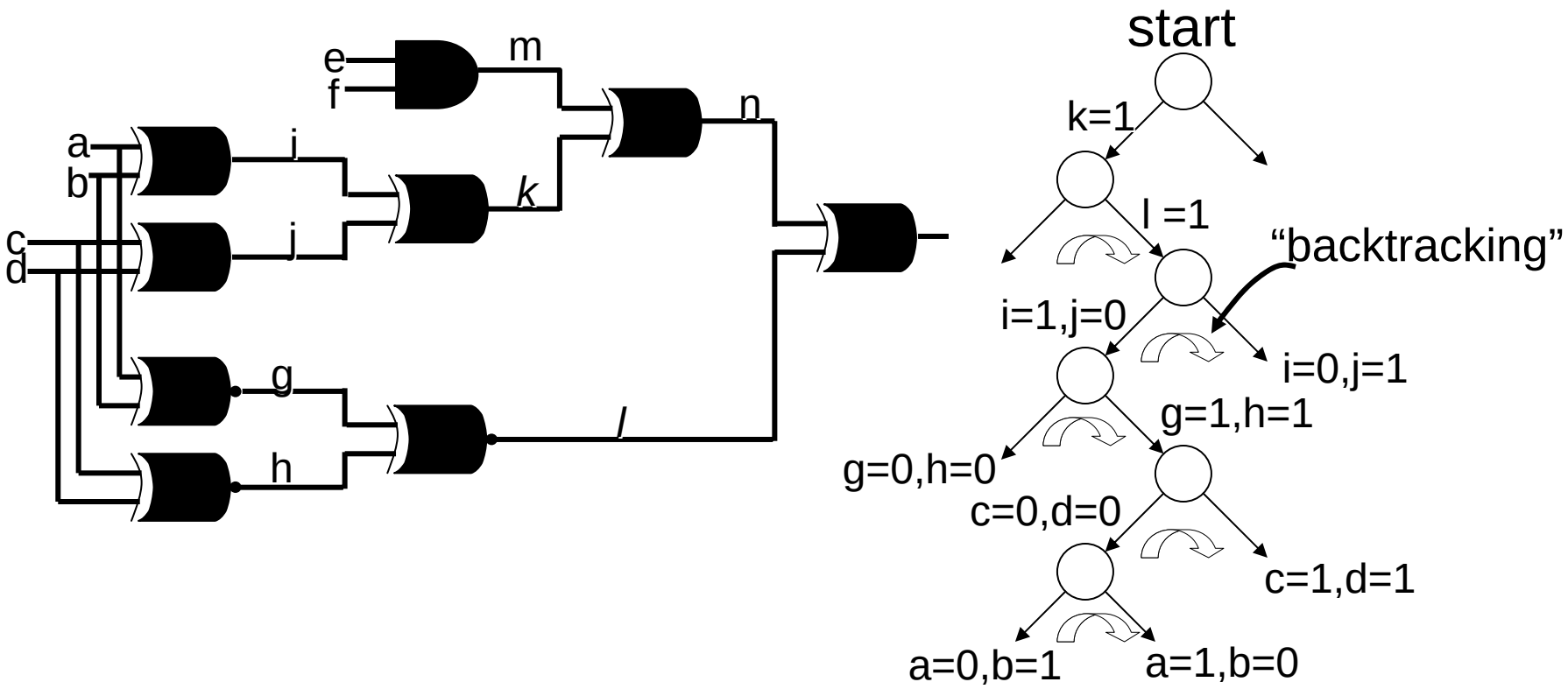
D-Algorithm: Decision Tree



Search performed level by level from PO’s to PI’s

BACKWARDS

D Algorithm – Line Justification

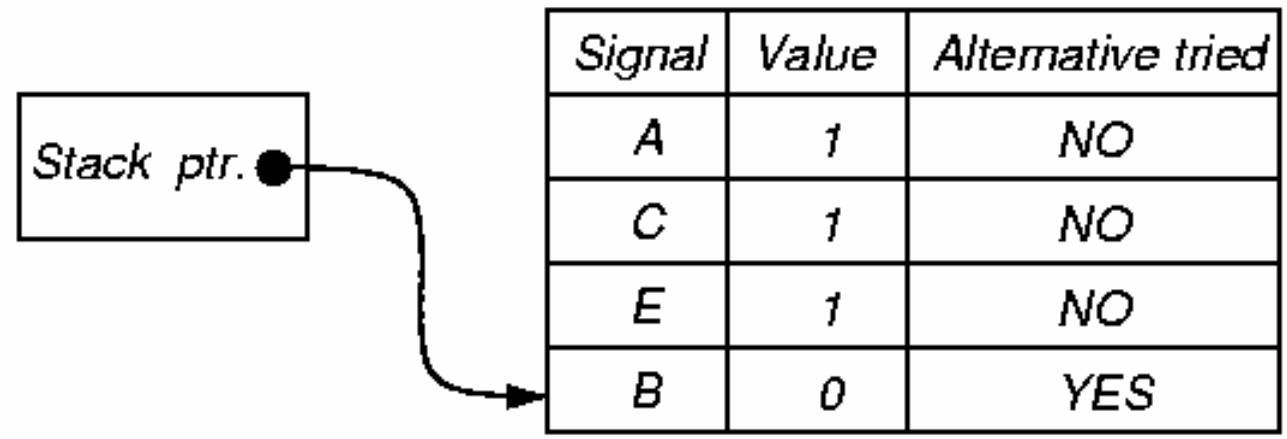
- Find input assignment for value *v* on line *g*
  - Propagating signals through gates
- Primitive Cube (PC) of gate – implicant of *f* or *f'*

	AND			NAND			OR			NOR		
	A	B	f	A	B	f	A	B	f	A	B	f
Implication	1	1	1	1	1	0	0	0	0	0	0	1
Decision (choice)	0	-	0	0	-	1	1	-	1	1	-	0
	-	0	0	-	0	1	-	1	1	-	1	0

Decisions might be reversed upon conflicts– keep track

Implication Stack

- Push-down stack. Records:
  - Each signal set in circuit by ATPG
  - Whether alternate signal value already tried
  - Portion of binary search tree already searched

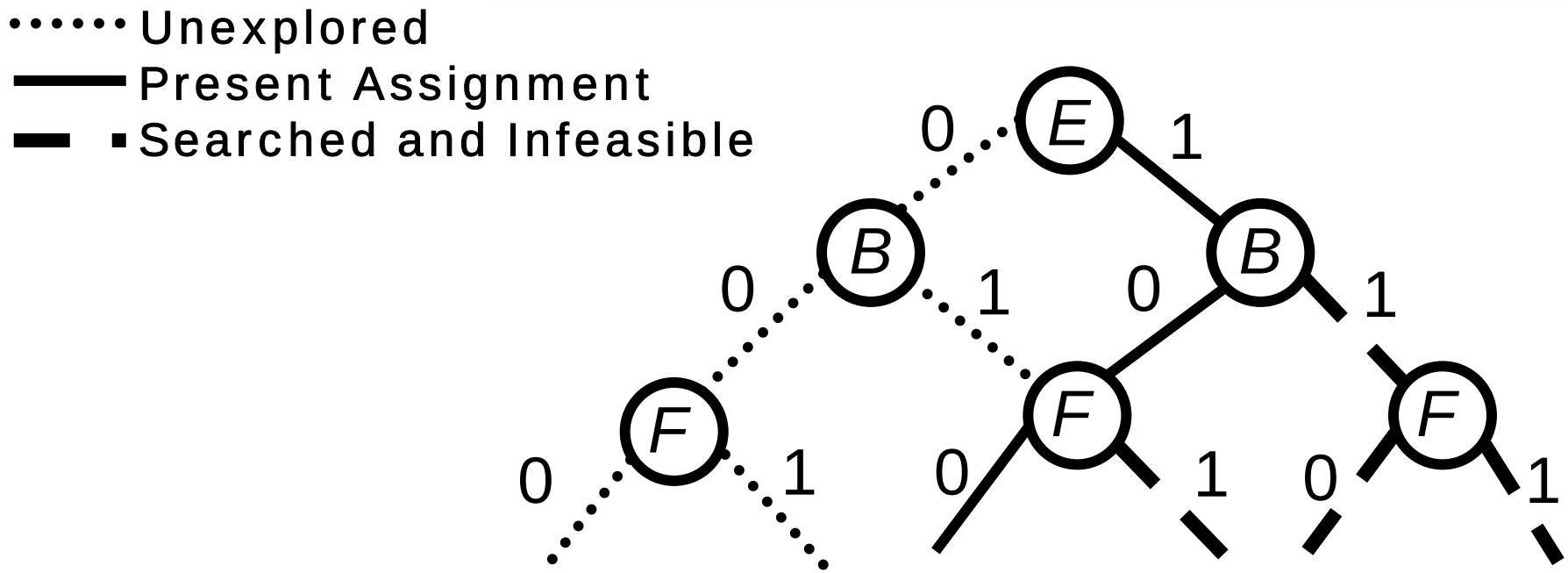


Signal	Value	Alternative tried
A	1	NO
C	1	NO
E	1	NO
B	0	YES

Implication Stack after Backtrack

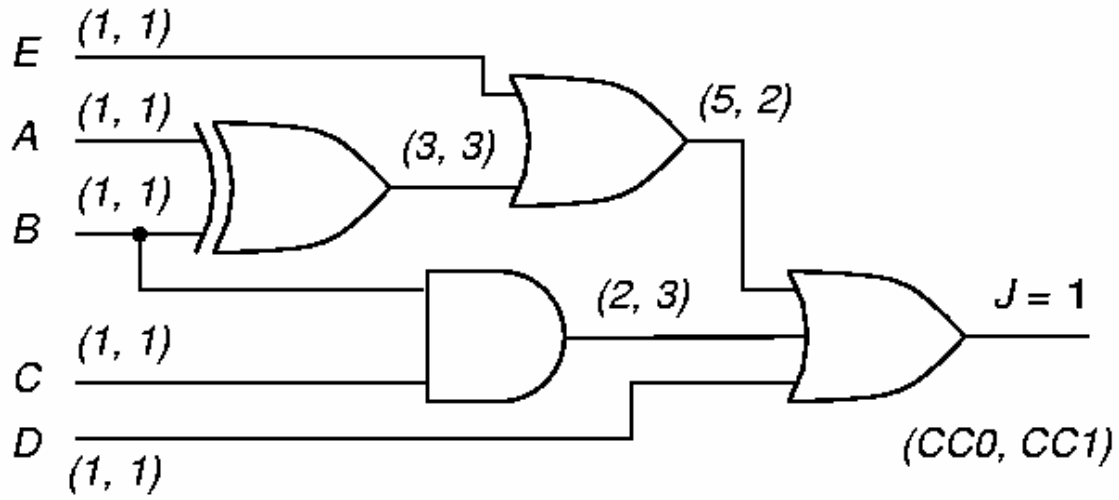
Stack ptr.

Signal	Value	Alternative tried
E	~~1~~ 0	~~NO~~ YES
B	0	YES
F	0	YES



Objectives and Backtracking of ATPG Algorithm

- *Objective* – desired signal value goal for ATPG
  - Guides it away from infeasible/hard solutions
- *Backtrace* – Determines which primary input and value to set to achieve objective
  - Use testability measures

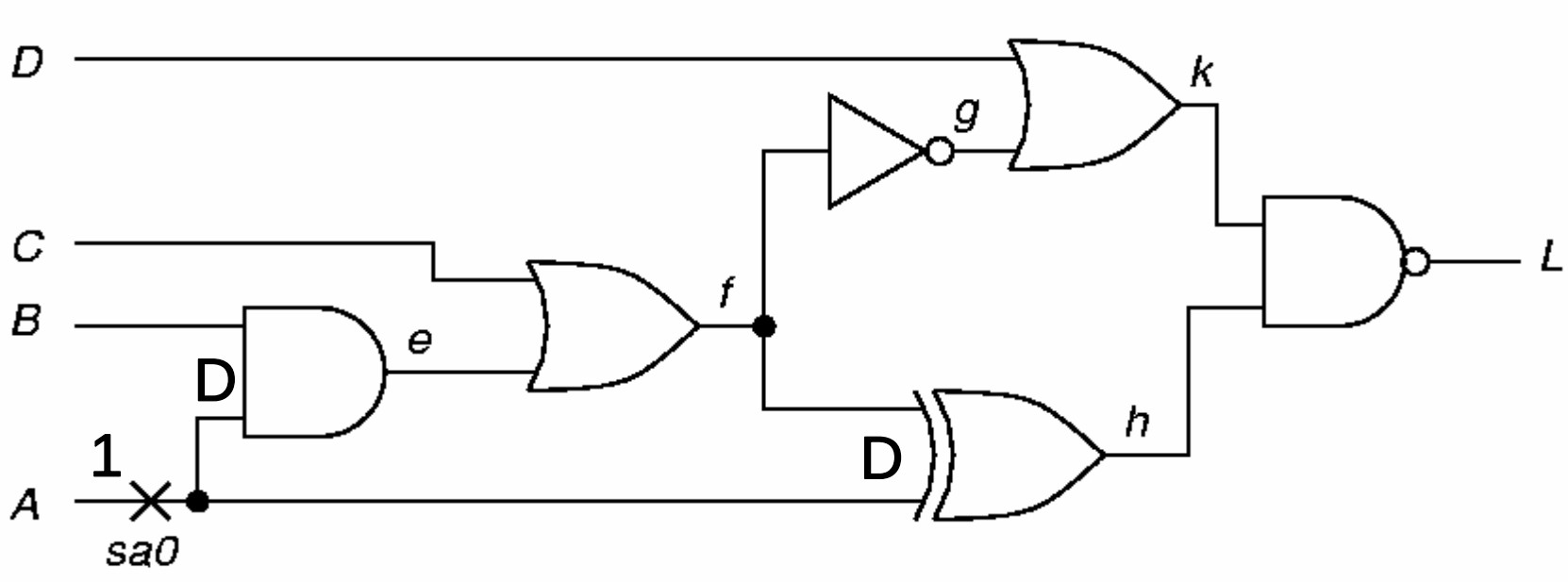


D-Algorithm D-Drive

```
while (untried fault effects on D-frontier)
    select next untried D-frontier gate for propagation;
    while (untried fault effect fanouts exist)
        select next untried fault effect fanout;
        generate next untried propagation D-cube;
        D-intersect selected cube with test cube;
        if (intersection fails or is undefined) continue;
        if (all propagation D-cubes tried & failed) break;
        if (intersection succeeded)
            add propagation D-cube to test cube -- recreate D-frontier;
            Find all forward & backward implications of assignment;
            save D-frontier, algorithm state, test cube, fanouts, fault;
            break;
        else if (intersection fails & D and D in test cube) Backtrack ();
        else if (intersection fails) break;
    if (all fault effects unpropagatable) Backtrack ();
```

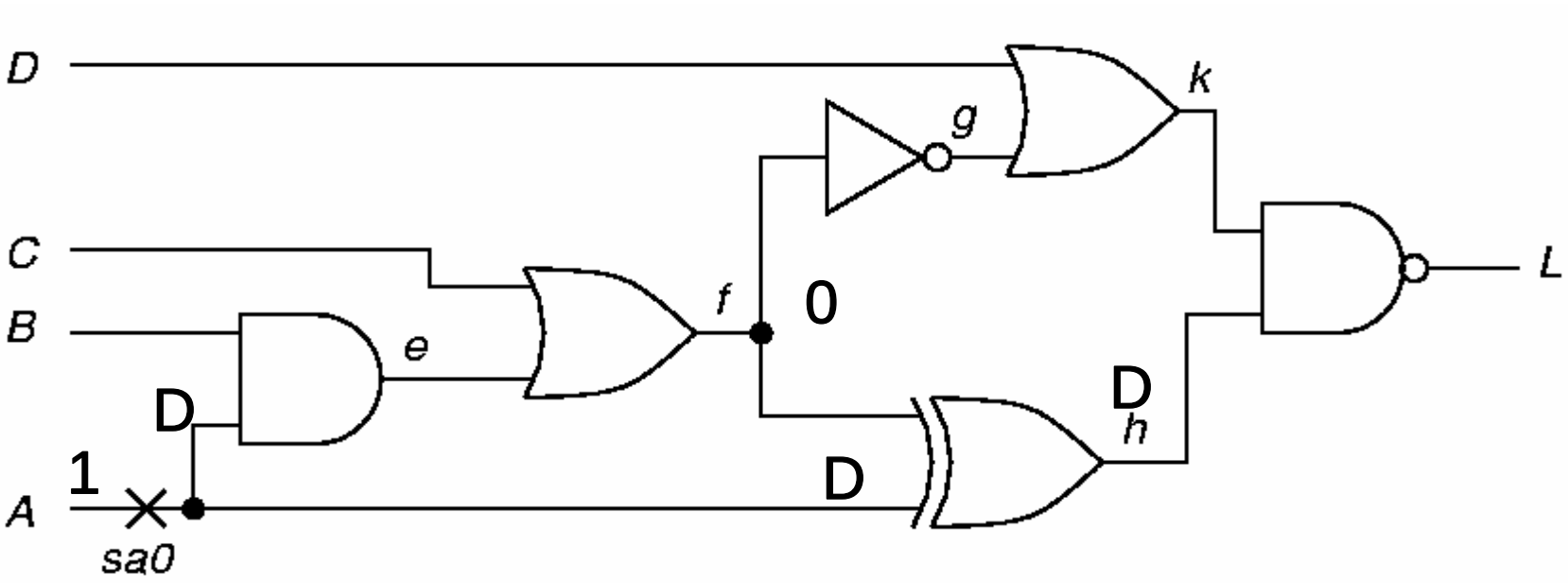
Example: Step 1: A: s-a-0

- **Step 1 – *D-Drive* – Set *A* = *1***



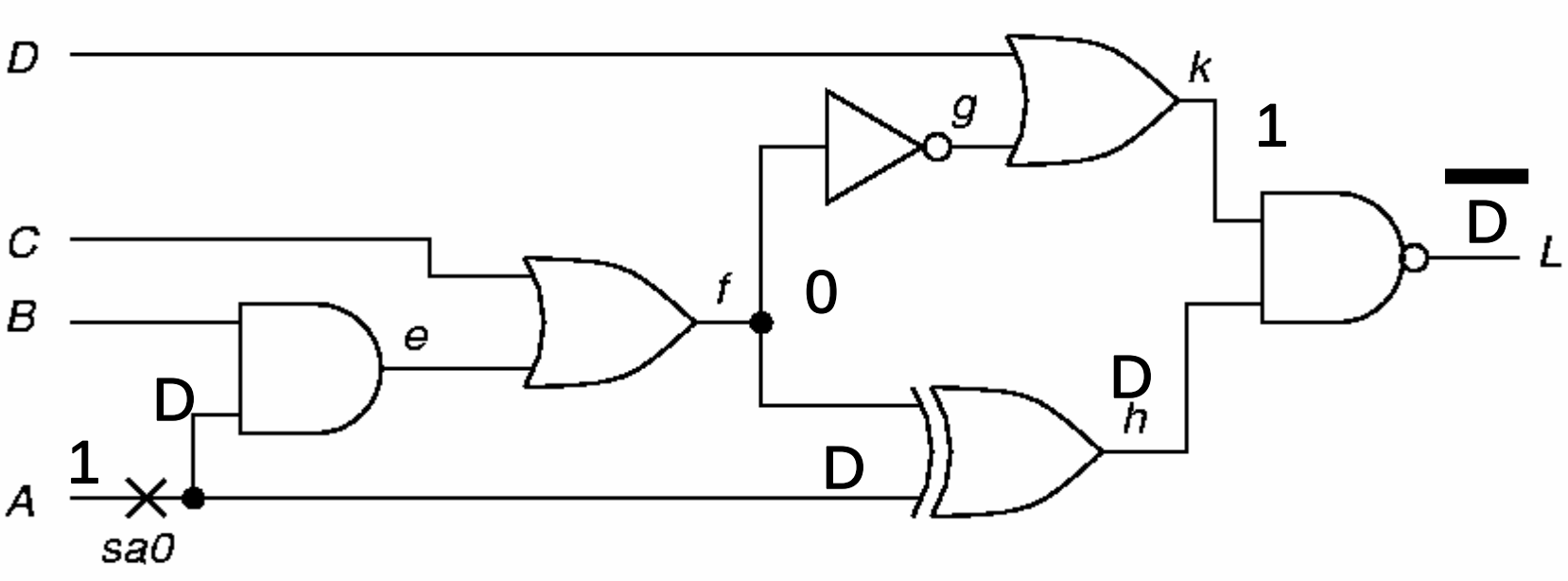
Example: Step 2: A: s-a-0

- **Step 2 – *D-Drive* – Set *f* = 0**



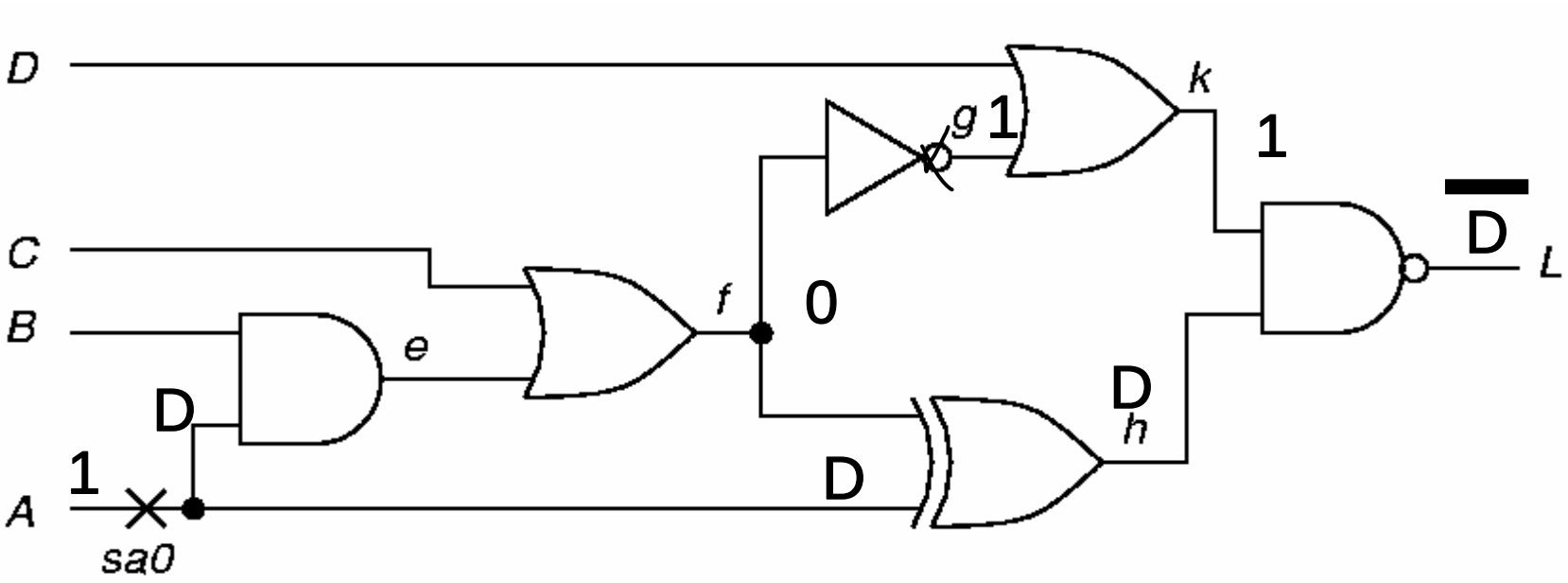
Example: Step 3: A: s-a-0

- **Step 3 – *D-Drive* – Set *k* = 1**



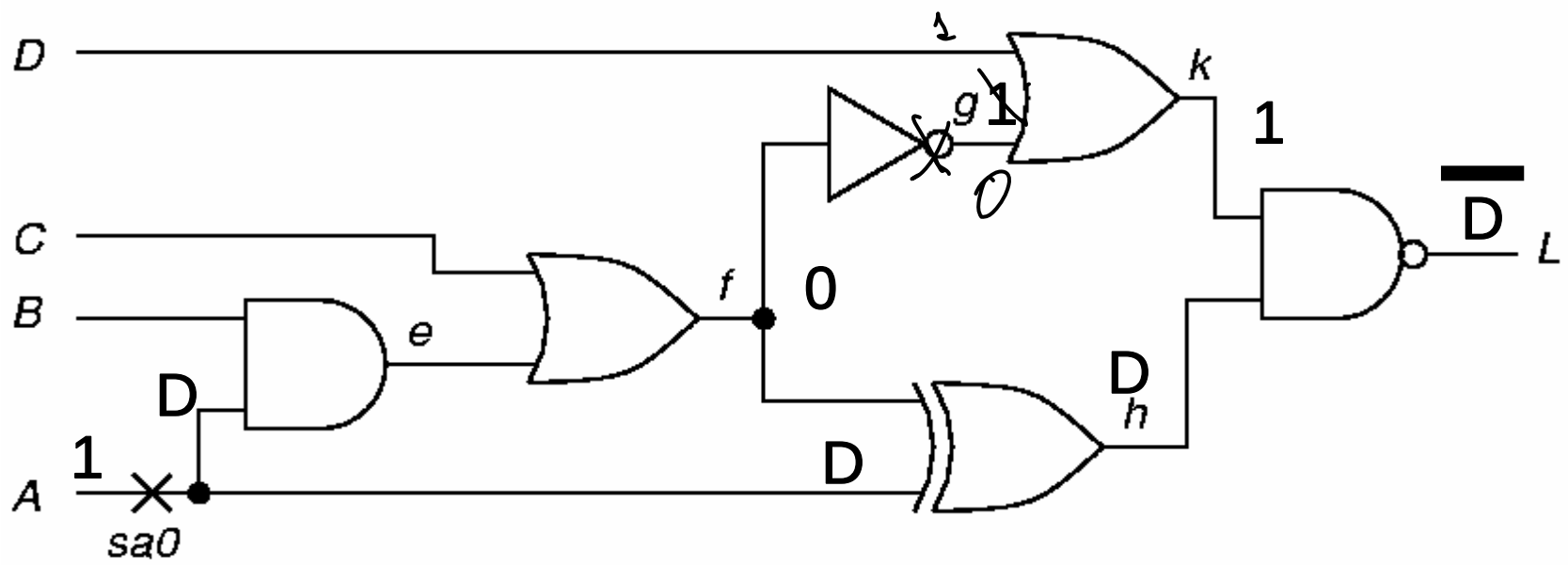
Example: Step 4: A: s-a-0

- **Step 4 – *Consistency* – Set *g* = 1**



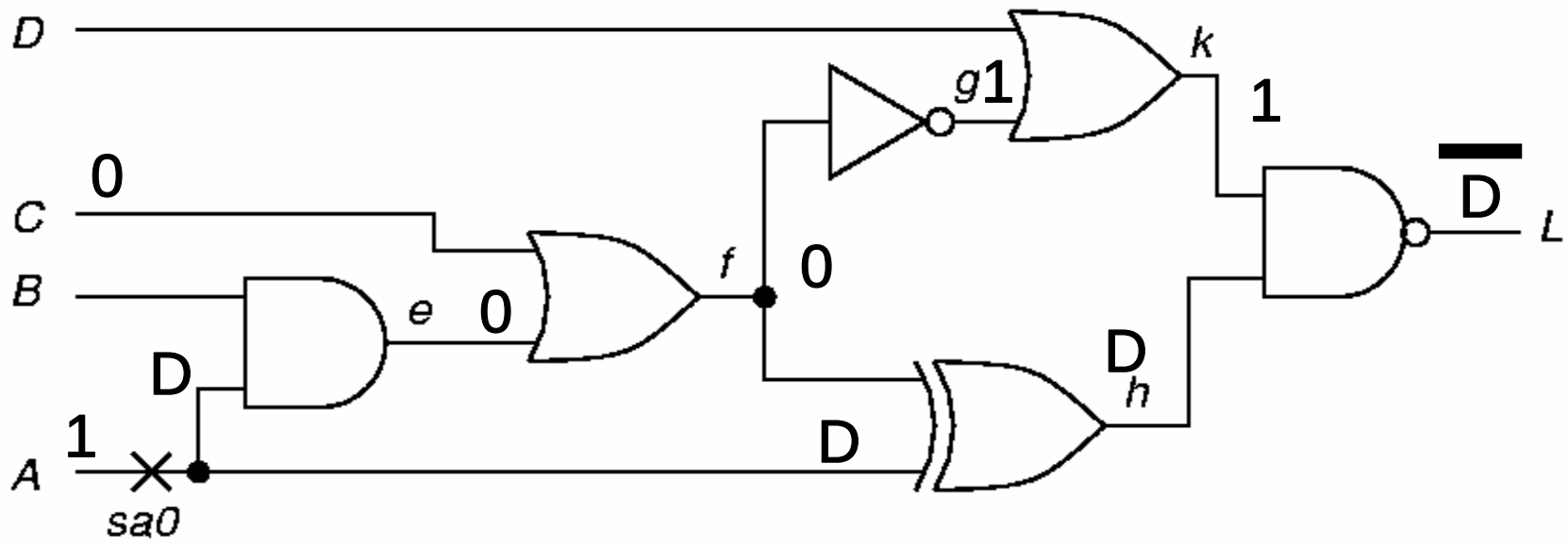
Example: Step 5: A: s-a-0

- **Step 5 – *Consistency* – *f* = 0 Already set**



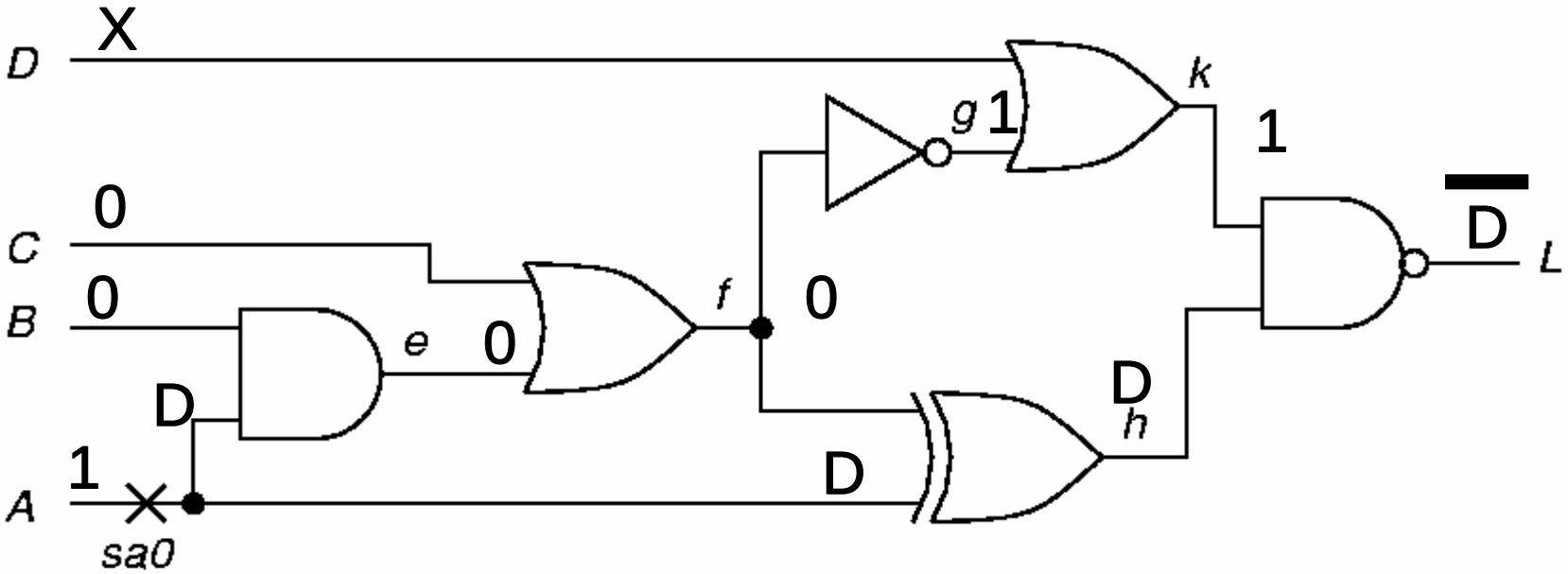
Example: Step 6: A: s-a-0

- **Step 6 – *Consistency* – Set *c* = 0, Set *e* = 0**

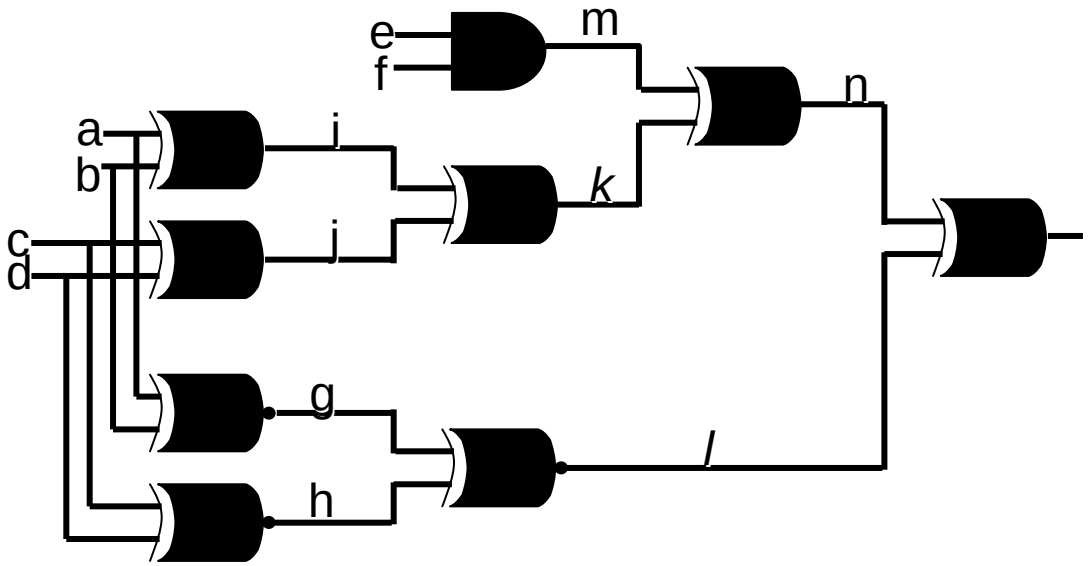


Example: Step 7: A: s-a-0

- **Step 7 – *Consistency* – Set *B* = 0**
  - ***D-Chain* dies**
- *Test cube*: *A*, *B*, *C*, *D*, *e*, *f*, *g*, *h*, *k*, *L*



D-algorithm - Problem



Note that *k* and *l* are complementary signals.

Assume *k* =1, *l* =1 is chosen as assignment by D-algorithm.

D-algorithm determines too late that this is inconsistent.

Solution: Backtrack only on PI values to determine consistency of signals

Implicit Enumeration: PODEM

Actual space of consistent assignments is only 2^n, where n is the number of primary inputs

Hence, search space can be greatly reduced (compared to D-algorithm) by enumerating over primary inputs only

PODEM (Path oriented decision making) is such an algorithm

Objectives -- bring ATPG closer to propagating D ($\overline{D}$) to PO

- *Backtracing*

Motivation

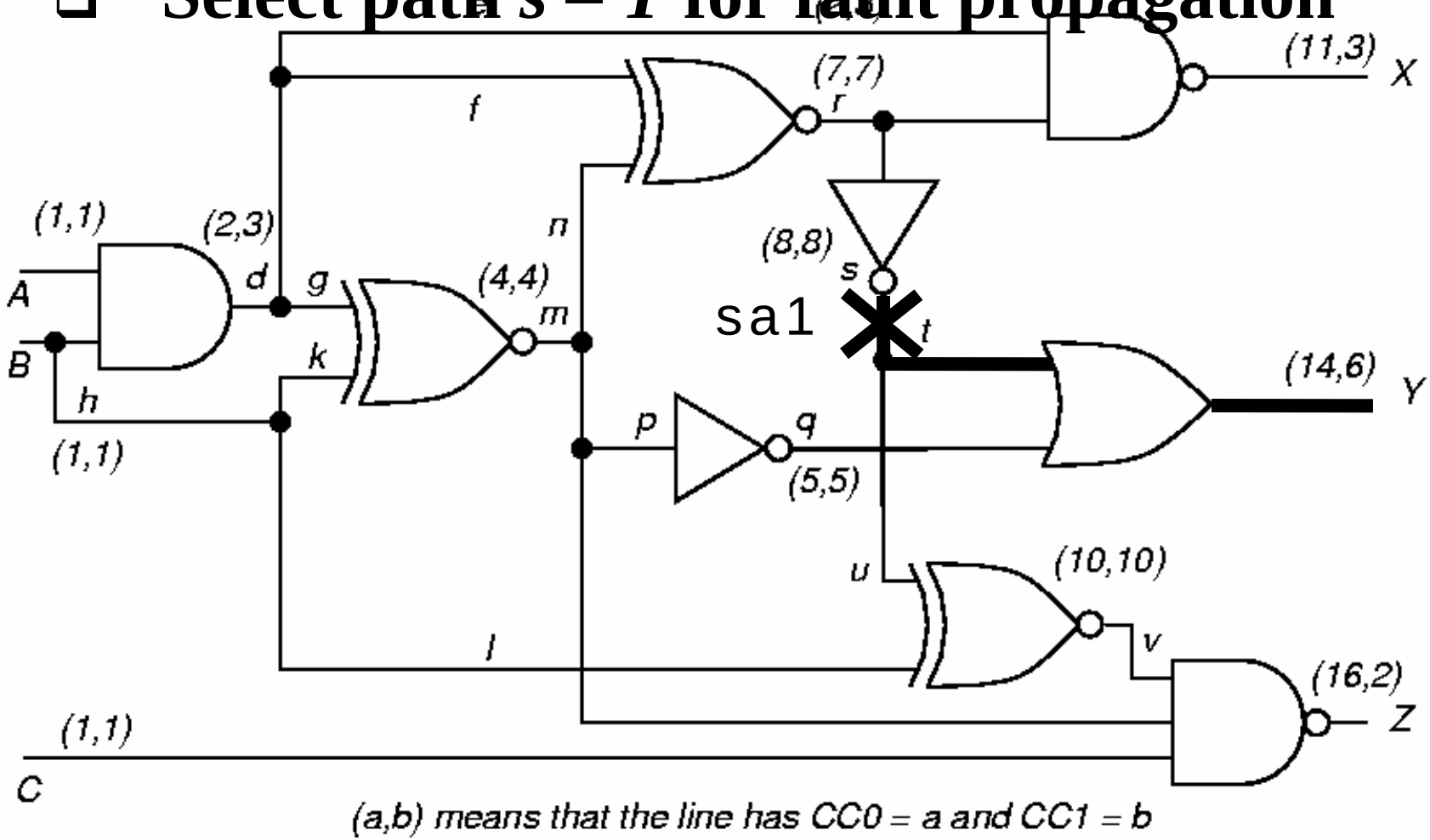
- **IBM introduced semiconductor DRAM memory into its mainframes – late 1970's**
- **Memory had error correction and translation circuits – improved reliability**
  - **D-ALG unable to test these circuits**
    - **Search too undirected**
    - **Large XOR-gate trees**
    - **Must set all external inputs to define output**
  - **Needed a better ATPG tool**

PODEM High-Level Flow

1. **Assign binary value to unassigned PI**
2. **Determine implications of all PIs**
3. **Test Generated? If so, done.**
4. **Test possible with more assigned PIs? If maybe, go to Step 1**
5. **Is there untried combination of values on assigned PIs? If not, exit: untestable fault**
6. **Set untried combination of values on assigned PIs using objectives and backtrace. Then, go to Step 2**

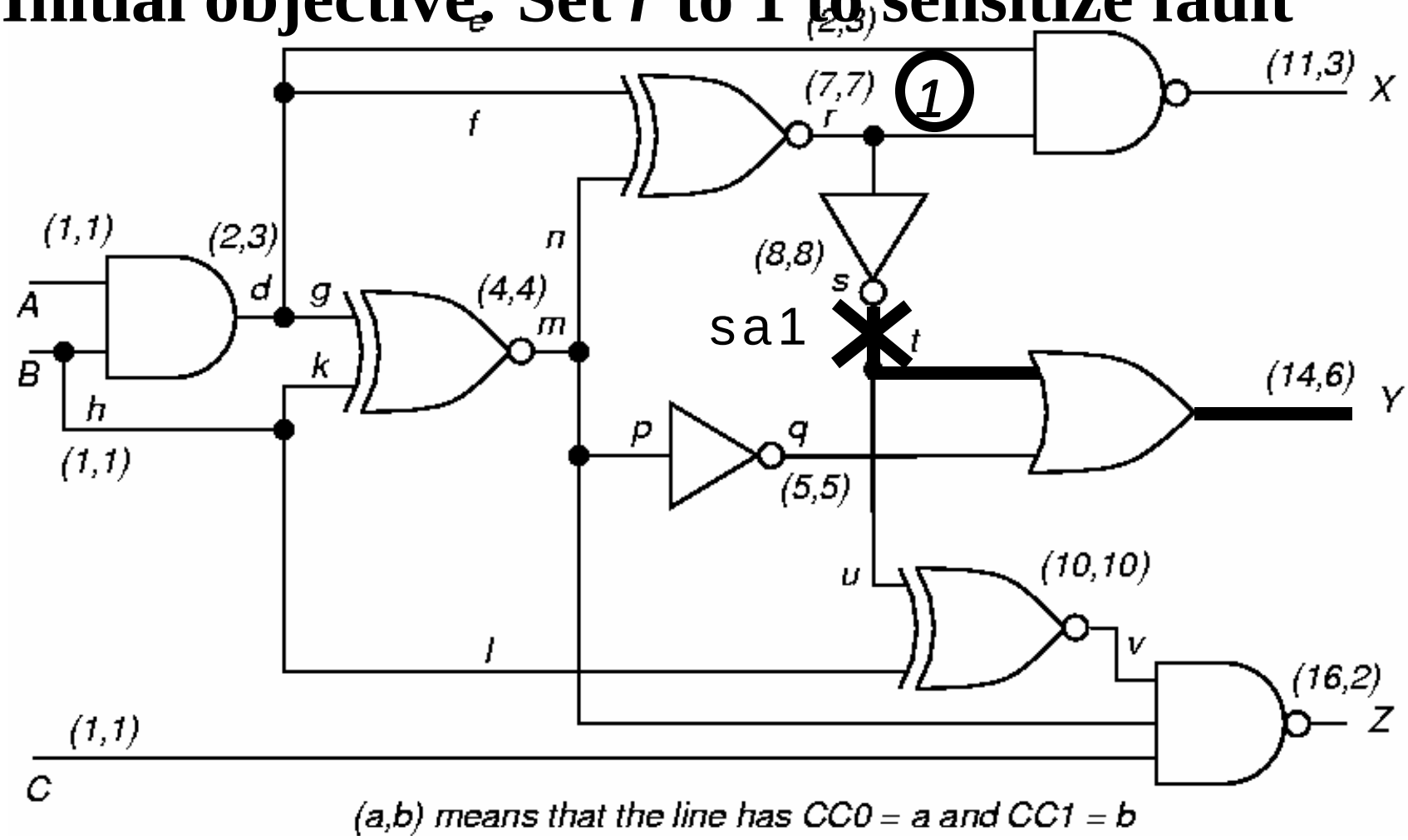
Example: PODEM

- **Select path *s* – *Y* for fault propagation**



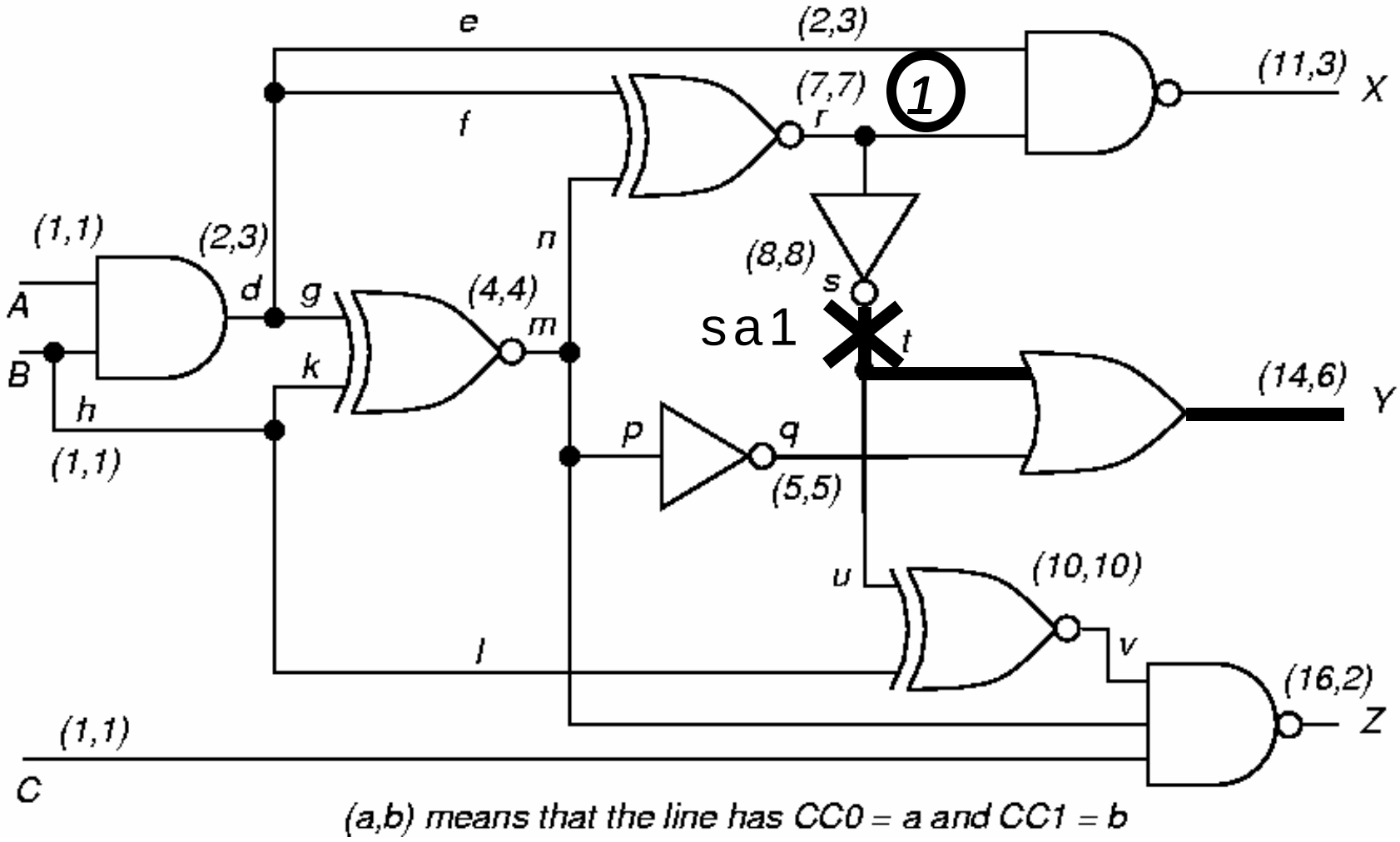
Step 2

- **Initial objective: Set *r* to 1 to sensitize fault**



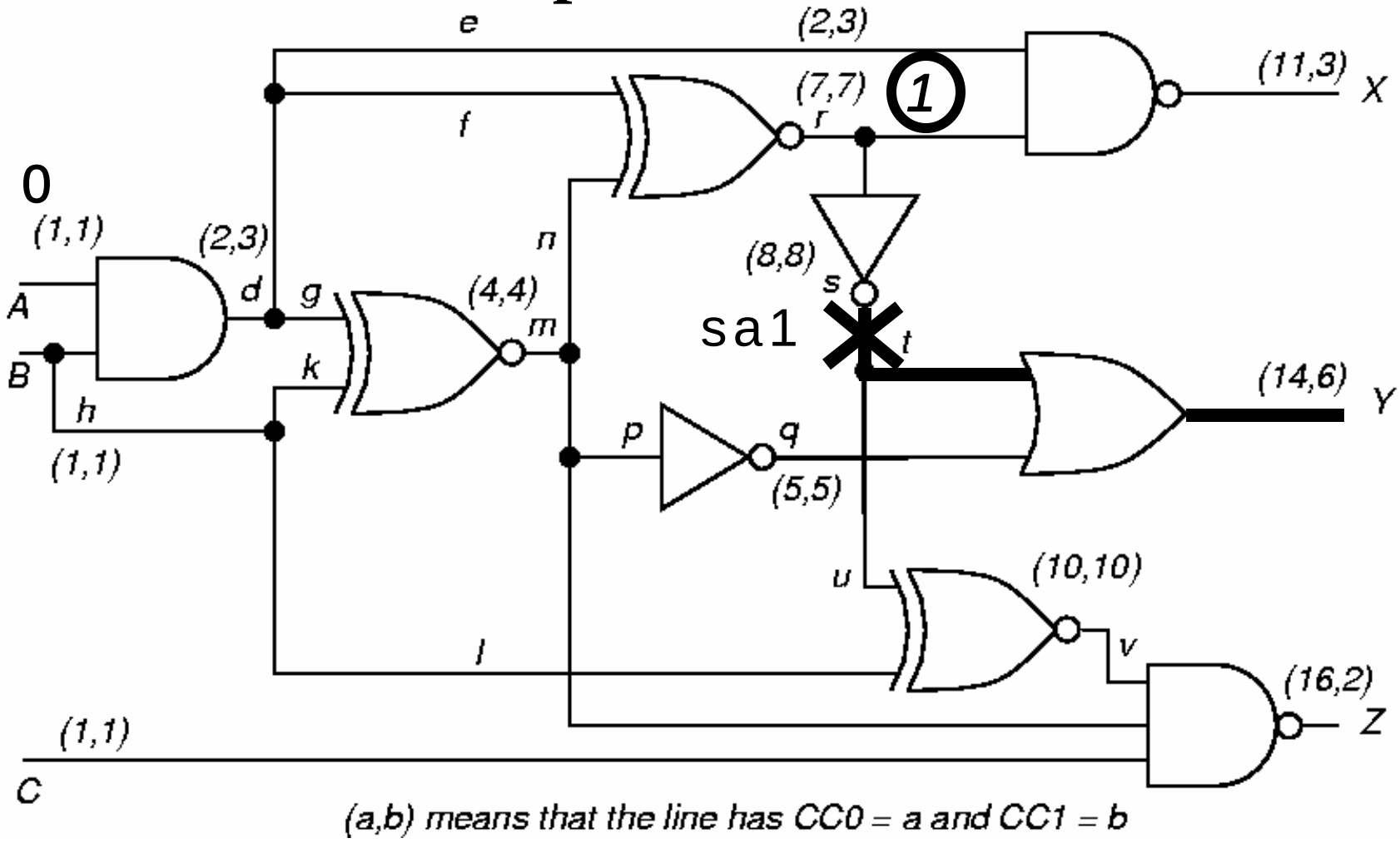
Step 3

- **Backtrace from *r***



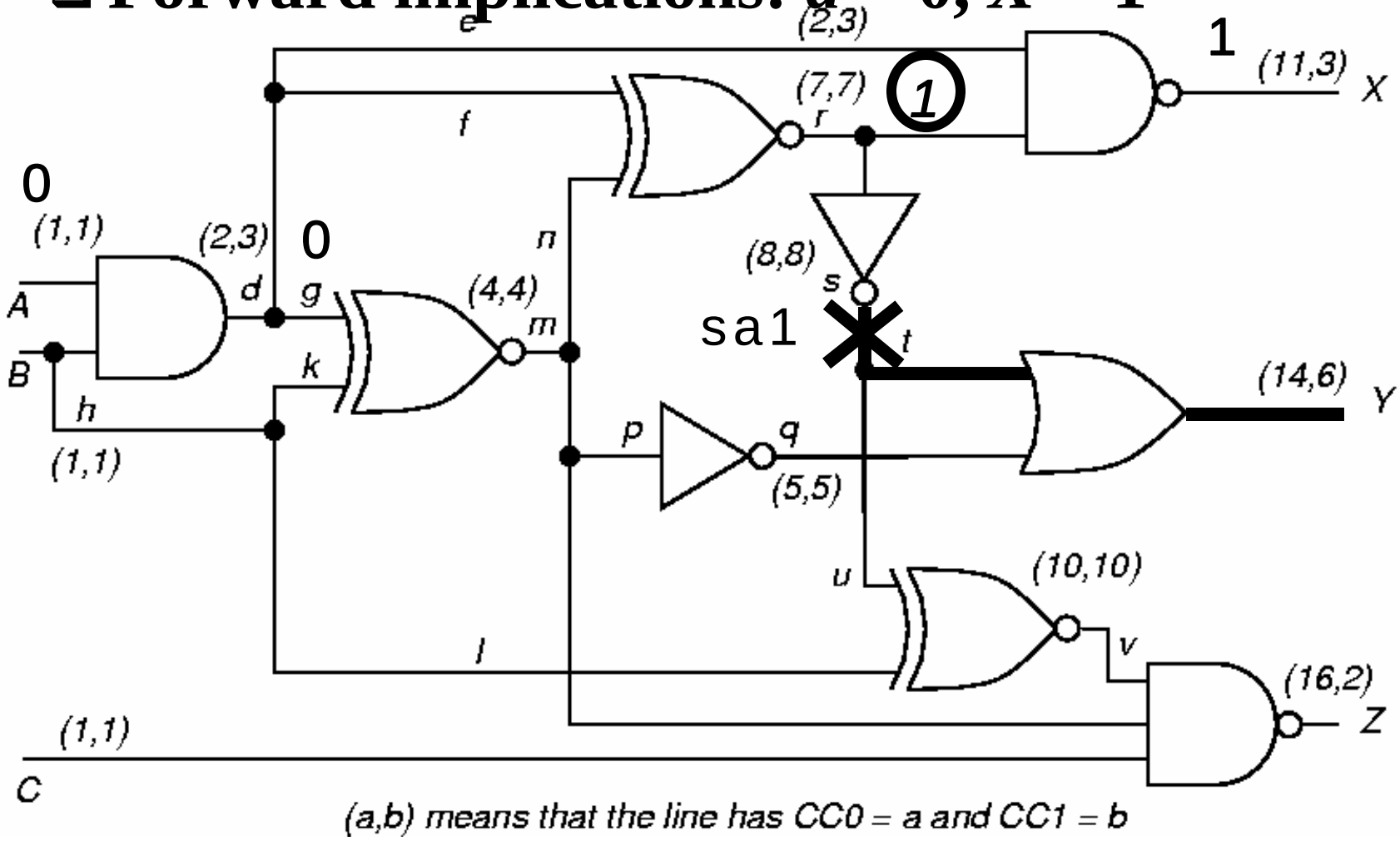
Step 4

- **Set *A* = 0 in implication stack**



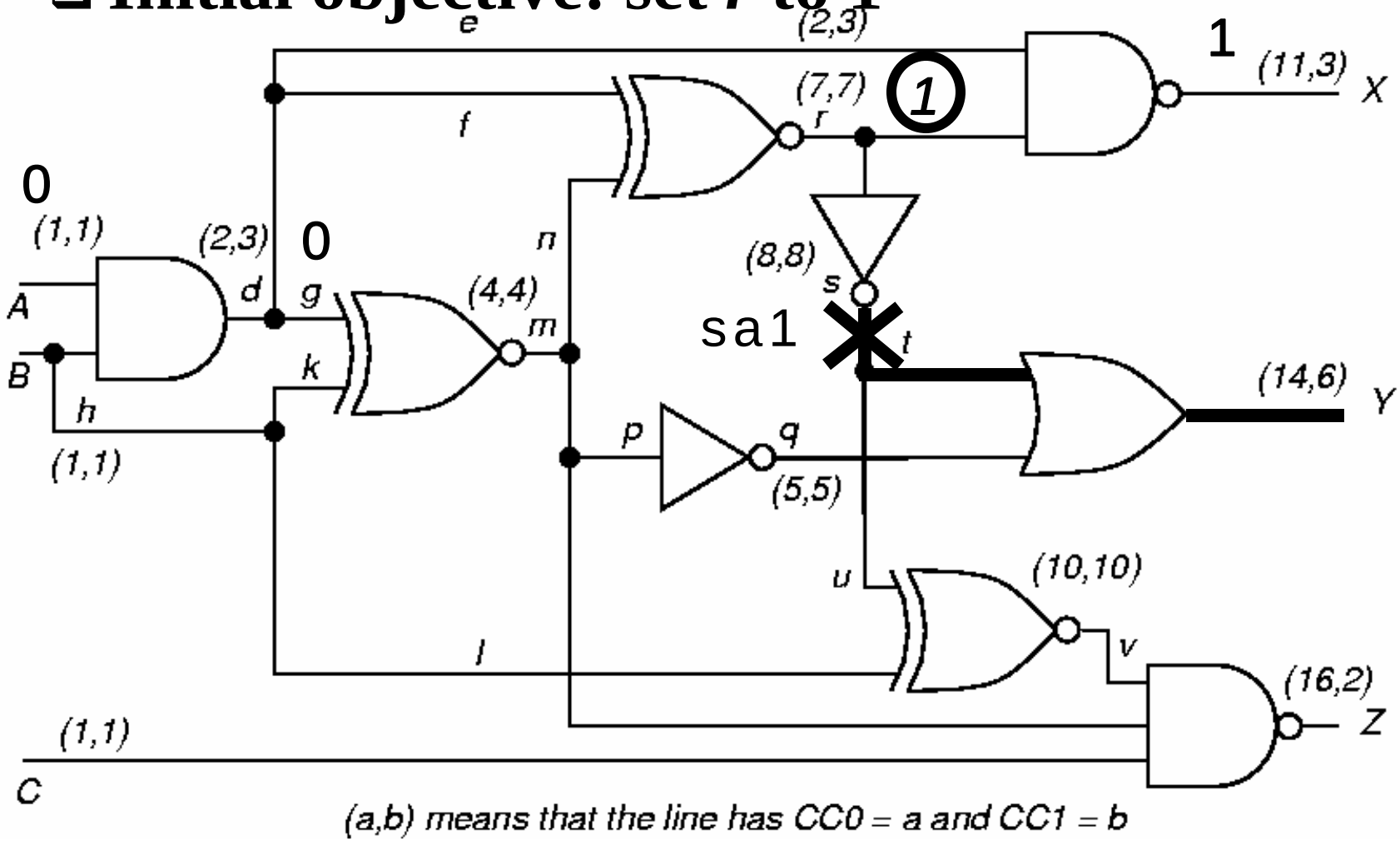
Step 5

- **Forward implications: $d = 0$, $X = 1$**



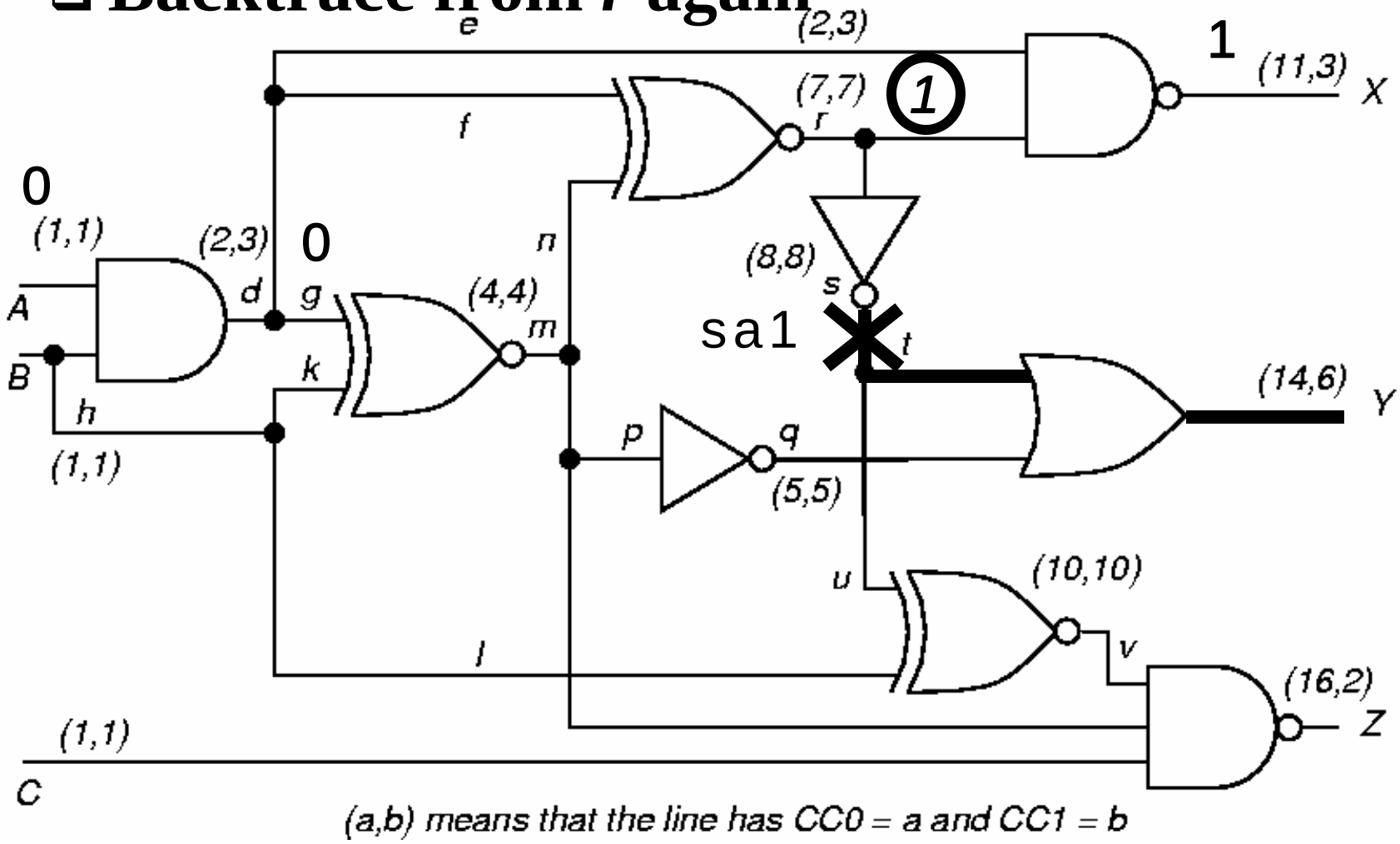
Step 6

- **Initial objective: set *r* to 1**



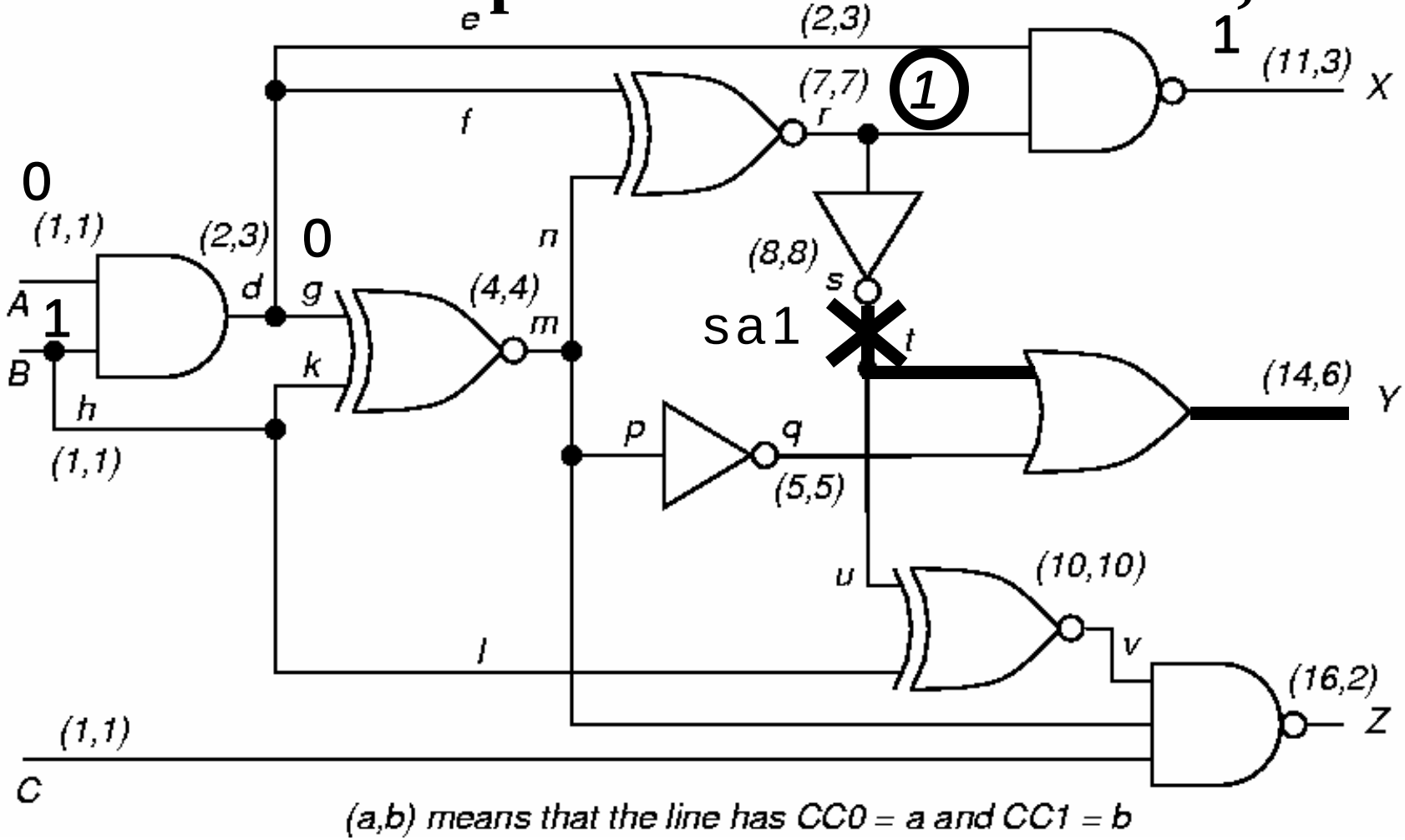
Step 7

- **Backtrace from *r* again**



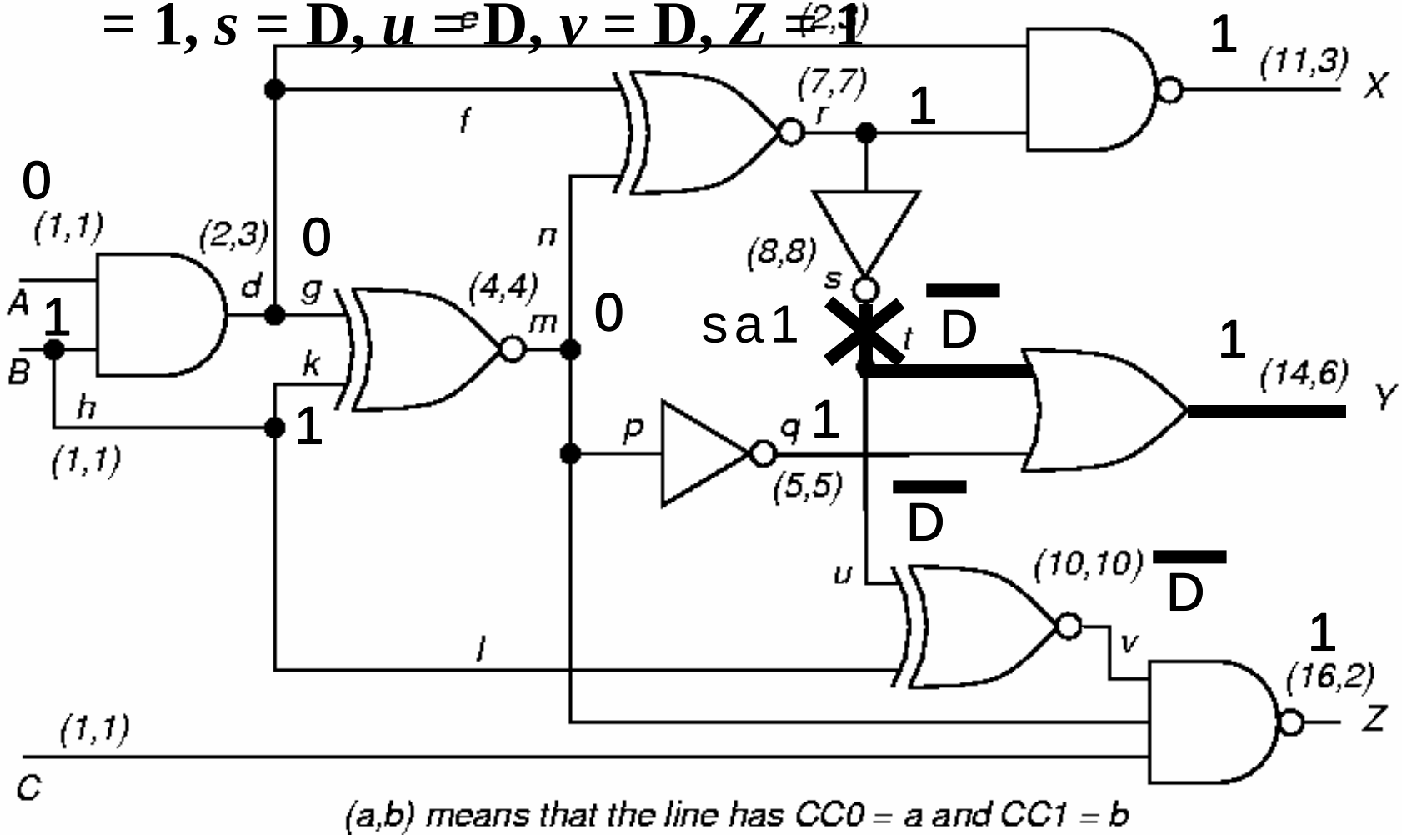
Step 8

- **Set *B* to 1. Implications in stack: *A* = 0, *B* = 1**



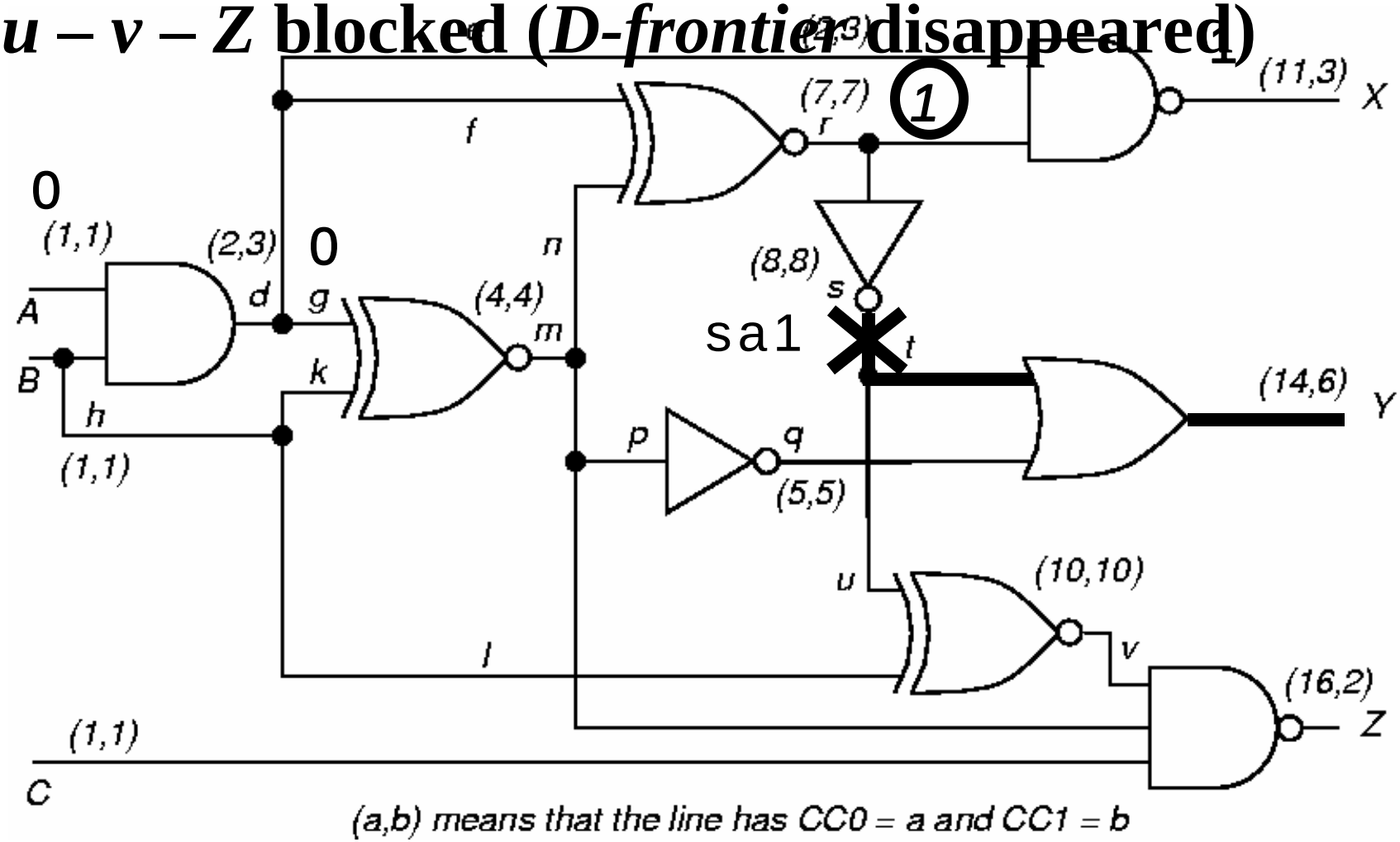
Step 9

- **Forward implications: $k = 1$, $m = 0$, $r = 1$, $q = 1$, $Y = 1$, $s = D$, $u = \overline{D}$, $v = D$, $Z = 1$**



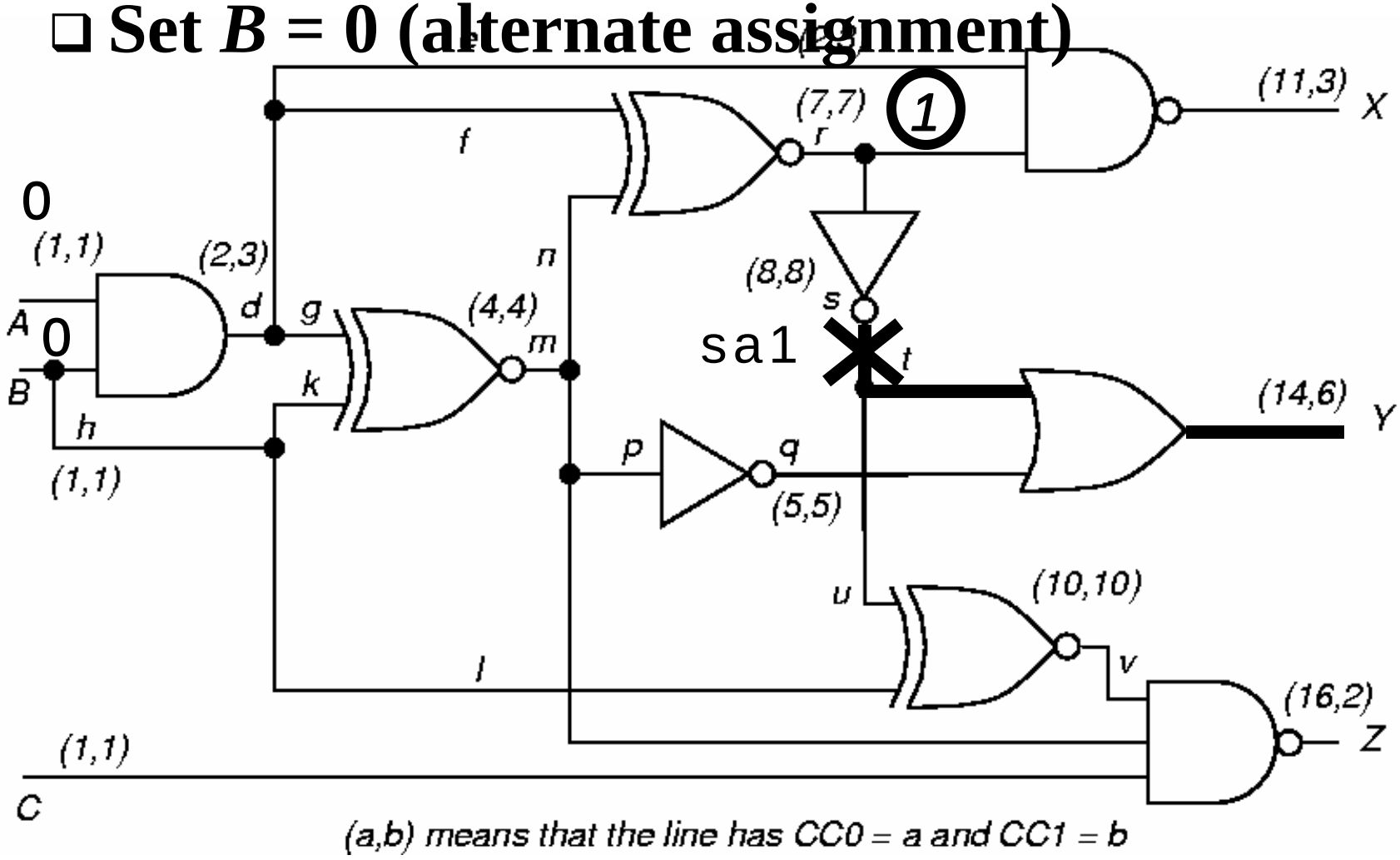
Step 10

- ***X-PATH-CHECK*** **shows paths *s – Y*** **and** **s – *u – v – Z*** **blocked (*D-frontier*** **disappeared)**



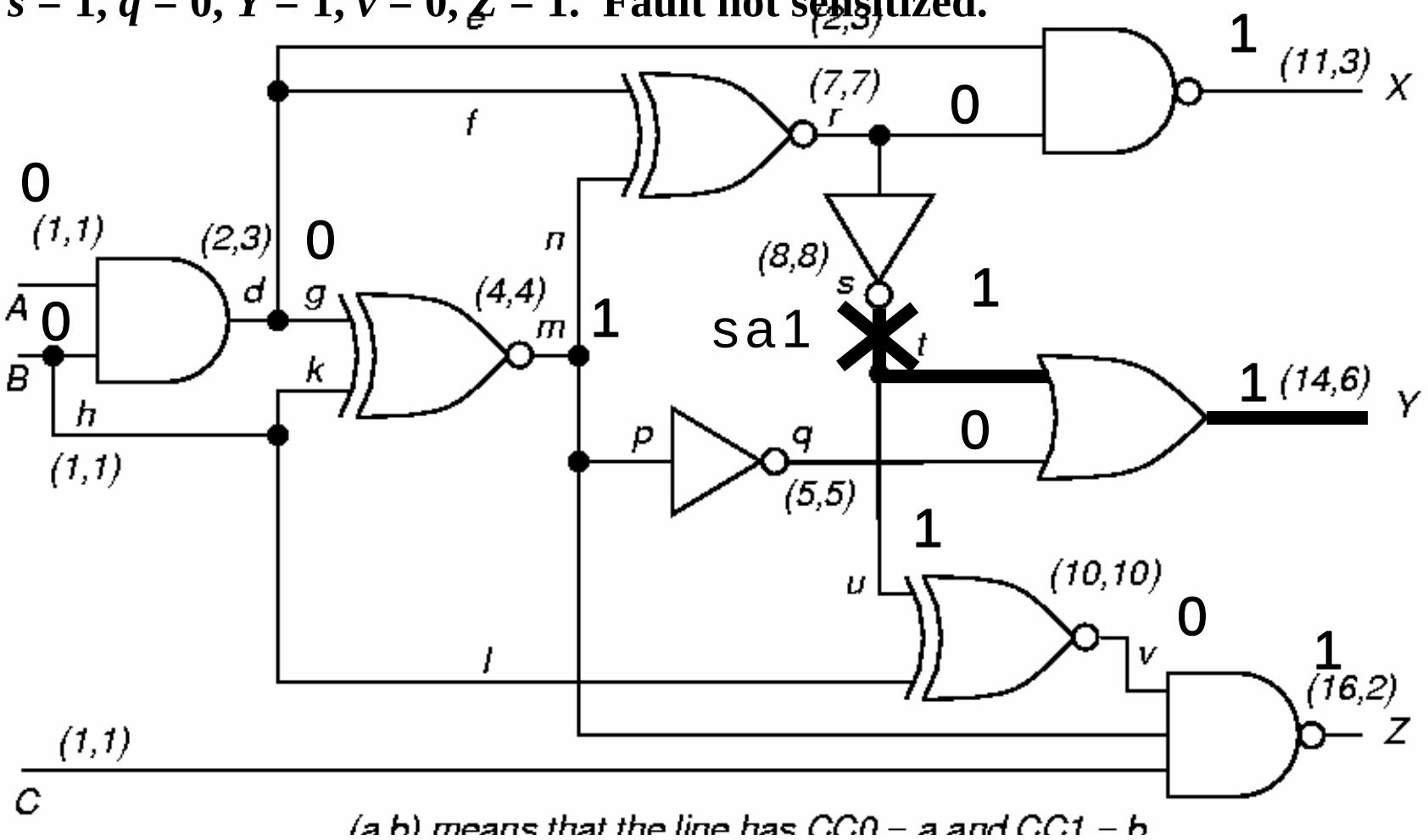
Step 11

- **Set *B* = 0 (alternate assignment)**



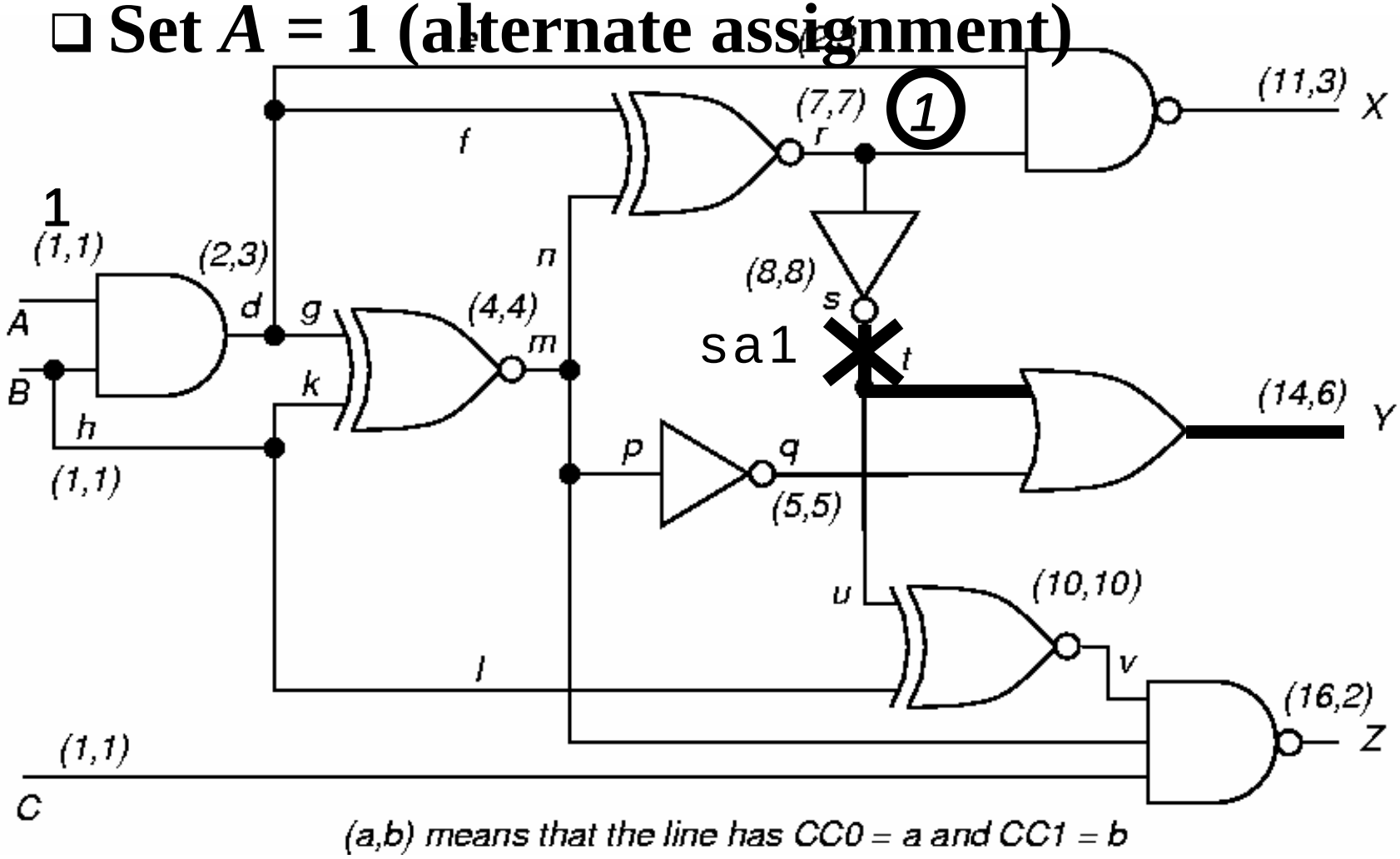
Step 12 - Backtrack

- **Forward implications: $d = 0$, $X = 1$, $m = 1$, $r = 0$, $s = 1$, $q = 0$, $Y = 1$, $v = 0$, $Z = 1$. Fault not sensitized.**



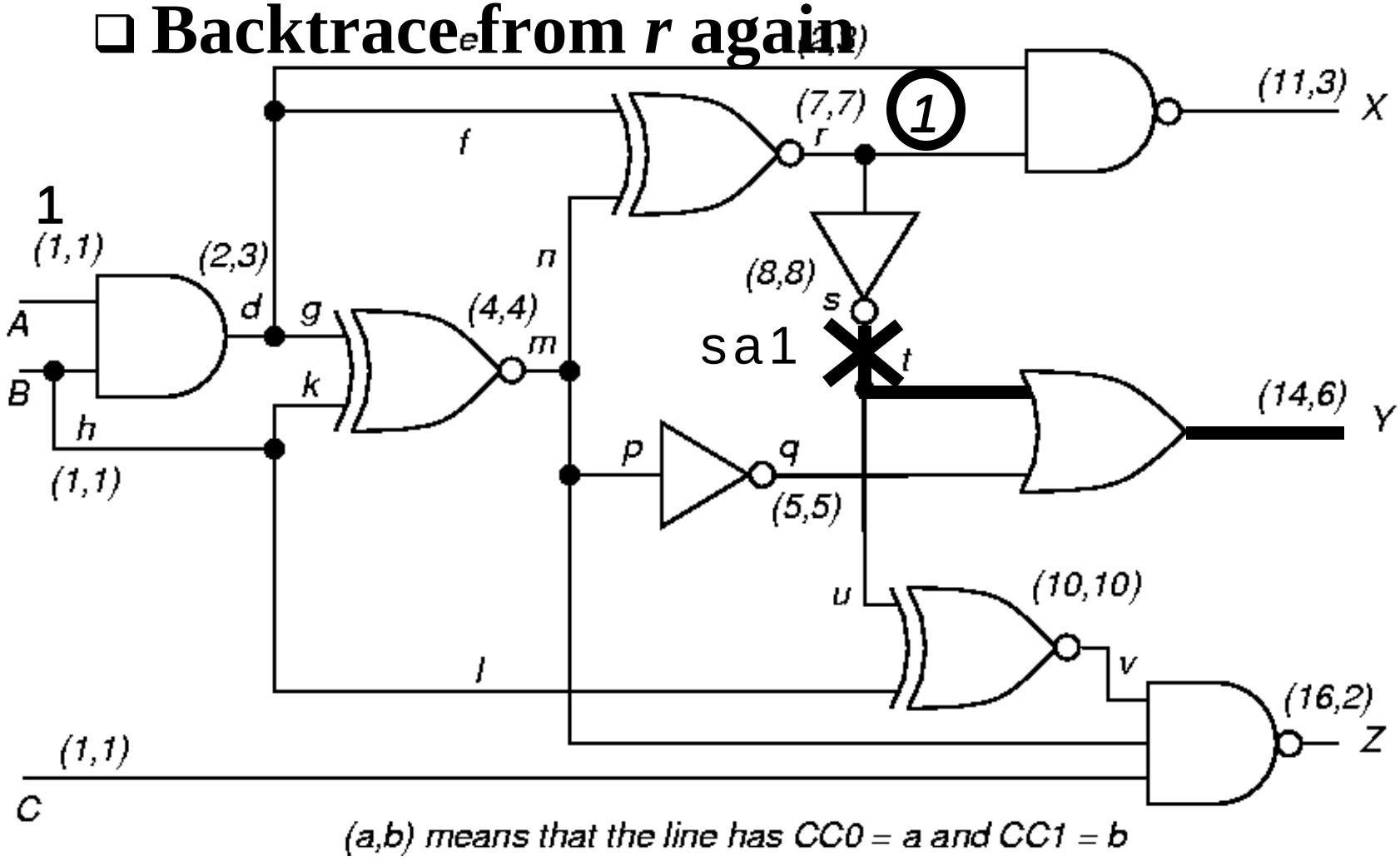
Step 13

- **Set *A* = 1 (alternate assignment)**



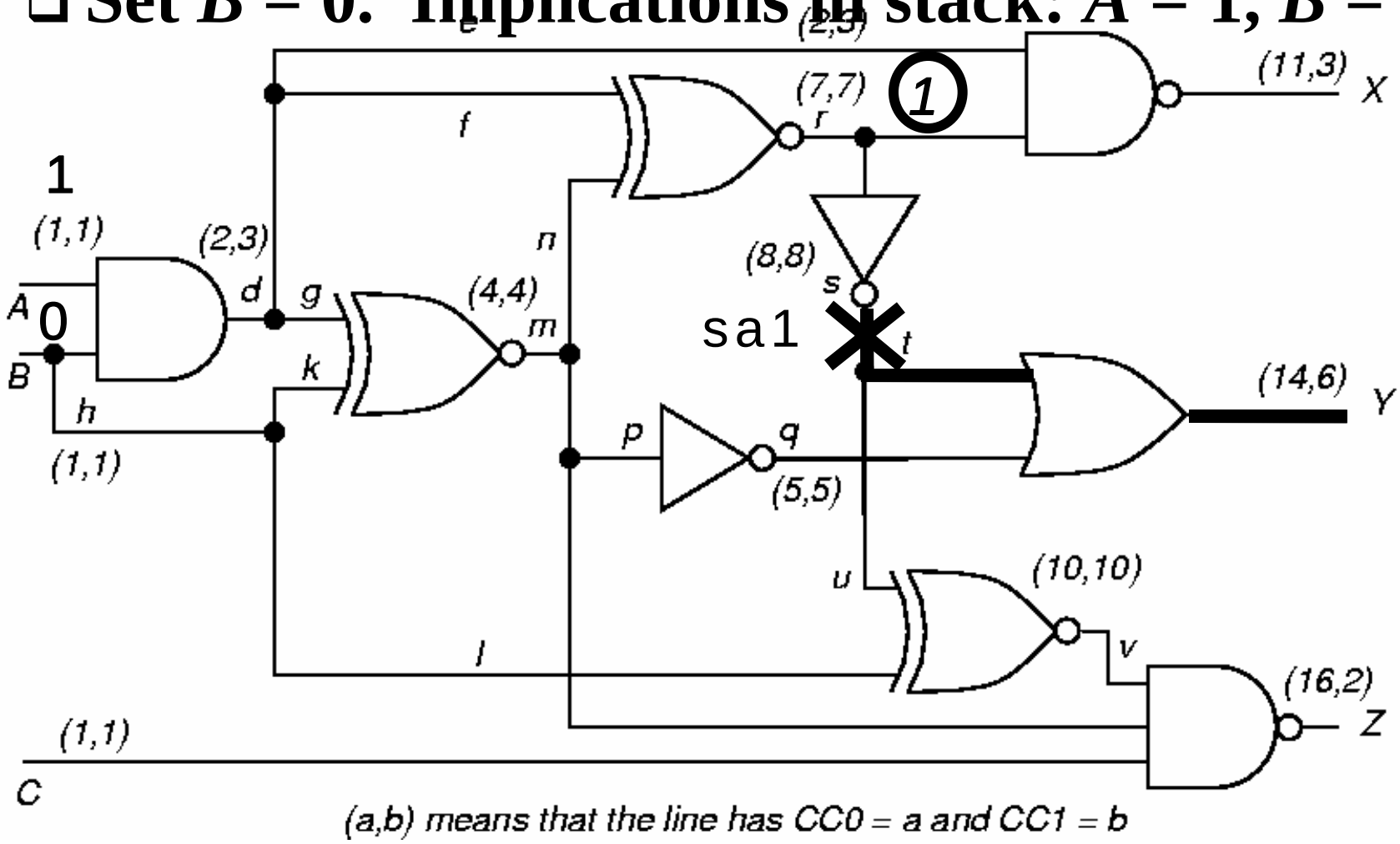
Step 14

- **Backtrace from *r* again**



Step 15

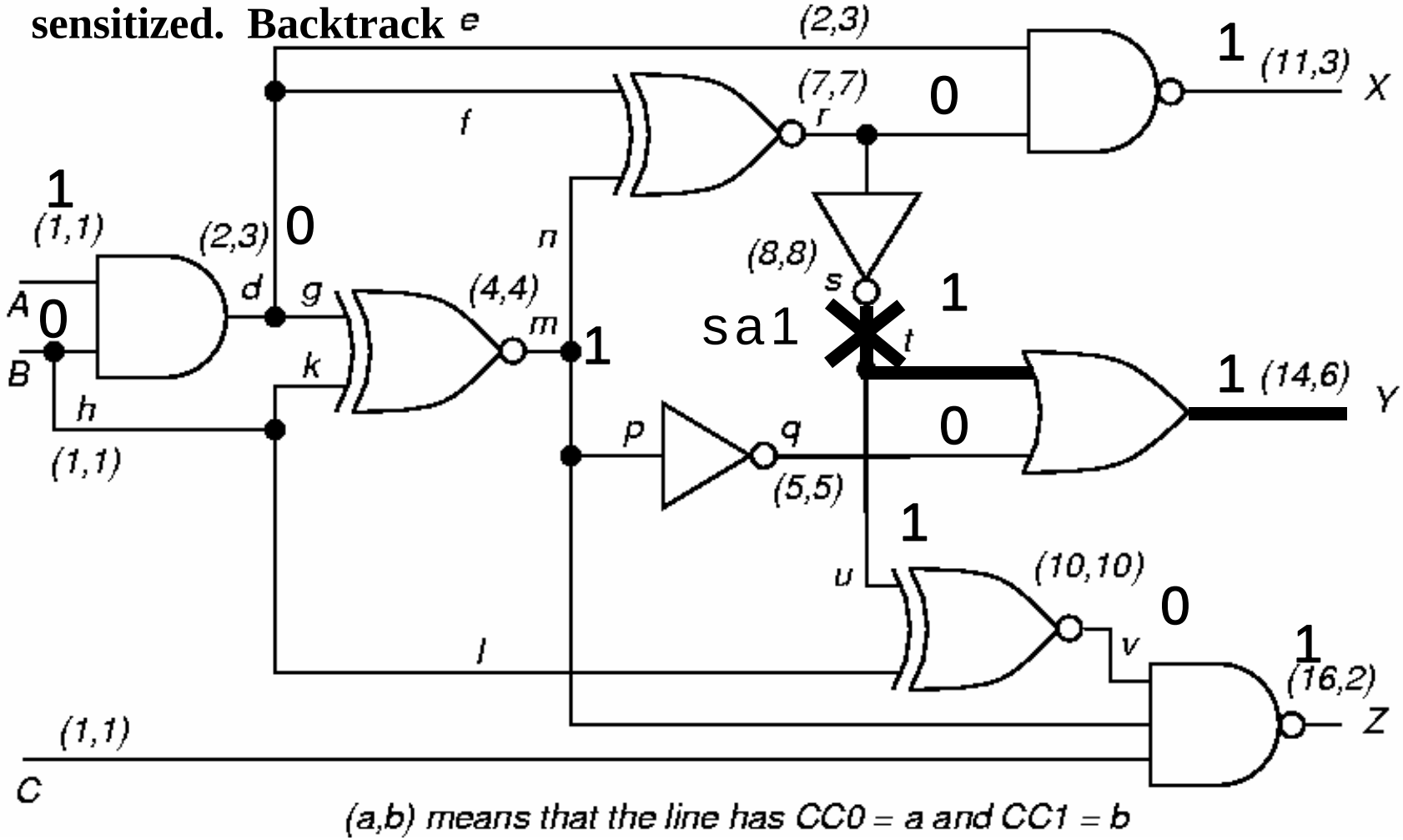
- **Set *B* = 0. Implications in stack: *A* = 1, *B* = 0**



(a,b) means that the line has CC0 = a and CC1 = b

Step 16 - Backtrack

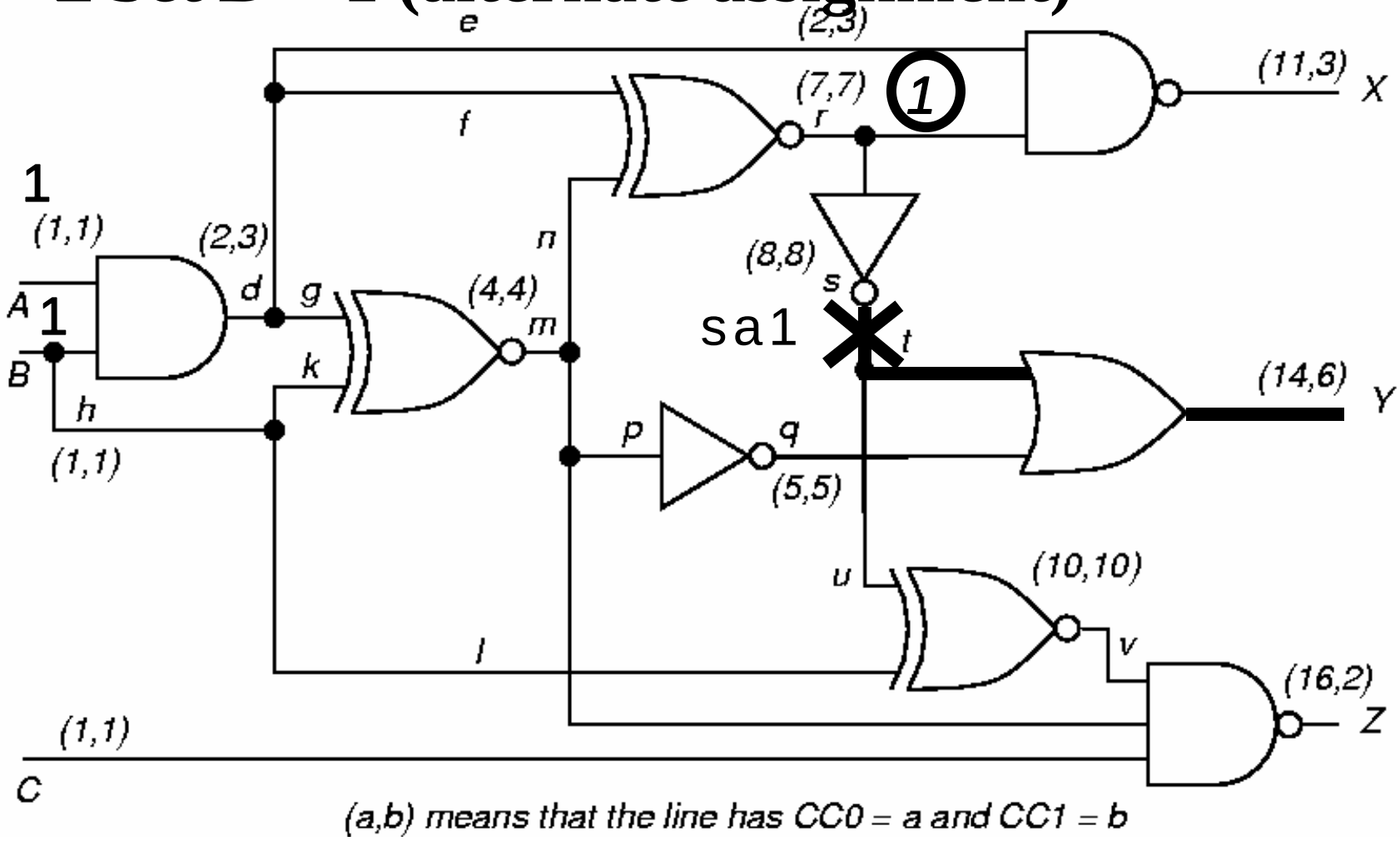
- **Forward implications: $d = 0$, $X = 1$, $m = 1$, $r = 0$. Conflict: fault not sensitized. Backtrack**



(a,b) means that the line has CC0 = a and CC1 = b

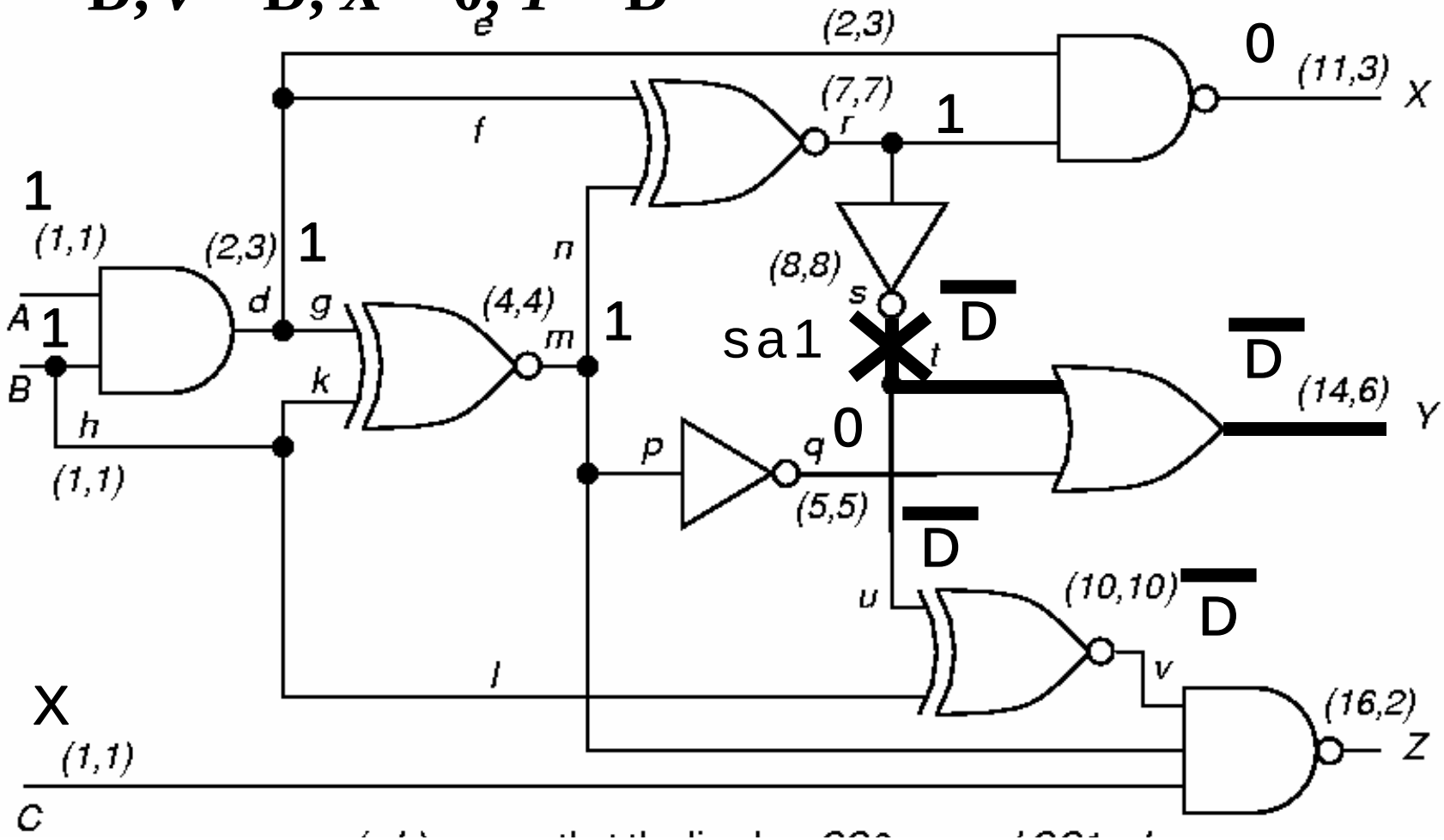
Step 17

- **Set *B* = 1 (alternate assignment)**

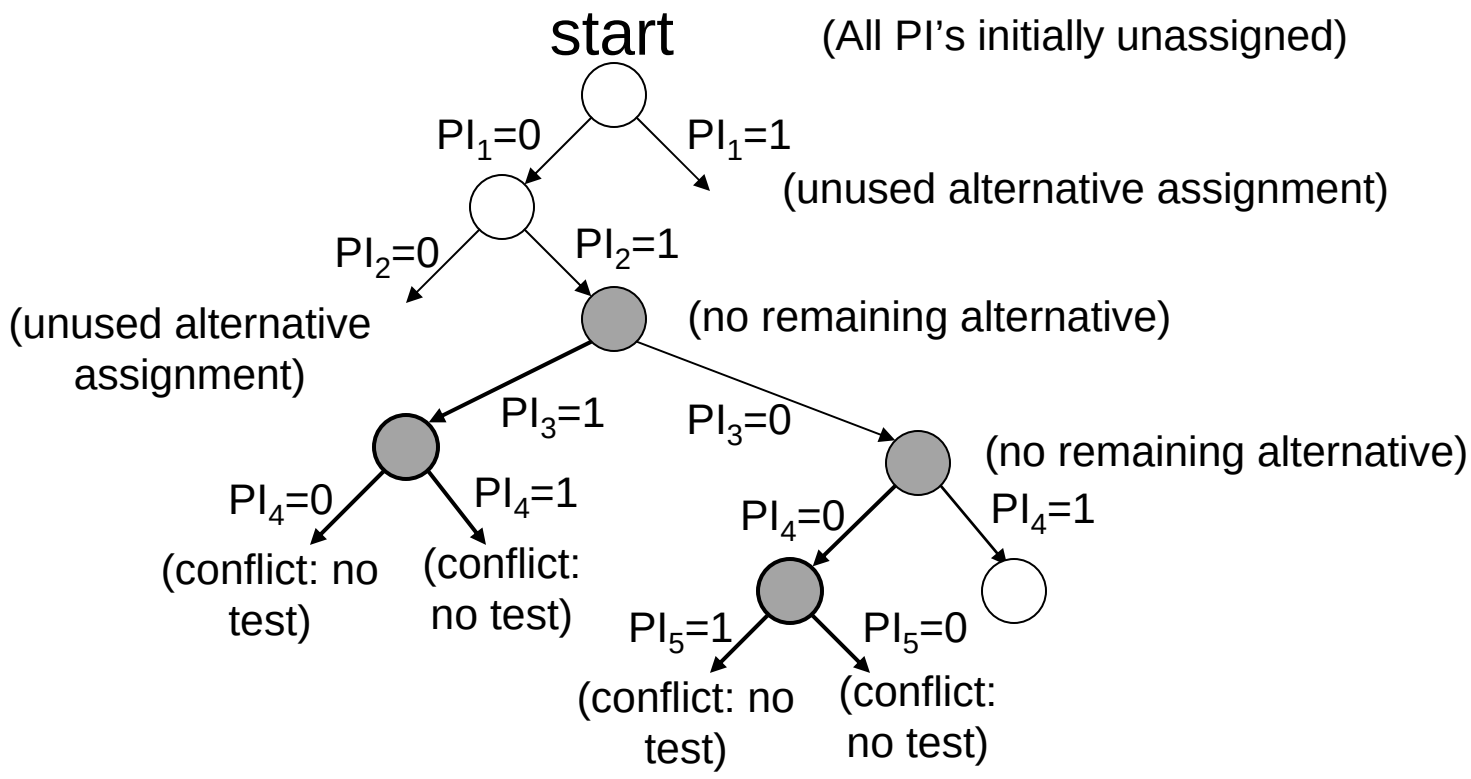


Step 18 - Fault Detected

- ~~**Forward implications:** $d = 1, m = 1, r = 1, q = 0, s = \overline{D}, v = \overline{D}, X = 0, Y = \overline{D}$~~



PODEM Decision Tree



indicates no remaining alternative at node

PODEM: Algorithm

1. Start with given fault, empty decision tree, all PI's set to *X*
2. 3 types of operations performed
   a) check if current PI assignment is *consistent*. If so, choose an unassigned PI and set it to 0 or 1
   b) If inconsistent and if alternative value of currently assigned PI has not been tried, try it and mark this PI as having *no remaining alternative*
   c) If no remaining alternative on this PI, *backup* to previous PI assigned, deleting the decision tree below

Algorithm complete: either terminates with a test (all PI's assigned) or proves fault is redundant

PODEM: Heuristics

Choosing which PI to assign next

- This depends on how the fault could propagate to a primary output
- Choose “closest” PO to which fault can propagate and determine which PI affects the propagation “the most”
- This is done by computing approximate node controllabilities and observabilities

Heuristic is quite ad-hoc.

PODEM is ineffective on large networks with a lot of reconvergence

FAN

- Improvements to PODEM:
  - Backward traversal only up to “head lines”
    - Lines that cannot cause conflict or input lines
      - Fanout stems
  - Immediate implications – forward, backward
  - Finds unique (single) sensitization paths
  - Breadth-first multiple backtrace

Socrates

Provides improvements to PODEM

- implications
- static and dynamic learning

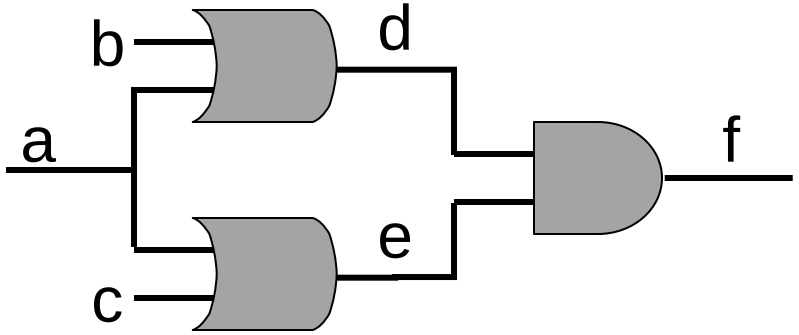
Basic Idea

- When a value is set on a node (due to current partial PI assignment) what conclusions can be made?
  - ✦ Values on other nodes/PI's

  This allows detection of inconsistencies, thereby causing early backtracking

Implications

EXAMPLE:

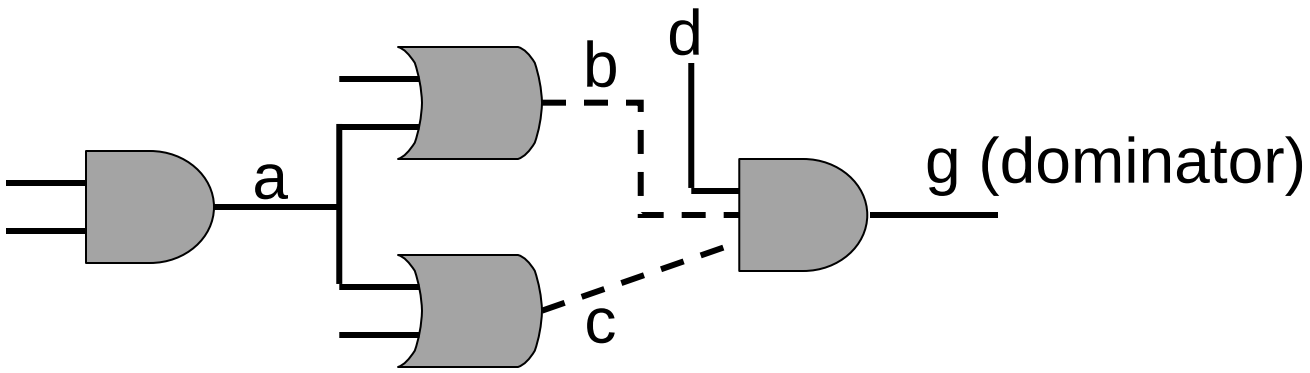


$a = 1 \Rightarrow (d = 1)\ \&\ (e = 1) \Rightarrow f = 1$

Hence, $f = 0 \Rightarrow a = 0$

Implications are computed in pre-processing phase and stored for use during backtracking and assignment phase of algorithm

Static and Dynamic Learning



If a has a D value and it must propagate through g, d must be set to 1. If it can't be, then D on a can't propagate.

- This is an implication learned from the topology of the network

$$a = D \Rightarrow d = 1$$

- Static learning: implications learned in pre-processing phase
- Dynamic learning: implications learned under partial PI assignments

Socrates: Algorithm

1. Perform implication of each single lead value and store information
2. Given a fault, determine implied values by static learning
3. Use PODEM and determine implied values using dynamic learning

Heuristic used to determine when to “learn”

1. (e.g. don’t learn implications which are trivial or already known from previous steps)

Socrates completely subsumed by SAT procedure

Recursive Learning and Graphs

- *Recursive* Socrates-style learning
  - Improvements to FAN [Kunz,Pradhan92]
  - Can learn more implications
    - Not all needed
  - Time exponential in recursion depth
    - Memory size linear
- Implication graph [Chakradhar et a. 93]
  - Construct graph of interesting implications by transitive closure
  - Efficient for large circuits

Alternatives to ATPG based on Structural Search

- Structural search like ATPG using data structure for representing circuit under test
  - First Step: Test patterns assigned at fault location to generate discrepancy between faulty and fault-free circuit
  - Second Step: Search for consistent values for all involved circuit lines such that faulty results visible at primary outputs
- Alternative solution - algebraic methods used instead of search on data structures representing circuit under test
  - Algebraic methods used to produce single equation describing all possible tests for particular fault

Algebraic Methods in ATPG

- Boolean difference
  - Boolean difference of function F w.r.t. variable *xi:*

$$dF/dx_i = F(x_1,\cdots,x_{i-1},0,x_{i+1},\cdots,x_n) \oplus F(x_1,\cdots,x_{i-1},1,x_{i+1},\cdots,x_n)$$

  - Set of test for *xi s-a-0: Xi**(*dF/dxi*) and for *xi s-a-1: Xi'**(*dF/dxi*)
    - *Xi* - function representing output of subcircuit with output at xi)
- Initial formula for Boolean difference is simplified using basic laws of Boolean difference
- In general this approach very time consuming - alternative Boolean satisfiability more promising

Boolean Satisfiability

- Boolean formula equivalent to Boolean difference
  - Solution based not on symbolic manipulations but obtained by running Boolean satisfiability
- Boolean satisfiability in finding test vectors
  - Step 1: Extraction of formula defining set of test patterns detecting given fault
  - Step 2: Running SAT algorithm to satisfy formula

CIRCUIT Satisfiability

Problem: Given a Boolean network, find a satisfying assignment to the primary inputs that makes at least primary output have value 1.

Applications:

Test pattern generation

- Combinational
- Sequential
- Delay faults

Timing analysis

Hazard detection

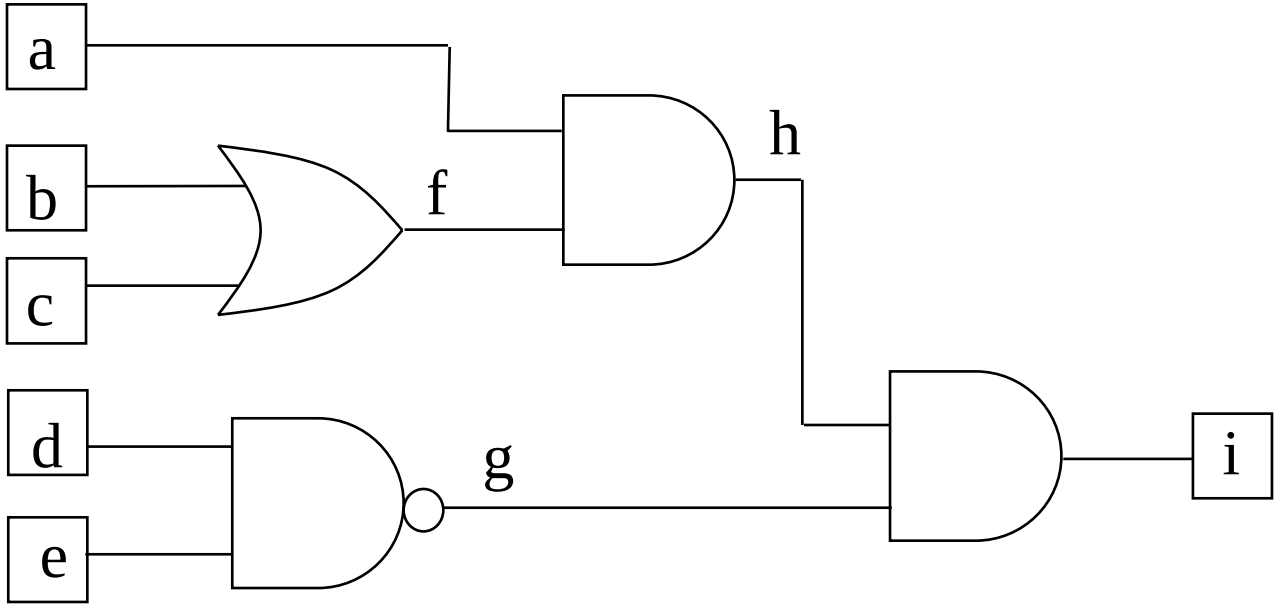
Image computation

Low power

In general, SAT is a good alternative to BDD's if

- only one solution is needed or
- a canonical form is not useful

The CIRCUIT-SAT problem



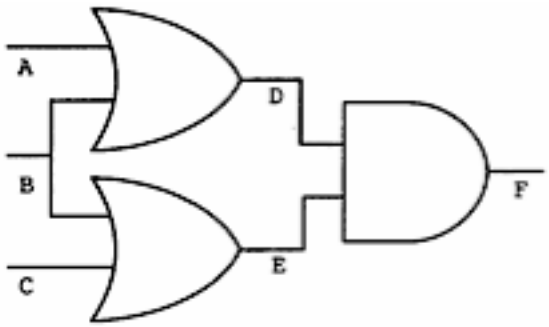
Does there exist a value assignment to the primary inputs which causes at least one primary output to assume logic value ‘1’ ?

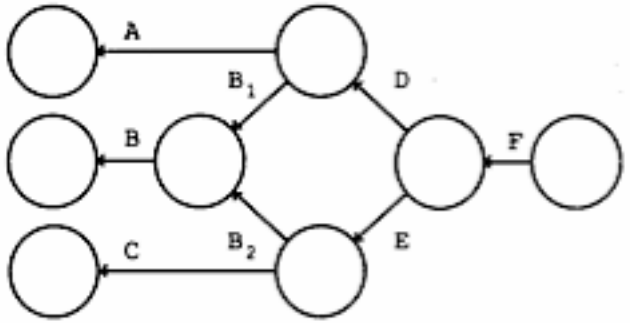
Circuit Representation

- Circuits represented in form of directed acyclic graphs similarly to structural ATPG
  - Graph nodes representing: inputs, oututs, gates, fan-outs
  - Graph edges standing for: circuit interconnects
    - Variable associated with each edge

Directed Acyclic Graphs in Circuit Representation

- Every node assigned formula representing function performed by gate or fan-out point
  - Formula at every node containing *only* variables for its incoming and outgoing edges
  - Example: Gate AND (inputs *X* and *Y*, output *Z*) associated with formula: $Z = X*Y$





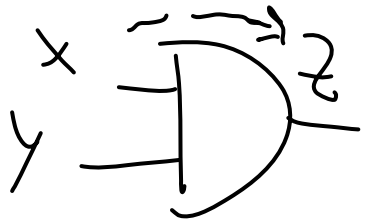
Boolean SAT

- 2SAT: finding set of values for *xi*'s satisfying equation:

$$\sum a_k b_k = 0 \quad (non - tauto\log y) \quad \prod (a_k + b_k) = 1 \quad (satisfiability)$$

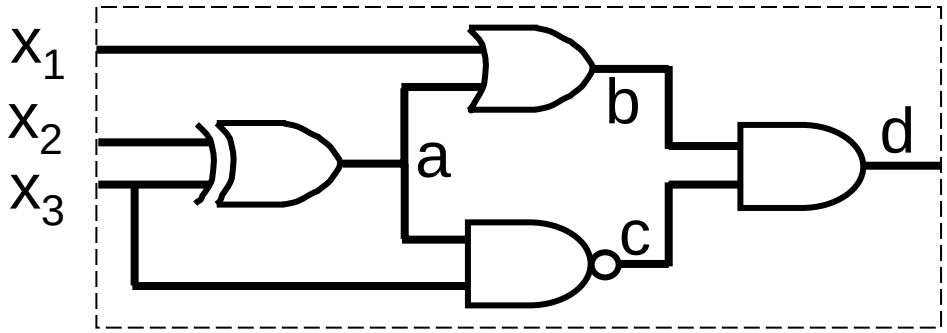
  - *ak* and *bk*: literals
  - Summation and product: Boolean OR and AND operations
  - Each term in Boolean SAT expression referred to as clause
- SAT using conjunctive normal form (CNF), I.e., product-of-sum Boolean circuit representation
- Clause standing for one sum in CFN formula
  - Clauses with one, two and three elements - unary, binary and ternary
- In 2-SAT problem: no ternary clauses
  - Solvable in polynomial time
- 3-SAT - ternary clauses present
  - Solvable in exponential time

Generating CNF Formulas



- Example: CNF for AND2 gate
  - Step 1 (I/O relation): $Z = X*Y$
  - Step 2 (Implications): Formula $P = Q$ logically equivalent to $(P \rightarrow Q)*(Q \rightarrow P)$
    - Example (AND2): $(Z->(X*Y))*(X*Y)->Z)$
  - Step 3 (CNF): Translation of all implications into disjunctions using Boolean relation: $P \rightarrow Q \Rightarrow P' + Q$
    - Example (AND2): $(Z'+X)*(Z'+Y)*(X'+Y'+Z)$
      - Formula evaluating to 1 iff values of variables consistent with AND2 truth table
      - Comparison: Disjunctive Normal Form: $(X*Y*Z) + (X'*Y*Z') + (X*Y'*Z') + (X'*Y'*Z')$

SAT Formulations for Circuit Gates



OR2:

If a = 1 then b = 1

(a -> b, I.e., a + b')

If x1 = 1 then b = 1

(x1 -> b, I.e., x1 + b')

If x1 = 1 or a = 1 then b =1

([x1 + a] -> b, I.e., [x1 + a + b'])

Hence: (a+b')(x1+b')(x1+a+b')

AND2: If b = 0 then d = 0 (b'->d' => b+d' =>(non-taut) b'd

If c = 0 then d = 0 (c'->d' => c+d' =>(non-taut) c'd)

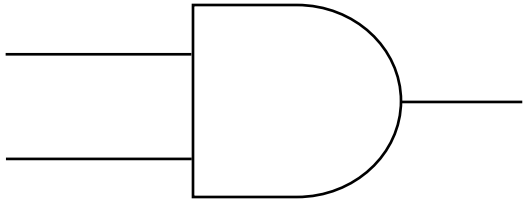
If d = 1 then b = 1 and c = 1 (d->bc => d'+bc=>(non-taut) d(bc)')

If b = 1 and c = 1 then d = 1 (bc->d => (bc)'+d=>(non-taut)bcd')

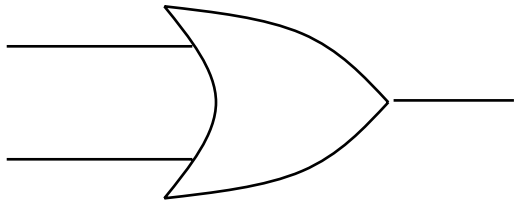
Hence: b'd+c'd+(bc)'d+bcd'=0 => b'd+c'd+bcd'=0

SAT: (b+d')(c+d')(b'+c'+d) = 1

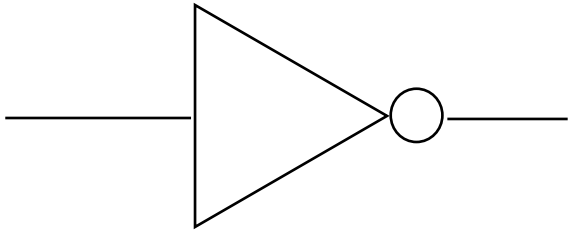
SAT Formulas for Simple Gates



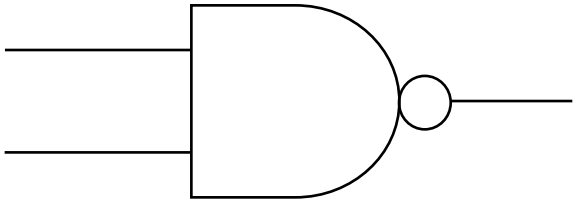
$(\overline{c} + a)(\overline{c} + b)(c + \overline{a} + \overline{b})$



$(c + \overline{a})(c + \overline{b})(\overline{c} + a + b)$



$(a + b)(\overline{a} + \overline{b})$



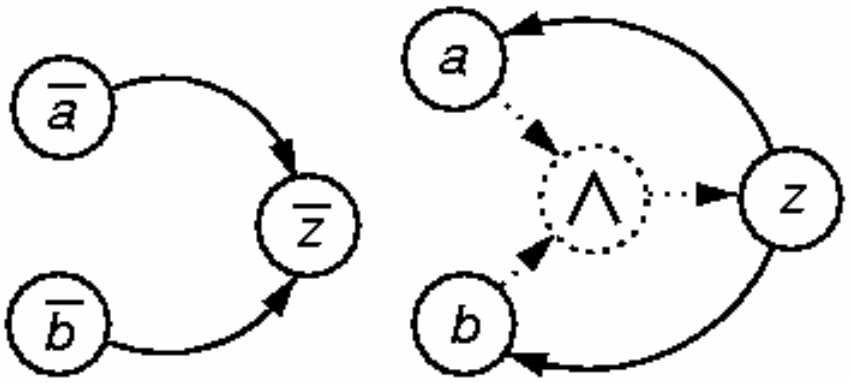
$(c + a)(c + b)(\overline{c} + \overline{a} + \overline{b})$

SAT Assignments

- Assignment of SAT variables through *implication graph*
- Graph structure
  - Node for each literal
    - Example: Boolean variable *x* represented by two nodes *x* and *x*'
  - Nodes true or false
    - For *x*=1 node *x* true, for *x*=0 node *x*' true
  - Two-variable "*if then*" clause represented by directed edge from literal expressing "*if*" condition to literal for "*then*" clause
- Graph transformed into
  - If node set true then all reachable nodes also true
    - Transitive closures determining more global signal relations in graph than other branch-and-bound search methods

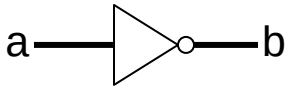
Implication Graph - AND Gate

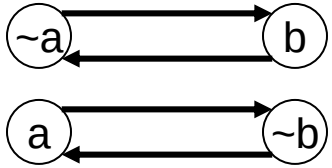
- Only binary implications (with two literals) represented by edge
- “ANDing” node (dotted lines) representing 3-SAT terms in AND gate SAT expression



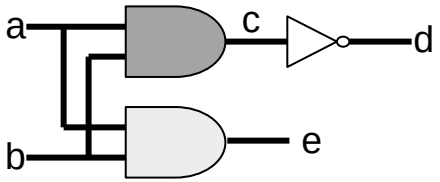
Implication Graph

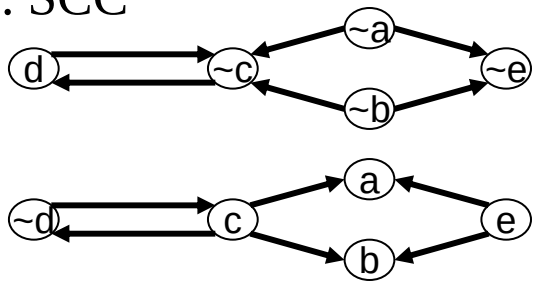
- View 2-clauses as pair of implications
  - $(a + \sim b) \Leftrightarrow (\sim a \rightarrow \sim b) \wedge (b \rightarrow a)$
  - forms implication graph
- Strongly-connected components (SCCs) are equivalent variables (inverters, buffers)





- More complex equivalences not detected.
  - Example: symmetry vs. SCC



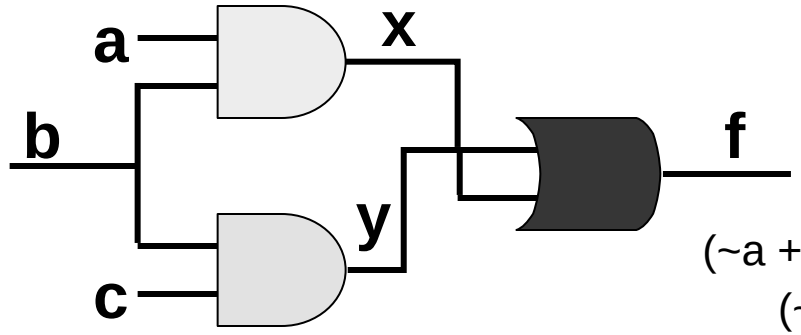


- $(\sim a+\sim b+c)(a+\sim c)(b+\sim c)(\sim a+\sim b+e)(a+\sim e)(b+\sim e)(\sim c+\sim d)(c+d)$

Non-Local Implications

- Explicit derivation of non-local implicants by examining reconvergent fan-outs
- All non-local implications of given variable listed by binding variable to some value and then noting direct implication
- All non-local implications to be added to SAT formulas for given circuit

Non-local Implications, cont.



(~a + ~b + x)(a + ~x)(b + ~x)
(~b+ ~c + y)(b + ~y)(c + ~y)
(x + y + ~f)(~x + f)(~y + f)

- Example: If b=1 then f=1, if b-0 then f=0 (discovered through analysis of circuit structure of Boolean description)

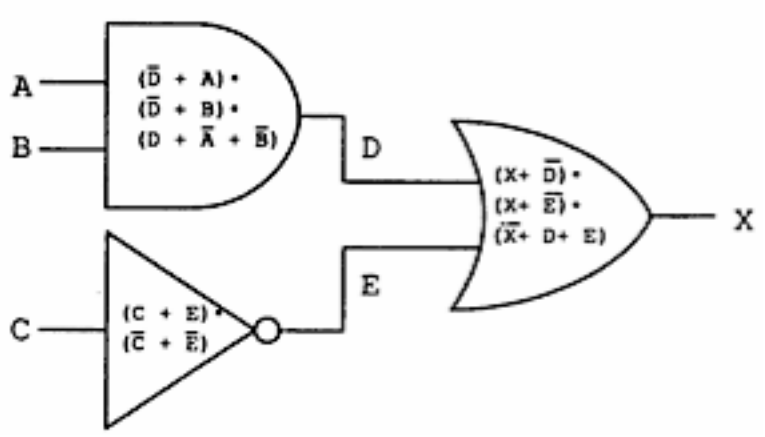
1. Find non-local implications for b:
   - Try asserting (~b)
   - (b + ~x) $\Rightarrow$ (~x), and (b + ~y) $\Rightarrow$ (~y)
   - (x + y + ~f) $\Rightarrow$ (~f)
   - Thus, (~b) $\Rightarrow$ (~f), so deduce (f) $\Rightarrow$ (b)
2. If contradiction, (e.g. f $\Rightarrow$ ~f) fix to other constant (e.g. f=0)
3. Repeat for every formula variable

Crucial for hard faults (esp. redundancies, where no test exists)

Example: Formula for Fault Free Circuits

- CNF for each gate and fan-out independently satisfied
  - Formulas for overall circuit generated starting from primary outputs and moving on DAG towards primary inputs by taking conjunction of all formulas of nodes visited so far

Formula for circuit outputs:
(X+D')(X+E')*(X'+D+E)*
**(D'+A)(D'+B)*(D+A'+B')*
**(C+E)*(C+E')*



Formula for Faulty Circuit

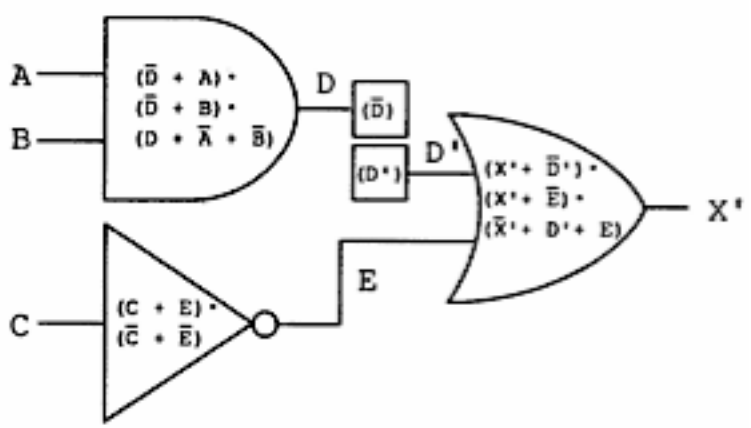
- Faulty version obtained by copying fault free circuit, renaming variables, and inserting two nodes representing disrupted connection in faulty circuit
  - Fault-free and faulty circuit of the same behavior at all remaining nodes not affected by fault
    - Only variables associated with wires lying on paths between fault and circuit output need to be renamed
- CNF for faulty circuit generated the same way as for fault free
  - Starting at fault output DAG circuit representation traversed generating conjunction of all encountered nodes

Example: Formula for Faulty Circuit

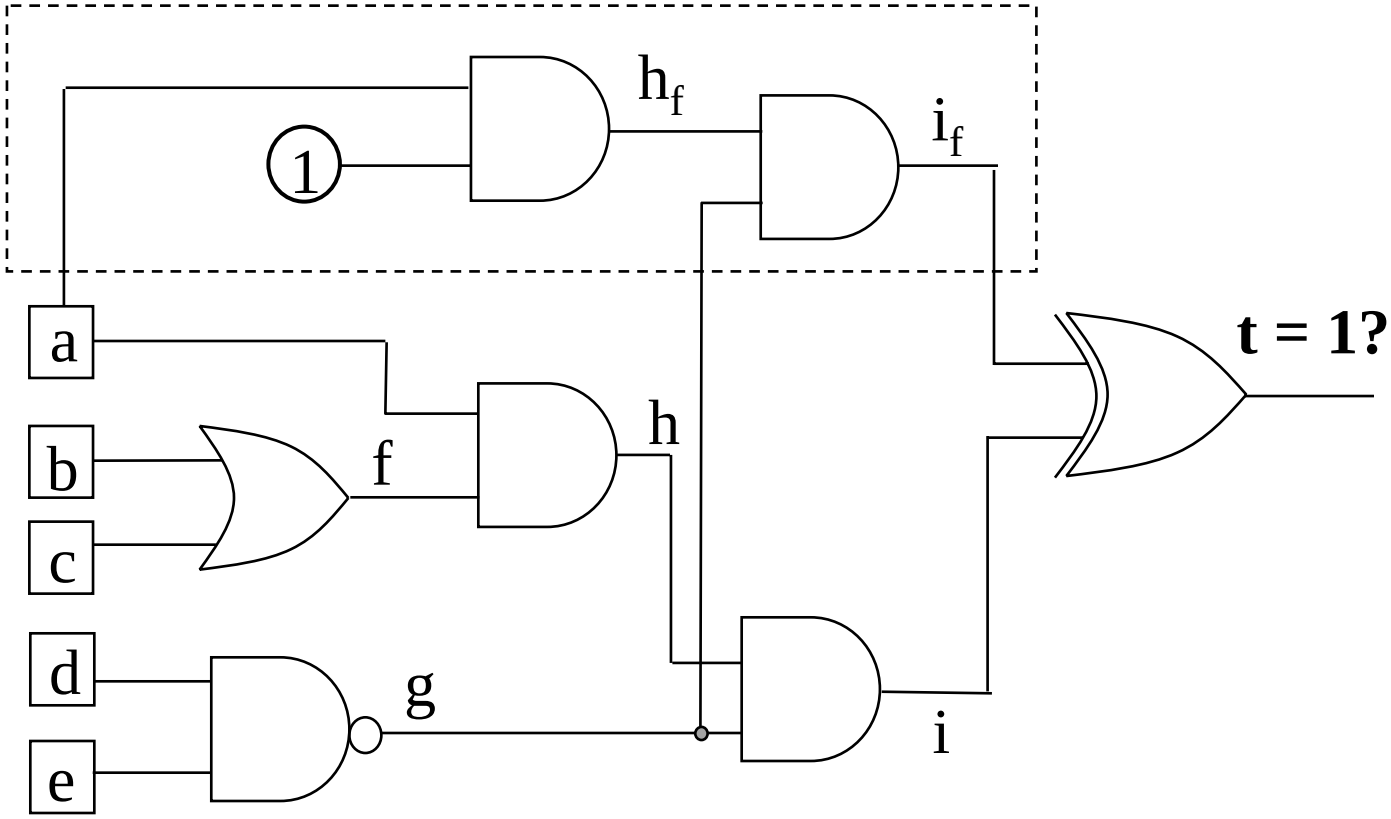
- Formula for faulty circuit with *s-a-1* fault at line *D*:

$$(X'+D_f')*(X'+E_f)*(X_f'+D'+E)*(D')*(C+E)*(C_f+E_f)$$

- Testing boils down to finding set of inputs causing faulty output to differ from fault-free
  - All possible test guaranteed if CNFs for fault-free and faulty XORed
    - CNF for xor of faulted and unfaulted outputs need to be added



ATPG as CIRCUIT-SAT Problem



Testing by 3-SAT

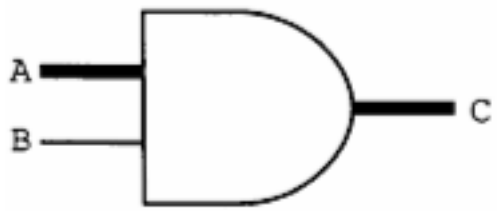
- Automatic Test Pattern Generation (ATPG) [Larrabee, 1992]
- Algorithm described so far working well
  - Several other heuristics added to improve performance
- Heuristics based on APTG D-Algorithm allowing speed up in fault propagation
- CNF: conditions for good circuit, fault excitation and propagation
- Clauses :
  - *Good Circuit*: All nodes - correct operation
  - *Faulty Circuit*: Fault fan-out cone
  - *Active*: Fault activation conditions
  - *Goal*: Observation conditions

Active Clauses - Definition

- Based on observations from D-Algorithm
  - At least one active path from fault location to primary outputs for fault to successfully reach output
    - Discrepancy between faulty and fault free circuit on every line of active path
    - Active line - each line member of active path
    - Note: each active line must have discrepancy, not all lines with discrepancies belonging marked as active wires

Determining Active Path

- Clauses to be added to describe active path
  - *Active variable* allocated for each path lying between fault location and primary outputs
  - Several clauses allocated for each gate lying between fault location and primary outputs
    - Clauses to ensure that:
      - If input to single output node active then output also active
      - If input to multi-output gate active then one of outputs also active



: If A is active, C must be active: $(\overline{Act_A} + Act_C)$

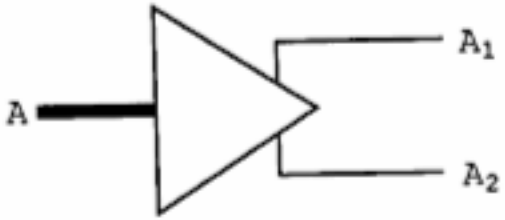


Figure 4: If A is active, either A_1 or A_2 must be active: $(\overline{Act_A} + Act_{A_1} + Act_{A_2})$

Determining Active Path, cont.

- Additional clauses guaranteeing all lines on active path to have different faulted and unfaulted values
  - Example: variables ActD and ActX allocated
    - Following clauses added to set of active clauses:

$$(\overline{Act_D} + D + D_f),(\overline{Act_D} + \overline{D} + \overline{D_f}),(\overline{Act_X} + X + X_f),(\overline{Act_X} + \overline{X} + \overline{X_f})$$

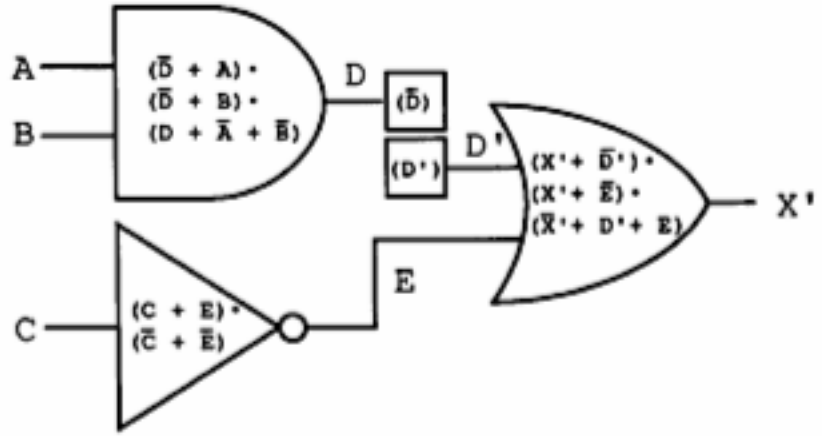


Figure 2: The Formula for the output is $(X' + \overline{D'}) \cdot (X' + \overline{E}) \cdot (\overline{X'} + D' + E) \cdot (D') \cdot (C + E) \cdot (\overline{C} + \overline{E})$.

Active Clauses

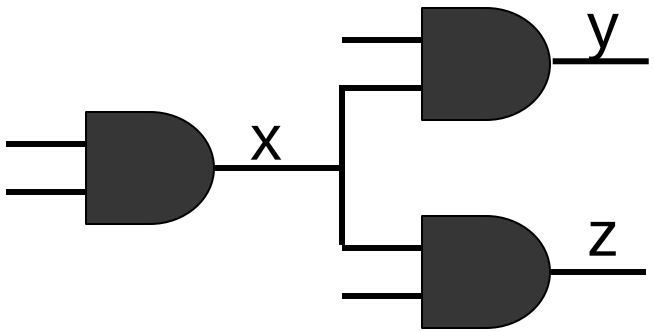
Problem: Good/faulty circuits related only at I/Os, slow to find contradictions

Solution: active clauses define relationships for internal nodes (Larrabee 1990)

- Active variable x_a is true if net x differs in good and faulty network. Here, x_g refers to signal x in good circuit and x_f to x in the faulty circuit:

$$(\sim x_a + x_g + x_f)(\sim x_a + \sim x_g + \sim x_f)$$

- If gate is active, we require that some fanout must be active



$$(\sim x_a + y_a + z_a)$$