Chapter 4

Test Generation

What is this chapter about?

- Introduce the basic concepts of ATPG
- Focus on a number of combinational and sequential ATPG techniques
  - Deterministic ATPG and simulation-based ATPG
  - Fast untestable fault identification
- ATPG for various fault models

Test Generation

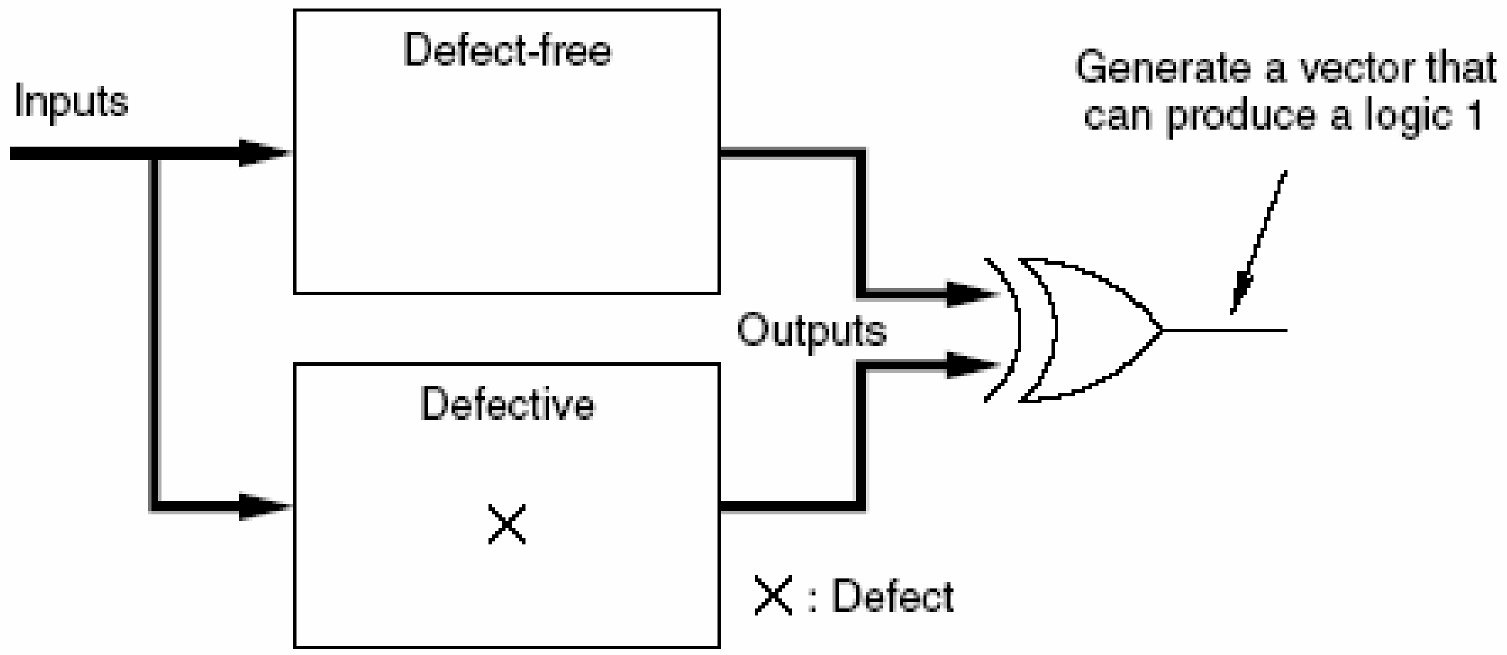
- ❑ Introduction
- ❑ Random Test Generation
- ❑ Theoretical Foundations
- ❑ Deterministic Combinational ATPG
- ❑ Deterministic Sequential ATPG
- ❑ Untestable Fault Identification
- ❑ Simulation-based ATPG
- ❑ ATPG for Delay and Bridge Faults
- ❑ Other Topics in Test Generation
- ❑ Concluding Remarks

Introduction

- Test generation is the bread-and-butter in VLSI Testing
  - Efficient and powerful ATPG can alleviate high costs of DFT
  - Goal: generation of a small set of effective vectors at a low computational cost
- ATPG is a very challenging task
  - Exponential complexity
  - Circuit sizes continue to increase (Moore's Law)
    - Aggravate the complexity problem further
  - Higher clock frequencies
    - Need to test for both structural and delay defects

Conceptual View of ATPG

- Generate an input vector that can distinguish the defect-free circuit from the hypothetically defective one

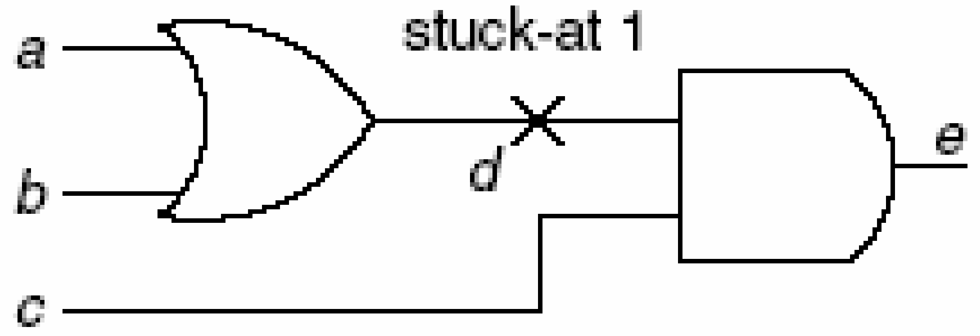


Fault Models

- Instead of targeting specific defects, fault models are used to capture the logical effect of the underlying defect
- Fault models considered in this chapter:
  - Stuck-at fault
  - Bridging fault
  - Transition fault
  - Path-delay fault

Simple illustration of ATPG

- Consider the fault d/1 in the defective circuit
- Need to distinguish the output of the defective circuit from the defect-free circuit
- Need: set d=0 in the defect-free circuit
- Need: propagate effect of fault to output
- Vector: abc=001 (output = 0/1)



A Typical ATPG System

- Given a circuit and a fault model
    - Repeat
    - Generate a test for each undetected fault
    - Drop all other faults detected by the test using a fault simulator
    - Until all faults have been considered
- Note 1: a fault may be untestable, in which no test would be generated
- Note 2: an ATPG may abort on a fault if the resources needed exceed a preset limit

Random Test Generation

- Simplest form of test generation
  - *N* tests are randomly generated
- Level of confidence on random test set *T*
  - The probability that *T* can detect all stuck-at faults in the given circuit
  - Quality of a random test set highly depends on the underlying circuit
  - Some circuits have many random-resistant faults

Weighted Random Test Generation

- Bias input probabilities to target random resistant faults
- Consider an 8-input AND gate
  - Without biasing input probabilities, the prob of generating a logic 1 at the gate output = $(0.5)^8 = 0.004$
  - If we bias the inputs to 0.75, then the prob of generating a logic 1 at the gate output = $(0.75)^8 = 0.100$
- Obtaining an optimal set of input probabilities a difficult task
- Goal: increase the signal probabilities of hard-to-test regions

Probability of Fault Detection

- Given a circuit with *n* inputs
- Let T_f be the set of vectors that can detect fault *f*
- Then $d_f = \frac{T_f}{2^n}$ is the prob that *f* can be detected by a random vector
- Let $e_f = 1 - d_f$ be the prob that a random vector cannot detect *f*

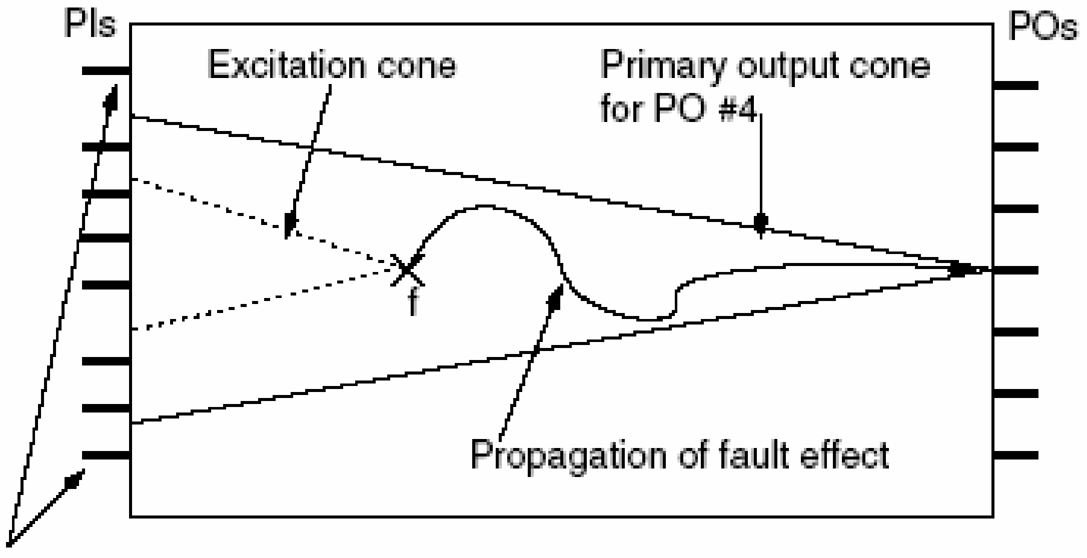
Prob of Fault Detection (Cont.)

- Then, $e_f^N = (1 - d_f)^N$ is the prob that *N* random vectors do not detect *f*
- Thus, the prob that at least one out of *N* random vectors can detect *f* is

$$1 - (1 - d_f)^N$$

Minimum Detection Probability

- The min detection prob of any detectable fault actually does not depend on *n*, the num of PIs
- Instead, it depends on the largest primary-output cone that it is in
- This is because any detectable fault must be excited and sensitized to a primary output



Inputs outside of PO cone are not needed for detection of fault f

Lemma 1

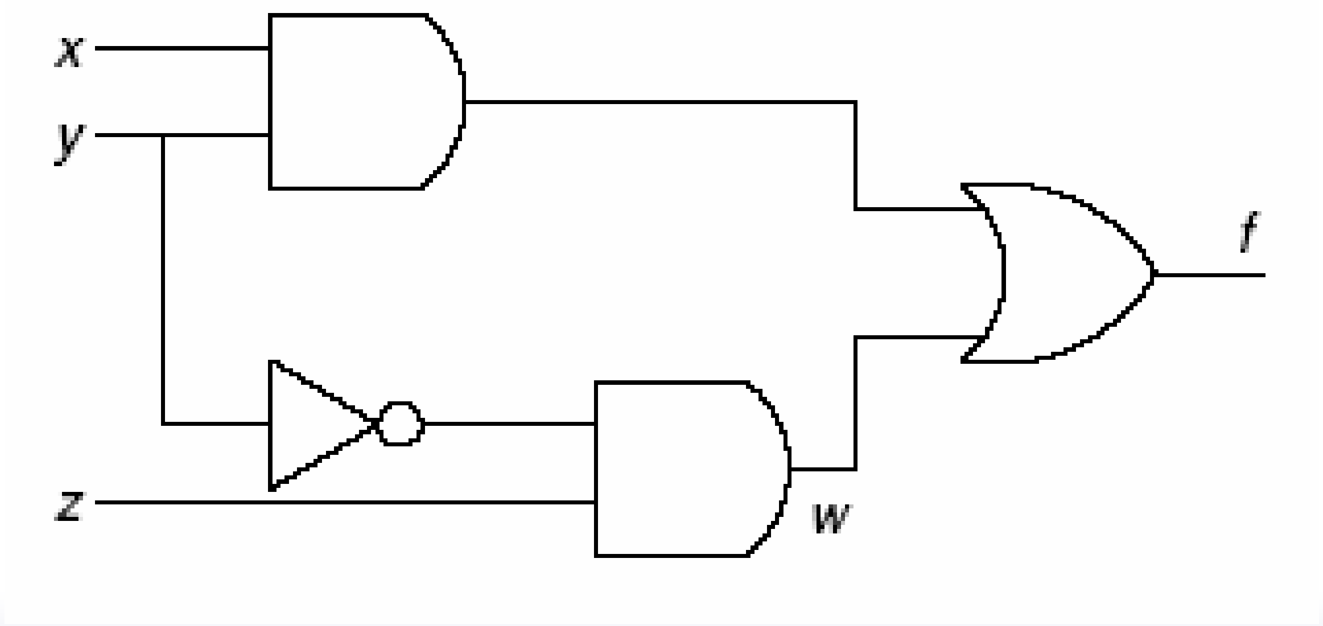
- In a combinational circuit with multiple outputs, let n_{max} be the number of primary inputs that can lead to a primary output. Then, the detection probability for the most difficult detectable fault, d_{min}, is: $d_{\min} \geq (0.5)^{n_{\max}}$

Exhaustive Test Generation

- Exhaustive Testing
  - Apply 2^n patterns to an *n*-input combinational *circuit under test* (CUT)
  - Guarantees all detectable faults in the combinational circuits are detected
  - Test time maybe be prohibitively long if the number of inputs is large
  - Feasible only for small circuits
- Pseudo-exhaustive Testing
  - Partition circuit into respective cones
  - Apply exhaustive testing only to each cone
  - Still guarantees to detect every detectable fault based on Lemma 1

Theoretical Foundations: Boolean Difference

- The function for the circuit is $f = xy + \bar{y}z$
- Let the target fault be *y/0,* then the function for the faulty circuit is *f' = f(y=0)*
- Goal of test generation: *find a vector that makes f XOR f' = 1*



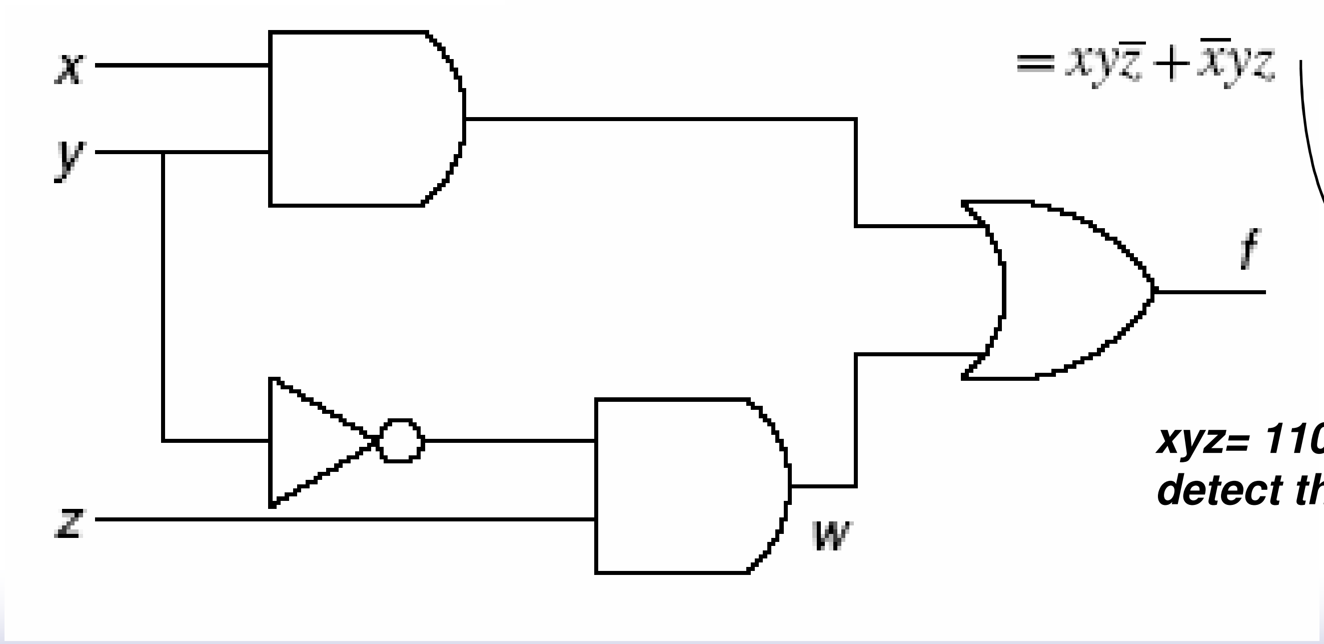
Boolean Difference Continued

- *f XOR f' = 1* iff *f* and *f'* result in opposing logic values
- Thus, any vector that can set *f XOR f' = 1* is able to produce opposing values at the outputs of the fault-free and faulty circuits respectively
- Definition: $$\frac{df}{dy} = f(y=1) \oplus f(y=0).$$

Boolean Difference Example

- To excite the fault *y/0, y=1*
- *Thus,*

$$\begin{aligned} y \cdot f(y=1) \oplus f(y=0) &= y \cdot (x \oplus z) \\ &= y \cdot (x\bar{z} + \bar{x}z) \\ &= xy\bar{z} + \bar{x}yz \end{aligned}$$



xyz= 110 or 011 can detect the fault

Another Example

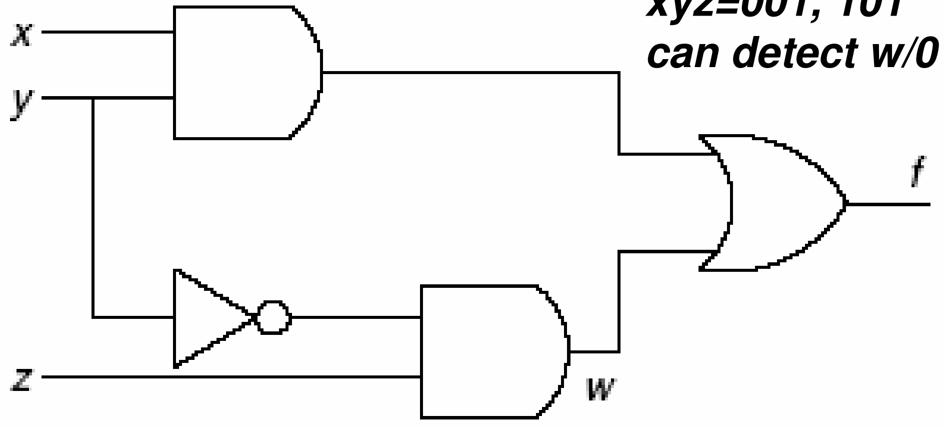
- Let target fault be *w/0*

$$w \cdot \frac{df}{dw} = 1$$
$$\Rightarrow \quad w \cdot (f(w=1) \oplus f(w=0)) = 1$$
$$\Rightarrow \quad w \cdot (1 \oplus xy) = 1$$
$$\Rightarrow \quad w \cdot (\overline{xy}) = 1$$
$$\Rightarrow \quad w \cdot (\overline{x} + \overline{y}) = 1$$
$$\Rightarrow \quad w\overline{x} + w\overline{y} = 1$$

But: $w = \overline{y} \cdot z$

$$w \cdot \overline{x} + w \cdot \overline{y} = 1$$
$$\Rightarrow \quad \overline{y} \cdot z \cdot \overline{x} + \overline{y} \cdot z \cdot \overline{y} = 1$$
$$\Rightarrow \quad \overline{x} \cdot \overline{y} \cdot z + \overline{y} \cdot z = 1$$
$$\Rightarrow \quad \overline{y} \cdot z = 1$$

xyz=001, 101 can detect w/0

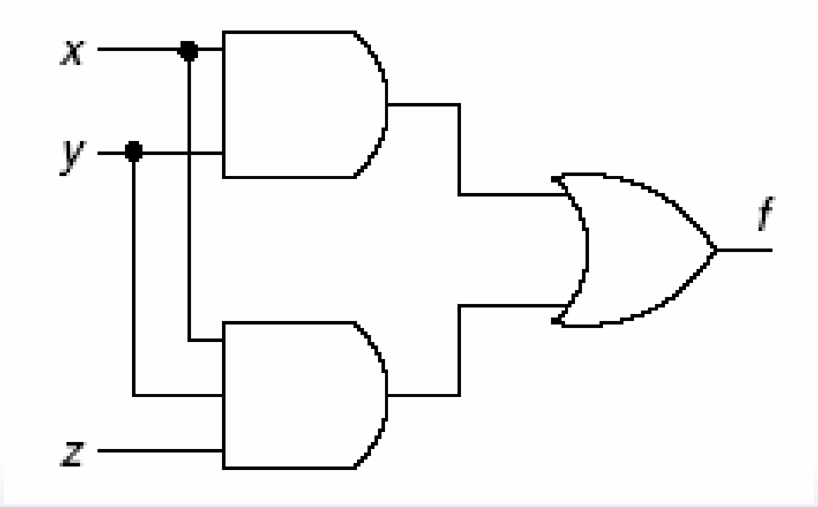


A Third Example

- Fault: *z/0*

$$
\begin{aligned}
& z \cdot \frac{df}{dz} = 1 \\
\Rightarrow \quad & z \cdot (f(z=1) \oplus f(z=0)) = 1 \\
\Rightarrow \quad & z \cdot (xy \oplus xy) = 1 \\
\Rightarrow \quad & z \cdot 0 = 1 \\
\Rightarrow \quad & \text{UNSATISFIABLE}
\end{aligned}
$$

This fault is untestable!



Wrap Up on Boolean Difference

- Given a circuit with output *f* and fault α/v;
- The set of vectors that can detect this fault includes all vectors that satisfy

$$(\alpha = \overline{v}) \cdot \frac{df}{d\alpha} = 1$$

Deterministic ATPG

- In general, we don’t need an entire set of vectors that can detect the target fault
- Instead, we just want to compute one vector quickly
- Rather than using Boolean Difference that can obtain all vectors
  - Simply use a branch-and-bound search to find one vector quickly
- Deterministic ATPG has two main goals
  - Excite the target fault
  - Propagate the corresponding fault effect to an output

5-valued Algebra for Comb. Circuits

- Instead of using two circuits (fault-free and the faulty)
  - We will solve the ATPG problem on one single circuit
- To do so, every signal value must be able to capture fault-free and faulty values simultaneously
- 5-Value Algebra: 0, 1, X, D, D-bar
  - D: 1/0
  - D-bar: 0/1

Boolean Operators on 5-Valued Algebra

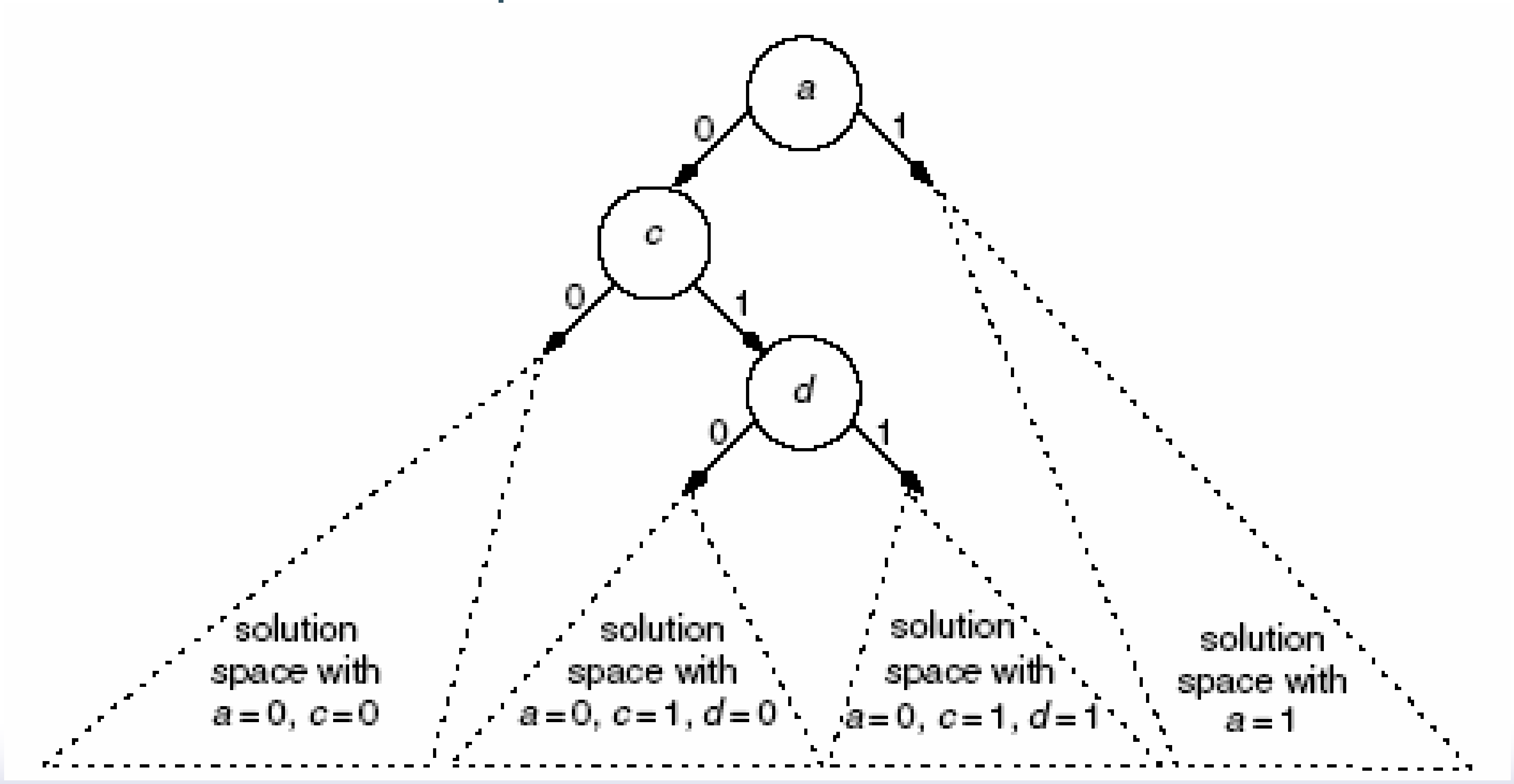
AND	0	1	D	$\overline{D}$	X
0	0	0	0	0	0
1	0	1	D	$\overline{D}$	X
D	0	D	D	0	X
$\overline{D}$	0	$\overline{D}$	0	$\overline{D}$	X
X	0	X	X	X	X

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

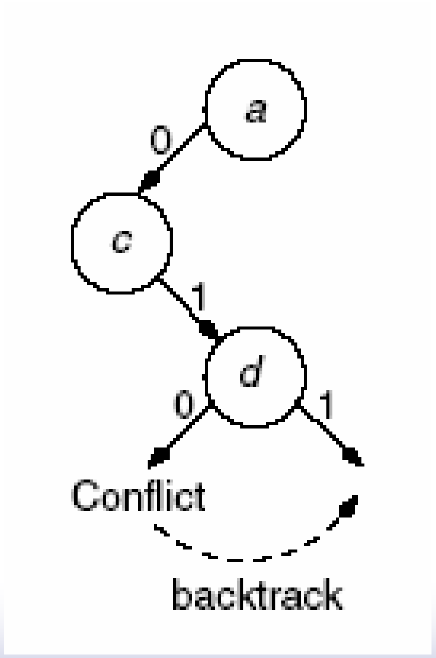
Decision Tree for Branch-and-Bound Search

- The ATPG systematically and implicitly searches the entire search space


a
0
1
c
0
1
d
0
1
solution space with a = 0, c = 0
solution space with a = 0, c = 1, d = 0
solution space with a = 0, c = 1, d = 1
solution space with a = 1

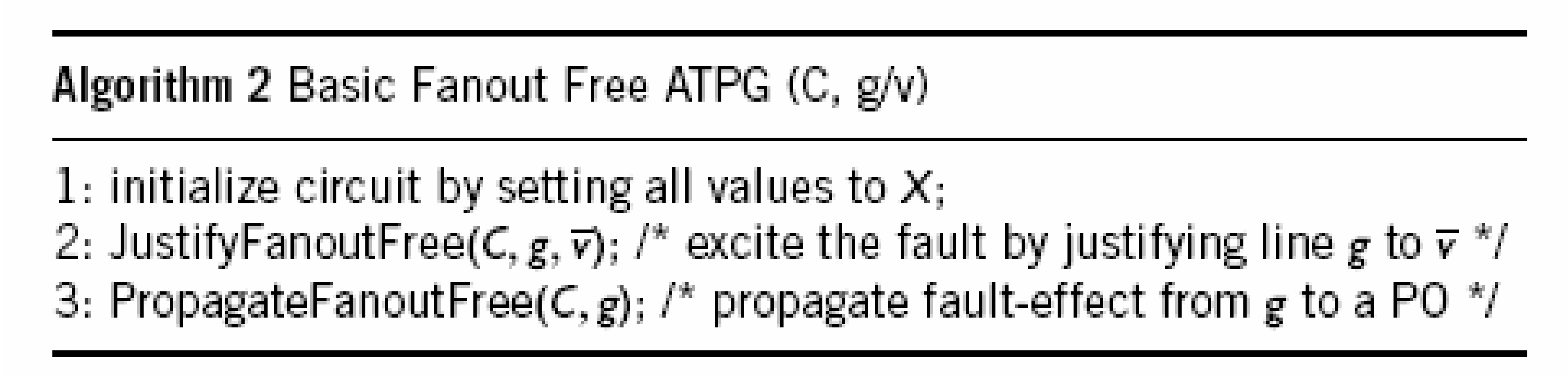
Backtracking

- The ATPG searches one branch at a time
- Whenever a conflict (e.g., all D's disappeared) arises, must backtrack on previous decisions



If d=1 also causes a conflict, backtrack to c=0

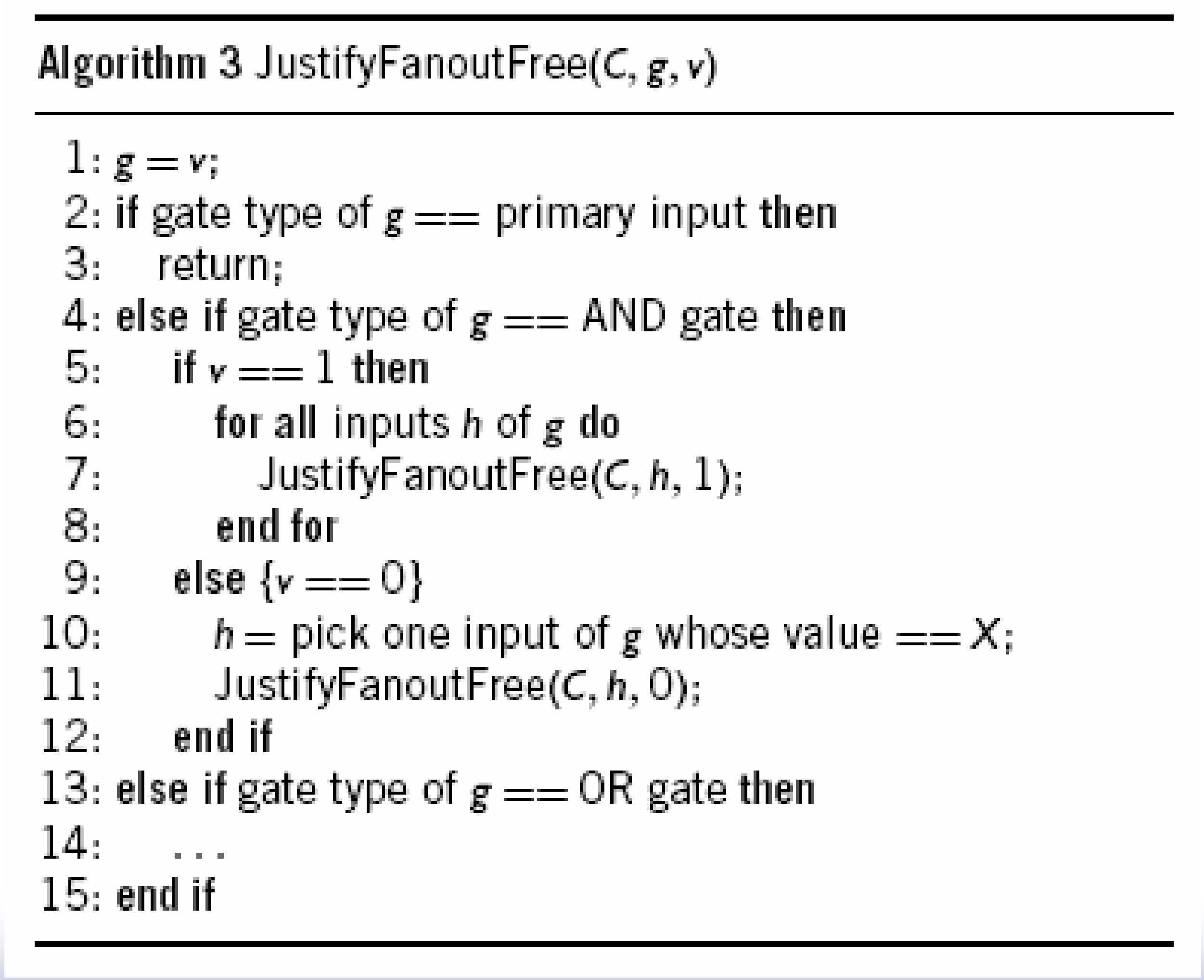
Basic ATPG for Fanout-Free Circuits



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

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

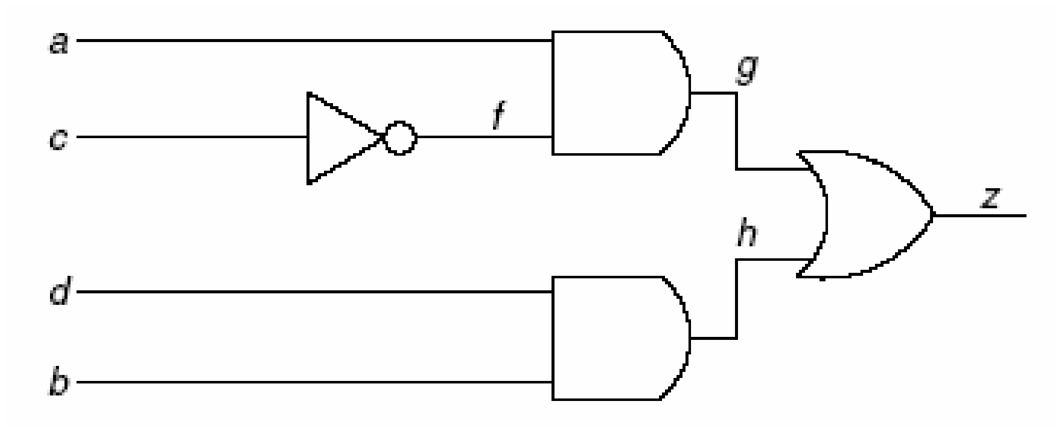
The Justify Routine



Algorithm 3 JustifyFanoutFree(C, g, v)

```
$g = v$;
if gate type of $g ==$ primary input then
    return;
else if gate type of $g ==$ AND gate then
    if $v == 1$ then
        for all inputs $h$ of $g$ do
            JustifyFanoutFree($C, h, 1$);
        end for
    else {$v == 0$}
        $h =$ pick one input of $g$ whose value $== X$;
        JustifyFanoutFree($C, h, 0$);
    end if
else if gate type of $g ==$ OR gate then
    ...
end if
```

Example

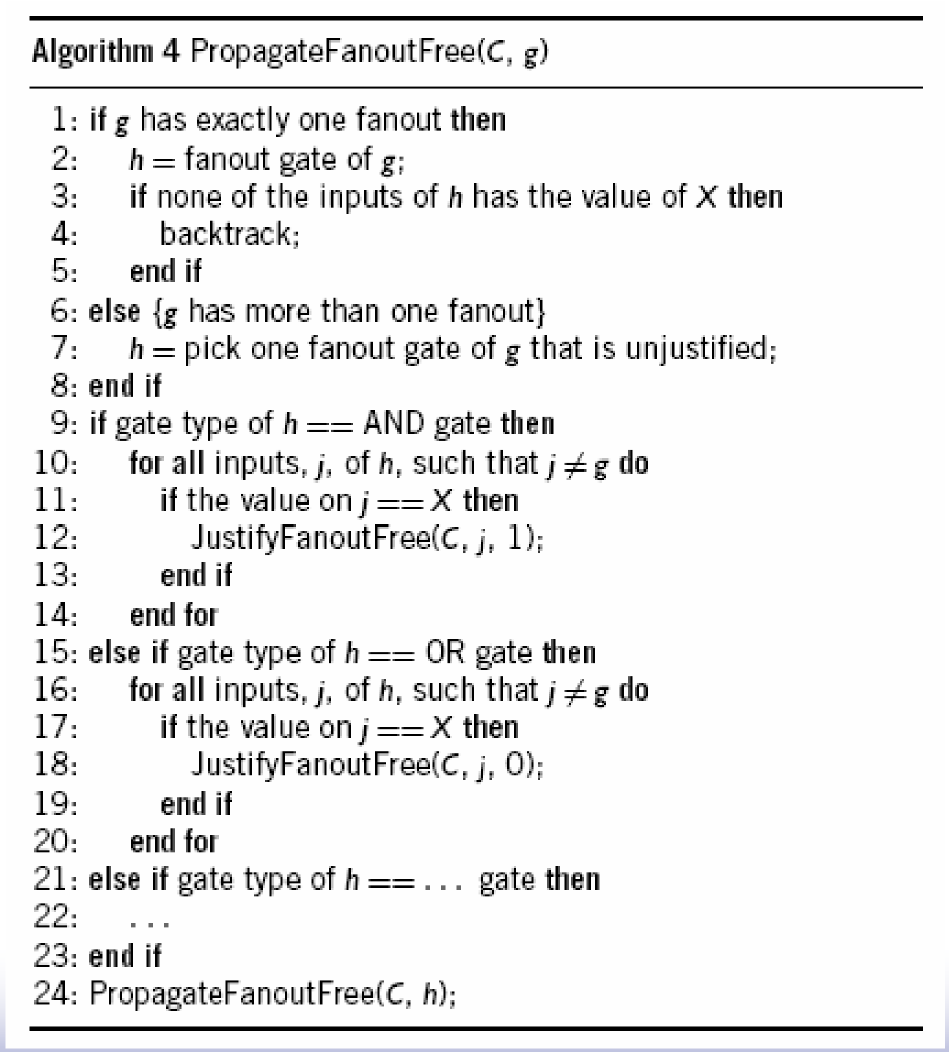


Fault: g/0

The recursive calls to JustifyFanoutFree():

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

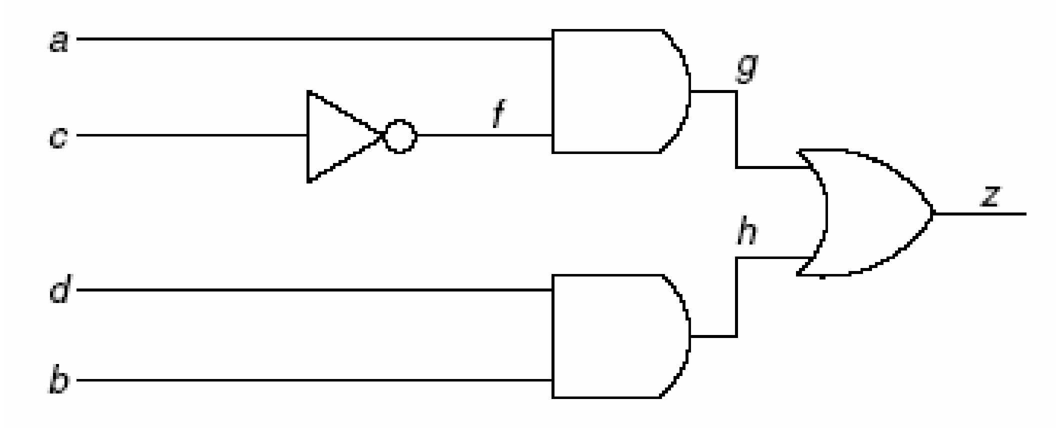
The Propagate Routine



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 $j \neq g$ 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 $j \neq g$ 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 $h == \ldots$ gate then
22:     ...
23: end if
24: PropagateFanoutFree($C$, $h$);
```

Example Continued



Propagate fault-effect from g to z

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

D Algorithm

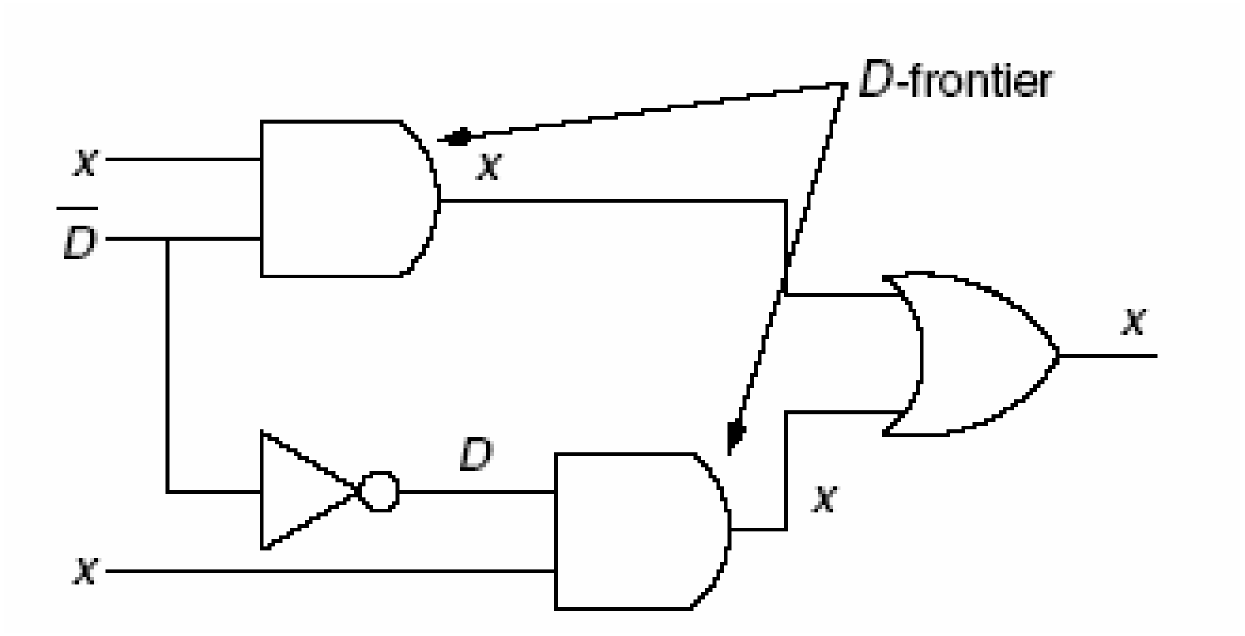
- Can handle arbitrary combinational circuits, with internal fanout structures
- Main idea: always maintain a non-empty D-frontier and try to propagate at least a fault effect to a primary output
- Initially, all circuit nodes are X, except for the fault cite, where a fault effect (D or D-bar) is placed.

D-Frontier and J-Frontier

- D-Frontier: All gates whose outputs are X but has at least one D or D-bar at the input of the gates
  - Initially, the D-frontier consists of only 1 gate (output of the fault-site)
- J-Frontier: All gates whose outputs are specified by are not justified by the input assignments

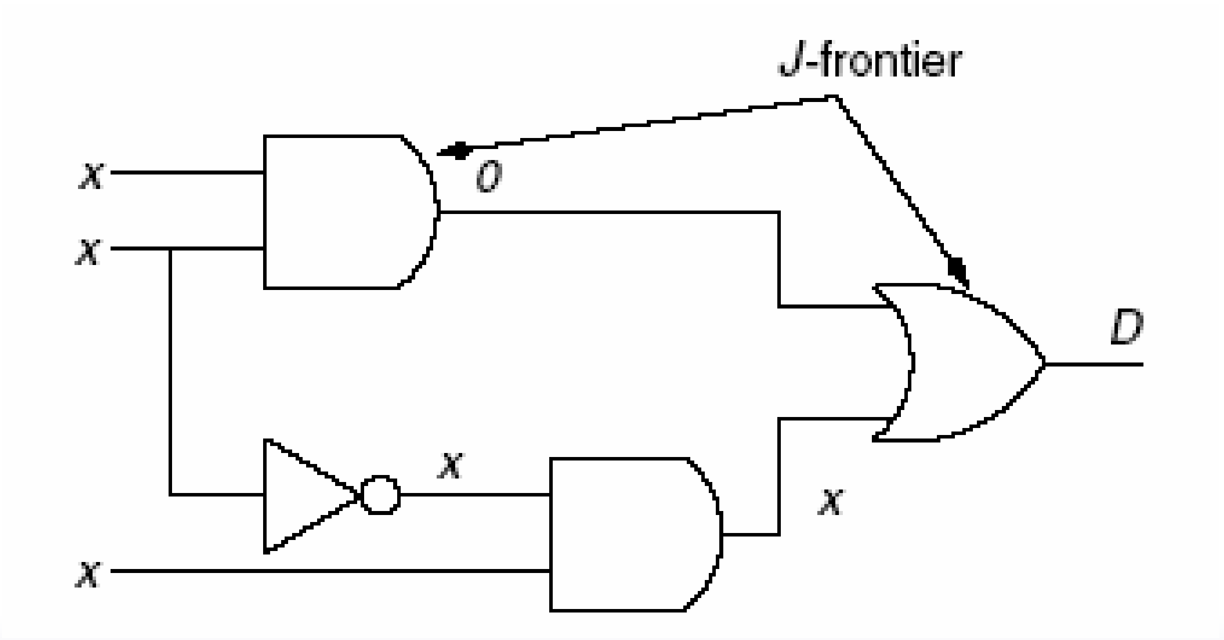
D-Frontier Example

- The D-frontier contains 2 gates



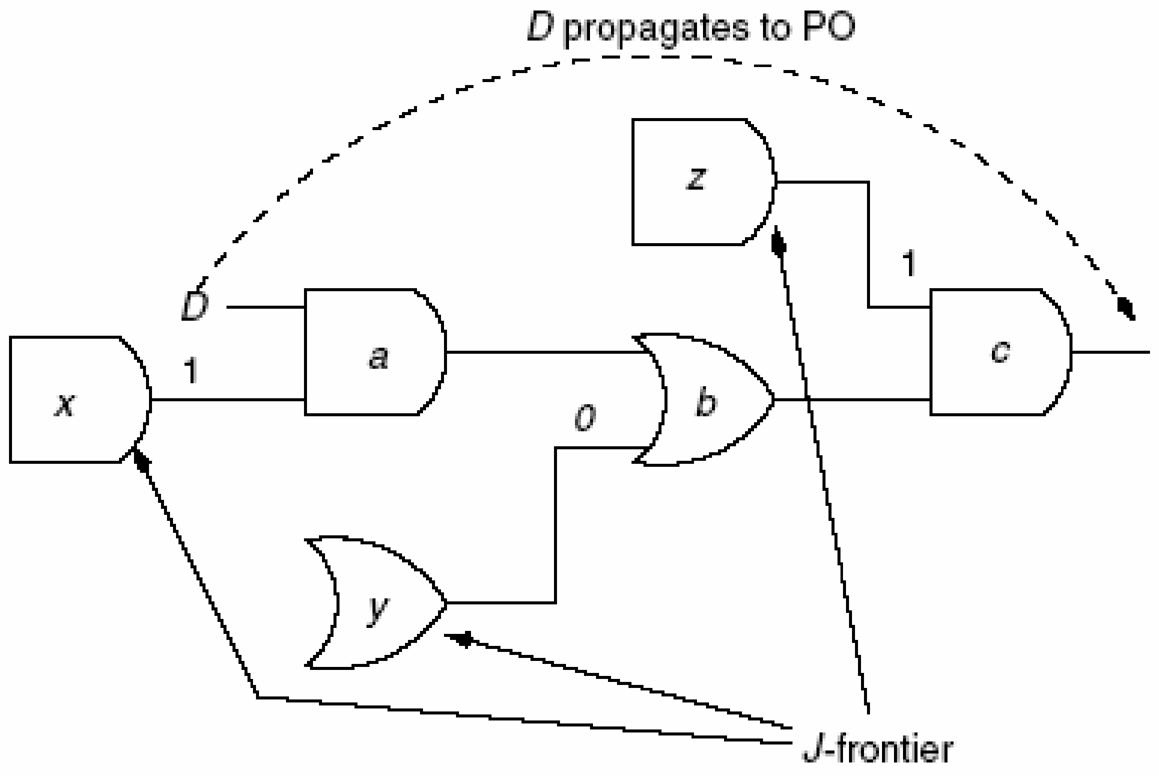
J-Frontier Example

- The J-Frontier contains 2 gates

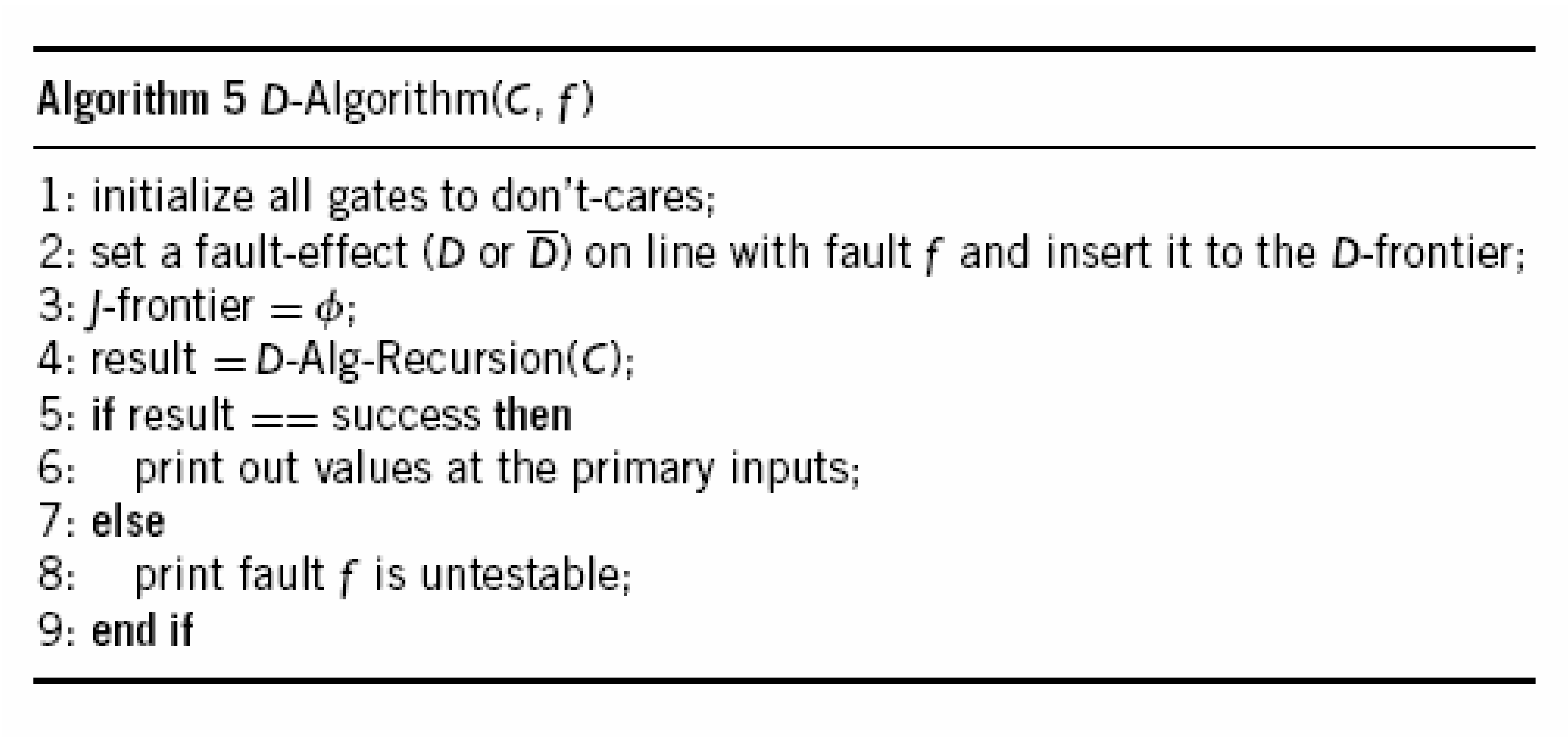


Idea Behind D Algorithm

- To advance the fault-effects in the D-frontier, add nodes to the J-frontier to justify



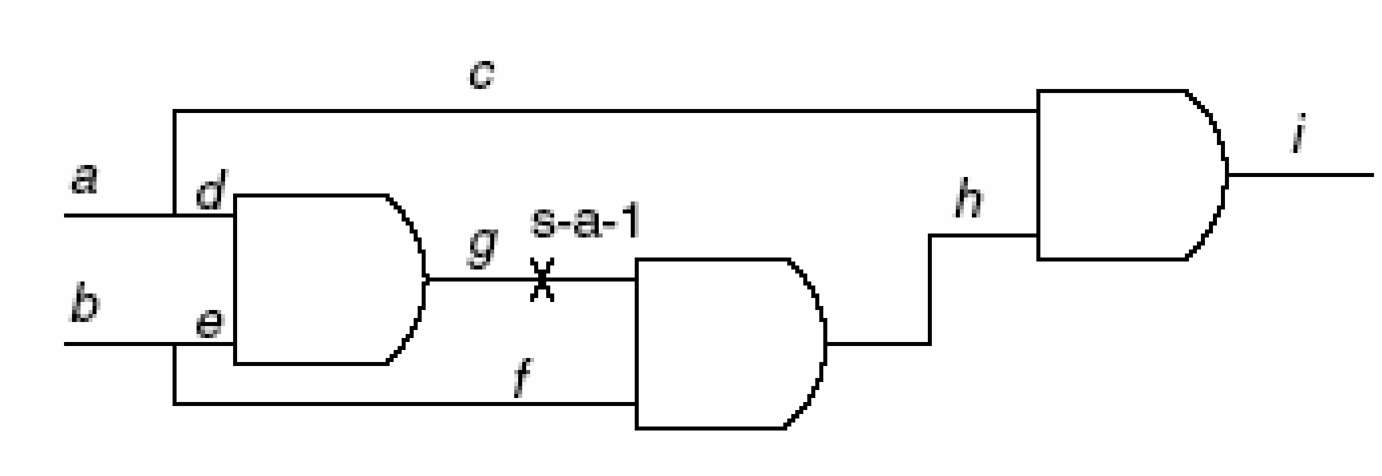
D Algorithm



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

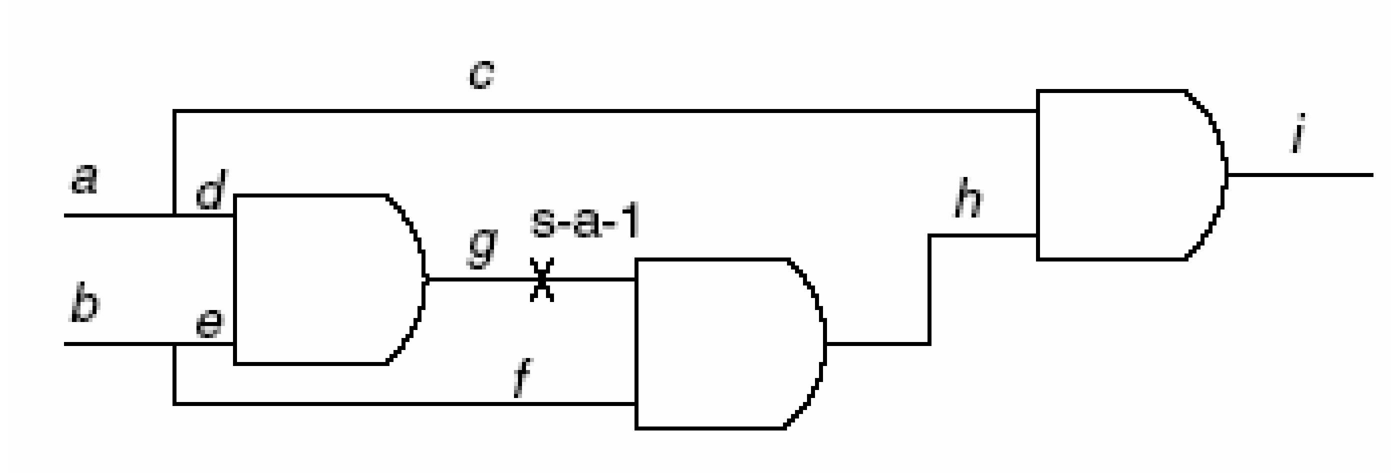
```
1: initialize all gates to don't-cares;
2: set a fault-effect ($D$ or $\overline{D}$) on line with fault $f$ and insert it to the $D$-frontier;
3: $J$-frontier $= \phi$;
4: result $= D$-Alg-Recursion($C$);
5: if result == success then
6:     print out values at the primary inputs;
7: else
8:    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

D Algorithm Example



- Target fault: g/1
- Initially, D-Frontier: {h}, J-Frontier={g=D-bar}
- To advance D-frontier, add f=1 and c=1 to J-frontier

D Algorithm Example (Cont.)



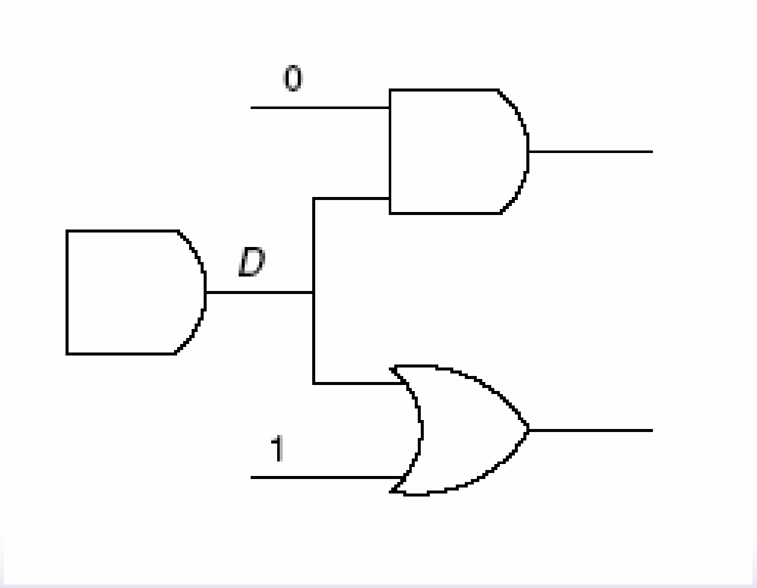
- ❑ Now justify every value in J-Frontier via branch-and-bound search
  - Must not make D-frontier empty or conflict with other J-frontier values
  - Otherwise backtrack
- ❑ Result: g/1 is untestable

PODEM

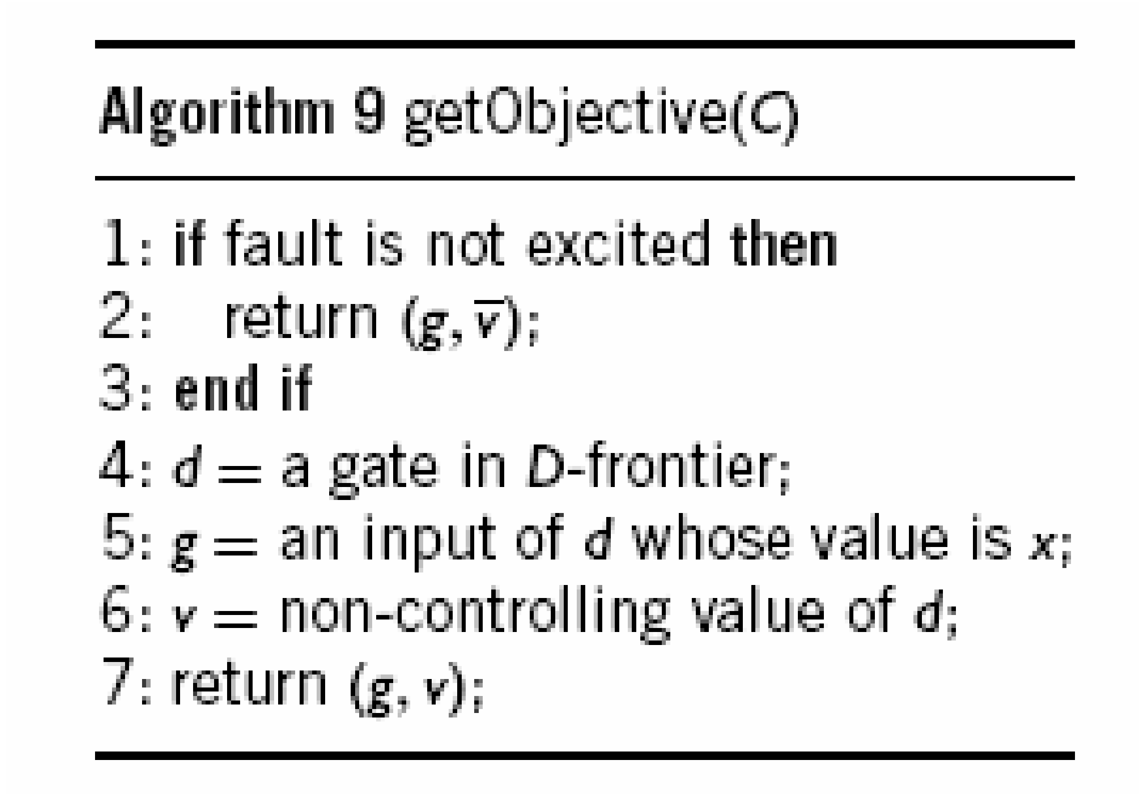
- Also a branch-and-bound search
- Decisions only on PIs
  - No J-Frontier needed
  - No internal conflicts
- D-frontier may still become empty
  - Backtrack whenever D-frontier becomes empty
  - Backtrack also when no X-path exists from any D/D-bar to a PO
- Decisions selected based on a backtrace from the current objective

X-Path

- The D in the circuit has no path of X's to any PO
  - i.e., the D is blocked by every path to any PO



Getting the Objective



Algorithm 9 getObjective(C)

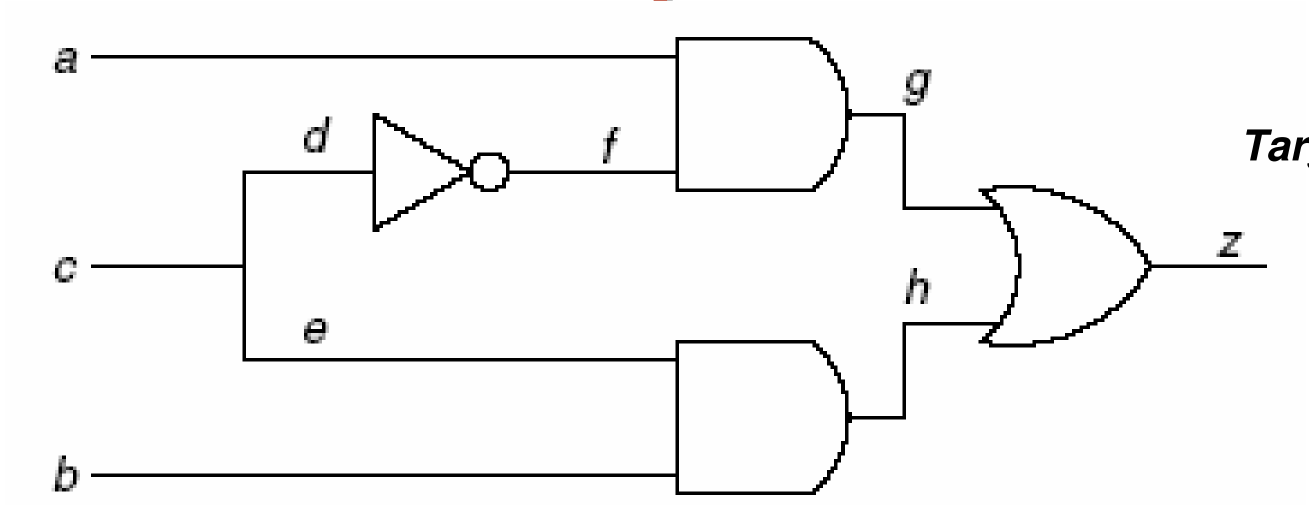
```
1: if fault is not excited then
2:     return $(g, \overline{v})$;
3: end if
4: $d$ = a gate in $D$-frontier;
5: $g$ = an input of $d$ whose value is $x$;
6: $v$ = non-controlling value of $d$;
7: return $(g, v)$;
```

Backtrace to Select a Decision

Algorithm 10 backtrace(C)

```
i = g;
num_inversion = 0;
while i ≠ primary input do
    i = an input of i whose value is x;
    if i is an inverted gate type then
        num_inversion++;
    end if
end while
if num_inversion == odd then
    v = $\overline{v}$;
end if
return(i, v);
```

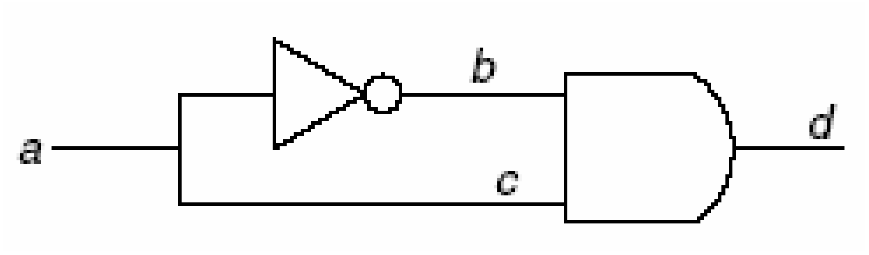
PODEM Example



Target fault: f/0

- 1st Objective: f=1in order to excite the target fault
- Backtrace from the object: c=0
- Simulate(c=0): D-Frontier = {g}, some gates have been assigned {c=d=e=h=0, f=D}
- 2nd Objective: advance D-frontier, a=1
- Backtrace from the object: a=1
- Simulate(a=0): Fault detected at z

Another PODEM Example



Target fault: b/0

- 1st Objective: excite fault: b=1
- Backtrace from objective: a=0
- Simulate(a=0): b=D, c=0, d=0: empty D-frontier. Must backtrack
- Change decision to a=1
- Simulate(a=1): b=0, c=1, d=1, D-frontier still emtpy
- Backtrack, no more decisions. Fault untestable.

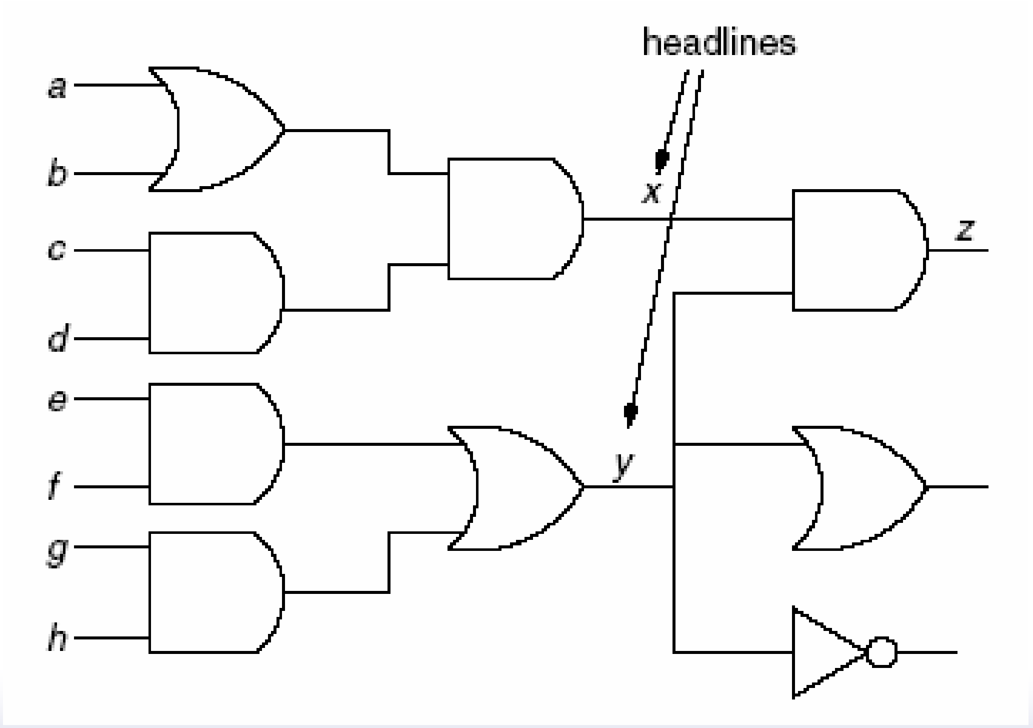
FAN

- Extend PODEM for an improved ATPG
- Concept of headlines to reduce the number decisions
- Multiple Objectives to reduce later conflicts

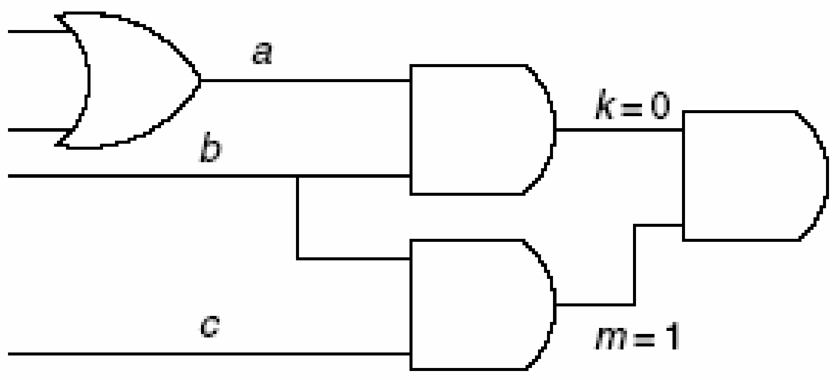
Headlines

- Output signals of fanout-free cones
- Any value on headlines can always be justified by the PIs

We only need to backtrace to the headlines to reduce the number of decisions



Multiple Objectives

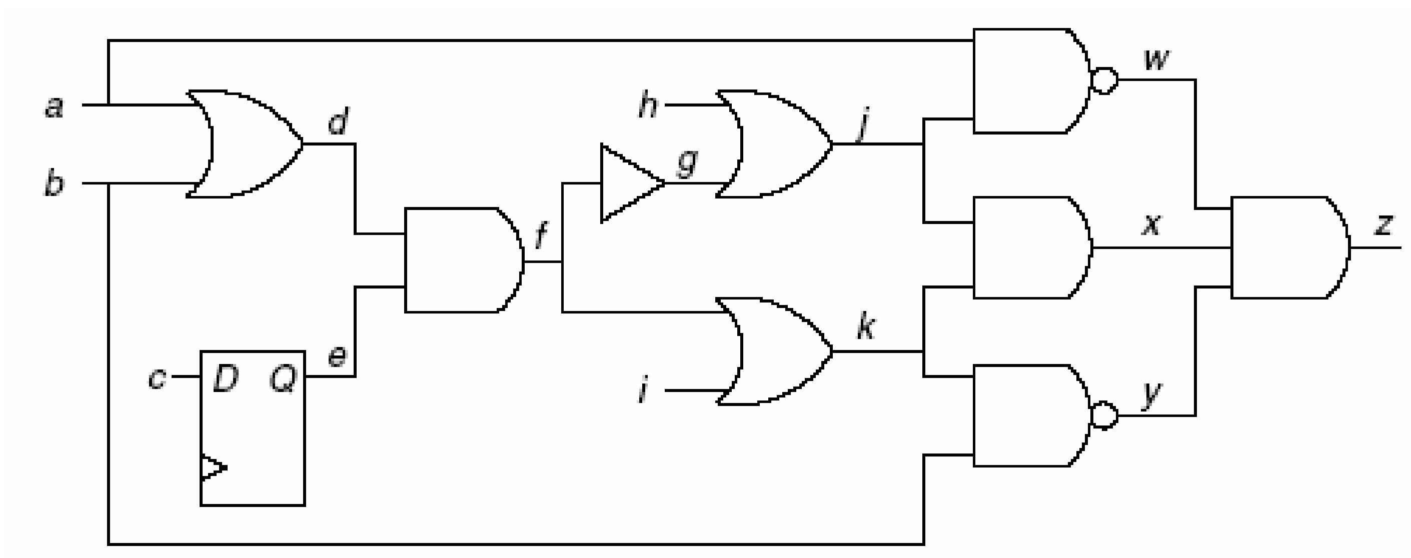


- Objectives: {k=0, m=1}
- Backtrace from k=0 may favor b=0, but simulate(b=0) would violate the second objective m=1!
- Makes backtrace more intelligent to avoid future conflicts

Static Logic Implications

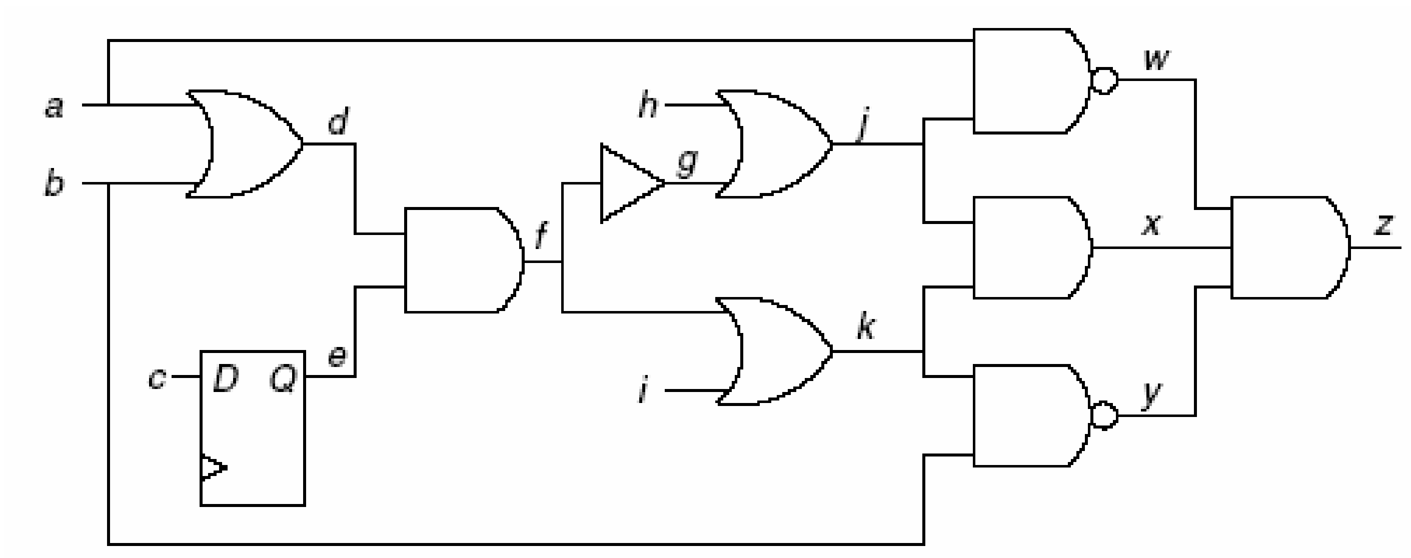
- Can help ATPG make better decisions
- Avoid conflicts
- Reduce the number of backtracks
- Idea: what is the effect of asserting a logic value to a gate on other gates in the circuit?

Direct Implications



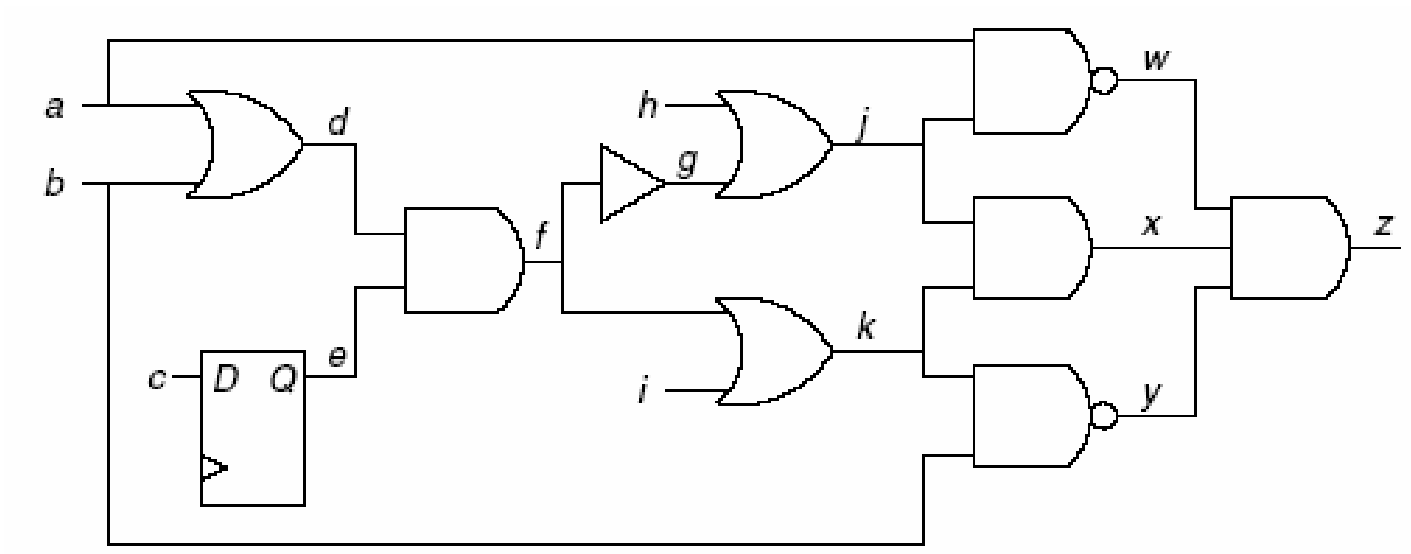
- Direct implications for f=1:
  - {d=1, e=1, g=1, j=1, k=1}
- Direct implications for j=0:
  - {h=0, g=0, f=0, w=1, w=0, z=0}

Indirect Implications



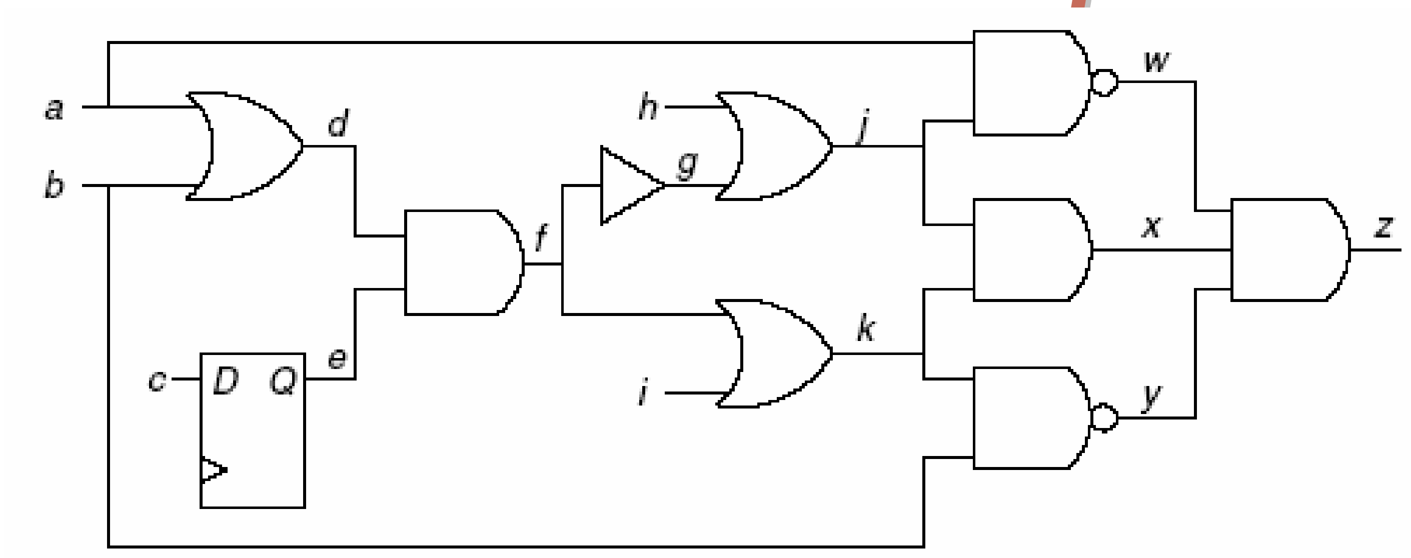
- Direct implications for f=1:
  - {d=1, e=1, g=1, j=1, k=1}
- Indirect Implications for f=1 obtained by simulating the direct implications of f=1:
  - {x=1}
- This is repeated for every node in the circuit

Extended Backward Implications



- Direct and indirect implications for f=1:
  - {d=1, e=1, g=1, j=1, k=1, x=1}
- Ext. Back. Implications obtained by enumerating cases for unjustified gates
  - Unjustified gates: {d=1}

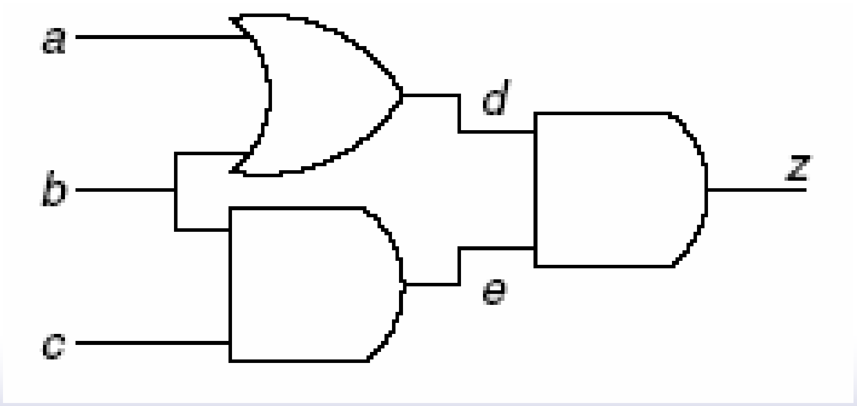
Extended Backward Implications



- In order to justify d=1, need either a=1 or b=1
  - Simulate(a=1, impl(f=1)) = Sa
  - Simulate(b=1, impl(f=1)) = Sb
- Intersection of Sa and Sb is the the set of ext. back. Implications for f=1
  - f=1 implies {z=0}
- This is repeated for every unjustified gate, as well as for every node in the circuit

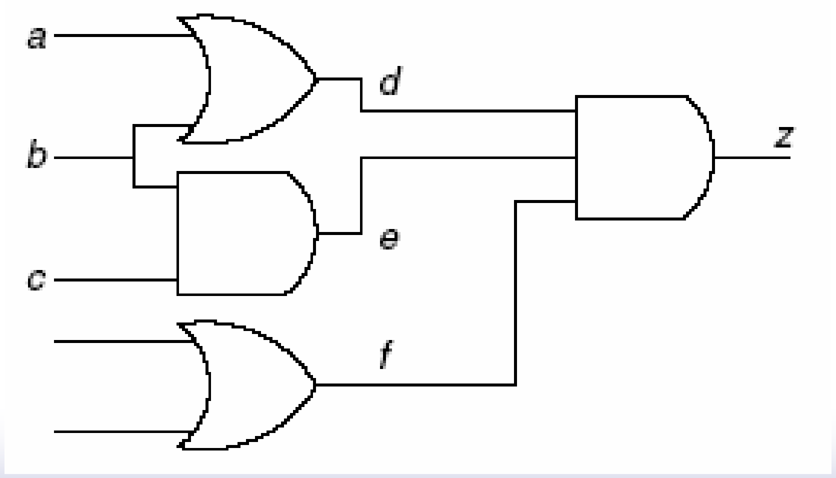
Dynamic Logic Implications

- Similar to Static Logic Implications, but has some signals already assigned values
- Suppose c=1 has already been assigned
  - Then to obtain z=0, b must be 0
  - This is the intersection of having either d=0 or e=0 in the presence of c=1



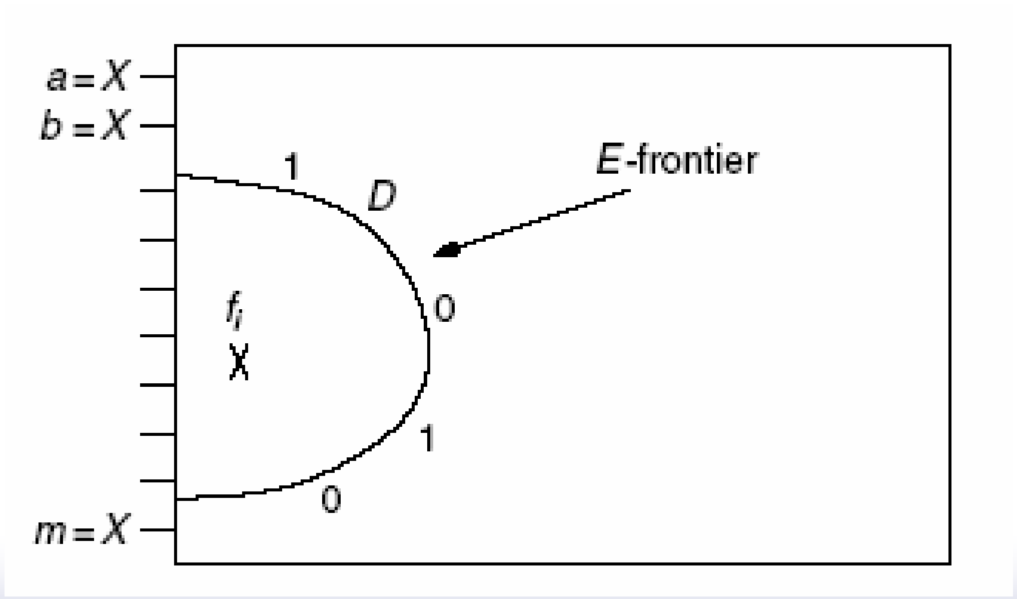
Another Dynamic Implications Example

- Suppose b=D
- In order to propagate the fault-effect to z, f = 1 is a necessary condition [Akers 76, Fujiwara 83]
- To take this further, the intersection of all the necessary assignments for all fault-effects in the D-frontier can be taken [Hamzaoglu99]



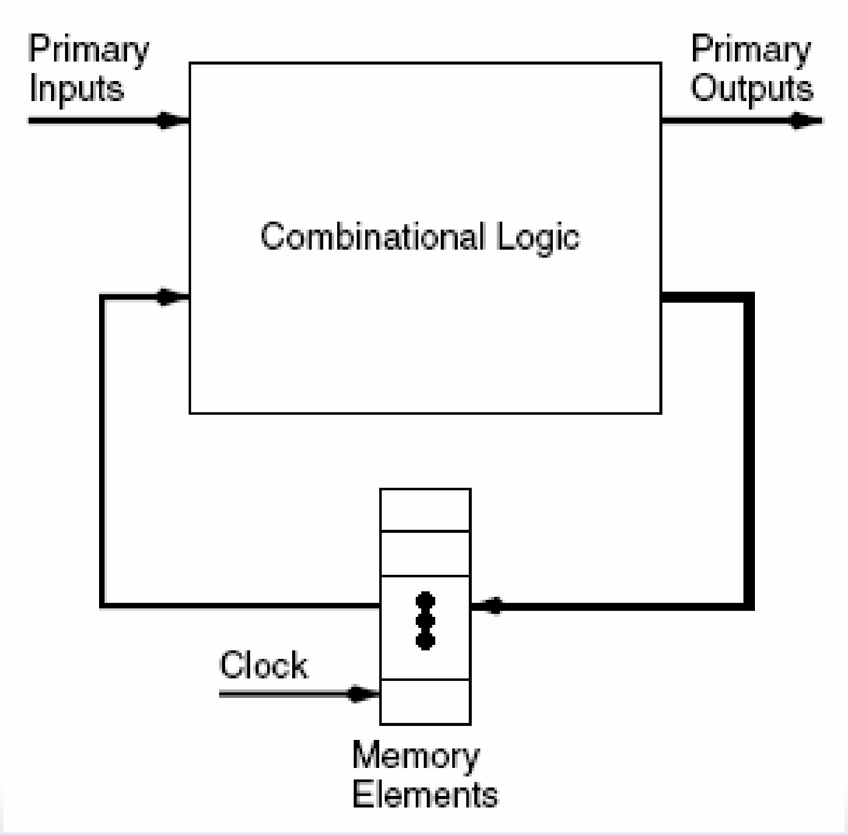
Evaluation Frontiers

- If two faults have the same E-frontier with at least one fault-effect, then the values on the unassigned PIs can be the same [Giraldi 90]

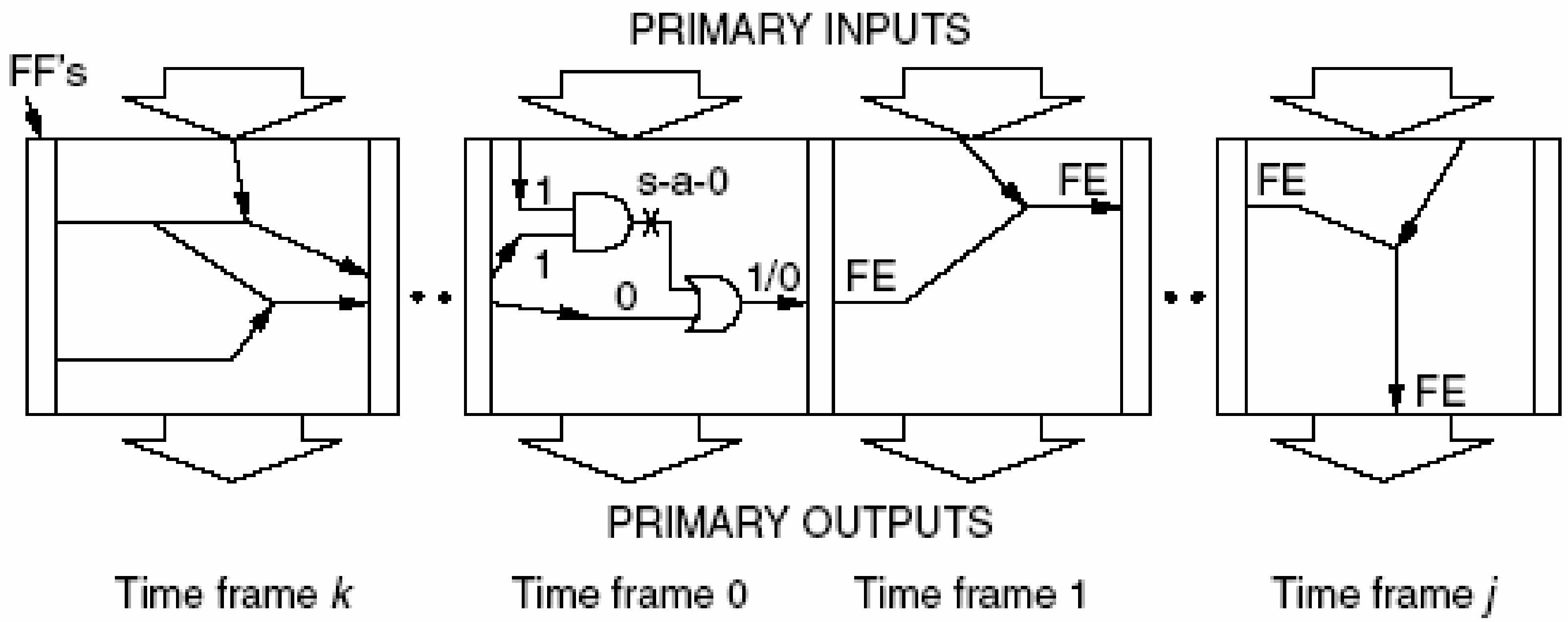


Sequential ATPG

- Huffman Model of a sequential circuit



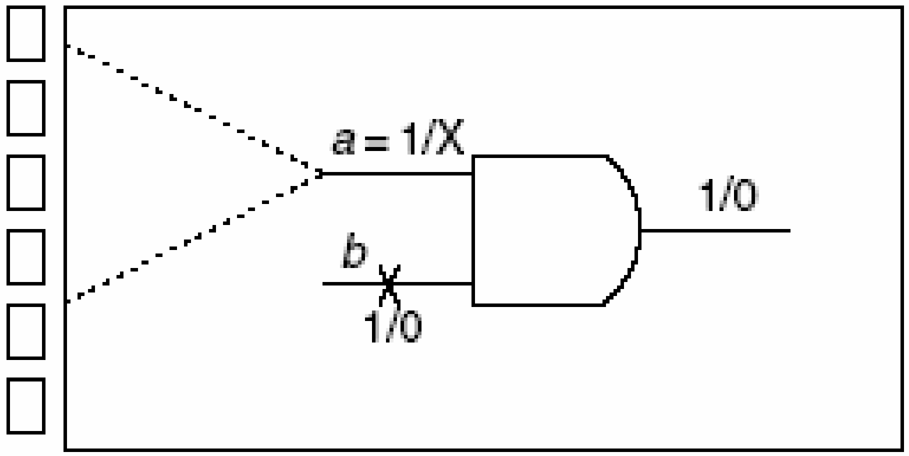
Iterative Logic Array Expansion



- To detect a fault, a sequence of vectors may be needed

Need 9-value Logic

- Consider b/0, it suffices to assign a=1/X to successfully propagate it
- The fault may or may not need to be excited and propagated in the previous time-frame!

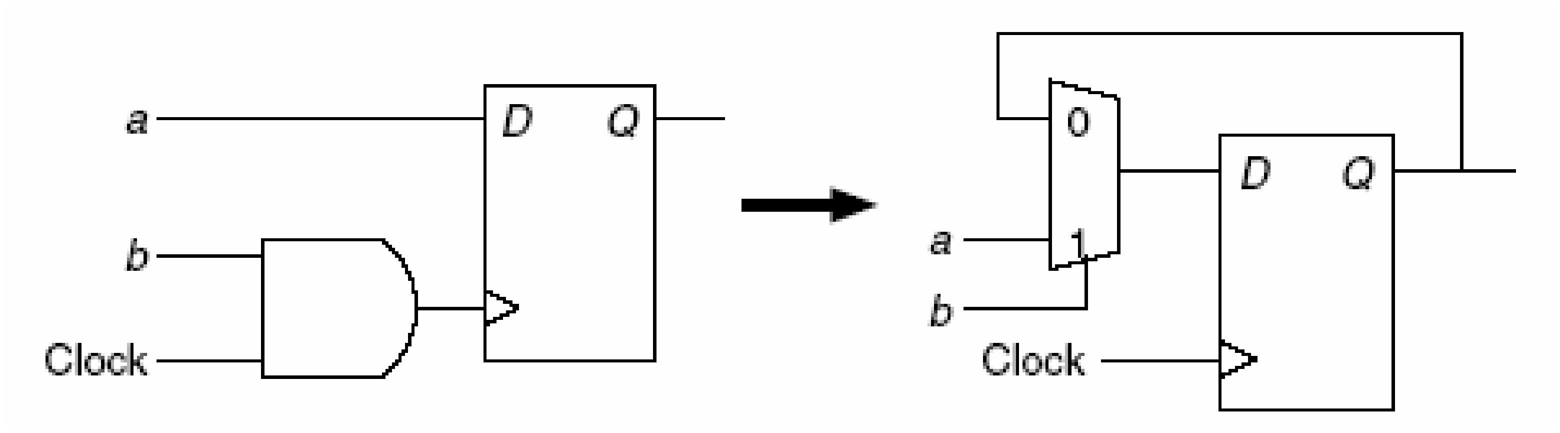


Basic Sequential ATPG Framework

- Based on Combinational ATPG
- Targets 1 time-frame at a time
- Excite the target fault in time-frame 0 and propagate it to a PO, possibly through several time-frames
- Justify the state needed at time-frame 0, via possibly several time-frames
- Sequential ATPG very complex, as backtracks can involve reversing decisions at different time-frames

Handling of Gated Clocks

- Simple circuit modification can help to handle gated clocks



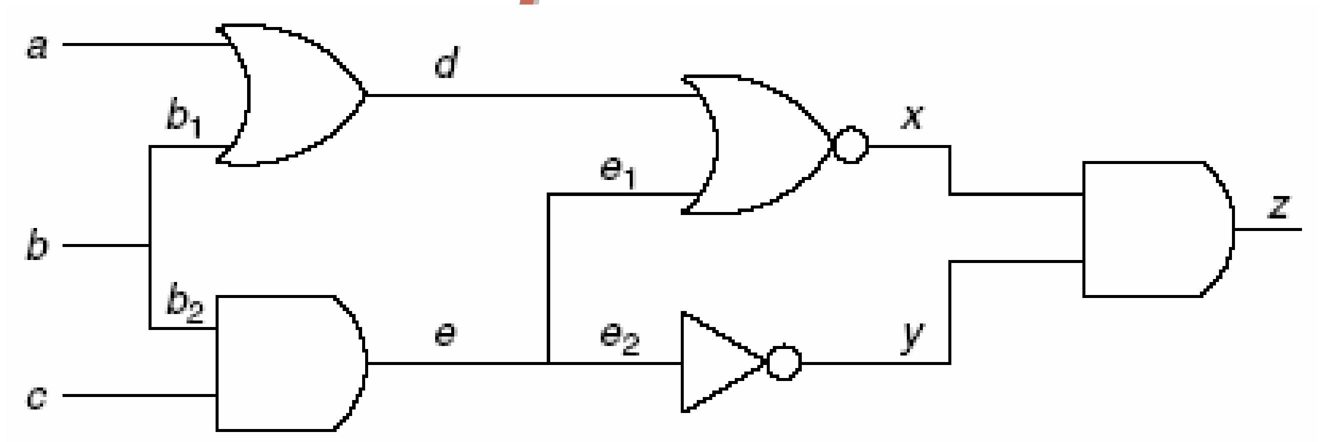
Fast Untestable Fault Identification

- Untestable faults are:
  - Those that could not be excited, or
  - Those that could not be propagated, or
  - Those that could not be simultaneously excited or propagated
- ATPG can spend a lot of time trying to generate a test for an untestable fault
  - Fast identification of untestable faults can allow the ATPG to skip those faults

FIRE [Iyer 1996]

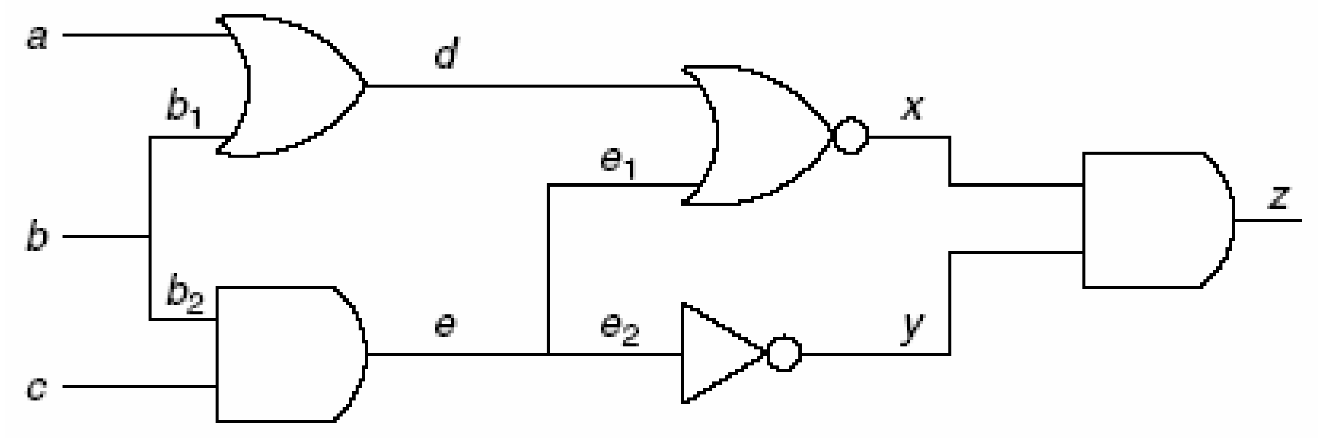
- Based on conflict analysis
- S0 = set of faults that are untestable when signal s=0
  - These faults must require s=1 to be detectable
- S1 = set of faults that are untestable when signal s=1
  - These faults must require s=0 to be detectable
- Intersection of S0 and S1 are definitely untestable
  - They require s=1 and s=0 simultaneously to be detectable!

FIRE Example



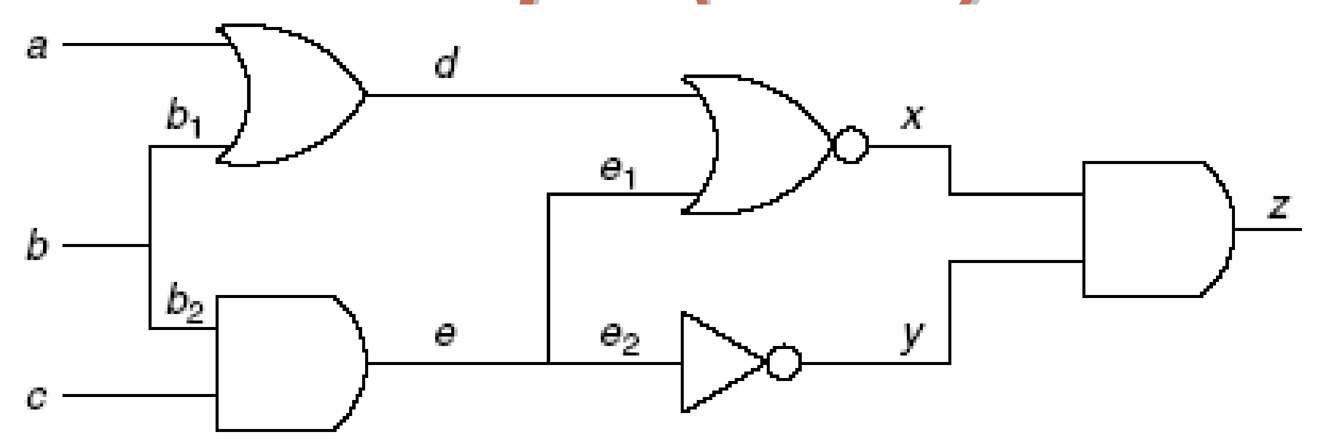
- Impl[b=1] = {b=1, b1=1, b2=1, d=1, x=0, z=0}
- Faults unexcitable when b=1: {b/1, b1/1, b2/1, d/1, x/0, z/0}
- Faults unobservable when b=1: {a/0, a/1, e1/0, e1/1, y/0, y/1, e2/0, e2/1}
- Faults undetectable when b=1: {a/0, a/1, b/1, b1/1, b2/1, d/1,e1/0, e1/1, e2/0, e2/1, x/0, y/0, y/1, z/0}

FIRE Example (Cont.)



- Impl[b=0] = {b=0, b1=0, b2=0, e=0, e1=0, e2=0, y=1}
- Faults unexcitable when b=0: {b/0, b1/0, b2/0, e/0, e1/0, e2/0, y/1}
- Faults unobservable when b=0: {c/0, c/1}
- Faults undetectable when b=0: {b/0, b1/0, b2/0, c/0, c/1, e1/0, e2/0, y/1}

FIRE Example (Cont.)

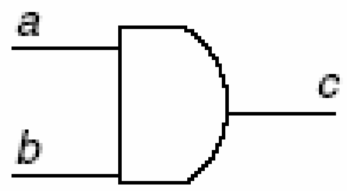


- Now that the two sets of faults undetectable when b=0 and b=1 have been computed
- The intersection of the two sets are those faults the require b=1 AND b=0 for detection, thus untestable:
  - {b2/0, c/0, c/1, e/0, e1/0, e2/0, y/1}

Generalization of FIRE

- Conflict on a single line: b=0 AND b=1
- Conflict on any illegal combination
  - Suppose FFs x=1, y=0, z=1 is illegal, then any fault that require x=1, y=0, and z=1 for detection will be untestable
  - This can be generalized to any illegal value combination in the circuit

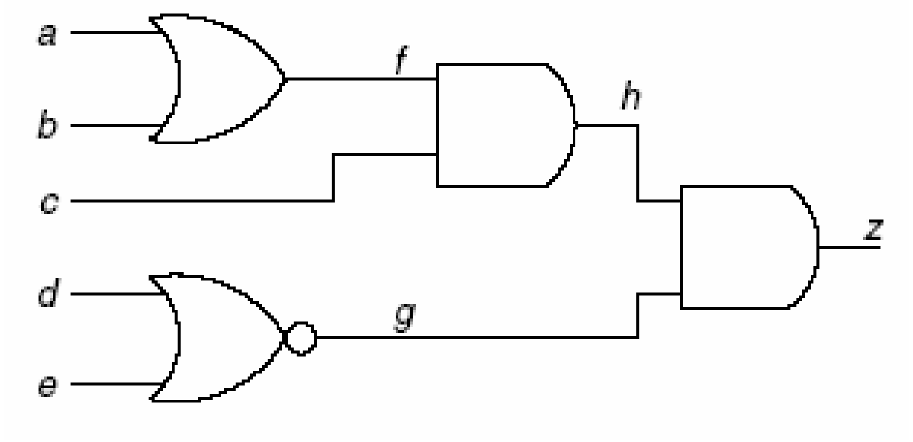
Multi-Line Conflict



- Consider the AND gate
- {a=0, c=1} is illegal (but this is captured by single-line conflicts)
- Likewise {b=0, c=1}
- But, {a=1, b=1, c=0} is a multi-line conflict not captured by single-line conflict
  - S_0—Set of faults not detectable when signal $a = 0$.
  - S_1—Set of faults not detectable when signal $b = 0$.
  - S_2—Set of faults not detectable when signal $c = 1$.

Intersection of S0, S1, S2 will be untestable faults due to this multi-line conflict

Multi-Line Conflicts (Cont.)



- Can extend the previous concept further
- Consider multi-line conflict {h=1, g=1, z=0}
- We can extend these values as far as possible: {f=1, c=1, d=0, e=0, z=0} is a multi-line conflict as well

Summary on Untestable Fault Identification

- First compute static logic implications
- Compute untestable faults based on single-line conflicts
- Compute untestable faults based on multi-line conflicts
- Remove all identified untestable faults from the fault list

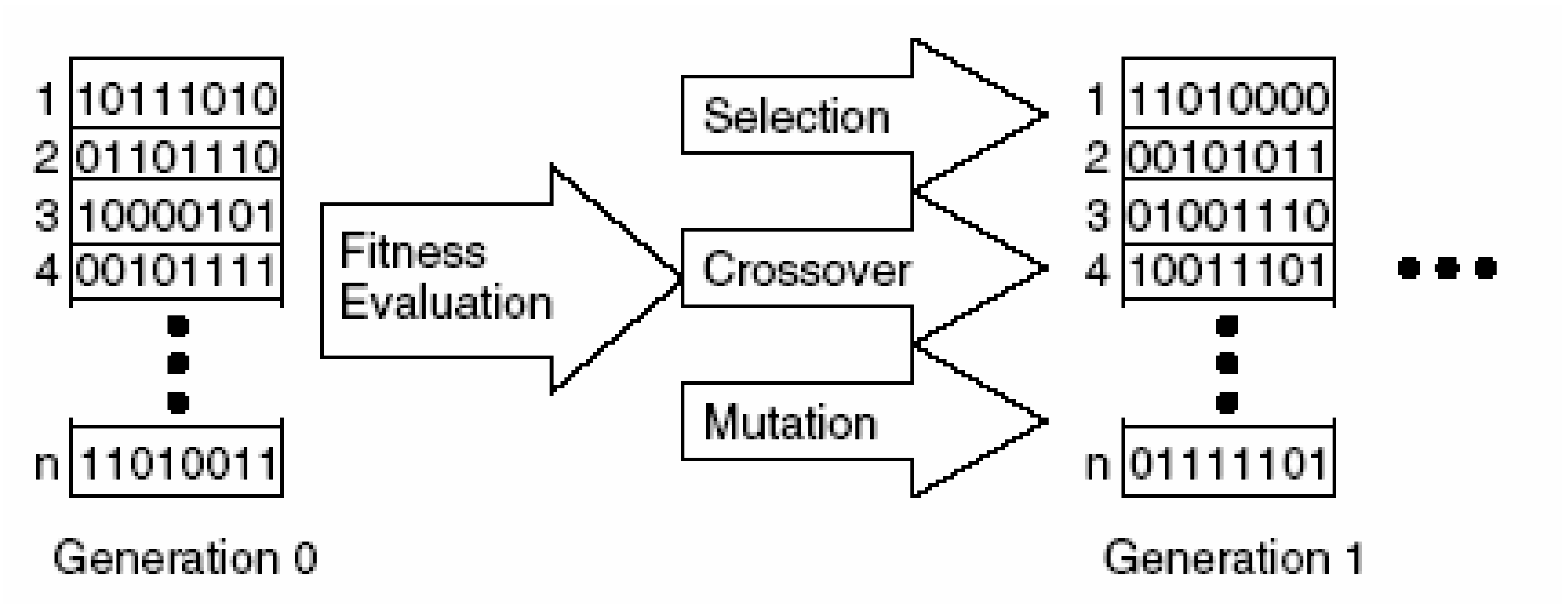
Simulation-Based ATPG

- Random and weighted-random TPG are the simplest forms of simulation-based ATPG
- Challenge: how to guide the search to generate effective vectors to obtain high fault coverage, low computation costs, and small test sets?

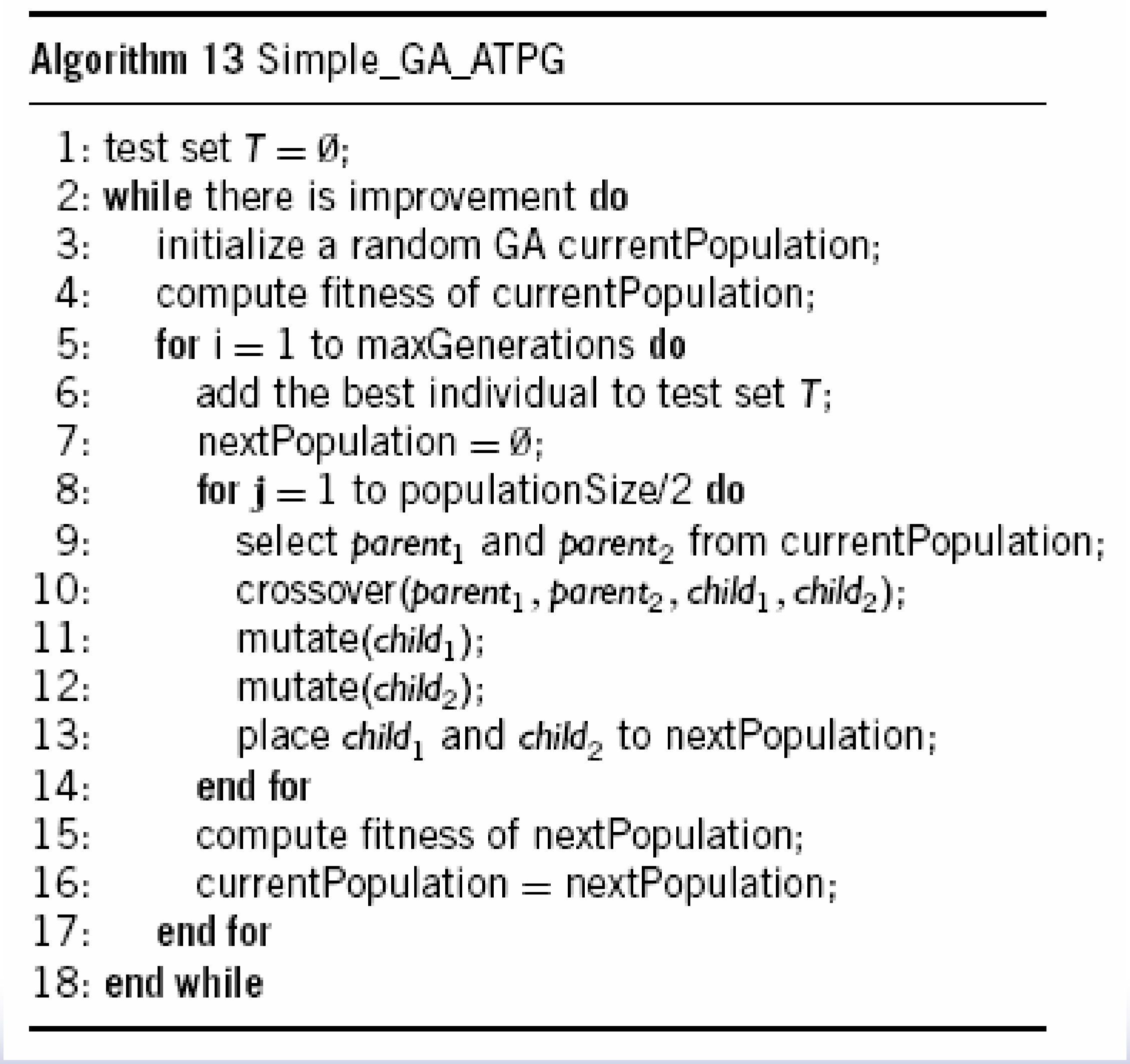
Genetic Algorithms for Sim-based ATPG

- A GA made up of
  - A population of individuals (chromosomes)
    - Each individual is a candidate solution
  - Each individual has an associated fitness
    - Fitness measures the quality of the individual
  - Genetic operators to evolve from one generation to the next
    - Selection, crossover, mutation

Illustration of GA process



Pseudo Code for GA

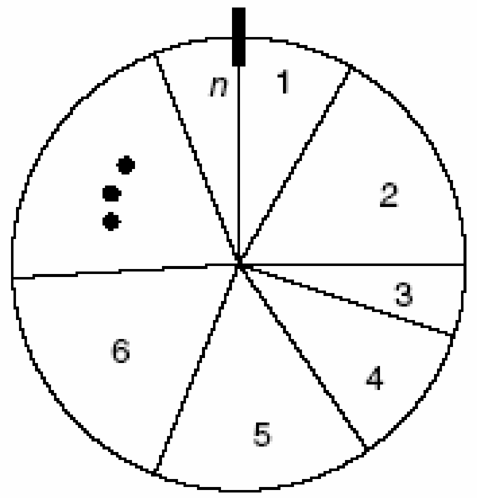


Algorithm 13 Simple_GA_ATPG

```
 1: test set $T = \emptyset$;
 2: while there is improvement do
 3:     initialize a random GA currentPopulation;
 4:     compute fitness of currentPopulation;
 5:     for i = 1 to maxGenerations do
 6:         add the best individual to test set $T$;
 7:         nextPopulation = $\emptyset$;
 8:         for j = 1 to populationSize/2 do
 9:             select $parent_1$ and $parent_2$ from currentPopulation;
10:             crossover($parent_1$, $parent_2$, $child_1$, $child_2$);
11:             mutate($child_1$);
12:             mutate($child_2$);
13:             place $child_1$ and $child_2$ to nextPopulation;
14:         end for
15:         compute fitness of nextPopulation;
16:         currentPopulation = nextPopulation;
17:     end for
18: end while
```

The Selection Operator

- ❑ Roulette Wheel Selection



- ❑ Tournament Selection

The Crossover Operator

- One-point crossover

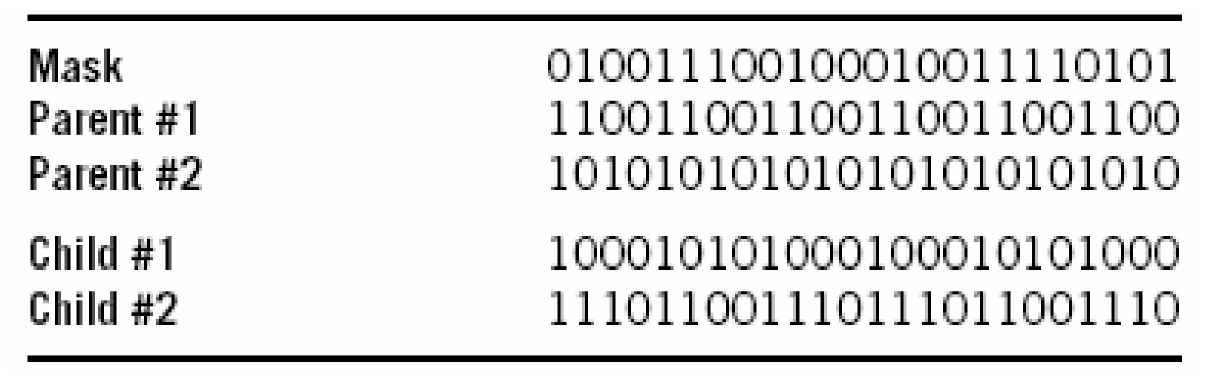
Parent #1	110011001100	110011001100
Parent #2	101010101010	101010101010
Child #1	110011001100	101010101010
Child #2	101010101010	110011001100

- Two-point crossover

Parent #1	11001100	11001100	11001100
Parent #2	10101010	10101010	10101010
Child #1	11001100	10101010	11001100
Child #2	10101010	11001100	10101010

Uniform Crossover

- The crossover is performed whenever a mask bit is set



Mask	010011100100010011110101
Parent #1	110011001100110011001100
Parent #2	101010101010101010101010
Child #1	100010101000100010101000
Child #2	111011001110111011001110

The Mutation Operator

- Random flip of a bit position
- Need to keep mutation rate small, so that the search will not seem randomized

GA Population Size

- Should be a function of the individual size
- Larger individuals require larger populations to allow for reasonable diversity
- Individual size depends on the number of PIs in the circuit
  - In sequential circuits, an individual may be a sequence of vectors
- Generation Gap: some individuals may be carried over from one generation to the next

Number of GA Generations

- Related to the population size
  - Larger populations usually demand more generations
  - Generation gap also will affect the number of generations needed to reach a satisfactory solution

The Fitness Function

- Measures the quality of the individual
- Essential for a GA to converge on a solution
- Example fitness functions:
  - Number of faults detected by the individual
  - Number of faults excited by the individual
  - Number of flip-flops set to a specified value (in seq ckts)
  - A weighted sum of various factors

CONTEST [1989]

- Two-stage process
- 1st stage: aim to detect as many faults as possible
  - Fitness = a x #detected + b x #excited
- 2nd stage: aim to detect remaining hard faults individually
  - Fitness depends on if the target fault has been excited, and how many fault effects are in the circuit

GATEST [1994]

- GA-based ATPG for seq ckts
- Tournament selection, uniform crossover
- 1st phase: initialize the seq ckts
- 2nd phase: detect & excite as many faults as possible
- 3rd phase: similar to phase 2, but to monitor fault-free and faulty ckt events
- 4th phase: individuals now become sequence of vectors, aim to detect & excite as many faults as possible

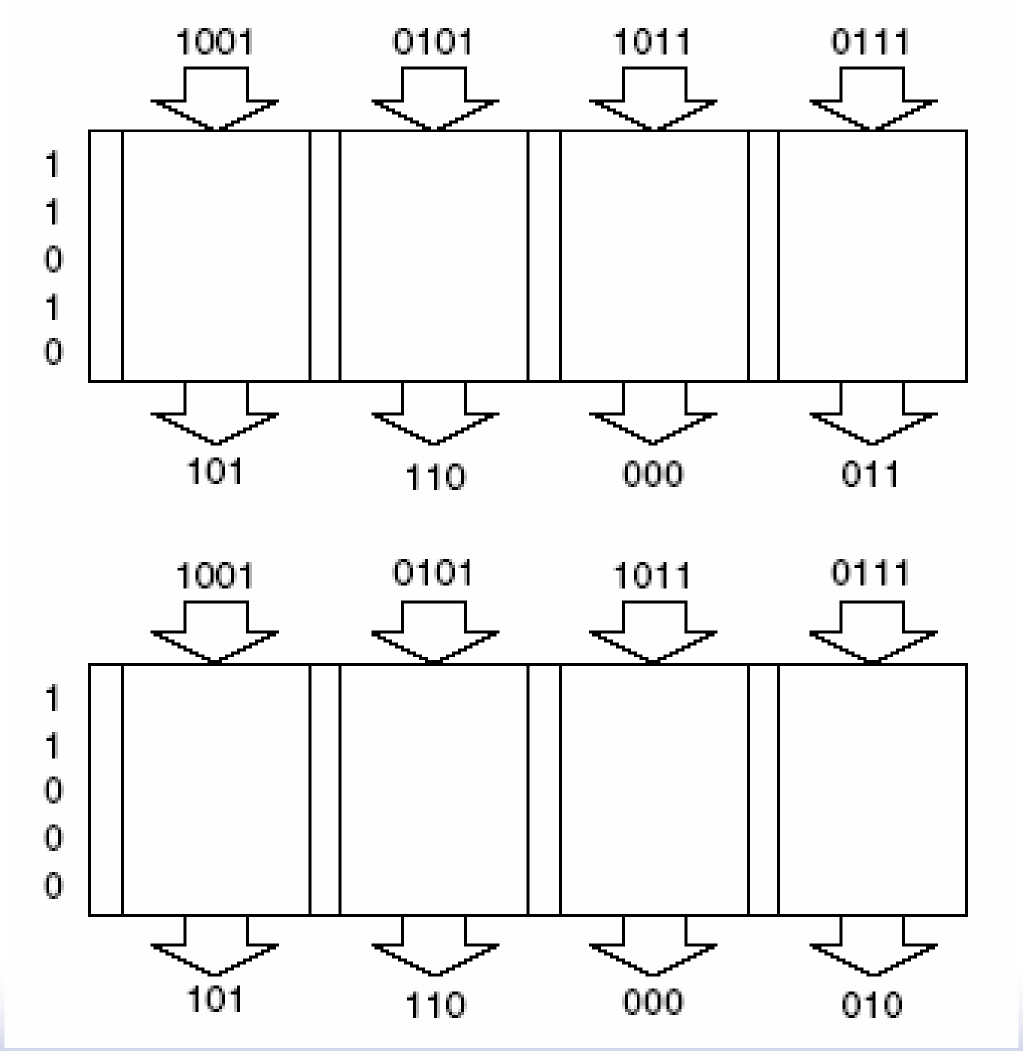
Seeding the Initial Population

- Place non-random individuals in the initial population
- This can reduce the number of generations needed for the GA to obtain a good solution
- Aggressively used in STRATEGATE [1997]
  - Target individual faults rather than groups of faults
  - Seeding of propagation sequences
  - Seeding of justification sequences

Distinguishing Sequences

- A dist sequence is a sequence that can generate different output sequences starting from two different initial states

Distinguishing Sequences to Help Propagate Fault Effects



- The 4-vector sequence can distinguish state 11010 from 11000
- If a fault has been excited and prop to FF#4, then this sequence has a good chance of propagating it to the last PO in the 4th vector

Storing Distinguishing Sequences

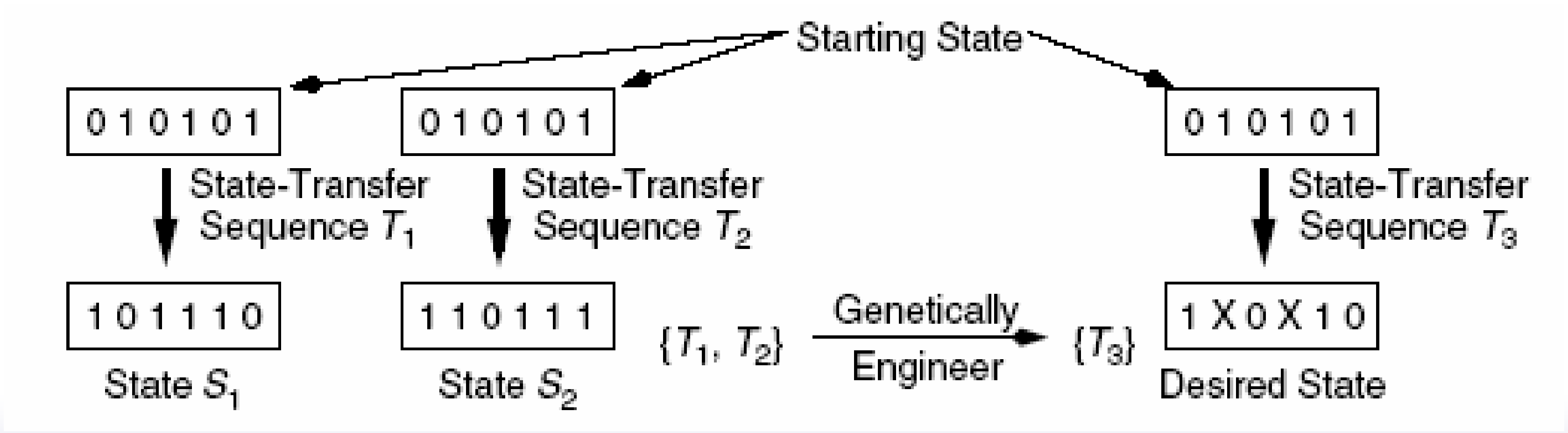
- STRATEGATE computes and stores a number of distinguishing sequences
- Whenever a target fault has been excited and its fault-effect propagated to a FF
  - The relevant distinguishing sequences are seeded into the population to help propagate the fault-effect to a PO

Justification Sequences

- Some hard faults require the circuit to be in a particular state in order to detect the fault
- State justification is a hard problem
- Idea: seed sequences that can reach similar states in the past in the initial population

State Justification (STRATEGATE)

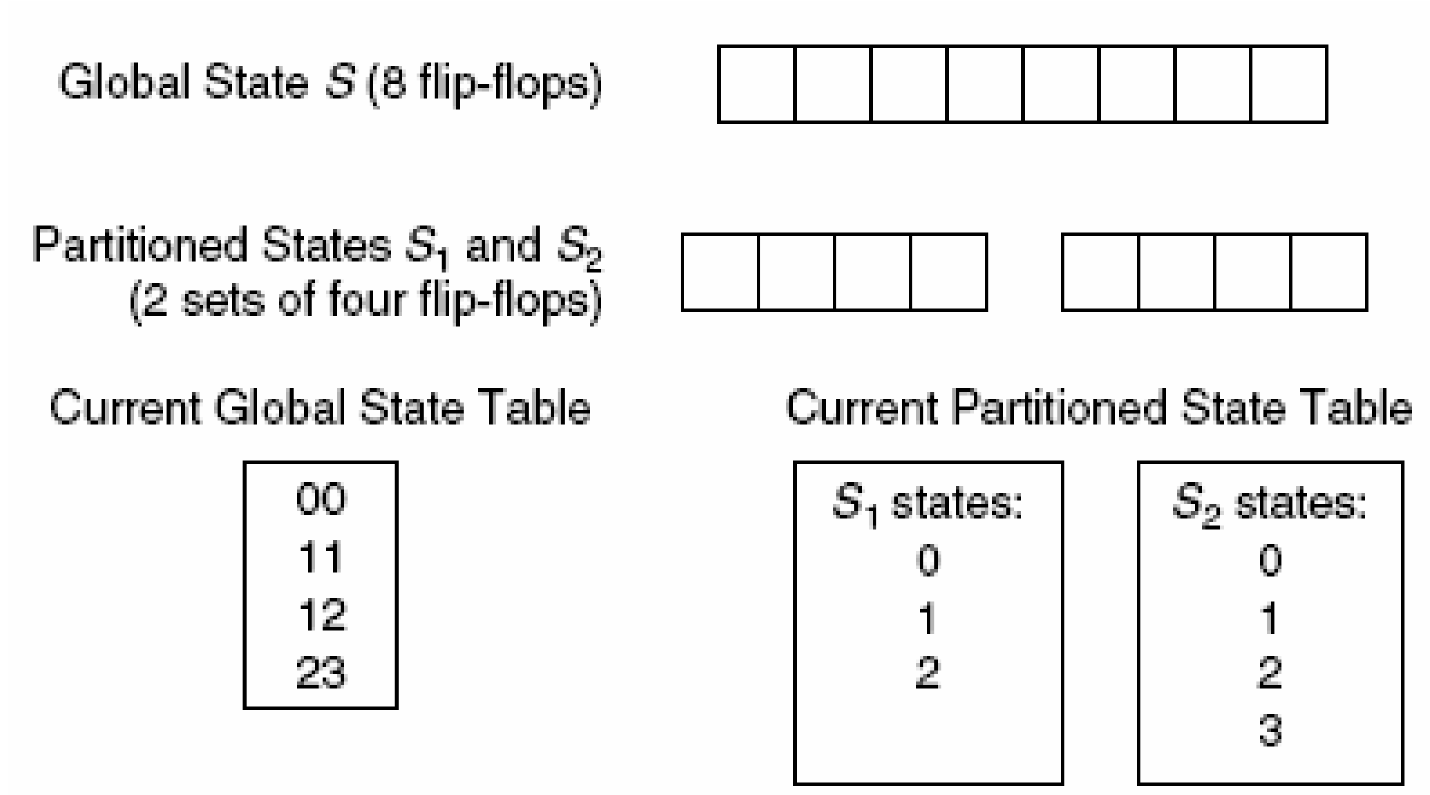
- Suppose the target/desired state is 1x0x10
- We have visited 101110, 110111, etc in the past
- Seed state-transfer sequences into the initial population



Logic Simulation-Based ATPG

- Fault simulation much more expensive than logic simulation
- Is it possible to achieve high-quality test sets with only logic simulation?
- CRIS [1992] uses fault-free circuit events in the fitness to guide the search
  - Good for combinational circuits
- LOCSTEP [1995] uses number of new states reached as fitness to guide the search in seq ATPG
  - Good for seq ckts with small number of flip-flops

State Space Partitioning for Logic Sim-Based ATPG



Global State S (8 flip-flops)
Partitioned States S_1 and S_2 (2 sets of four flip-flops)
Current Global State Table
00
11
12
23
Current Partitioned State Table
S_1 states:
0
1
2
S_2 states:
0
1
2
3

- New state 44 better than another new state 22 because both partitions see something new

State-Partition Based ATPG

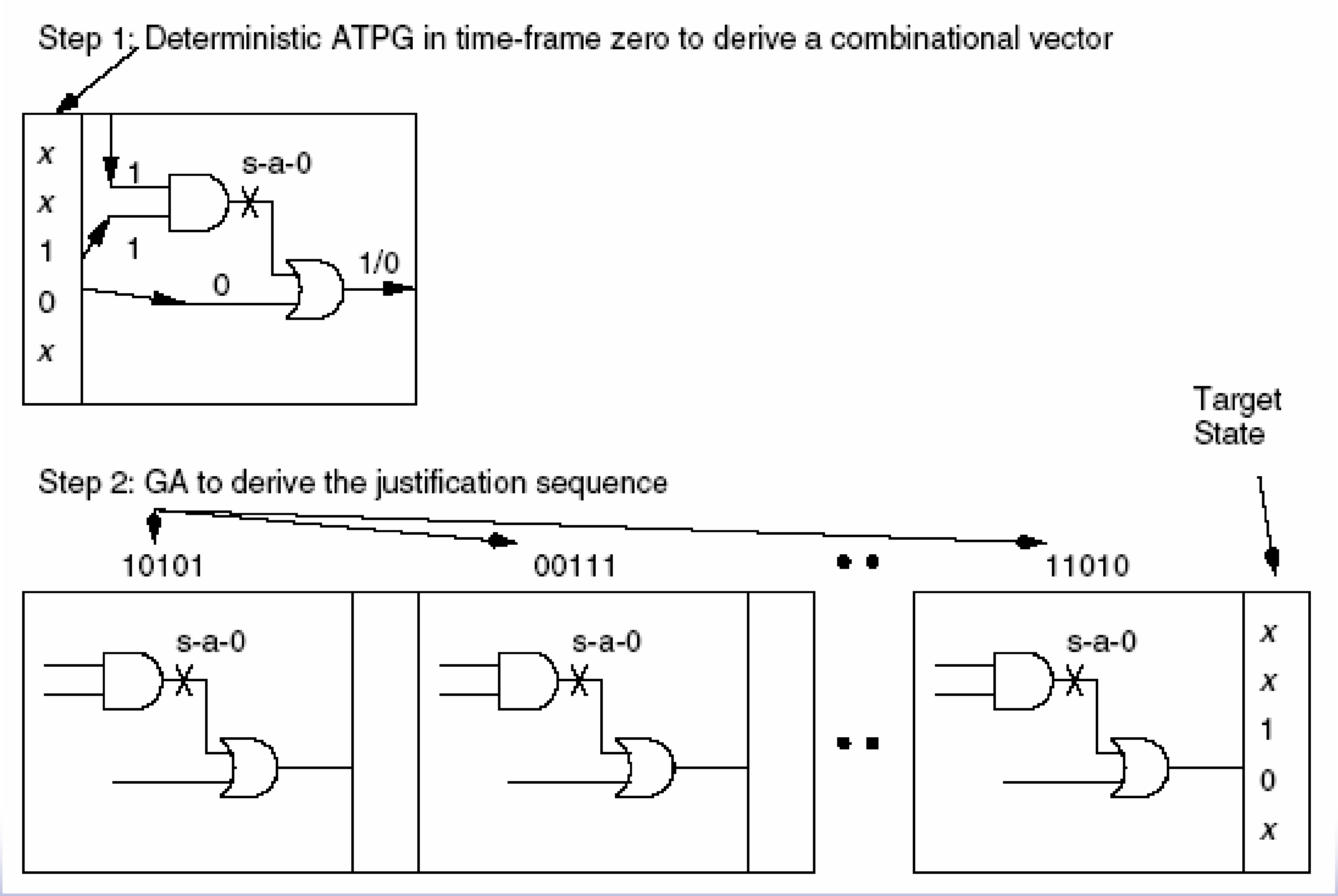
- Partition the FFs
- A state is considered new only if at least one state group has a new value from the state
- This helps filter many noise in trying to reach as many new states as possible
- Very high fault coverage is achieved

Spectrum-Based ATPG

- View the seq ckt as a black box
- Extract spectral characteristics of the primary inputs
- Use the spectral characteristics to generate effective vectors

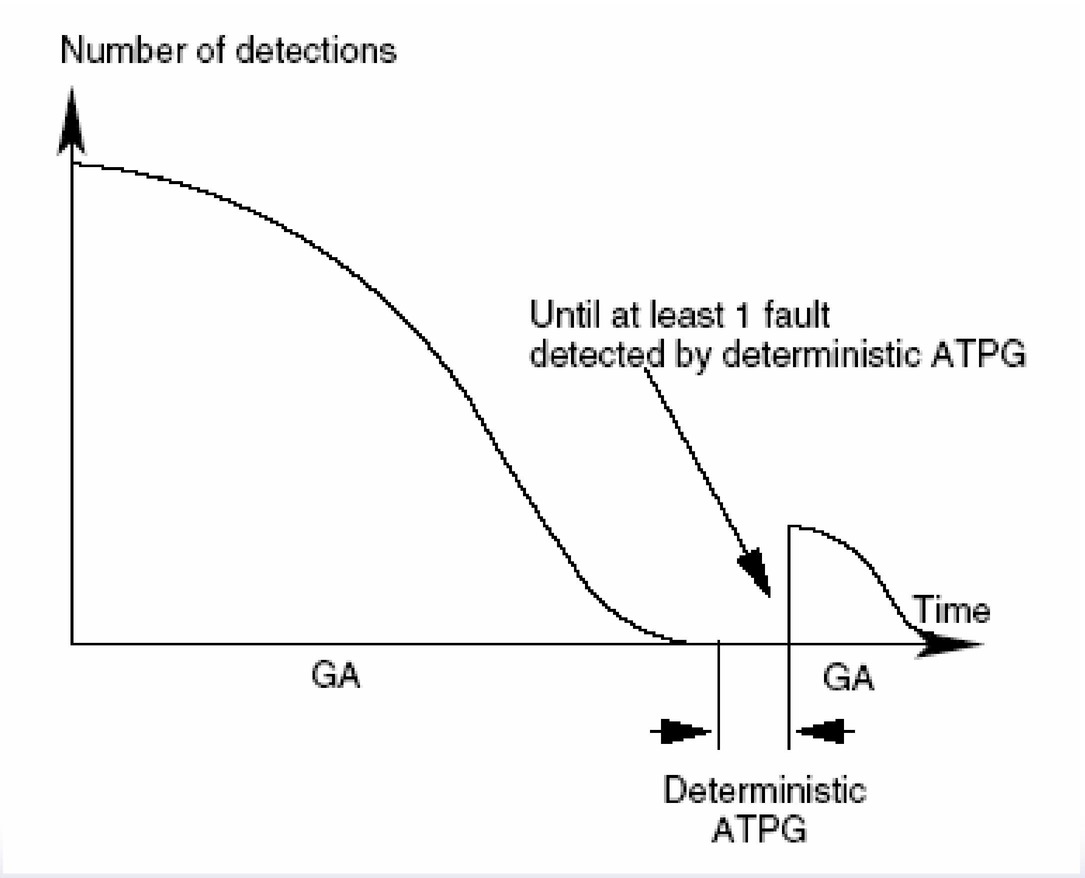
Hybrid Deterministic / Sim-based ATPG

- GA-HITEC [1995]



Step 1: Deterministic ATPG in time-frame zero to derive a combinational vector
x
x
1
0
x
1
1
0
s-a-0
1/0
Target State
Step 2: GA to derive the justification sequence
10101
00111
11010
s-a-0
s-a-0
s-a-0
x
x
1
0
x

ALT-TEST Hybrid [1996]



- ❑ Alternates between GA and deterministic ATPG
- ❑ GA passes 1 hard faults to det. ATPG
- ❑ Det ATPG passes vector sequence used as seed for next GA run

Delay Testing

- Delay defects: class of defects that affects the functionality only when the circuit is running at a high speed
- Stuck-at fault model insufficient to model all delay-related defects
- Delay fault models
  - Path delay fault
  - Transition fault
  - Segment delay fault

Applications of Delay Tests

- Launch on capture (aka broadside or double capture)
  - V1 is arbitrary, v2 is derived from v1 through the circuit function
- Launch on shift (aka skewed load)
  - V1 is arbitrary, v2 is derived by a 1-bit shift of v1
- Enhanced scan
  - V1 and V2 are uncorrelated

Launch-on-Capture / Broadside / Double Capture

- True at-speed test
- Benefits
  - Detect intra-clock-domain faults and inter-clock-domain structural faults or delay faults at-speed
  - Facilitate physical implementation
  - Avoid some of functionally infeasible paths
  - Ease integration with ATPG

Launch-on-Shift / Skewed-Load

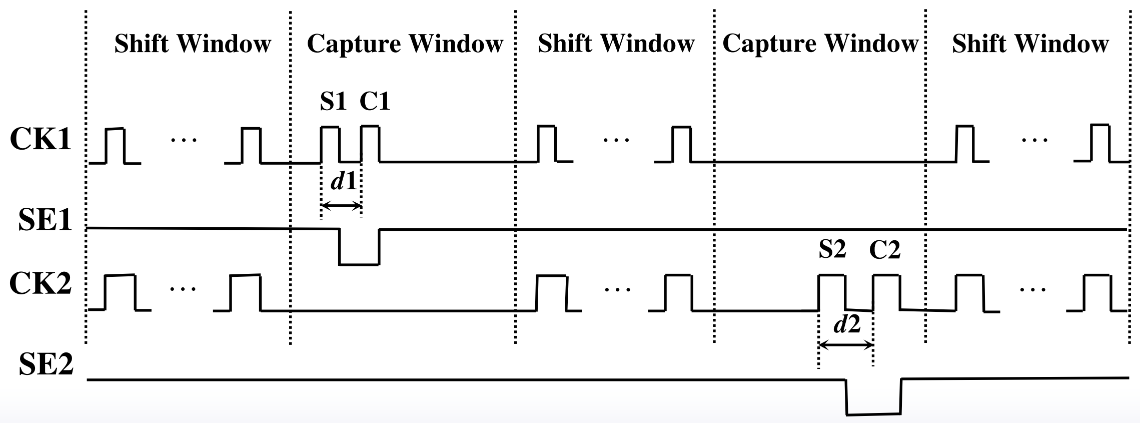
- An at-speed delay test technique
- Can address intra-clock-domain delay faults
- V1 and V2 correlated
  - May exercise functionally infeasible paths
- Three approaches (details in chapter 5)
  - One-hot skewed-load
  - Aligned skewed-load
  - Staggered skewed-load

One-Hot Skewed-Load

Tests all clock domains one by one by applying a-shift-followed by-a-capture pulses to detect intra-clock-domain delay faults.

Drawbacks:

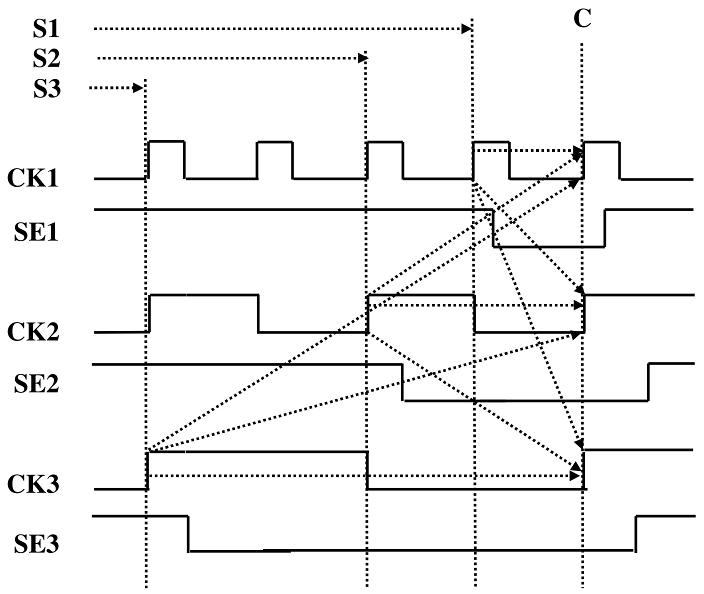
(1) Cannot detect inter-clock-domain delay faults
(2) Test time is long
(3) Single and global scan enable (GSE) signal can no longer be used.



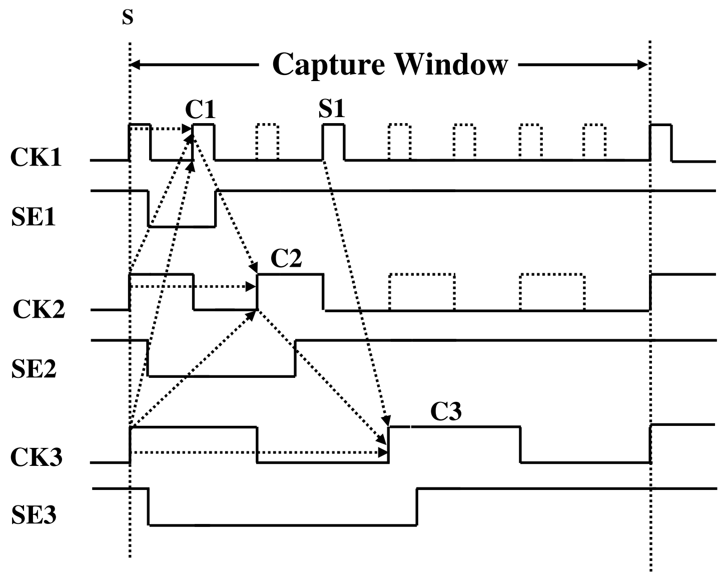
Aligned Skewed-Load

- Solve the long test time problem
- Test all intra-clock-domain and inter-clock-domain faults
- Need complex timing-control

Aligned Skewed-Load



Aligned skewed-load in capture

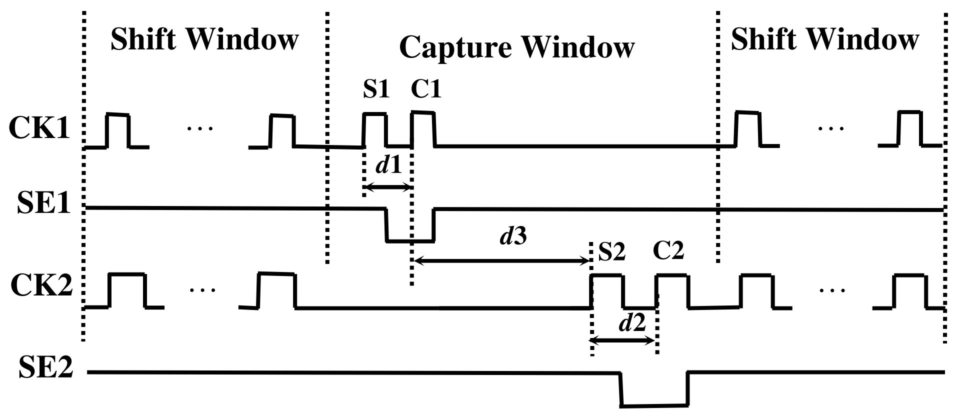


Aligned skewed-load in shift

Staggered Skewed-Load

When two test clocks cannot be aligned precisely, we can simply insert a proper delay to eliminate the clock skew. The two last shift pulses are used to create transitions and their output responses are caught by the next two capture.

Drawback: Need at-speed scan enable signal for each clock domain

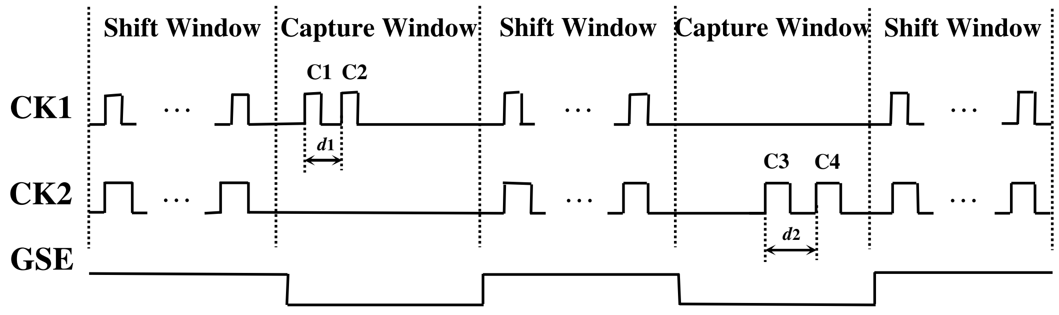


Staggered skewed-load

One-Hot Double-Capture

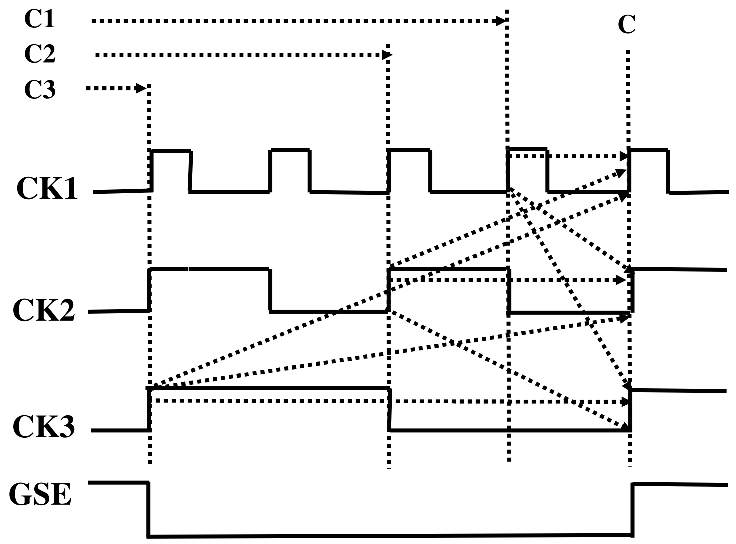
Test all clock domains one by one by applying two consecutive capture pulses at their respective domains' frequencies to test intra-clock-domain delay faults.

Benefit: true at-speed testing of intra-clock-domain delay faults
Drawbacks: (1) Cannot detect inter-clock-domain delay faults
(2) Test time is long

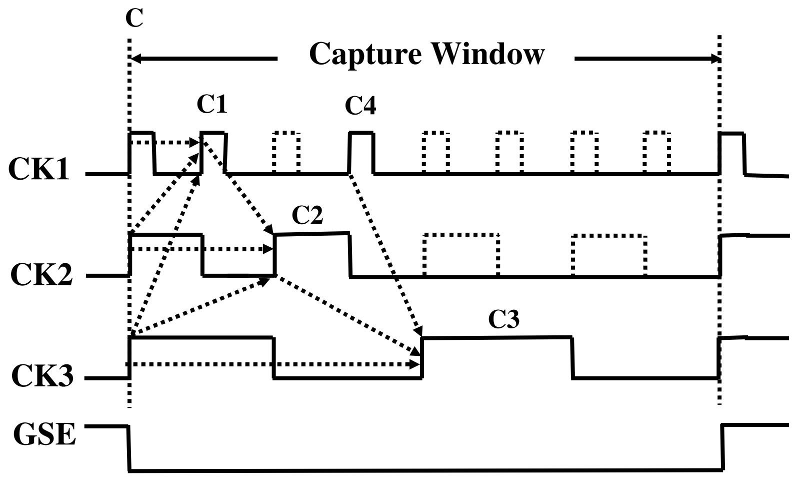


One-Hot double-capture

Aligned Double-Capture



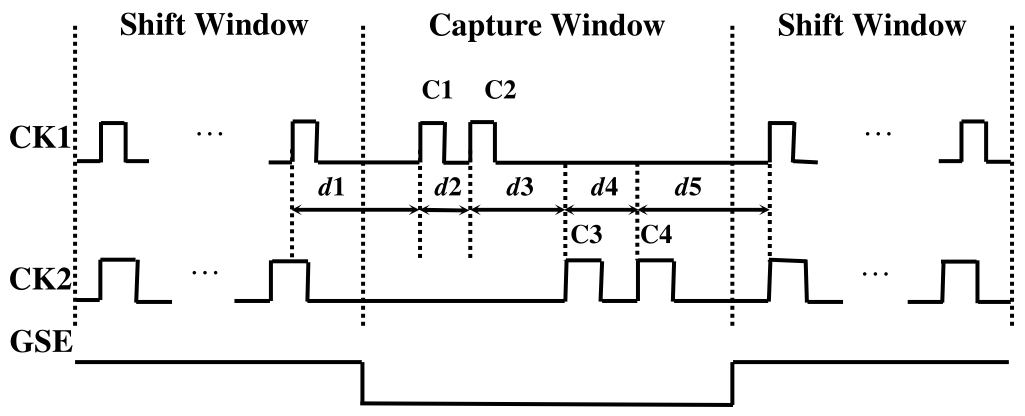
Aligned double-capture - I



Aligned double-capture - II

Staggered Double-Capture

In the capture window, two capture pulses are generated for each clock domain. The first two capture pulses are used to create transitions at the outputs of scan cells, and the output responses to the transitions are caught by the next two capture pulses, respectively.



Staggered double-capture

Enhanced Scan

- Enhanced-scan cells needed
- Larger cells to hold two values at each FF
- Can apply two uncorrelated vectors consecutively
  - Can achieve highest coverage, since all V1-V2 combinations are possible

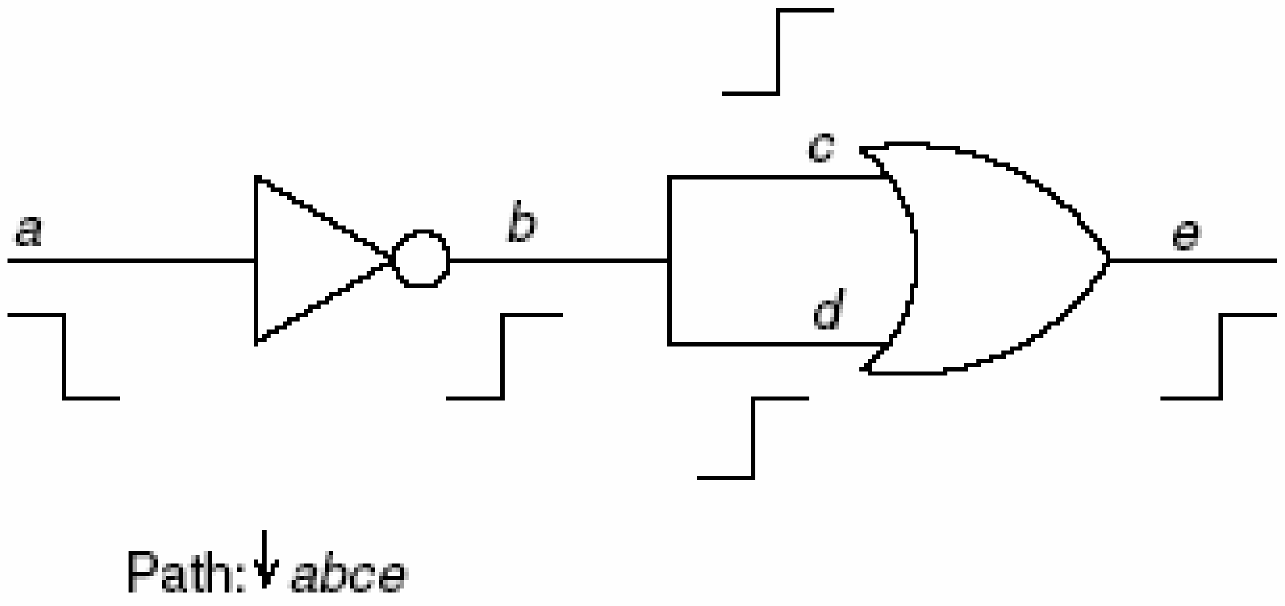
Path Delay Fault

- Models a combinational path in the circuit
  - Considers the cumulative effect of the delay along the path
  - On-inputs of a path
  - Off-inputs of a path
- A transition is launched at the start of the path, and a test must propagate the transition to the end of the path
  - Two faults associated with every path: rising and falling transition at the start of the path
- Number of paths can be exponential to the number of gates in the circuit
- Two vectors needed
  - V1: initialization vector
  - V2: launch and capture vector

Classification of Path Delay Faults

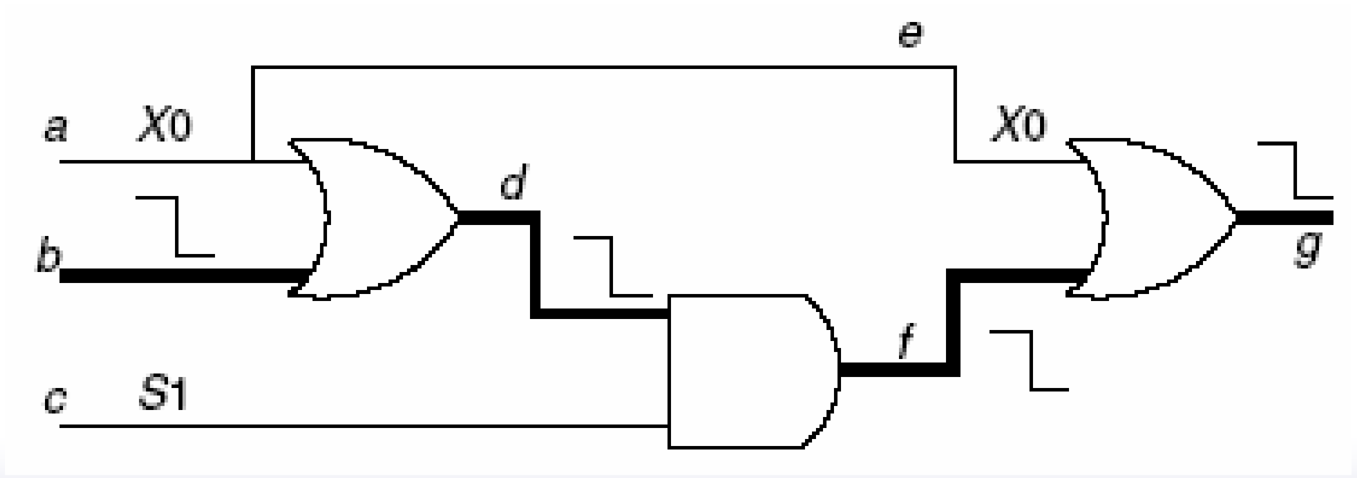
- Statically sensitizable: all off-inputs of a path P can be assigned to non-controlling values by some vector
- Single-path sensitizable: all off-inputs of a path can be set to non-controlling values for both vectors of a test
- False path: a transition cannot propagate from the start to the end of path
  - Not all necessary off-input values can be set to non-controlling values simultaneously

Statically Unsensitzable Path



Robustly Testable Paths

- Single-path sensitization is too stringent
- May not need to set all off-inputs to non-controlling values in V1 in order to propagate a transition
  - Highlighted path is robustly testable

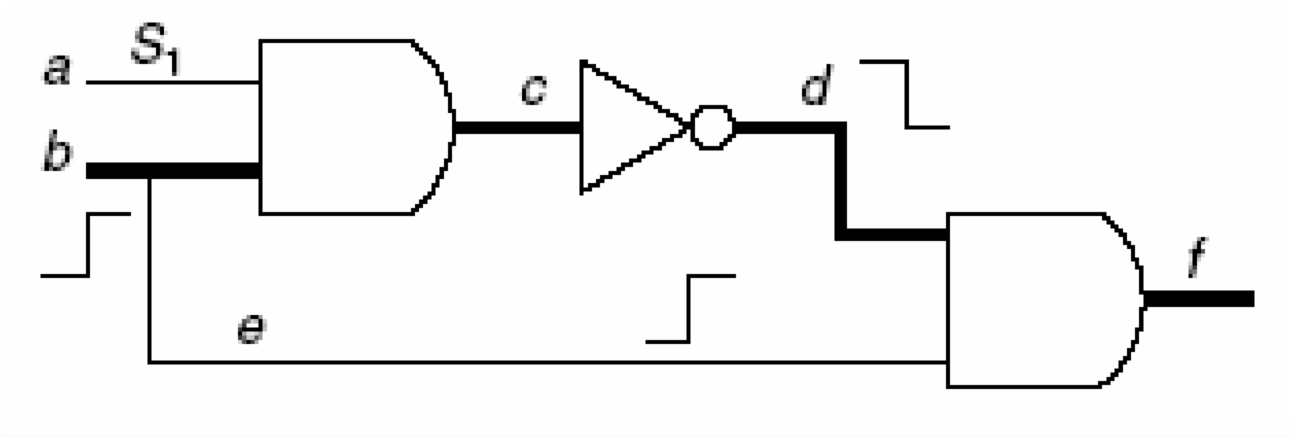


Robustly Testable (Cont.)

- If a path is robustly testable, then the corresponding test can verify the correctness of the path irrespective of other delays in the circuit
- Value criteria for robust testable path:
  - When the corresponding on-input of P has a controlling to non-controlling transition, the value in the first vector for the off-input can be X with the value for the off-input as a non-controlling value in the second vector.
  - When the corresponding on-input of P has a non-controlling to controlling transition, the values for the off-input must be a steady non-controlling value for both vectors.

Non-robustly Testable Path

- Not all paths are robustly testable
- Further relax requirements for V1
- Test is valid if circuit has no other delay faults
  - Highlighted path is non-robustly testable

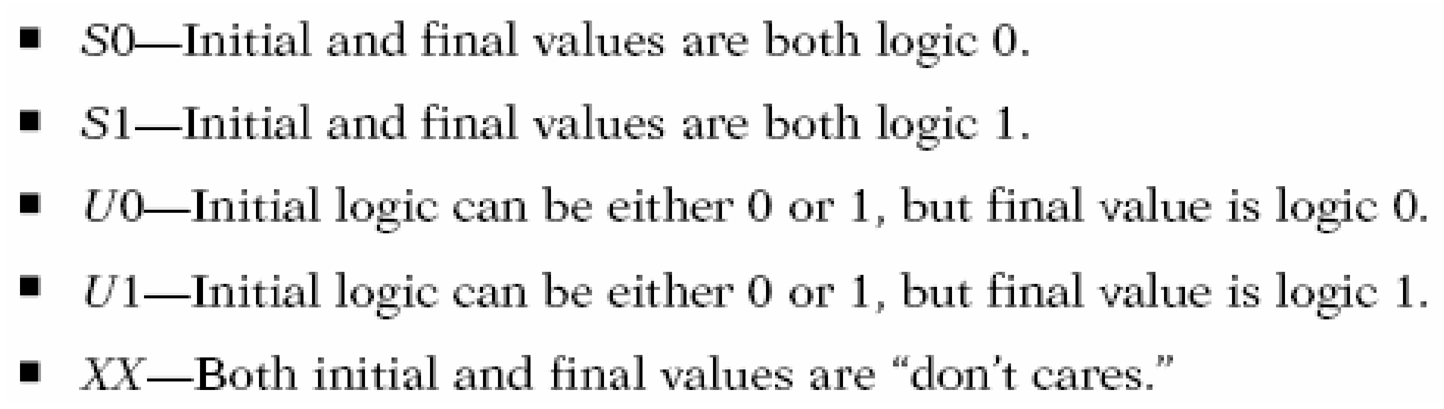


Non-robust Path (cont.)

- Non-robust test only valid if no other delay fault is present in the circuit
- Value criteria for non-robust testing:
  - Irrespective of the transition on the on-input, the value in the first vector for the off-input can be X, with the value for the off-input as a non-controlling value in the second vector.

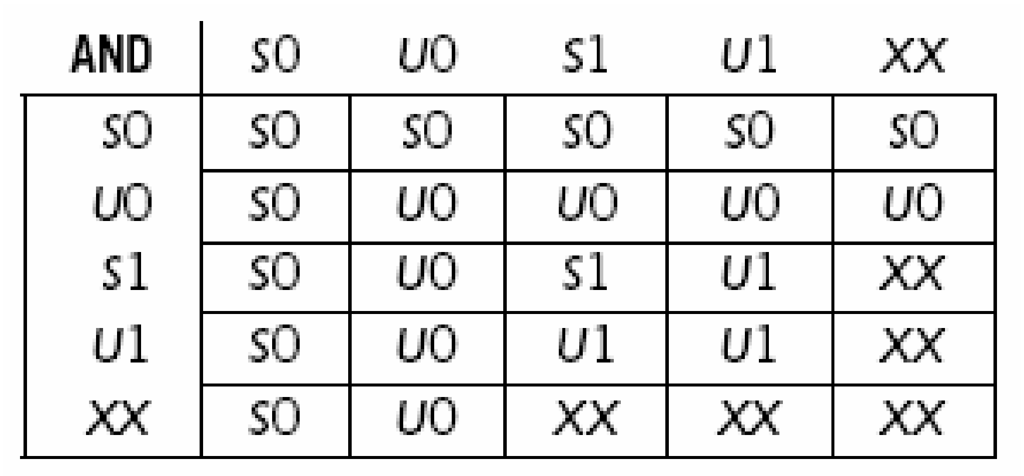
ATPG for Path-Delay Faults

- Can use new value algebra to consider both vectors simultaneously during ATPG

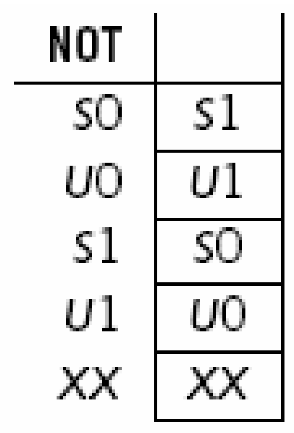


- $S0$—Initial and final values are both logic 0.
- $S1$—Initial and final values are both logic 1.
- $U0$—Initial logic can be either 0 or 1, but final value is logic 0.
- $U1$—Initial logic can be either 0 or 1, but final value is logic 1.
- XX—Both initial and final values are “don’t cares.”

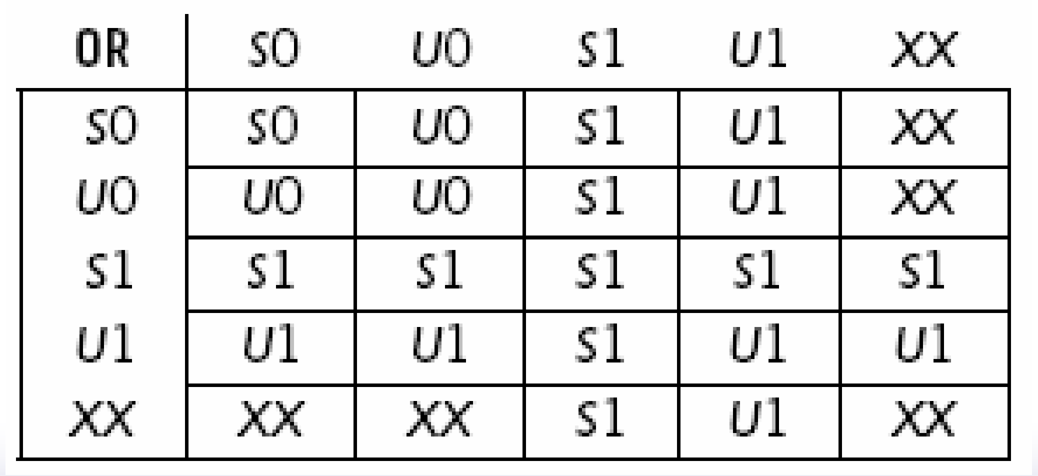
Boolean Operations



AND	S0	U0	S1	U1	XX
S0	S0	S0	S0	S0	S0
U0	S0	U0	U0	U0	U0
S1	S0	U0	S1	U1	XX
U1	S0	U0	U1	U1	XX
XX	S0	U0	XX	XX	XX



NOT	
S0	S1
U0	U1
S1	S0
U1	U0
XX	XX



OR	S0	U0	S1	U1	XX
S0	S0	U0	S1	U1	XX
U0	U0	U0	S1	U1	XX
S1	S1	S1	S1	S1	S1
U1	U1	U1	S1	U1	U1
XX	XX	XX	S1	U1	XX

RESIST [1994]

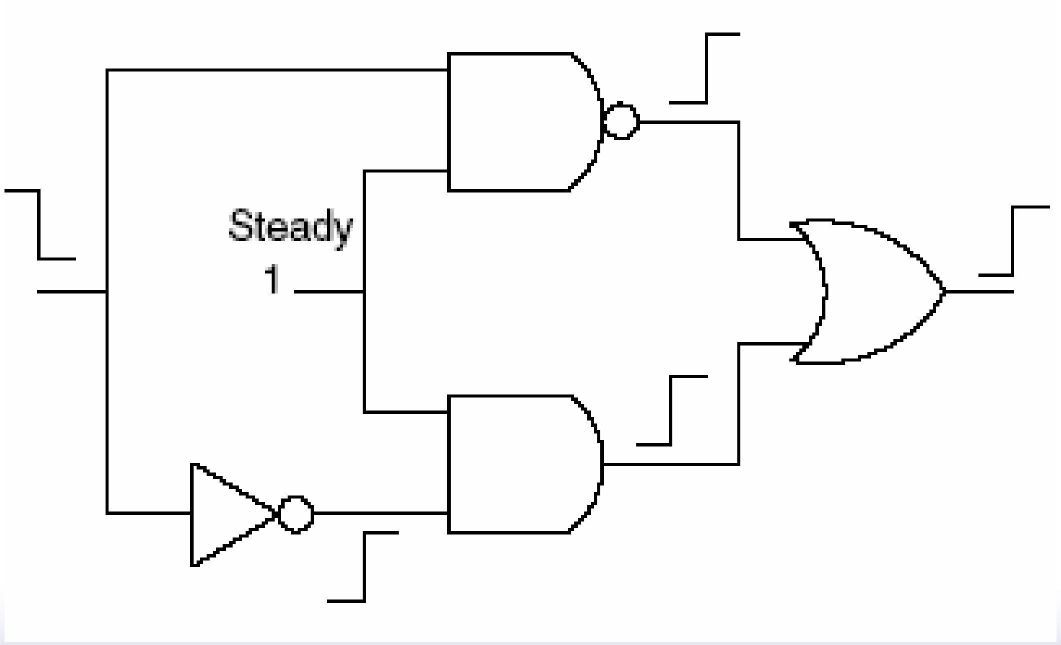
- Recursion-based path-delay-fault ATPG
  - Starts at a PI
  - Depth-first-search through the circuit along each path
  - Generate a test for each path
  - Takes advantage of many paths that share common path-segments

Transition Fault Model

- Assumes a large/gross delay is present at a circuit node
- Irrespective of which path the effect is propagated, the gross delay defect will be late arriving at an observable point
- Most commonly used in industry
  - Simple and number of faults linear to circuit size
  - Also needs 2 vectors to test
- Node x slow-to-rise (x-STR) can be modeled simply as two stuck-at faults
  - First time-frame: x/1 needs to be excited
  - Second time-frame: x/0 needs to be excited and propagated

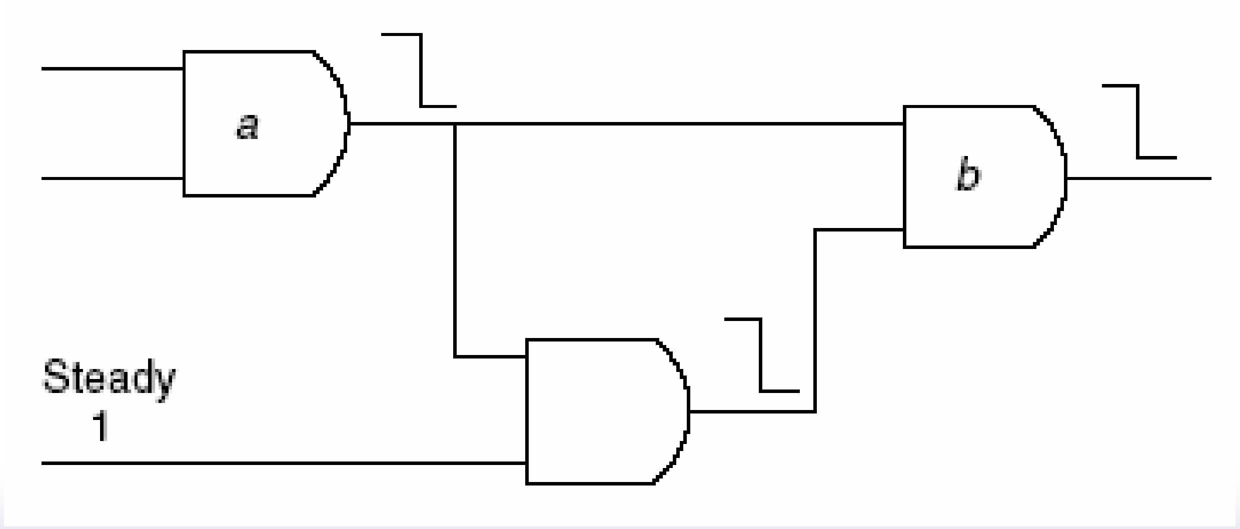
Transition Fault Properties

- Lemma: a transition fault may be launched robustly, non-robustly, or neither
- Example: STR at output of OR gate



Transition Fault Properties (cont.)

- Lemma: a transition fault may be propagated robustly, non-robustly, or neither
- Example: STF at output of gate ‘a’



Transition Fault Testing with Stuck-At ATPG

- Simply treat each transition fault as two stuck-at faults
- Can test it with broadside, skewed-load, or enhanced scan

Transition Fault Testing with Stuck-At Vectors for Enhanced Scan

- First perform Stuck-at ATPG for stuck-at faults
- Then build a dictionary for the vectors generated
- Use the dictionary to identify vector-pairs for each transition fault

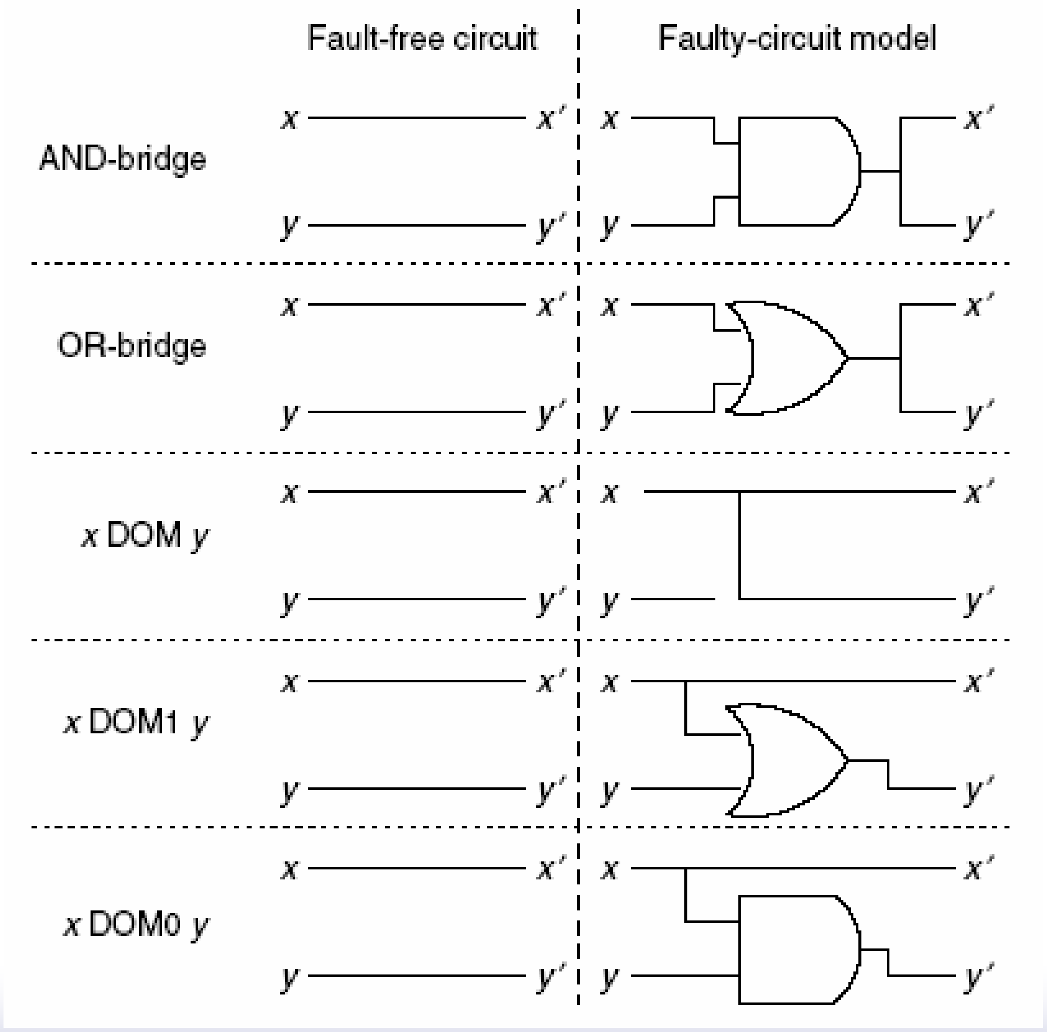
Properties of Chaining Stuck-at vectors

- Consider a sequence of 3 vectors: (vi, vj, vk) forming two vector-pairs (vi, vj) and (vj, vk)
- Theorem: Transition faults detected by (vi,vj) and pattern (vj,vk) are mutually exclusive.
- Why?

Bridging Fault

- Models shorts between two circuit nodes
- The bridge fault is not excited unless the two circuit nodes have opposing logic values
- Faulty value depends on the bridge-fault type:
  - AND bridge: faulty value is the AND of the two involved nodes' values
  - OR bridge: faulty value is the OR of the two involved nodes' values
  - X Dom y: value of x dominates
  - X Dom1 y: x dominates y if x=1
  - X Dom0 y: x dominates y if x=0

Illustration of the Bridge Fault Models



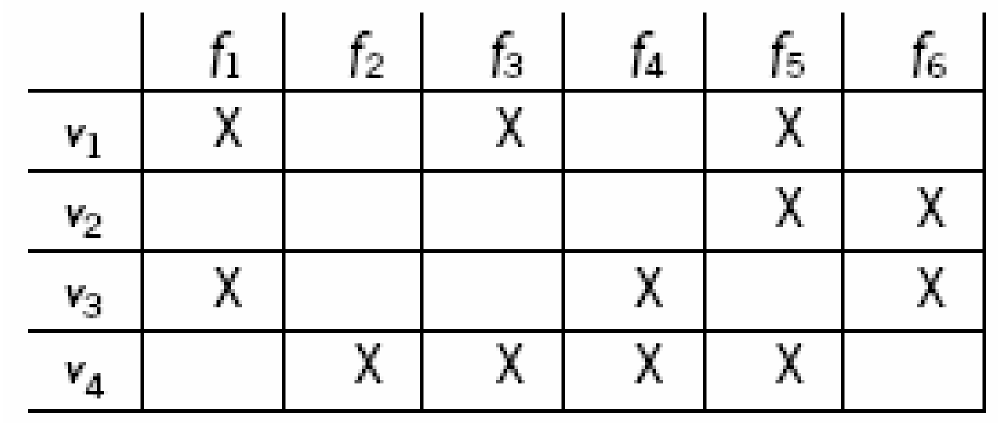
Bridging Fault ATPG

- Modeled as a constrained stuck-at ATPG
- Consider AND-bridge(x,y), we can do either:
  - Detect x/0 with setting y=0
  - Detect y/0 with setting x=0
- Conventional stuck-at ATPG can be modified to handle bridge faults

Combinational Test Set Compaction

- Want to reduce the test set size to reduce test data storage and test application time
- Idea: find a minimal set of vectors that can detect every fault
- First build a detection dictionary

Test Set Compaction (cont.)



	f_1	f_2	f_3	f_4	f_5	f_6
v_1	X		X		X	
v_2					X	X
v_3	X			X		X
v_4		X	X	X	X	

- Essential vector: a vector that detects some faults that no other vector can detect
  - V4 is essential
- A set covering algorithm is applied to find a min test set such that every fault is covered

Test Set Compaction (cont.)

- If vectors are incompletely specified
  - Some vectors may be compatible: 1X0X and X100 are compatible. Just one vector 1100 is sufficient
- Reverse-order simulation
  - Simulate the test set in reverse order, some vectors may no longer be needed

Sequential Test Set Compaction

- Much more complex than combinational test set compaction due to memory elements
- Idea: remove subsequences that are unnecessary to detect faults
- Vector-restoration algorithm

N-Detect ATPG

- Idea: detect every fault at least N times
  - N vectors that detect a fault must be different
- Although the same fault coverage, can significantly enhance the defect coverage
  - If x/0 is detected 2 times, one with y=1, and the other with y=0, then the AND-bridge fault of (x,y) would have been detected by the second test
- ATPG can be modified to N-Detect ATPG

Finite-State-Machine Testing

- A form of high-level testing
- Aim to generate a test set that visits
  - Every state in the FSM
  - Every transition/edge in the FSM
- Idea: a fault present in the circuit must alter the functionality of the FSM somehow

Concluding Remarks

- Covered a number of topics
  - Theoretical Foundations
  - Combinational & sequential ATPG
  - Untestable fault identification
  - Simulation-based & hybrid ATPG
  - Delay testing
  - Bridging fault testing
  - Compaction, N-Detect, FSM testing
- Challenges Ahead
  - Fast untestable fault identification essential to remove large numbers of stuck-at, bridge, delay faults
  - Sequential ATPG remains an open research area