CIS 4930 Digital System Testing
Testing for Single Stuck-at Faults (SSFs)

Dr Hao Zheng
Comp. Sci. & Eng.
U of South Florida

Testing Generation

Testing Generation (TG) is a complex problem

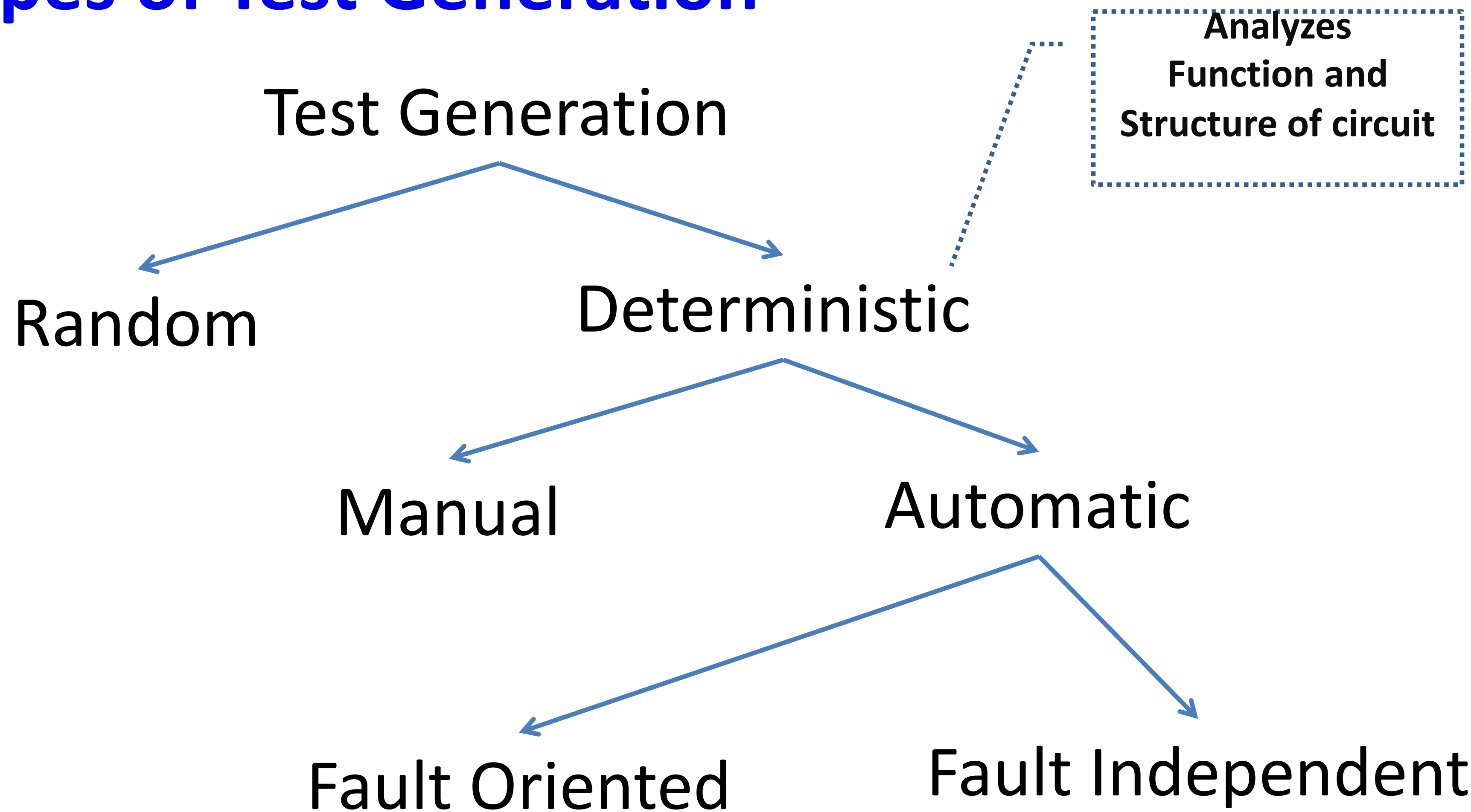
We are interested in:

→ The **cost of generating** the test

→ The **quality** (fault coverage) of the test

→ The **cost of applying** the test

Types of Test Generation



Deterministic TG System

- Model is analyzed to generate test & expected responses
- Diagnostic data can be saved for fault location

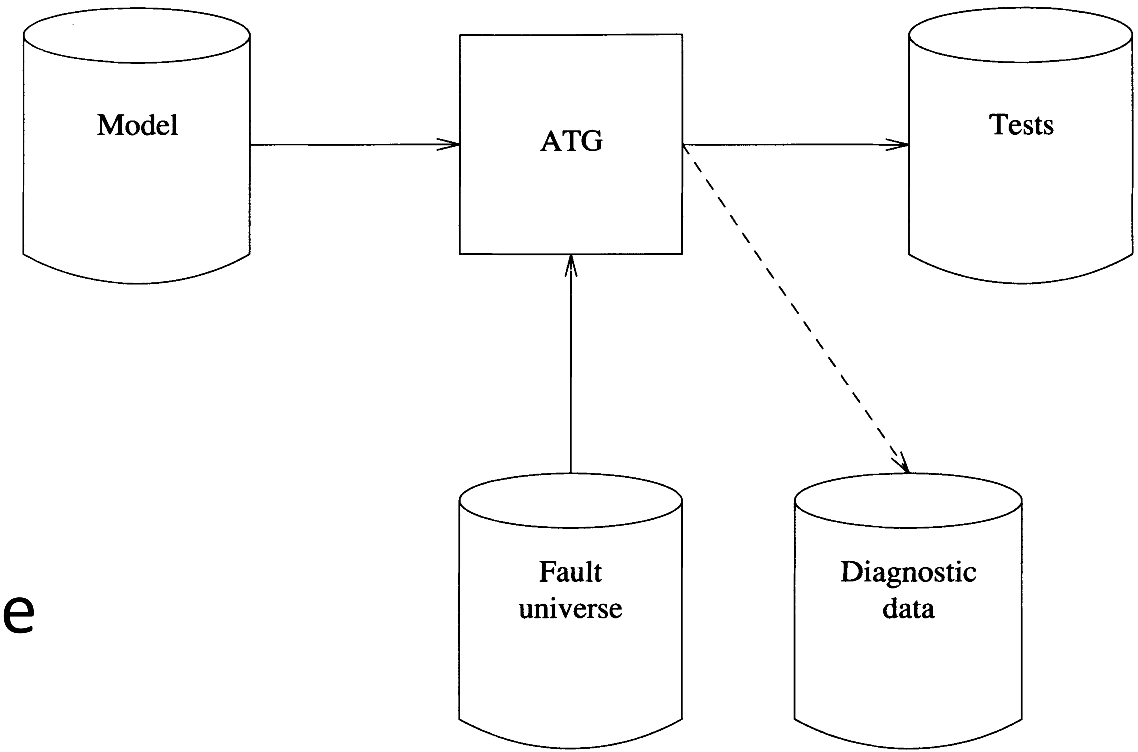


Figure 6.1 Deterministic test generation system

6.2.1 Fault-oriented ATG

- Circuit model – gate-level combinational circuit
- Basic Algorithm – Fanout Free
- Backtracking Algorithm
- *D* Algorithm
- PODEM (Path Oriented Decision Making)
- FAN extends PODEM

Line Justification

- To detect a fault
  - **Activate** the fault
  - **Propagate** the fault to a PO

Activating a fault a l *s-a-v*:

- Determine PI values that force value on line l to $\bar{v}$

This is known as the **line-justification** problem

Composite Logic Values

Let D represent 1/0 and $\overline{D}$ represent 0/1

v/v$_f$	
0/0	0
1/1	1
1/0	$\boldsymbol{D}$
0/1	$\overline{\boldsymbol{D}}$

AND	**0**	**1**	**D**	$\overline{\boldsymbol{D}}$	**X**
0	0	0	0	0	0
1	0	1	D	$\overline{\boldsymbol{D}}$	X
D	0	D	D	0	X
$\overline{\boldsymbol{D}}$	0	D’	0	$\overline{\boldsymbol{D}}$	X
X	0	X	X	X	X

OR	**0**	**1**	**D**	$\overline{\boldsymbol{D}}$	**X**
0	0	1	D	D’	0
1	1	1	1	1	1
D	D	1	D	1	X
$\overline{\boldsymbol{D}}$	$\overline{\boldsymbol{D}}$	1	1	$\overline{\boldsymbol{D}}$	X
X	X	1	X	X	X

Fig 6.3 TG for *l s-a-v* in Fanout Free circuit

begin

set all values to x // initialization of all wires to X

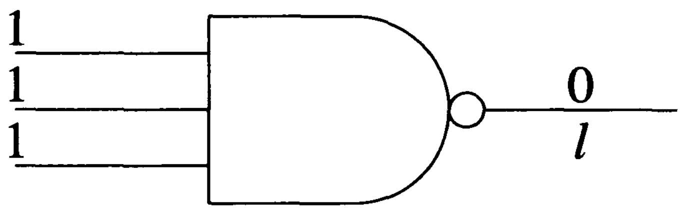
Justify$(l, \overline{v})$ // justification of line *l*

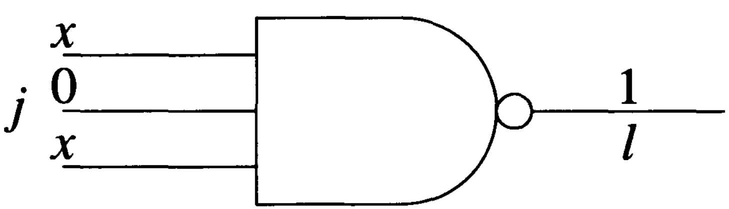
if $v = 0$ **then** *Propagate* (l, D)

else *Propagate* $(l, \overline{D})$

end

Line Justification

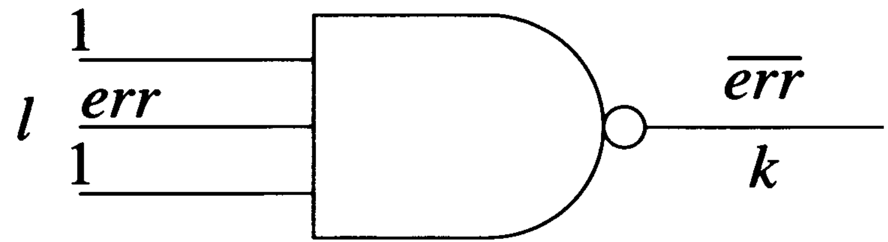




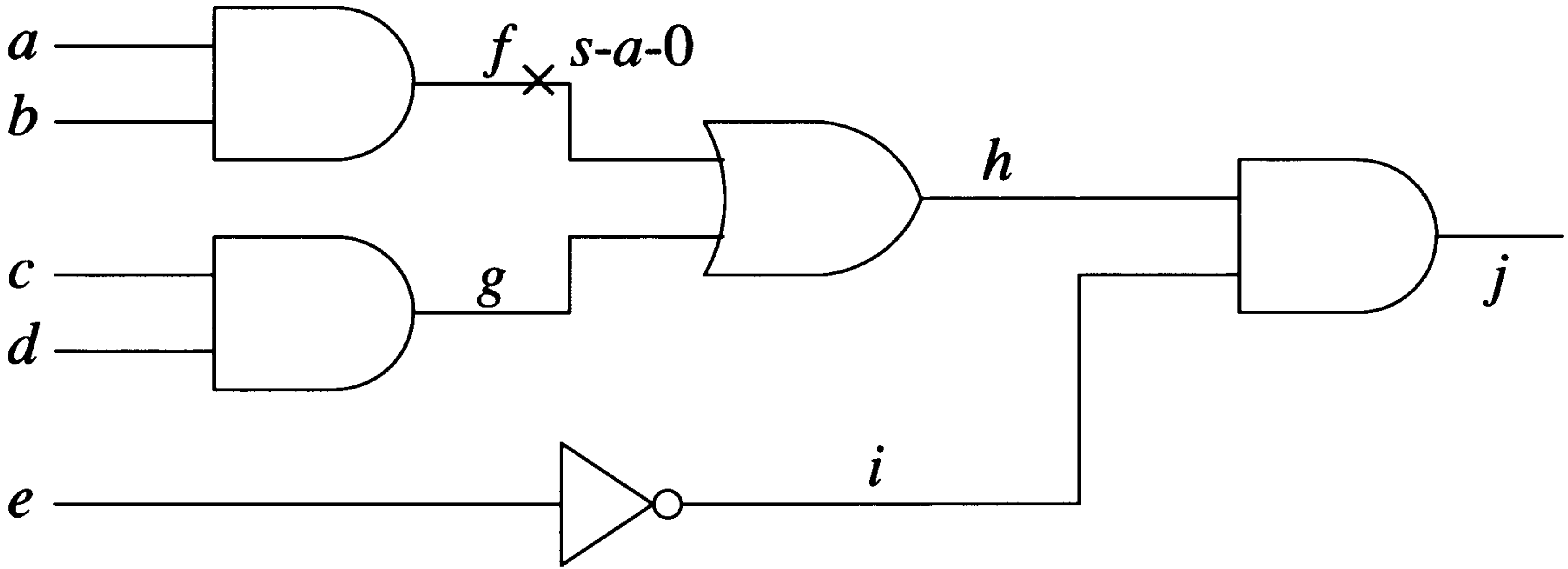

```
Justify (l, val)
begin
    set l to val
    if l is a PI then return
    /* l is a gate (output) */
    c = controlling value of l
    i = inversion of l
    inval = val⊕i
    if (inval = c̄)
        then for every input j of l
            Justify (j, inval)
        else
            begin
                select one input (j) of l
                Justify (j, inval)
            end
end
```

Error Propagation – Fanout Free circuit

```
Propagate (l, err)
/* err is D or D̄ */
begin
    set l to err
    if l is PO then return
    k = the fanout of l
    c = controlling value of k
    i = inversion of k
    for every input j of k other than l
        Justify (j, c̄)
    Propagate (k, err⊕i)
end
```



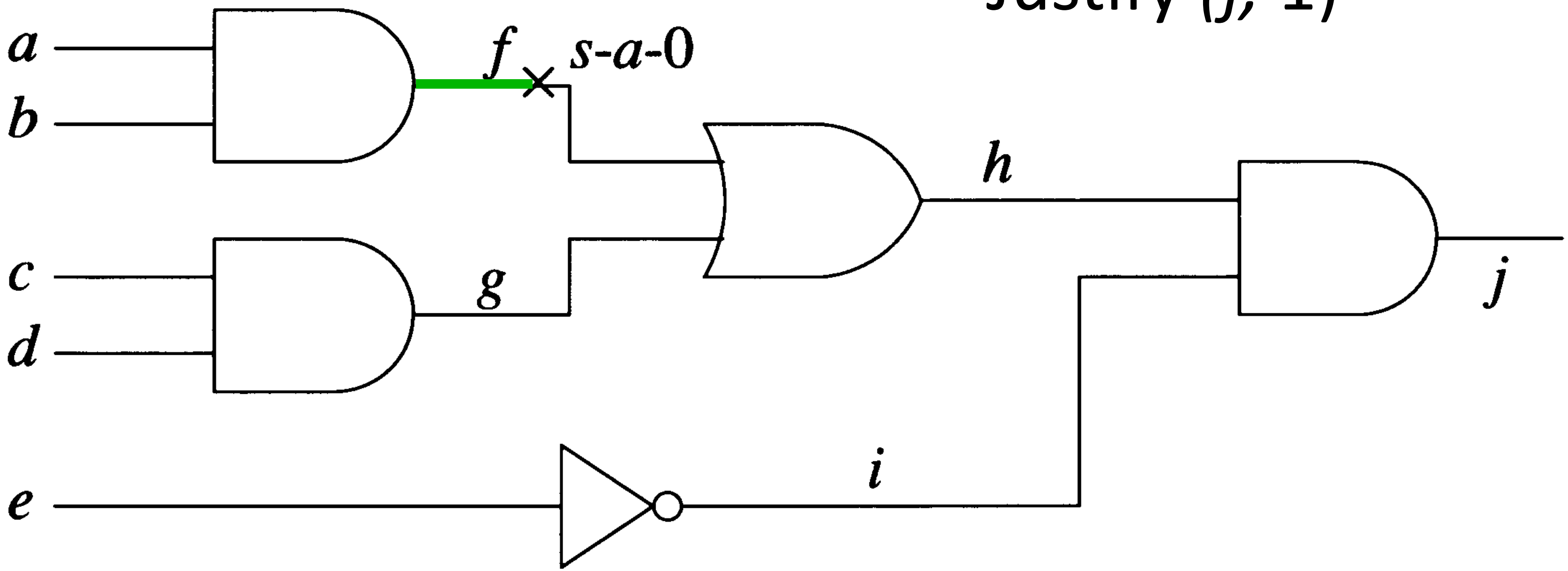
Example 6.1



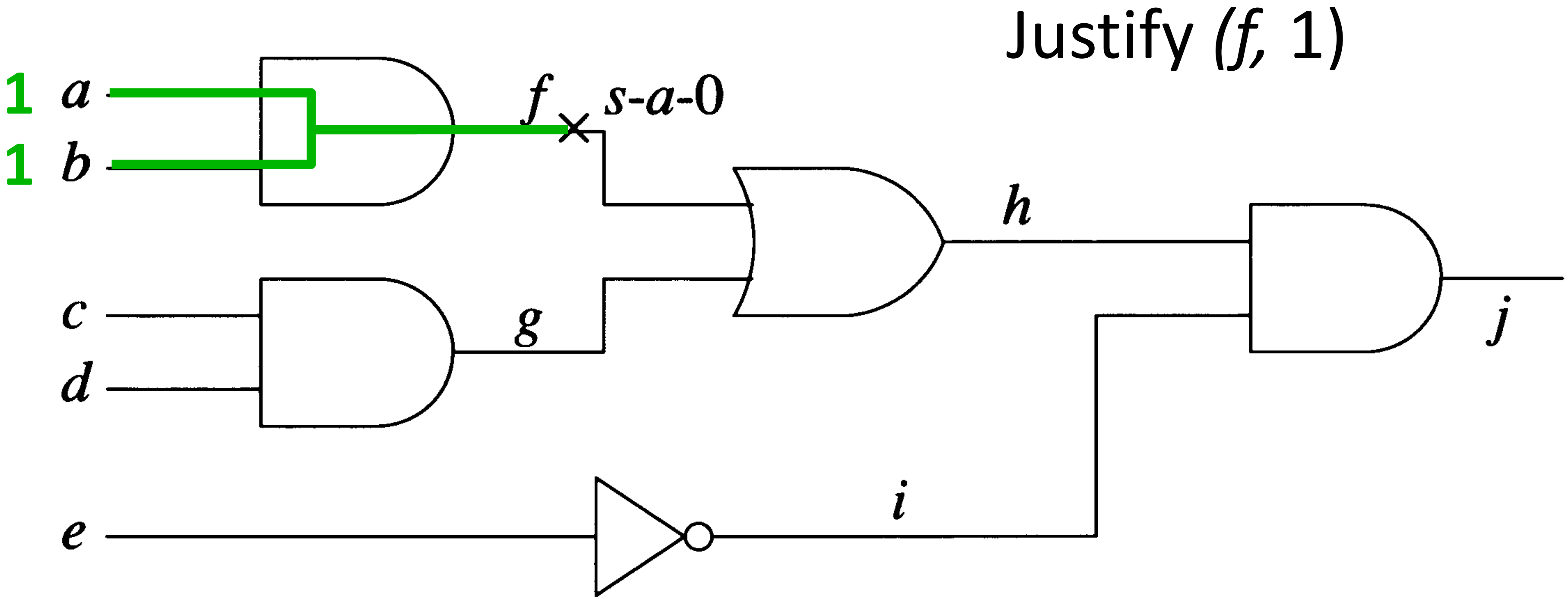
Find an input vector such that *f s-a-0* is observable on *j*

Example 6.1

Justify (*f*, 1)



Example 6.1

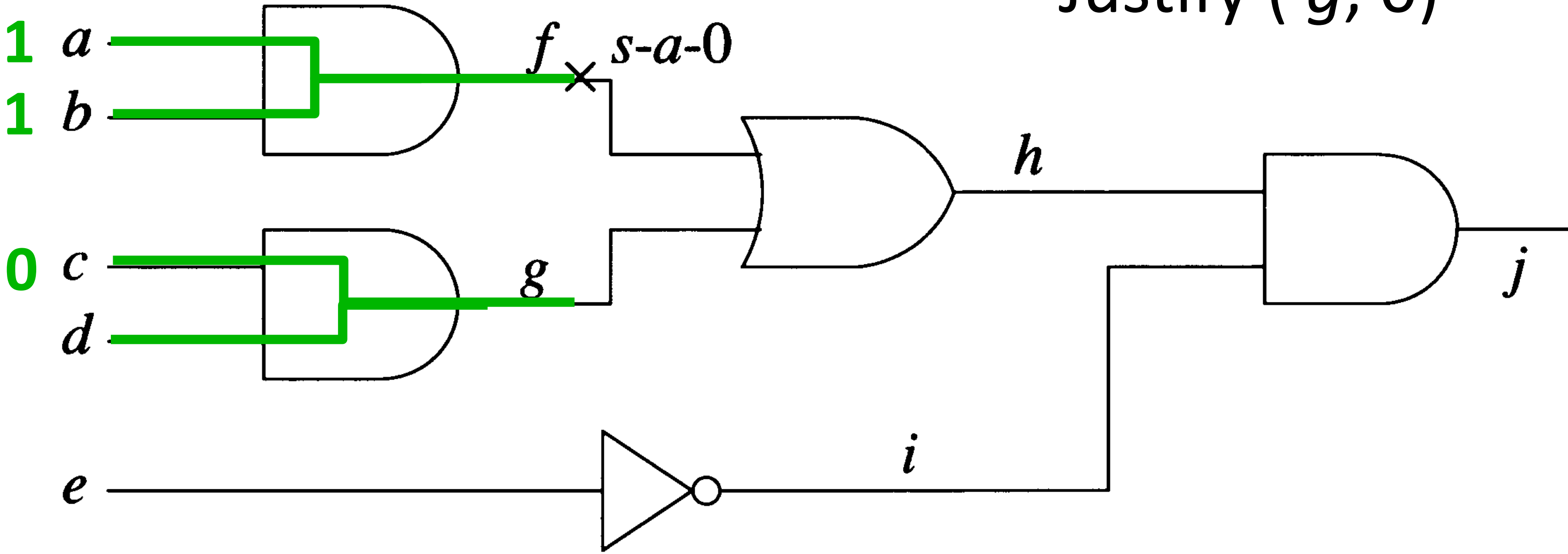


Example 6.1

Propagate (*f*, D)

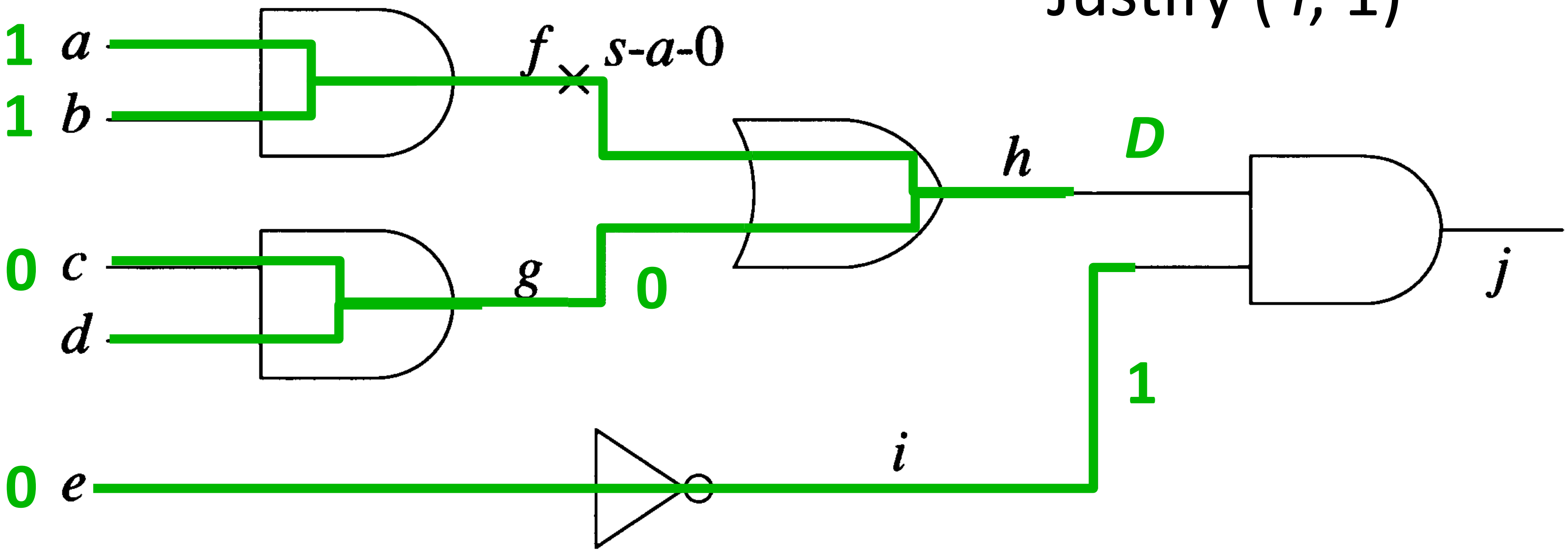
↓

Justify (*g*, 0)



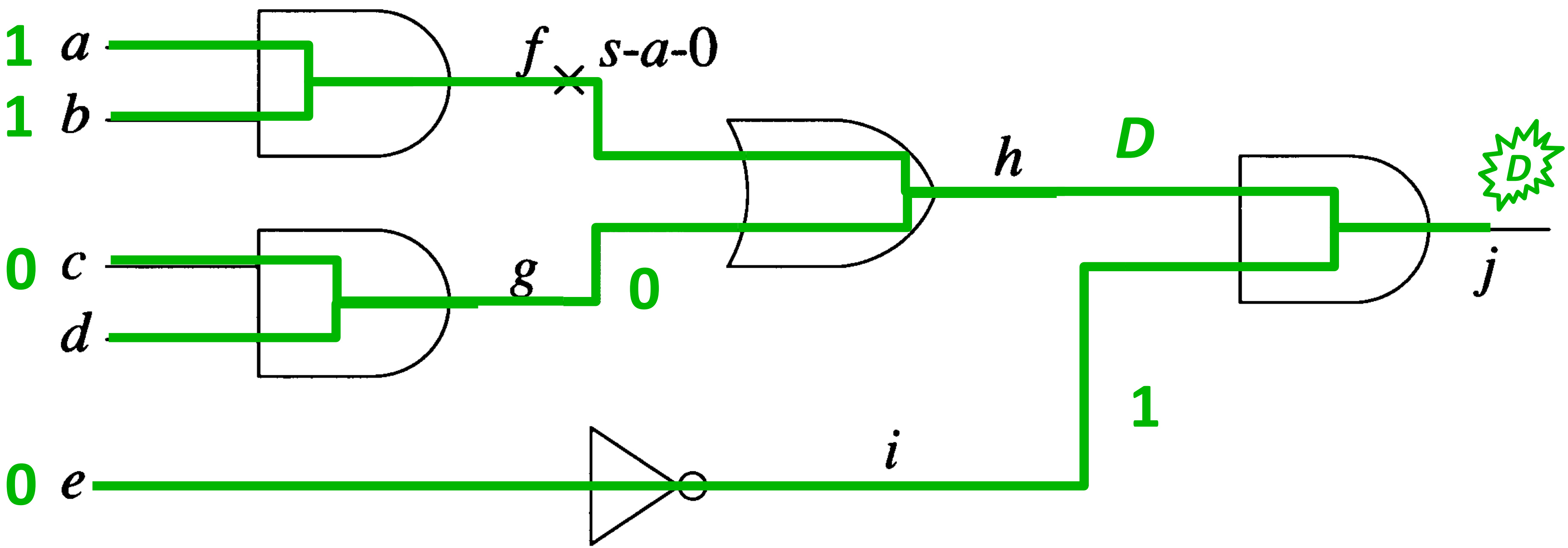
Example 6.1

Propagate (*h*, D)
↓
Justify (*i*, 1)

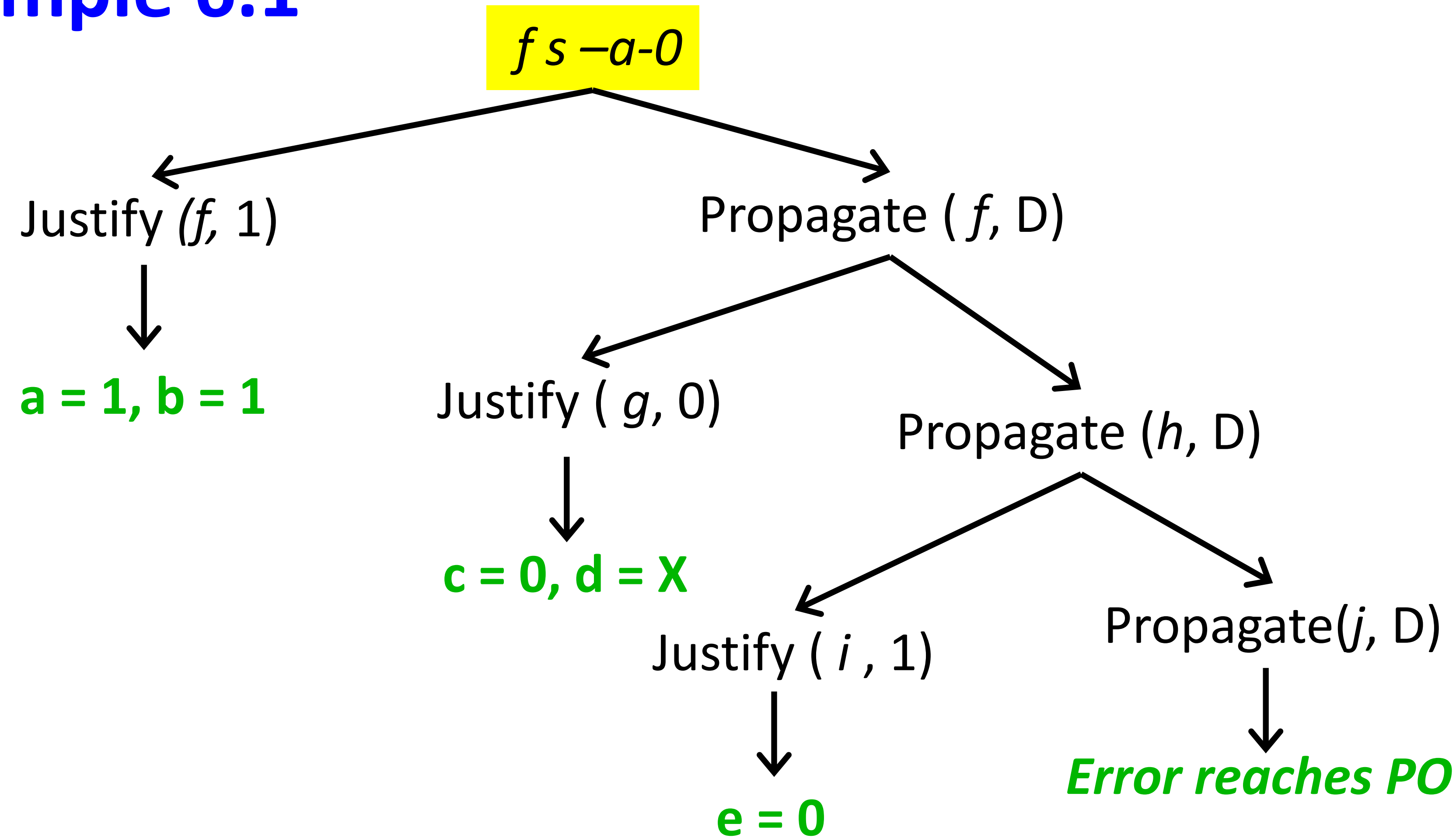


Example 6.1

Propagate (*j*, D)



Example 6.1



f s –*a*-0
Justify (*f*, 1)
a = 1, b = 1
Propagate (*f*, D)
Justify (*g*, 0)
c = 0, d = X
Propagate (*h*, D)
Justify (*i* , 1)
e = 0
Propagate(*j*, D)
Error reaches PO!

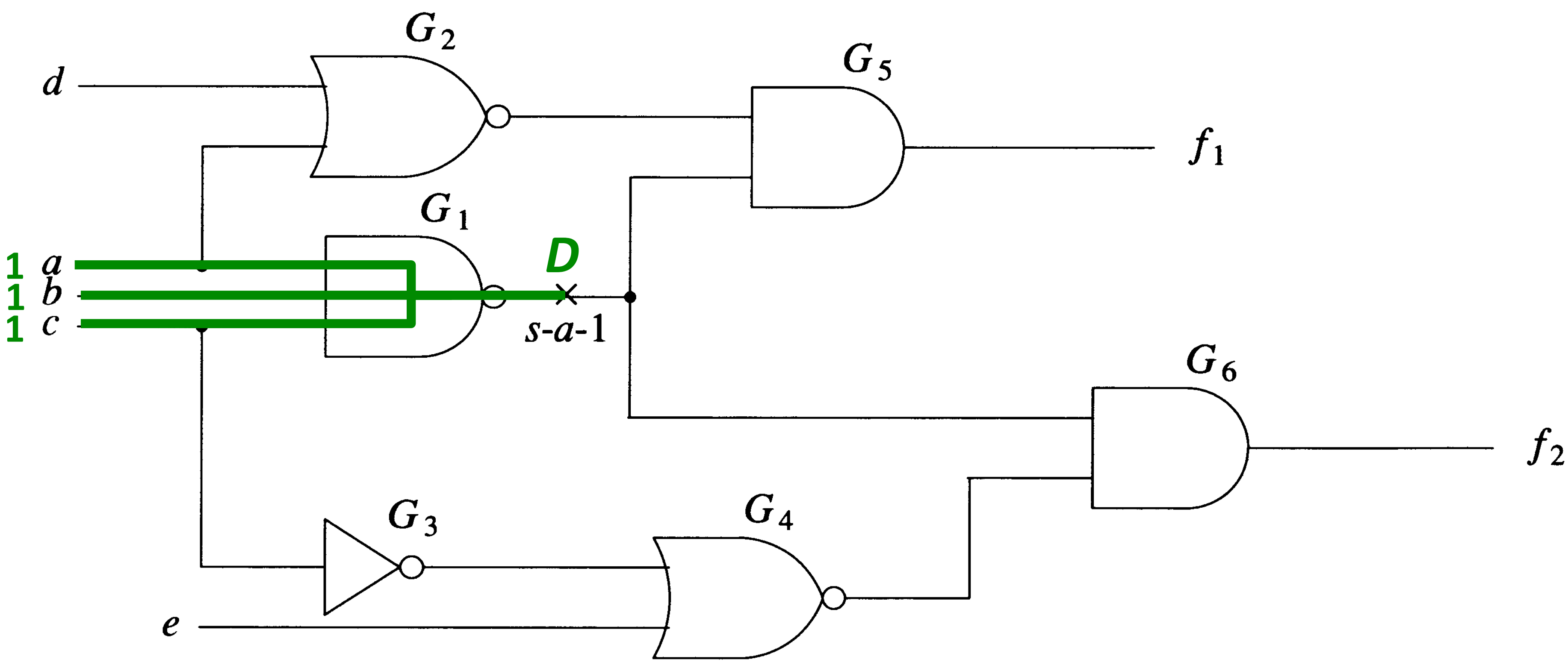
Fanout Free vs. Fanout

- For Fanout Free circuit
  - Line justification problems are independent
  - Sets of PI's assigned to justify required values are mutually disjoint

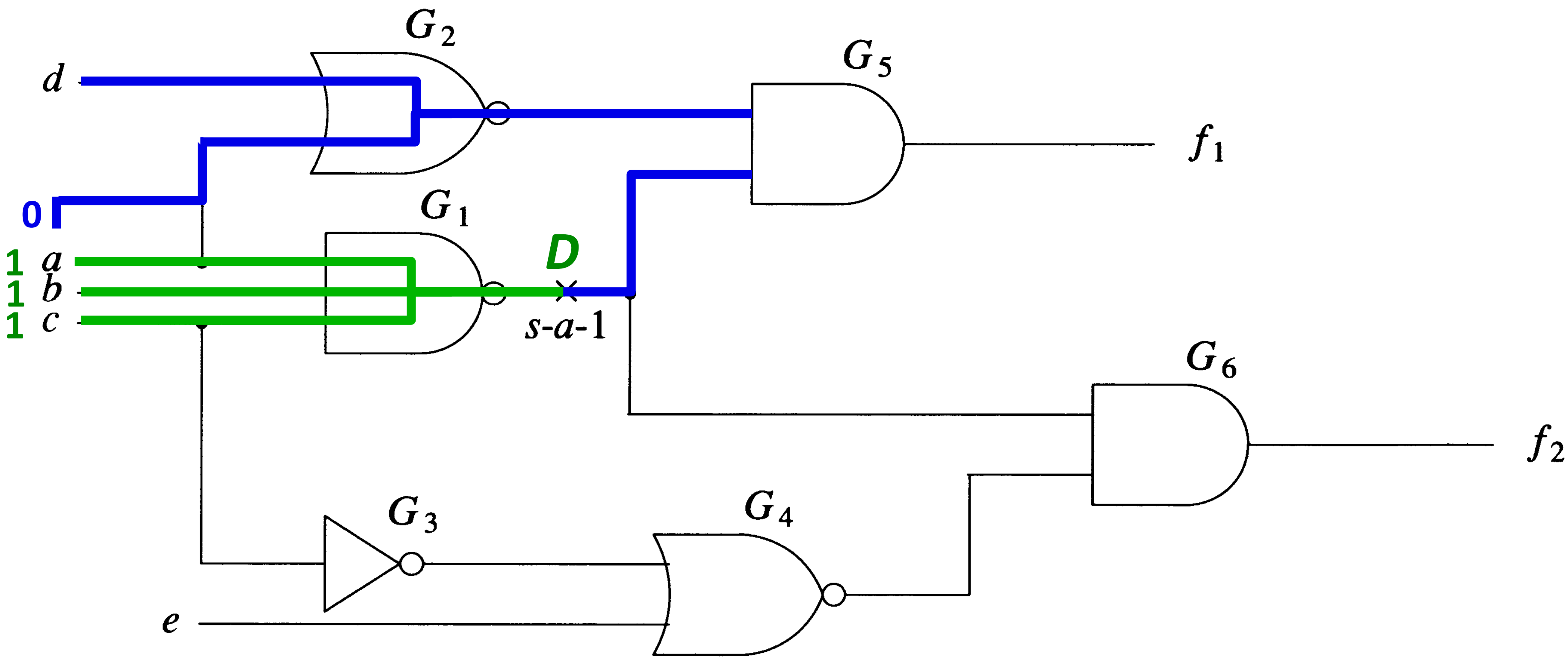
- Circuits with Fanout
  - Several ways to propagate error to PO
  - Fundamental difficulty: see following examples

    resulting line justification problems are no longer independent

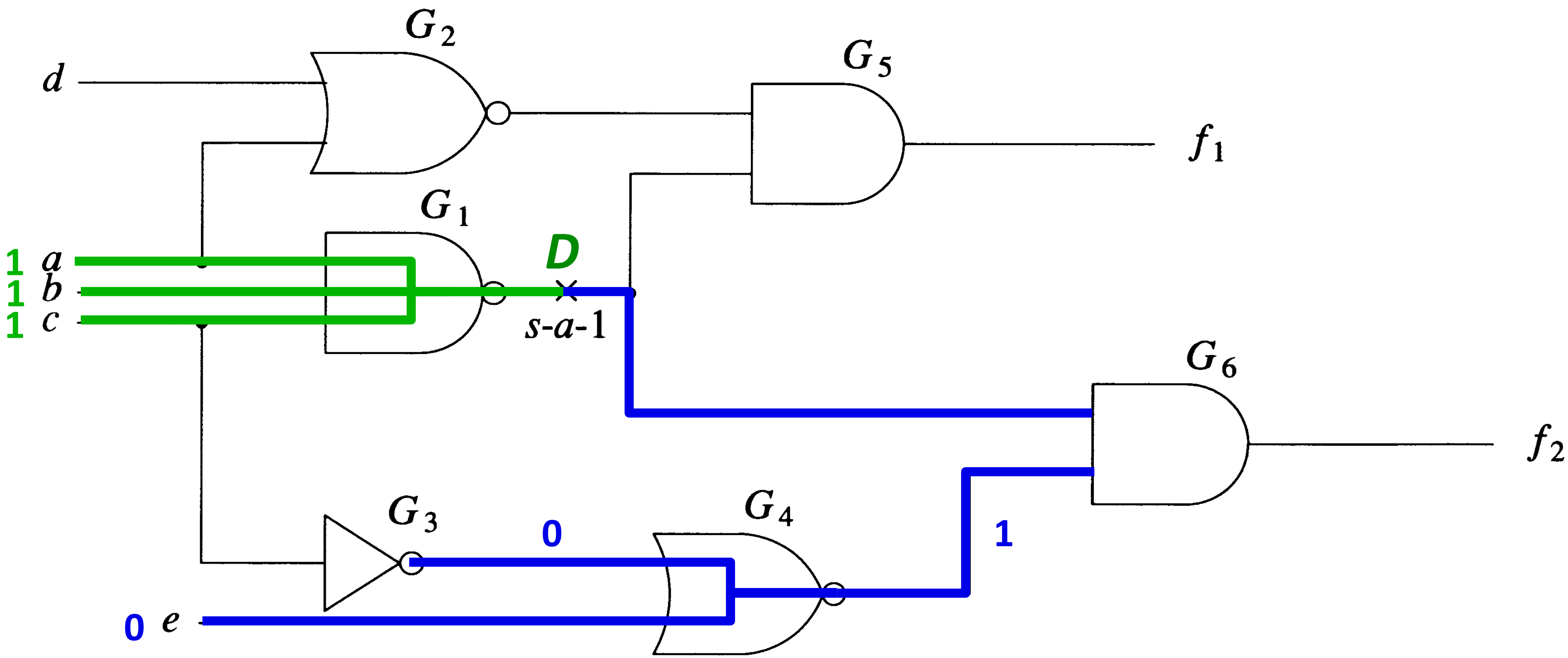
Example 6.2



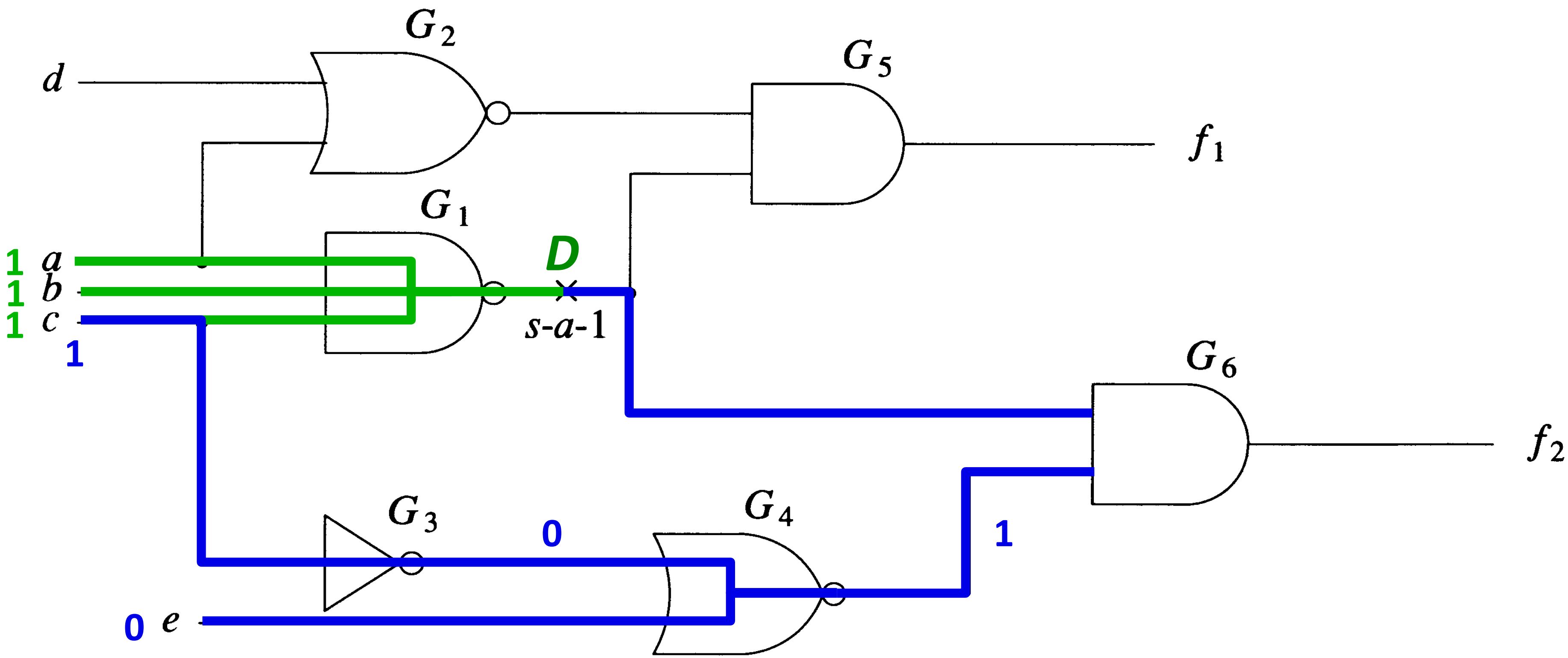
Example 6.2



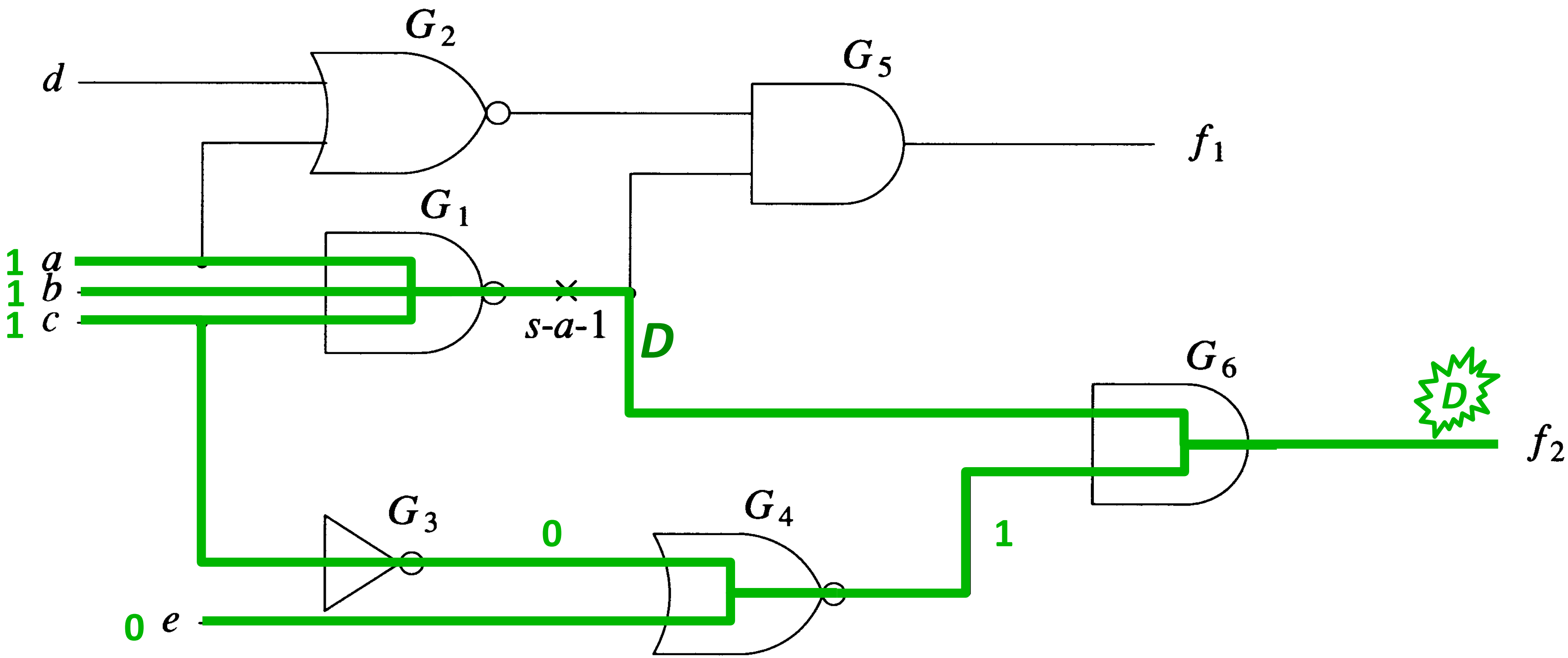
Example 6.2



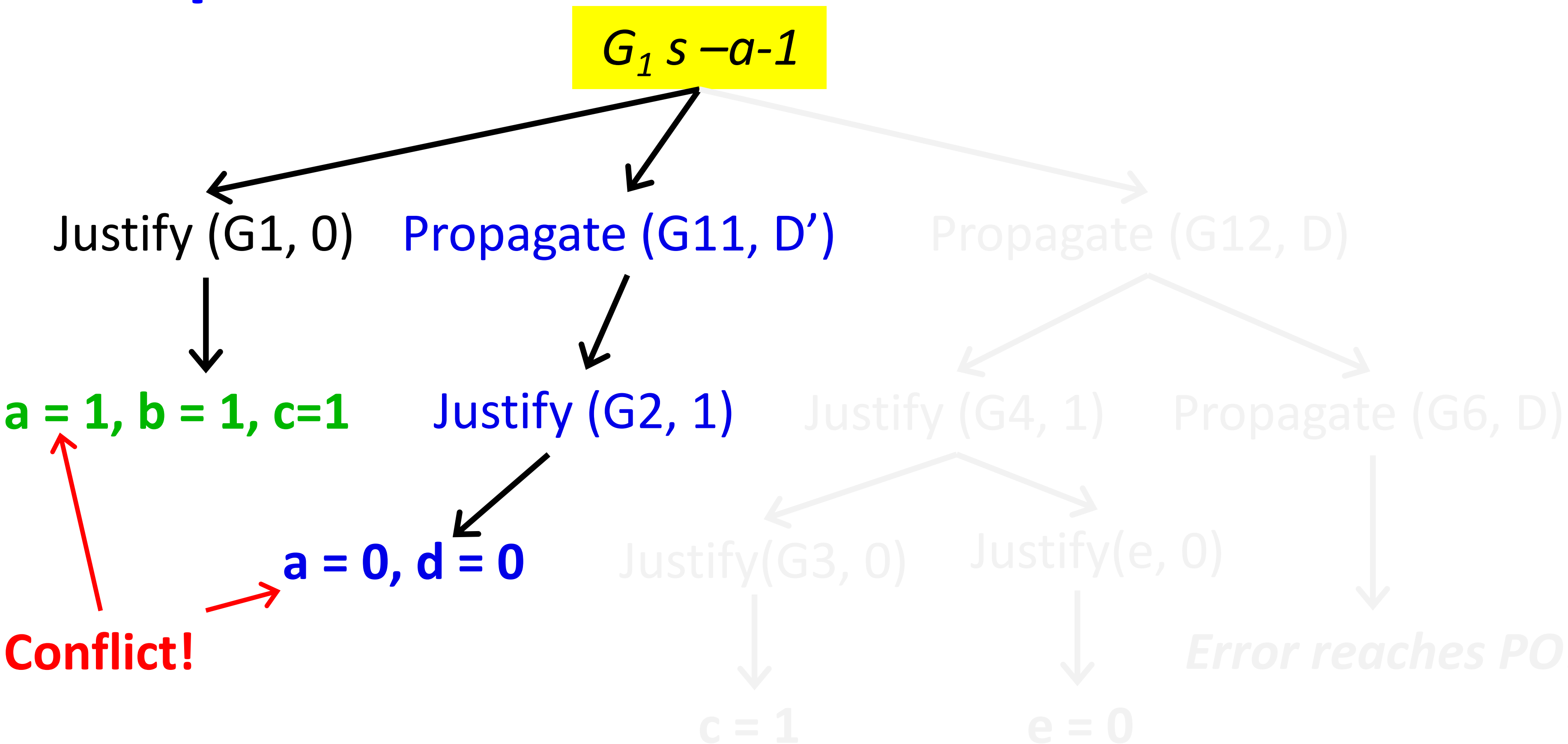
Example 6.2



Example 6.2

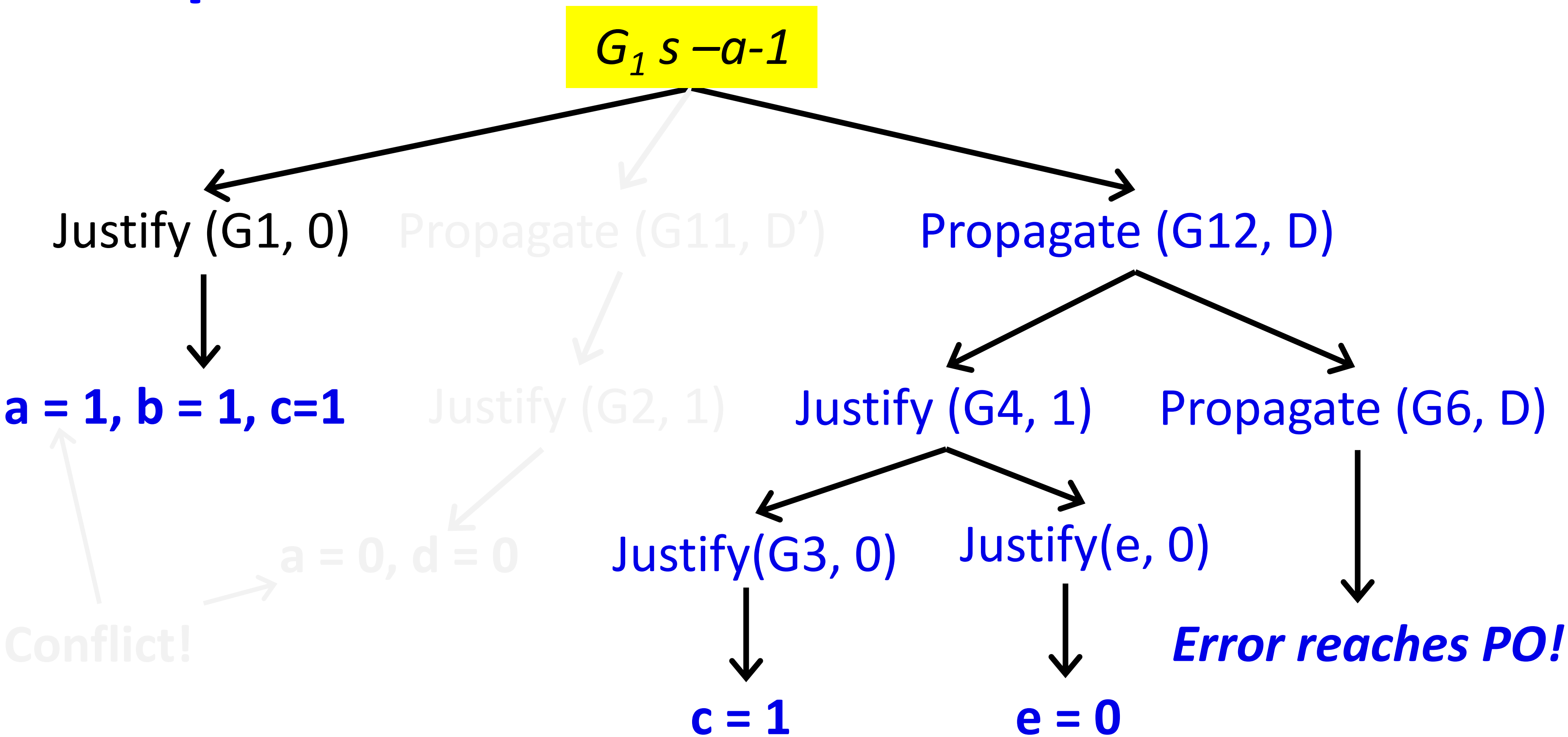


Example 6.2



G_1 s-a-1
Justify (G1, 0)
Propagate (G11, D')
Propagate (G12, D)
a = 1, b = 1, c=1
Justify (G2, 1)
Justify (G4, 1)
Propagate (G6, D)
a = 0, d = 0
Justify(G3, 0)
Justify(e, 0)
Conflict!
Error reaches PO!
c = 1
e = 0

Example 6.2

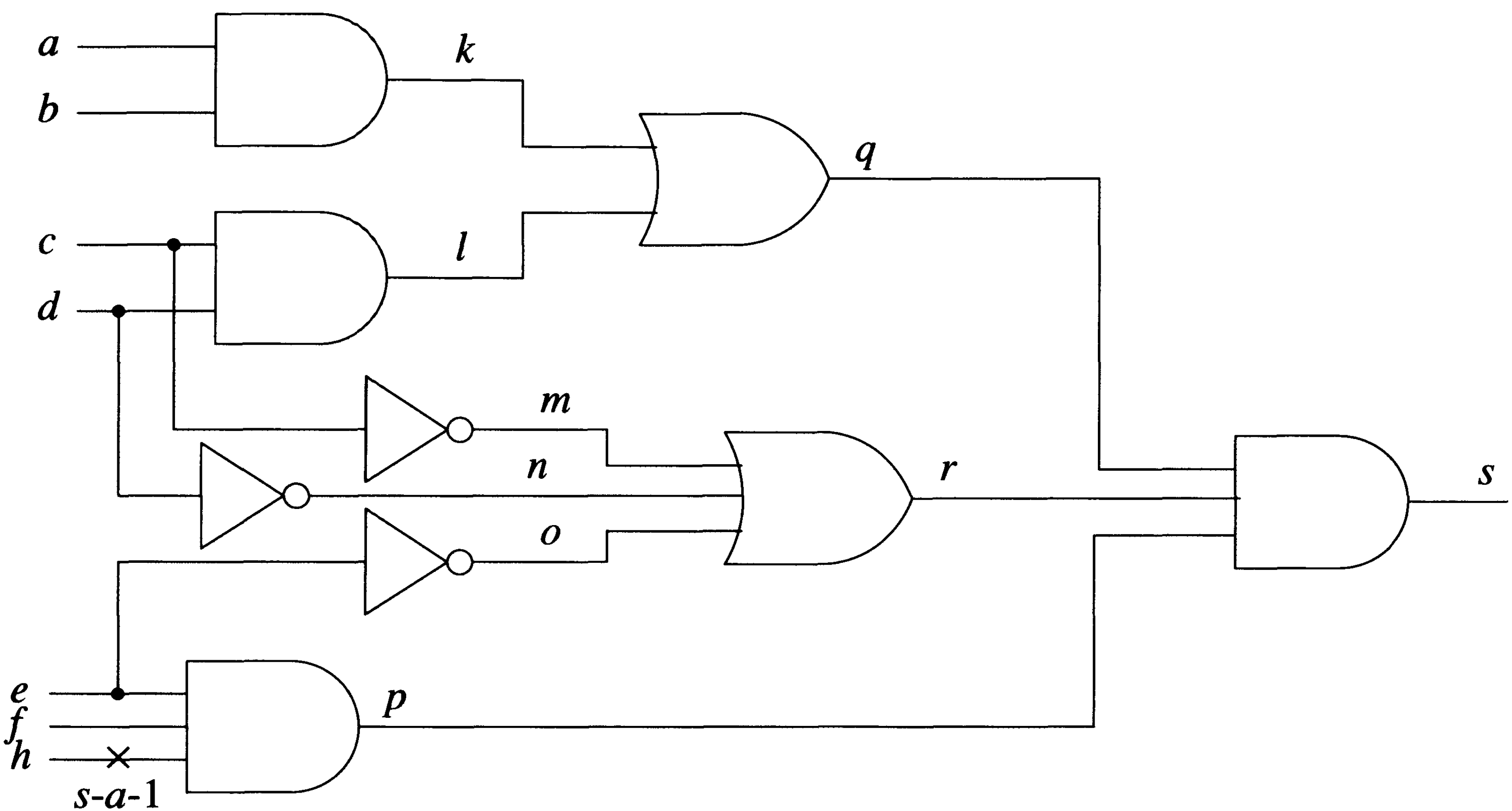


G1 s –a-1
Justify (G1, 0)
Propagate (G11, D')
Propagate (G12, D)
a = 1, b = 1, c=1
Justify (G2, 1)
Justify (G4, 1)
Propagate (G6, D)
a = 0, d = 0
Justify(G3, 0)
Justify(e, 0)
Conflict!
Error reaches PO!
c = 1
e = 0

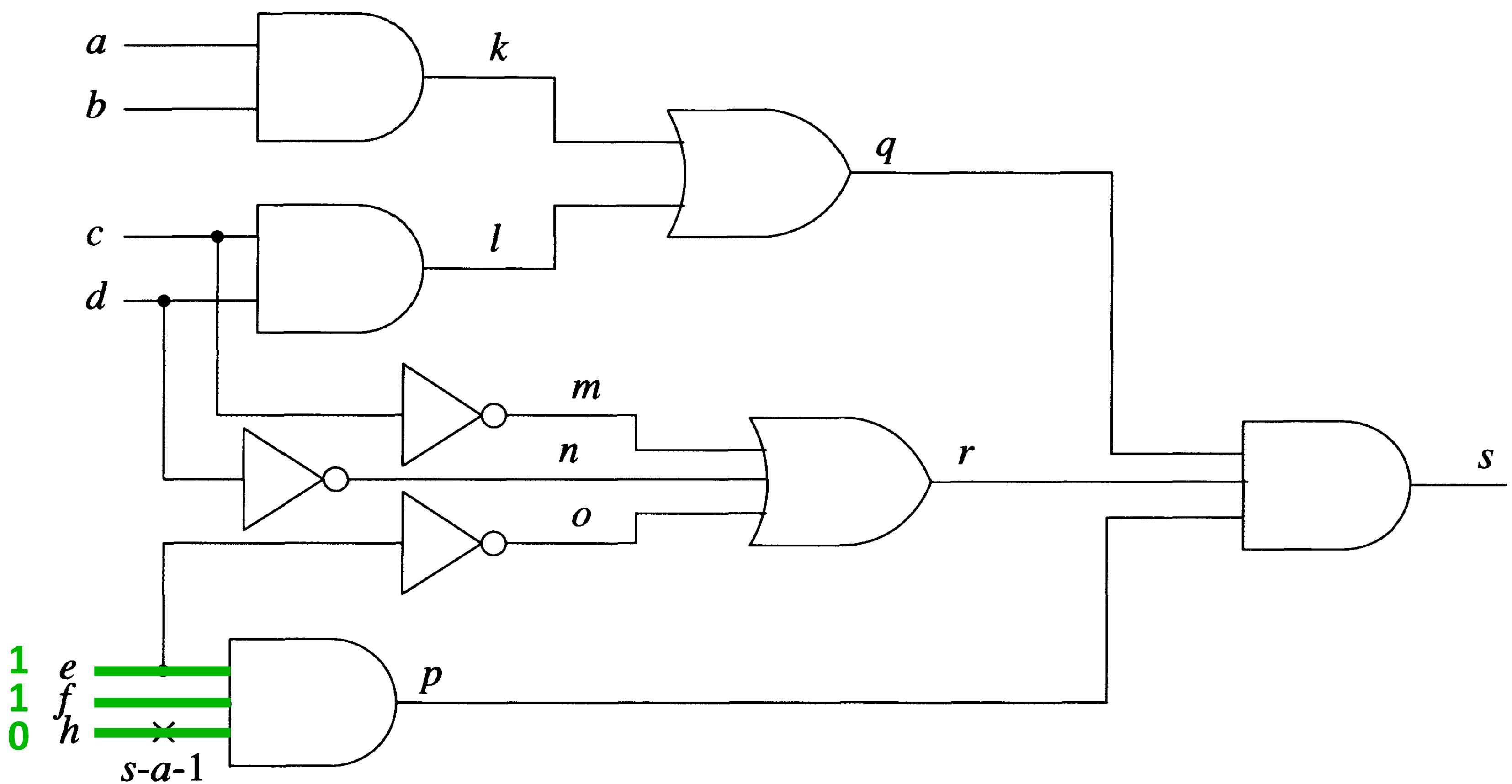
Backtracking Strategy

- Search for a test vector $\rightarrow$ decision process
- Several alternatives for a line justification problem
  - Pick one alternative
  - If it leads to an inconsistency, then backtrack!
- Backtracking Strategy
  - Systematic exploration
  - Recovery from incorrect decisions
    - Invert all values assigned since last decision

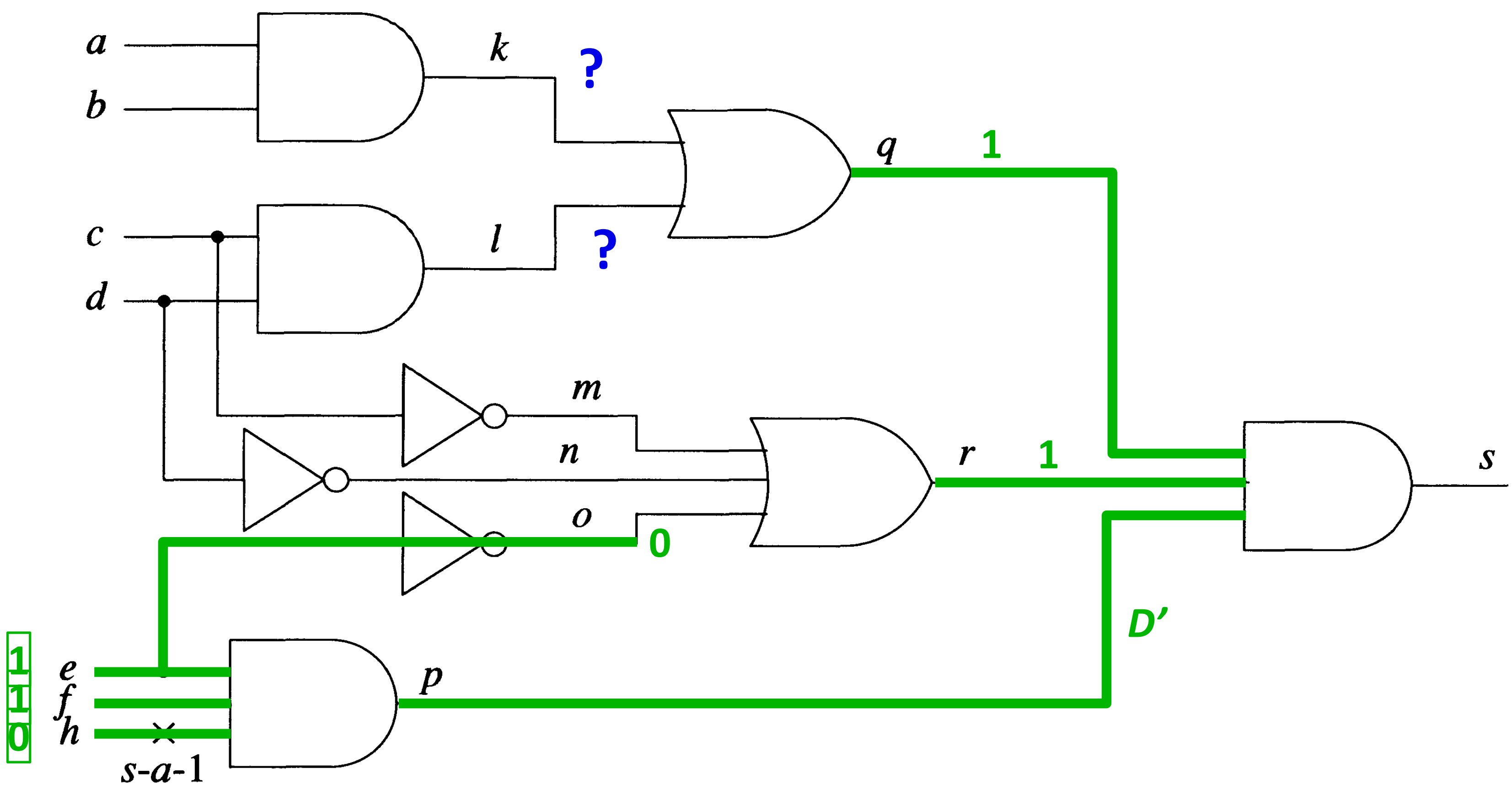
Example 6.3



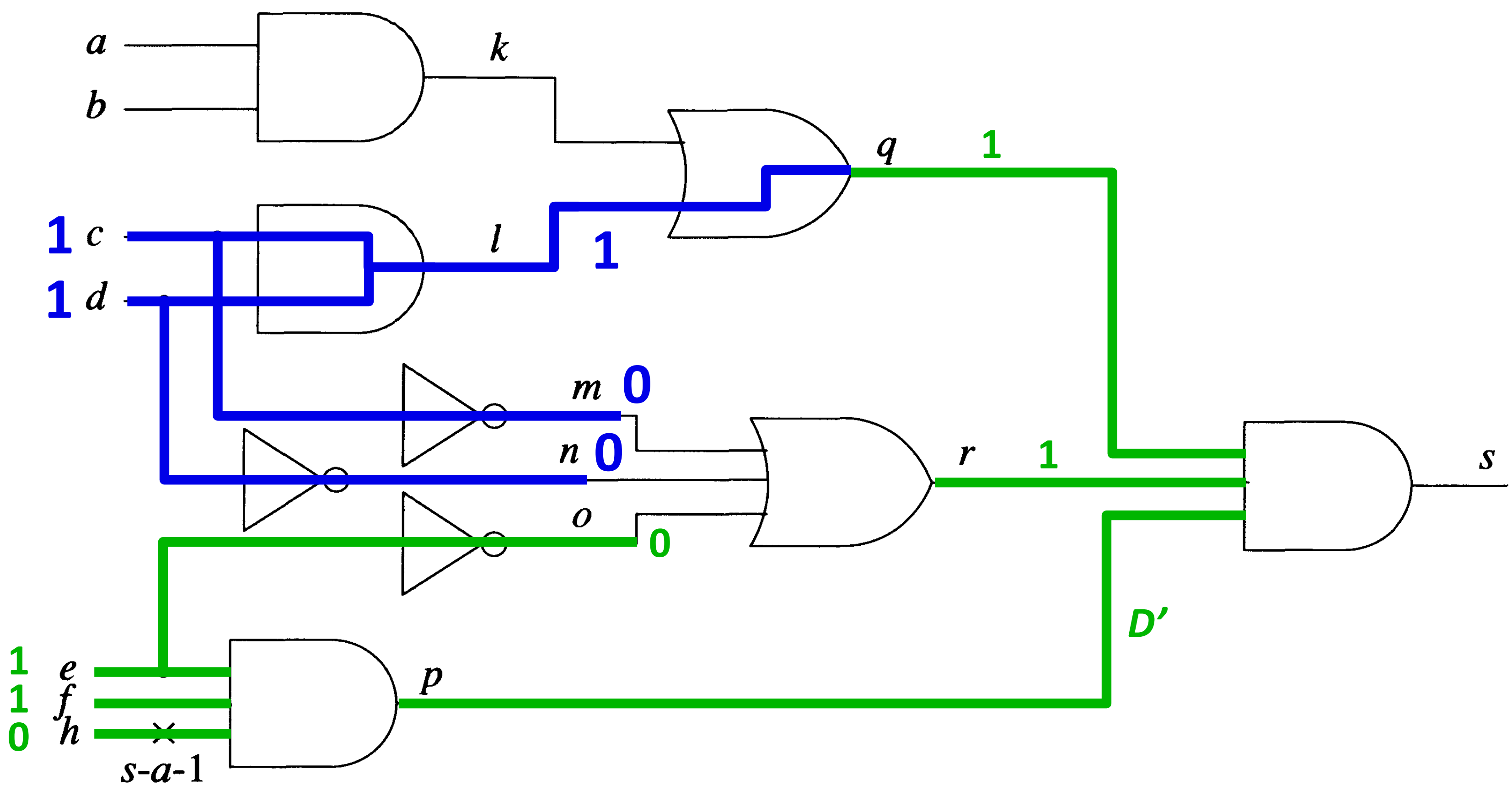
Example 6.3



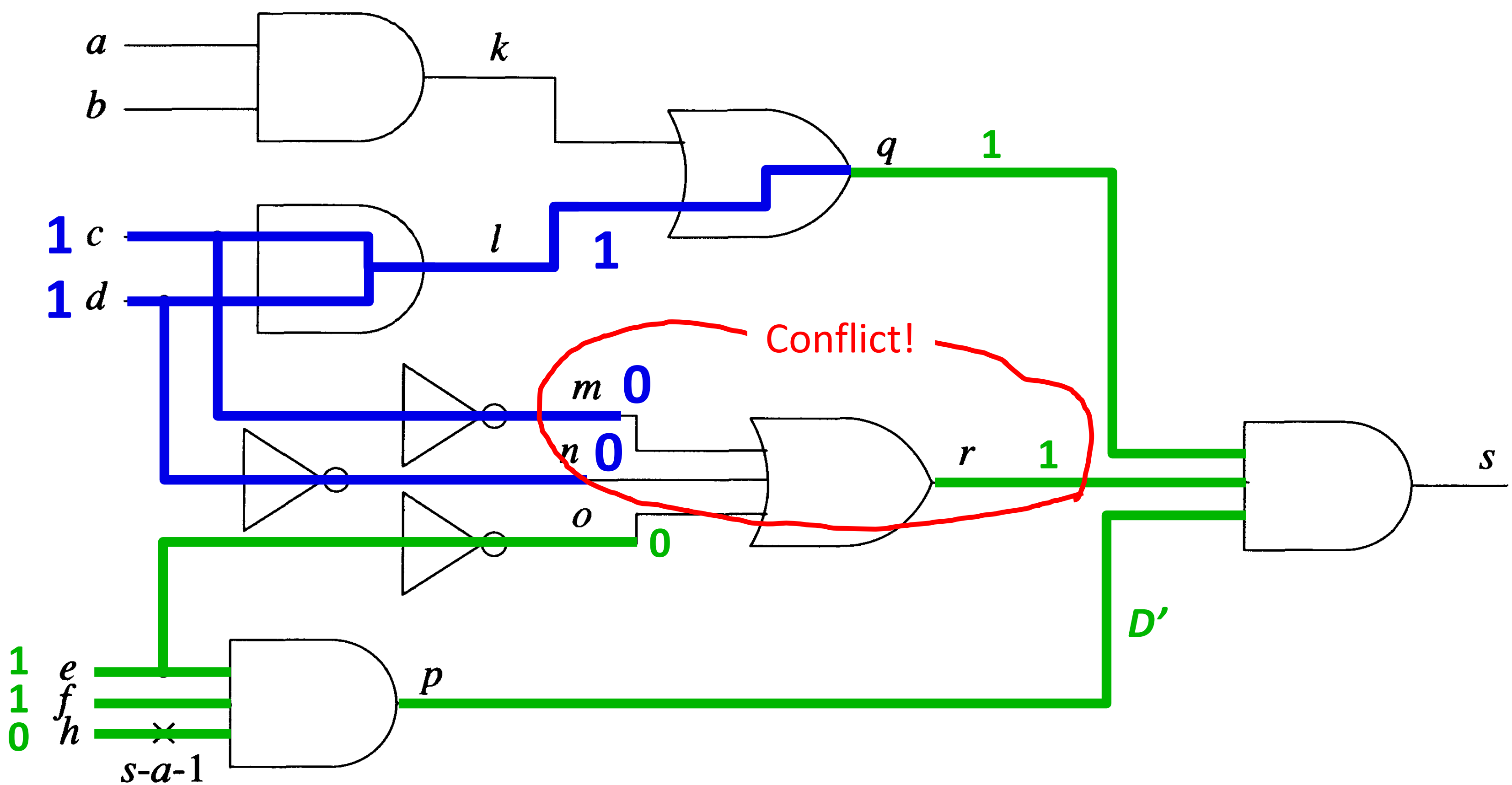
Example 6.3



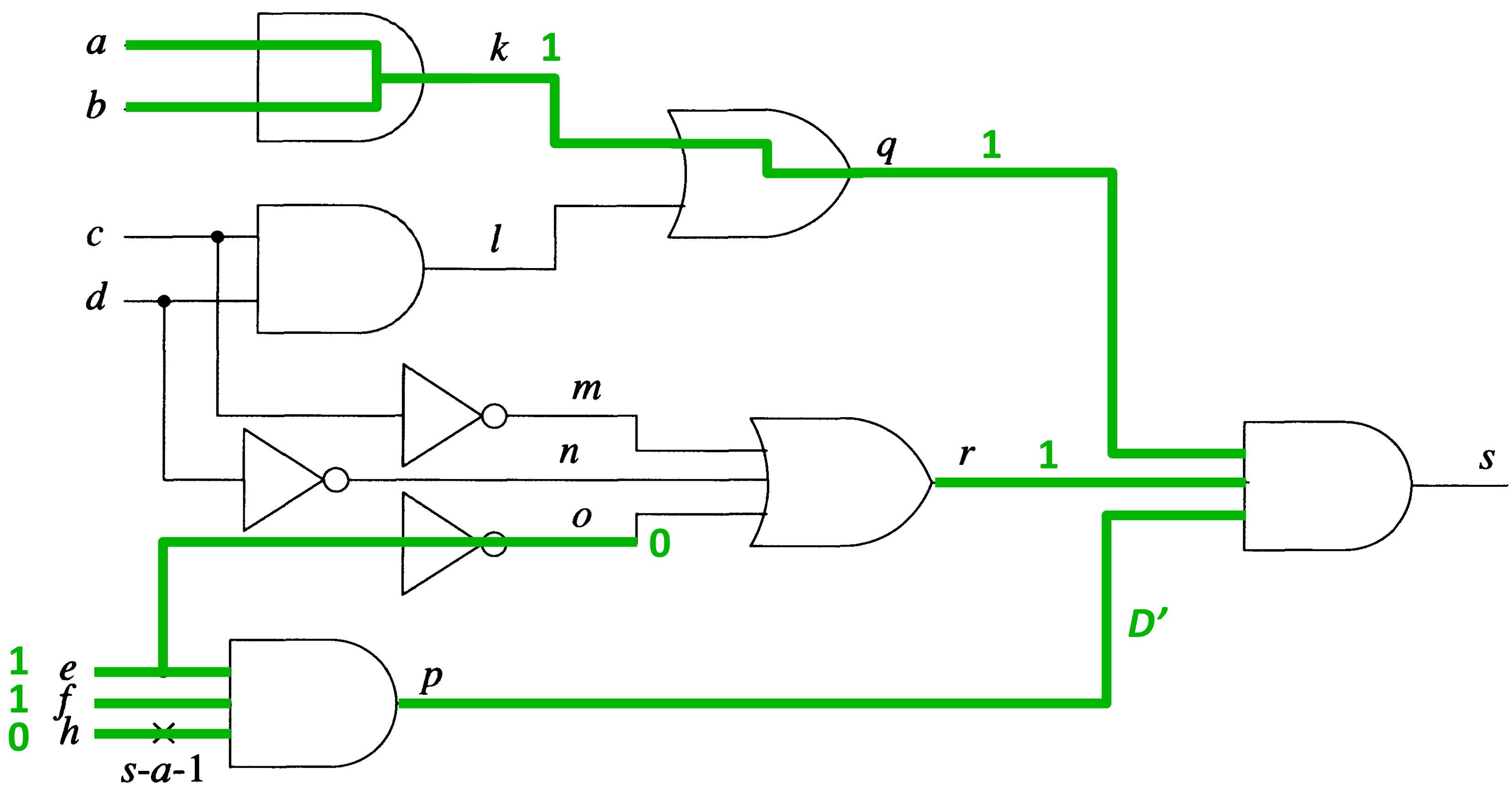
Example 6.3



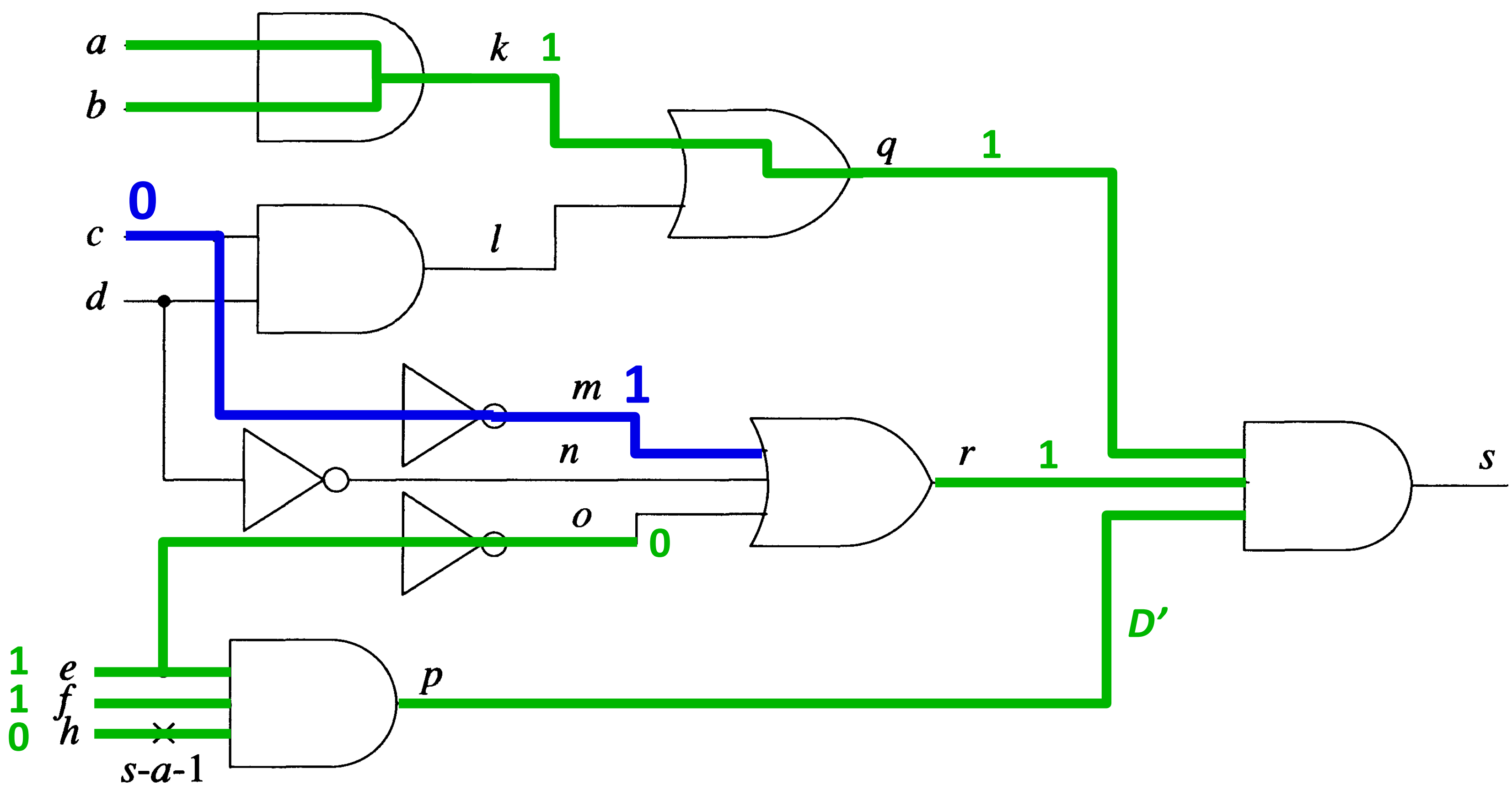
Example 6.3



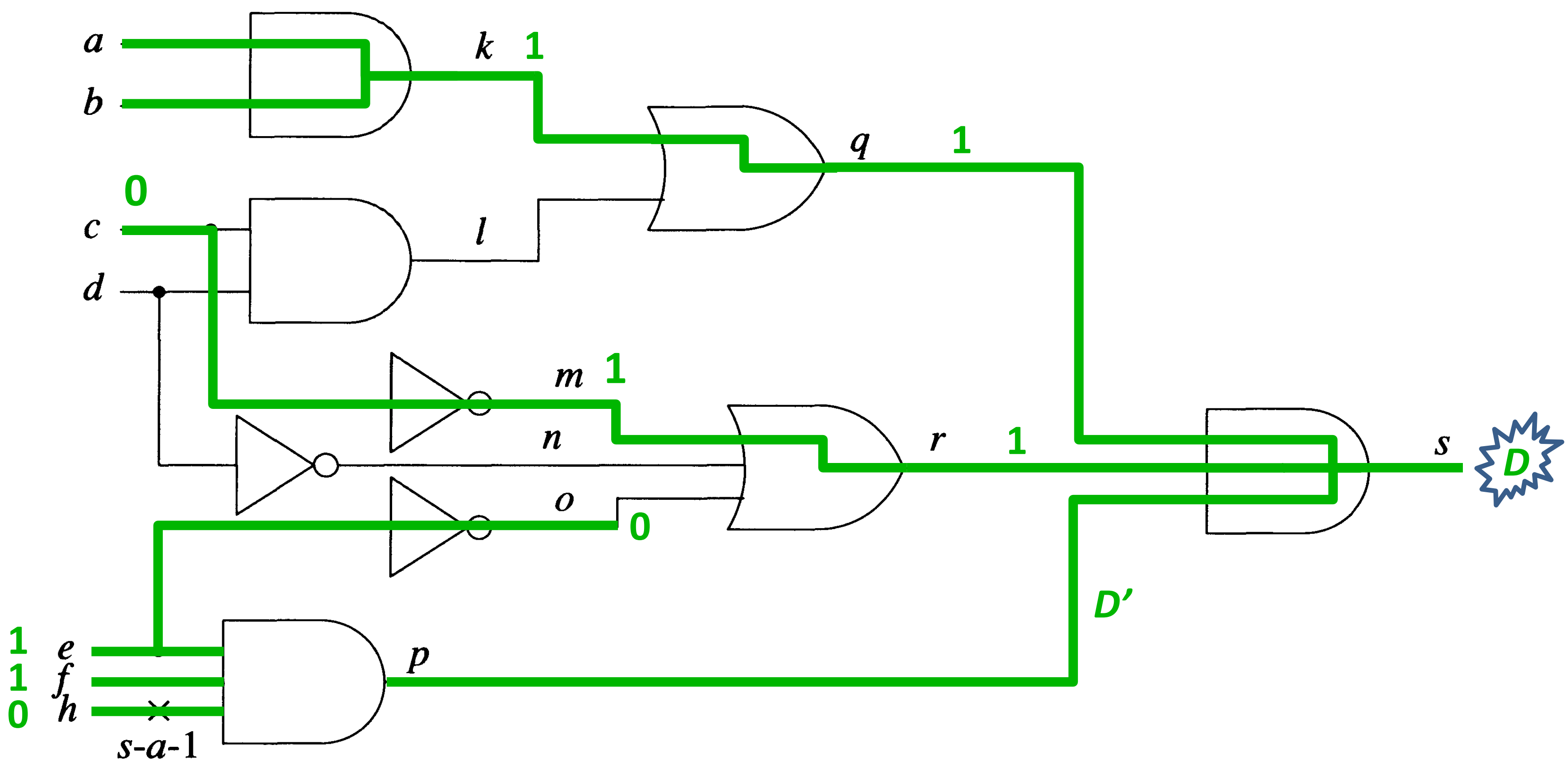
Example 6.3



Example 6.3



Example 6.3



Decision: choose one alternative if there are multiple alternatives to justify() or propagate()

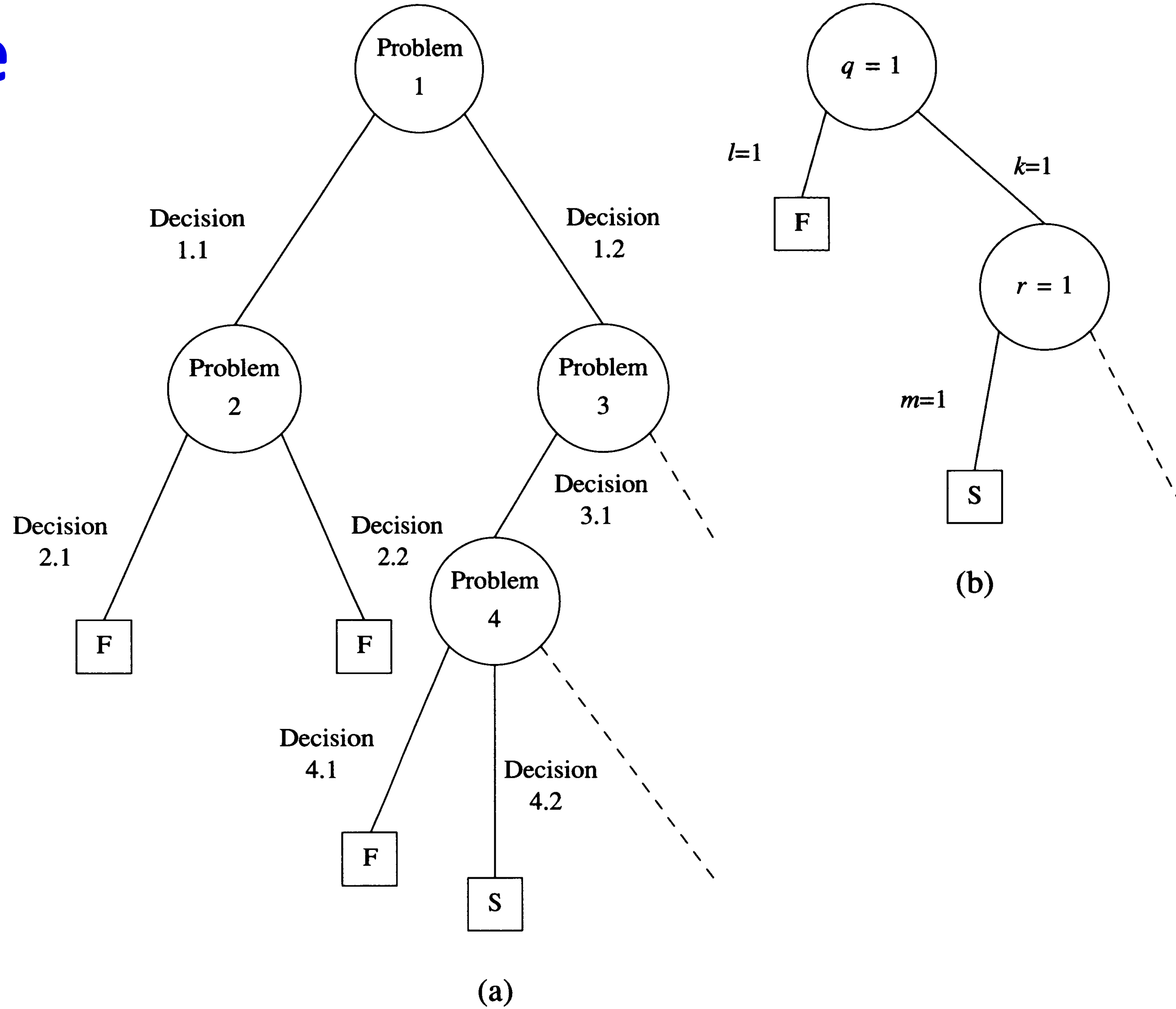
Implication: compute new values as a result of **decision**, and check inconsistencies.

Decisions	Implications	Remarks
	h = D' e = 1 f = 1 p = D' r = 1 q = 1 o = 0 s = D'	Initial Implications
l =1	c = 1 d = 1 m = 0 n = 0 r = 0	To justify q=1 **Contradiction**
k = 1	a = 1 b = 1	To justify q = 1
m = 1	c = 0 *l* = 0	To justify r=1

Fig 6.10 TG Algorithm Outline

```
Solve()
begin
    if Imply_and_check() = FAILURE then return FAILURE
    if (error at PO and all lines are justified)
       then return SUCCESS
    if (no error can be propagated to a PO)
       then return FAILURE
    select an unsolved problem
    repeat
       begin
          select one untried way to solve it
          if Solve() = SUCCESS then return SUCCESS
       end
    until all ways to solve it have been tried
    return FAILURE
end
```

Decision Tree



6.2.1.1 Common Concepts

TG Failure for an Undetectable Fault

- Solve() is exhaustive – guarantee to find a test if one exists.
  - worst case complexity is exponential

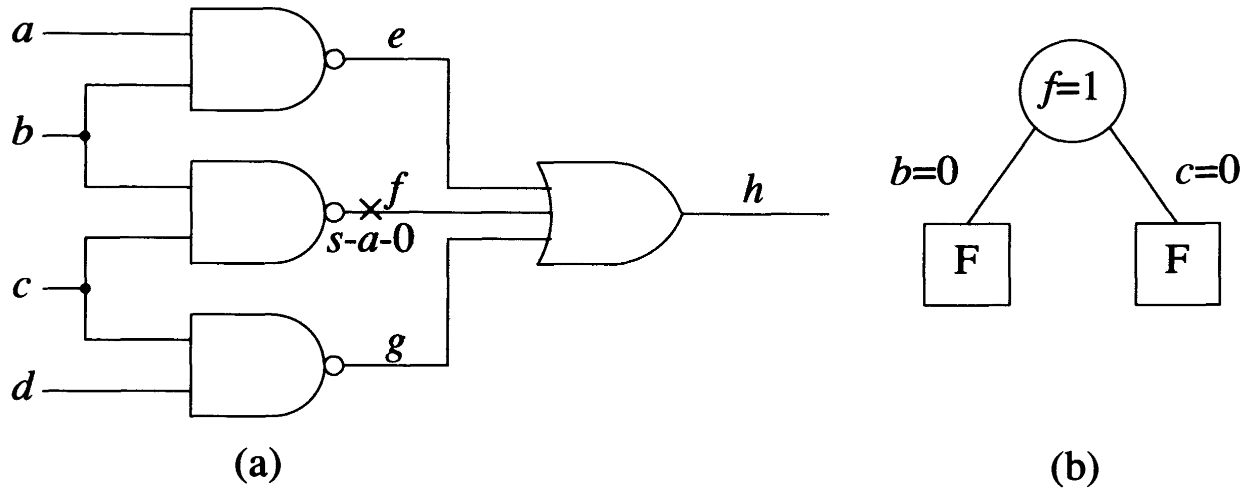
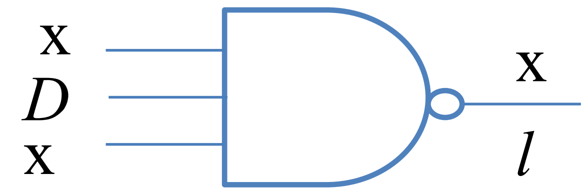


Figure 6.12 TG failure for an undetectable fault (a) Circuit (b) Decision tree

D-Frontier

- ***D-frontier*** – all gates whose output value is currently *x* but have one or more error signals on their inputs.
- *D-drive* operation –

  Pick a gate and try to propagate error

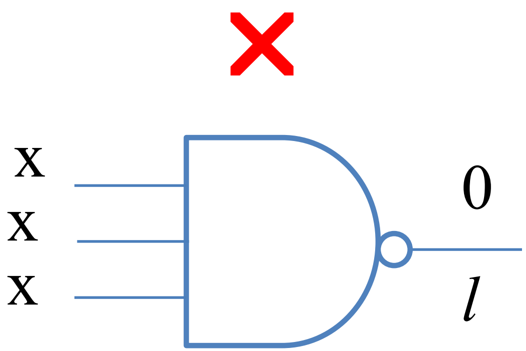


- If ***D*-frontier** becomes empty
  - ⇒ No error can be propagated to PO
  - ⇒ Backtracking should occur

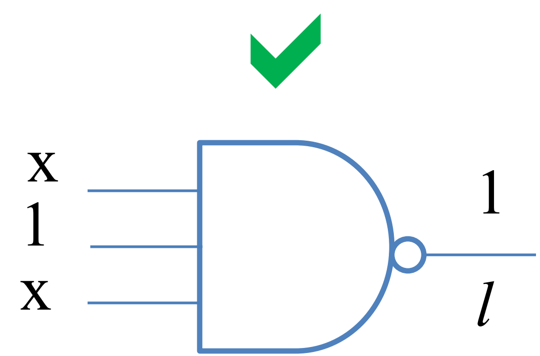
Gates in D-frontier indicate necessary decisions in order to proceed.

J-Frontier

- ***J-frontier*** – all gates whose output value is known, but not implied by its input values
- Helps keep track of *currently unsolved* line-justification problems



All inputs are implied to be 1.



No implications on x-inputs.

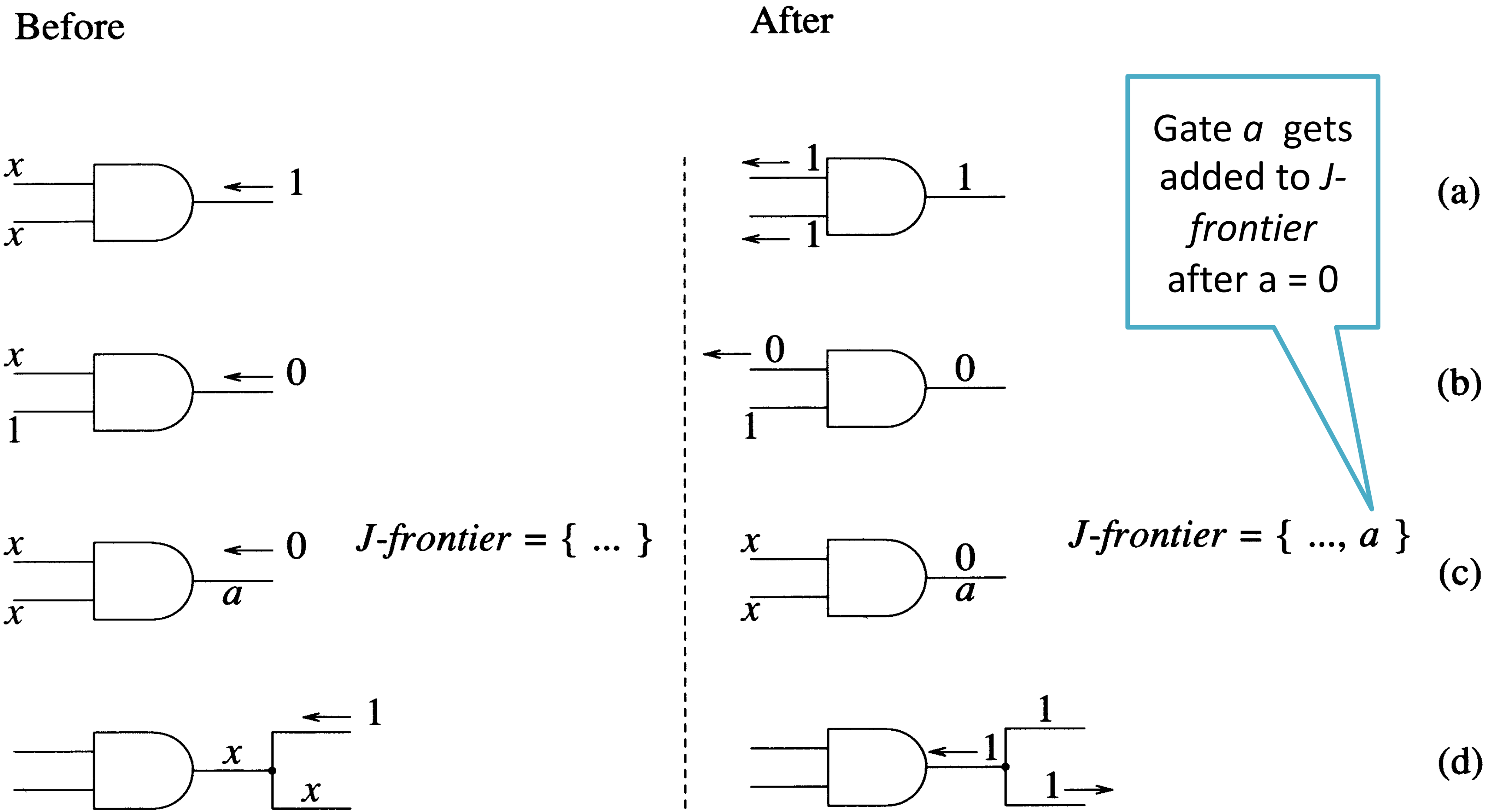
Implication Process

3 Steps:

1. Compute all values that can be uniquely determined by implication
2. Check for consistency and assign values
3. Maintain the *D-frontier* and *J-frontier*

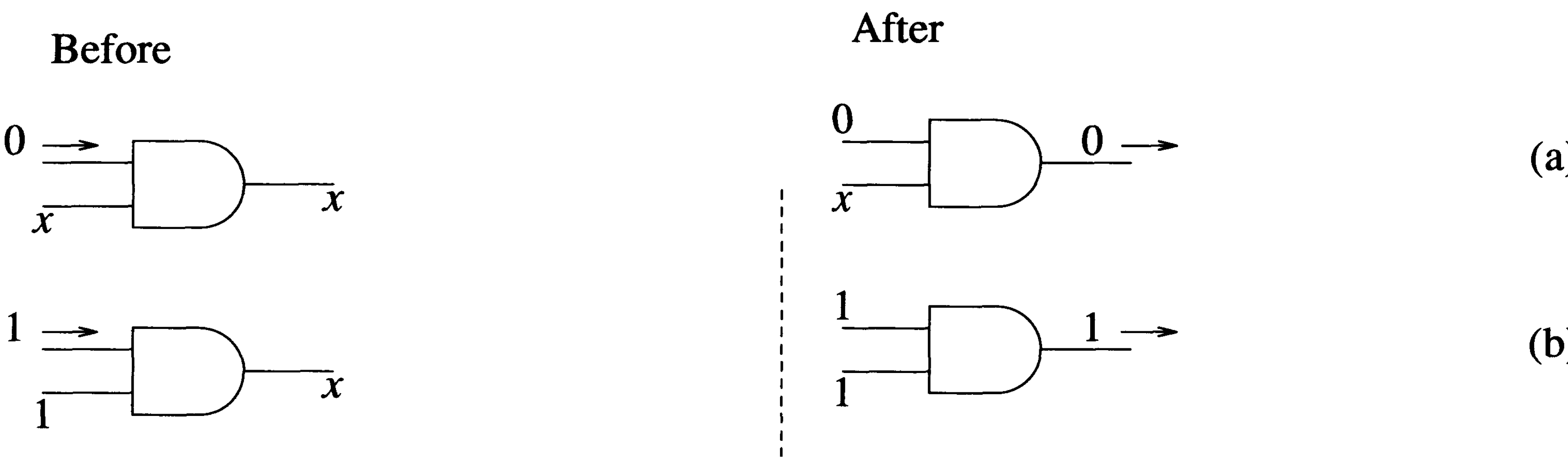
Implication can be forward or backward.

Backward Implication Propagation



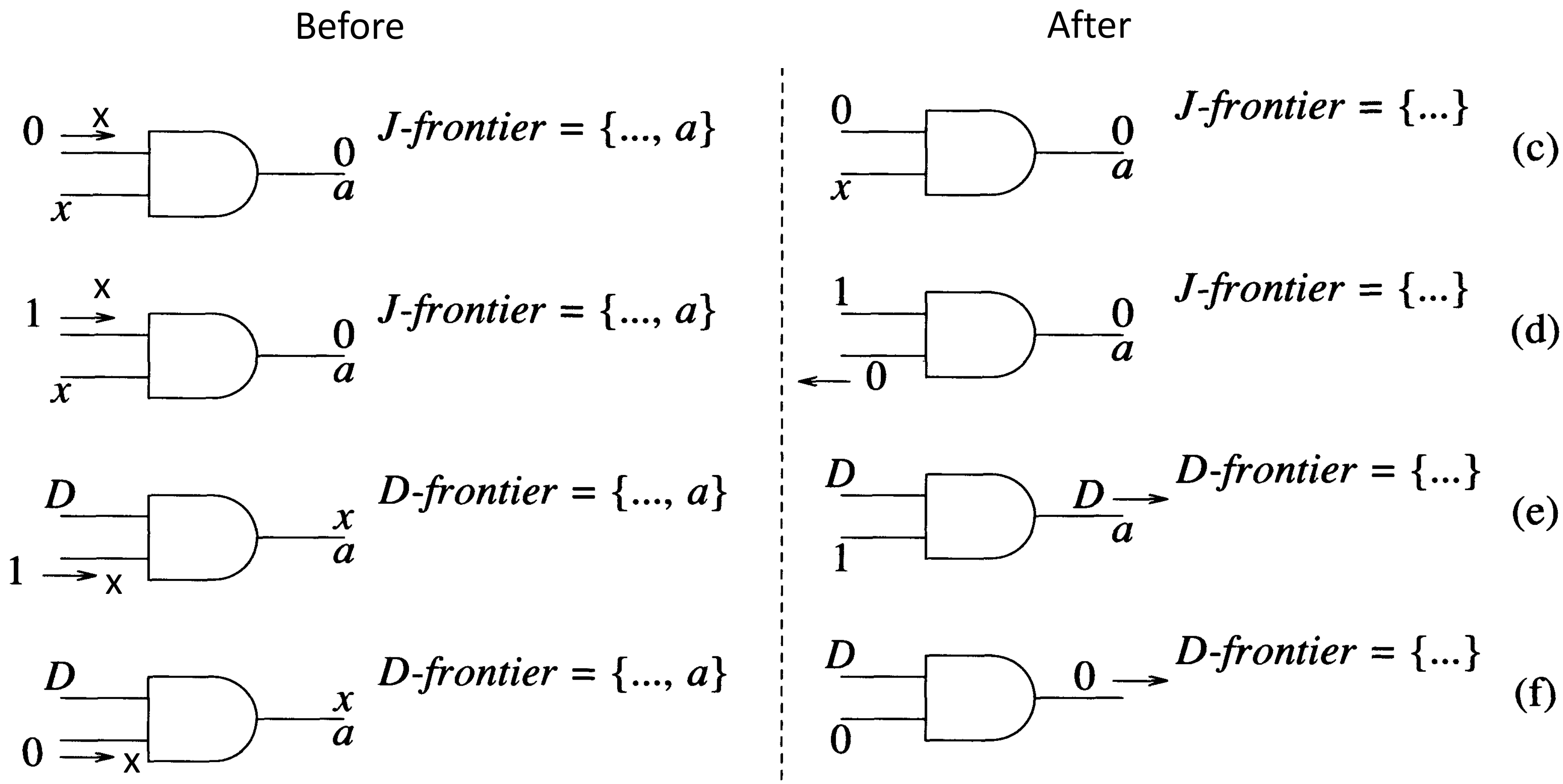
Forward Implication Propagation

Figure 6.15



Forward Implication Propagation

Figure 6.15



Forward Implication Propagation – cont'd

Figure 6.16

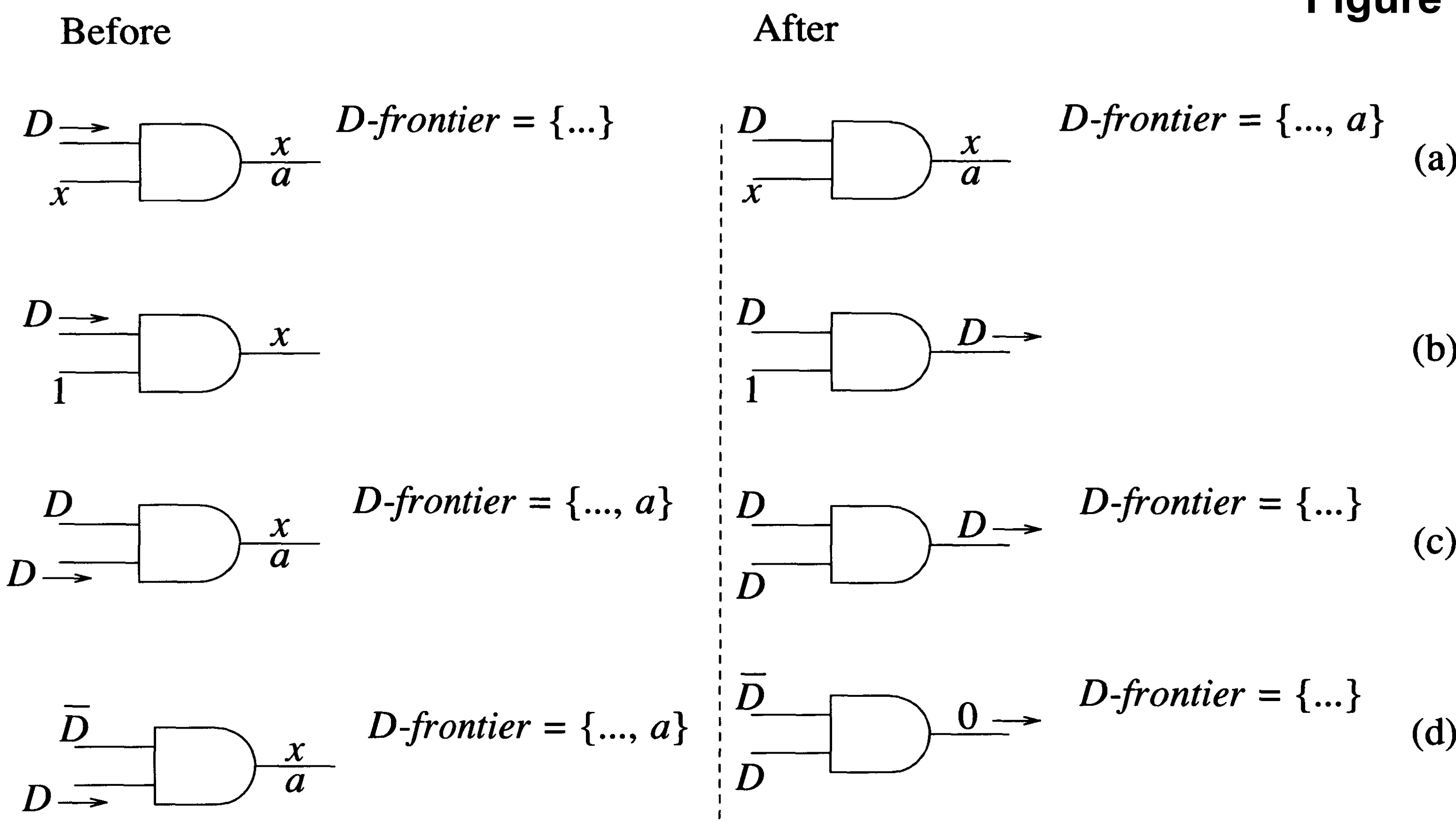


Figure 6.17 Unique *D*-Drive

- When only gate remains in the *D*-frontier.
  - There is only one way to propagate *D.*

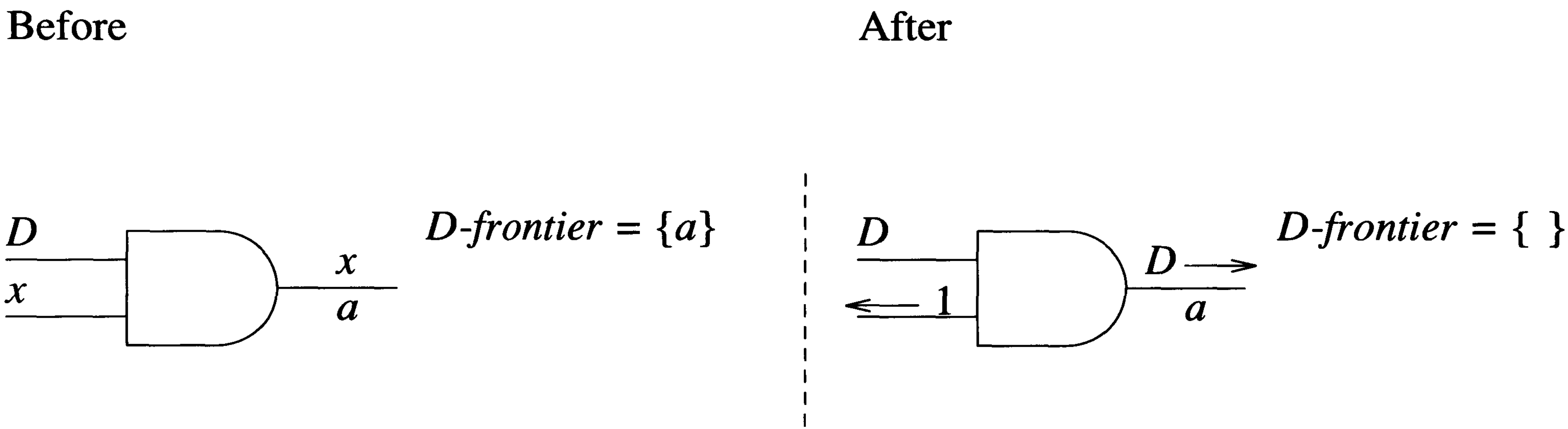
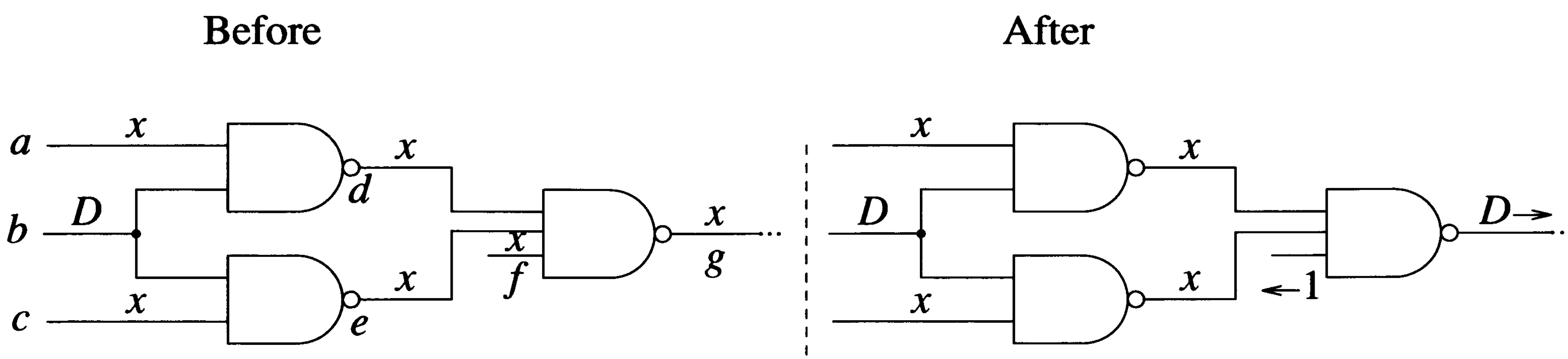


Figure 6.18 Future Unique *D*-drive

- *D*-frontier = {d, e}
- Eventually we end up unique *D*-drive with gate *g* only



This type of propagations is *global implication*.

Reversing Incorrect Decisions

- Assume that *a = 0* failed ***irrespective*** of *b* and *c*

  ⇒ *a* must be *1*!

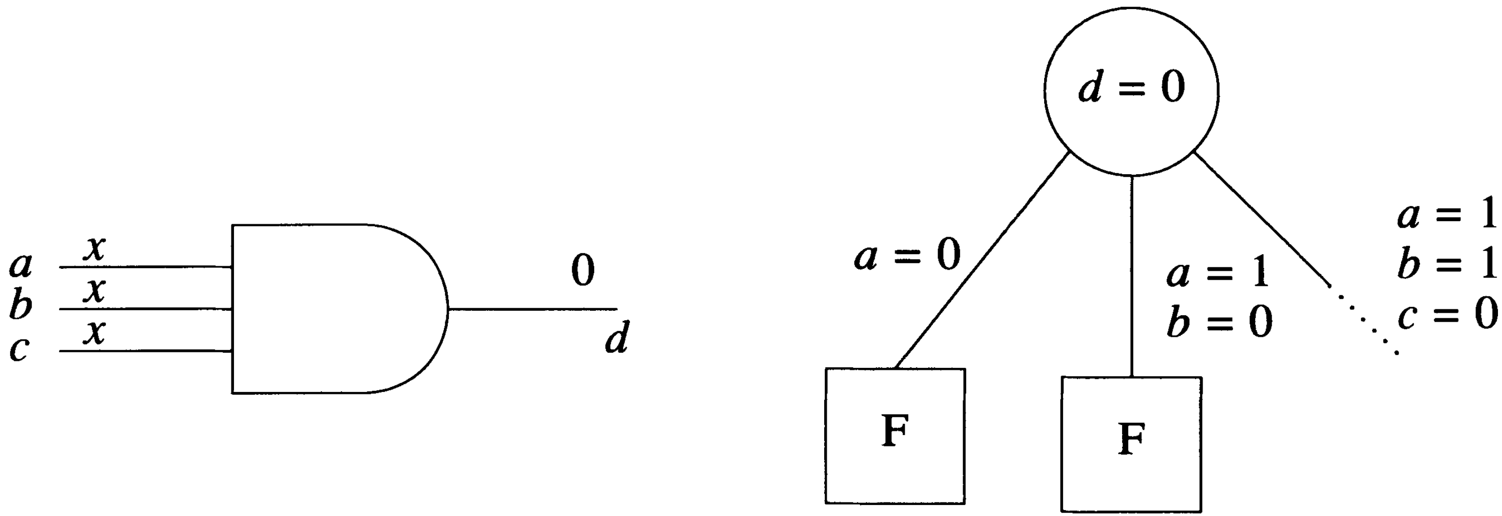


Figure 6.21 Reversing incorrect decisions

Look-Ahead in Error Propagation

- No matter how we propagate, D-frontier will be empty!
- Look-ahead: Error propagation is possible only if there is at least one ***x-path*** from gate *G* in *D*-frontier to at least one PO. (a necessary condition)
- X-paths used to avoid failed decisions.

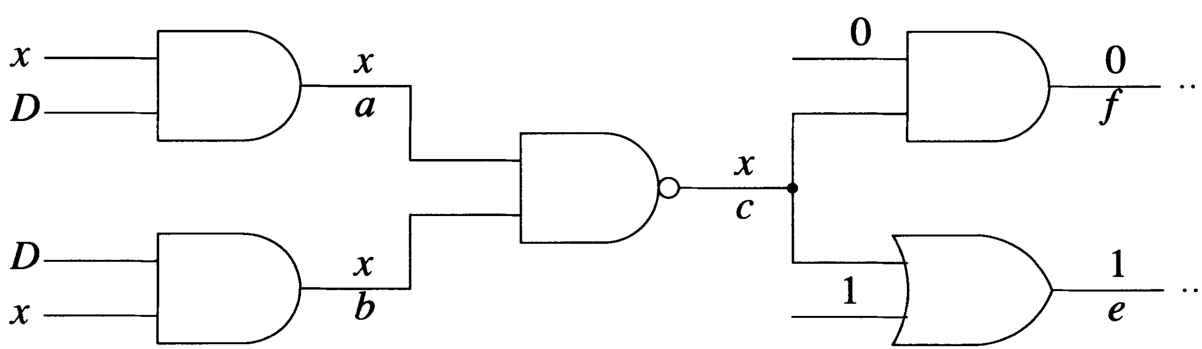


Figure 6.22 The need for look-ahead in error propagation

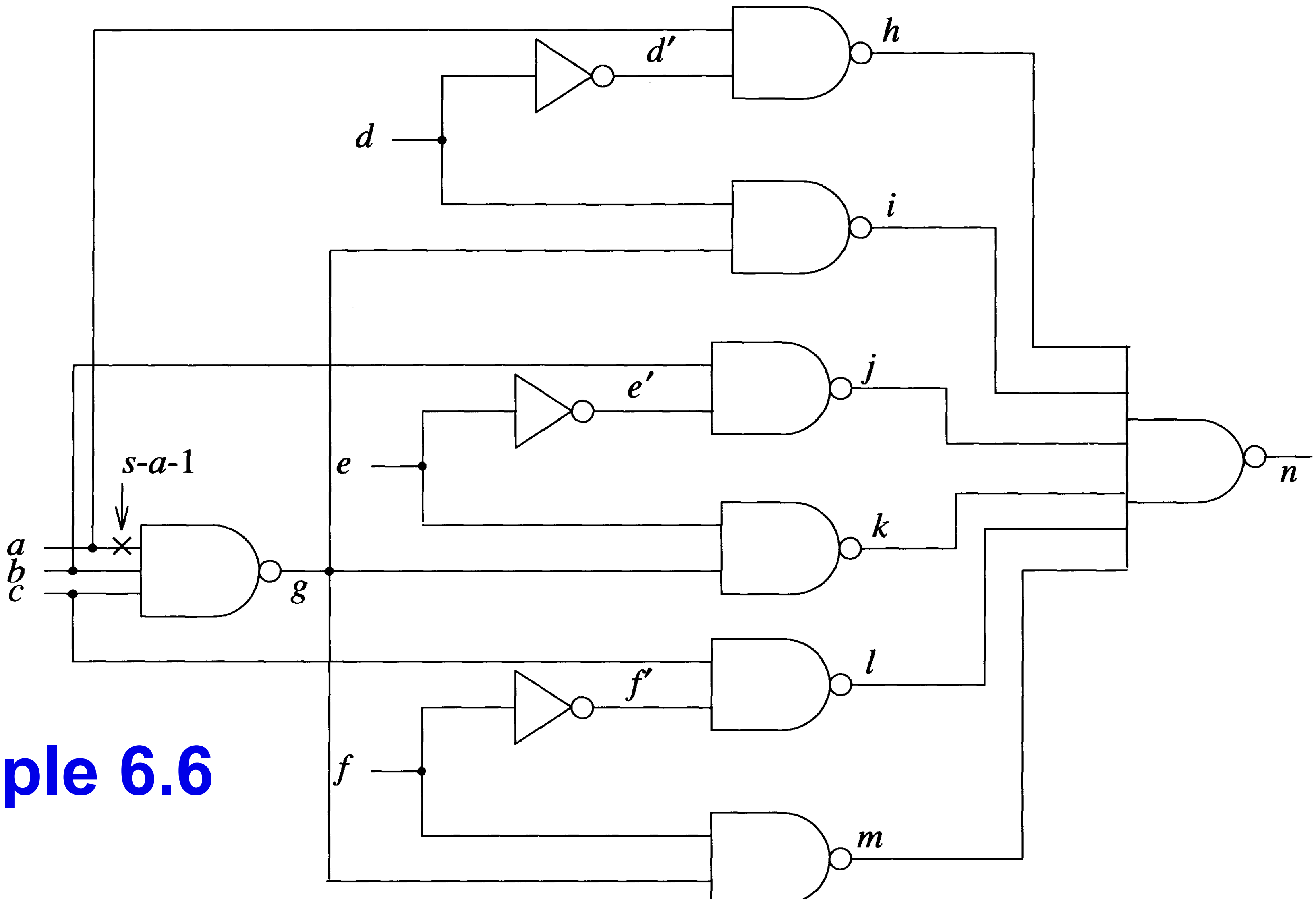
D-Algorithm

- Ability to propagate errors on several reconvergent fanouts
- We assume error propagation is given priority over justification problems (simplifying assumption)
- "assign" means "add the value to the assignment queue"
- *Imply_and_check()* handles the assignments

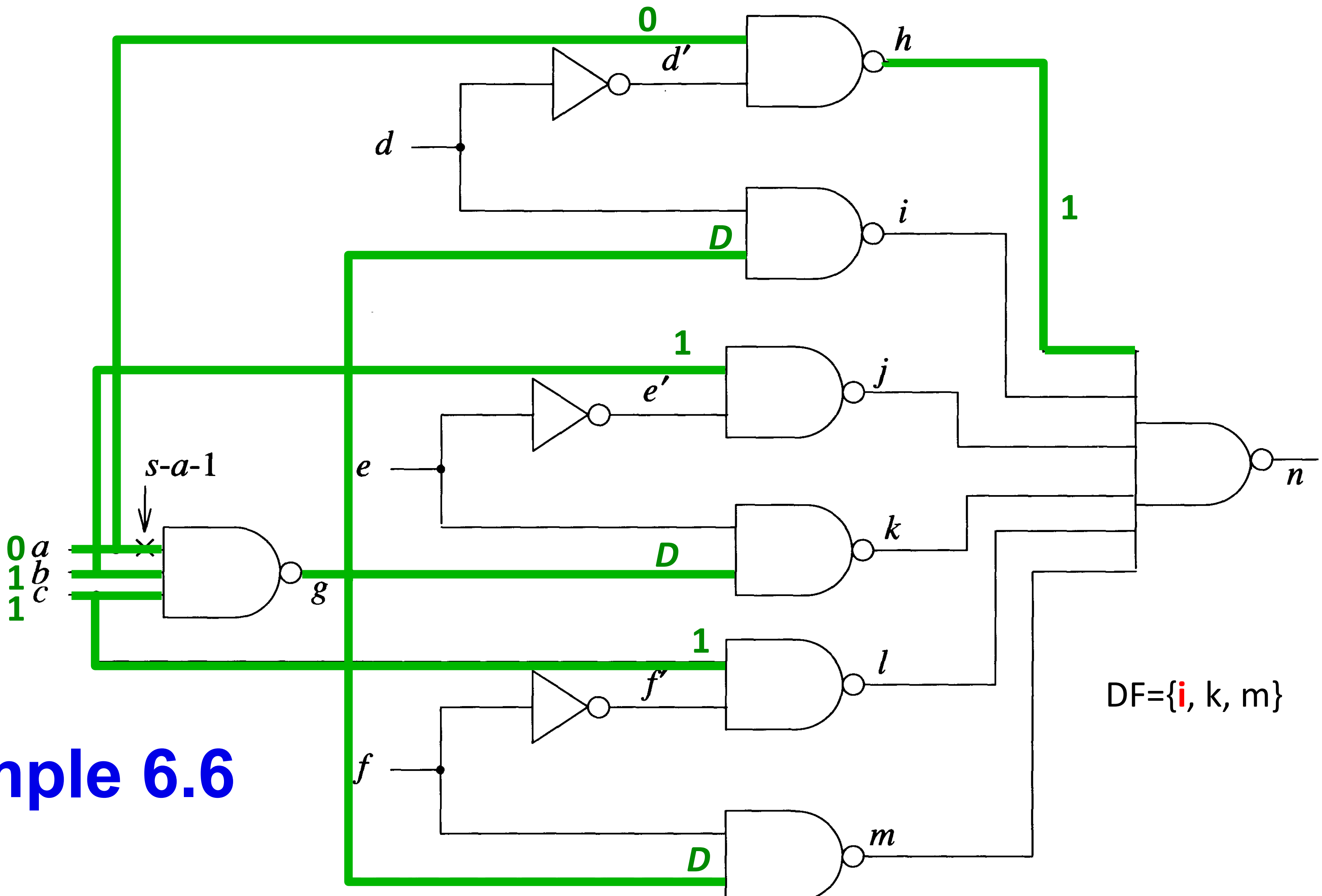
```
D-alg()
begin
    if Imply_and_check() = FAIL then return FAIL
    if (error not at PO) then   /* error propagation */
      begin
            if D-frontier = ∅ then return FAIL
            repeat
              begin
                    select an untried gate (G) from D-frontier
                    c = controlling value of G
                    assign c' to every input of G with input x
                    if D-alg() = SUCCESS then return SUCCESS
              end
            until all gates from D-frontier have been tried
            return FAIL
      end
```

Continued...

```
    /* error propagated to a PO */
    if J-frontier = ∅ then return SUCCESS
    select a gate (G) from the J-frontier
    c = controlling value of G
    repeat
      begin
        select an input (j) of G with value x
        assign c to j
        if D-alg() = SUCCESS then return SUCCESS
        assign c' to j  /* reverse decision */
      end
    until all inputs of G are specified
    return FAIL
end
```

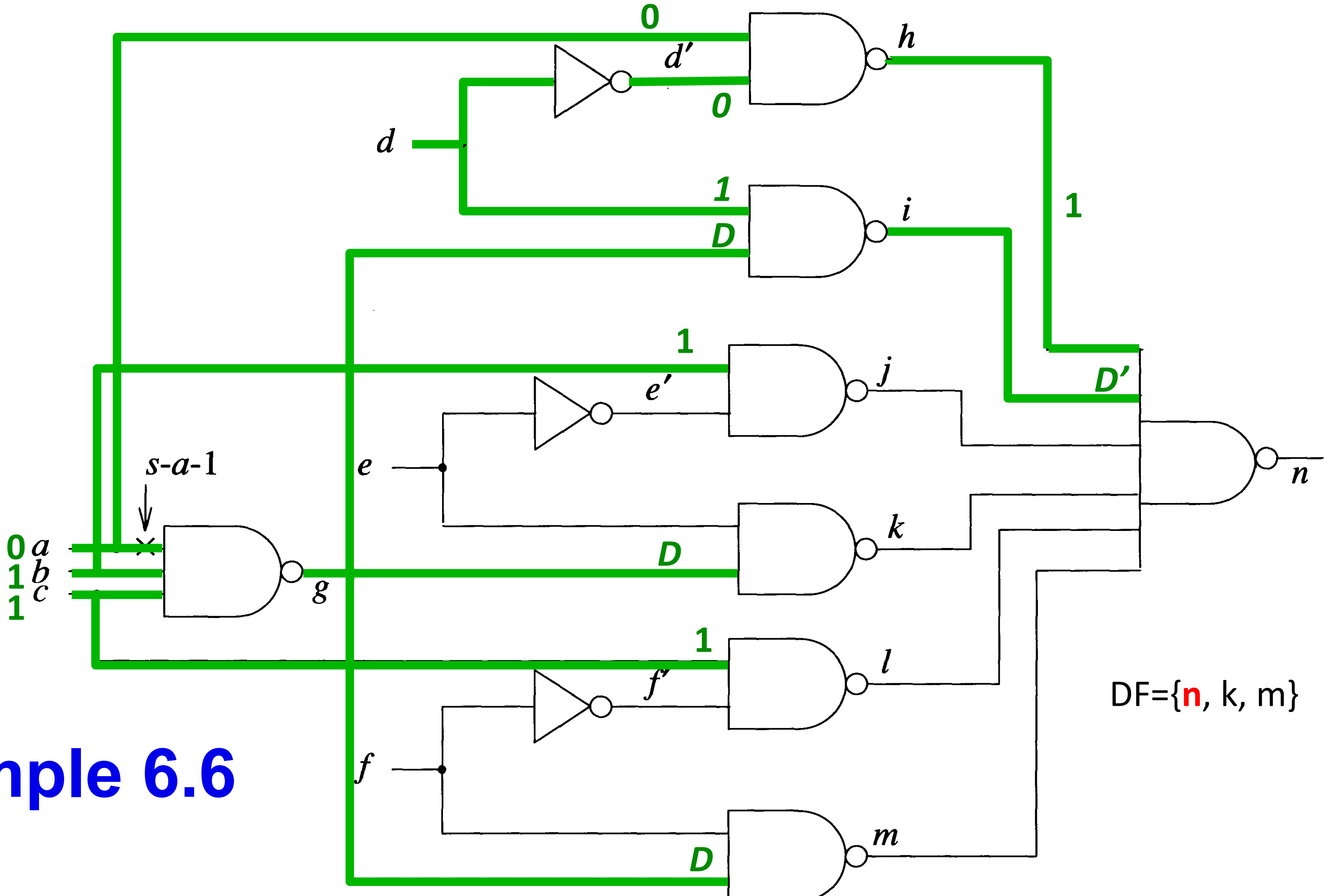


Example 6.6



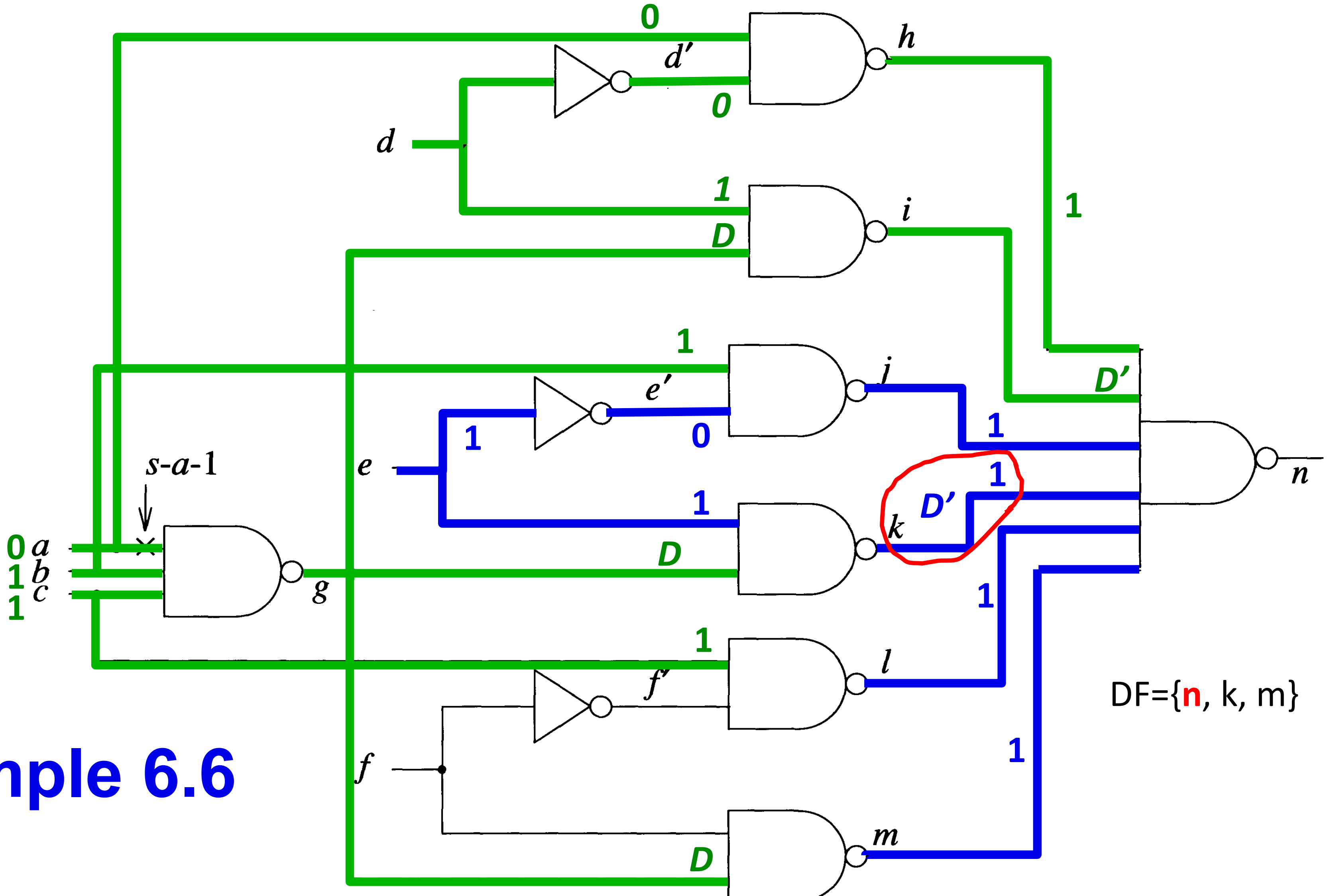
DF={i, k, m}

Example 6.6



DF={n, k, m}

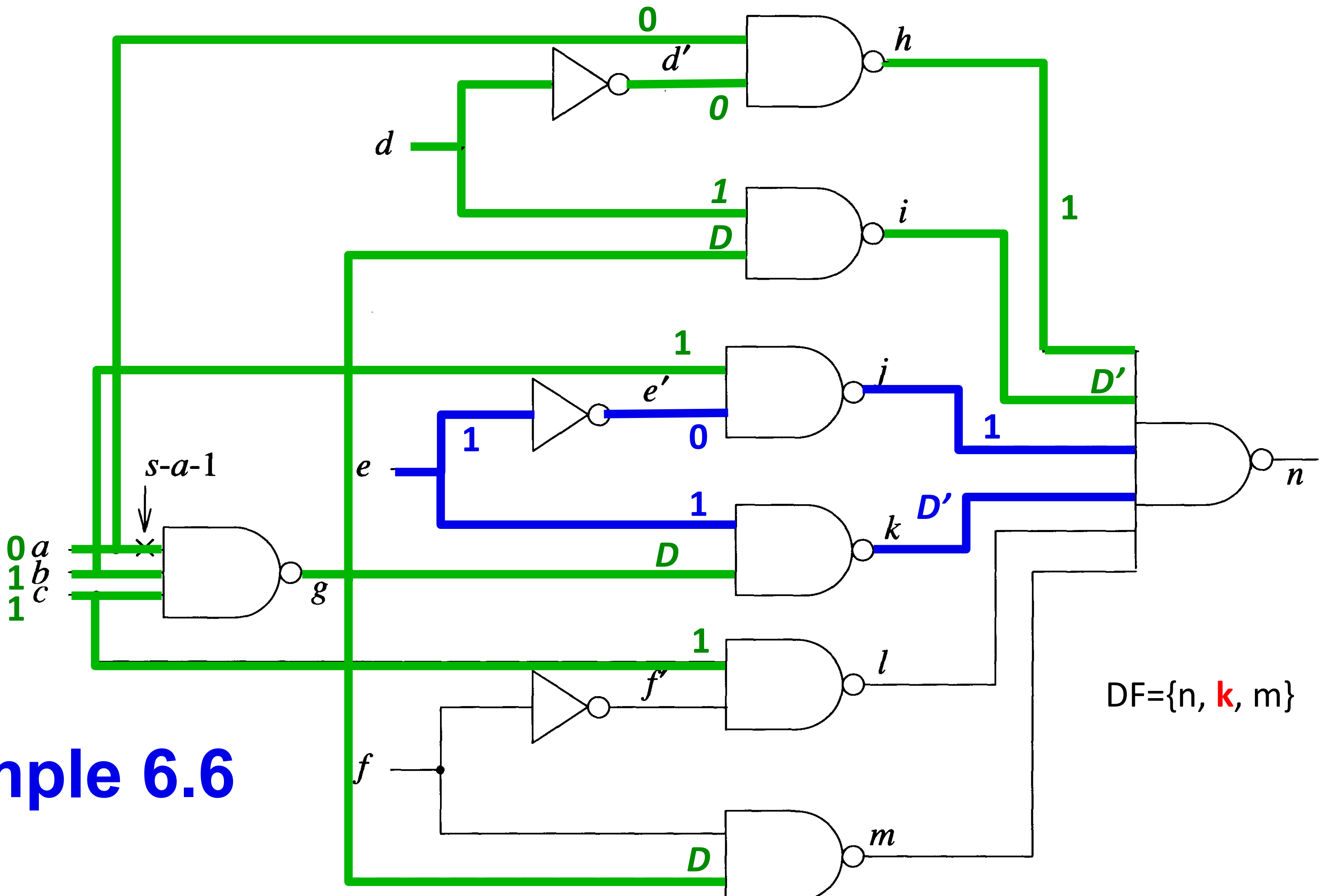
Example 6.6



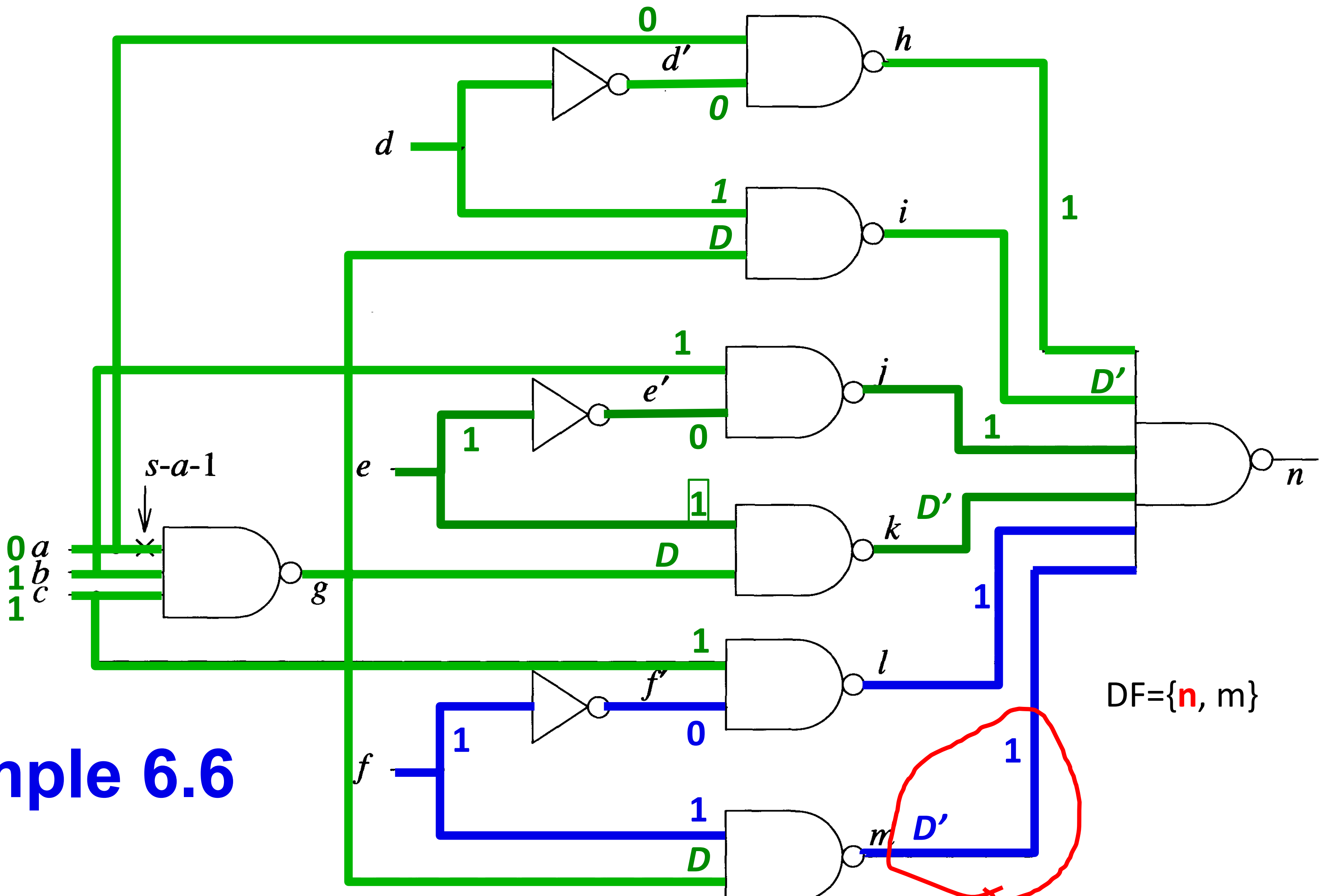
DF={**n**, k, m}

Example 6.6

Example 6.6



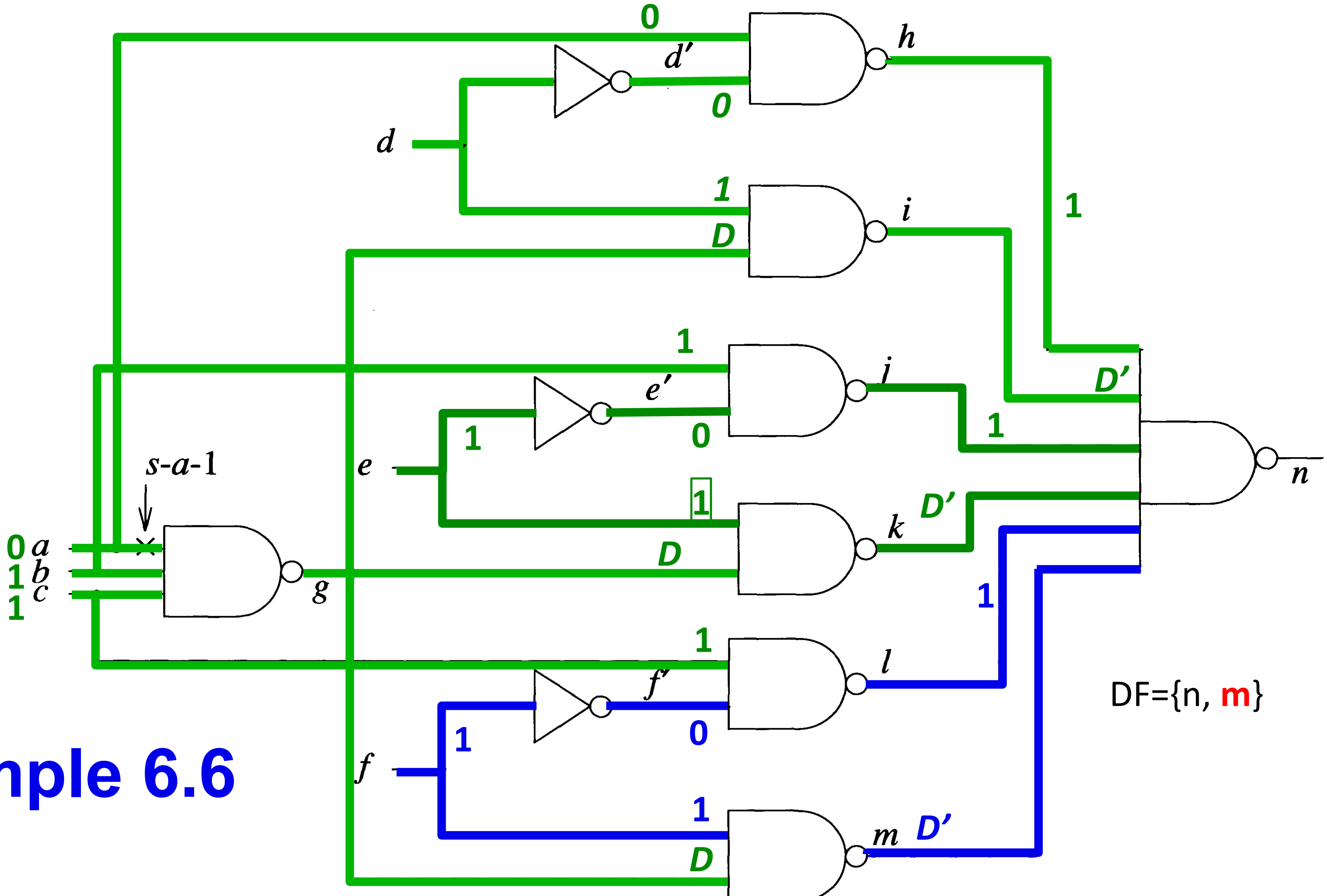
DF={n, k, m}



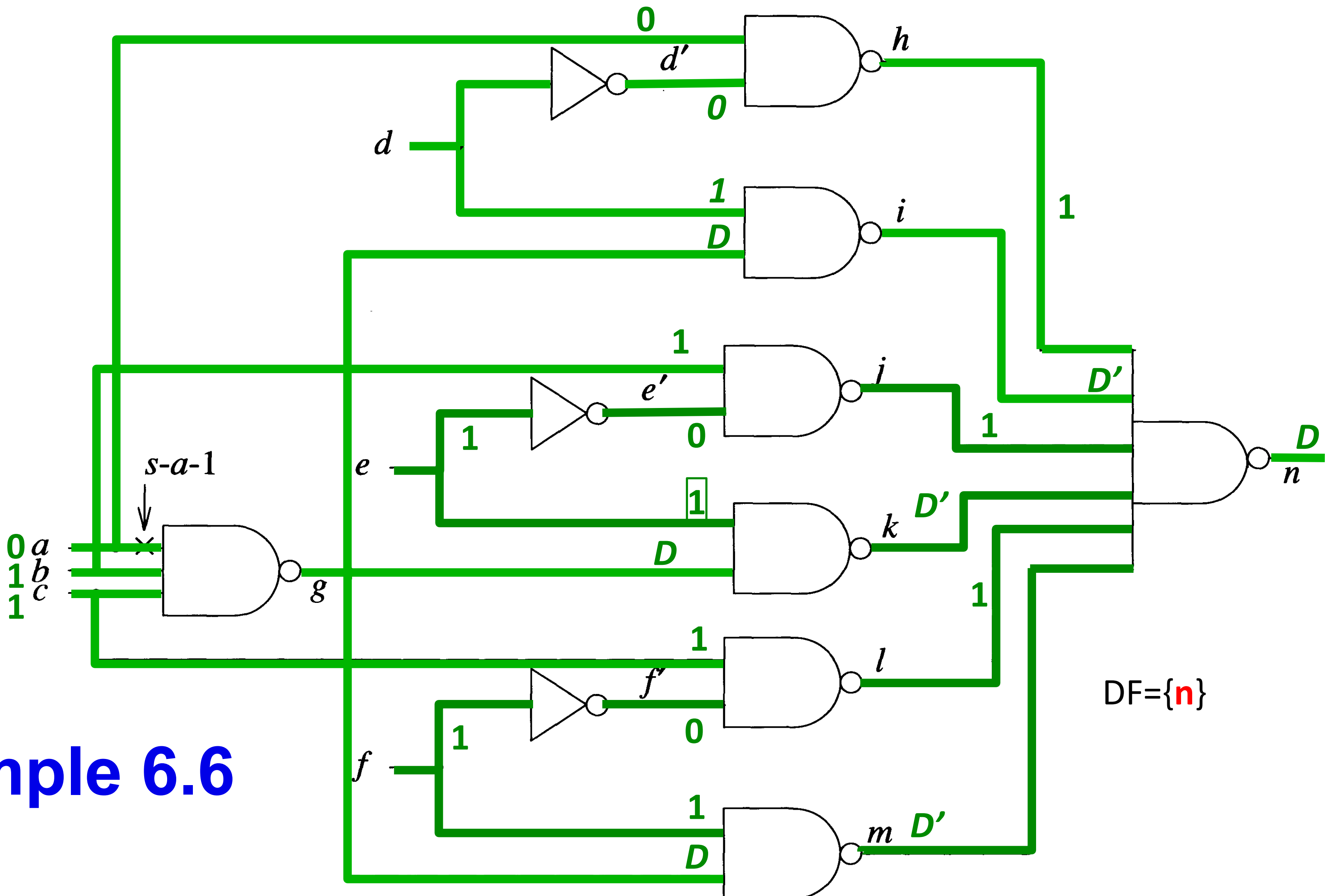
DF={n, m}

Example 6.6

Example 6.6

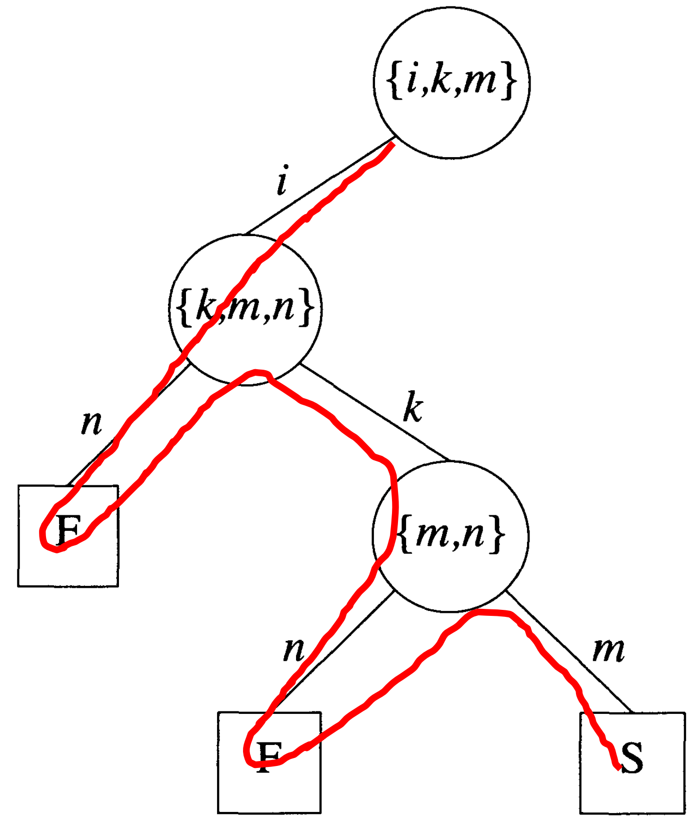


DF={n, m}



Example 6.6

Decisions	Implications	
	a=0	Activate the fault
	h=1	
	b=1	Unique D-drive through g
	c=1	
	g=D	
d=1		Propagate through i
	i=$\overline{D}$	
	d'=0	
j=1		Propagate through n
k=1		
l=1		
m=1		
	n=D	
	e'=0	
	e=1	
	k=$\overline{D}$	Contradiction
e=1		Propagate through k
	k=$\overline{D}$	
	e'=0	
	j=1	
l=1		Propagate through n
m=1		
	n=D	
	f'=0	
	f=1	
	m=$\overline{D}$	Contradiction
f=1		Propagate through m
	m=$\overline{D}$	
	f'=0	
	l=1	
	n=D	



Each node is a D-frontier.

PODEM – *Path Oriented Decision Making*

- Direct search process
  - Decisions only about PI assignments.
  - In *D*-algorithm, decisions on PIs are indirect.
- Value v_k to be justified on line k
  = *Objective* (k, v_k) to achieve via PI assignments.
- Backtracing of an objective
  - Maps a desired objective into a PI assignment
- Note that no values are assigned during backtracing

```
Backtrace (k, v_k)
/* map objective into PI assignment */
begin
    v = v_k
    while k is a gate output          // Recursive generating
        begin                         // objectives until it reaches PI
            i = inversion of k
            select an input (j) of k with value x
            v = v ⊕ i
            k = j
        end
    /* k is a PI */
    return (k, v)
end
```

Backtrace – An Example

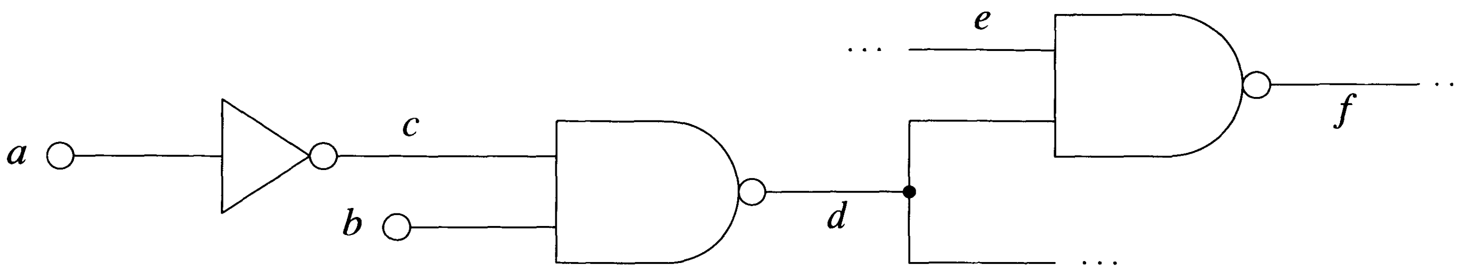


Figure 6.28

- Objective(*f*, 1)
- First Backtrace (*f*, 1) call:
  - Path (*f, d, b)* is tried with b=1 as PI assignment
  - But b=1 is not enough to achieve objective (*f*, 1)
- Second Backtrace (*f*, 1) call:
  - Path (*f, d, c, a)* is tried with a = 0
  - Now with a=0, we can achieve *objective (f, 1)*

Selecting an Objective

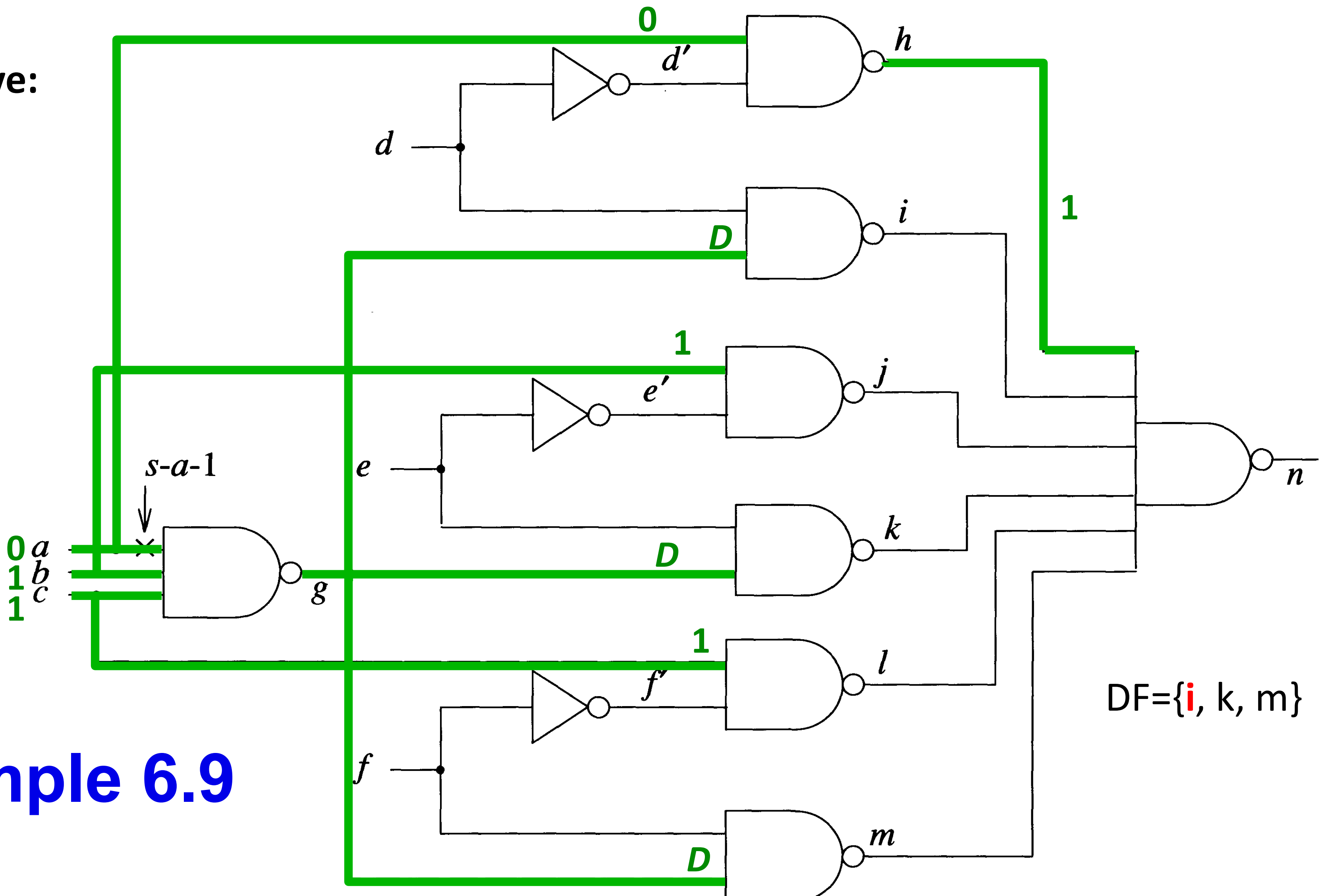
Objective()
begin
    /* the target fault is l *s-a-v* */
    if (the value of l is x) **then return** $(l,\overline{v})$
    select a gate (G) from the *D-frontier*
    select an input (j) of G with value x
    c = controlling value of G
    return $(j,\overline{c})$
end

Activate fault

Find the necessary inputs to propagate fault

```
PODEM()    // All lines are initialized to x
begin
    if (error at PO) then return SUCCESS
    if (test not possible) then return FAILURE
    (k,v_k) = Objective()
    (j,v_j) = Backtrace(k,v_k)  /* j is a PI */
    Imply (j,v_j)    // 5-value simulation with PI assignments
    if PODEM() = SUCCESS then return SUCCESS
    /* reverse decision */
    Imply (j,v̄_j)
    if PODEM() = SUCCESS then return SUCCESS
    Imply (j,x)
    return FAILURE   // D-frontier becomes empty
end
```

Objective:
d=1



DF={i, k, m}

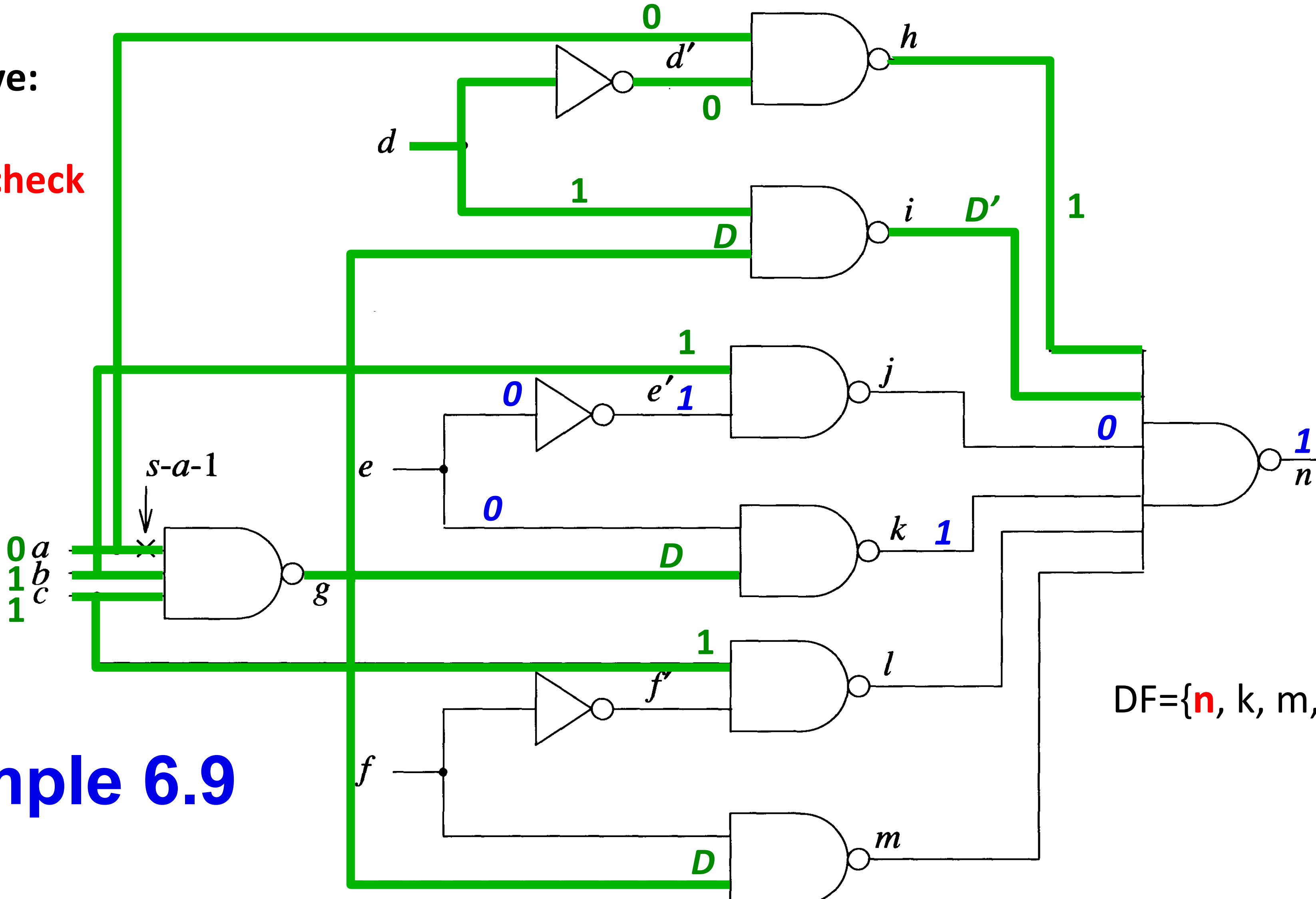
Example 6.9

Objective:

k=1

X-path check

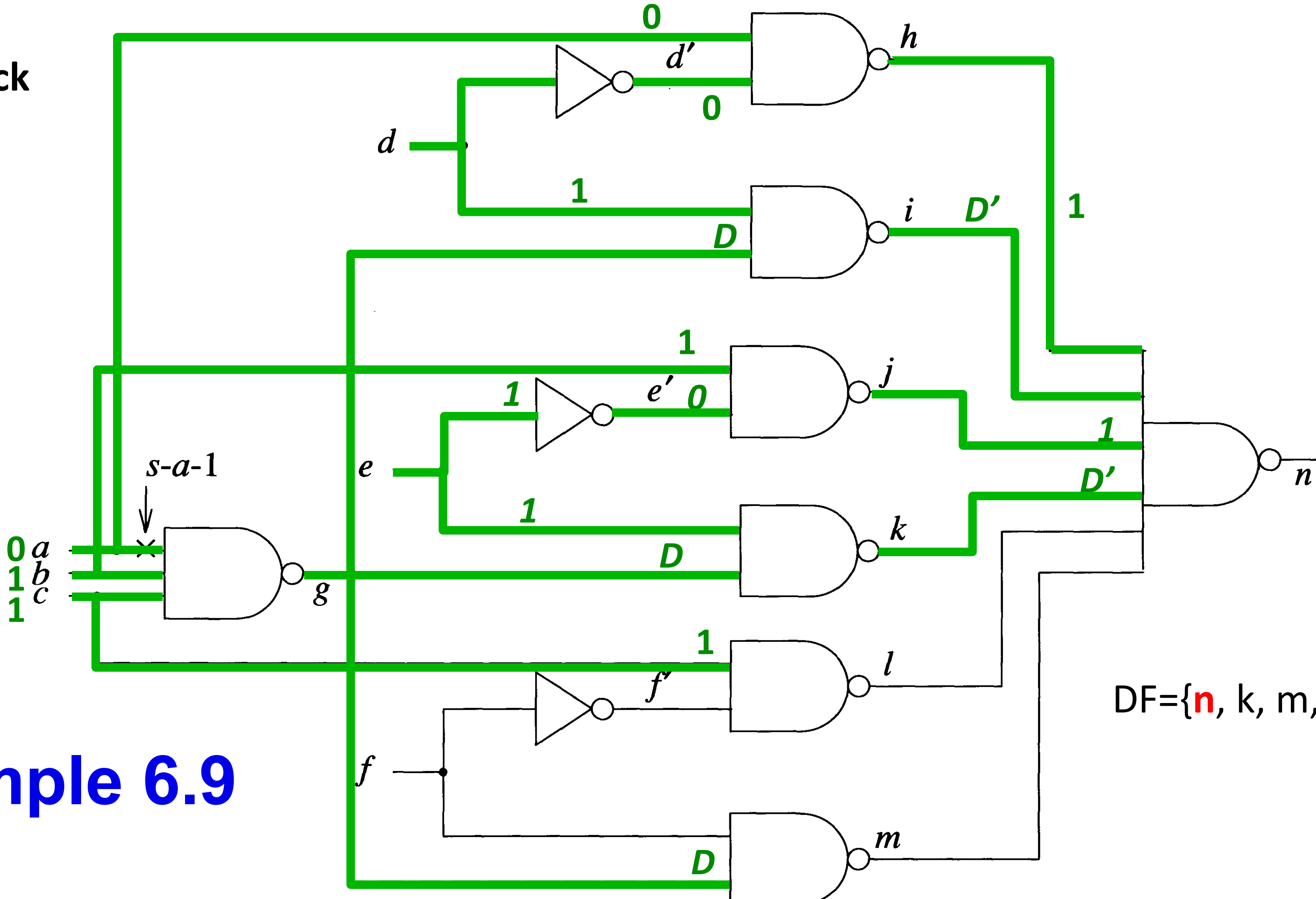
Failed!



DF={n, k, m,}

Example 6.9

Backtrack



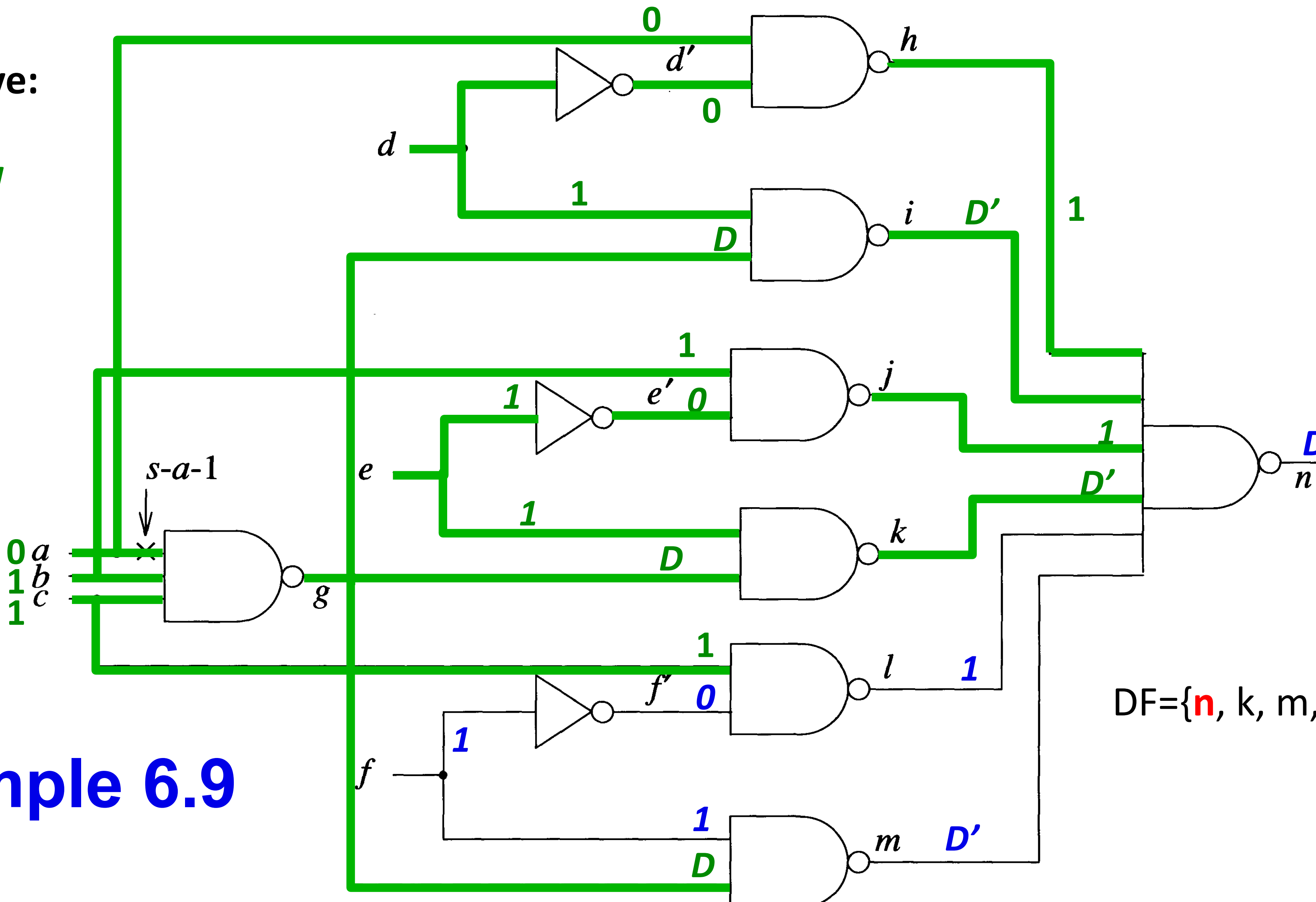
DF={**n**, k, m,}

Example 6.9

Objective:

l=1

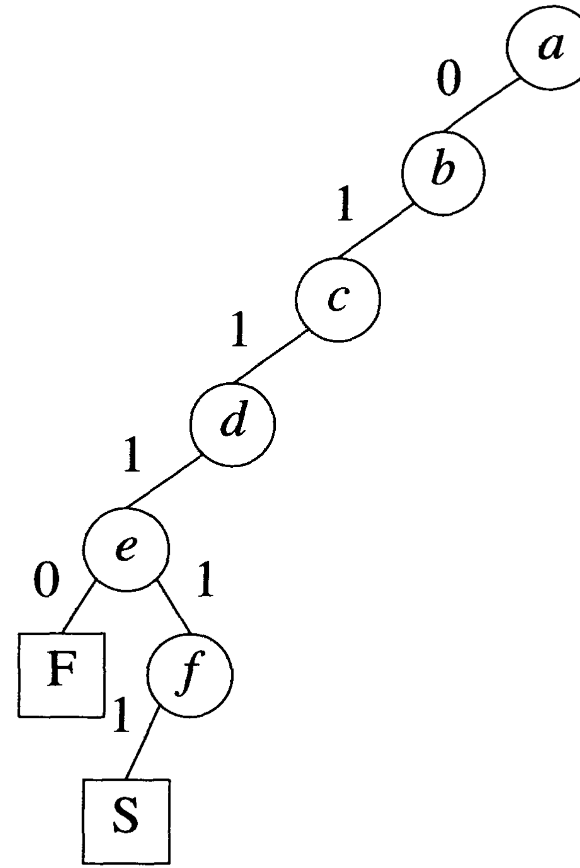
Success!



DF={n, k, m,}

Example 6.9

Objective	PI Assignment	Implications	*D-frontier*	
a=0	a=0	h=1	g	
b=1	b=1		g	
c=1	c=1	g=D	i,k,m	
d=1	d=1	d'=0 i=$\overline{D}$	k,m,n	
k=1	e=0	e'=1 j=0 k=1 n=1	m	x-path check fails
	e=1	e'=0 j=1 k=$\overline{D}$ n=x	m,n	reversal
l=1	f=1	f'=0 l=1 m=$\overline{D}$ n=D		



6.2.1.2 Algorithms

D-Algorithm vs PODEM

- PODEM does not need
  - Consistency check
  - *J*-frontier
  - Backward implication propagation
- Backtracking in PODEM is more simplified.
- Overall, PODEM is more efficient.

Selection Criteria

- Search process involves decisions
- Decisions on how to:
  - Select one of several unsolved problems: *fault propagation/line justification*.
  - Select one *possible* way to solved the selected problem: *several possible inputs to justify output 0 of AND gate.*

What are the selection criteria?

Some principles to speed up the search process.

Selection Criteria - Principles

- Among different unsolved problems, first attack the most difficult one
  - Thus avoid useless time spent in solving the easier problems when a harder one cannot be solved

- Among different solutions of a problem, first try the easiest one

- Difficulty is measured by *cost functions.*

Cost Functions

- *Controllability measures*
  - Related to the Line Justification problem
  - Relative difficulty of setting a line to a value

  Ex: select most difficult line-justification problem
- *Observability measures*
  - Related to the Error Propagation problem
  - Relative difficulty of propagating an error from a line to a PO

  Ex: select the gate from *D*-frontier whose input error is easiest to observe

Important: Must be relative measures and easy to compute.

Distance Based Cost Functions

- Any cost function should show that
  - PIs are the easiest to control
  - POs are the easiest to observe
- Therefore
  - Difficulty of controlling a line *increases* with its distance from PIs

    $\Rightarrow$ Line Level can be used as a controllability measure!
  - Difficulty of observing a line increases with its distance from POs

    $\Rightarrow$ Shortest distance of a line to PO can be used as a observability measure!

Main Drawback: Does not take into account the logic function

Controllability Measure C(l)

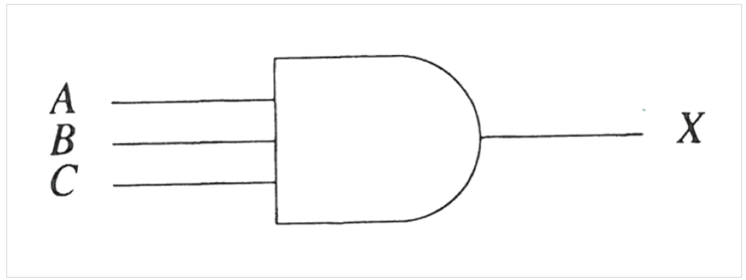
For every signal we want to compute:

- C0(l) = Relative difficulty of setting line l to 0
- C1(l) = Relative difficulty of setting line l to 1

Assume we know C0 and C1 costs of all inputs of the AND gate,

To set X to 0:

- C0(X) = *min* {C0(A), C0(B), C0(C) }



To set X to 1:

- C1(X) = C1(A) + C2(B) + C3(C)
- assuming A, B, C are independent (i.e., do not depend on common PIs)

We can develop similar cost functions for other gates. OR gate?

Controllability Measure Computation

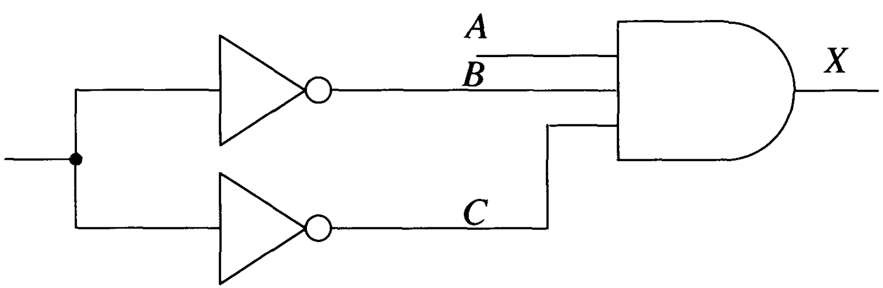
- Set C0 and C1 for every primary input to 1
- Compute C0's and C1' level by level
  - Cost are computed only after predecessor costs are known
- Costs can be computed in one forward traversal
- Linear in number of gates

Issues

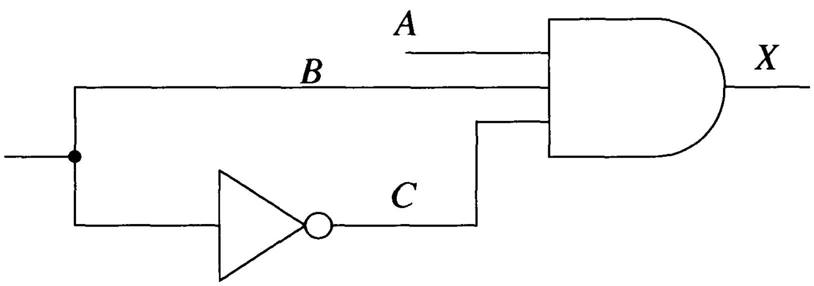
If inputs of a gate are not independent, it can lead to incorrect results

In (a) cost of controlling B and C is the same

In (b) B and C cannot be set to 1 simultaneously, so C1(X) should show that setting X=1 is impossible



(a)



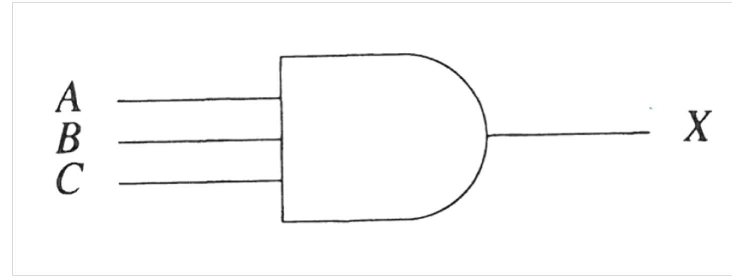
(b)

Observability Measure O(*l*)

Cost of observing the input A?

- We must set B and C to 1
- Propagate error from X to a PO

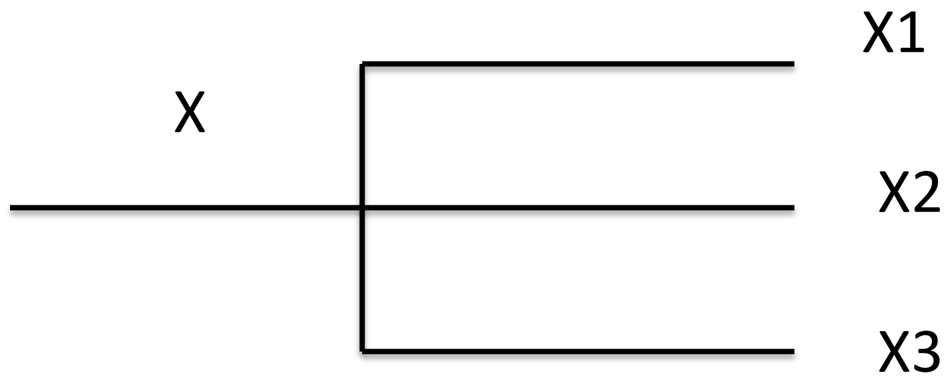
  O(A) = C1(B) + C1(C) + O(X) … Eq (3)



Assuming controlling B=1, C=1, and propagating Err(X) to PO are independent problems

What about OR gate?

Observability of a Stem X



$$O(X) = \min \{ O(X1), O(X2), O(X3)\} \quad \ldots \quad \text{Eq } (4)$$

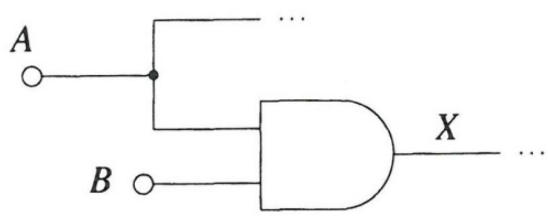
Assuming single path propagation is possible

Observability Measure Computation

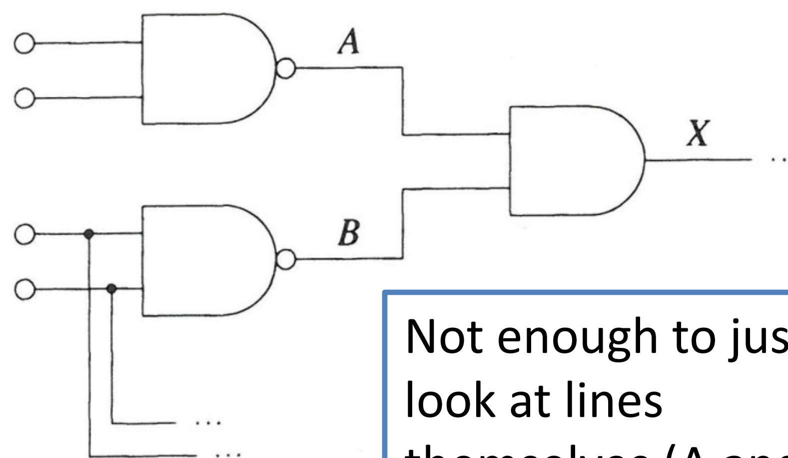
- Set observability cost of every PO to 0
- Compute observabilities level by level backward manner using eq 3 and 4.
  - Cost are computed only after successor costs are known
- Costs can be computed in one backward traversal
- Linear in number of lines
- Assume controllability measure is known.

Fanout-Based Cost Functions

- Reconvergent fanout makes TG difficult.
- A line with fanout has high potential causing conflict.



Setting B = 0 is better than A = 0



Not enough to just look at lines themselves (A and B)!!

Fanout-Based Controllability Measure

- C(l) depends on
  - Fanout count of l
  - Fanout count of predecessors of l

$$C(l) = \sum_i C(i) + f_l - 1 \qquad (6.5)$$

Where f_l is the fanout count of l

A line *l* with *C(l) = 0* means it does not depend on any fanout lines.

Example

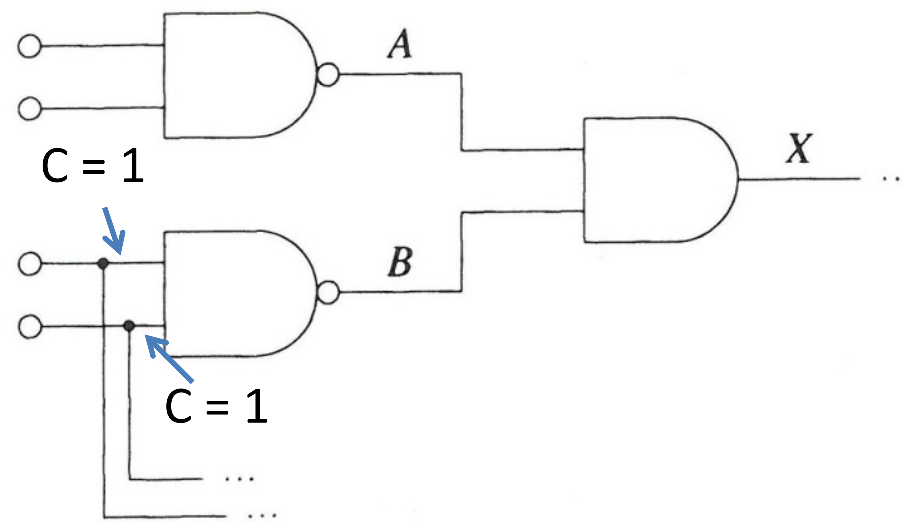
$$C(l) = \sum_i C(i) + f_l - 1$$

C(A) = 0

C(B) = 2

C(X) = 2

Therefore, select A=0 to justify X=0.



C0(l) and C1(l) – More Accurate Cost Func.

- Eq (6.5) does not distinguish between setting a line to 0 and to 1

  For the AND gate we have:

  $$C0(l) = min\,\{C0(i)\} + f_l - 1$$

  and

  $$C1(l) = \sum_i C1(i) + f_l - 1$$

What about OR gate?

Example

$$C0(l) = min \{C0(i)\} + f_l - 1$$

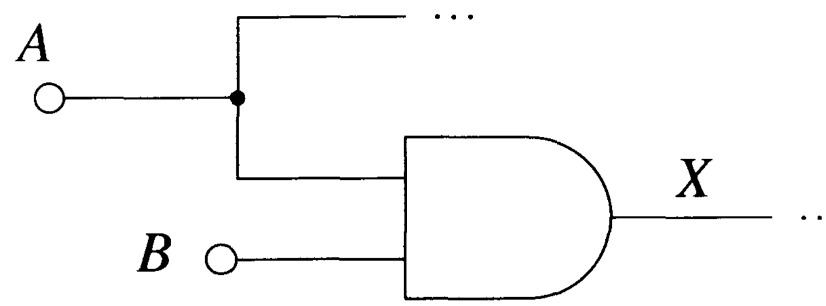
$$C(l) = \sum_i C(i) + f_l - 1$$

C0(A) = C1(A) = 1

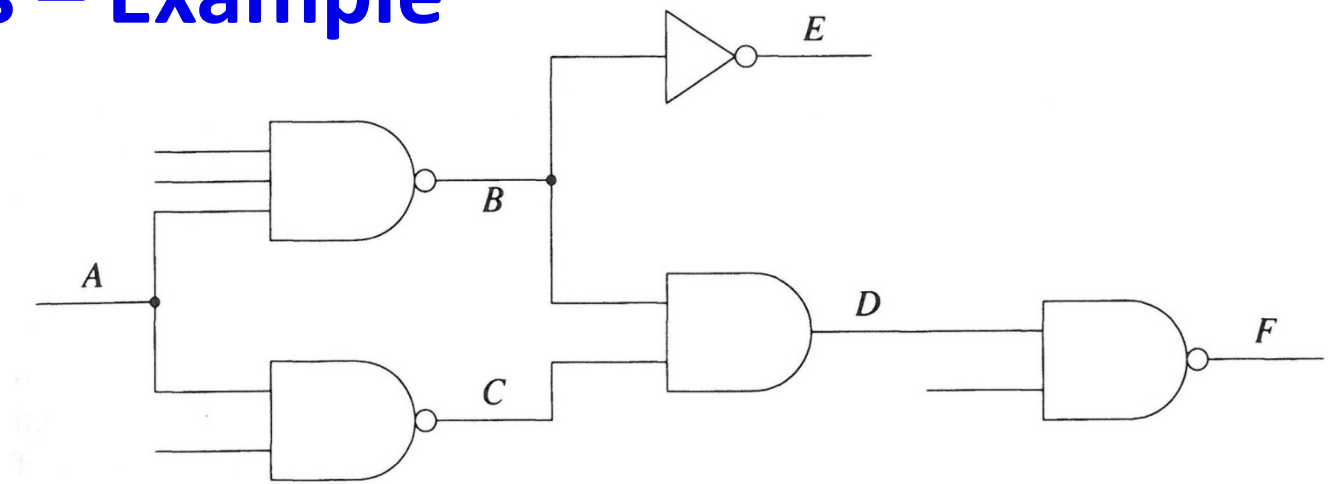
C0(B) = C1(B) = 0

C0(X) = 0,

C1(X) = 1.



Side Effects – Example

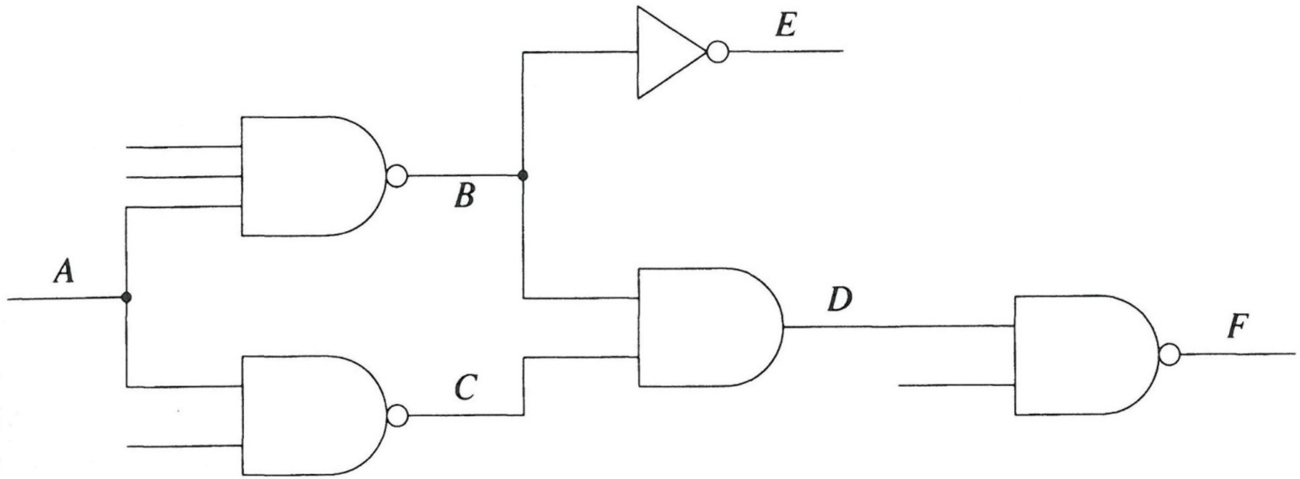


- C0(A) and C1(A) both have corrective terms =1
- A = 0 has greater potential of conflicts than A = 1
  - A = 0 results in B, C, D, E being set to binary values
  - Less x-paths for error propagation.

Side Effects Cost Function

- Side-Effects Cost Functions: CS0(l) and CS1(l) to account for relative potential for conflicts caused by setting l to 0 and 1
- Computed by simulating $l = v$ ($v \in \{0, 1\}$) in a circuit initialized with all-x state, and then
  - A gate whose output is set to a binary value increases cost by 1
  - A gate with n inputs whose output remains at x but which has m inputs set to a binary value, increases the cost by m/n

Side Effect Function – Example



- CS0(A) =4(1/2)
- CS1(A) = (1/3) + (1/2) = 5/6

Cost Functions with Side-Effects

$$C0(l) = min\ \{C0(i)\} + CS0(l)$$

$$C1(l) = \sum_{i} C1(i) + CS1(l)$$

- Require circuit simulation after assigning *l* to 0 or 1
  - Cause additional complexity

Cost Functions: Summary

- Complexity of cost function computation must be low.

- Cost functions are based heuristics.

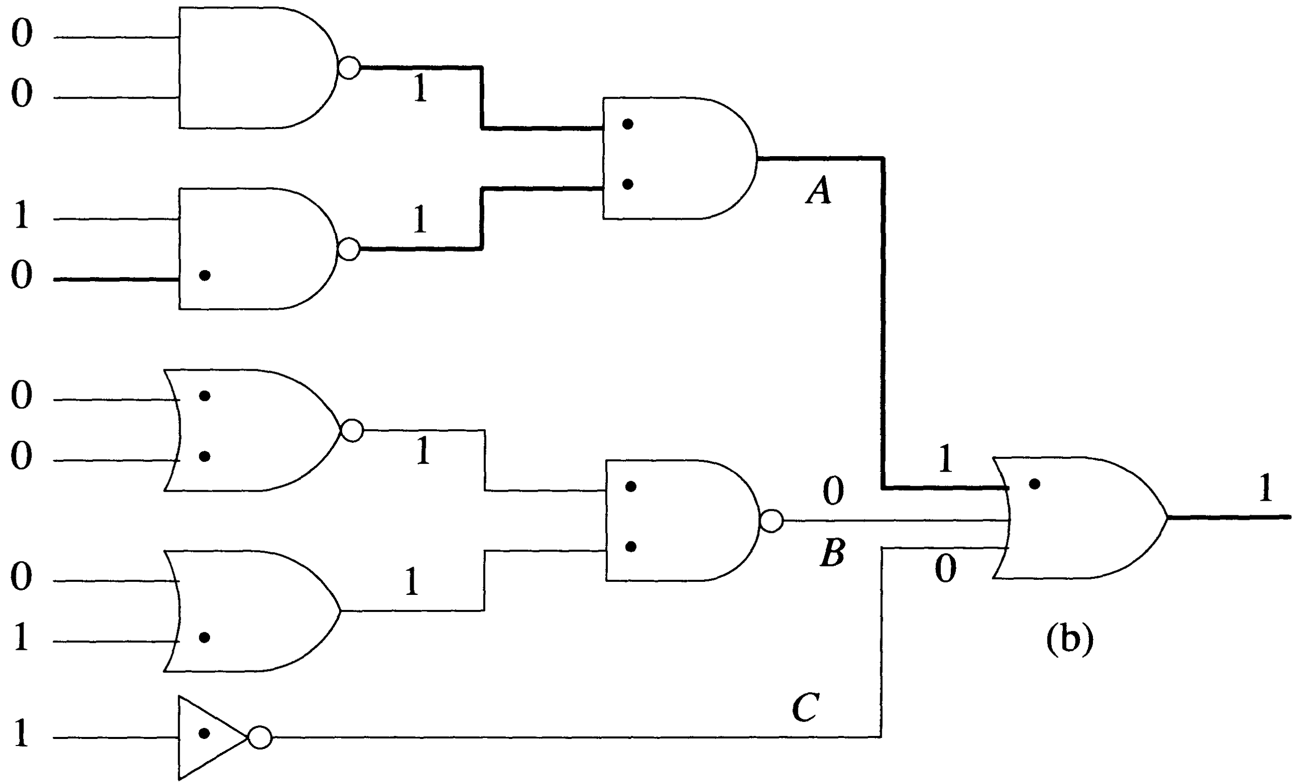
- Dynamic cost functions may lead to better performance.

Backup

Fault Independent ATG

- Fault-oriented algorithm targets a given fault and generate a test vector
- Fault-independent algorithm's goal:
  - Derive a set of test that detect a large set of SSFs w/o targeting individual faults
- CPT -- Half of the SSFs on a path critical in a test *t* are detected by *t*
  - ⇒ Generate tests that produce **long** critical paths
  - ⇒ Critical path TG algorithm

Critical Paths – Basic Concept



The input vector detects output s-a-0 fault and other faults on the critical path

Critical-path TG Algorithm

Basic Steps

1. Select a PO and assign it a critical 0-value or 1-value (Recall that a PO is always critical)
2. Recursively justify the PO value, trying to justify any critical value on a gate output by critical values on the gate inputs

Line Justification – 3 Input AND gate

By Primitive Cubes

A	B	C	Z
1	1	1	1
0	x	x	0
x	0	x	0
x	x	0	0

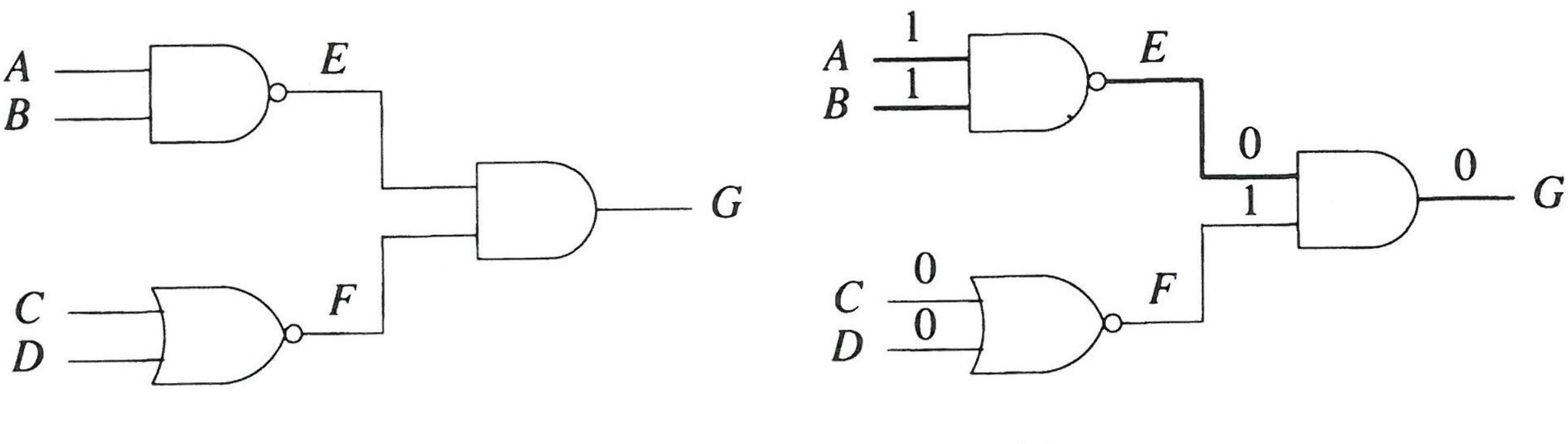
(a)

By Critical Cubes

A	B	C	Z
1	1	1	1
0	1	1	0
1	0	1	0
1	1	0	0

(b)

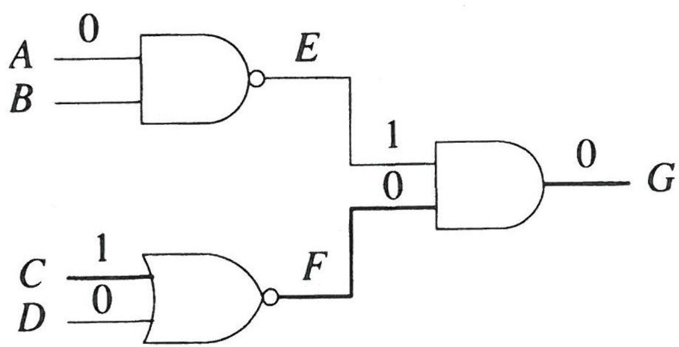
Critical-path TG - Example



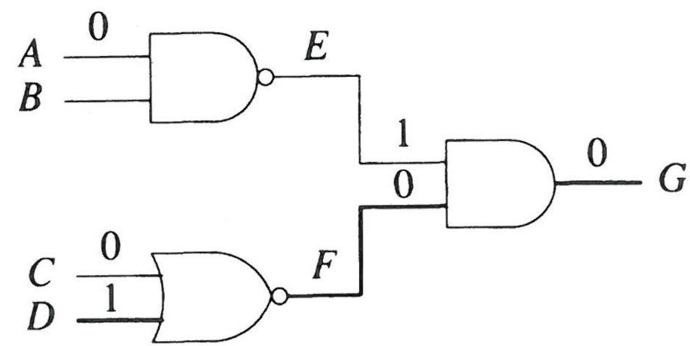
(a) (b)

What SSFs can be detected by this input vector?

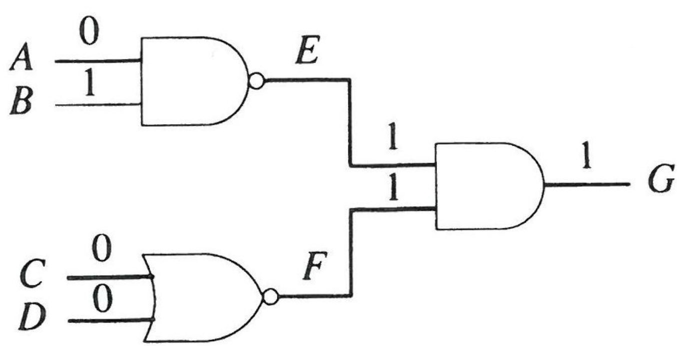
Critical-path TG – Example …contd.

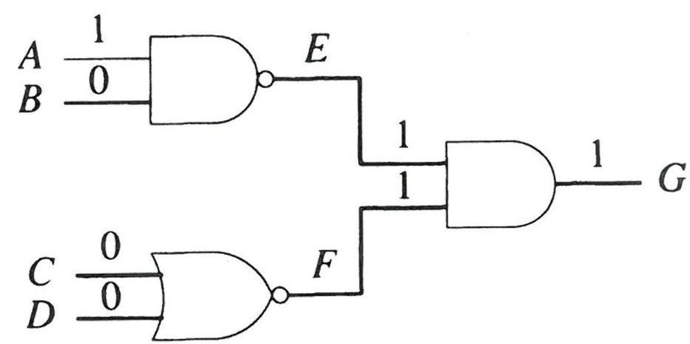


(c)



(d)





```
CPTGFF()
begin
    while (Critical ≠ ∅)
        begin
            remove one entry (l,val) from Critical
            set l to val
            mark l as critical
            if l is a gate output then
                begin
                    c = controlling value of l
                    i = inversion of l
                    inval = val ⊕ i
                    if (inval = c̄)
                        then for every input j of l
                            add (j,c̄) to Critical
                        else
                            begin
                                for every input j of l
                                    begin
                                        add (j,c) to Critical
                                        for every input k of l other than j
                                            Justify (k,c̄)
                                        CPTGFF()
                                    end
                                return
                            end
                end
        end
    /* Critical = ∅ */
    record new test
    return
end
```

- Critical Path TG Fanout Free
- To generate complete test set for a FF circuit whose PO is Z,

    add (Z, 0) to *Critical*
    CPTGFF()
    add (Z, 1) to *Critical*
    CPTGFF()

Decision Tree

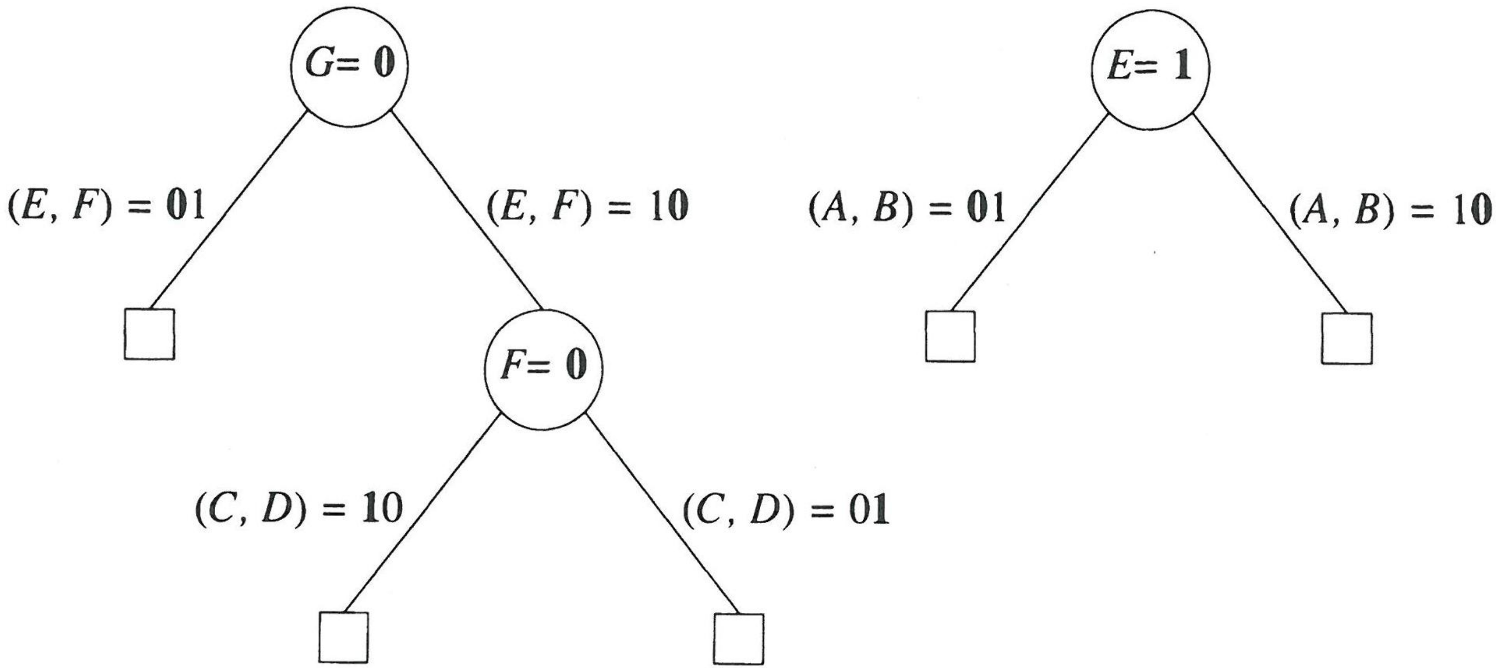


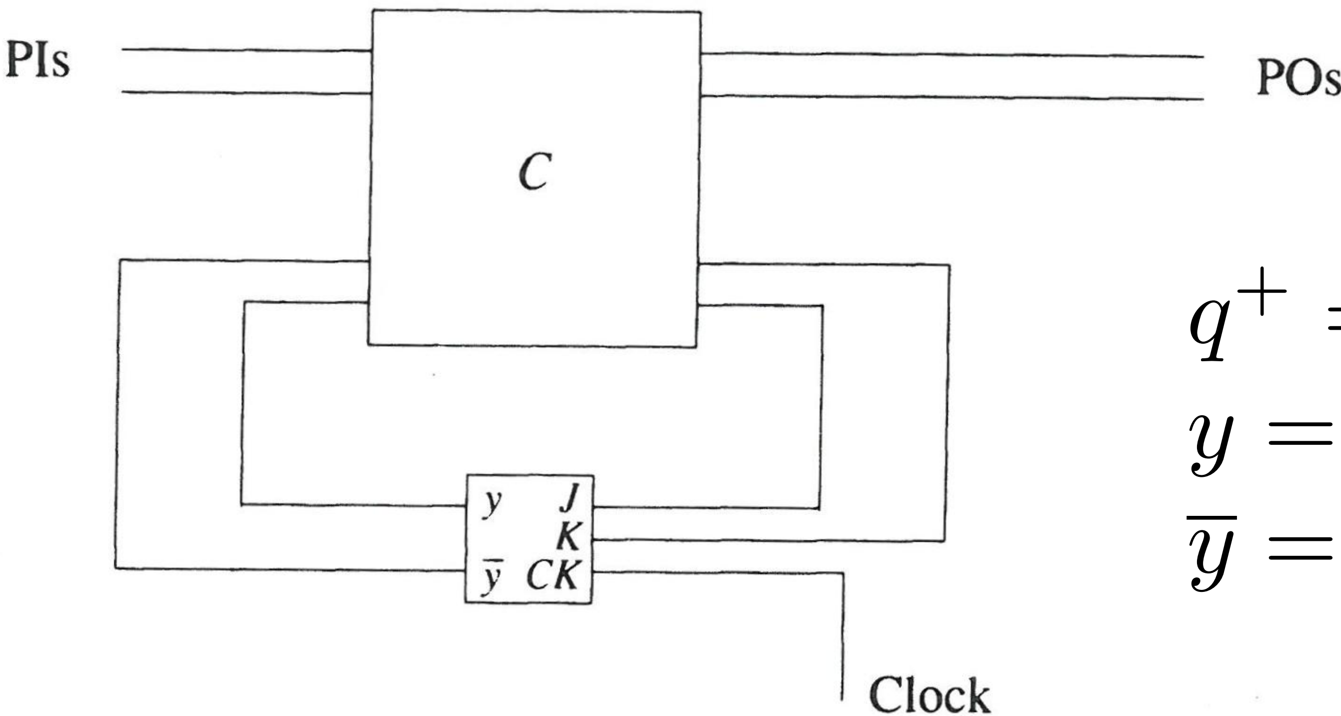
Figure 6.47 Decision trees for Example 6.12

The number of terminal nodes equals the number of tests generated.

ATG for SSFs in Sequential Circuits

- TG using Iterative Array Model
  - Extends TG methods of combinational circuits to sequential circuits
- Transform Synchronous sequential circuit into an iterative combinational array.
  - Unroll the circuit for *k* times.
  - One cell in the array -> *time frame*
- Assume all FFs are driven by a fault-free clock line.
- An input vector for the array is a sequence of *k* input vectors for the synchronous circuit.

Synchronous State m/c model

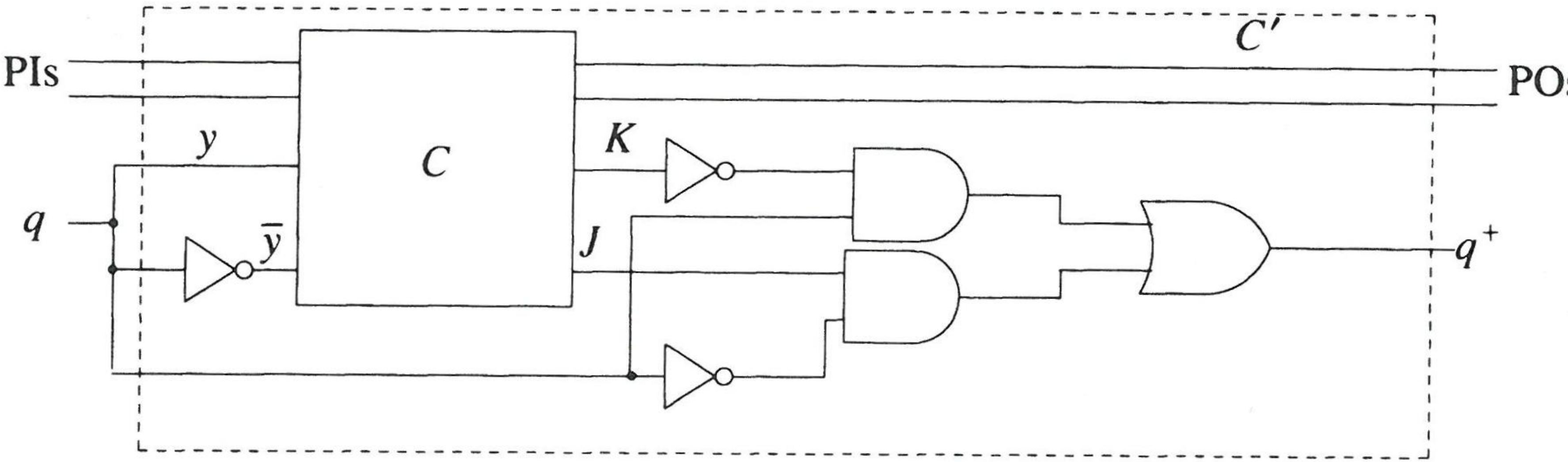


$$q^{+} = J\overline{q} + \overline{K}q$$

$$y = q$$

$$\overline{y} = \overline{q}$$

Model for one time frame



- Since the circuit is same for every frame, we do not have to generate *n* copies
- However, we should separately maintain signal values of each time frame

Some observations

- C' is a combinational circuit, so any combinational TG algorithm (D, PODEM, CPTG, etc.) can be applied
- A test vector *t* for *C'*, may specify PI and q values
  - q values must be justified in previous timeframe
- *t* may not propagate an error to a PO but to a q+ variable
  - Error must be propagated to next time frame
- In general, search process
  - May span multiple time frames
  - Going backward and forward in time

Fault Propagation

- Target fault can be present in every time frame!
  - Error value (D or D') may propagate onto the faulty line itself

Value propagated onto line l	Fault of line l	Resulting value of line l
D	s-a-0	D
D	s-a-1	1
$\overline{D}$	s-a-0	0
$\overline{D}$	s-a-1	$\overline{D}$

Figure 6.73 Result of a fault effect propagating to a faulty line

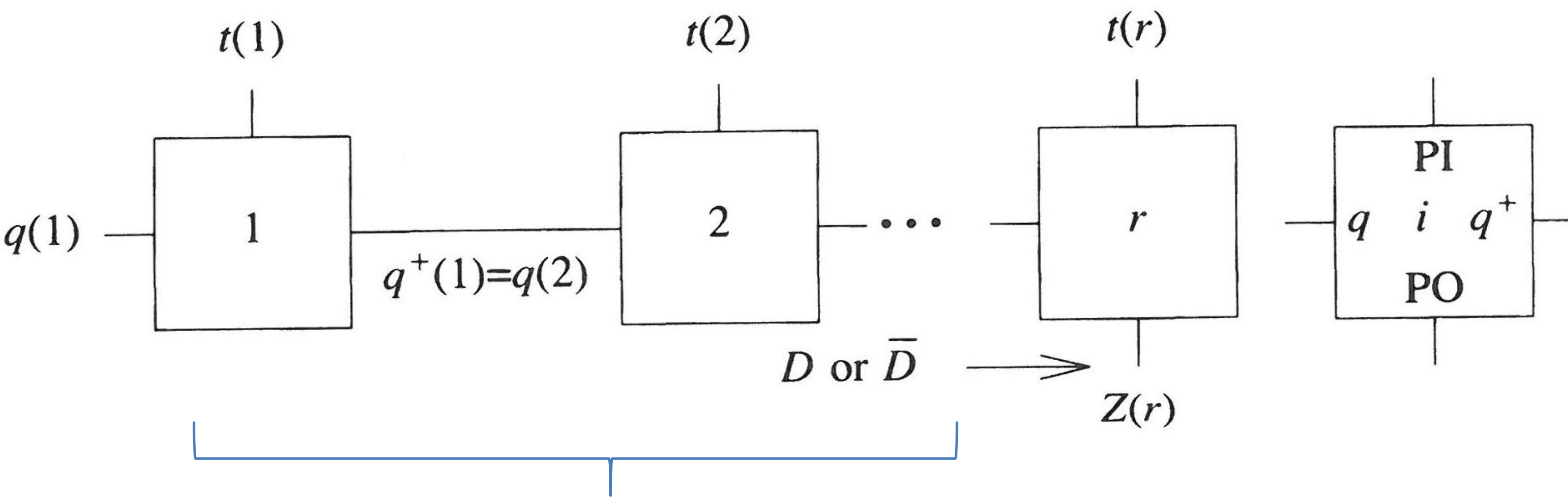
TG from a Known Initial State

r=1
repeat
    begin
        build model with r time frames
        ignore the POs in the first $r-1$ frames
        ignore the q^+ outputs in the last frame
        $q(1)$ = given initial state
        if (test generation is successful) **then return** SUCCESS
        /* no solution with r frames */
        $r = r + 1$
    end
until $r = f_{max}$
return FAILURE

Once circuit is unrolled, we can use any of the test generation algorithm we studied for combinational circuits, such as D-alg(), PODEM, etc.

Maximum Unroll factor

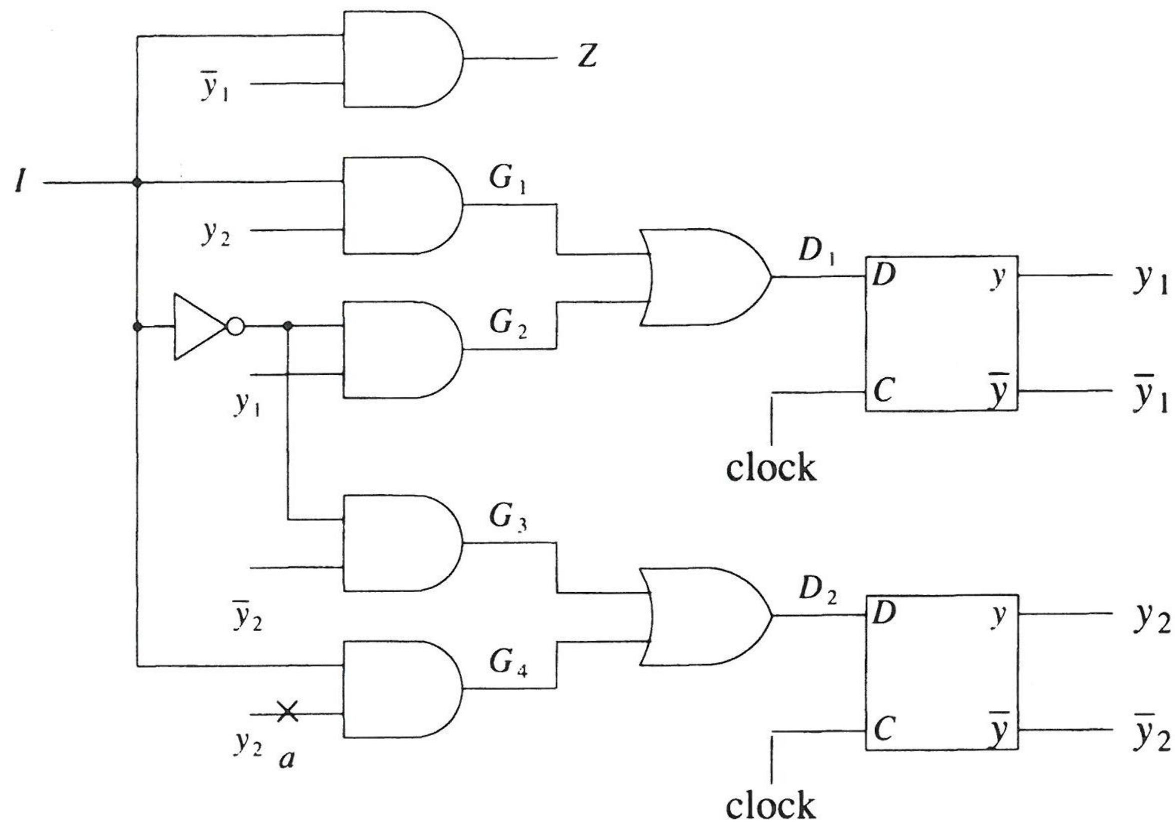
Iterative Array Model



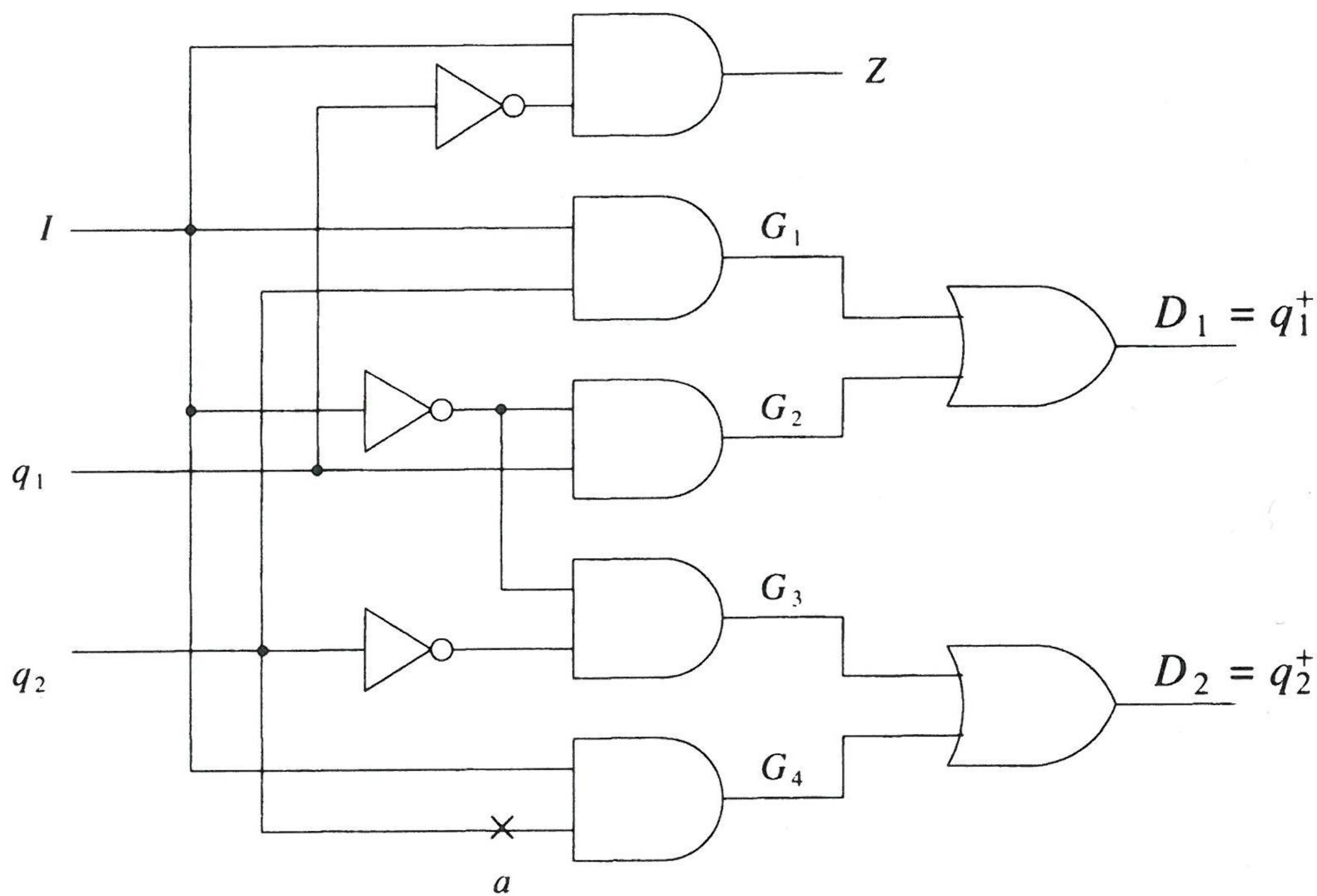
Ignore POs in
r-1 slices

Example

- Assume $q_1 = q_2 = 0$

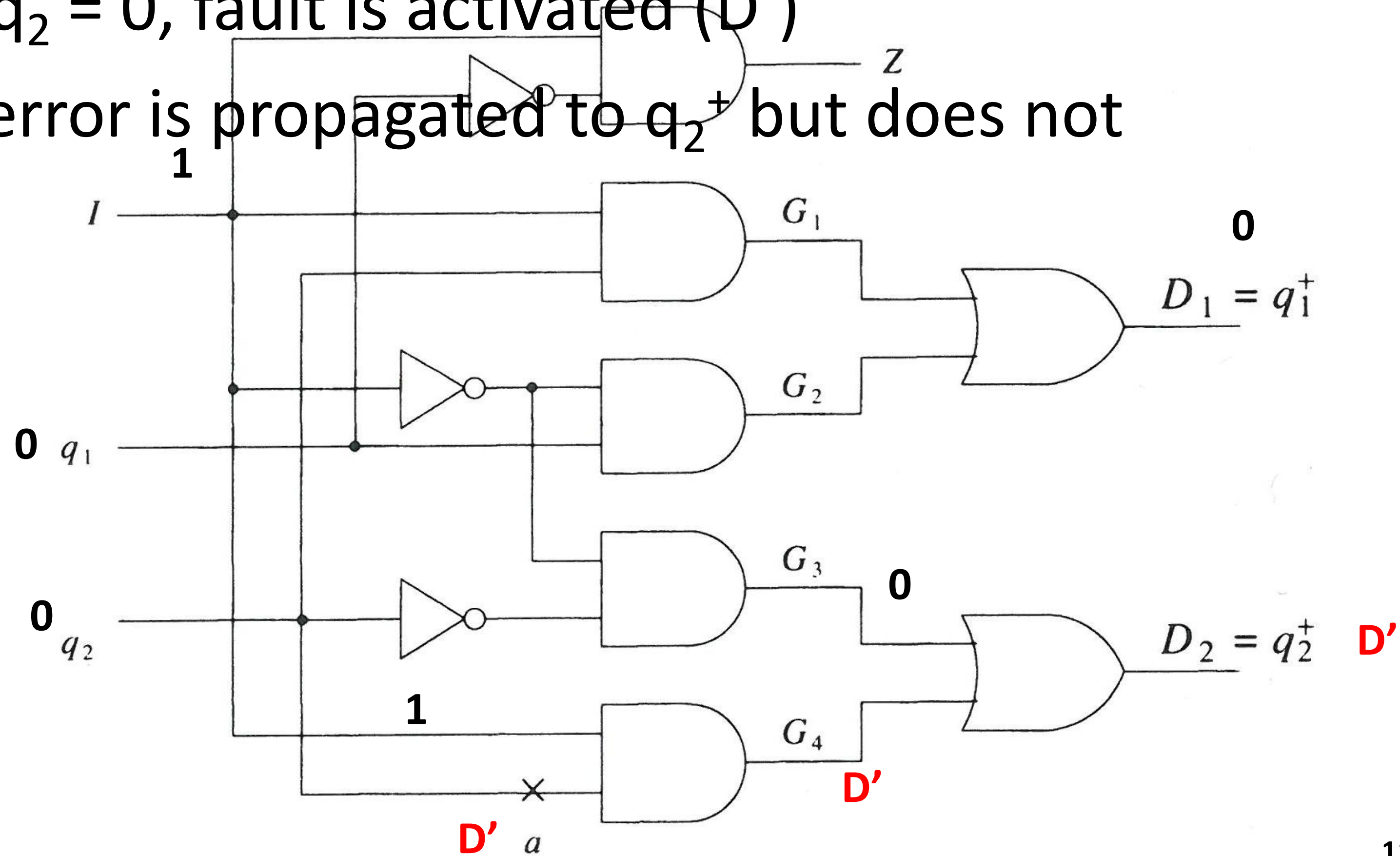


Time Frame



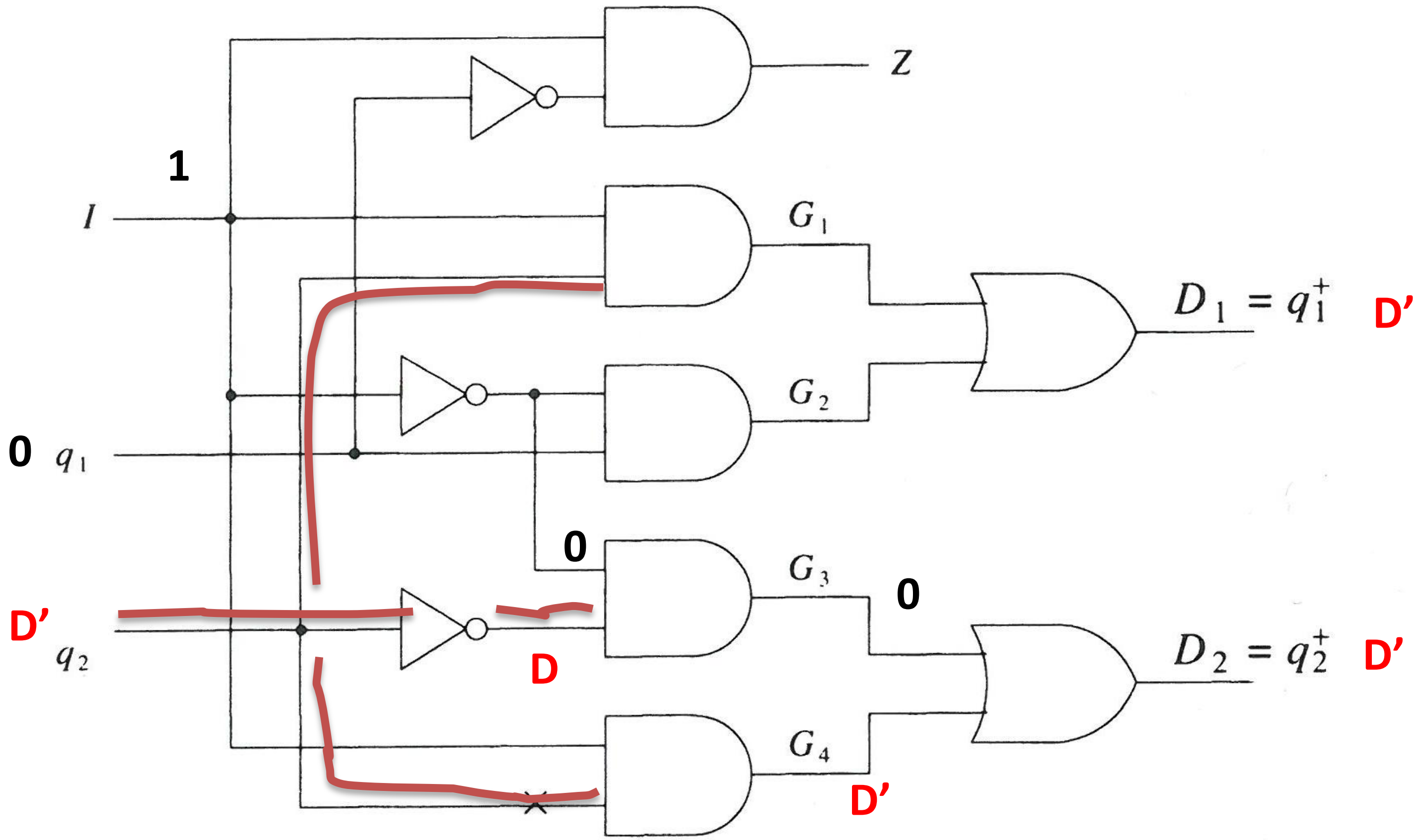
Time Frame 1

- With $q_1 = q_2 = 0$, fault is activated (D')
- With I=1, error is propagated to q_2^+ but does not reach Z

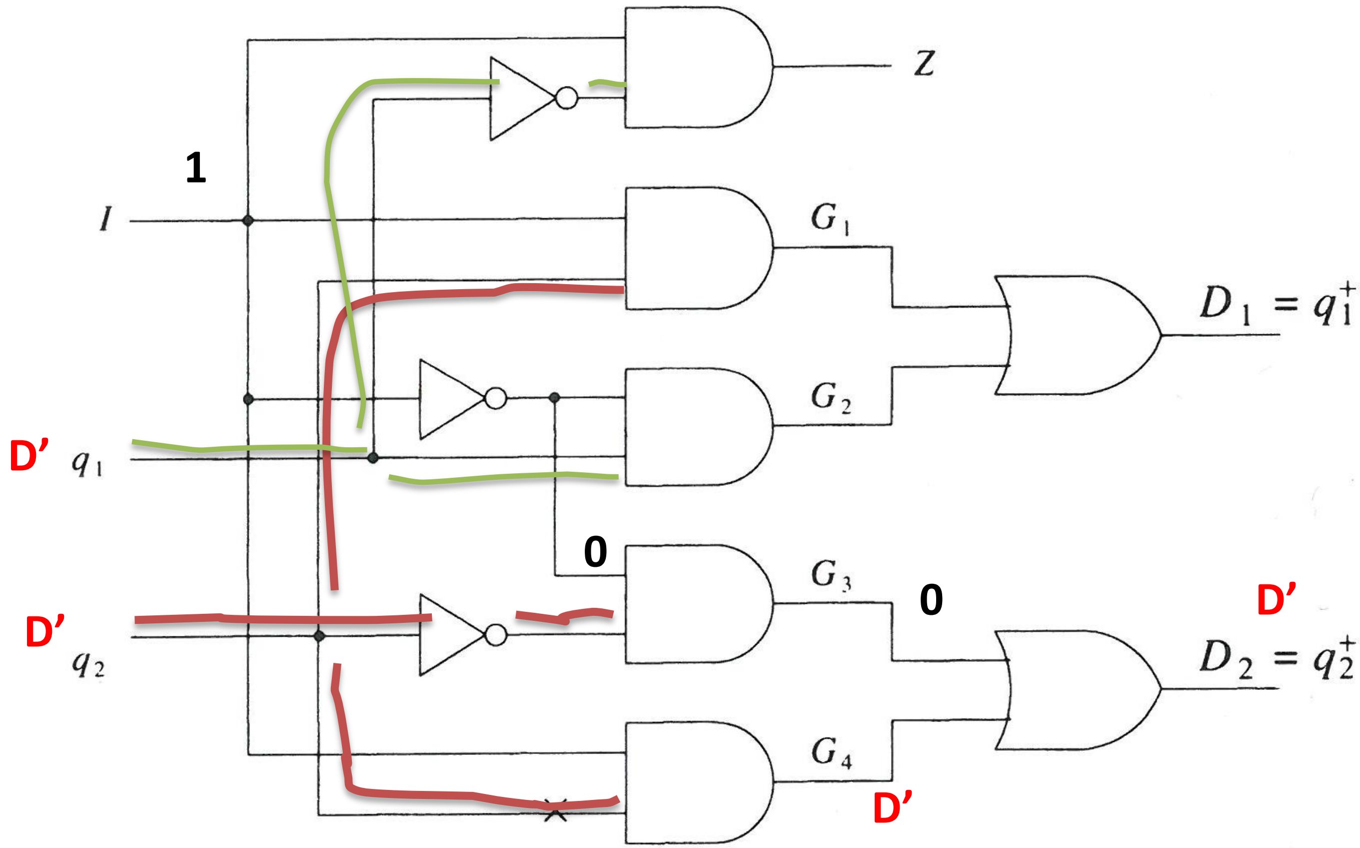


Time Frame 2

- D-frontier = {G1, G3, G4}
- If G1 or G4 is chosen, then I = 1 gives q_1^+ = D' and q_2^+ =D'
- If G3 is selected with I = 0 gives q_1^+ = 0 and q_2^+ =D



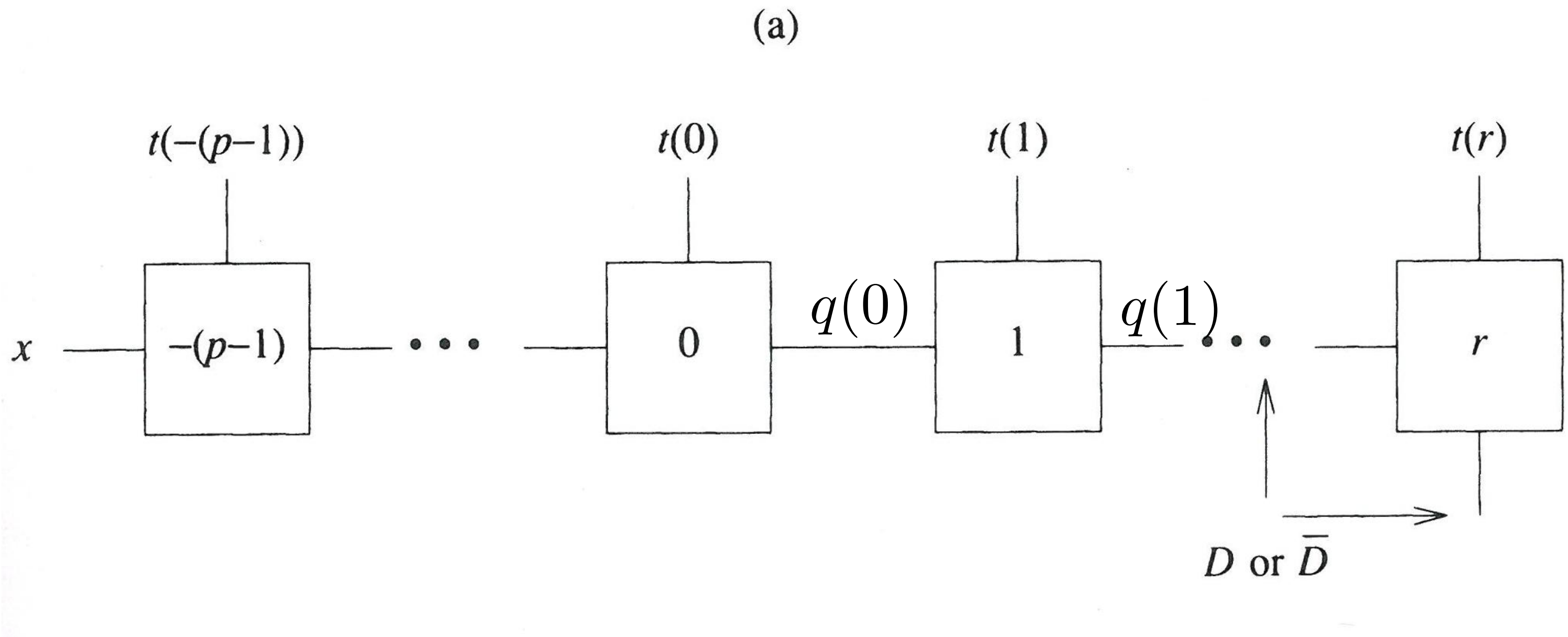
- D-frontier = {Z, G1, G2, G3, G4}
- With I=1, we get Z = D, error propagated to a PO!
- Desired test sequence is I = (1, 1, 1)



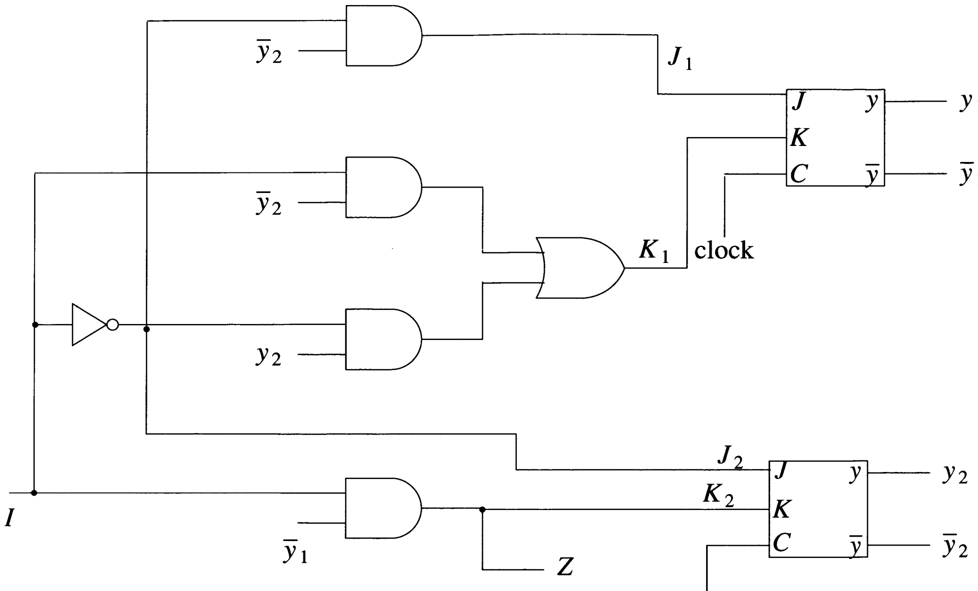
Generation of Self-initializing Test Sequences

```
r = 1
p = 0
repeat
    begin
        build model with p + r time frames
        ignore the POs in the first p + r − 1 frames
        ignore the q+ outputs in the last frame
        if (test generation is successful and every q input in the first frame has
            value x) then return SUCCESS
        increment r or p
    end
until (r+p=f_max)
return FAILURE
```

Generation of Self-initializing Test Sequences

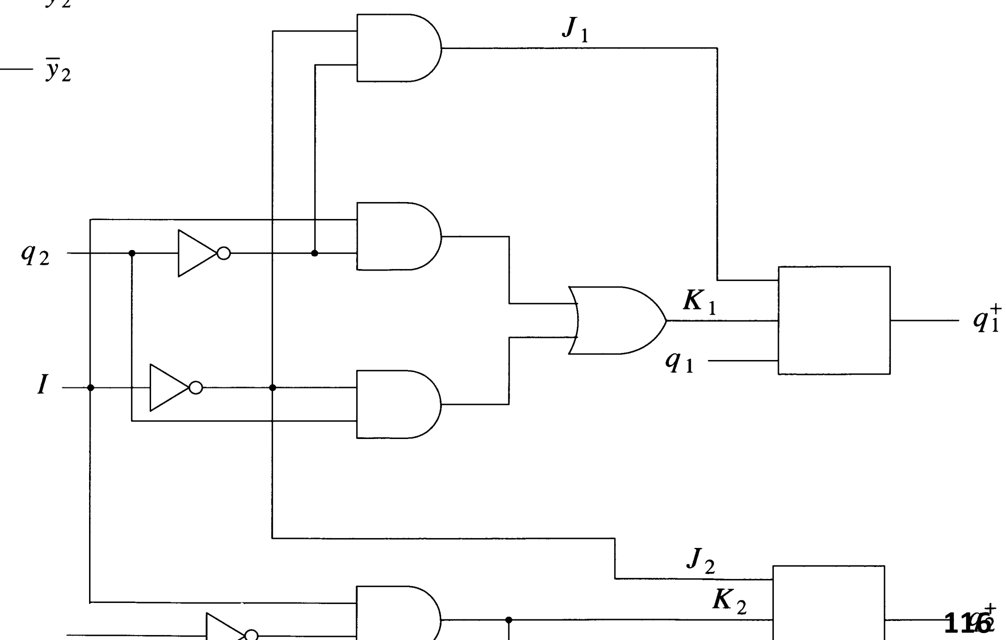


1. Activate fault in frame 1, and propagate it to PO using r frames.
2. If q(0) is not all x, justify q(0) by backward propagation of p frames.

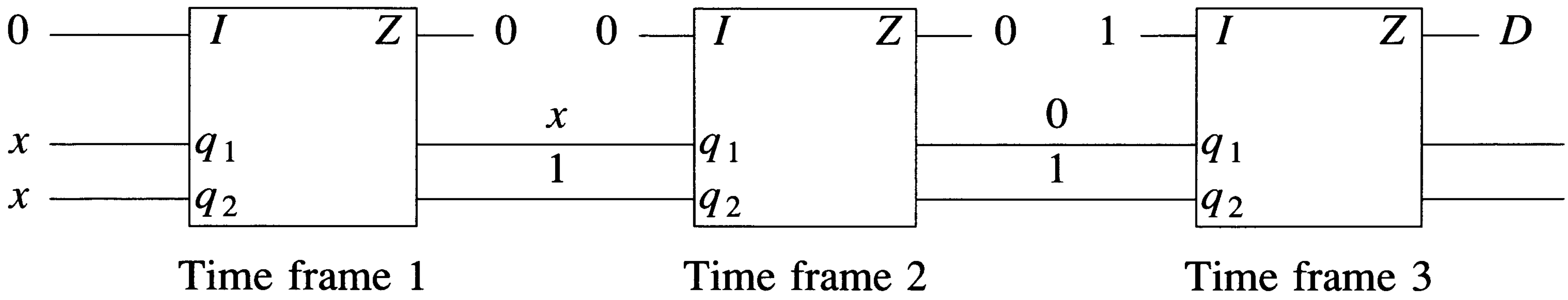


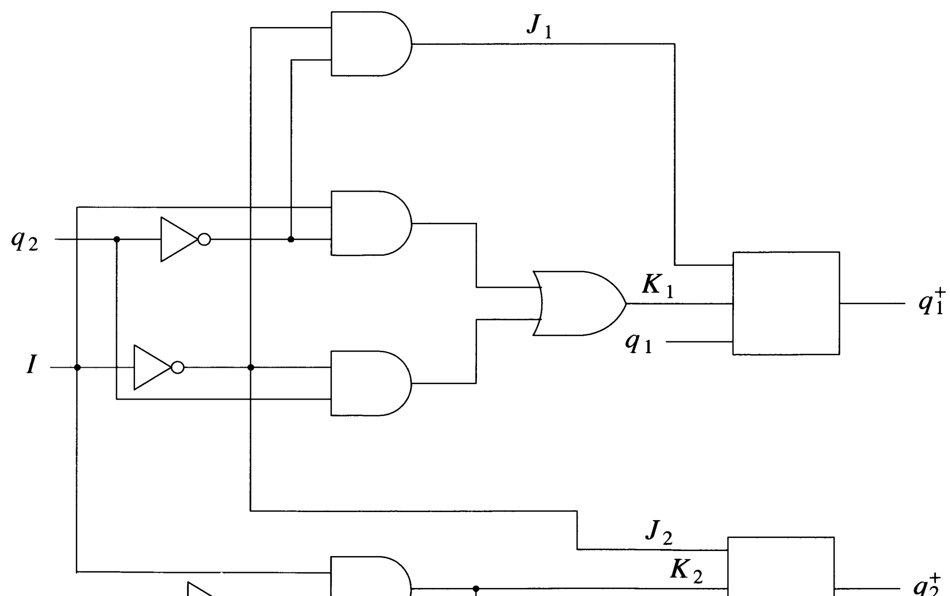
(a)

$$q^+ = J\overline{q} + \overline{K}q$$



Example: Iterative Array: Detect Z s-a-0





$$q^+ = J\overline{q} + \overline{K}q$$

Example: Iterative Array: Detect Z s-a-0