EEDG/CE 6303: Testing and Testable Design

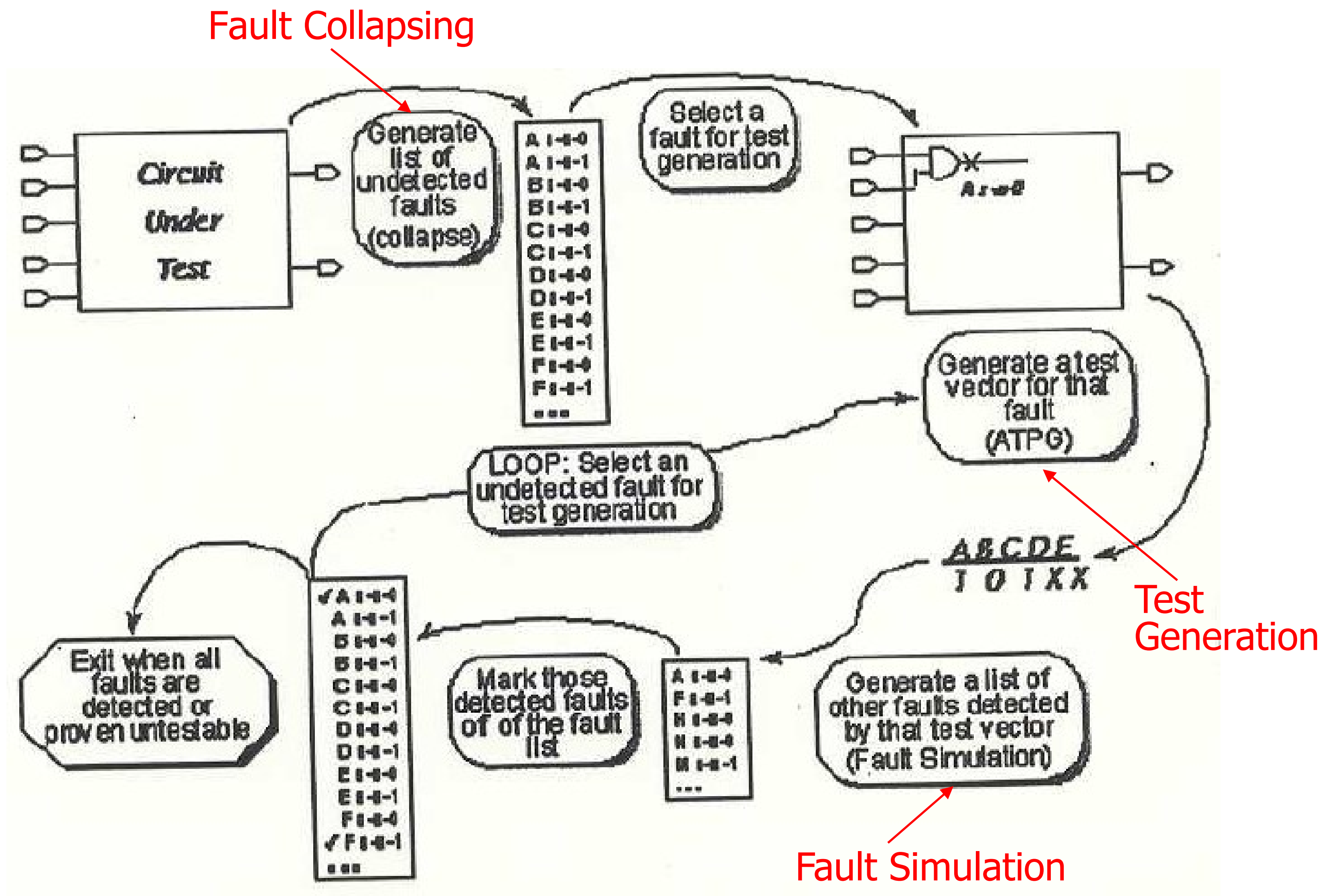
Mehrdad Nourani

Dept. of ECE
Univ. of Texas at Dallas

Session 04

Fault Simulation

Fault Analysis System (Review)


Fault Collapsing
Circuit Under Test
Generate list of undetected faults (collapse)
A s-a-0
A s-a-1
B s-a-0
B s-a-1
C s-a-0
C s-a-1
D s-a-0
D s-a-1
E s-a-0
E s-a-1
F s-a-0
F s-a-1
...
Select a fault for test generation
A s-a-0
Generate a test vector for that fault (ATPG)
Test Generation
LOOP: Select an undetected fault for test generation
ABCDE
10 1XX
Generate a list of other faults detected by that test vector (Fault Simulation)
Fault Simulation
A s-a-0
F s-a-1
H s-a-0
N s-a-0
M s-a-1
...
Mark those detected faults of of the fault list
✓A s-a-0
A s-a-1
B s-a-0
B s-a-1
C s-a-0
C s-a-1
D s-a-0
D s-a-1
E s-a-0
E s-a-1
F s-a-0
✓F s-a-1
...
Exit when all faults are detected or proven untestable

Methods: 1. Parallel, 2. Deductive, 3. Concurrent, 4. Critical Path Tracing

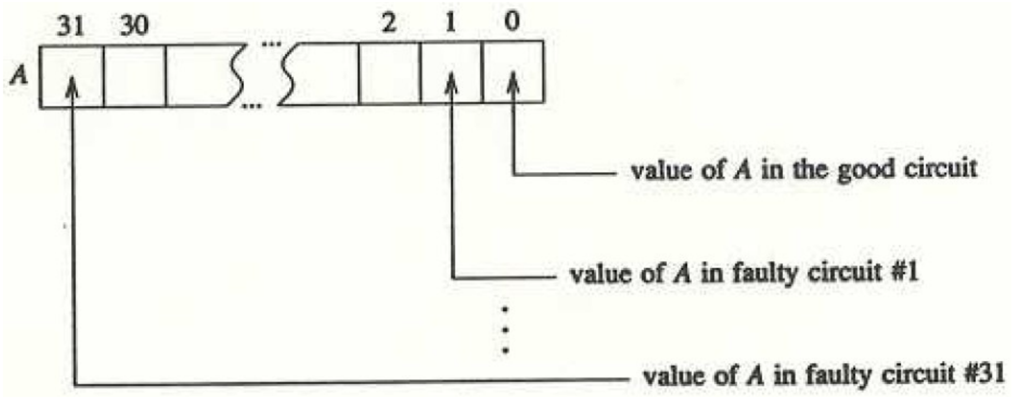
Fault Simulation

- **Goal**: to determine the list of faults in a CUT (Circuit Under Test) and to simulate the fault-free and faulty circuits efficiently to determine if their outputs are different
- Five tasks involved:
  1. Fault-free (good) circuit simulation
  2. Fault specification (fault list generation and collapsing)
  3. Fault insertion (fault selection to be simulated and tracing the presence of the faults)
  4. Fault effect generation and propagation
  5. Fault detection and discarding

Parallel Fault Simulation

Parallel Fault Simulation

- The good circuit and a fixed number (e.g. W) of faulty circuits are simultaneously simulated.
- For a group of F faults, $\lceil F/W \rceil$ passes are required for fault simulation.
- The values of lines in the circuit are packed into the words of the host computer (i.e. in the same memory location). Example:
  - —Word length = 32
  - —2-valued logic
  - —W = 32 – 1 = 31
  - —1 bit good value and 31 bits bad values

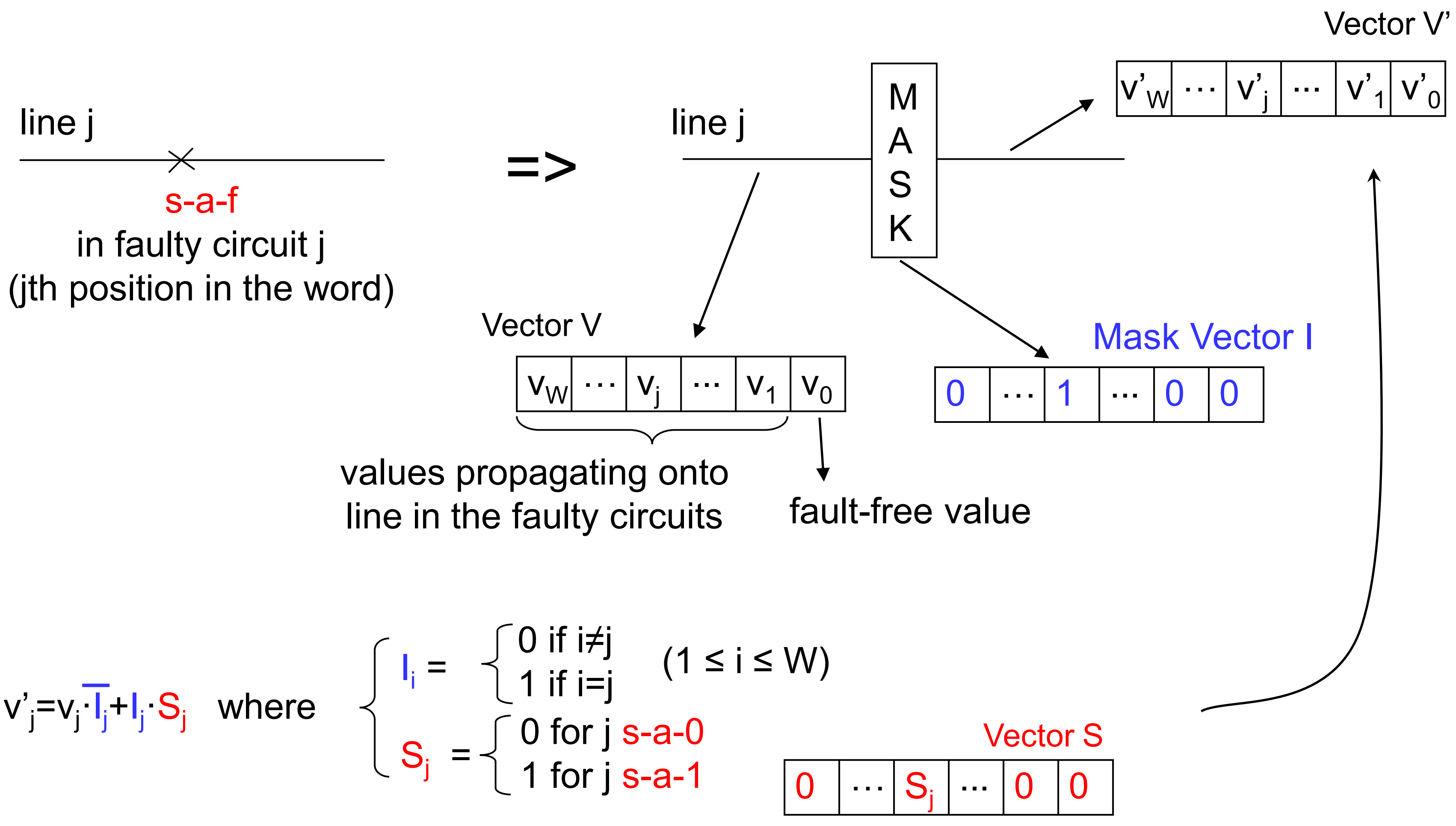

31
30
2
1
0
A
value of A in the good circuit
value of A in faulty circuit #1
value of A in faulty circuit #31

Parallel Fault Simulation (cont’d)

- Bitwise logical operations (AND, OR, XOR, NOT…) are used to simulate the gates of the circuit-under-test for all W+1 values.
- Sequential elements (e.g. FFs) can be represented by their Boolean (characteristic) expression and then bitwise computation is performed.

Parallel Fault Simulation (cont'd)

- Fault insertion is also handled by bitwise operations using "masks".



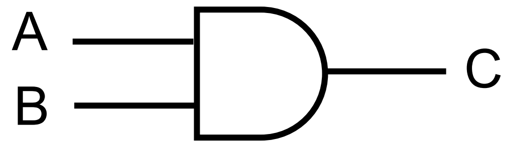
line j
s-a-f
in faulty circuit j
(jth position in the word)
=>
line j
MASK
Vector V'
v'_W … v'_j … v'_1 v'_0
Vector V
v_W … v_j … v_1 v_0
values propagating onto line in the faulty circuits
fault-free value
Mask Vector I
0 … 1 … 0 0
Vector S
0 … S_j … 0 0

$$v'_j = v_j \cdot \overline{I_j} + I_j \cdot S_j \quad \text{where} \quad \begin{cases} I_i = \begin{cases} 0 \text{ if } i \neq j \\ 1 \text{ if } i = j \end{cases} & (1 \leq i \leq W) \\ S_j = \begin{cases} 0 \text{ for } j \text{ s-a-0} \\ 1 \text{ for } j \text{ s-a-1} \end{cases} \end{cases}$$

Parallel Fault Simulation (cont'd)

- Example: assume the word-length is 5 (1 fault-free and 4 faulty values).

Faults	Bit Position
Fault-free	0
A s-a-0	1
B s-a-1	2
C s-a-0	3
C s-a-1	4



Lines	I Mask	S Stuck-At Values
A	I_A= [0 0 0 1 0]	S_A= [0 0 0 0 0]
B	I_B= [0 0 1 0 0]	S_B= [0 0 1 0 0]
C	I_C= [1 1 0 0 0]	S_C= [1 0 0 0 0]

- Assume values before fault insertion are:
  - A ="1"= [1 1 1 1 1]
  - B ="0"= [0 0 0 0 0]

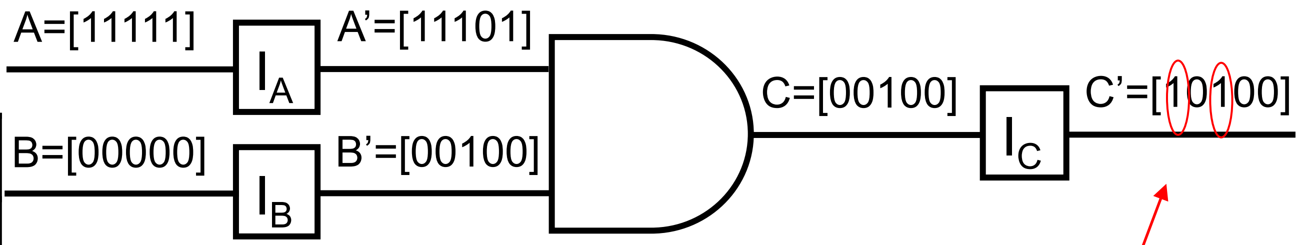
Parallel Fault Simulation (cont'd)

$A' = A \cdot \overline{I_A} + I_A \cdot S_A$

$B' = B \cdot \overline{I_B} + I_B \cdot S_B$

$C' = C \cdot \overline{I_C} + I_C \cdot S_C$

Lines	I Mask	S Stuck-At Values
A	I_A= [0 0 0 1 0]	S_A= [0 0 0 0 0]
B	I_B= [0 0 1 0 0]	S_B= [0 0 1 0 0]
C	I_C= [1 1 0 0 0]	S_C= [1 0 0 0 0]



- For example: [00100][00111]+[11000][10000]=[10100]
- The resulting word shows that the faults corresponding to positions 2 and 4 (i.e. B s-a-1 and C s-a-1) which are different from the fault-free value (0) have been detected.

Parallel Fault Simulation (cont'd)

- There is a tradeoff between memory consumption and the running time. For the previous example (a 2-input AND gate):
  - Serial simulation (one fault at a time) needs 3 bits and 5 passes.
  - Parallel simulation needs around 6x5=30 bits and 1 pass.

Deductive Fault Simulation

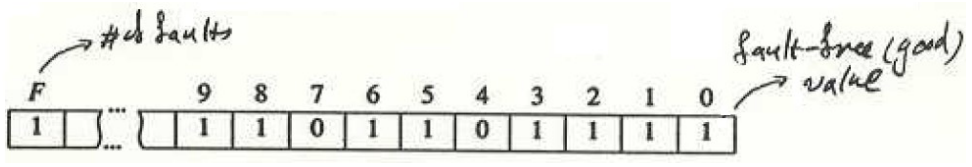
Deductive Fault Simulation

- Explicitly simulates the behavior of the good circuit only.
- Simultaneously deduces the behavior of all faulty circuits (all faults that are detected at any time) from the good circuit.
- Only one pass through the circuit is needed, although it takes a long time to make this pass.
- Requires less memory than parallel simulation.
- For each line i, a fault list L_i that shows the complement of the good value at line i is calculated:

  L_i ={all faults that cause $v_f \neq v$ at the current simulation time}

Deductive Fault Simulation (cont'd)

- Difference between fault effect representation (for line i):
  - Parallel:



  - Deductive: $L_i = \{4,7\}$
- In deductive fault simulation, waste of memory is minimized by keeping the bit positions of those faults that are **different** from the good value.
- A deductive fault simulator propagates:
  - Logic events (changes in signal values)
  - Fault-list events (occurs when a fault list changes, i.e., a fault is either added or deleted from the list)

Two-Valued Deductive Simulation

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

- Rules for fault-effect propagation:
  - I = set of inputs of a gate z (output is z)
  - c,i = controlling and inversion values for gate z
  - $C \subseteq I$ = set of inputs with value c
- (i) if $C=\Phi$ then:

$$L_z = \left\{ \bigcup_{j \in I} L_j \right\} \cup \{z \text{ s-a-}(c \oplus i)\}$$

i.e. if no input has the controlling value c, any fault effect on an input propagates to the output.

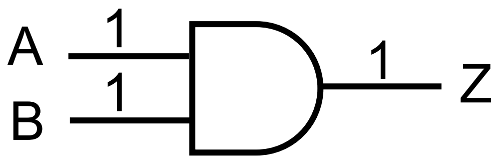
- (ii) else:

$$L_z = \left(\left\{ \bigcap_{j \in C} L_j \right\} - \left\{ \bigcup_{j \in I-C} L_j \right\} \right) \cup \{z \text{ s-a-}(\bar{c} \oplus i)\}$$

i.e. if some inputs have value c, only a fault effect that affects all the inputs at c without affecting any of the inputs at $\bar{c}$ propagates to the output.

Two-Valued Deductive Simulation (cont'd)

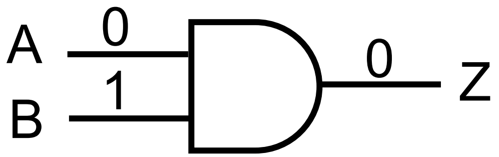
- Example for (i):



$$L_Z = L_A \cup L_B \cup \{Z \text{ s-a-}0\}$$

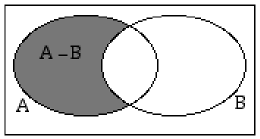
if A=B=1, then any fault that causes a "0" on A or B will cause Z to be erroneously "0".

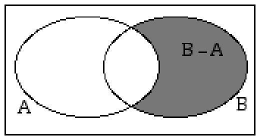
- Example for (ii):



$$L_Z = (L_A - L_B) \cup \{Z \text{ s-a-}1\}$$

if A=0,B=1, then any fault that causes A to be "1" without changing B, will cause Z to be in error (i.e. $L_A \cap \overline{L_B} = L_A - L_B$; $\overline{L_B}$ = set of faults not in L_B)



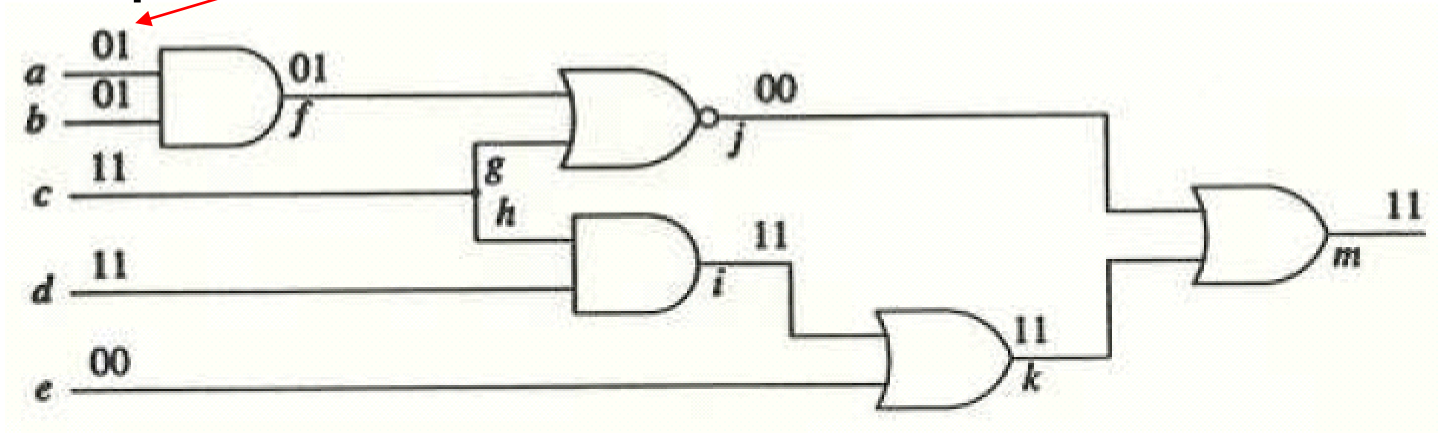


Example: $\{a, b, c\} - \{a, c, e\} = \{b\}$

Two-Valued Deductive Simulation (cont'd)

- Example:

Two patterns applied



- Suppose, after fault collapsing:

$$F = \{a_0, a_1, b_1, c_0, c_1, d_1, e_0, g_0, h_0, h_1\}$$

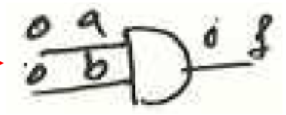
where $\alpha_v \equiv \alpha$ s-a-v

Two-Valued Deductive Simulation (cont'd)

- Fault List: $F=\{a_0,a_1,b_1,c_0,c_1,d_1,e_0,g_0,h_0,h_1\}$
- First pattern to apply: abcde = 00110

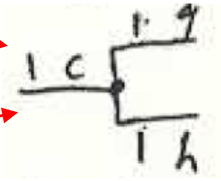
$$L_a=\{a_1\};L_b=\{b_1\};L_c=\{c_0\};L_d=\phi;L_e=\phi$$

$$L_f=L_a\cap L_b=\phi$$

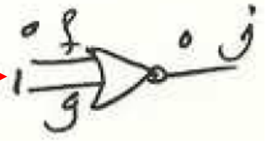


$$L_g=L_c\cup\{g_0\}=\{c_0,g_0\}$$

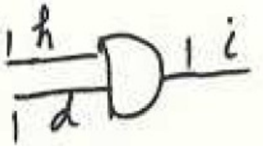
$$L_h=L_c\cup\{h_0\}=\{c_0,h_0\}$$



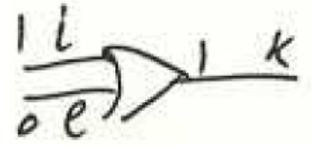
$$L_j=L_g-L_f=\{c_0,g_0\}$$



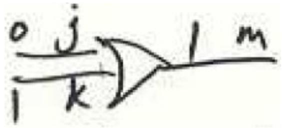
$$L_i=L_d\cup L_h=\{c_0,h_0\}$$



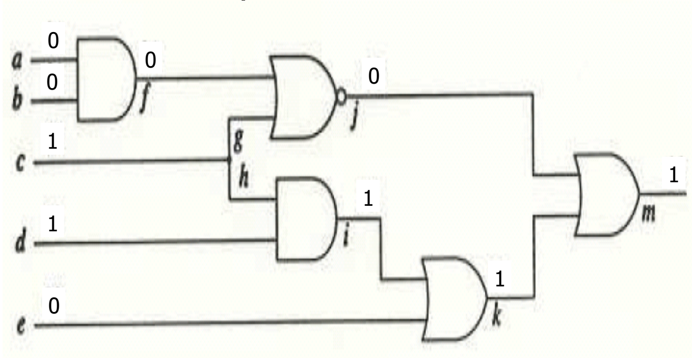
$$L_k=L_i-L_e=\{c_0,h_0\}$$



$$L_m=L_k-L_j=\{h_0\}$$



Consider pattern: abcde = 00110



- It means h_0 is detected by this pattern, drop it from the list and continue.

Two-Valued Deductive Simulation (cont'd)

- Second pattern to apply: abcde = 11110
- The process should be repeated similarly. At the end of process, we will get $L_m=L_k-L_j=\{c_0\}$ which means the pattern will detect c_0. The fault c_0, then, can be removed from the list and the process continues if there are more patterns to apply.

Concurrent Fault Simulation

Concurrent Fault Simulation

- It takes advantage of the fact that most of the values in a faulty circuit are the same as their corresponding values in the fault-free circuit.
- It simulates only those elements that are different in fault and fault-free circuits.
- It needs one pass through the circuit.
- It's more efficient than parallel in terms of run time but less efficient in terms of memory.

Concurrent Fault Simulation (cont.)

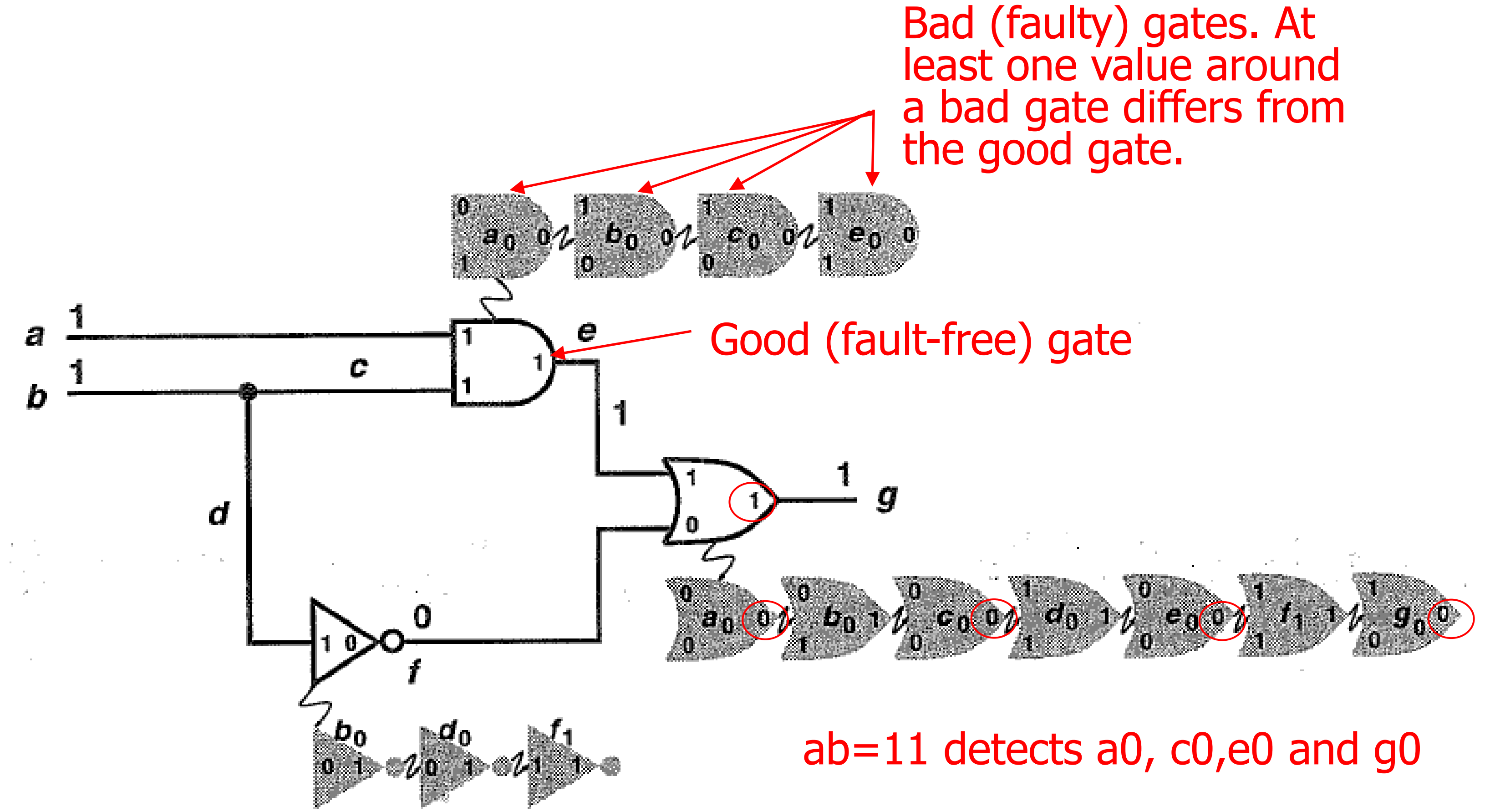
- Circuit Representation
  - One complete copy of fault-free circuit
  - A copy of those gates that have at least one of these conditions:

    1. fault f is local to the gate, i.e. it is associated with an input or output of the gate

    2. The value implied at least one input or output of the gate is different from that implied at the corresponding line in the fault-free circuit
- Each version of the circuit is simulated concurrently. In the process, faulty gates can be added or removed.

Concurrent Fault Simulation (cont.)

- Example (consider all s-a-0 and s-a-1 faults on all lines a through g) – No fault collapsing applied here:
  - $F=\{a_0,a_1,b_0,b_1,c_0,c_1,d_0,d_1,e_0,e_1,f_0,f_1,g_0,g_1\}$

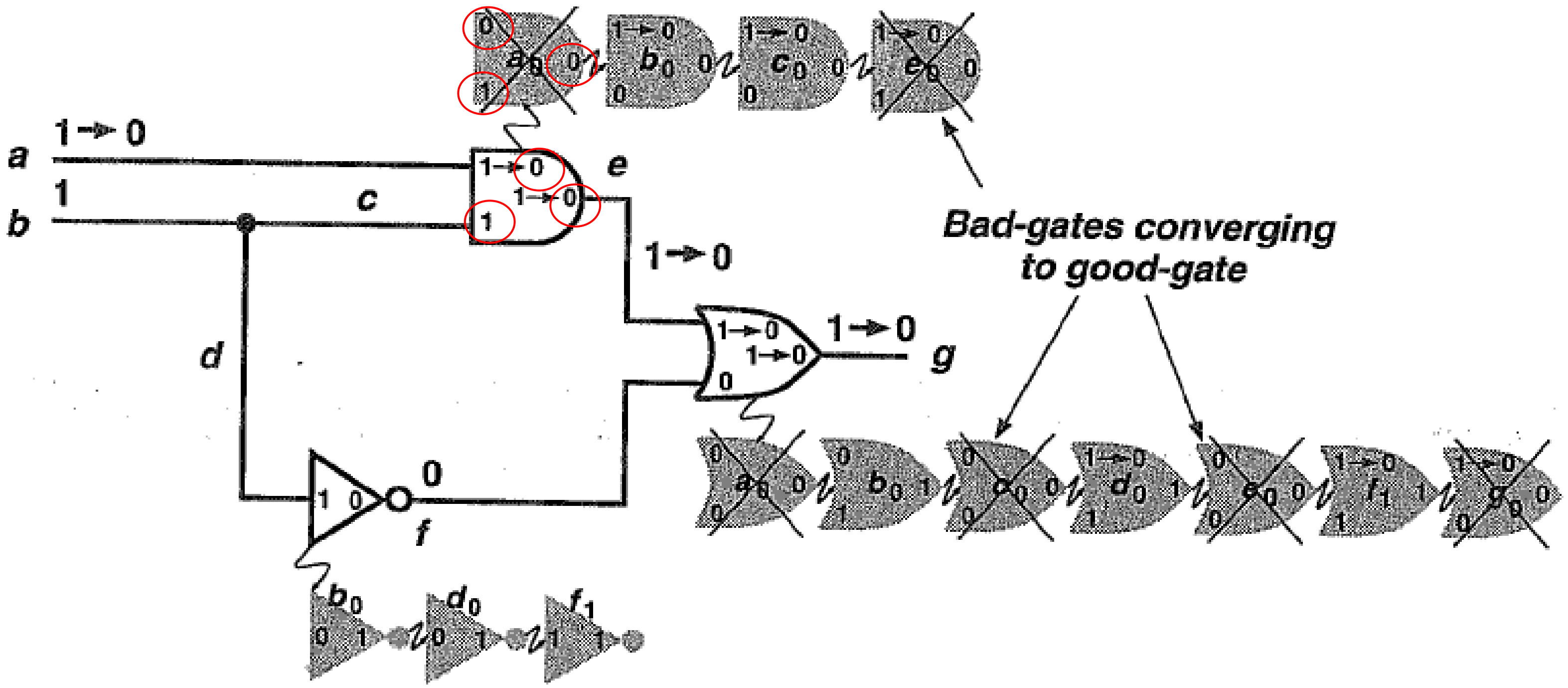


Bad (faulty) gates. At least one value around a bad gate differs from the good gate.
Good (fault-free) gate
a
b
c
d
e
f
g
1
0
a0
b0
c0
d0
e0
f1
g0

ab=11 detects a0, c0,e0 and g0

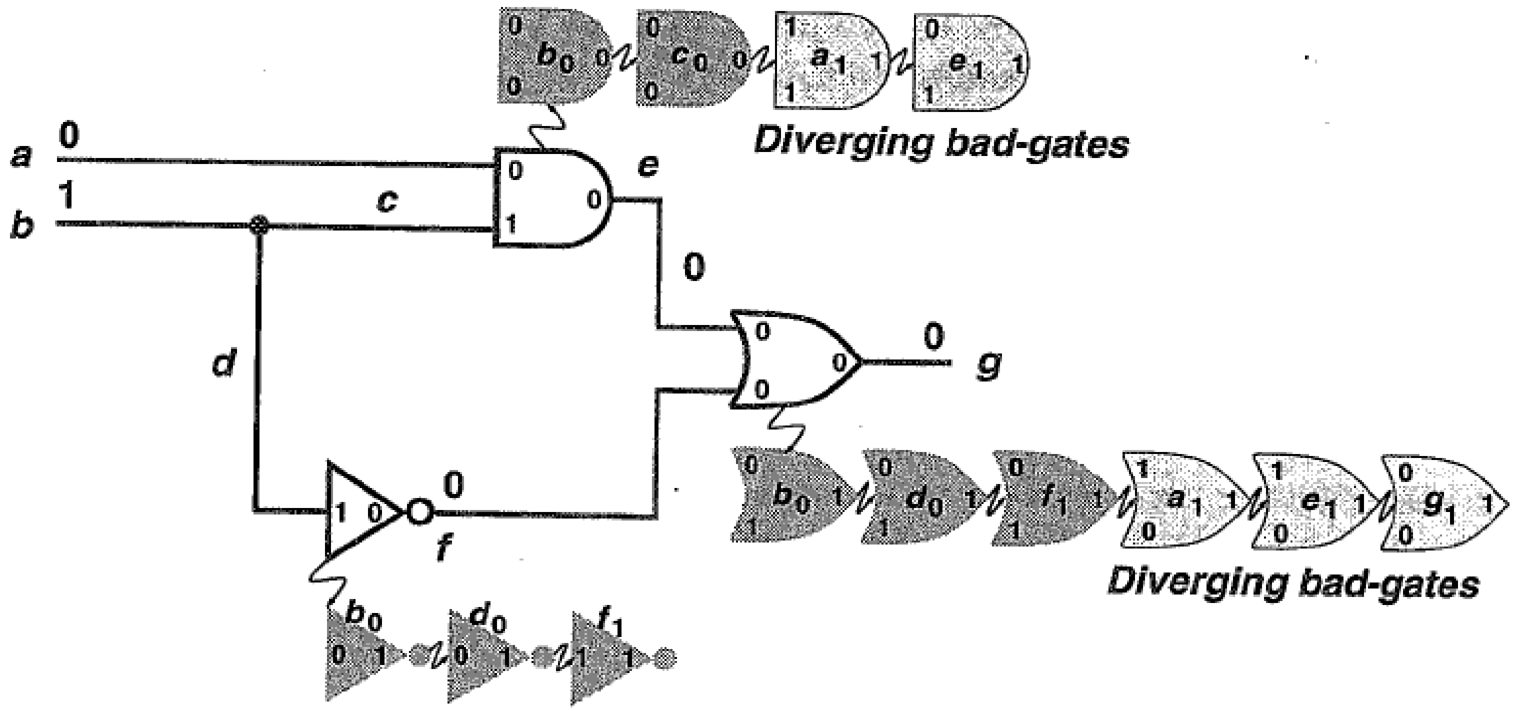
Concurrent Fault Simulation (cont.)

- In event processing
  - $F=\{a_0,a_1,b_0,b_1,c_0,c_1,d_0,d_1,e_0,e_1,f_0,f_1,g_0,g_1\}$
  - Remove those bad gates whose inputs and outputs become the same as a fault-free gate. They can no longer be detected. Bad gates are said to be **converging** to good gates.



Concurrent Fault Simulation (cont.)

- In event processing
  - F={a_0,a_1,b_0,b_1,c_0,c_1,d_0,d_1,e_0,e_1,f_0,f_1,g_0,g_1}
  - Add new bad gates whose inputs and outputs differ from the good gate. They are potentially detectable faults. This is called bad gates **divergence**.



Critical Path Tracing

Critical Path Tracing

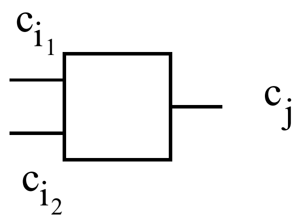
- Efficient since it implicitly targets all faults in a circuit in a single pass
- However, in its basic form applicable only to fanout-free circuits, i.e., circuits with no fanout systems
- Since most practical circuits have fanouts
  - —Circuits are partitioned into fanout-free regions
  - —Critical path tracing applied within each fanout-free region
  - —Additional steps are needed to check criticality of the paths across fanout-free regions

Critical Path Tracing (cont'd)

- Key concepts
  - Given a vector P = (p_1, p_2,..., p_n), assume fault-free circuit simulation has been performed to compute the value implied at each line by P
    - An input c_{i_l} of a gate G is said to be **sensitive** at its output c_j, if complementing the values at c_{i_l} (without changing values at any of G's other inputs) will complement the value at c_j
      - Denoted by $sens(c_{i_l}) = \begin{cases} 1, & \text{if } c_{i_l} \text{ is sensitive for the vector,} \\ 0, & \text{otherwise} \end{cases}$
      - Value at c_{i_l} may be complemented due to an appropriate sutck-at fault at c_{il} or due to one in a line in its transitive fanin
    - A line c_i in a circuit C is said to be **critical** for vector P, if P detects a SAϖ Fault at c_i, when P implies a value w at c_i
      - Denoted as $Cr(c_{i_l}) = \begin{cases} 1, & \text{if vector } P \text{ detects the SA}\overline{w} \text{ fault at } c_i, \\ 0, & \text{otherwise} \end{cases}$

Critical Path Tracing (cont'd)

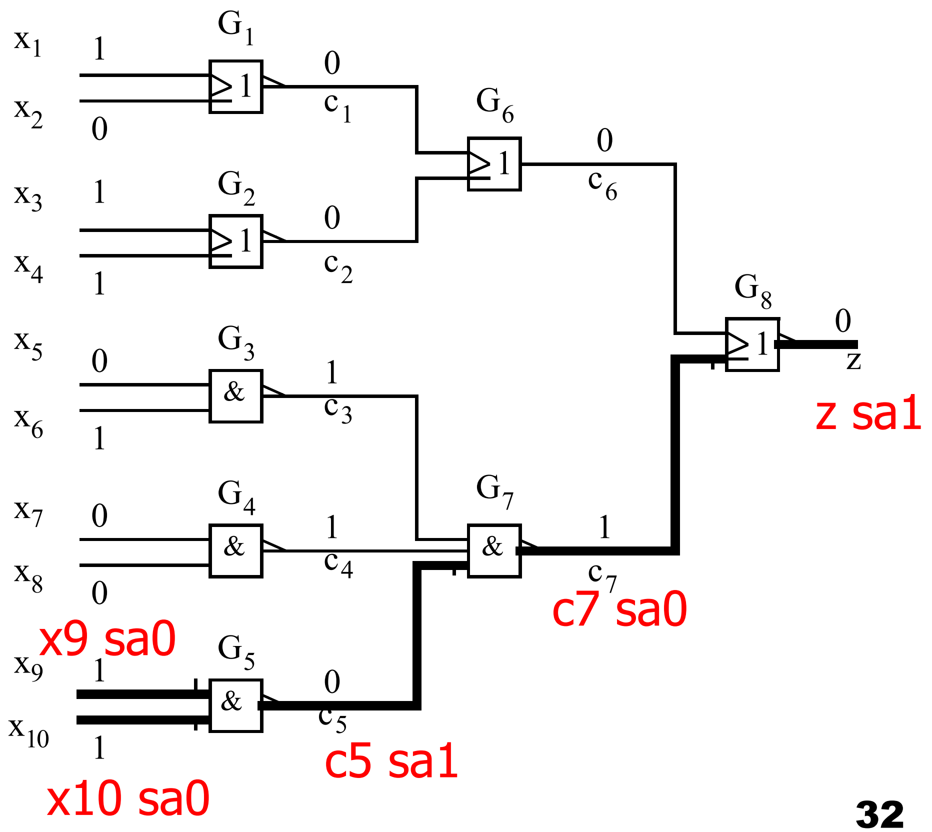
- In a fanout-free circuit, the effect of a single stuck-at fault propagates to no more than **one input** of any gate
- Due to the above property, we can use the notion of sensitivity to compute criticality as



  - (c_{i_1} is critical) if and only if [(c_j is critical) and (c_{i_1} is sensitive at c_j)]
  - That is, $Cr(c_{i_1}) = sens(c_{i_1})Cr(c_j)$
- This property can be used iteratively which traversing circuit lines from the output to inputs

Critical Path Tracing (cont'd)

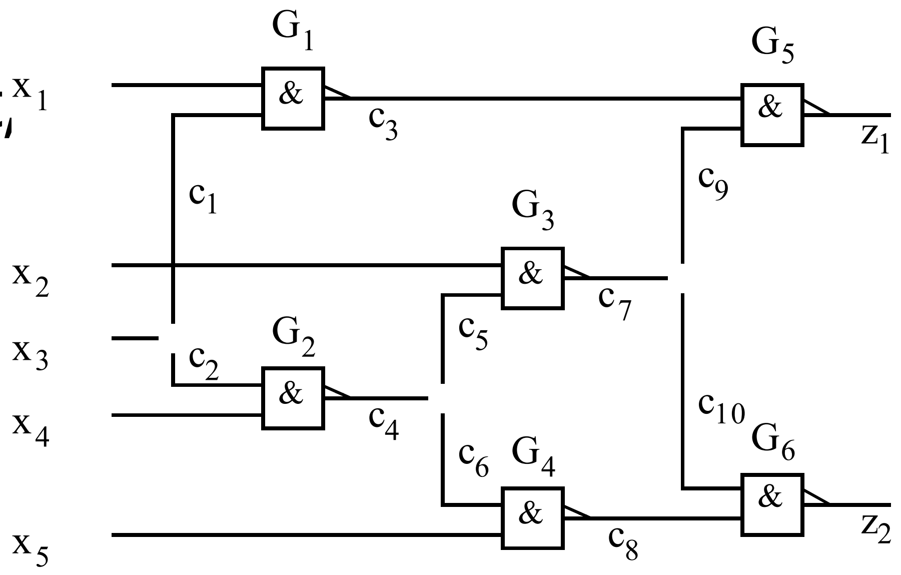
- Perform fault-free circuit simulation
- Compute sensitivity of each input of each gate
- Mark the output as critical
  - —Traversing the circuit lines from output to inputs, compute criticality of each line



Faults Detected →

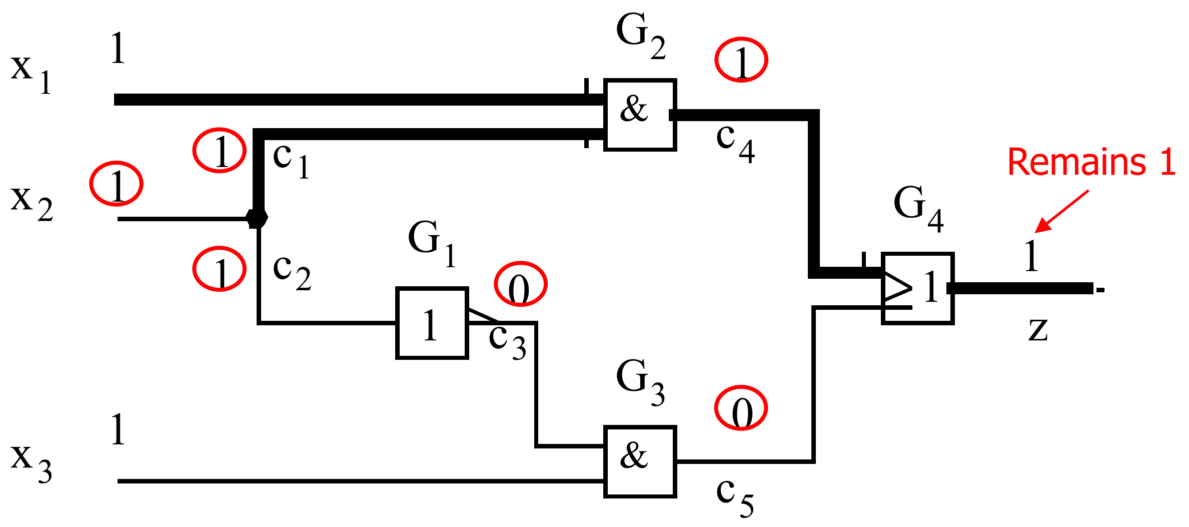
Critical Path Tracing (cont'd)

- A circuit with fanouts can be partitioned into **fanout-free regions** (FFRs) by disconnecting each stem from its branches
  - Each connected sub-circuit is an FFR
  - Each FFR can be identified by its output, which is either a primary output or a stem of a fanout system
  - Hence, we will refer to an FFR by its output, as FFR(x_3), FFR(c_4), FFR(c_7), FFR(z_1), and FFR(z_2)



Critical Path Tracing (cont'd)

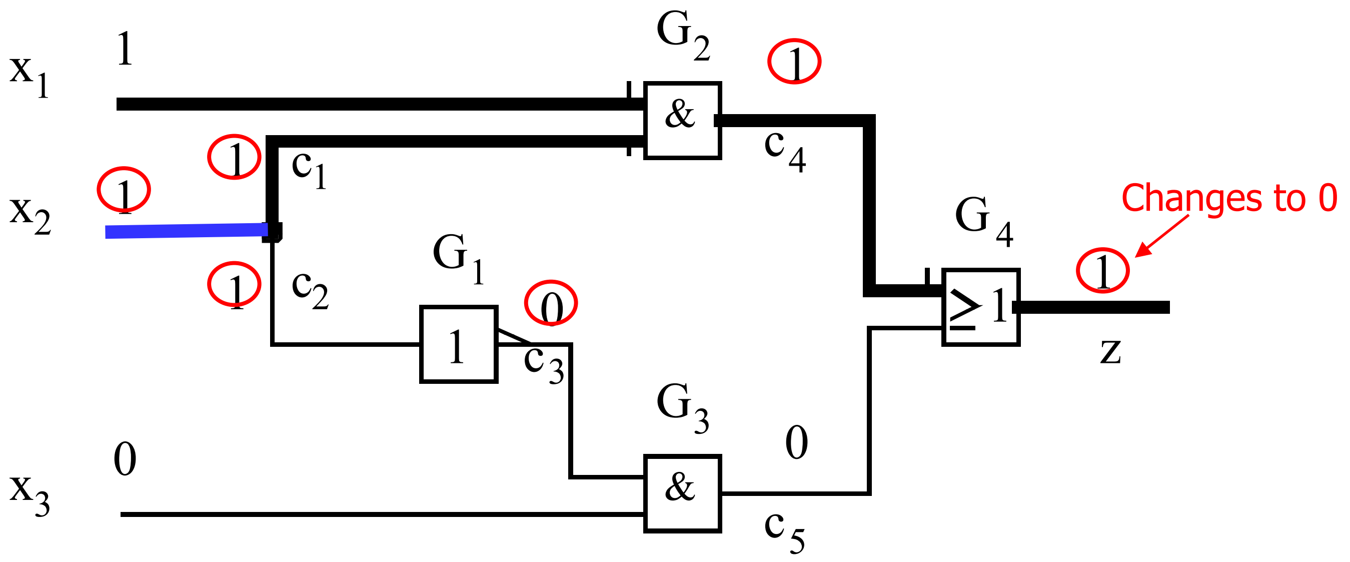
- Difficulty with applying critical path tracing to circuits with fanouts
  - This circuit can be partitioned into two FFRs: FFR(z) and FFR(x_2)



  - Critical path tracing on FFR(z) for this vector identifies x_1, c_1, c_4, and z as critical
  - Even though c_1 is critical, explicit fault simulation shows that **the stem x_2 is not critical**

Critical Path Tracing (cont'd)

- For this vector, critical path tracing of FFR(z) again identifies x_1, c_1, c_4, and z as critical



- However, in this case, **the stem x_2 is critical**
- Hence, no simple universally applicable relation exists between criticality of a stem and those of its branches

Critical Path Tracing (cont'd)

- Above examples show that no universally applicable relation exists between the criticality of a stem and those of its branches
- To obtain a complete fault simulation
  - Critical path tracing can be used within each FFR
  - Explicit fault simulation must be performed for faults at a stem to determine its criticality
  - The results of above two steps can be combined to compute the criticality of all lines in an arbitrary circuit

Critical Path Tracing (cont'd)

- Since explicit fault simulation is more expensive than critical path tracing, extensive explicit fault simulation increases the run time complexity of such a fault simulation

- Key question: How to reduce the amount of explicit fault simulation required?

Critical Path Tracing (cont'd)

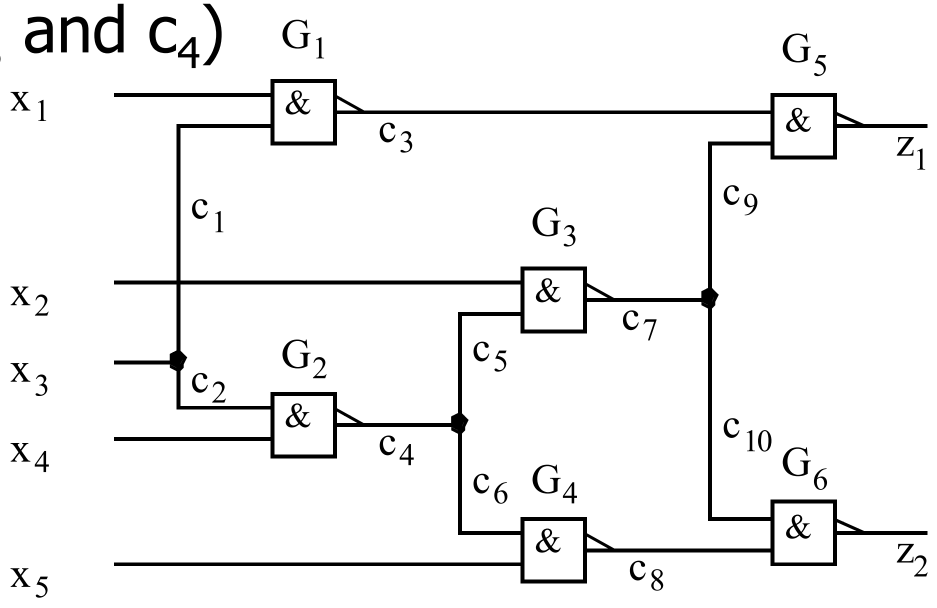
- Let Trfan(c) be the set of primary outputs in transitive fanout (fanout of fanout of …) of line c
- A branch c_{j_l} of a fanout system with stem c_i and branches $c_{j_1}, c_{j_2}, \ldots, c_{j_\beta}$ is **independent** if

$$Trfan(c_{j_l}) \bigcap \left[\bigcup_{\gamma=1,2,\ldots,\beta;\gamma\neq l} Trfan(c_{j_\gamma}) \right] = \{\}$$

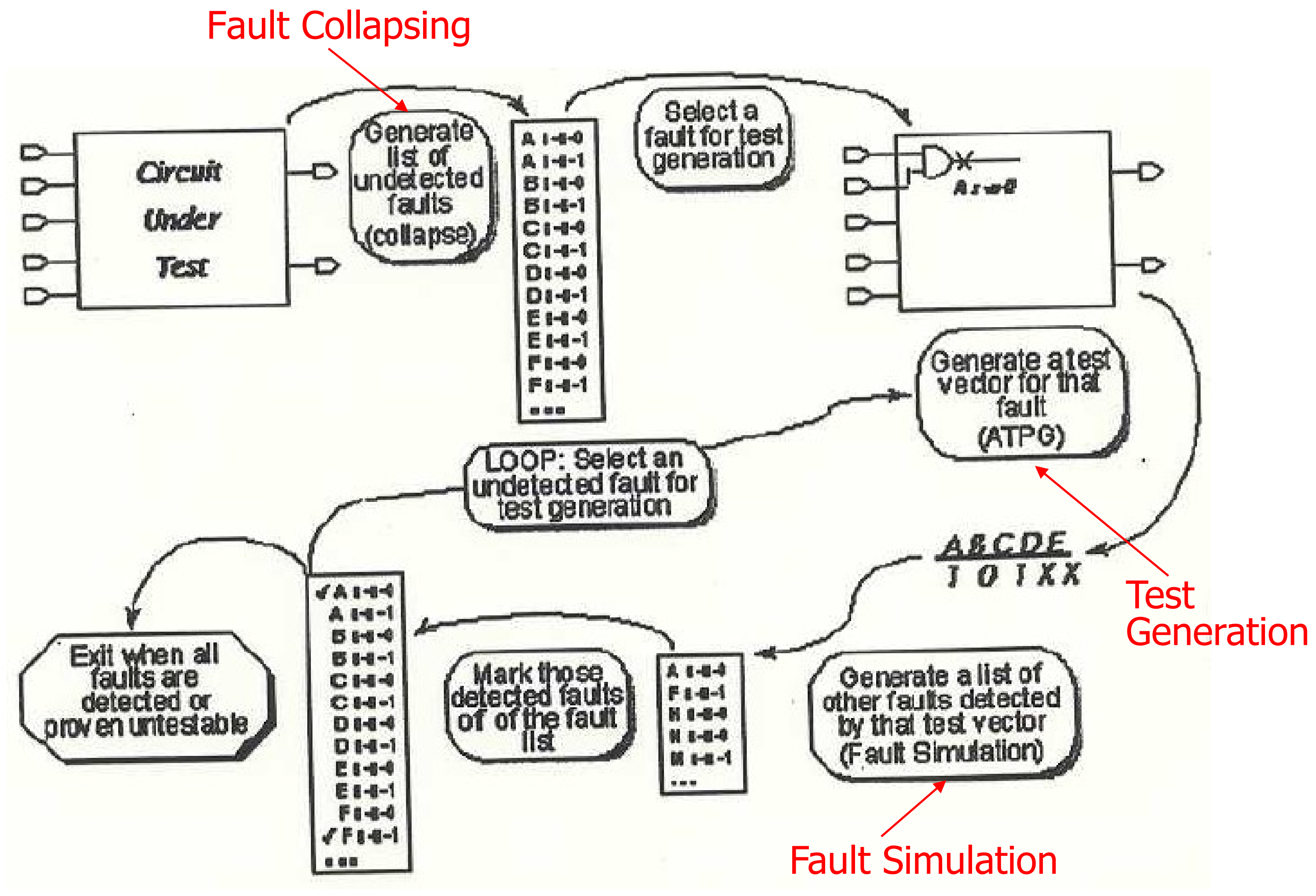
  - That is, the outputs in transitive fanout of c_{j_l} (paths that can propagate the fault effect) are disjoint from those in any of the other branches
- In such a case, if c_{j_l} is critical, then c_i is critical (while the converse is not true)

Critical Path Tracing (cont'd)

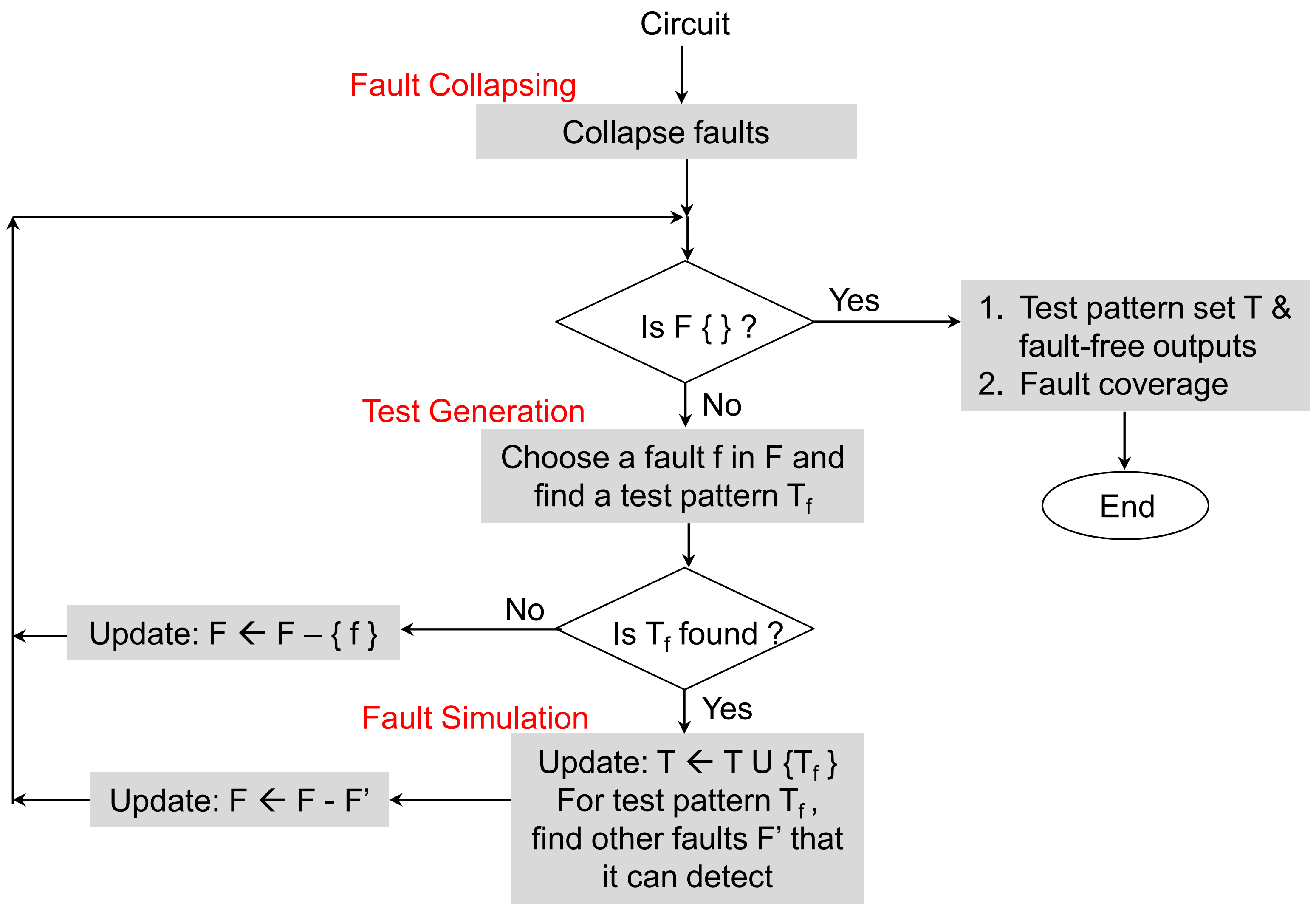
- The above fact can be used to reduce the number of cases where explicit fault simulation is required
  - Branches c_9 and c_{10} are independent so no explicit fault simulation required for stem c_7
  - Explicit fault simulation required for the other two stems (x_3 and c_4)



Fault Analysis System – Key Steps


Fault Collapsing
Circuit Under Test
Generate list of undetected faults (collapse)
A s-a-0
A s-a-1
B s-a-0
B s-a-1
C s-a-0
C s-a-1
D s-a-0
D s-a-1
E s-a-0
E s-a-1
F s-a-0
F s-a-1
...
Select a fault for test generation
A s-a-0
Generate a test vector for that fault (ATPG)
Test Generation
LOOP: Select an undetected fault for test generation
ABCDE
10 1XX
Generate a list of other faults detected by that test vector (Fault Simulation)
Fault Simulation
A s-a-0
F s-a-1
H s-a-0
H s-a-0
M s-a-1
...
Mark those detected faults off of the fault list
✓A s-a-0
A s-a-1
B s-a-0
B s-a-1
C s-a-0
C s-a-1
D s-a-0
D s-a-1
E s-a-0
E s-a-1
F s-a-0
✓F s-a-1
...
Exit when all faults are detected or proven untestable

Fault Analysis System – Process

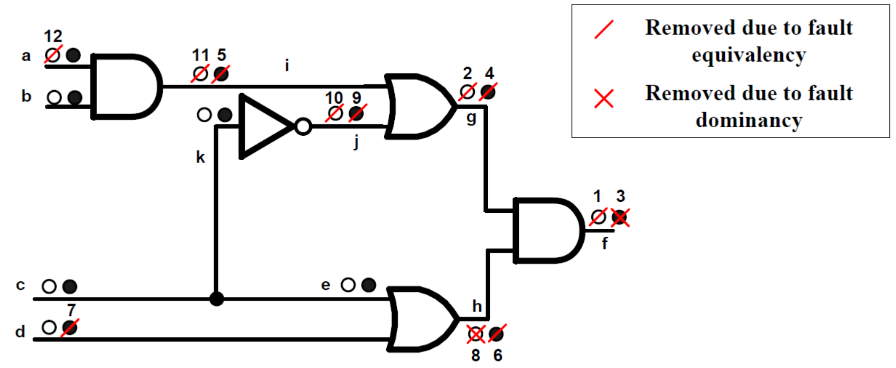


Fault Analysis System – Fault Collapsing

- For illustration, we have shown two methods for fault collapsing here. In general, only one method is needed to in this step.

- Applying checkpoint theorem is often more efficient specially when you solve problems by hand.

Fault collapsing, method 1.

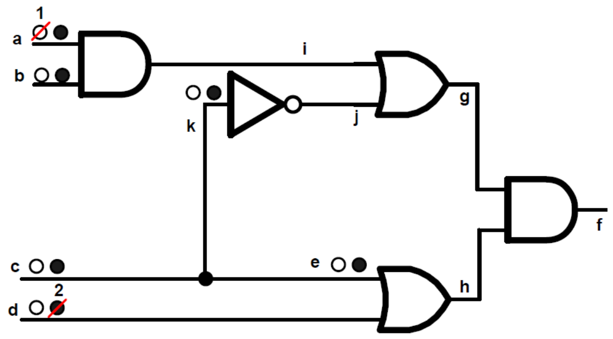


Removed due to fault equivalency
Removed due to fault dominancy
a
b
c
d
e
f
g
h
i
j
k
12
11 5
10 9
2 4
1 3
7
8 6

Total number of faults = **22**

Number of faults remaining = **10**

Fault collapsing, method 2 (checkpoint theorem).



Total number of faults = **22**

Number of faults remaining = **10**

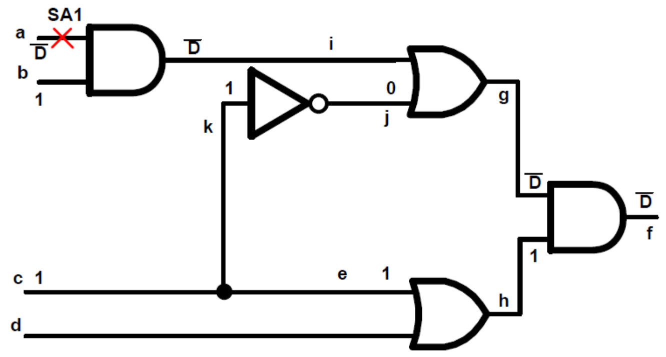
Collapse ratio = Number of faults remaining/ Total number of faults = 10/22 = **45.5%**

Collapsed Faults = **{a1, b0, b1, c0, c1, d0, e0, e1, k0, k1}**

Fault Analysis System – Iteration 1

Test Generation: Current Fault List = { **a1, b0, b1, c0, c1, d0, e0, e1, k0, k1** }

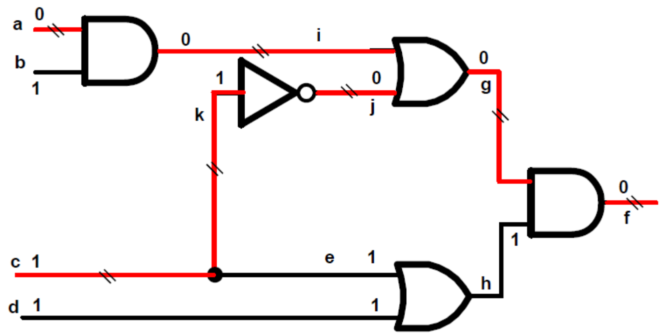
I. consider **SA1** Fault at **a**



Test pattern: abcd=011X=**0110 or 0111**

Test pattern set T = {0111}
Fault-free output = {0}

Fault Simulation, abcd=0111:



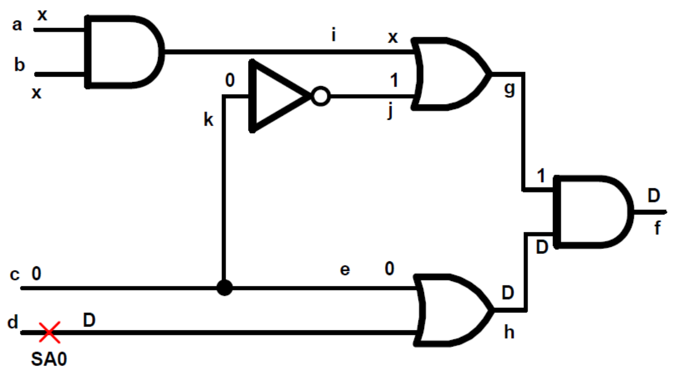
Faults detected by 0111 **= {a1, c0, k0, i1, j1, g1, f1}**

Updated fault list **= {b0, b1, c1, d0, e0, e1, k1}**

Fault Analysis System – Iteration 2

Test Generation: Current Fault List = { **b0, b1, c1, d0, e0, e1, k1** }

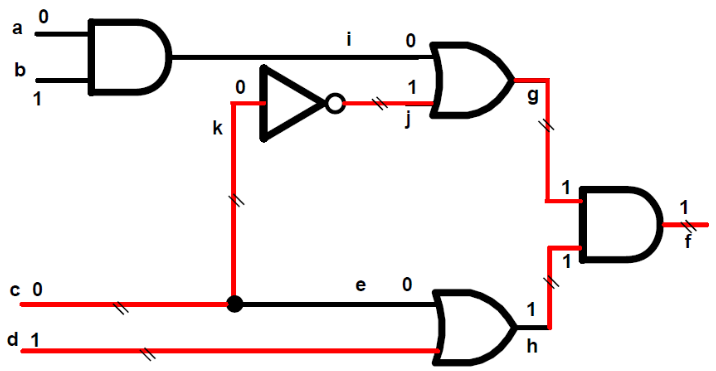
II. consider **SA0** Fault at **d**



Test pattern: abcd= **XX01=0001 or 0101 or 1001 or 1101**

Test pattern set T = {0111, 0101}
Fault-free output = {0, 1}

Fault Simulation, abcd=0101:



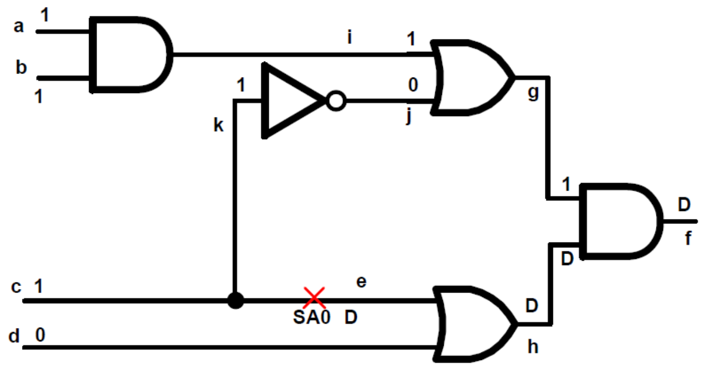
Faults detected by 0101 = **{c1, d0, k1, j0, g0, h0, f0}**

Updated fault list = **{b0, b1, e0, e1}**

Fault Analysis System – Iteration 3

Test Generation: Current Fault List = { **b0, b1, e0, e1** }

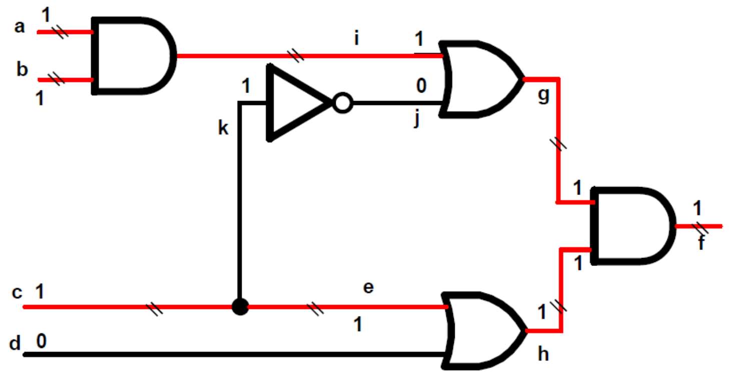
III. consider **SA0** Fault at **e**



Test pattern: abcd= **1110**

Test pattern set T = {0111, 0101, 1110}
Fault-free output = {0, 1, 1}

Fault Simulation, abcd=1110:



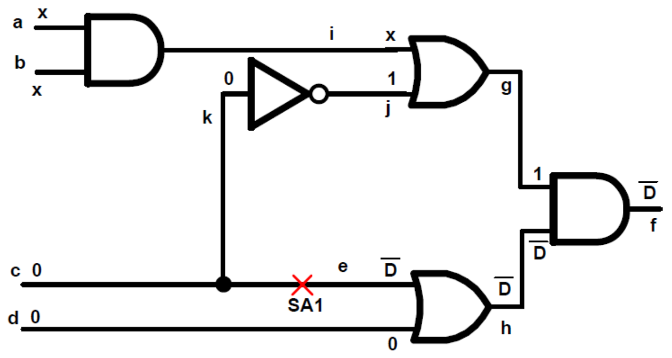
Faults detected by 1110 = **{a0, b0, c0, i0, g0, e0, h0, f0}**

Updated fault list = **{b1, e1}**

Fault Analysis System – Iteration 4

Test Generation: Current Fault List = { **b1, e1** }

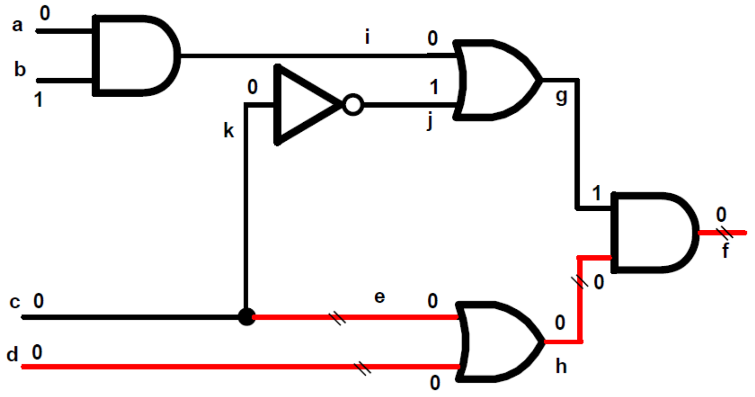
IV. consider **SA1** Fault at **e**



Test pattern: abcd= **XX00 = 0000 or 0100 or 1000 or 1100**

Test pattern set T = {0111, 0101, 1110, 0100}
Fault-free output = {0, 1, 1, 0}

Fault Simulation, abcd=0100:



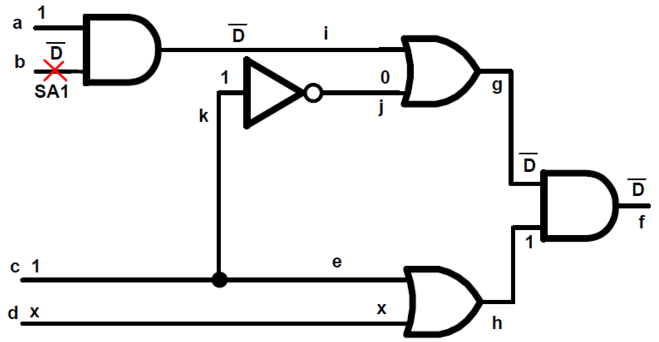
Faults detected by 0100 = **{d1, e1, h1, f1}**

Updated fault list = **{b1}**

Fault Analysis System – Iteration 5

Test Generation: Current Fault List = { **b1** }

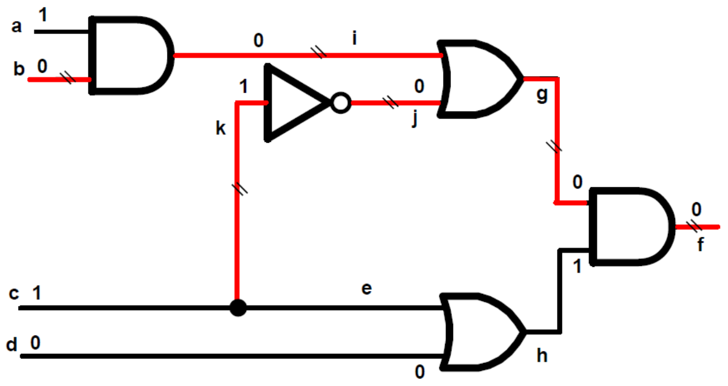
V. consider **SA1** Fault at **b**



Test pattern: abcd=**101X = 1011 or 1010**

Test pattern set T = {0111, 0101, 1110, 0100, 1010}
Fault-free output = {0, 1, 1, 0, 0}

Fault Simulation, abcd=1010:



Faults detected by 1010 **= {b1, k0, i1, j1, g1, f1}**

Updated Fault list = {}

Fault Coverage= 10/10=100%