Combinational Automatic Test Pattern Generation (ATPG)

Introduction

- **Basic problem:**
  - Input:
    - A combinational circuit
    - A fault list
  - Output:
    - A set of test vectors
    - List of undetected faults

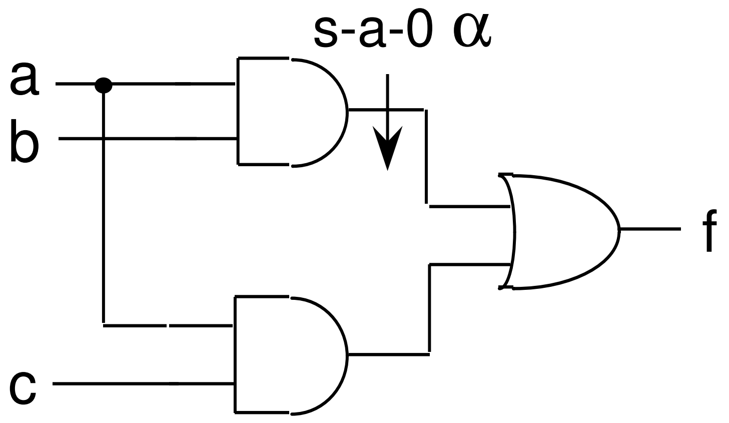
Broad Approaches

- From truth table
  - Impractical
- From Boolean equation
  - High complexity
- Using Boolean difference
  - Difficult to automate
- From circuit structure
  - D-Algorithm (Roth 1967), 9-V Algorithm (Cha 1978), PODEM (Goel 1981), FAN (Fujiwara 1983)

Test Generation Methods

(From Truth Table)

Ex: How to generate tests for the stuck-at 0 fault (fault α)?



	abc	f	f_α
	000	0	0
	001	0	0
	010	0	0
	011	0	0
	100	0	0
	101	1	1
√	110	1	0
	111	1	1

Impractical !!

Test Generation Methods

(Using Boolean Equation)

Since $f = ab+ac$, $f_\alpha = ac$

T_α = the set of all tests for fault α

$= \text{ON\_set}(f) * \text{OFF\_set}(f_\alpha) + \text{OFF\_set}(f) * \text{ON\_set}(f_\alpha)$

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

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

$= \{ (110) \}.$

High complexity !!

Since it needs to compute the faulty function for each fault.

* *ON_set(f): All input combinations that make f have value 1.*
OFF_set(f): All input combinations that make f have value 0.

Boolean Difference

- Algebraic method for determining the complete set of tests that detect a given fault.
- Definition:
  - The Boolean difference of a function $f(x_1,x_2,\ldots,x_n)$ with respect to one of its variables x_i is defined as

$$\frac{df(X)}{dx_i} = f(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n) \oplus f(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n)$$

$$= f_i(0) \oplus f_i(1)$$

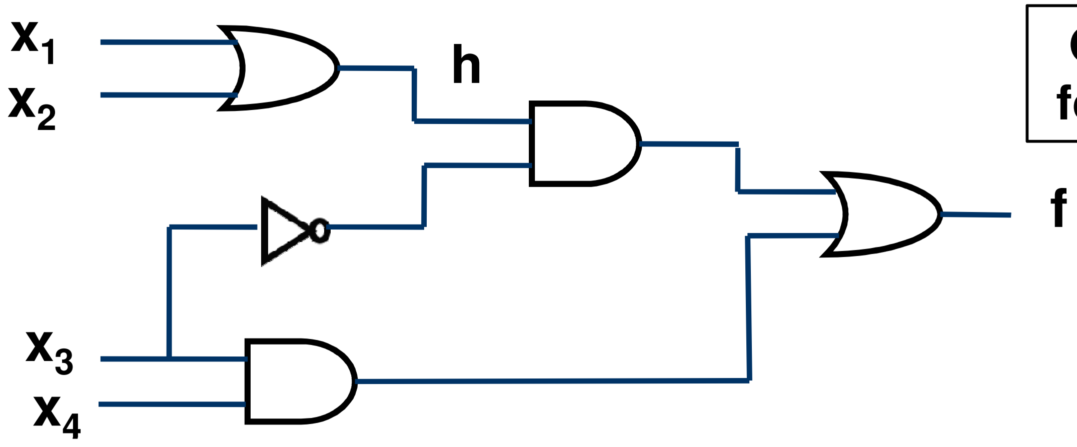
- **Some results:**
  - The set of tests which detect the fault $x_i/0$ is given by the equation

$$x_i \frac{df(X)}{dx_i} = 1$$

  - The set of tests which detect the fault $x_i/1$ is given by the equation

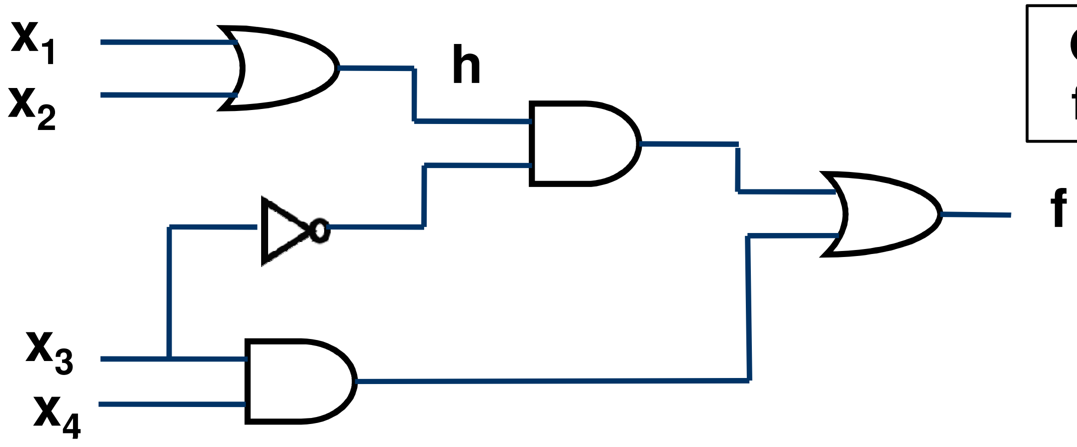
$$x_i' \frac{df(X)}{dx_i} = 1$$

Example



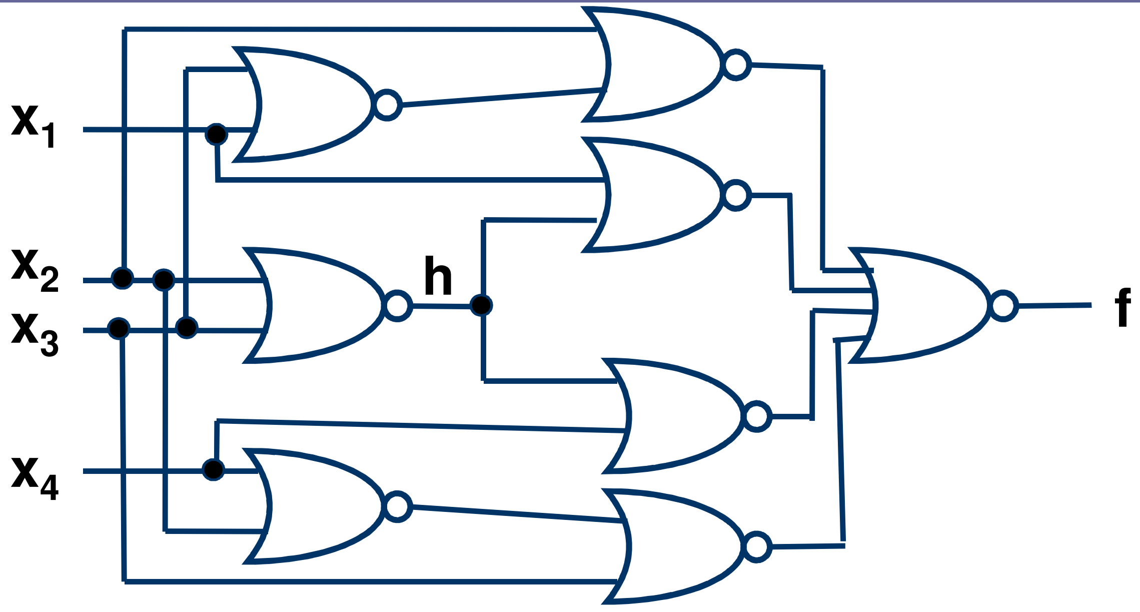
Generate tests for $x_3/0$ and $x_3/1$

Example



Generate tests for h/0 and h/1

Example



Test Generation by Path Sensitizing

- Test generation done from circuit structure.
- We first talk about 1-D path sensitization.
- Basic Principle:
  - Step 1: At the site of the fault, assign a logic value complementary to the fault being tested.
  - Step 2: Select a path from the site of the fault to a circuit output. Sensitize the path by assigning inputs to the gates along the path so as to propagate the effect of fault.

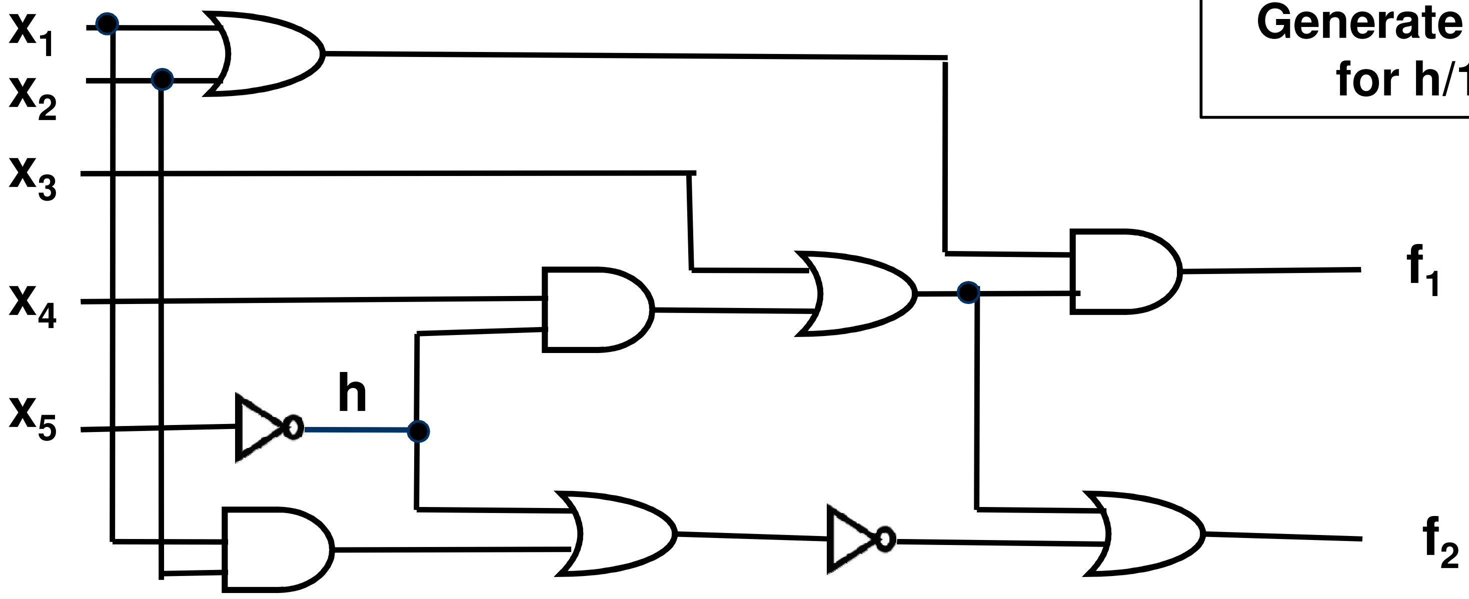
(FORWARD DRIVE PHASE)

– Step 3: Determine the primary inputs that will produce all the necessary signal values specified in Step 2. This requires tracing the signals backwards from each of the gates along the path to the primary inputs.

(BACKWARD TRACE PHASE)

Example

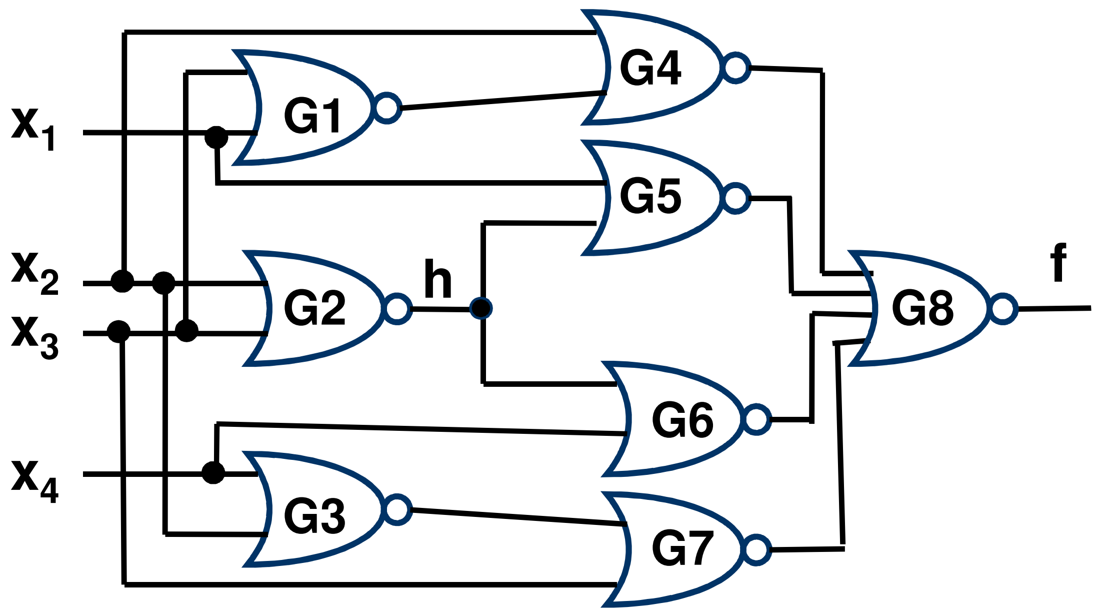
Generate test for h/1



- <u>Limitations of the Method</u>:
  - Suppose for the same fault we decide to observe the circuit response at f_2.
    - The input vector X=(1,0,0,1,1) sensitizes two paths emanating from h and terminating in f_2.
    - But the fault effect does not propagate to f_2.
  - This occurs when there exists reconvergent fanouts with unequal inversion parities.
    - To test such fault, a vector must be found which sensitizes only one of the two paths.
    - In this example, X=(0,0,0,0,1).

- <u>**Major Advantage of the Method**</u>**:**
  - **In many cases a test for a PI (primary input) is also a test for all the lines along the sensitized path.**
    - **Fanouts and reconvergence complicates the problem.**

Another Example



Generate test for h/0

- **<u>Analysis</u>:**
  - **If we were allowed to sensitize both paths through G5 and G6 simultaneously, a test can be generated.**
  - **We, therefore, need 2-D path sensitization.**

Test Generation using Path Sensitization

- **Basically a two-step process:**
  1. **<u>Activate the fault</u>: Set PI values that causes line ‘h’ to have value v′ (for h/v fault).**
  2. **<u>Propagate the fault</u>: Through forward drive and backward trace phases, set PI values that propagates the fault effest to one of the primary outputs.**

Composite Value System

- To keep track of error propagation, we must consider values in both the fault-free circuit N and the faulty circuit N_f.
  - For this we define composite logic values of the form v/v_f, where v and v_f are values of the same signal line in N and N_f respectively.
  - 1/0 is denoted by the symbol D

    0/1 is denoted by the symbol D′

    0/0 is denoted by the symbol 0

    1/1 is denoted by the symbol 1

- Any logic operation between two composite values can be done separately by processing the fault-free and faulty values.
- For example,

  $D' + 0 = 0/1 + 0/0 = (0+0) / (1+0) = 0/1 = D'$

- We also use a fifth value ‘X’ to denote an unspecified composite value; that is, any value in the set {0, 1, D, D′}.

	0	1	D	D′	X
0	0	0	0	0	0
1	0	1	D	D′	X
D	0	D	D	0	X
D′	0	D′	0	D′	X
X	0	X	X	X	X

AND

	0	1	D	D′	X
0	0	1	D	D′	X
1	1	1	1	1	1
D	D	1	D	1	X
D′	D′	1	1	D′	X
X	X	1	X	X	X

OR

	0	1	D	D′	X
0	0	1	D	D′	X
1	1	0	D′	D	X
D	D	D′	0	1	X
D′	D′	D	1	0	X
X	X	X	X	X	X

EXOR

- It is easy to verify that D behaves consistently with the rules of Boolean algebra:

  $D + D' = 1$

  $D \cdot D' = 0$

  $D + D = D \cdot D = D$

  $D' + D' = D' \cdot D' = D'$

The Basic Algorithm

- **Structure of the algorithm to generate a test for line ‘h’ s-a-v:**

```
begin
   set all values to ‘X’;
   Justify (h, v′);
   if  (v = 0)
       then   Propagate (h, D);
       else   Propagate (h, D′);
end
```

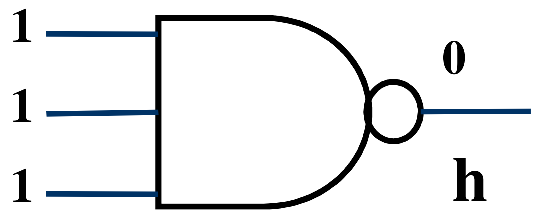
Some Definitions

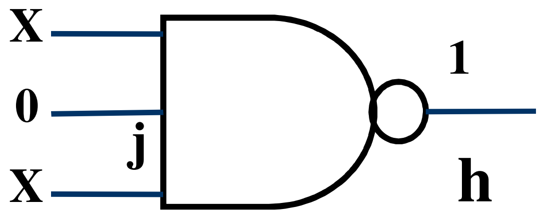
- **Basic gates like AND, OR, NAND and NOR can be characterized by two parameters:**
  1. **<u>Controlling value</u>: The value of an input is said to be controlling if it determines the value of the gate output regardless of the values of the other inputs.**
  2. **<u>Inversion</u>: If 'c' is the controlling value of an input to a gate with inversion 'i', the value of the gate output will be c⊕i .**

	c	i
AND	0	0
OR	1	0
NAND	0	1
NOR	1	1

Line Justification Algorithm

```
Justify  (h, val)
begin
    set line ‘h’ to ‘val’;
    if ‘h’ is a PI then return;
    c = controlling value of ‘h’;
    i = inversion of ‘h’;
    inval = val $\oplus$ i;
    if  (inval = c′)
        then  for every input ‘j’ of ‘h’
                        Justify (j, inval);
        else  begin
                        select any one input ‘j’ of ‘h’;
                        Justify (j, inval);
              end
end
```

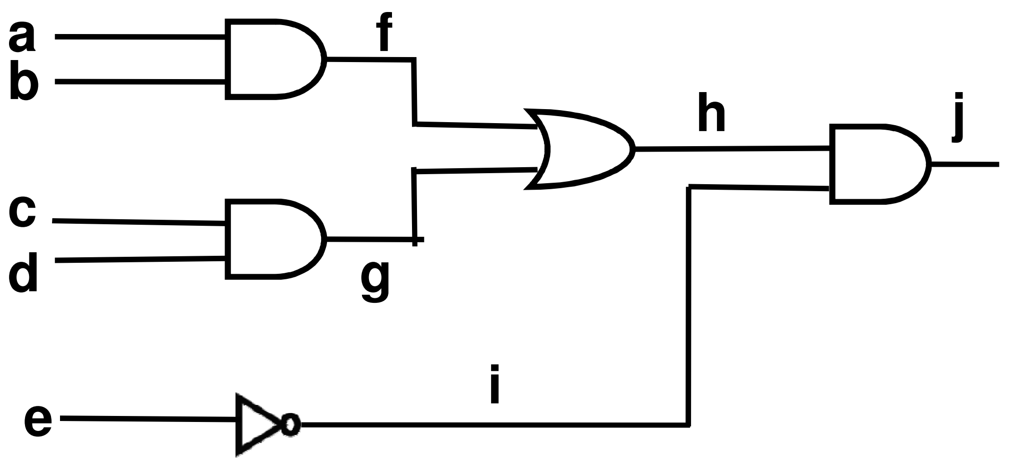




Error Propagation Algorithm

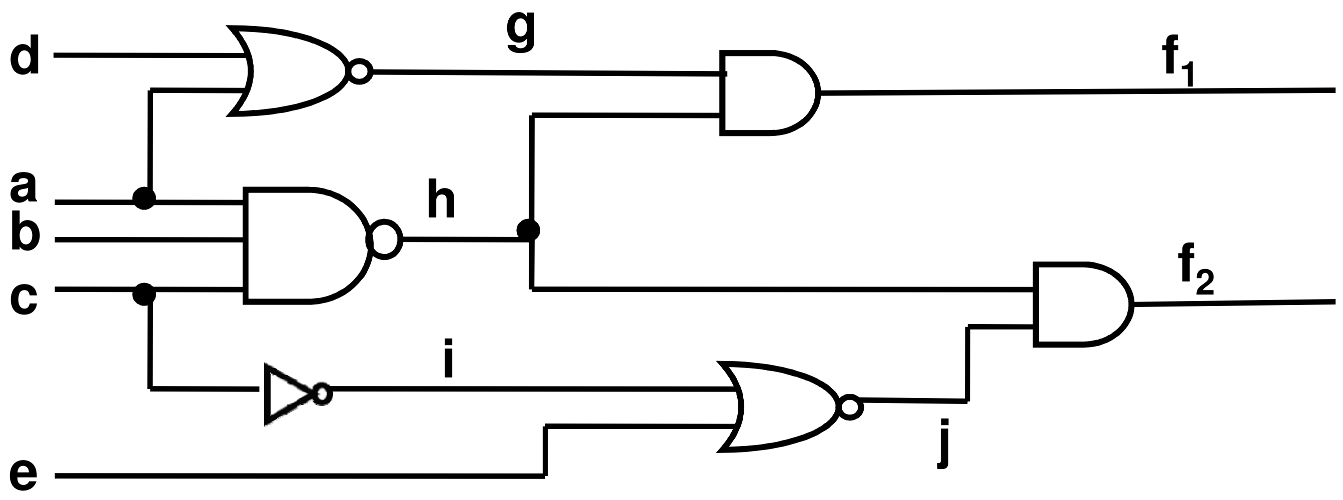
```
Propagate  (h, err)    /* err is D or D′  */
begin
    set line ‘h’ to ‘err’;
    if ‘h’ is PO then return;
    k = the gate driven by ‘h’;
    c = controlling value of ‘k’;
    i = inversion of ‘k’;
    for every input ‘j’ of ‘k’ other than ‘h’
            Justify  (j, c′);
    Propagate  (k, err ⊕ i);
end
```

Example 1



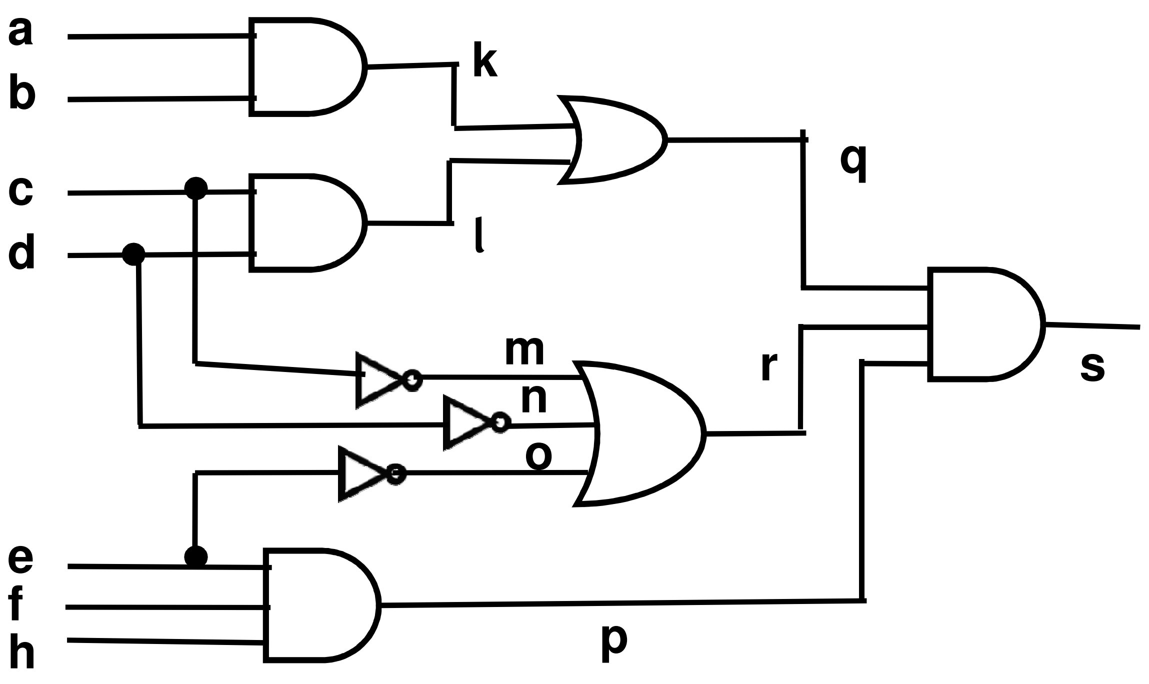
Generate test for f/0

Example 2



Generate test for h/1

Example 3



Generate test
for h/1

Some Comments

- It is seen that in general, search for a test may need backtracking.
- The algorithms as presented does not implement backtracking.
  - Must be incorporated in any real implementation.
  - This makes the algorithm more complex.
  - May even fail to generate test for some faults.

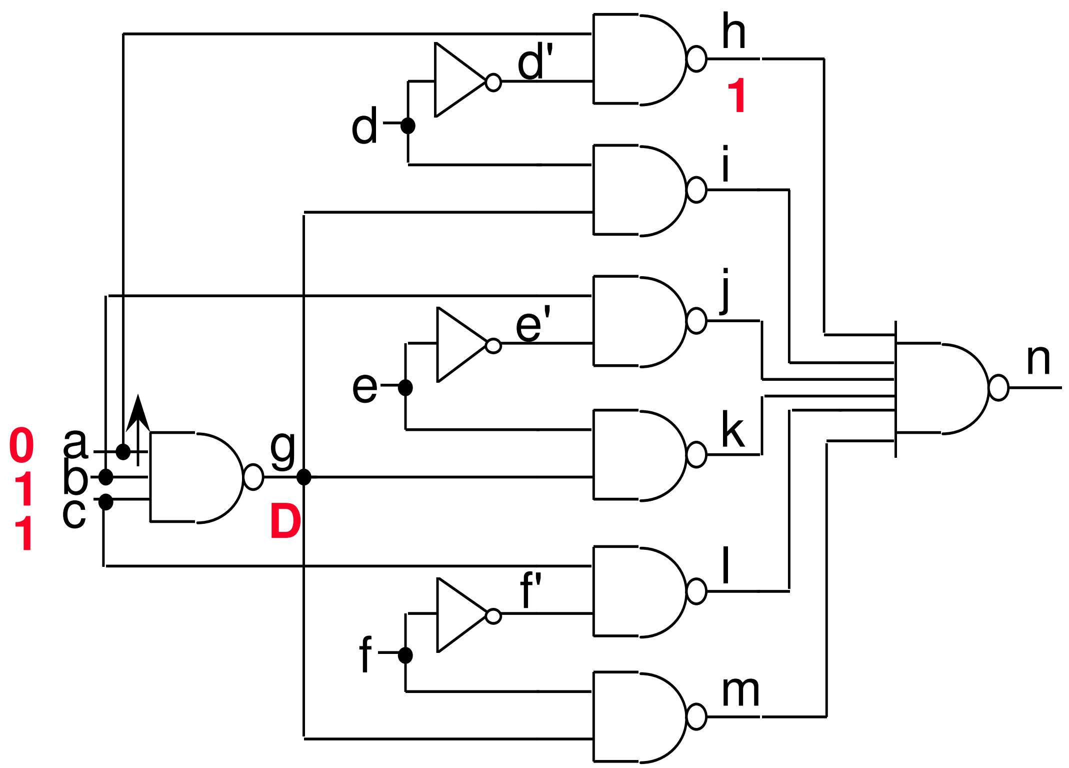
D Algorithm

- Concepts presented builds the foundation for D algorithm, and its derivatives.
- Definition:
  - D-Frontier: It consists of all gates whose output value is currently ‘X’ but have one or more error signals (D or D′) on their inputs).
- Error Propagation:
  - Select one gate from D-frontier and assign values to the unspecified gate inputs so that the gate output becomes D or D′.

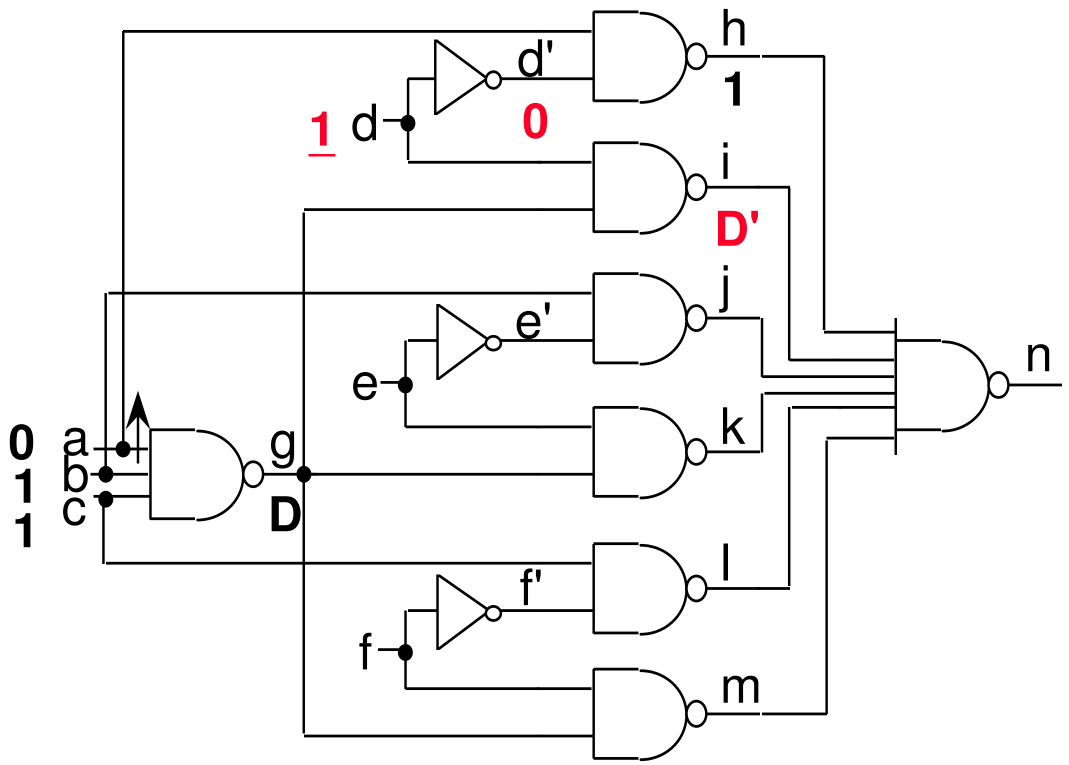
    (called D-DRIVE OPERATION)

D-Algorithm: Example (1/6)

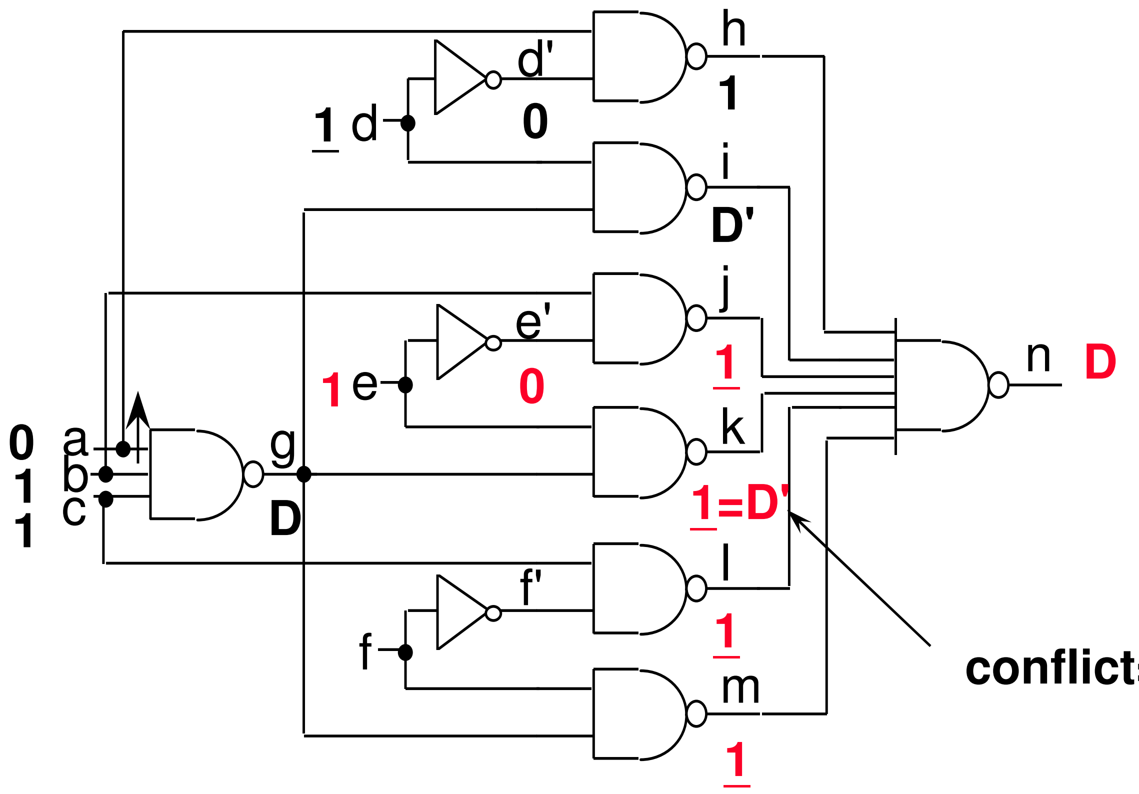
- **Logic values = {0, 1, D, D', x}.**



D-Algorithm: Example (2/6)

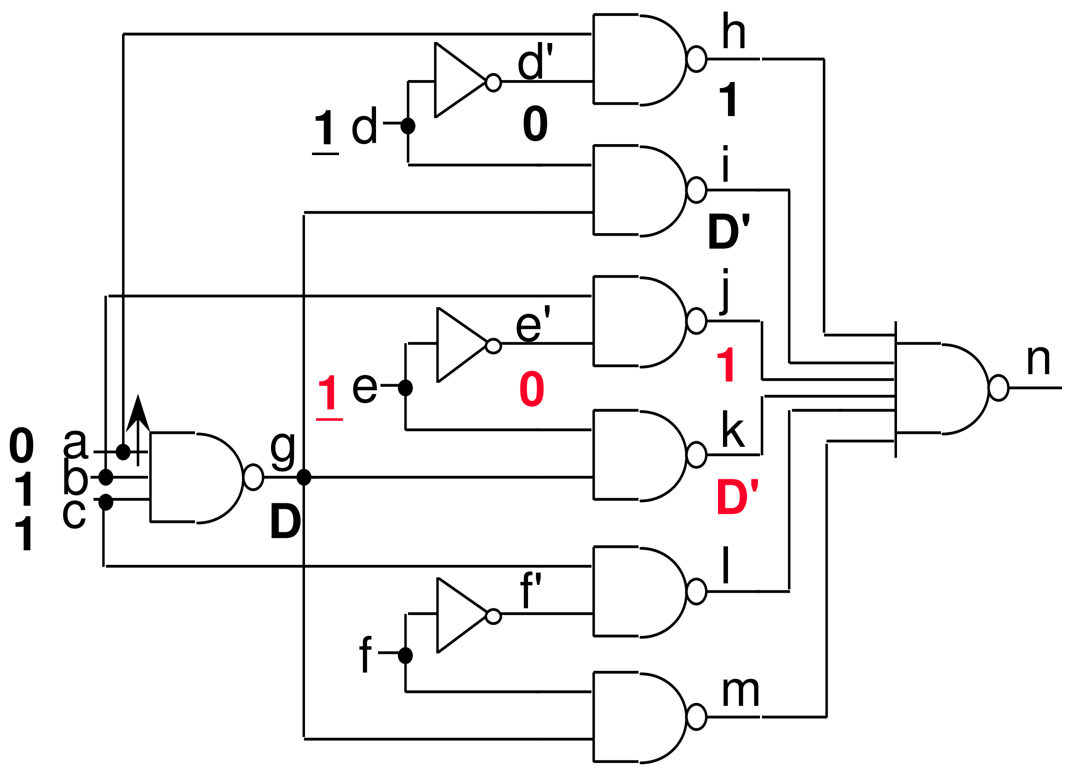


D-Algorithm: Example (3/6)

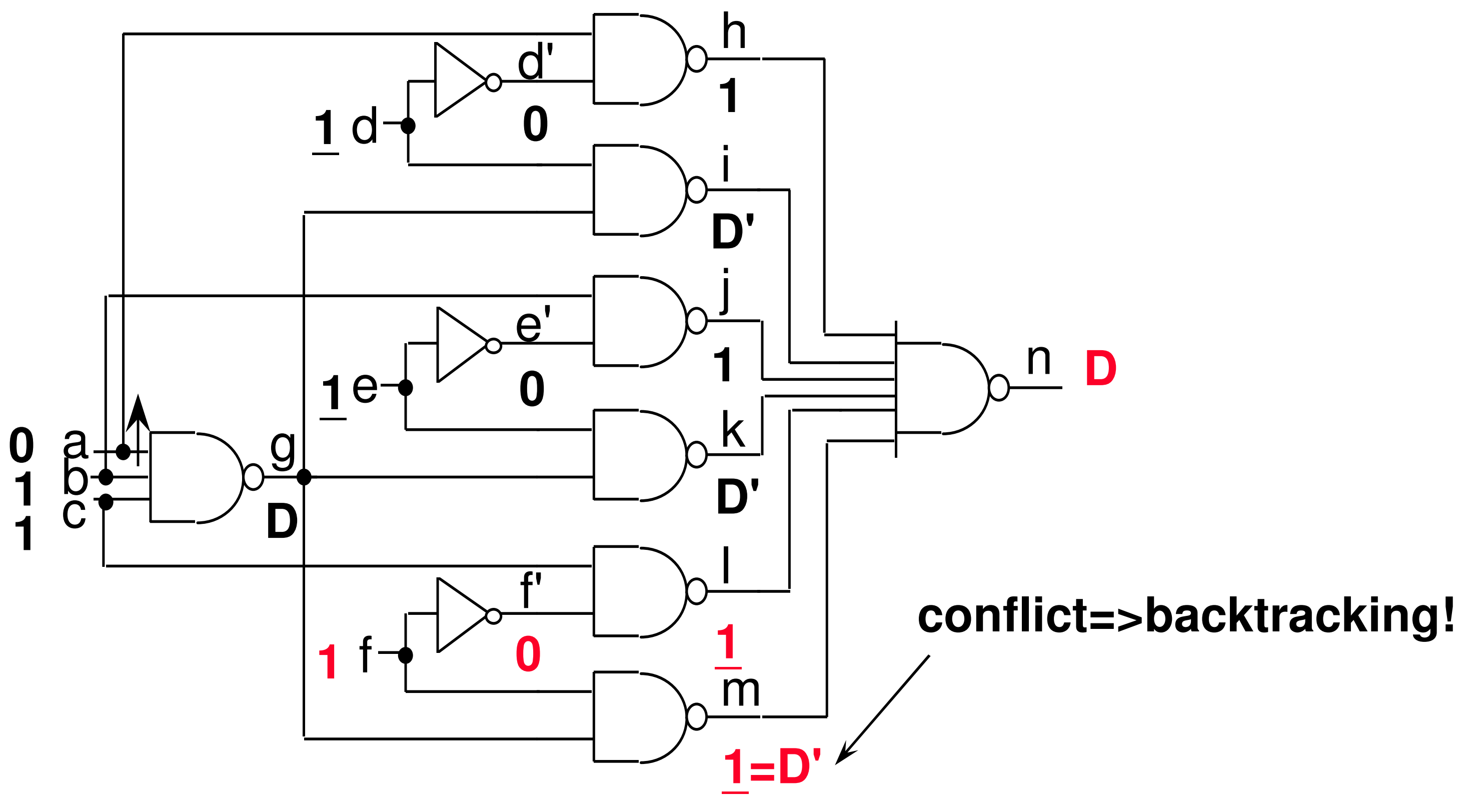


conflict=>backtracking!

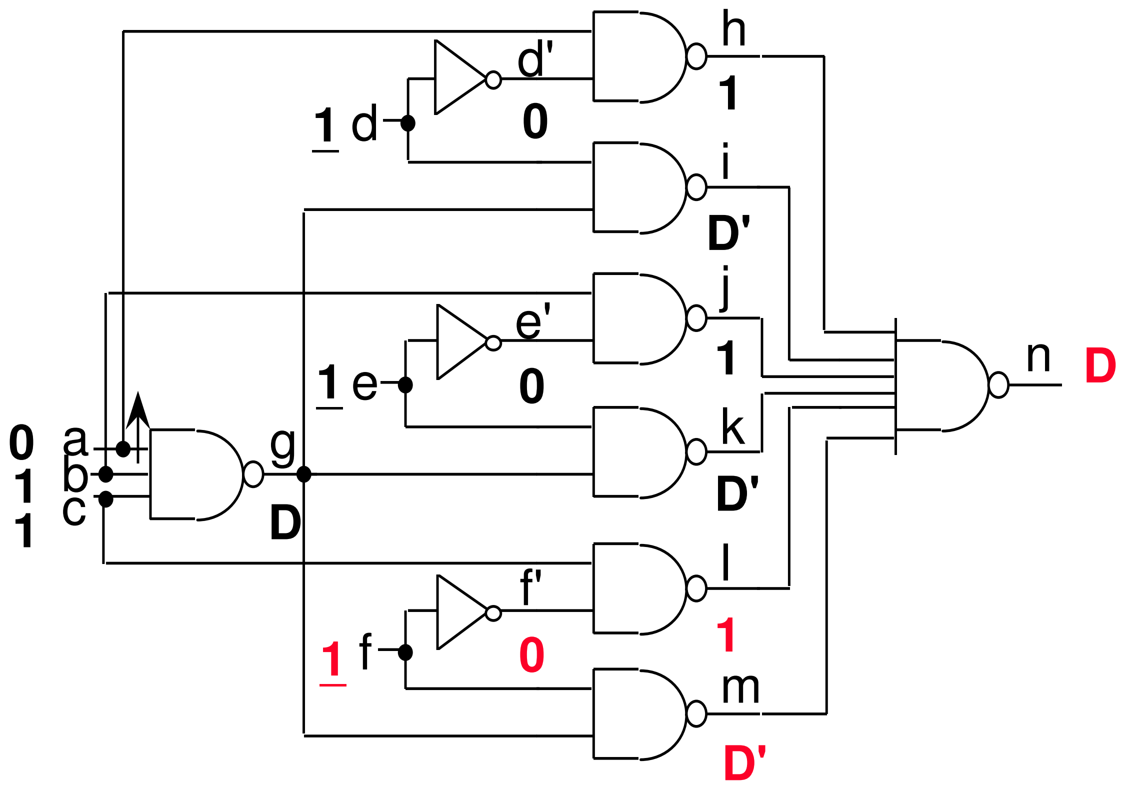
D-Algorithm: Example (4/6)



D-Algorithm: Example (5/6)



D-Algorithm: Example (6/6)



h
1
d'
1 d
0
i
D'
j
e'
1
1 e
0
k
n D
0 a
1 b
1 c
g
D
D'
l
f'
1
1 f
0
m
D'