Week 4: Course Material

Test Generation
Lecture 17

Test Generation

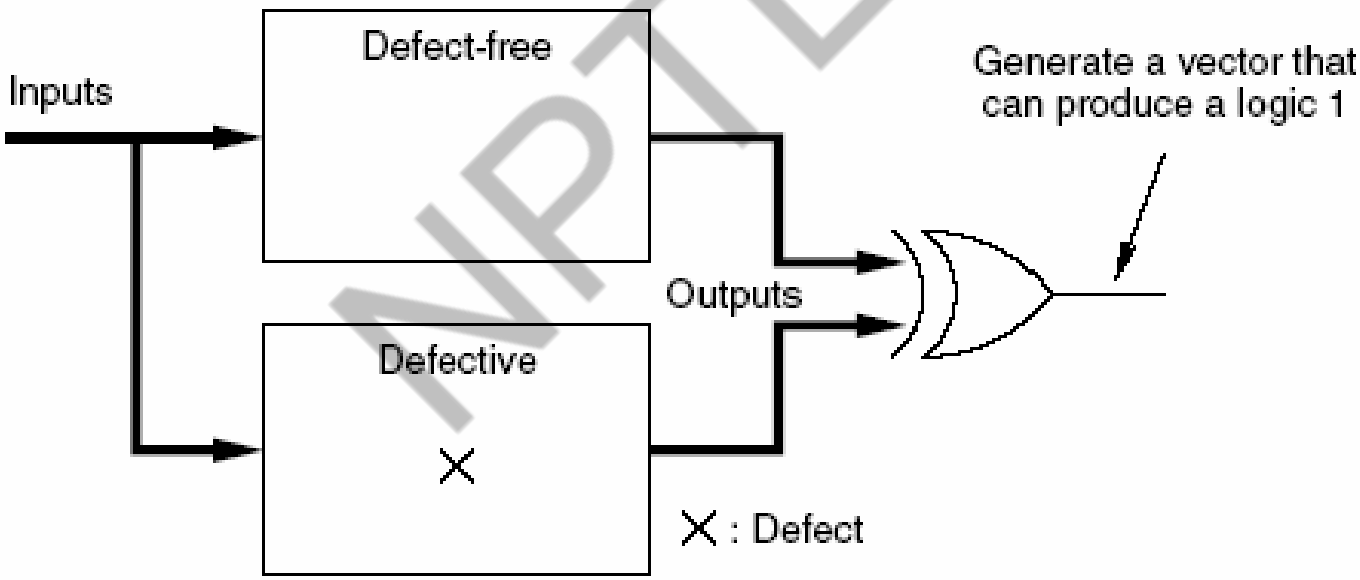
- Introduction
- Random Test Generation
- Theoretical Foundations
- Deterministic Combinational ATPG
- Untestable Fault Identification
- Simulation-based ATPG
- ATPG for Delay and Bridge Faults
- Other Topics in Test Generation
- Concluding Remarks

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

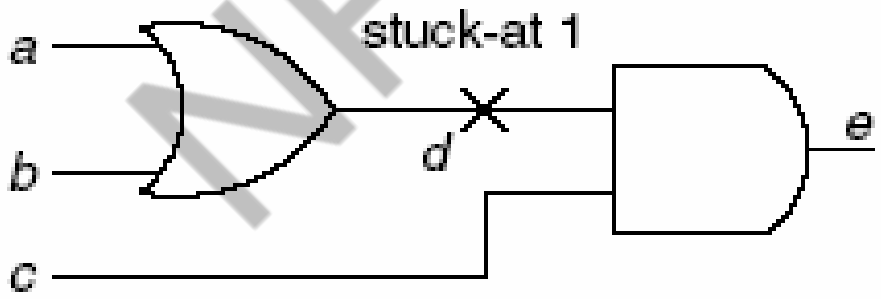


Fault Models

- 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

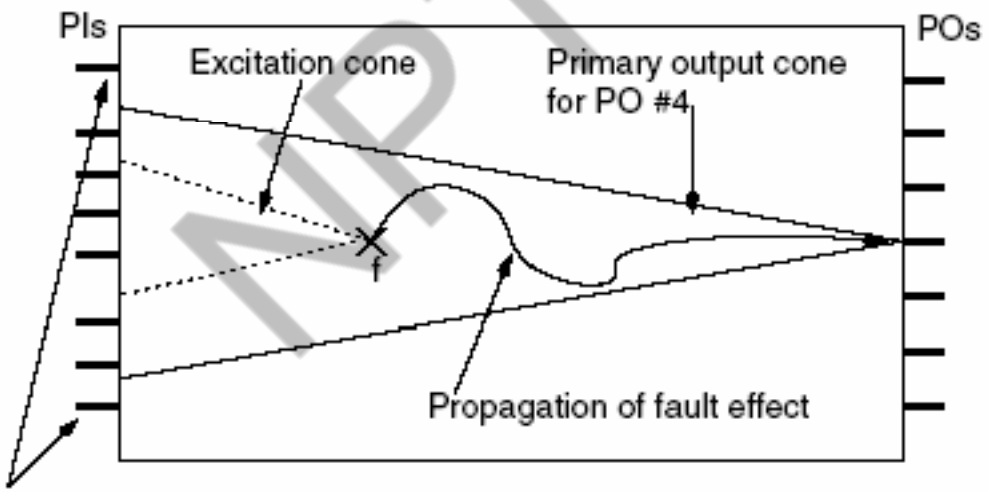
Prob of Fault Detection (Cont.)

- 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

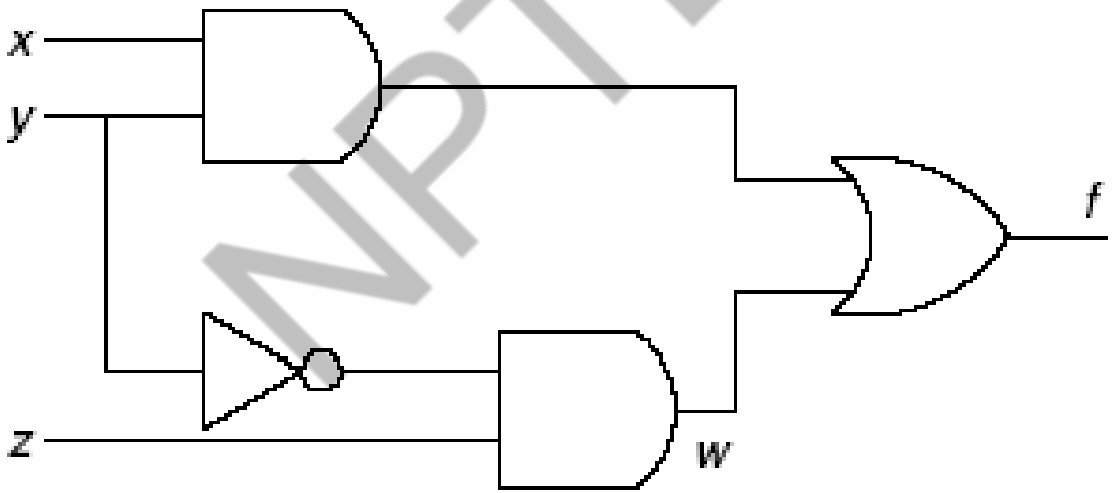
- Exhaustive Testing
  - Apply 2^n patterns to an *n*-input combinational *circuit under test* (CUT)
  - Guarantees all detectable faults in the combinational circuits are detected
  - Test time maybe be prohibitively long if the number of inputs is large
  - Feasible only for small circuits
- Pseudo-exhaustive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

Week 4: Course Material

Test Generation (Contd.)
Lecture 18

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*



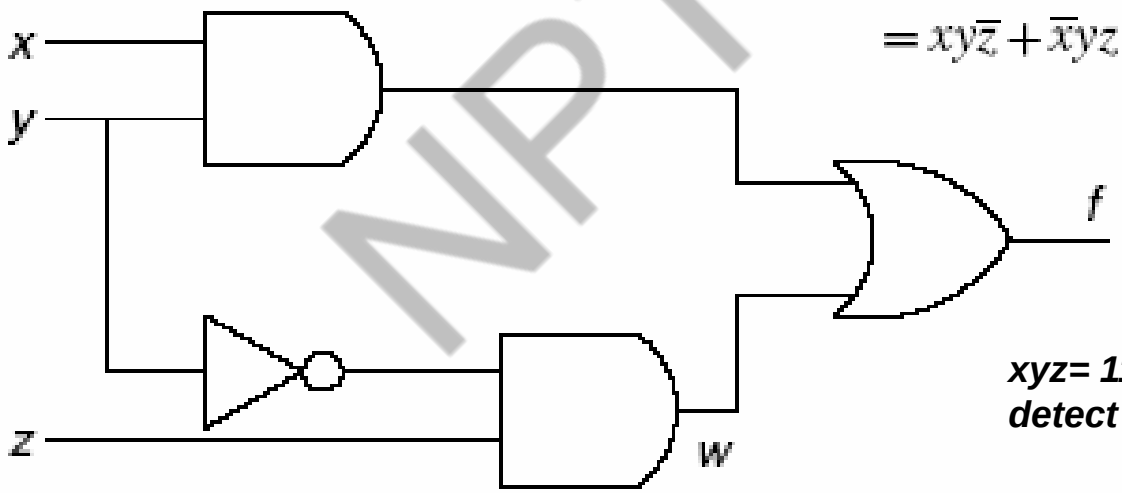
Boolean Difference Continued

- *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,*

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



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

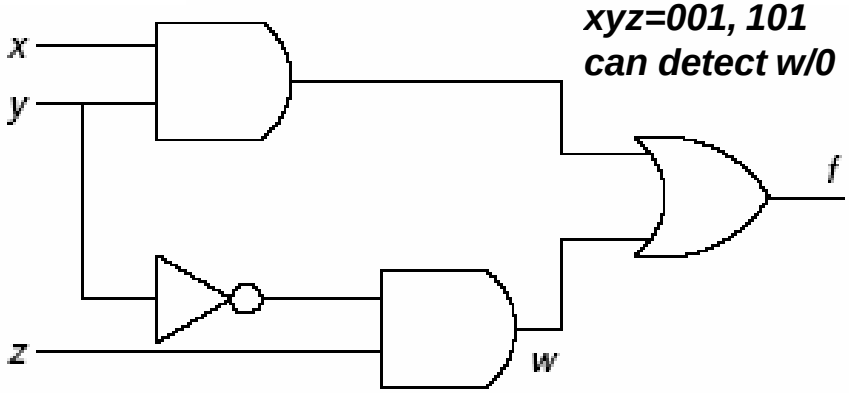
$$w \cdot \overline{x} + w \cdot \overline{y} = 1$$

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

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

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

xyz=001, 101 can detect w/0



A Third Example

- Fault: *z/0*

$$z \cdot \frac{df}{dz} = 1$$

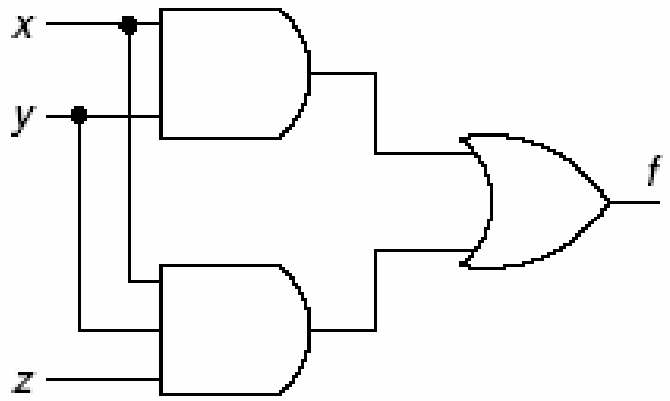
$$\Rightarrow \quad z \cdot (f(z=1) \oplus f(z=0)) = 1$$

$$\Rightarrow \quad z \cdot (xy \oplus xy) = 1$$

$$\Rightarrow \quad z \cdot 0 = 1$$

$$\Rightarrow \quad \text{UNSATISFIABLE}$$

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*

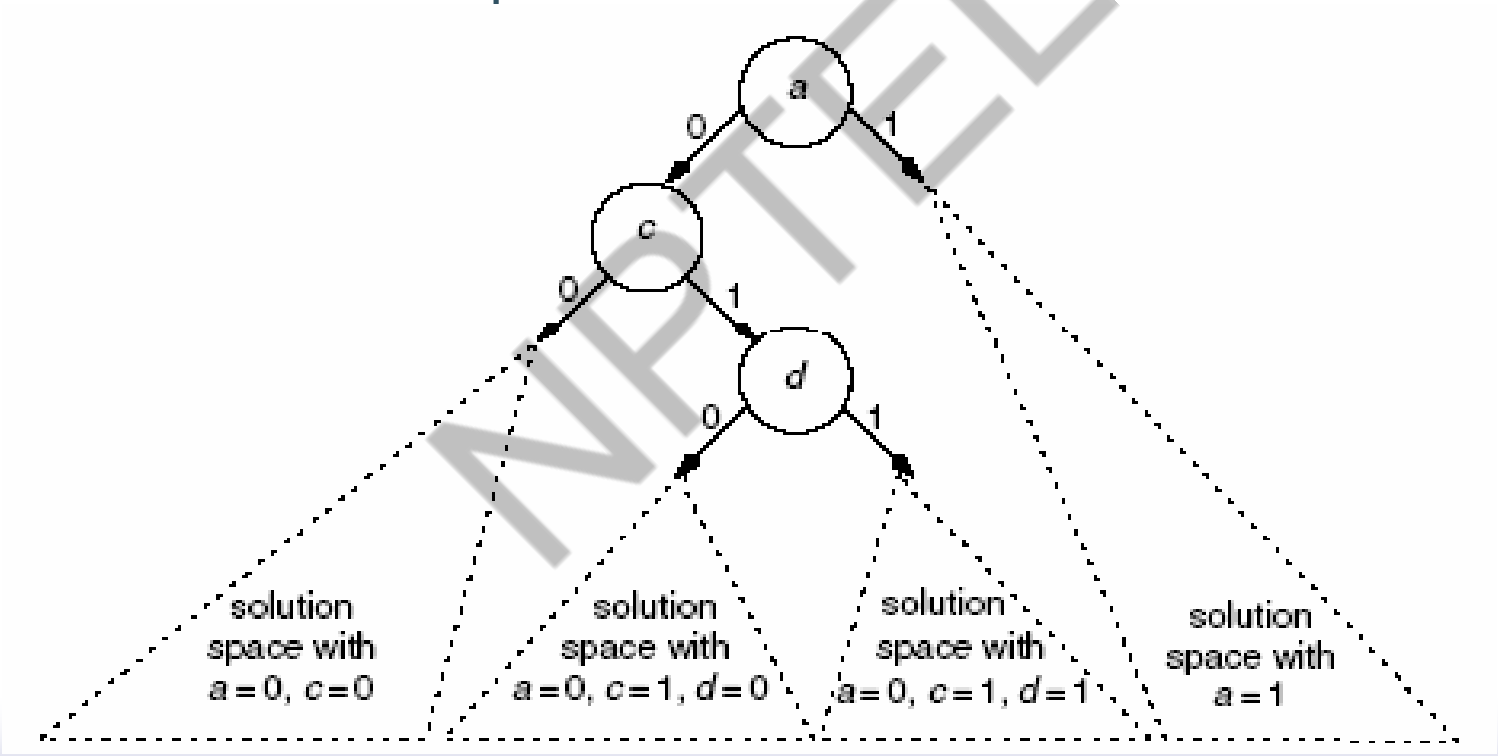
AND	0	1	D	$\overline{D}$	X
0	0	0	0	0	0
1	0	1	D	$\overline{D}$	X
D	0	D	D	0	X
$\overline{D}$	0	$\overline{D}$	0	$\overline{D}$	X
X	0	X	X	X	X

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

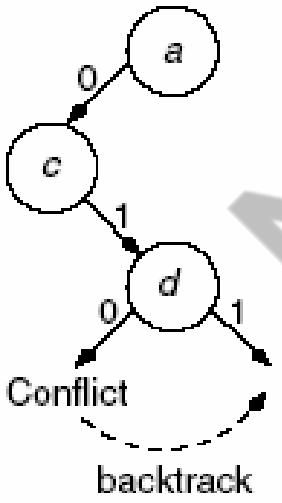
Decision Tree for Branch-and-Bound Search

- The ATPG systematically and implicitly searches the entire search space



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)

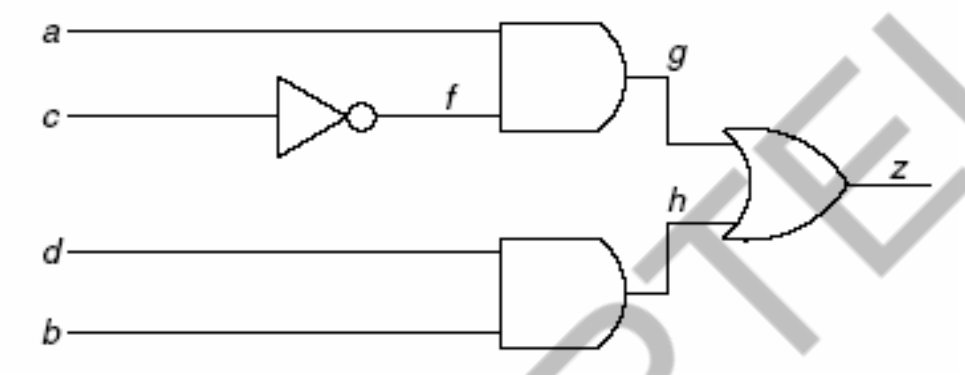
1: initialize circuit by setting all values to X;
2: JustifyFanoutFree($C, g, \overline{v}$); /* excite the fault by justifying line g to $\overline{v}$ */
3: PropagateFanoutFree(C, g); /* propagate fault-effect from g to a PO */

The Justify Routine

Algorithm 3 JustifyFanoutFree(C, g, v)

```
 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
```

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

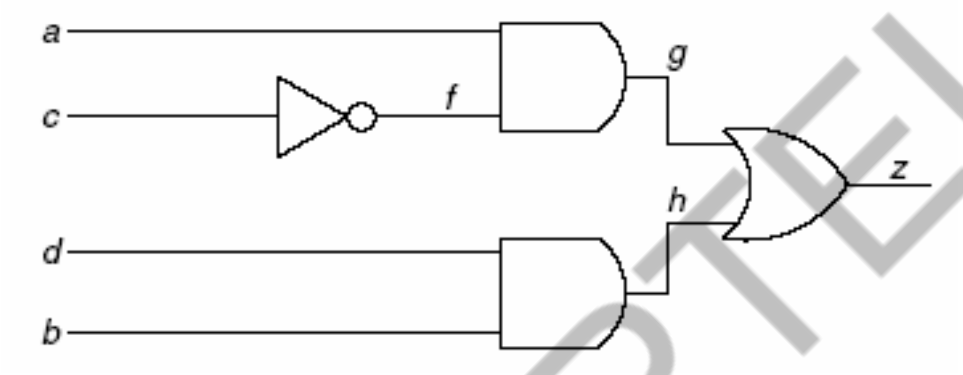
Algorithm 4 PropagateFanoutFree(C, g)

```
 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 ≠ 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);
```

Week 4: Course Material

Test Generation
Lecture 19

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

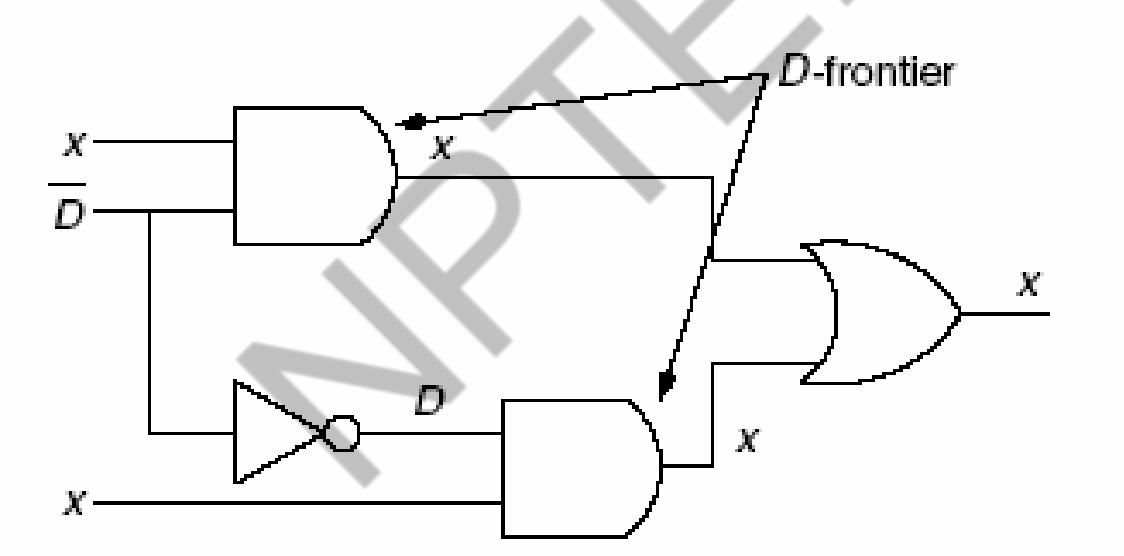
- Can handle arbitrary combinational circuits, with internal fanout structures
- Main idea: always maintain a non-empty D-frontier and try to propagate at least a fault effect to a primary output
- Initially, all circuit nodes are X, except for the fault cite, where a fault effect (D or D-bar) is placed.

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

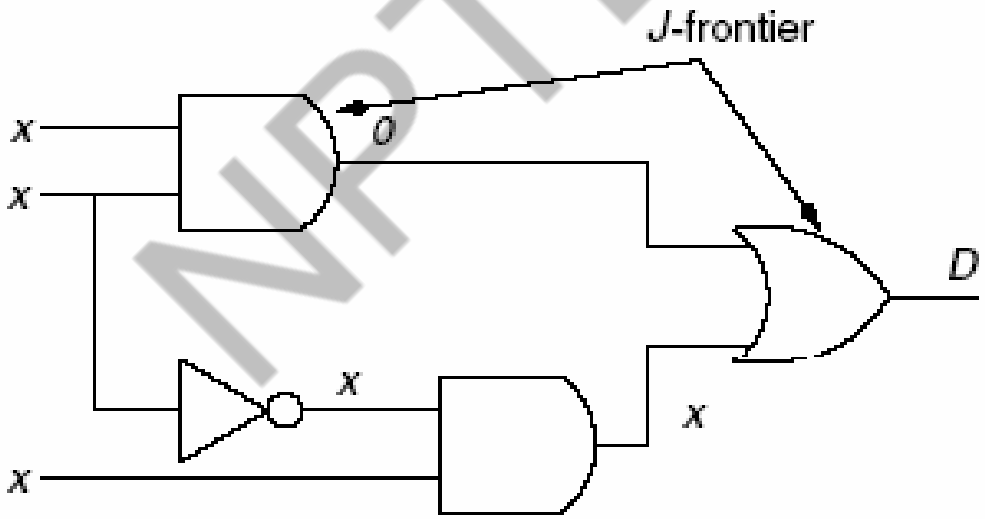
D-Frontier Example

- The D-frontier contains 2 gates



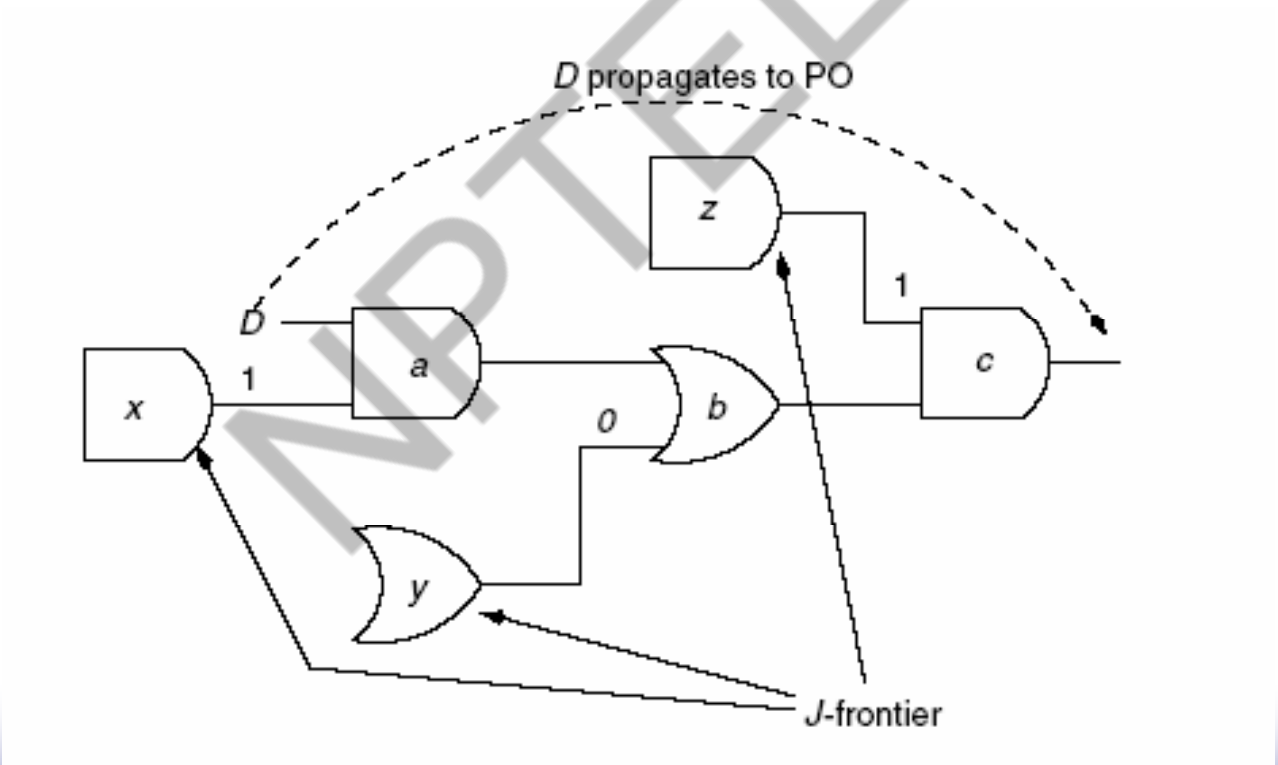
J-Frontier Example

- 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

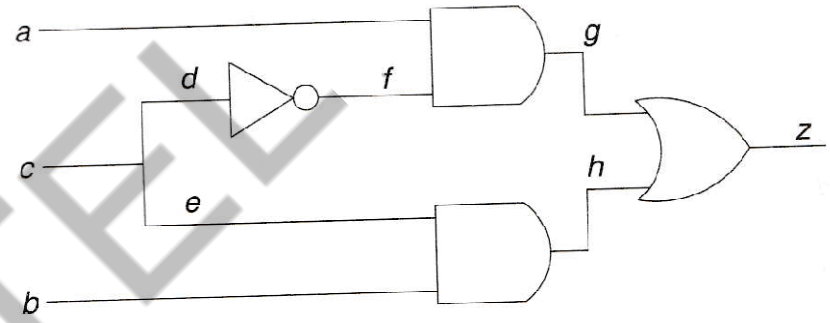
Algorithm 5 *D*-Algorithm(C, f)

```
1: initialize all gates to don't-cares;
2: set a fault-effect (D or D̄) on line with fault f and insert it to the D-frontier;
3: J-frontier = φ;
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
```

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

```
1: if there is a conflict in any assignment or D-frontier is Ø 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 Ø 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);
```

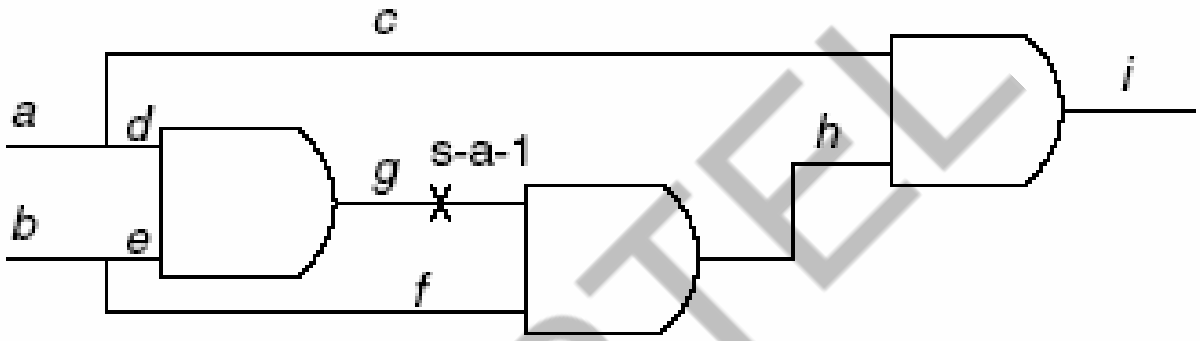
D Algorithm Example



Target *f* stuck-at-0

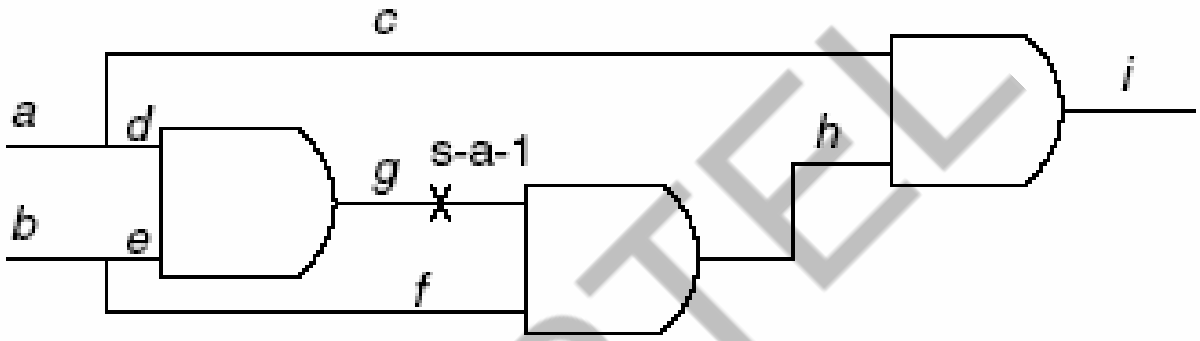
Initialize all gates to X
Places D on line *f*
Propagate fault effect to *z*
Places *a*=1 in J-frontier, followed by
h=0
Fault effect has reached primary output
Try to justify entries in J-frontier
a is already justified as it is primary input
For *f*=D, *d* = 0, making *c*=0
For *h*=0, either *e*=0 or *b*=0 is sufficient
Test found.

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.)



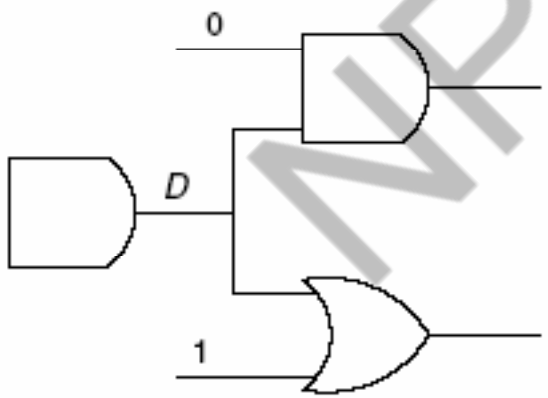
- Now justify every value in J-Frontier via branch-and-bound search
  - Must not make D-frontier empty or conflict with other J-frontier values
  - Otherwise backtrack
- Result: g/1 is untestable

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

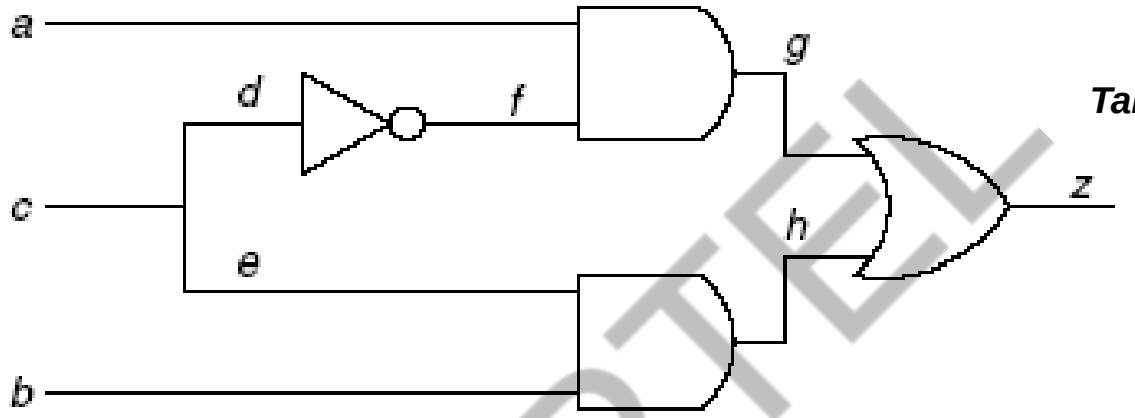
Algorithm 10 backtrace(C)

```
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);
```

Week 4: Course Material

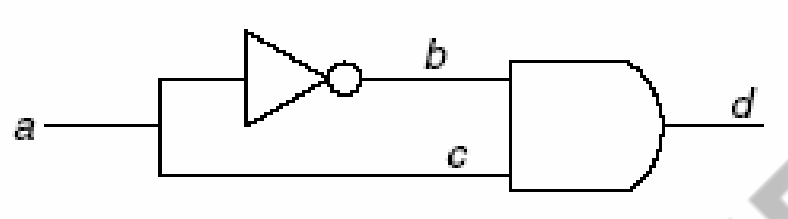
Test Generation
Lecture 20

PODEM Example



- 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.

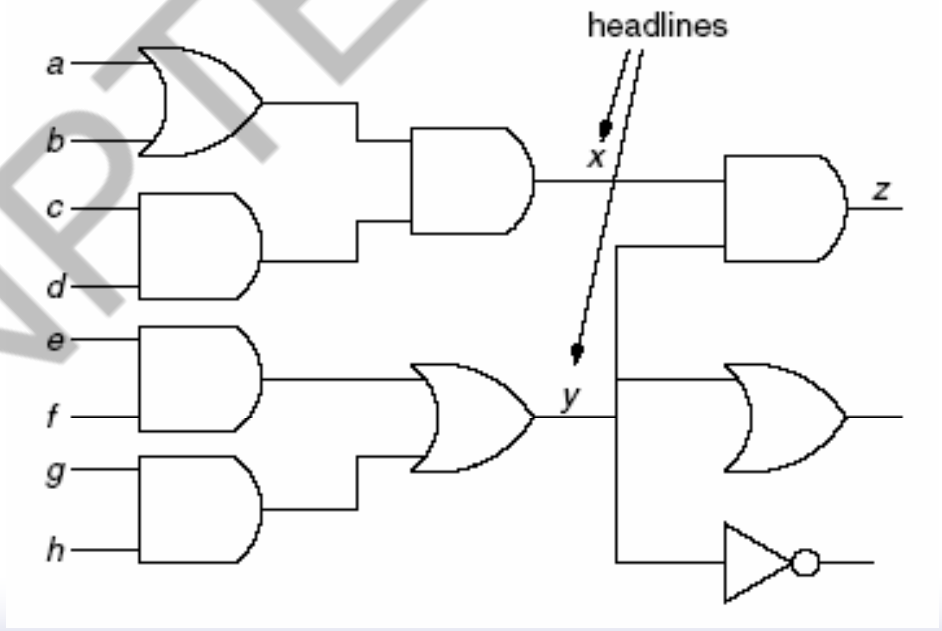
FAN

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

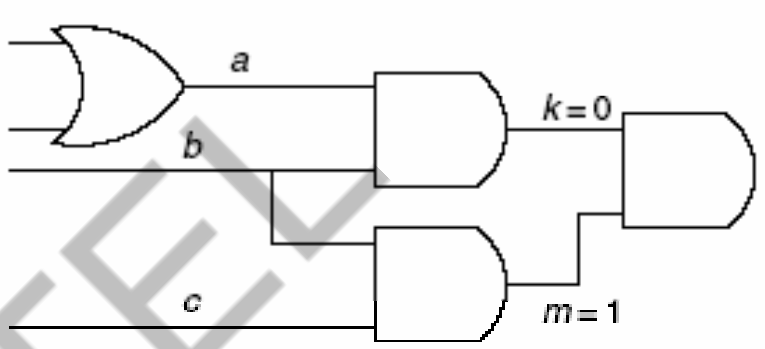
Headlines

- 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

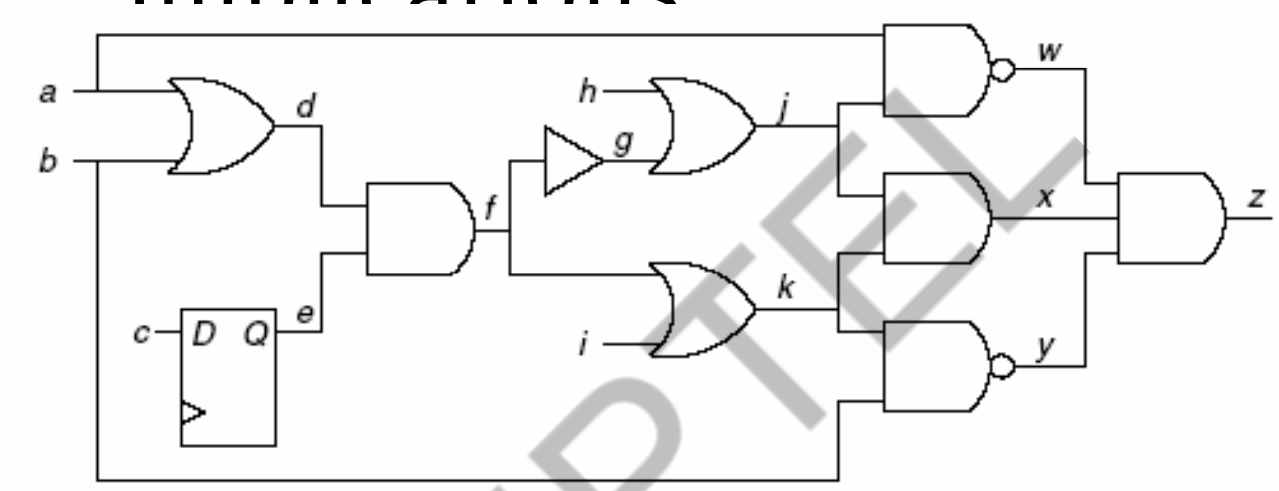


- Objectives: {k=0, m=1}
- Backtrace from k=0 may favor b=0, but simulate(b=0) would violate the second objective m=1!
- Makes backtrace more intelligent to avoid future conflicts

Static Logic Implications

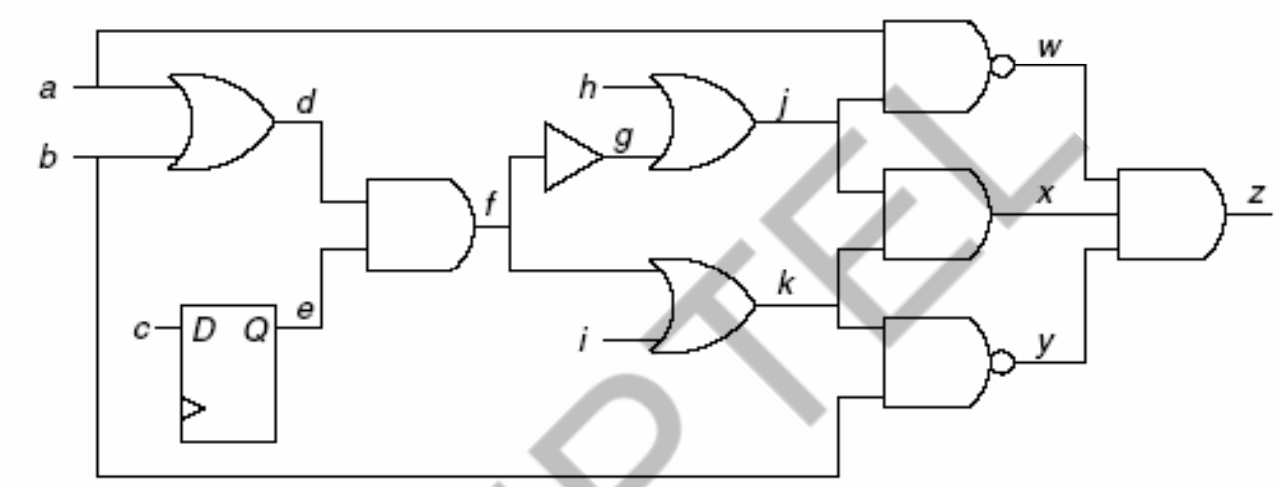
- Can help ATPG make better decisions
- Avoid conflicts
- Reduce the number of backtracks
- Idea: what is the effect of asserting a logic value to a gate on other gates in the circuit?

Direct Implications



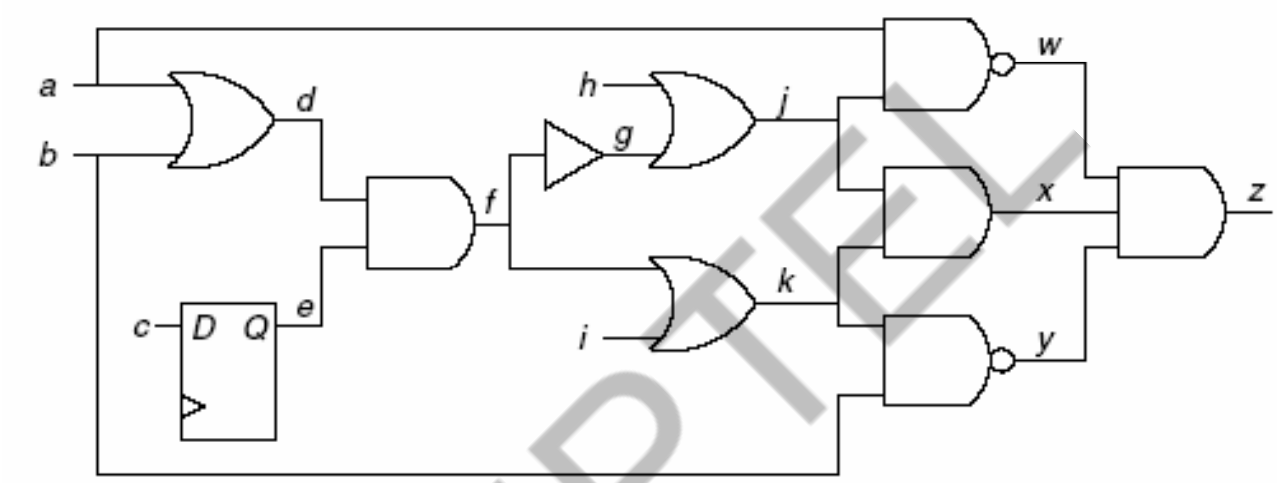
- Direct implications for f=1:
  - {d=1, e=1, g=1, j=1, k=1}
- Direct implications for j=0:
  - {h=0, g=0, f=0, w=1, w=0, z=0}

Indirect Implications



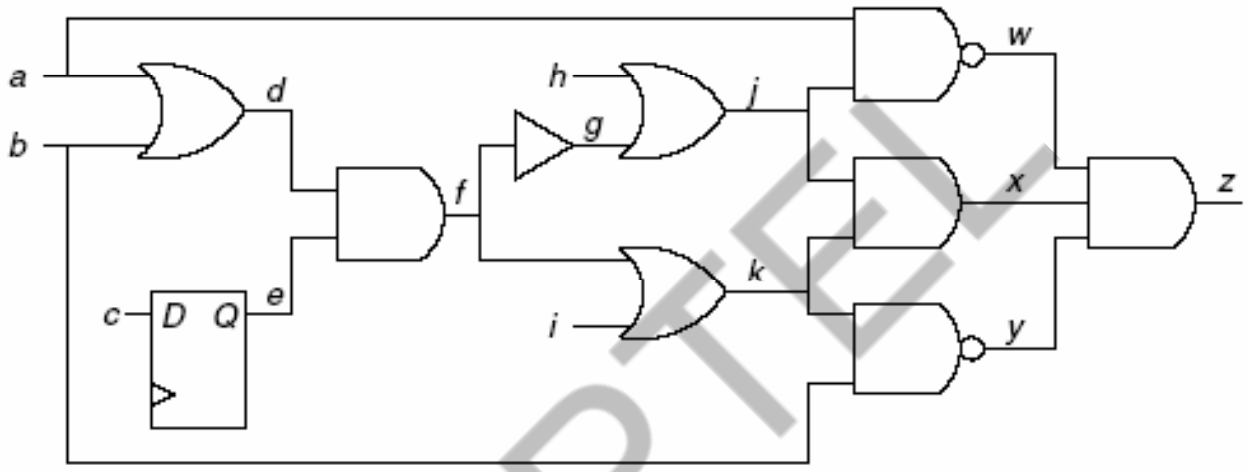
- Direct implications for f=1:
  - {d=1, e=1, g=1, j=1, k=1}
- Indirect Implications for f=1 obtained by simulating the direct implications of f=1:
  - {x=1}
- This is repeated for every node in the circuit

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}

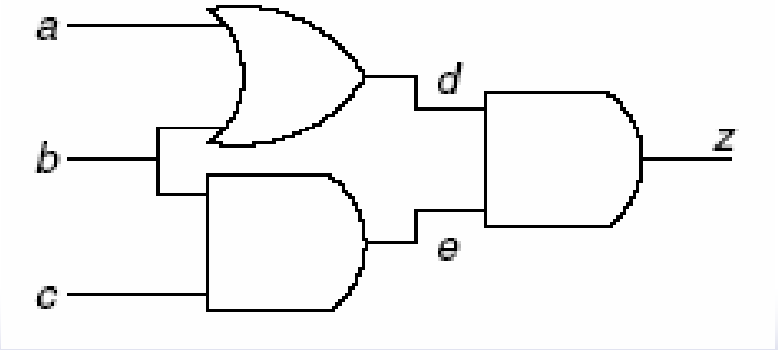
Extended Backward Implications



- 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

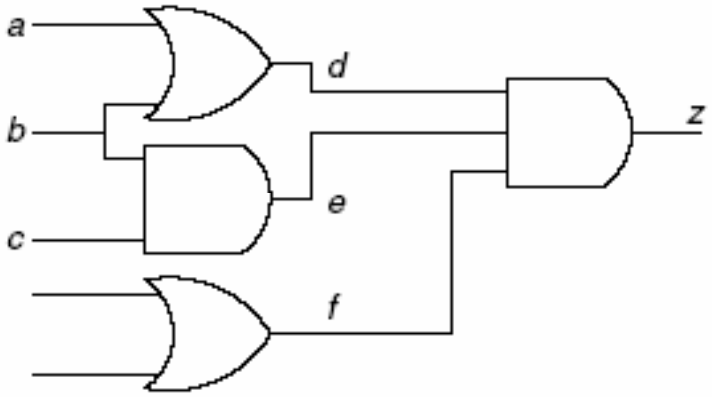
- Similar to Static Logic Implications, but has some signals already assigned values
- Suppose c=1 has already been assigned
  - Then to obtain z=0, b must be 0
  - This is the intersection of having either d=0 or e=0 in the presence of c=1



Another Dynamic Implications

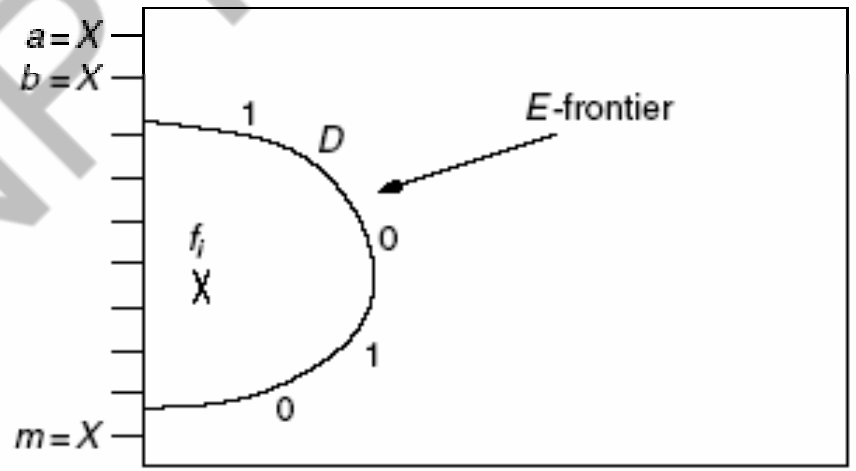
Example

- 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]



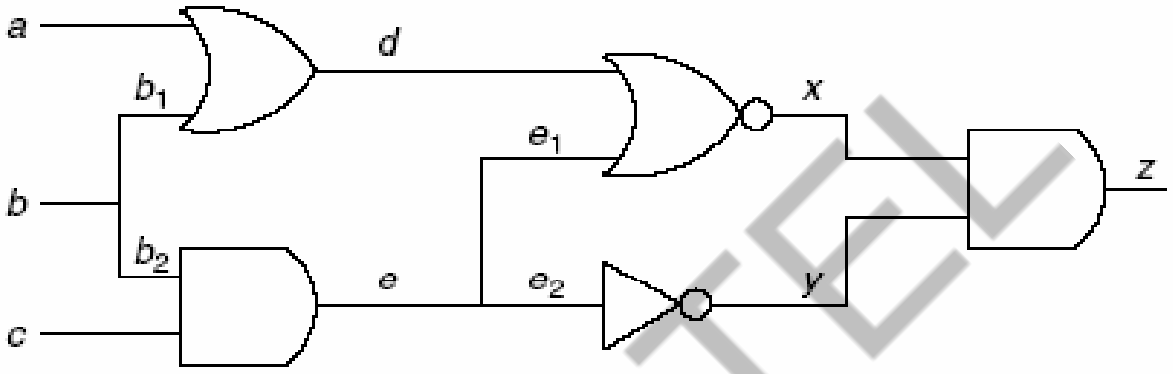
Fast Untestable Fault Identification

- 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]

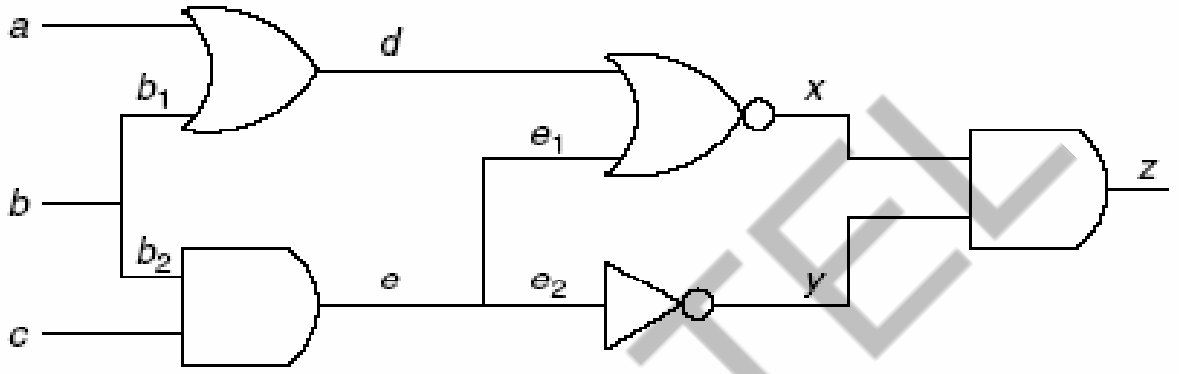
- Based on conflict analysis
- S0 = set of faults that are untestable when signal s=0
  - These faults must require s=1 to be detectable
- S1 = set of faults that are untestable when signal s=1
  - These faults must require s=0 to be detectable
- Intersection of S0 and S1 are definitely untestable
 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

FIRE Example



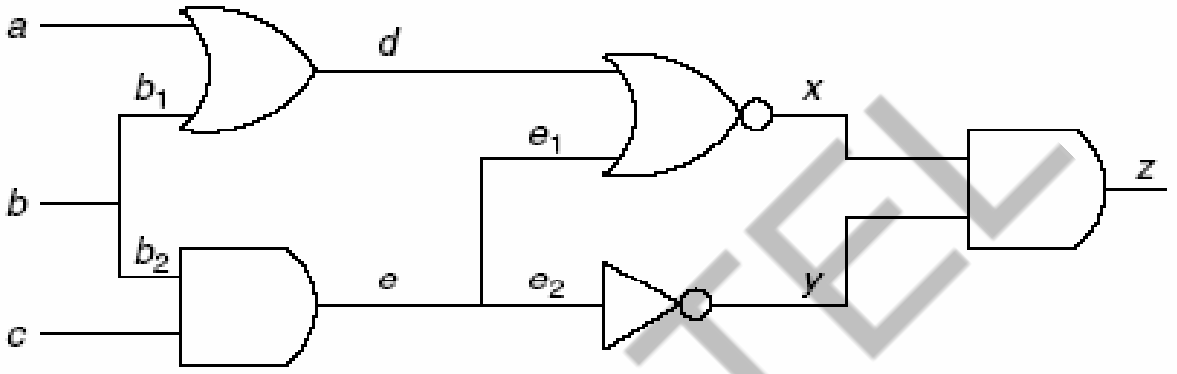
- Impl[b=1] = {b=1, b1=1, b2=1, d=1, x=0, z=0}
- Faults unexcitable when b=1: {b/1, b1/1, b2/1, d/1, x/0, z/0}
- Faults unobservable when b=1: {a/0, a/1, e1/0, e1/1, y/0, y/1, e2/0, e2/1}
- Faults undetectable when b=1: {a/0, a/1, b/1, b1/1, b2/1, d/1,e1/0, e1/1, e2/0, e2/1, x/0, y/0, y/1, z/0}

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}

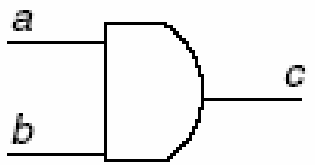
Week 4: Course Material

Test Generation
Lecture 21

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

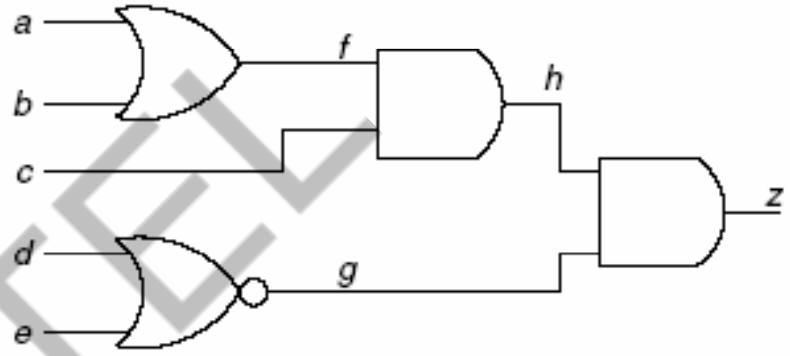
Multi-Line Conflict



- 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

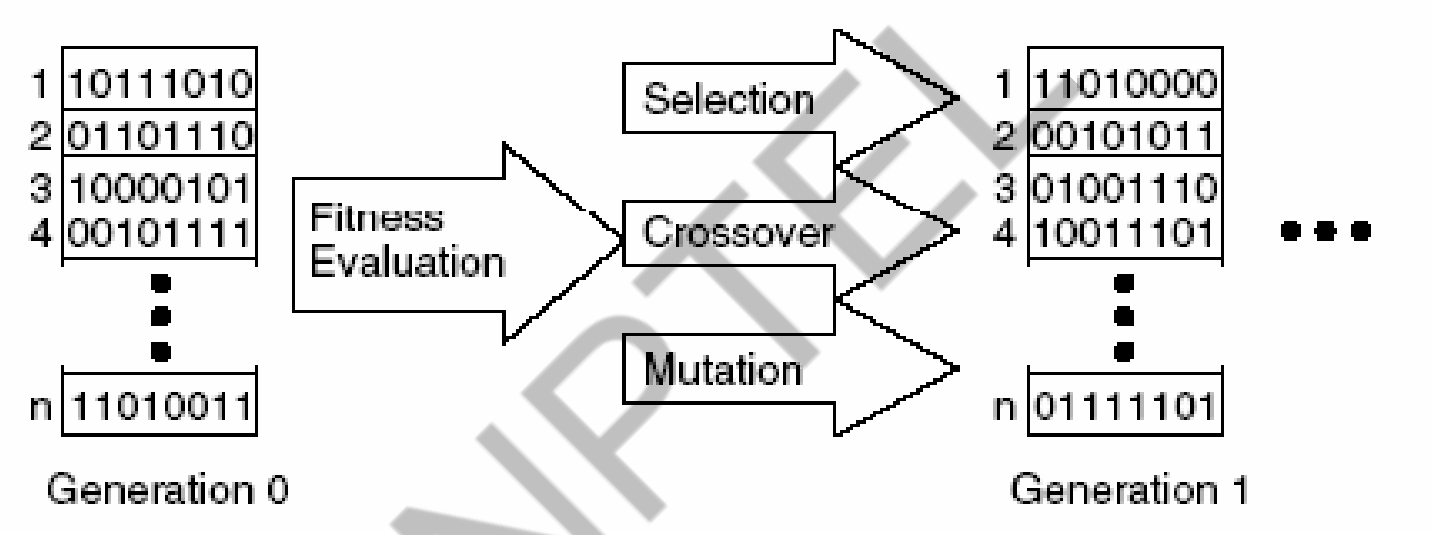
Simulation-Based ATPG

- 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



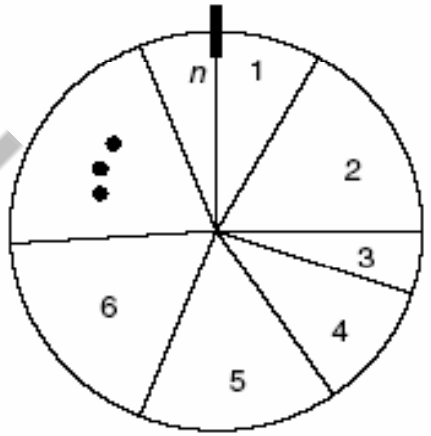
Pseudo Code for GA

Algorithm 13 Simple_GA_ATPG

```
1: test set T = Ø;
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 = Ø;
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	01001110010001011110101
Parent #1	11001100110011011001100
Parent #2	10101010101010101010101
Child #1	10001010100010001010100
Child #2	11101100111011101001110

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