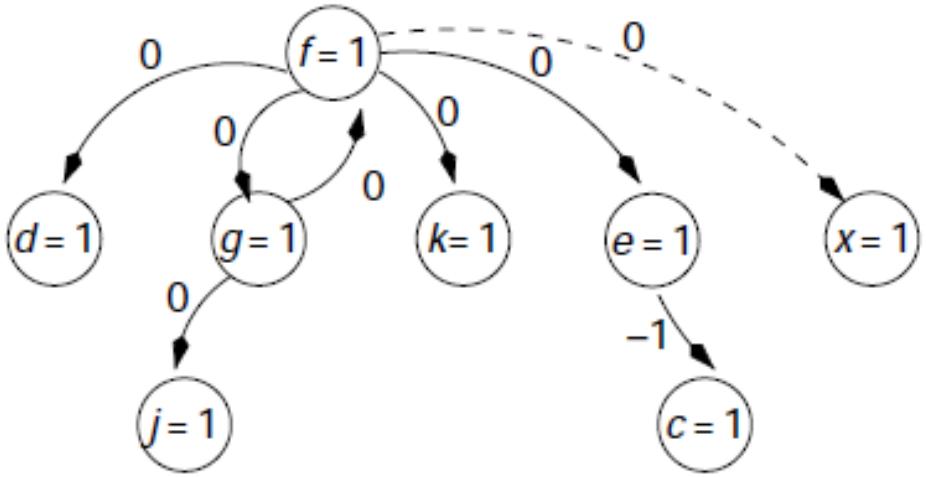
- Direct implications
- Example, implication of logic value $f = 1$.





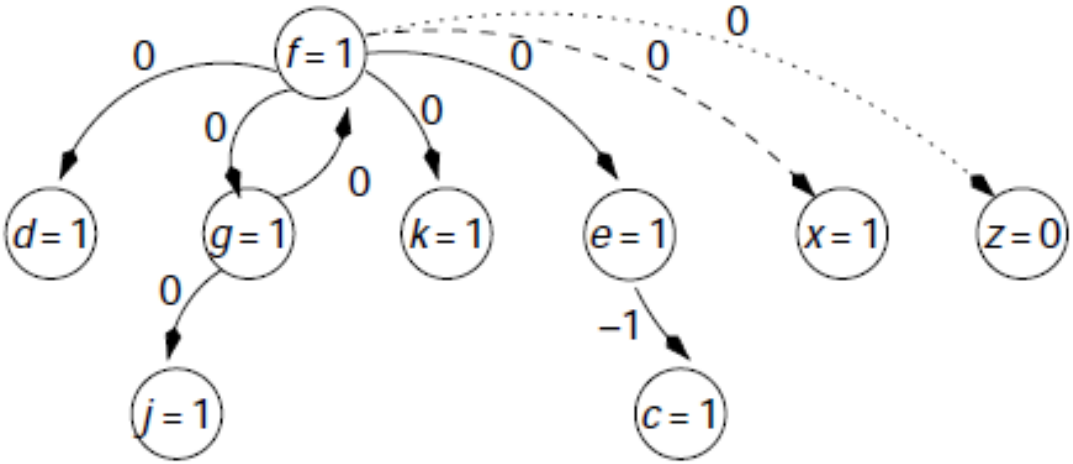
Static Logic Implications

- Indirect implications
- Can be computed by performing logic simulation on the current set of logic implications.
- Example, implication of logic value $f = 1$.



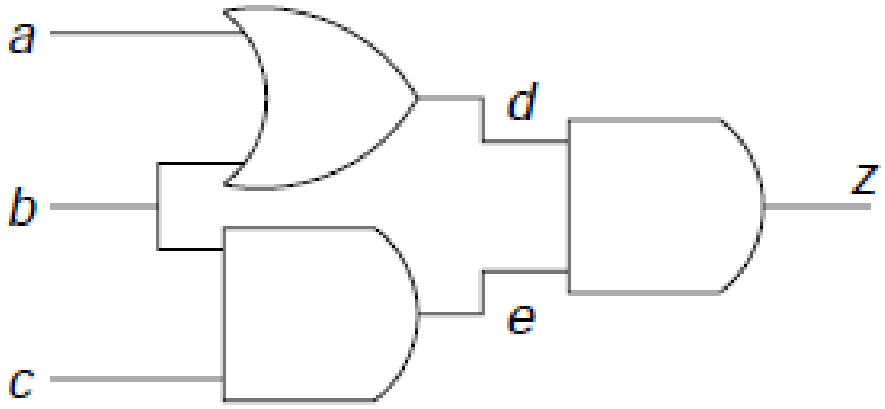
Static Logic Implications

- Extended backward implications
- Can be computed by performing logic simulation on the current set of logic implications.
- Example, implication of logic value $f = 1$.



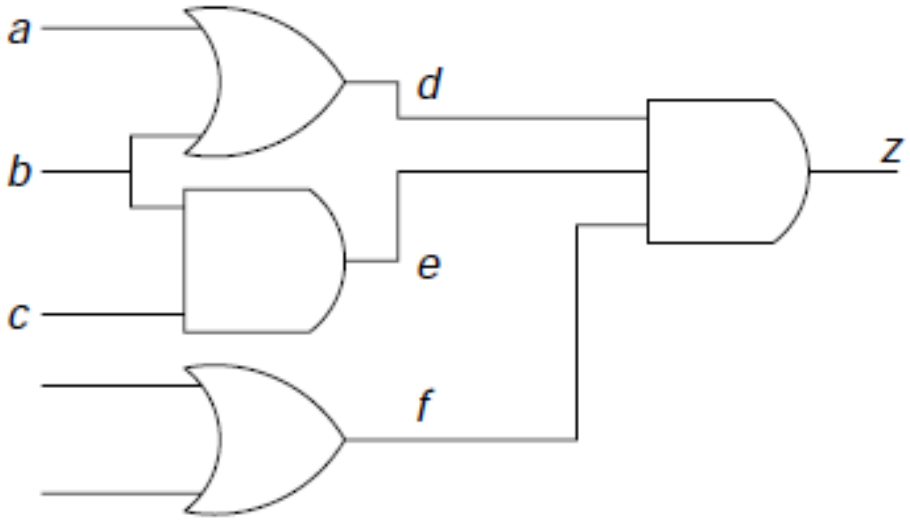
Dynamic Logic Implications

- Static logic implications are computed once for the entire circuit, dynamic implications are performed during the ATPG process.
- Example, implication of logic value for $z = 0$ by $c = 1$.
- d=0 -> {a=0, b=0} e=0 -> {b=0}
- Dynamic implication for $z = 0$ by $c = 1$ -> {b=0}



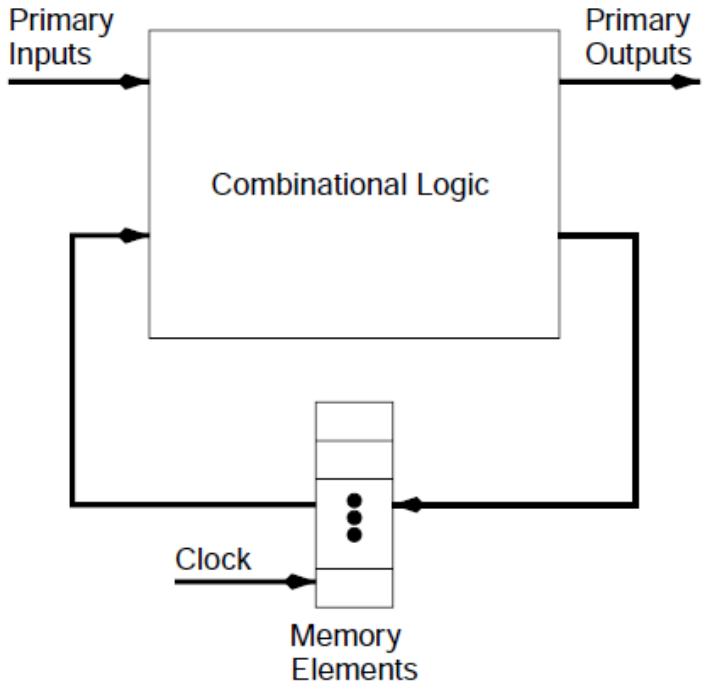
Dynamic Logic Implications

- Dynamic logic implications can be used also for signals with fault-effect.
- Example, fault signal is on node *b*. In order to propagate fault signal to PO *z*, necessary condition is *f*=1.



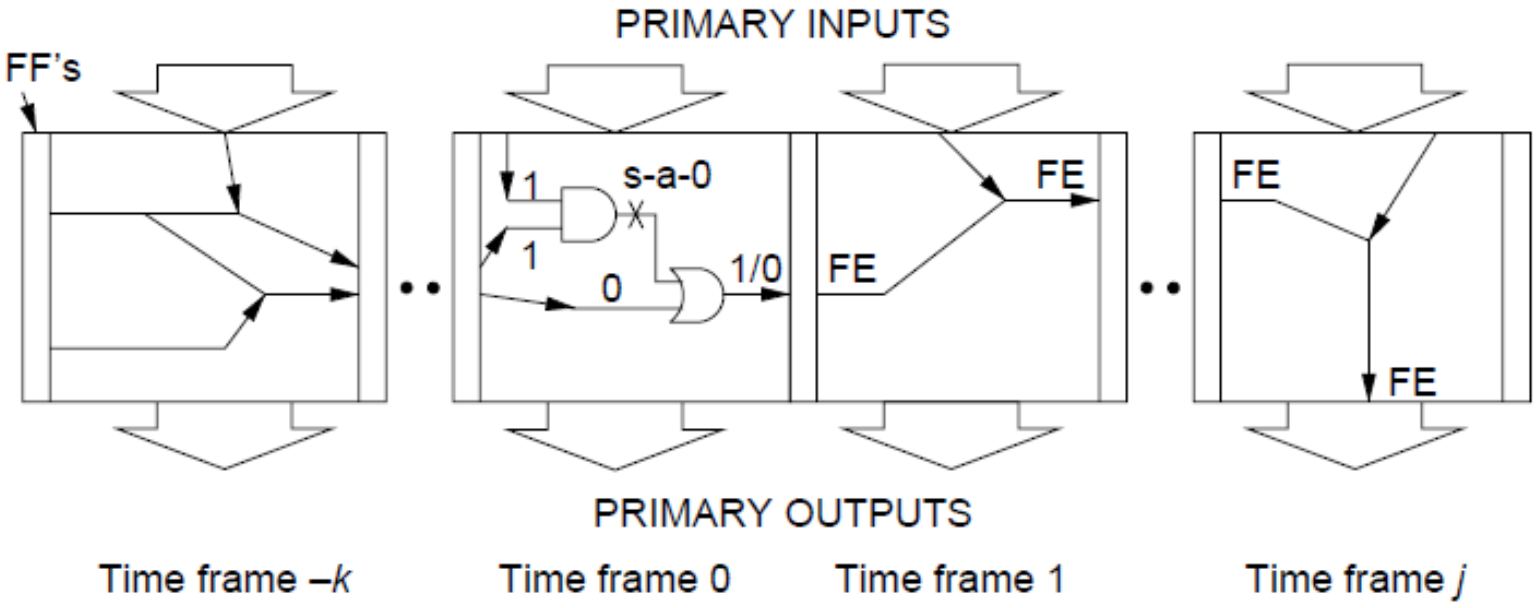
Test Generation for Sequential Circuits

- One test vector may be insufficient to detect the target fault since the exication and propagation conditions may necessitate some of the flip-flop values.
- General model of sequential circuit:



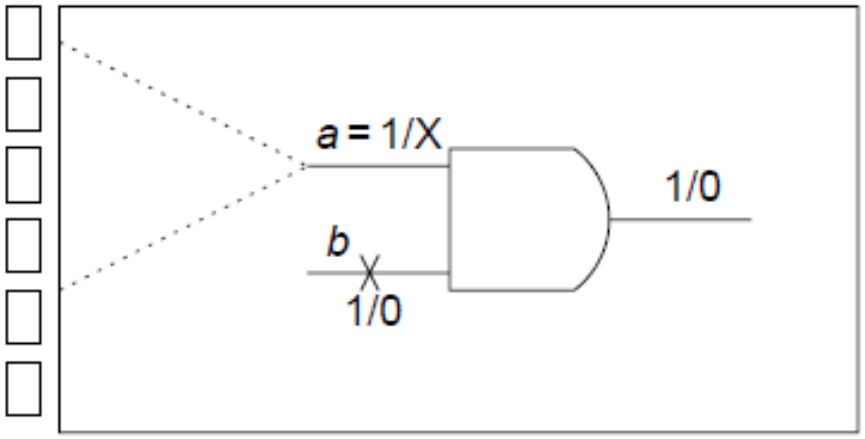
Time Frame Expansion

- Method for transforming sequential circuit into combintional circuit over several time frames (iterative logic array).
- Target fault is present in every time frame.



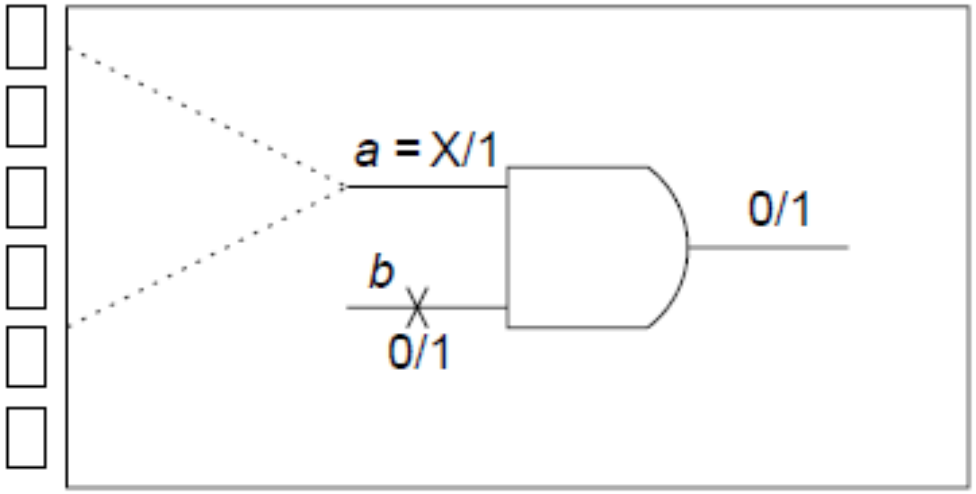
5-Valued Algebra is Insufficient

- Consider the target fault b/0.
- Values *a*=1/0 or *a*=1/1 propagates the fault signal over the AND gate.
- This can be written as *a*=1/X.



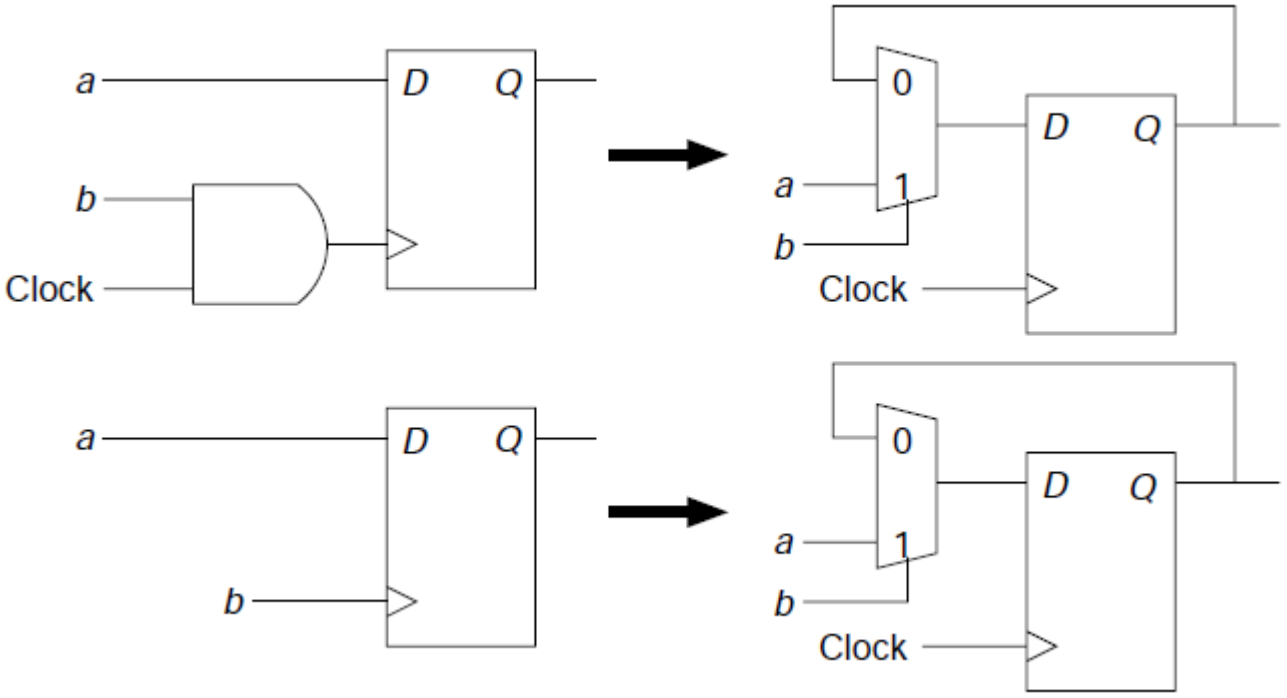
5-Valued Algebra is Insufficient

- Consider the target fault b/1.
- Values *a*=1/1 or *a*=0/1 propagates the fault signal over the AND gate.
- This can be written as *a*=X/1.
- Additional values are therefore: **1/X**, **0/X**, **X/1**, **X/0**, all together **9 values (9-Valued Algebra)**



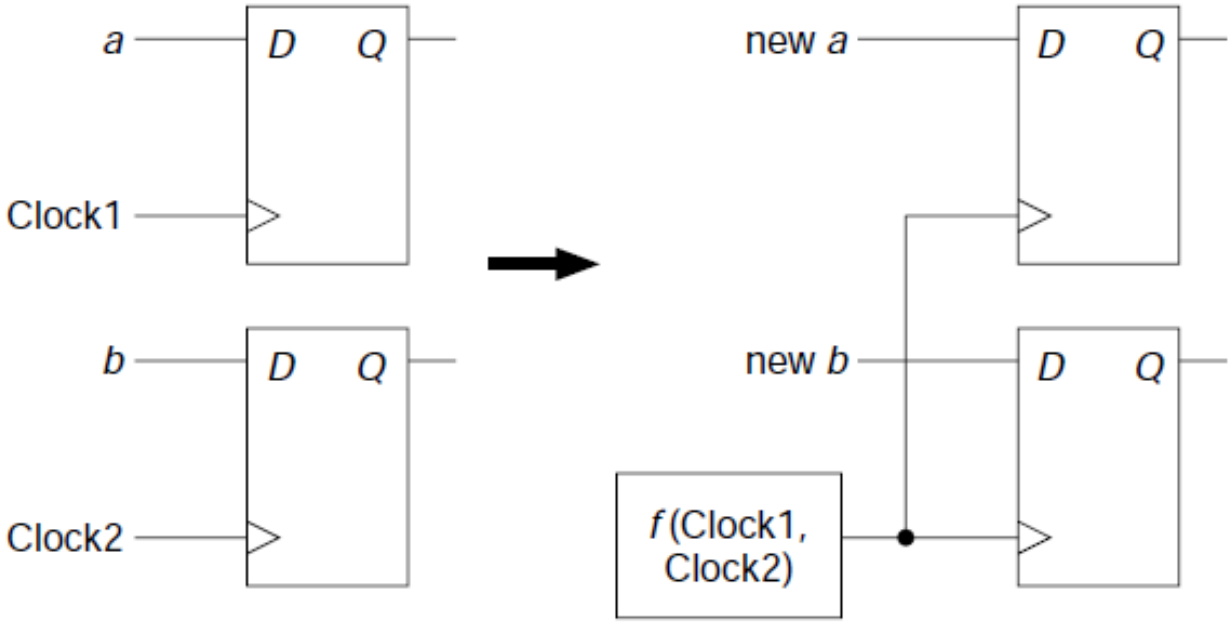
Gated Clocks

- Gated clocks are used for power saving.
- Tranformation are used to ease ATPG process.



Multiple Clocks

- Multiple clocks benefit performance and power as circuit blocks are partitioned to different clock domains.
- Example of transformation:



Other issues in Deterministic TPG

- Untestable fault identification
- Multiple-line conflict analysis
- Genetic algorithms
- Testing for bridging and delay faults
- Testing of acyclic sequential circuits
- Using testing for logic and power optimization