

Testing of Digital Systems

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

Why Testing?

- Digital System
 - Complex
 - At various levels of abstraction
 - Errors/faults can be introduced easily.
- Need to guarantee its correctness
 - Cost of having bad products in customers' hands is really high.
- Cost to testing is increasing fast.

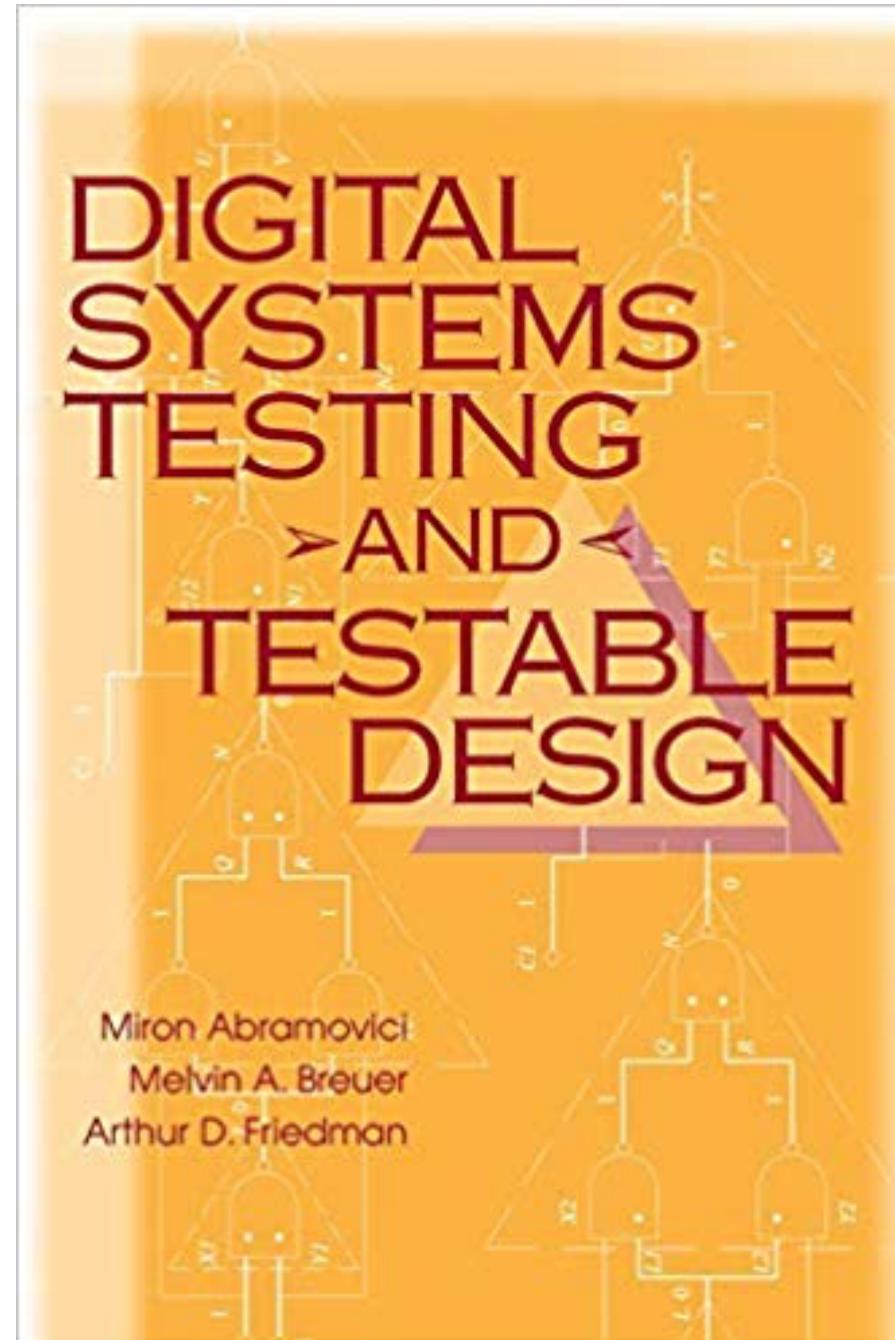
Digital System Testing

- How can we guarantee correctness of a digital system?
 - This course introduces basic concepts and principles for efficient testing.
 - Algorithms and design features for testing
- **Testing** -- Exercise a system and analyze its responses
 - Isolate good ones from faulty ones
- **Diagnosis** -- locate the faults and repair
 - Assume knowledge of internal structure

Prerequisites

- Catalog prerequisites
 - COP 4530 Data Structures
 - CDA 3201 Logic Design
- Highly desirable background
 - COT 4400 Analysis of Algorithms
 - CDA 4213 CMOS VLSI Design
 - Proficient in writing code in C/C++ or any other high level language.

Textbook



Evaluation

Homework/Exams	Weight	Date
Homework	20%	TBA
Exam 1	25%	Around 6th week
Exam 2	25%	Around 13th week
Final Project	30%	Starts after the Spring break

Final Grading Scale

$x < 60\%$	$60\% \leq x < 70\%$	$70\% \leq x < 80\%$	$80\% \leq x < 90\%$	$x \geq 90\%$
F	D	C	B	A

Communications

- Canvas
 - Assignments & grades
 - Announcements
- www.cse.usf.edu/~haozheng/teach/psv
 - Lecture slides, reading assignments,
 - Other relevant material

Topics (Tentative)

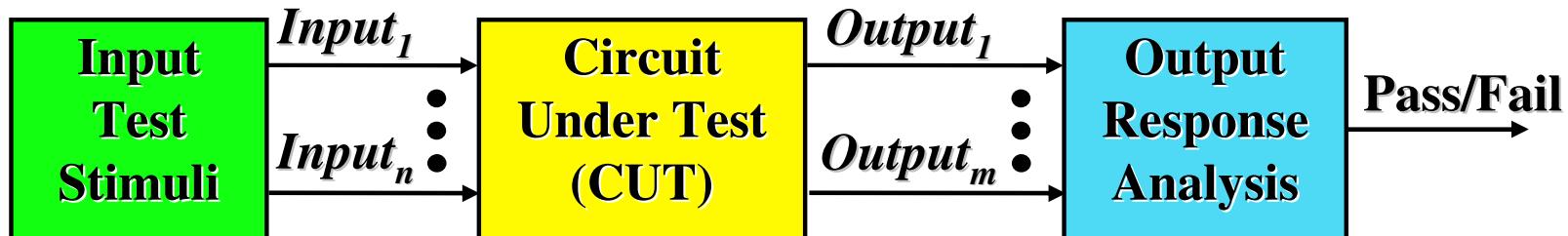
- Circuit modeling/simulation – chapter 2 & 3
- Fault modeling – chapter 4
- Fault simulation – chapter 5
- Testing for single stuck-at fault – chapter 6
- Testing for bridging fault – chapter 7
- Functional testing – chapter 8
- Design for testability – chapter 9
- Built-in self-test – chapter 11
- System-level diagnosis – chapter 15

Levels of Abstraction

- Based on granularities of information
 - System level
 - Processor level
 - Register transfer level
 - *Logic level – focus of this course*
 - Transistor-level
- This course focuses on issues in functional testing
 - Electrical characteristics are ignored.

Testing During VLSI Life Cycle

- Testing typically consists of
 - Applying set of test stimuli to
 - Inputs of *circuit under test* (CUT), and
 - Analyzing output responses
 - If incorrect (fail), CUT assumed to be faulty
 - If correct (pass), CUT assumed to be fault-free



Testing Scenarios

- “Testing” is a general term used for widely different activities & environments
- Examples
 - One or more subsystems testing another by sending and receiving messages
 - A processor testing itself by executing a diagnostic program
 - Automatic Test Equipment (ATE) checking a circuit by applying and observing binary patterns

Errors and Faults

- **Observed error:** instance of incorrect operation
 - Important for testing
- Causes of observed errors
 - **Design Errors:** inconsistent implementation & spec
 - **Fabrication Errors:** due to human errors
 - **Fabrication Defects:** due to imperfect manufacturing
 - **Physical Failures:** due to system operation

Design Errors

- Errors due to incorrect design/understanding/implementation
- Examples
 - Incomplete and inconsistent specifications
 - Incorrect mappings between different levels of design
 - Violations of design rules

Fabrication Errors

- Due to mistakes made during fabrication/assembly sequence
- Examples
 - Wrong components
 - Incorrect wiring
 - Shorts caused by improper soldering

Fabrication Defects

- Errors due to **imperfect** manufacturing process
- Examples
 - Shorts or Opens
 - Improper doping profiles,
 - mask alignment errors

Physical Failures

- Errors occurring during system lifetime due to component wear-out and/or environmental factors
- Examples
 - Electro-migration
 - Temperature/Humidity/Vibration
 - Cosmic radiation and α -particles

Physical Faults

- Design Errors
- Fabrication Errors
- Fabrication Defects
- Physical Failures



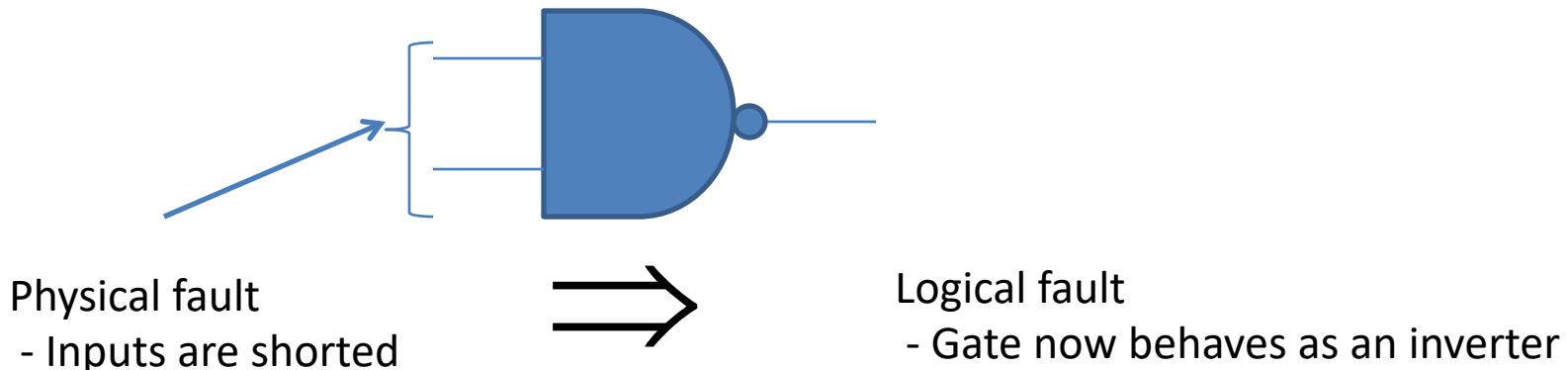
Physical Faults

Stability of Physical Faults:

- Permanent
- Intermittent
- Transient

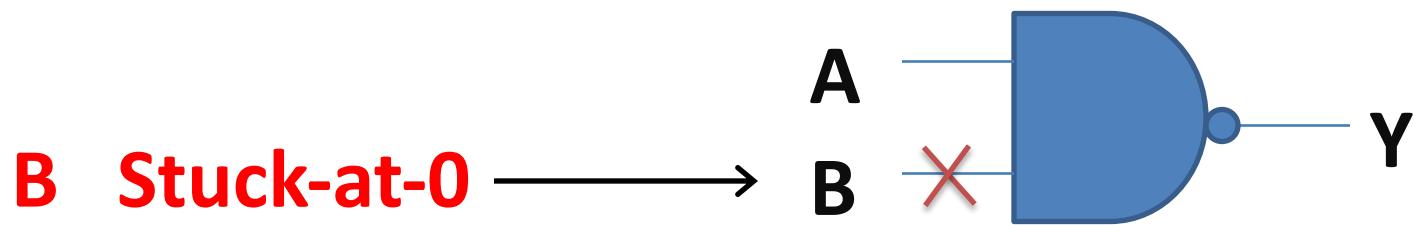
Physical & Logical Faults

- Physical faults are difficult to handle mathematically.
- Physical faults are represented by the **logical fault**.
 - Example:



Fault Model

- If a fault occurs, then we need to *detect* it
- To detect a fault, we should be able to observe the error cause by the fault
- A fault model refers to natures of logical faults.
- Widely used model:
 - A single line **stuck** at a logic value



Modeling & Simulation

- Models are used for pre-fab testing.
- Model of a system
 - Digital computer representation of the system in terms of data structures and/or programs
- Logic Simulation
 - Model exercised by stimulating its inputs with logic values.
 - Evolution of signals over time in response to an applied input sequence.

Test Evaluation

- Objective: determine the quality of a test.

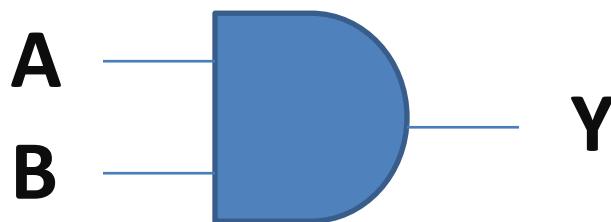
$$\text{fault coverage} = \frac{\text{Number of faults detected}}{\text{Total number of faults}}$$

- Evaluated wrt a fault model`
- Performed via a simulated testing experiment known as **fault simulation**.

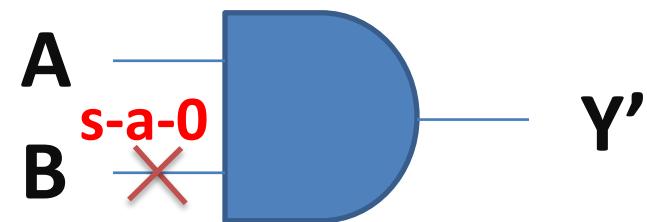
Test Evaluation

Question: Assume a fault B s-a-0 (stuck-at-0). Find a test vector (t) that detects this fault.

Without fault



With fault



Hint: When you apply the test, the outputs should differ in the presence and absence of the fault

Types of Testing

Criterion	Attribute of testing method	Terminology
When is testing performed?	<ul style="list-style-type: none">Concurrently with the normal system operationAs a separate activity	On-line testing Concurrent testing Off-line testing
Where is the source of the stimuli?	<ul style="list-style-type: none">Within the system itselfApplied by an external device (tester)	Self-testing External testing
What do we test for?	<ul style="list-style-type: none">Design errorsFabrication errorsFabrication defectsInfancy physical failuresPhysical failures	Design verification testing Acceptance testing Burn-in Quality-assurance testing Field testing Maintenance testing

Types of Testing...contd.,

What is the physical object being tested?	<ul style="list-style-type: none">• IC• Board• System	Component-level testing Board-level testing System-level testing
How are the stimuli and/or the expected response produced?	<ul style="list-style-type: none">• Retrieved from storage• Generated during testing	Stored-pattern testing Algorithmic testing Comparison testing
How are the stimuli applied?	<ul style="list-style-type: none">• In a fixed (predetermined) order• Depending on the results obtained so far	Adaptive testing

Types of Testing...contd.,

Criterion	Attribute of testing method	Terminology
How fast are the stimuli applied?	<ul style="list-style-type: none">Much slower than the normal operation speedAt the normal operation speed	DC (static) testing AC testing At-speed testing
What are the observed results?	<ul style="list-style-type: none">The entire output patternsSome function of the output patterns	Compact testing
What lines are accessible for testing?	<ul style="list-style-type: none">Only the I/O linesI/O and internal lines	Edge-pin testing Guided-probe testing Bed-of-nails testing Electron-beam testing In-circuit testing In-circuit emulation
Who checks the results?	<ul style="list-style-type: none">The system itselfAn external device (tester)	Self-testing Self-checking External testing

Testing Techniques

- Diagnostic Programs
- In-circuit Emulation
- On-line Testing
- Guided-probe Testing
- Misc Testing

Testing – Diagnostic Programs

- Stimuli – generated within system itself
 - By software or firmware
- Some parts of the system (*hardcore*) must be fault-free
- Can be adaptively applied
- Usually for field or maintenance testing

Testing – In-circuit Emulation

- Remove the need for a fault-free hardcore
- Used for μ P-based systems
- μ P removed from the board during testing
- Tester emulates the function of μ P

On-line Testing

- Stimuli + Response are not known in advance
 - Stimuli are provided by patterns received during the normal operation mode
- Object of interest
 - Some property of the response which should be invariant throughout testing process

Example 1: Fault-free decoder (only one o/p should be high at any time)

Example 2: Word Parity

- Self-checking sub-circuits known as **checkers**.

Guided-Probe Testing

- Used for board-level testing
- If errors found, then tester decided which internal lines should be monitored
- Placed a probe on selected lines and re-tested
- Goal is to trace back to source of the error
- Sophisticated testers can monitor
 - A group of lines
 - All accessible internal lines (using a bed-of-nails fixture)

Other Testing Approaches

- **Algorithmic Testing** -- Generation of input patterns during testing; counters, feedback shift registers used to generate patterns
- **Comparison Testing** – expected responses generated from good known copy or by real-time emulation
- **Compact Testing** – Check some function $f(R)$ instead of the response R itself.
 - Example: $f = \text{no. of } 1's \text{ in } R$
 - Advantages: simplified and less memory intensive

Diagnosis and Repair

- After an error is observed, its cause is diagnosed.
- Once the cause is understood, the UUT may be repaired.
- Two diagnosis approaches for logical faults
 - Cause-effect analysis
 - Effect-cause analysis

Test Generation (TG)

- A process of determining the stimuli for testing a system.
- Different testing methods require different TG.
- Fault-oriented TG – targeted at certain fault models
 - Easier to automate
- Function-oriented TG – targeted at certain functions
 - May require substantial manual efforts.

Design For Testability

- Digital circuits come with additional circuitry to assist testing and diagnosis.
- Should be considered at early design stage to balance additional cost to the implementation and reduction in testing cost.
- More important as the complexity of modern circuits increases drastically.

Backup

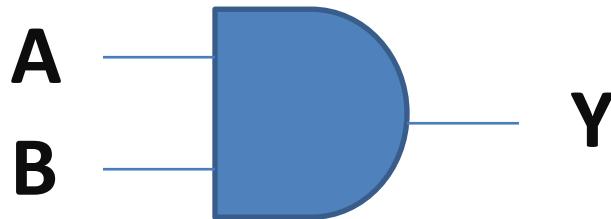
Main Topics (Tentative)

- Circuit modeling
- Fault modeling & simulation
- Testing for different types of faults
- Functional testing
- Design for testability
- Built-in self test
- System-level diagnosis
- Other contemporary testing issues

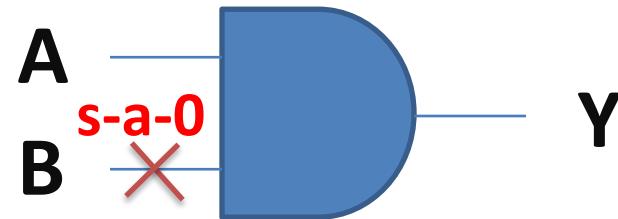
Test Vector

Question: Assume a fault B s-a-0 (stuck-at-0). Find a test vector (t) that detects this fault *and compute its fault coverage.*

Without fault



With fault



- 1) What is the size (N) of the fault universe ?
- 2) How many faults (M) can the test vector detect?
- 3) Fault Coverage = $(M/N) =$

CIS 4930 Digital System Testing Modeling

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

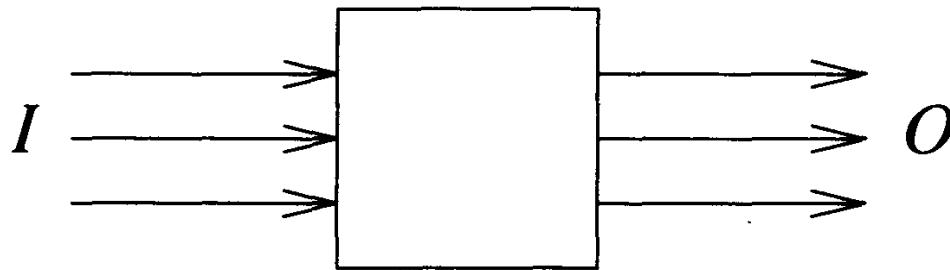
Reading

- Chapter 2

2.1 Basic Concepts

- Functional modeling at logic level
- Functional modeling at register level
- Structural models

Black Box View



- A system can be viewed as a black box i.e., generates **outputs** in response to **inputs**.
- Behavior is defined by the I/O mapping.
 - i.e. value transformation
 - Different value representations at different levels.

Behavioral/Functional Model

- **System Behavior** is defined by
 - I/O Mapping occurs over time
- A **logic function** is a I/O Mapping without timing.
- A **functional model** is a representation of logic function
- A **behavioral model** is
 - A functional Model + A representation of timing relations

Structural Model

- Collection of interconnected smaller boxes called components/elements
- Often hierarchical
 - A component itself represented by a structural model.
- Bottom-level boxes known as **primitive** elements
- Can shown as block Diagrams, or schematics
- In practice structural and functional are inter-mixed.
- Often necessary to describe large systems.

2.2 Functional Modeling at Logic Level

- Truth Tables
- Primitive Cubes
- State Tables
- Flow Tables
- BDDs – Binary Decision Diagrams
- Programs

Truth Tables & Primitive Cubes

x_1	x_2	x_3	Z
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

(a)

x_1	x_2	x_3	Z
x	1	0	0
1	1	x	0
x	0	x	1
0	x	1	1

(b)

Figure 2.2 Models for a combinational function $Z(x_1, x_2, x_3)$ (a) truth table
(b) primitive cubes

Cubical Notation

x_1	x_2	x_3	Z
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

- First two rows can be combined
 $00x \mid 1$
where X is a don't care
- A cube of a function $Z(x_1, x_2, x_3)$ has the form
 $(v_1, v_2, v_3 \mid v_Z)$
where $v_Z = Z(v_1, v_2, v_3)$
- A cube of Z can represent multiple rows in a truth table

Cubical Notation

- If cube q can be obtained from cube p by replacing one or more x in p by 0 or 1
then p **covers** q

Eg: $00x \mid 1$ covers cubes $000 \mid 1$ and $001 \mid 1$

Implicants

- An **implicant** g of Z is a sub-function such that
$$(g = 1) \Rightarrow (Z = 1)$$
- An implicant “covers” one or more minterms in a truth table.
 - A disjunction of some minterms.
- If an implicant covers at least one row that is not covered by any other implicants, it is known as a **prime implicant**.

Prime Implicant and Primitive Cube

- Cube $00x \mid 1$ represents implicant

$$\overline{x_1} \quad \overline{x_2}$$

- A cube that represents a prime implicant is known as **primitive cube**.

Primitive Cube Representation

x_1	x_2	x_3	Z
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

(a)

x_1	x_2	x_3	Z
x	1	0	0
1	1	x	0
x	0	x	1
0	x	1	1

(b)

Compact
Representation
of function Z

Figure 2.2 Models for a combinational function $Z(x_1, x_2, x_3)$ (a) truth table
(b) primitive cubes

Intersection Operator

\cap	0	1	x
0	0	\emptyset	0
1	\emptyset	1	1
x	0	1	x

Figure 2.3 Intersection operator

x_1	x_2	x_3	Z
x	1	0	0
1	1	x	0
x	0	x	1
0	x	1	1

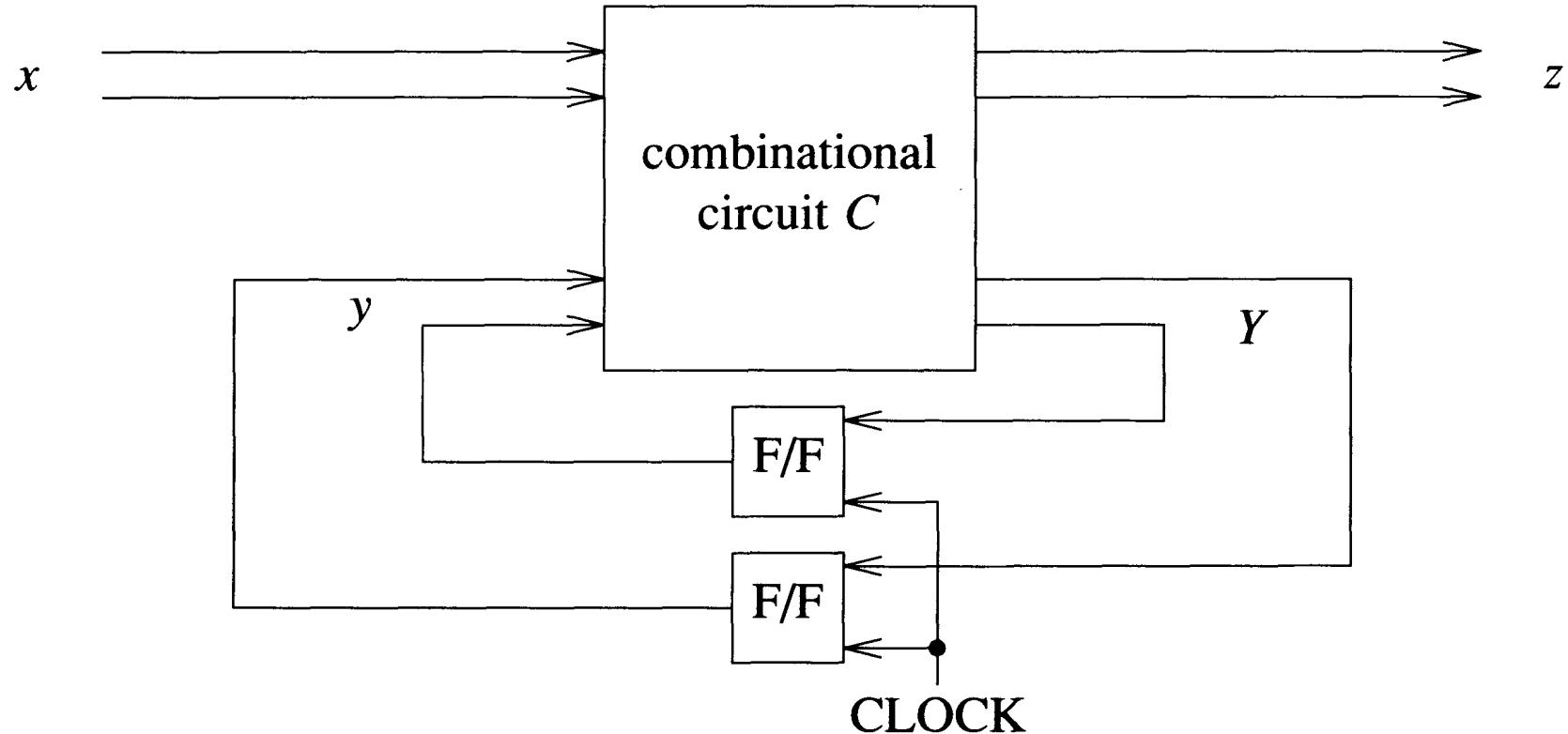
Find Z for 001

- $x10 \cap 001$ -- Inconsistent
- $11x \cap 001$ -- Inconsistent
- $x0x \cap 001$ -- Consistent
- $0x1 \cap 001$ -- Consistent

Output is $Z = 1$

Modeling Synchronous Sequential Circuit

- Canonical structure of synchronous circuits



State Table Representation

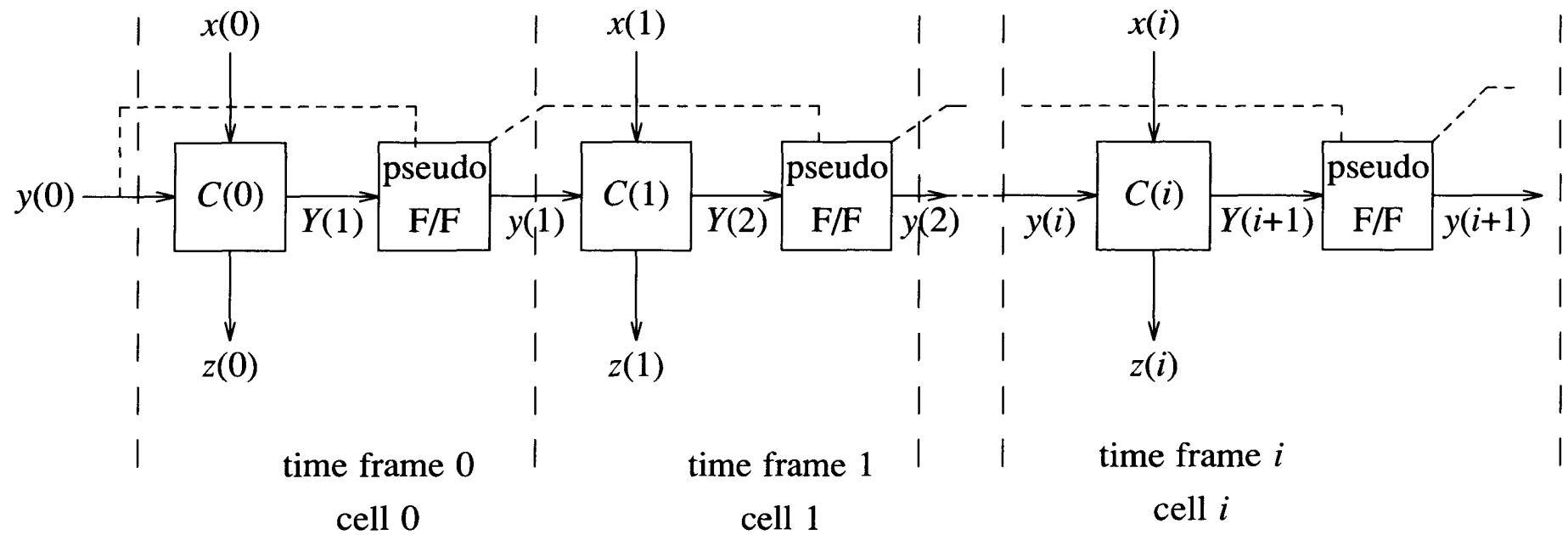
- FSM – Finite State Machine can be represented by a state table
- Inputs are sampled at regular intervals and next state and output computed

		x	
		0	1
q	1	2,1	3,0
	2	2,1	4,0
	3	1,0	4,0
	4	3,1	3,0

$N(q,x), Z(q,x)$

Pseudo-combinational -- Iterative Array

- Unroll seq circuit \rightarrow pseudocombinational cir.



Asynchronous Circuit

- Represented by a flow table
- Single input change can lead to multiple state changes until a stable configuration is reached
- Stable configurations are shown in **bold** font

	x_1x_2			
	00	01	11	10
1	1,0	5,1	2,0	1,0
2	1,0	2,0	2,0	5,1
3	3,1	2,0	4,0	3,0
4	3,1	5,1	4,0	4,0
5	3,1	5,1	4,0	5,1

Figure 2.7 A flow table

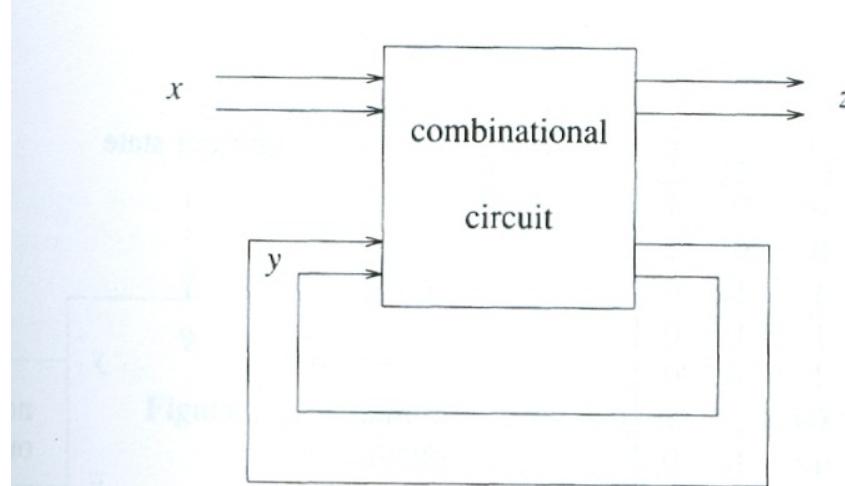


Figure 2.8 Canonical structure of an asynchronous sequential circuit

Binary Decision Diagrams (BDDs)

- A graph representation of circuit functions

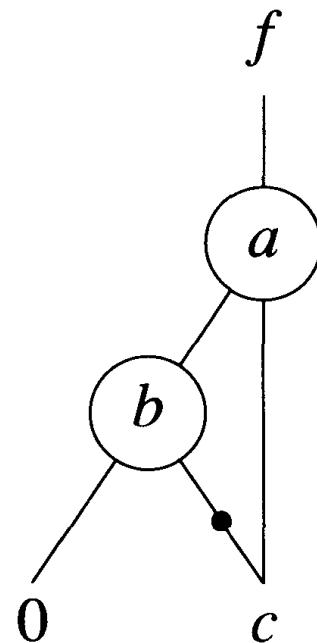
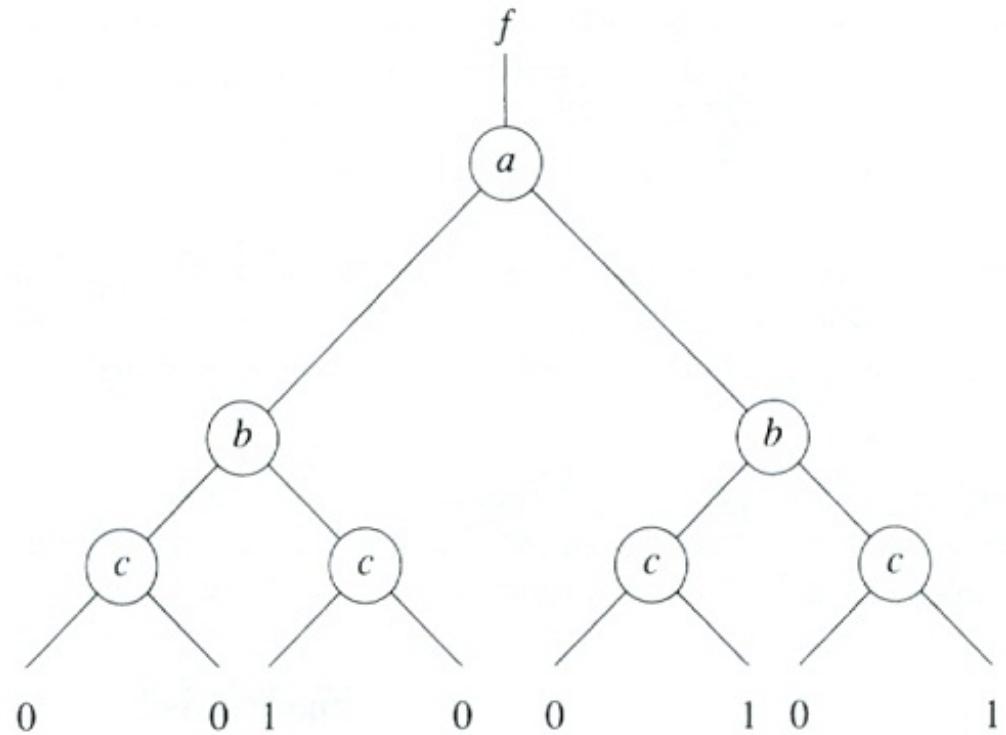


Figure 2.11 Binary decision diagram of $f = \bar{a}b\bar{c} + ac$

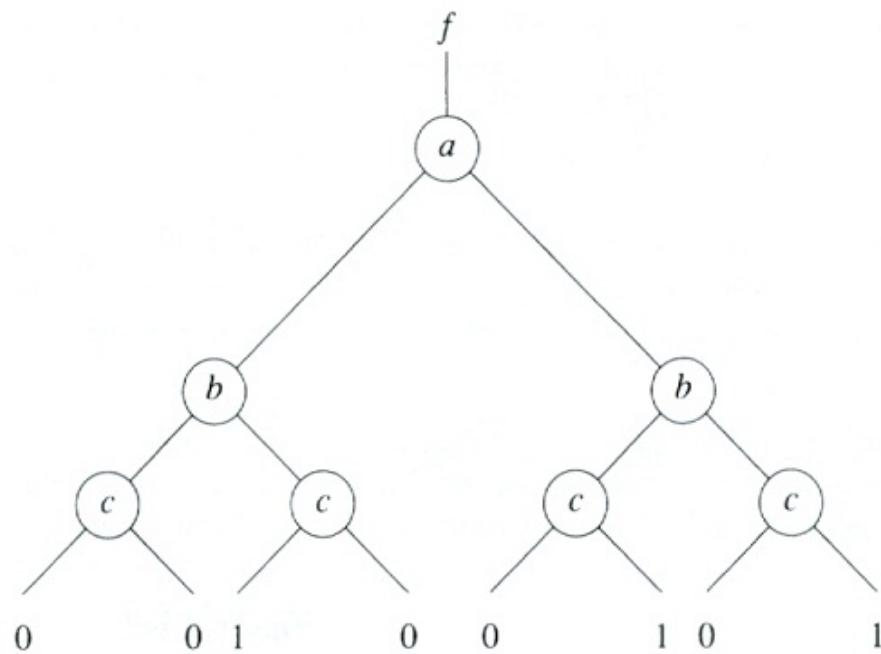
Constructing BDD

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

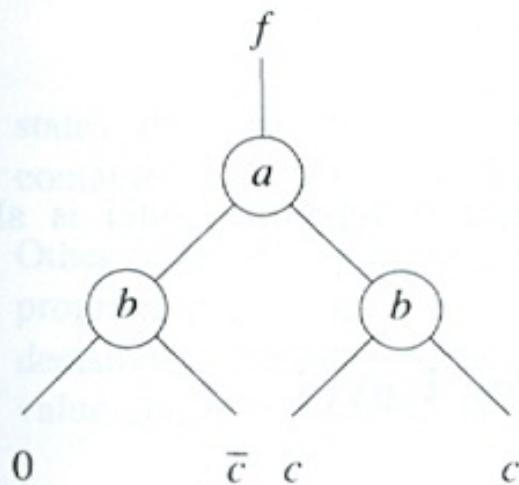
(a)



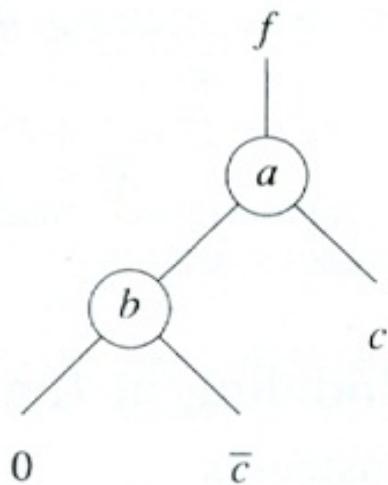
(b)



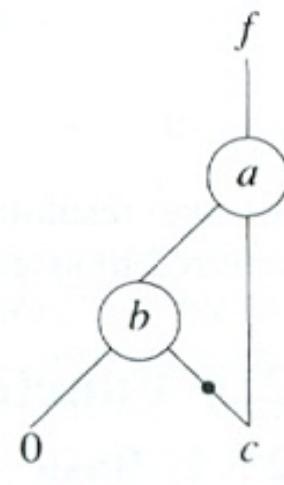
(b)



(c)



(d)



(e)

BDD for a JK FF

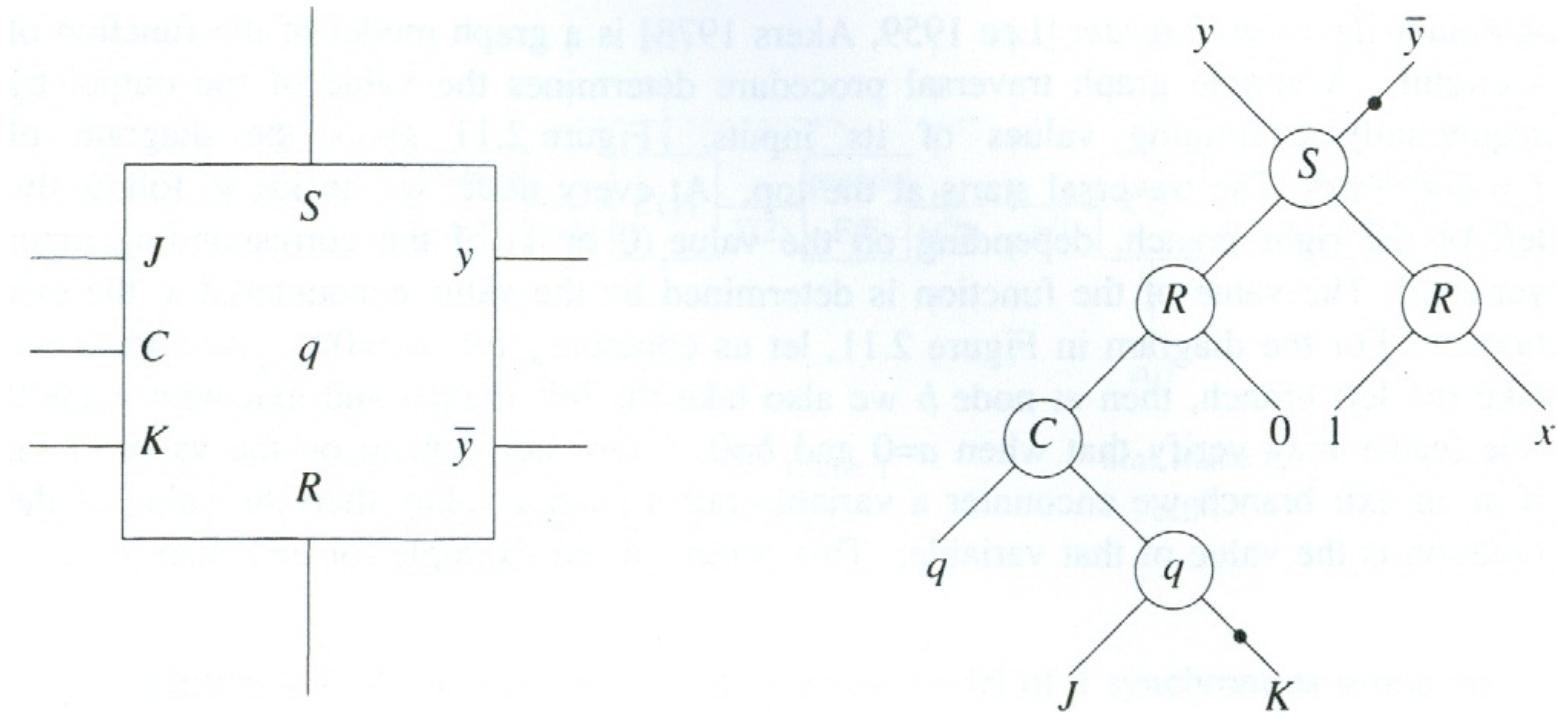


Figure 2.12 Binary decision diagram for a *JK* F/F

Program as Functional Model

Assembly Program

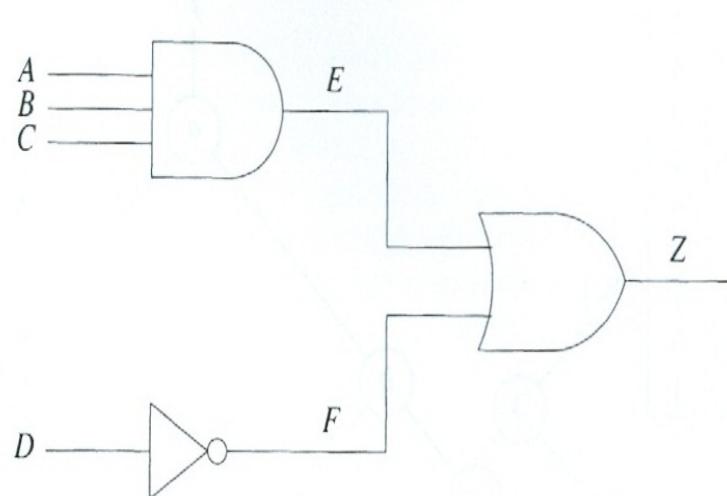


Figure 2.14

```
LDA A /* load acc with A */  
AND B /* compute A.B */  
AND C /* compute A.B.C */  
STA E /* store partial result */  
LDA D /* load acc with D */  
INV /* compute not D */  
OR E /* compute A.B.C + ~(D) */  
STA Z /* store results */
```

C Program

“Compiled Code Model”

```
E = A & B & C ;  
F = ~ D;  
Z = E | F ;
```

2.3 Functional Modeling at Register Level

- System model at register and instruction set levels

- Storage declarations:

```
register IR [0 → 7]           //8-bit register
```

```
memory ABC [0 → 255; 0 → 15] // 256 word memory
```

- Data paths implicitly defined with operations on registers.
- Data processing by primitive operators:

```
C = A + B      // add two registers A and B
```

RTL constructs

- Control flow:
 - **if** X **then** $C = A + B$
 - **if** (*CLOCK and* $(AREG < BREG)$) **then** $AREG = BREG$
 - **test** (*IR* [$0 \rightarrow 3$])
 - case 0:** *operation0*
 - case 1:** *operation1*
 - :
 - testend**

FSM modeling

- One block per state
- Only one state is active/current at any time

state S_1, S_2, S_3 /* state register */

S_1 : **if** X **then**

begin

$$P = Q + R$$

go to S_2

end

else

$$P = Q - R$$

go to S_3

S_2 : ...

Procedural and Non-procedural RTLs

- Procedural RTLs (sequential semantics, ex. C)
 - Use a sequential programming language
 - implicit cycle-based timing model (i.e., assures that the state of the model accurately reflects state of processor at the end of a cycle)
- Non-procedural RTLs (concurrent semantics, ex. VHDL)
 - Statements conceptually executed in parallel
- Example: $A = B$
 $C = A$
 - if the above is procedural code, register C gets value B after execution
 - If the above is non-procedural code, then C gets old value of A

Timing Modeling

- Explicitly specify delay to add more details.
 - Make a model more accurate.
- Examples:

```
// delay updating C by 100.  
C = A + B, delay = 100
```

```
// delay every update to C by 100.  
delay C 100
```

2.4 Structural Models

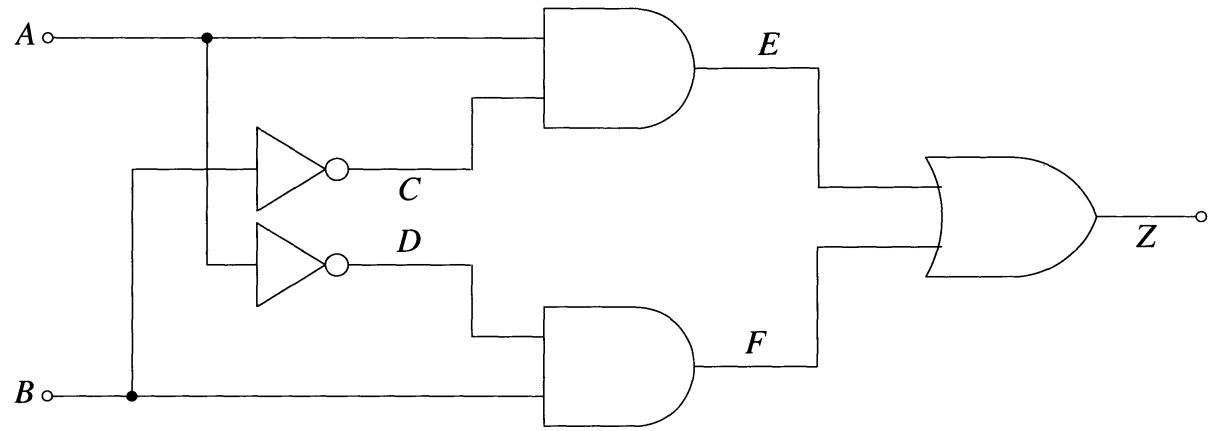


Figure 2.15 Schematic diagram of an XOR circuit

CIRCUIT XOR
INPUTS = A,B
OUTPUTS = Z

NOT	D, A
NOT	C, B
AND	E, (A, C)
AND	F, (B, D)
OR	Z, (E, F)

Connections are implicitly done via name matching.

Fanout-Free Circuit

- If a gate drives more than one gate then it has a **fanout**.
- A fanout-free circuit can be represented as a tree

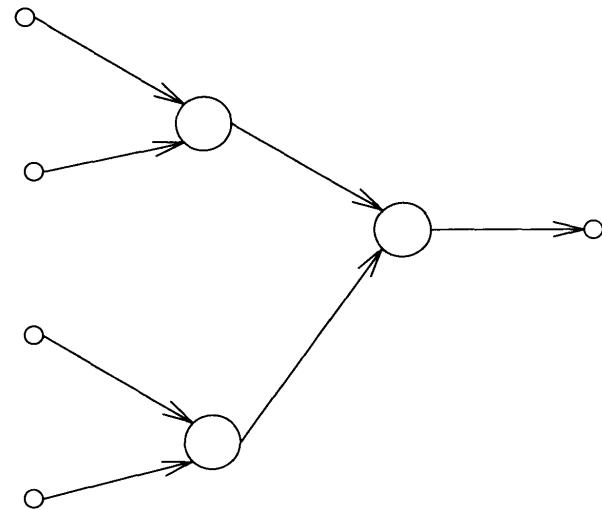
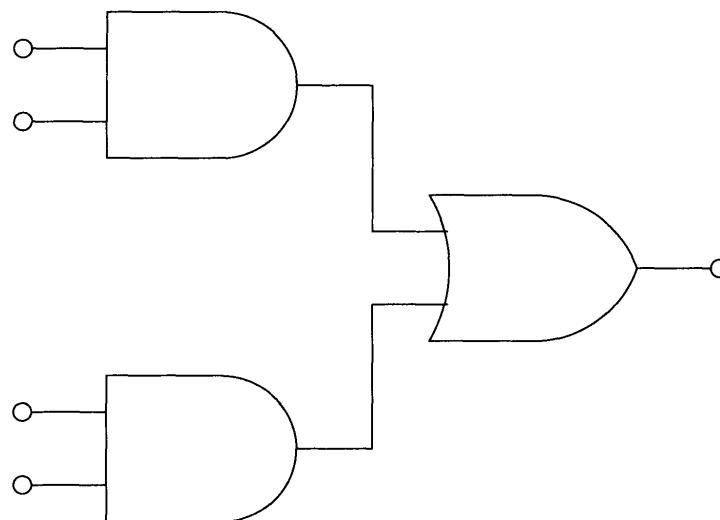


Figure 2.17 Fanout-free circuit and its graph (tree) representation

Reconvergent Fanout

- A signal fans out into two or more paths which later converge at a gate.
- Common in real circuits
- Makes fault detection problem more difficult

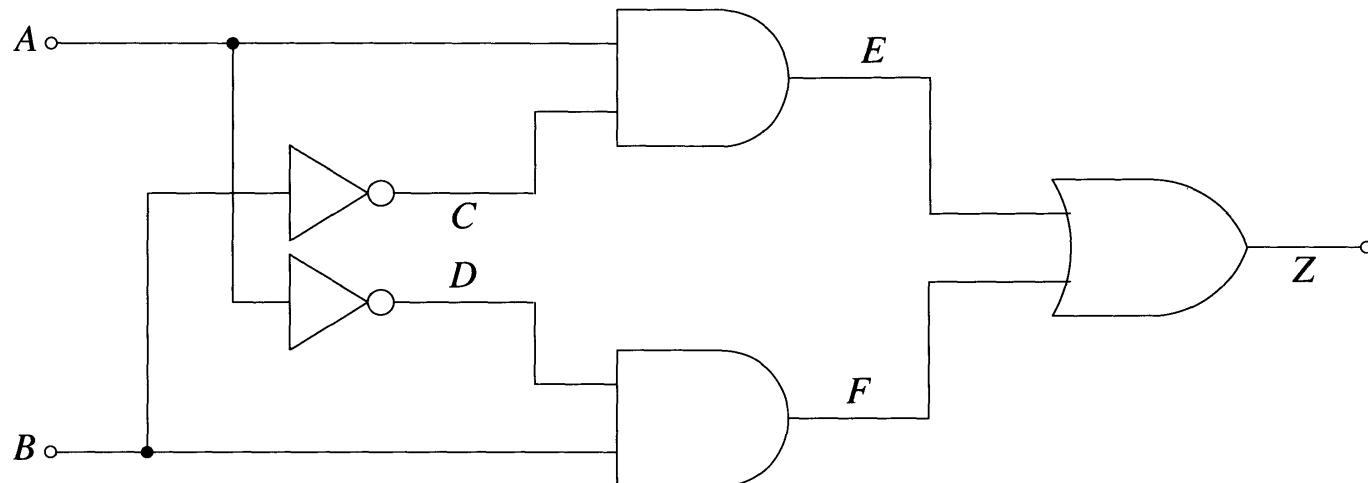
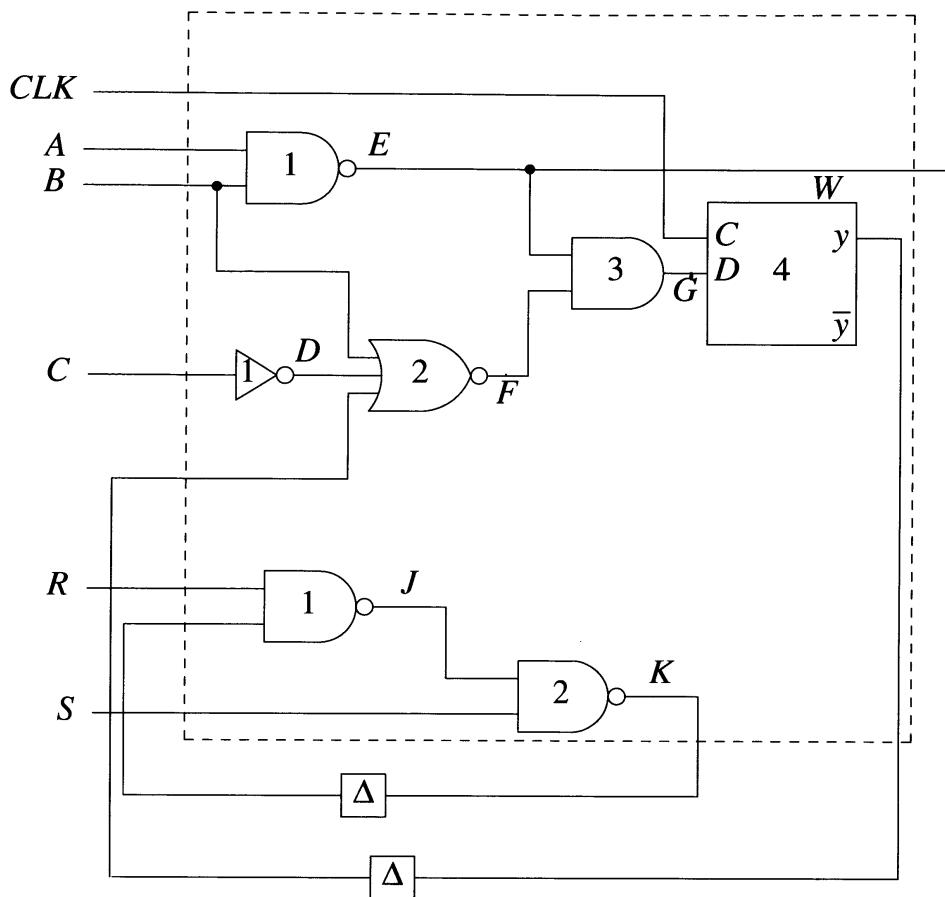


Figure 2.15 Schematic diagram of an XOR circuit

Logic Levels

- Logic level of a gate is it's distance from primary inputs
- Can be computed in a breadth-first manner



Summary

- Digital system model
 - representation of the actual system
 - can be done at logic and RT-levels
- Behavioral model = Functional Model + Timing
- Models for combinational circuits: truth table, primitive cubes, BDDs, programs (compiled-code models)
- Sequential circuits: state tables, RT level case statement
- Reconvenger Fanout – signal branches and later converges on a gate
- Non-procedural language used for RTL modeling to mimic concurrent execution of the hardware

Backup

Fanout Terminology

- Stem and fanout branches

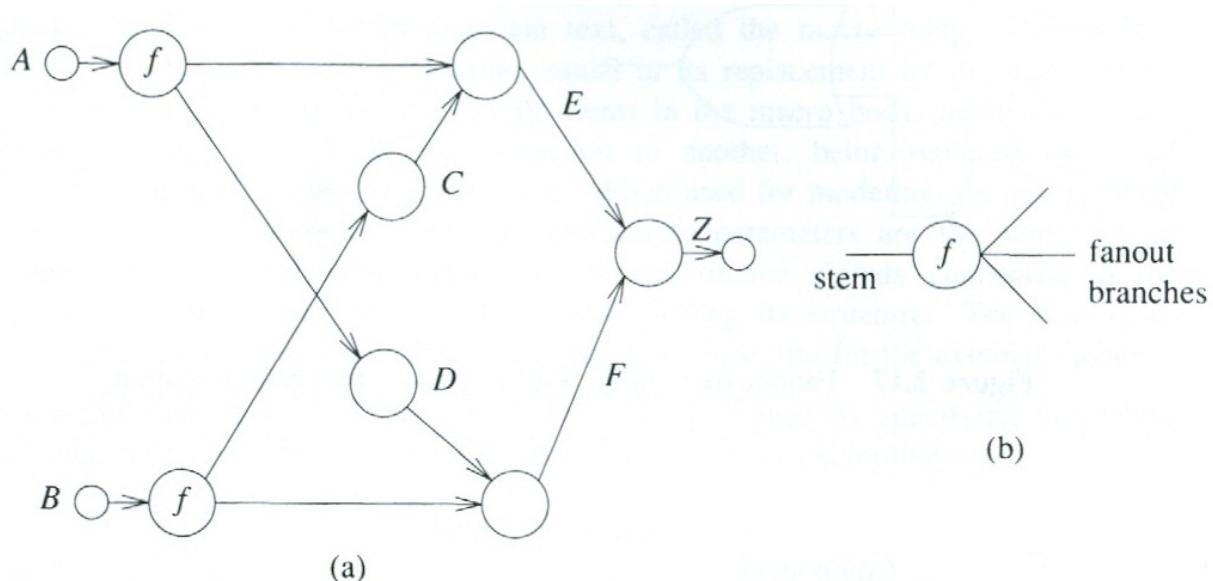


Figure 2.19 (a) Graph model for the circuit of Figure 2.15 (b) A fanout node

CIS 4930 Digital System Testing

Logic Simulation

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

Overview

- What is Logic Simulation?
- Types of Simulation
 - Compiled Simulation
 - Event Driven Simulation
- Delay models
- Hazards

3.1 What is Logic Simulation?

- Based on the logic model, it is the process of evolving signals in time in response to input stimuli

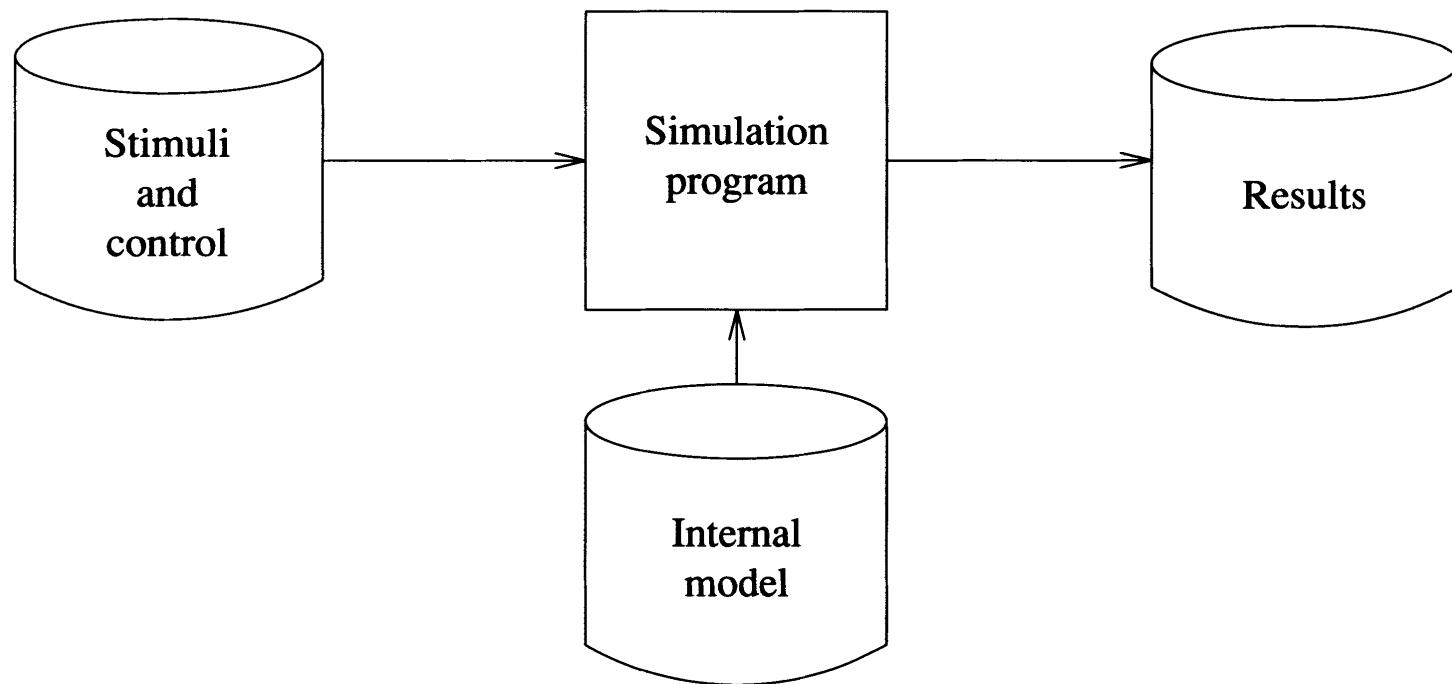


Figure 3.1 Simulation process

Why Logic Simulation?

- Verify design correctness i.e., design has desired behavior (function + timing)
- Verify against expected results
- Can be used to answer questions such as:
 - Correct independent of initial state?
 - Sensitive to delay variations of components?
 - Free of critical races, oscillations, etc.?

Why Logic Simulation?...contd.,

- We can also use it to:
 - Evaluate design alternatives
 - Evaluate proposed changes
 - Documentation (timing diagrams)
- Against a prototype, simulation has following advantages:
 - Checking various error conditions
 - Ability to change delays in the model
 - Start the simulated circuit in any desired state
 - Precise control of timing of async events (interrupts)
 - Automated testing

3.2 Problems in Logic Simulation

- How does one generate input stimuli?
 - Test generation
- How does one know the results are correct?
 - Output checking
- How “good” are the applied input stimuli, i.e., how “complete” is the testing experiment?
 - Test coverage

Your design is as good as your testing cases!

- Only the design behavior covered by the test cases is checked!

3.3 Types of Simulation

- **Compiled simulation**
 - Compiled code generated from RTL functional/structural models
 - Cannot deal with timing/delay
 - Applicable to synchronous functional testing.
- **Event-drive simulation**
 - Model interpreted from the data structures generated from RTL functional/structural models
 - Simulate only *active* part of the circuit.
 - Can simulate model with various delay specifications.
 - More general.

3.4 The Unknown Logic Value

- When a circuit is powered on, initial state of FFs and RAMs is unpredictable.
- Special logic value, u , to indicate unknown logic value
 - need to extend Boolean operators to 3-valued logic
- u represents one value in set $\{0,1\}$
- 0 and 1 represented by sets $\{0\}$, $\{1\}$ respectively

Boolean Operation with u

$$\begin{aligned}\text{AND}(0, u) &= \text{AND}(\{0\}, \{0,1\}) \\&= \{\text{AND}(0,0), \text{AND}(0,1)\} \\&= \{0, 0\} \\&= \{0\} = 0\end{aligned}$$

$$\begin{aligned}\text{OR}(0, u) &= \text{OR}(\{0\}, \{0,1\}) \\&= \{\text{OR}(0,0), \text{OR}(0,1)\} \\&= \{0, 1\} \\&= u\end{aligned}$$

$$\text{NOT}(u) = \text{NOT}(\{0,1\}) = \{\text{NOT}(0), \text{NOT}(1)\} = \{1, 0\} = u$$

Truth Tables for 3-valued Logic

AND	0	1	<i>u</i>
0	0	0	0
1	0	1	<i>u</i>
<i>u</i>	0	<i>u</i>	<i>u</i>

OR	0	1	<i>u</i>
0	0	1	<i>u</i>
1	1	1	1
<i>u</i>	<i>u</i>	1	<i>u</i>

NOT	0	1	<i>u</i>
	1	0	<i>u</i>

Figure 3.2 Truth tables for 3-valued logic

\cap	0	1	<i>x</i>	<i>u</i>
0	0	\emptyset	0	\emptyset
1	\emptyset	1	1	\emptyset
<i>x</i>	0	1	<i>x</i>	<i>u</i>
<i>u</i>	\emptyset	\emptyset	<i>u</i>	<i>u</i>

Figure 3.3 Modified intersection operator

Evaluation of 3-valued Logic

- Evaluate $f(x_1, x_2, \dots, x_n)$ for (v_1, v_2, \dots, v_n)
 - Evaluate the vector with a primitive cube of f with the intersection operation
 - If the intersection is consistent, the output value of the primitive cube is the output of the vector on f . Otherwise, the vector leads to u .

Example: For a AND,

- What primitive cube is compatible with $(u, 0)$?
- What primitive cube is compatible with $(u, 1)$?

3-valued Logic – Pessimistic

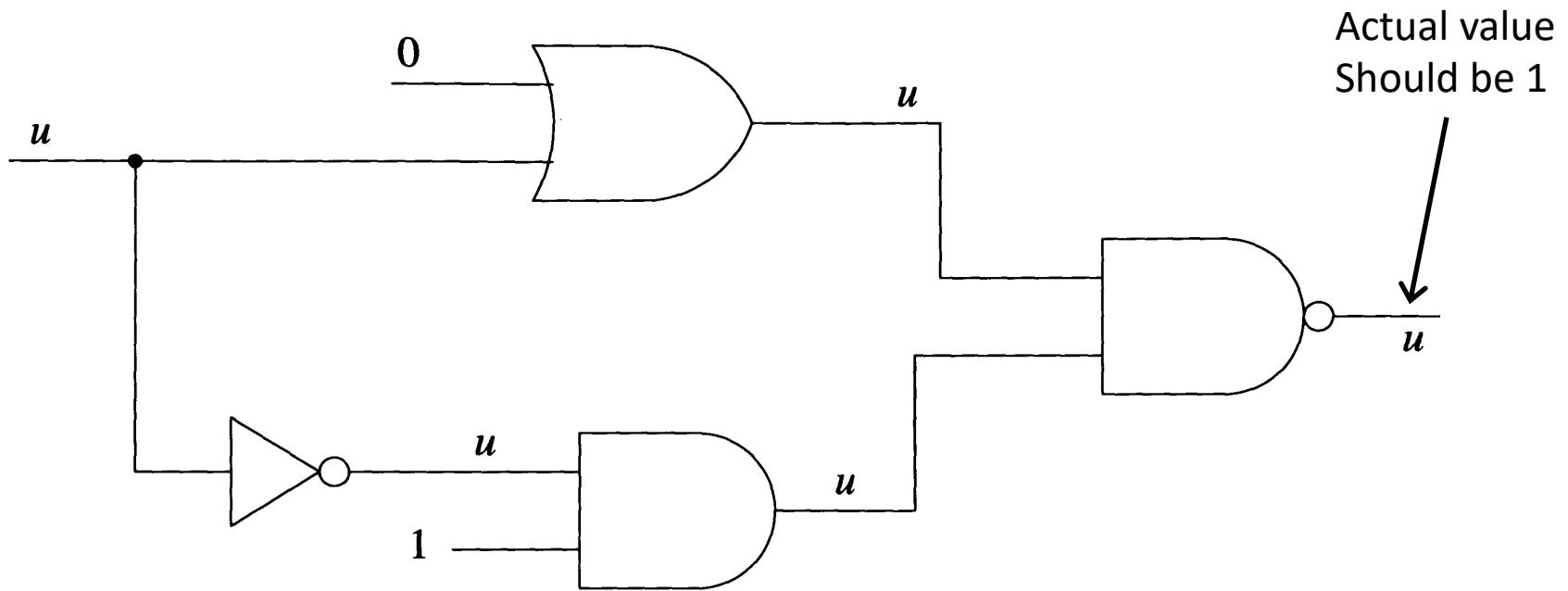


Figure 3.4 Pessimistic result in 3-valued simulation

Pessimistic result due to reconvergent fanout!!

3.5 Compiled Simulation

- Compiled model becomes part of the simulator
- In extreme cases, compiler model is the simulator!!
- Example: Assume data F to FF satisfies setup time constraint. Therefore, we can ignore gate delays

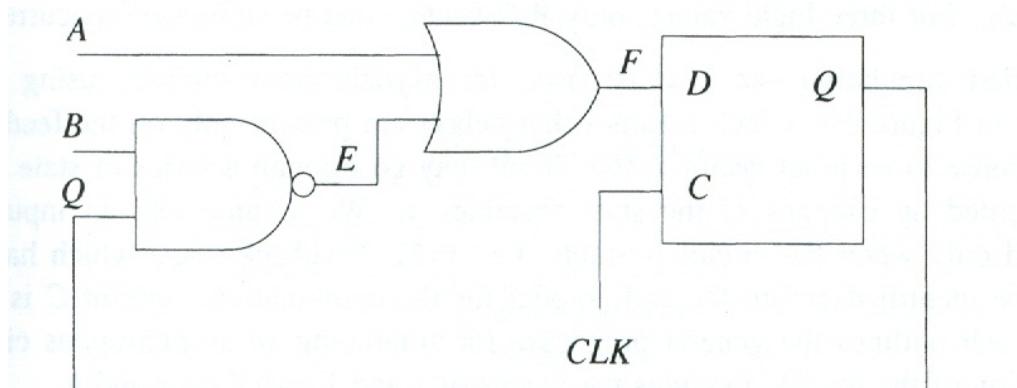


Figure 3.7 Synchronous circuit

Compiled Simulation - Example

- Simulate level by level

Note: for every input vector every element is evaluated

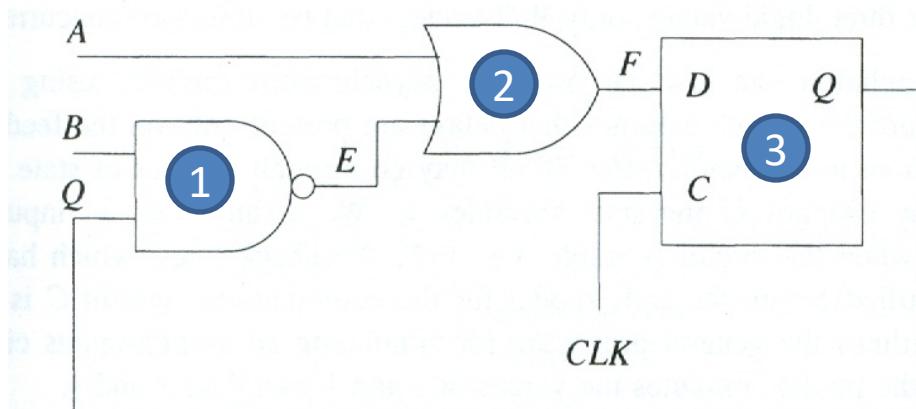


Figure 3.7 Synchronous circuit

Assembly Code

```
LDA    B  
AND    Q  
INV  
STA    E  
OR     A  
STA    F  
STA    Q
```

Compiled Simulation: Asynchronous Circuits

- For asynchronous circuits, delay modeling in the feedback path is necessary

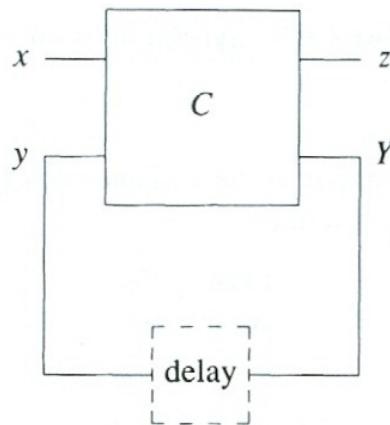


Figure 3.8 Asynchronous circuit model

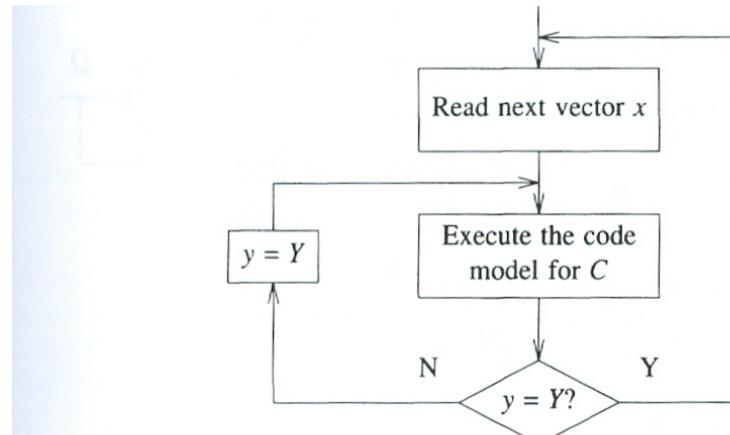
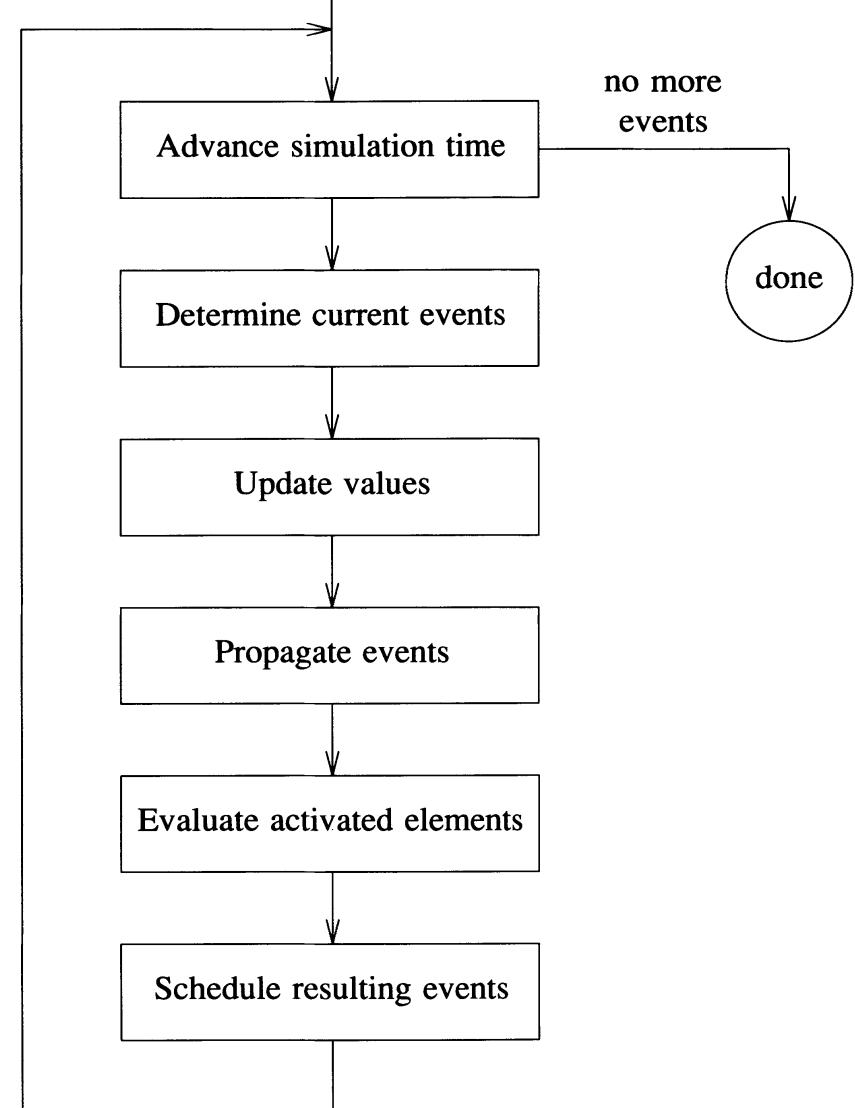


Figure 3.9 Asynchronous circuit simulation with compiled-code model

3.6 Event-driven simulation

- **Event** – Change in the value of a signal
- **Active Signal** – Signal with changing value at some arbitrary time
- **Activated element** – an element with at least one input signal with an event
- **Evaluation** – process of determining the output values of an element



Event-driven simulation: Terminology

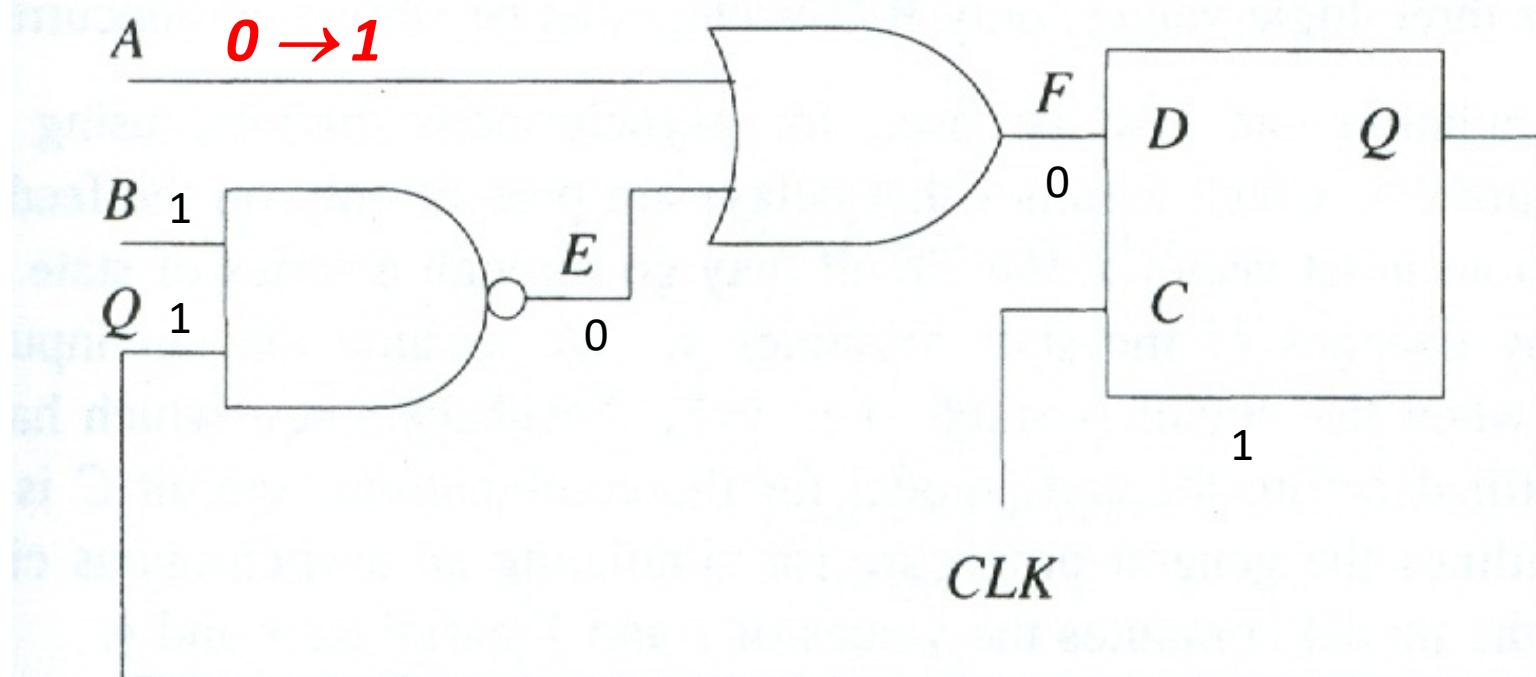
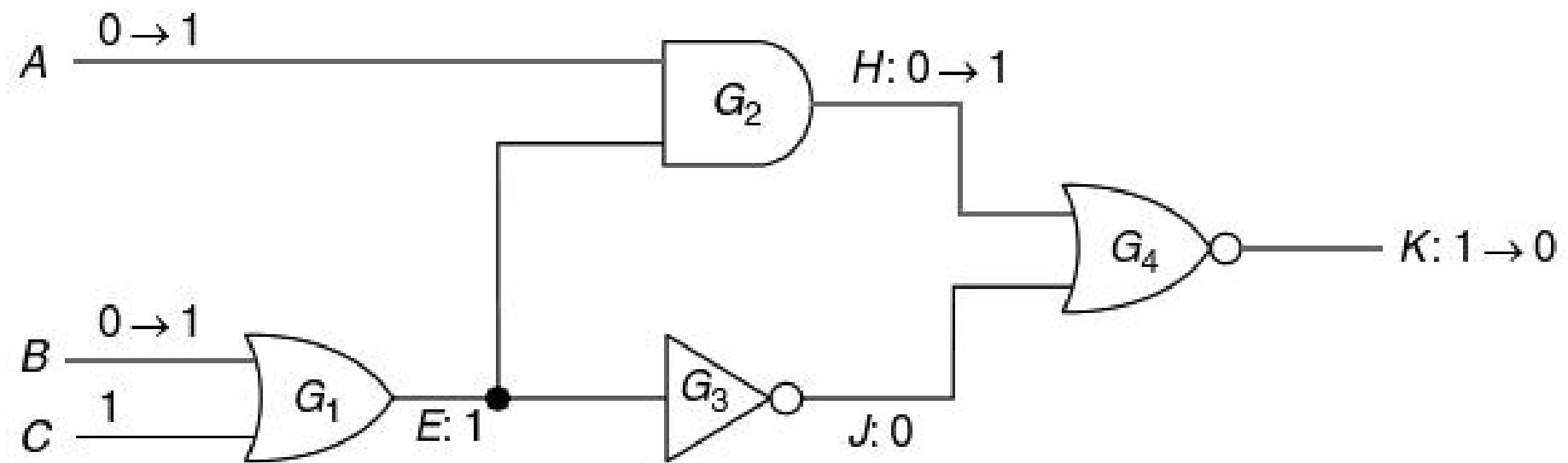


Figure 3.7 Synchronous circuit

Event-Driven Simulation: Example



Event-Driven Simulation

- Uses structural model to propagate events
- Stimulus file – changes in the values on primary inputs
- Events on other lines are produced by evaluations of activated elements
- Event occurs at a certain (simulated) time
- Applied stimuli – event sequences in time
- Can have events scheduled in future i.e., *pending* events
- Simulator maintains an *event list*

3.7. Delay Modeling

- Every gate introduces delay on signals propagating through it
- Behavior of a gate can be separated into functional and timing
- Delay Models
 - *Zero delay* – All gates have zero delay; used for functional verification
 - *Unit Delay* – All gates have delay = 1
 - *Real Delay* – All gates have different delays

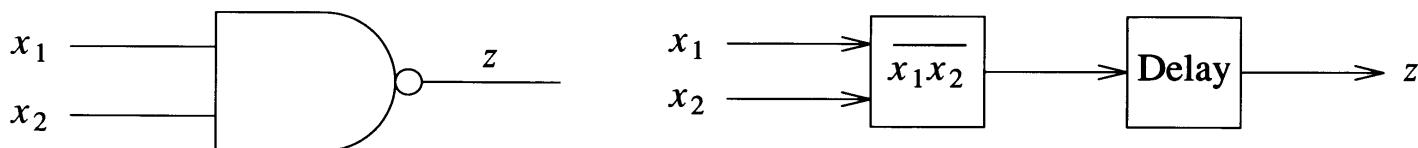
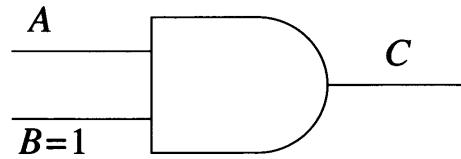


Figure 3.13 Separation between function and delay in modeling a gate

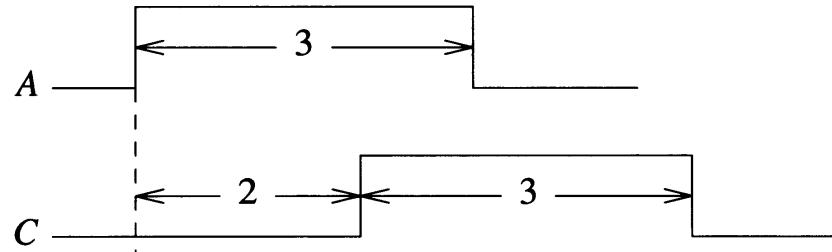
Delay Models

- **Transport Delay**
 - Interval separating output change from the associated input change
- **Inertial Delay**
 - All circuits require energy to switch states
 - Energy in a signal is a function of its amplitude and duration.
 - Therefore, an input signal of minimum duration is required to cause a change on the output.
 - Minimum duration is known as input inertial delay d ,

Example: Transport & Inertial Delay



Transport



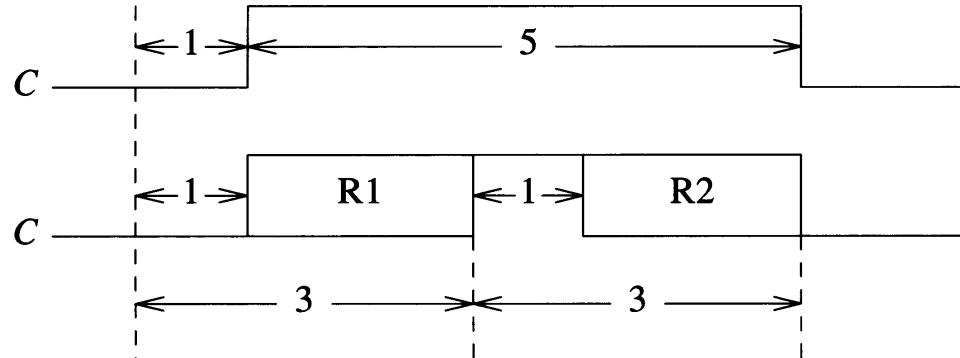
(a) $d=2$

Rise & Fall



(b) $d_r=1, d_f=3$

Ambiguous

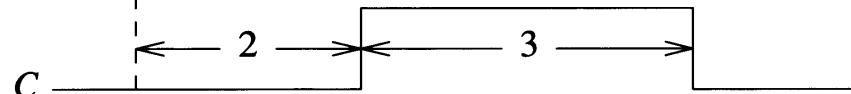


(c) $d_m=1, d_M=3$

Inertial

(d) $d_I=4$

Inertial



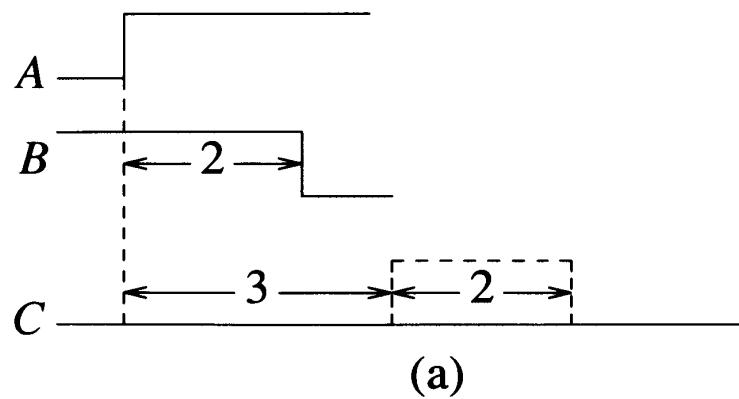
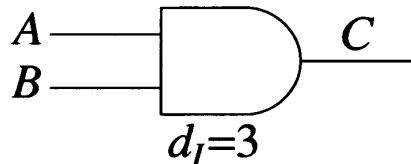
(e) $d_I=2, d=2$

Inertial Delay

- Inertial delay can be modeled at the input or output of a gate
- Input inertial delay
 - No input $< d$, can propagate through circuit
- Output inertial delay
 - No output $< d$, can be generated

Output Inertial Delay

An output pulse caused by close input transitions



Example for which input and output inertial delay models will give different results

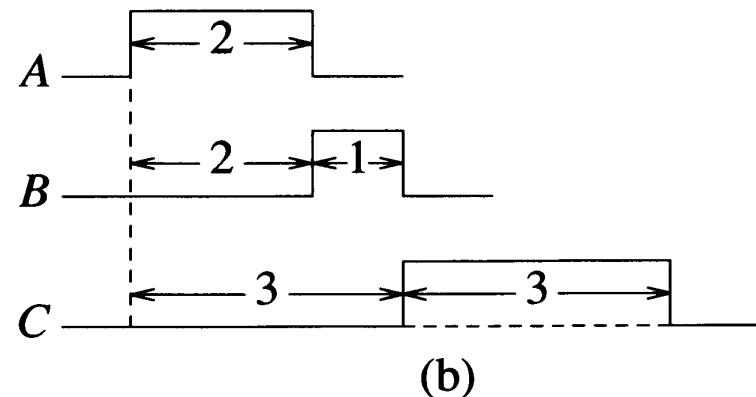
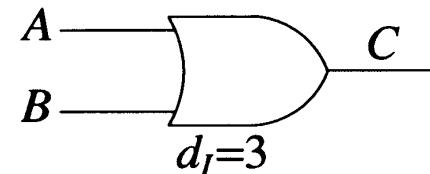
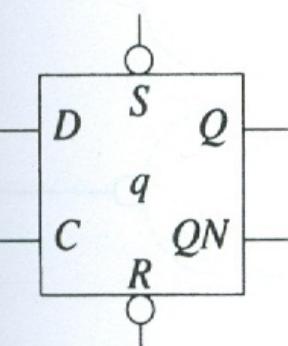


Figure 3.15 Output inertial delay

Delay Modeling - FF

- Delays in FF more complex
- Rise, Fall, input-to-output (I/O) delays

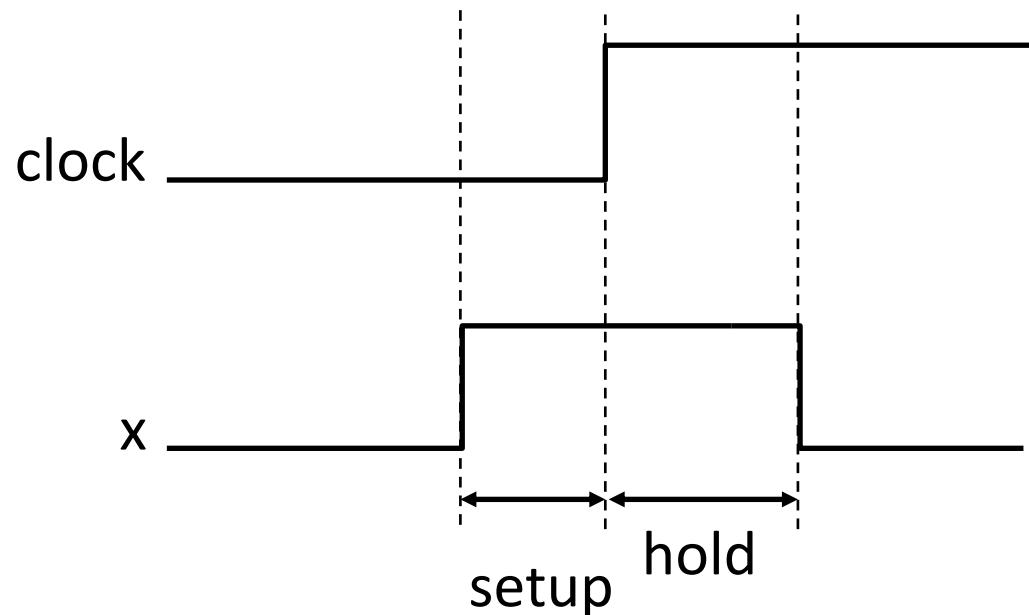


q	S	R	C	D	Q	QN	Delays	
0	0	1	x	x	1	0	$d_{S/Q} = 4$	$d_{S/QN} = 3$
1	1	0	x	x	0	1	$d_{R/Q} = 3$	$d_{R/QN} = 4$
1	1	1	\uparrow	0	0	1	$d_{C/Q}^f = 8$	$d_{C/QN}^r = 6$
0	1	1	\uparrow	1	1	0	$d_{C/Q}^r = 6$	$d_{C/QN}^f = 8$
x	0	0	x	x	u	u		

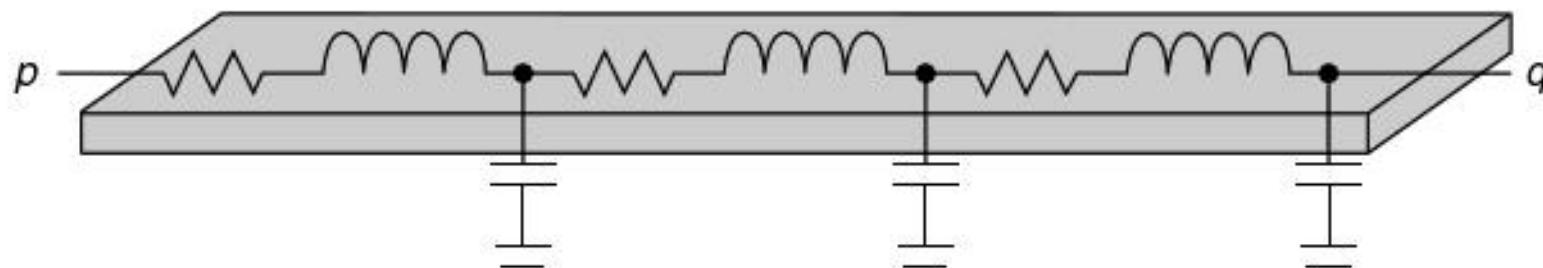
Figure 3.16 I/O delays for a D F/F

Race Condition

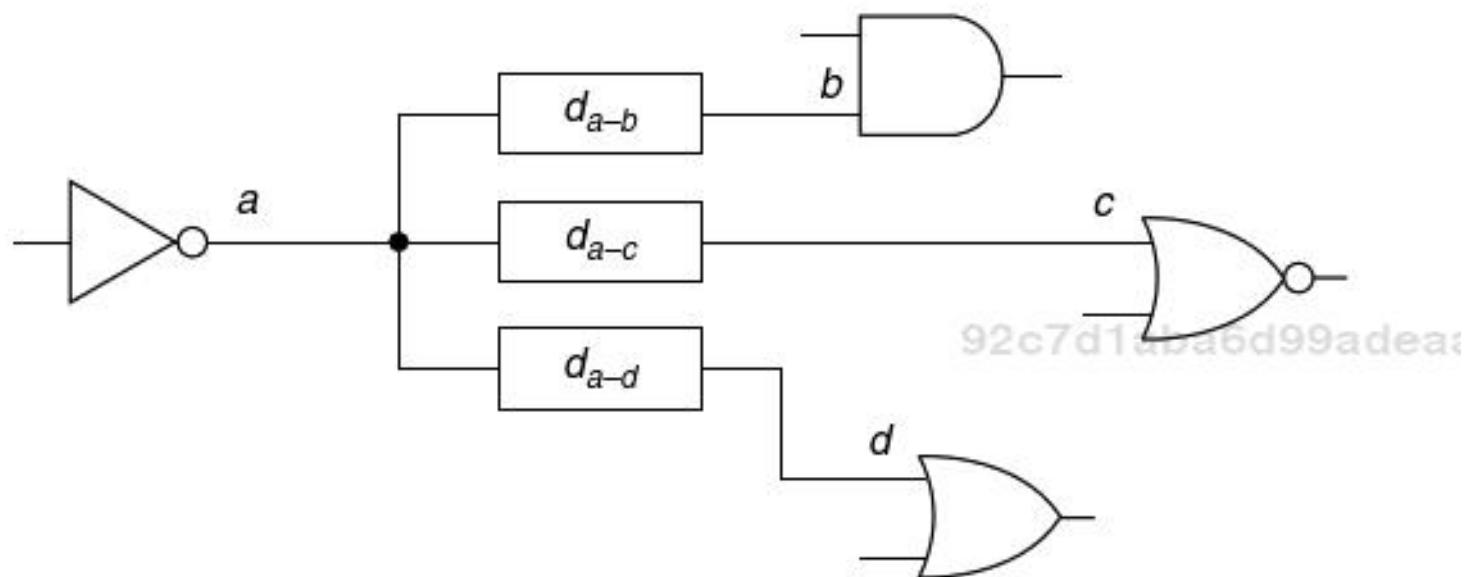
- Setup time
 - Minimum interval preceding an active clock edge.
- Hold time
 - Minimum interval following an active clock edge.



Wire Delay



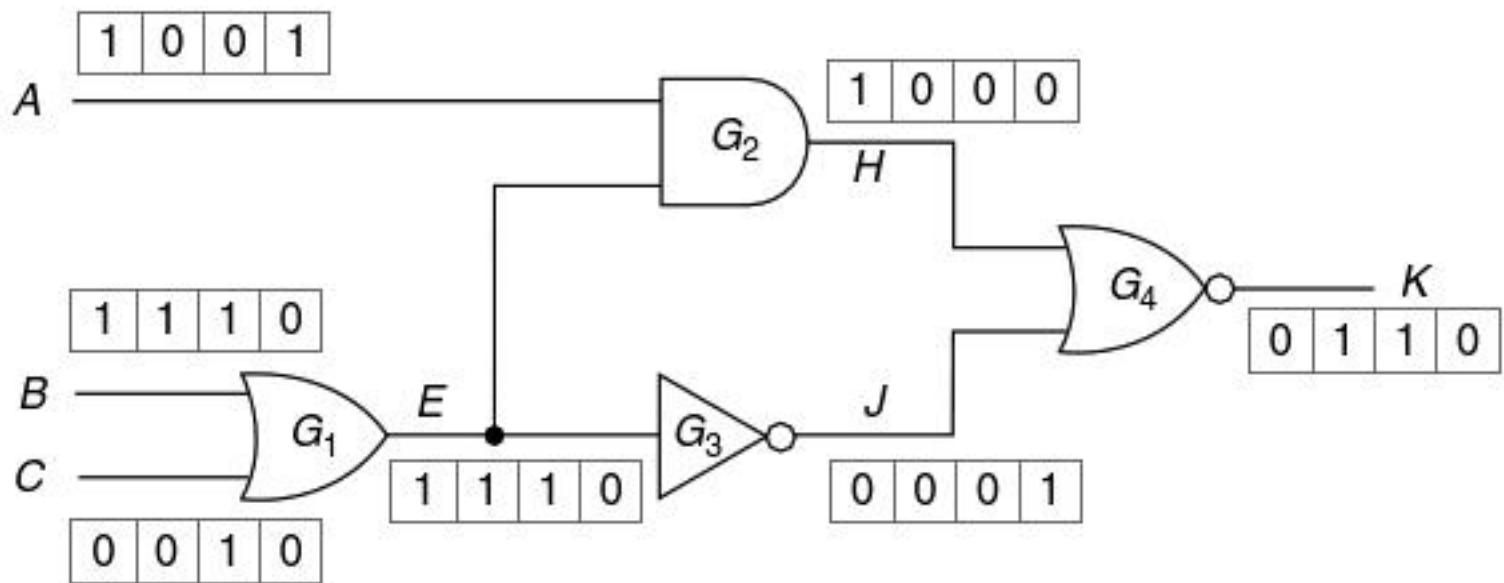
(a) Distributed wire delay model



(b) Fanout delay modeling

3.8 Evaluation of Logic Elements

- A process computing output of a logic element based on its inputs and current state.
 - Truth table is an obvious choice.
 - More advanced techniques to speed up simulation:*input scanning, input counting, parallel simulation.*



Zoom Tables

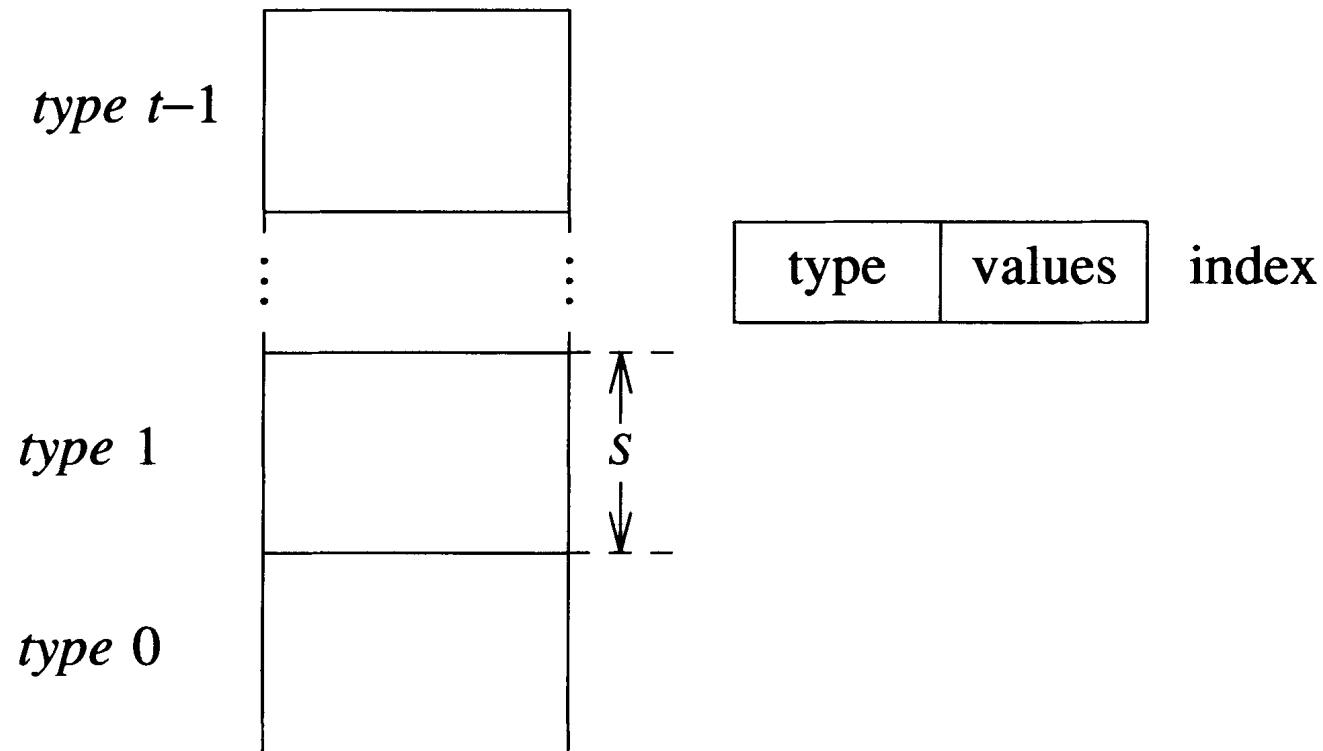


Figure 3.19 Zoom table structure

Logic Gate Characterization

c	x	x	$ $	$c \oplus i$
x	c	x	$ $	$c \oplus i$
x	x	c	$ $	$c \oplus i$
\bar{c}	\bar{c}	\bar{c}	$ $	$\bar{c} \oplus i$

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

Figure 3.20 Primitive cubes for a gate with controlling value c and inversion i

- Controlling value c – uniquely determines output
- Inversion i

Input Scanning

```
evaluate ( $G$ ,  $c$ ,  $i$ )
begin
     $u\_values$  = FALSE
    for every input value  $v$  of  $G$ 
        begin
            if  $v = c$  then return  $c \oplus i$ 
            if  $v = u$  then  $u\_values$  = TRUE
        end
        if  $u\_values$  return  $u$ 
    return  $\bar{c} \oplus i$ 
end
```

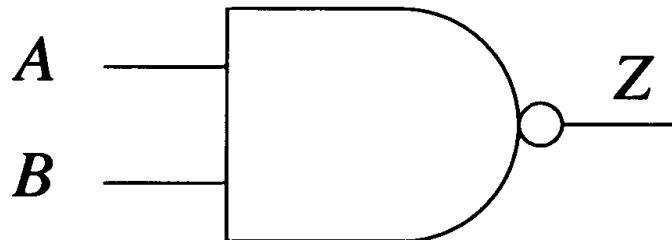
Figure 3.21 Gate evaluation by scanning input values

Input Counting

```
evaluate ( $G$ ,  $c$ ,  $i$ )
begin
    if  $c\_count > 0$  then return  $c \oplus i$ 
    if  $u\_count > 0$  then return  $u$ 
    return  $\bar{c} \oplus i$ 
end
```

Figure 3.22 Gate evaluation based on input counting

3.9 Hazard Detection



Ideally,

$$A: 0 \rightarrow 1, B: 1 \rightarrow 0, Z: 1 \rightarrow 1$$

Possibly;

$$\begin{aligned} A: 0 &\rightarrow 1 \rightarrow 1, \\ B: 1 &\rightarrow 1 \rightarrow 0, \\ Z: 1 &\rightarrow 0 \rightarrow 1 \end{aligned}$$

Hazard Detection

- If wire S changes at times t and $t+1$, assume its value between t and $t+1$ is unknown.
- Evaluate the logic element, check if u appears on output between t and $t+1$.
- Ex: A: $0 \rightarrow u \rightarrow 1$, B: $1 \rightarrow u \rightarrow 0$, Z: $1 \rightarrow u \rightarrow 1$, hazard!

3.9 Hazard Detection

- Static hazards under different delay models.

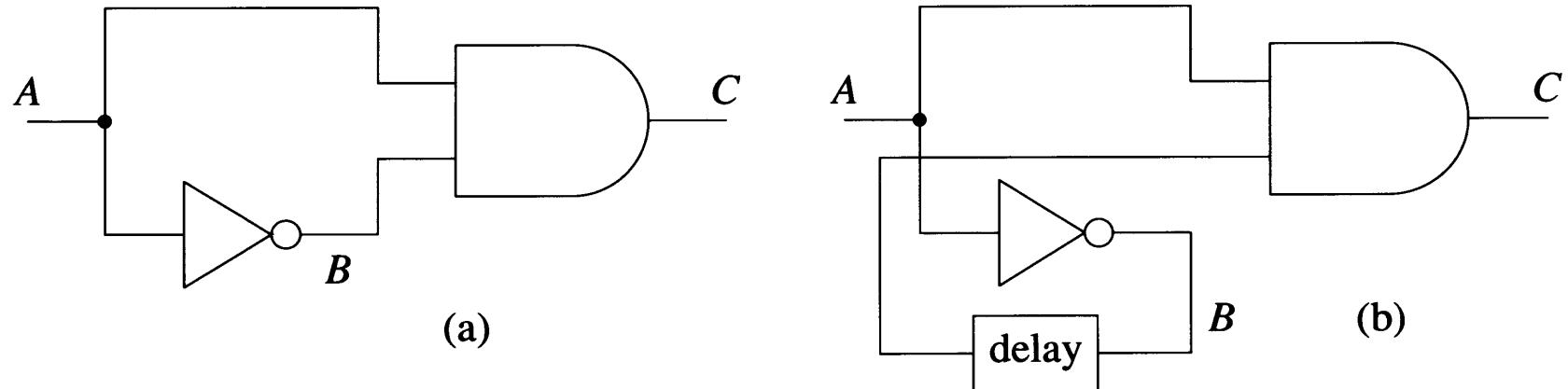


Figure 3.10 (a) Circuit used as pulse generator (b) Correct model for compiled simulation

Consider input sequence $A = 010$ for (a).

- For 0-delay model, $B = 101$, and $C = 000$ – no static hazard
- For unit-delay model, $B = 1101$, and $C = 0010$ – static hazard

3.10 Gate-level Event-Driven Simulation

Assume transition independent nominal transport delays

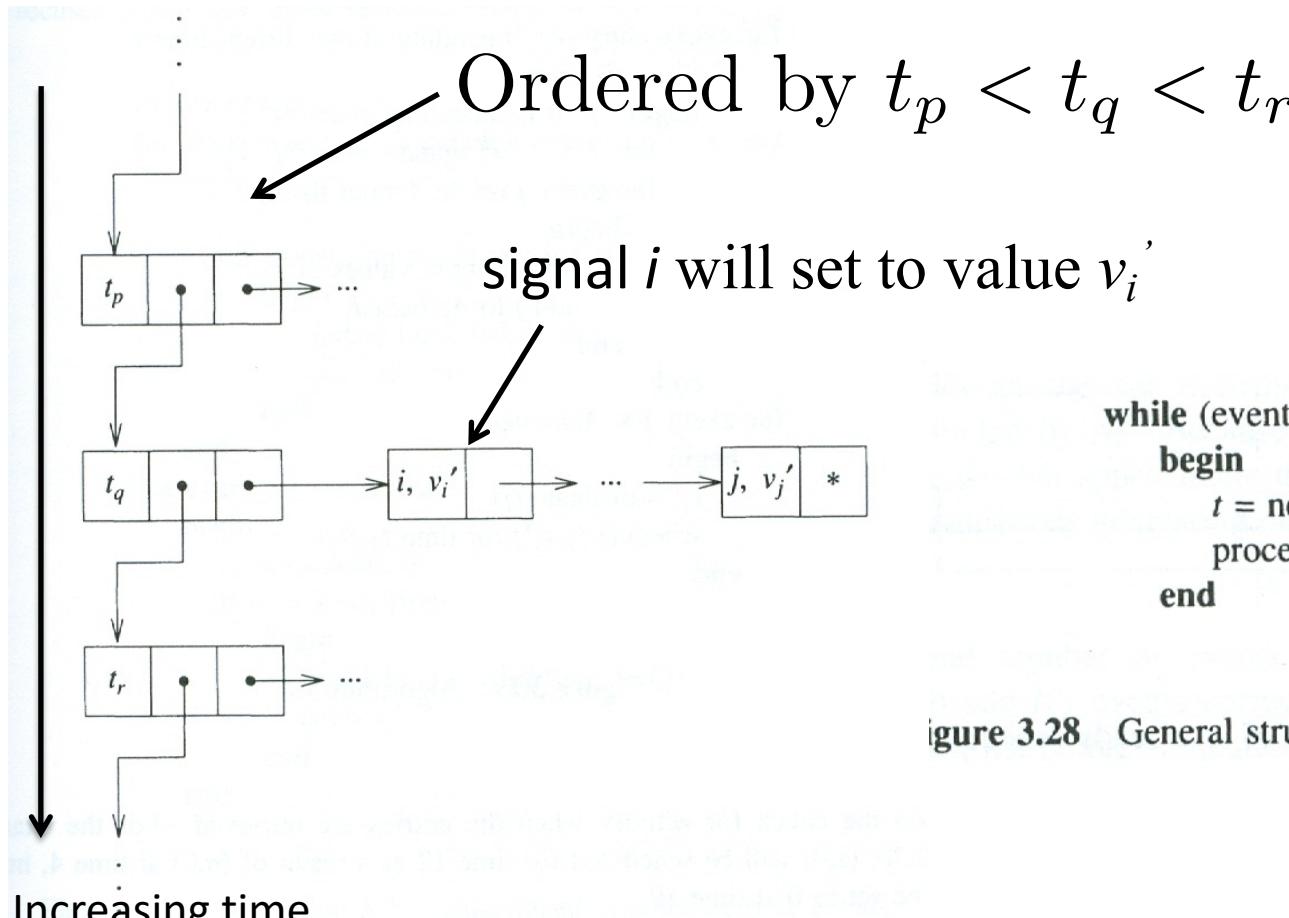


Figure 3.27 Event list implemented as a linked list structure

```
while (event list not empty)
begin
    t = next time in list
    process entries for time t
end
```

figure 3.28 General structure for event-driven simulation

Algorithm 3.1 - Basic Algorithm

```
Activated = Ø /* set of activated gates */  
for every entry  $(i, v_i')$  pending at the current time  $t$   
    if  $v_i' \neq v(i)$  then  
        begin /* it is indeed an event */  
             $v(i) = v_i'$  /* update value */  
            for every  $j$  on the fanout list of  $i$   
                begin  
                    update input values of  $j$   
                    add  $j$  to Activated  
                end  
    end  
for every  $j \in \text{Activated}$   
    begin  
         $v_j' = \text{evaluate}(j)$   
        schedule  $(j, v_j')$  for time  $t+d(j)$   
    end
```

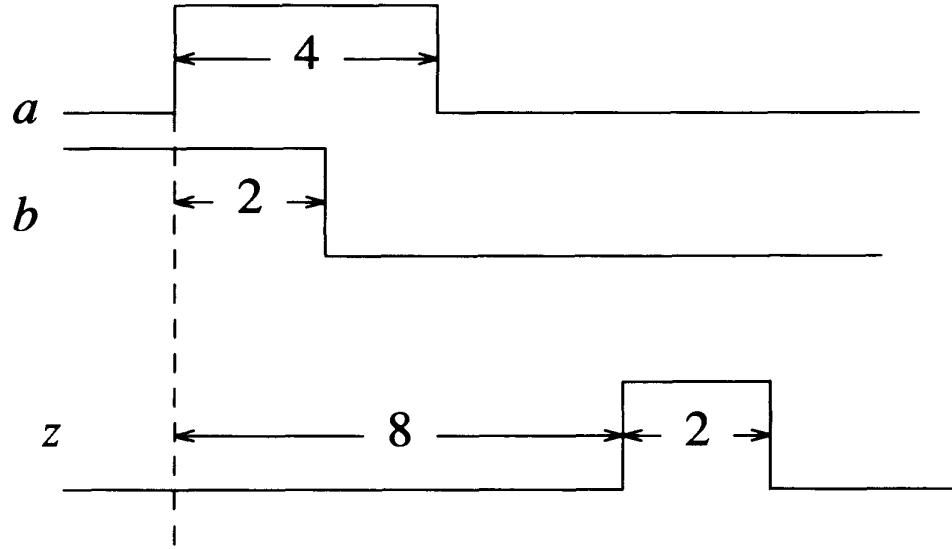
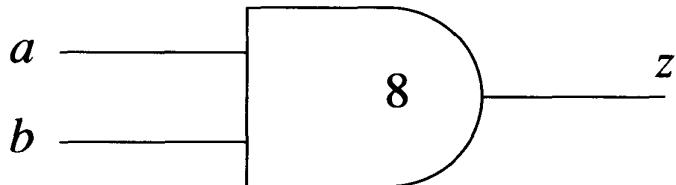
Pass 1:

Find all activated gates for all events at current time

Pass 2:

For each activated gate, evaluated, schedule new events

Example



- Algorithm 3.1 will schedule two $(z, 0)$ events at times $t=10$ and $t=12$ (wasteful)
- Does not guarantee scheduling of *only events*

Algorithm 3.3 - One Pass Strategy

for every event (i, v_i') pending at the current time t

begin

$v(i) = v_i'$

for every j on the fanout list of i

begin

update input values of j

$v_j' = \text{evaluate}(j)$

if $v_j' \neq lsv(j)$ **then**

begin

schedule (j, v_j') for time $t+d(j)$

$lsv(j) = v_j'$

end

end

end

Avoid redundant events

Avoids building Activated element list

“ lsv ” is last scheduled value

Problem with Algorithm 3.3

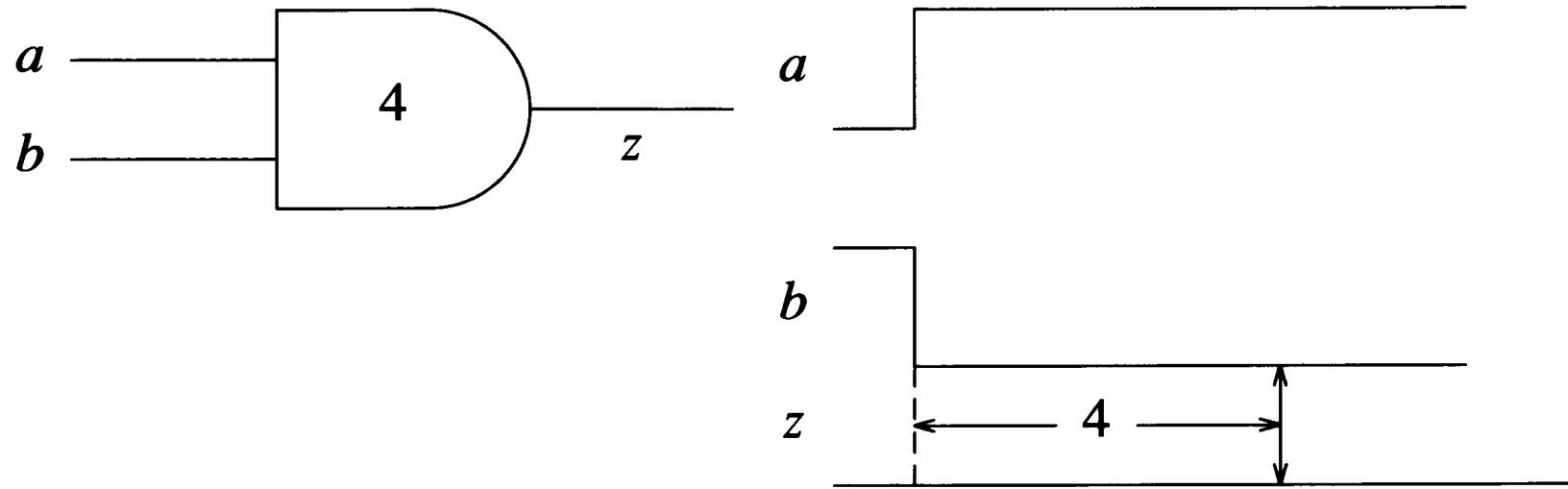


Figure 3.33 Processing of multiple input changes by Algorithm 3.3

When a changes before b , a zero-spike is generated at time 4!

Algorithm 3.4 – Suppression of zero-width spikes

for every event (i, v_i') pending at the current time t

begin

$v(i) = v_i'$

for every j on the fanout list of i

begin

update input values of j

$v_j' = \text{evaluate}(j)$

if $v_j' \neq lsv(j)$ **then**

begin

$t' = t + d(j)$

if $t' = lst(j)$

then cancel event $(j, lsv(j))$ at time t'

schedule (j, v_j') for time t'

$lsv(j) = v_j'$

$lst(j) = t'$

end

end

end

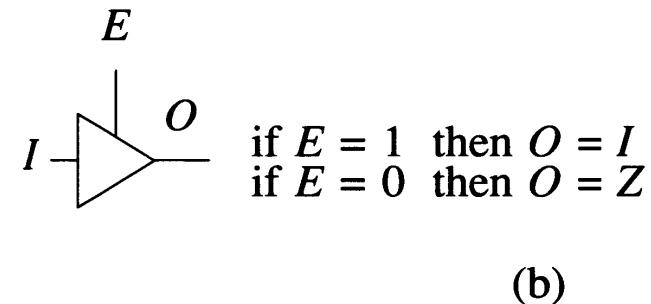
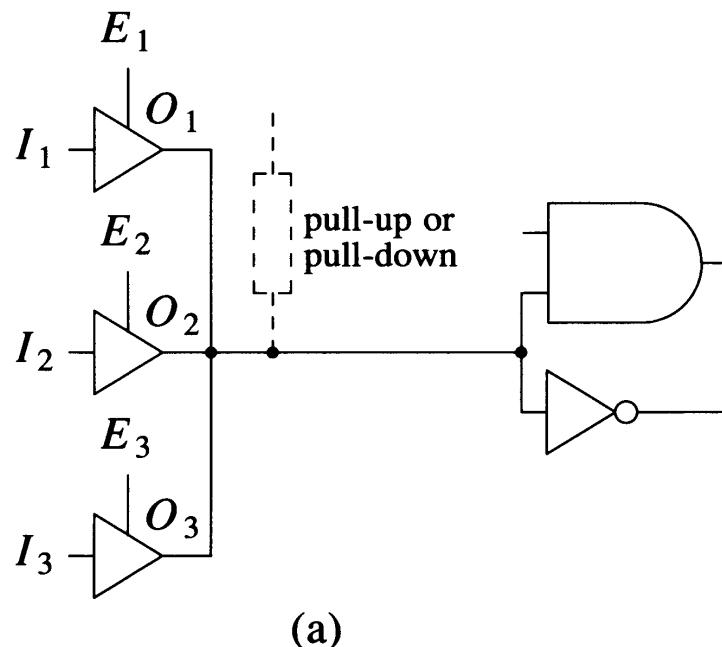
If two events are
scheduled at the same
time, cancel the
previously scheduled one.

lst: Last Scheduled Time



Simulating Tri-state Logic

- Only one driver should drive the bus
- Multiple drivers → bus conflict (can damage the bus)
- If all drivers in Z state → bus is either pulled up or down



Z: High Impedance

Figure 3.36 (a) Bus (b) bus driver

Logic Function – Bus

	0	1	Z	u
0	0 ¹	u ²	0	u ³
1	u ²	1 ¹	1	u ³
Z	0	1	Z ⁴	u
u	u ³	u ³	u	u ³

Figure 3.37 Logic function of a bus with two inputs (1 — report multiple drivers enabled; 2 — report conflict; 3 — report potential conflict; 4 — transform to 1(0) if pull-up (pull-down) present)

Logic Function – Bus Driver

		E		
		0	1	u
		0	0	{0,Z}
I	1	Z	1	{1,Z}
	u	Z	u	{u,Z}

Figure 3.38 Truth table for a bus driver

Compiled vs. Event-driven

Compiled	Event-driven
Applicable only to synchronous circuits	Both Synchronous and asynchronous
Allows inputs to change only when circuit is stable	Allows real-time inputs
Inefficient	Efficient – simulates only when required
Cannot handle races/hazards	Can handle races/hazards

Other delay models

- We will not consider the following delay models.
 - Rise & Fall Delay Models
 - Straightforward extension of Algorithms 3.1-3.4. How?
 - Inertial Delay Model
 - Ambiguous Delays
 - Oscillation Control

Summary

- Logic Simulation provides a mechanism to verify the behavior of the system
- Two types of simulation
 - Compiled Simulation
 - Event Driven Simulation (more efficient and commonly used)
- Delay models
 - Transport, Inertial, Rise, Fall, Ambiguous

Backup

Compiled Simulation: Asynchronous Circuits

- Delay modeling in the feedback path is necessary
- Compiled sim model of (a) will not give correct results!

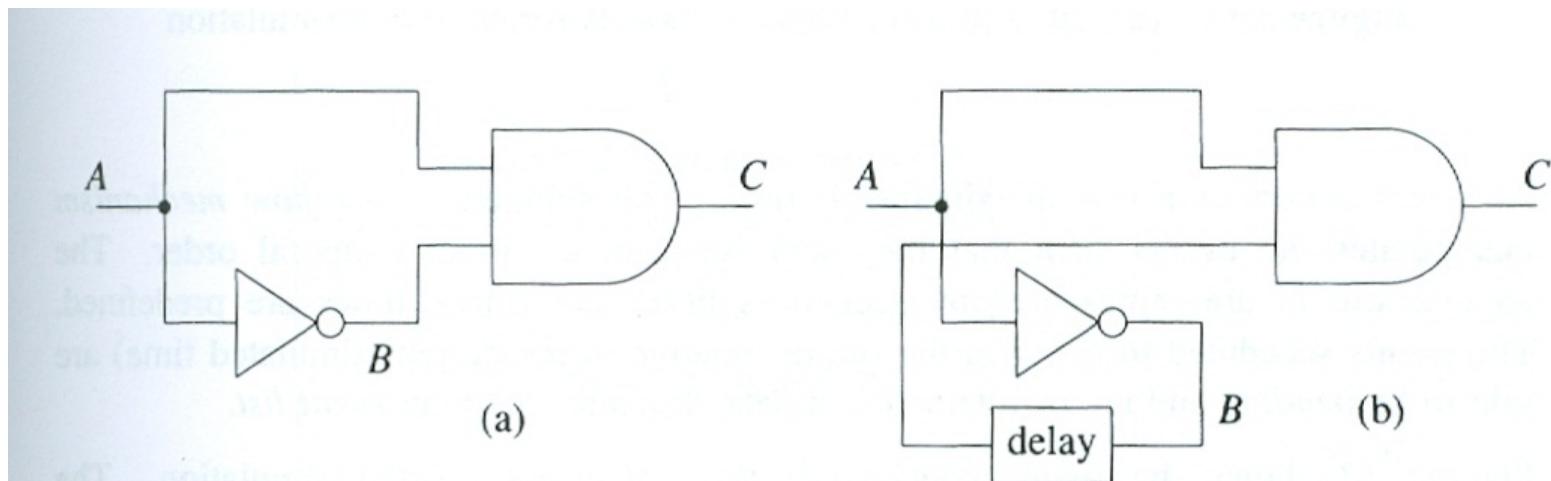
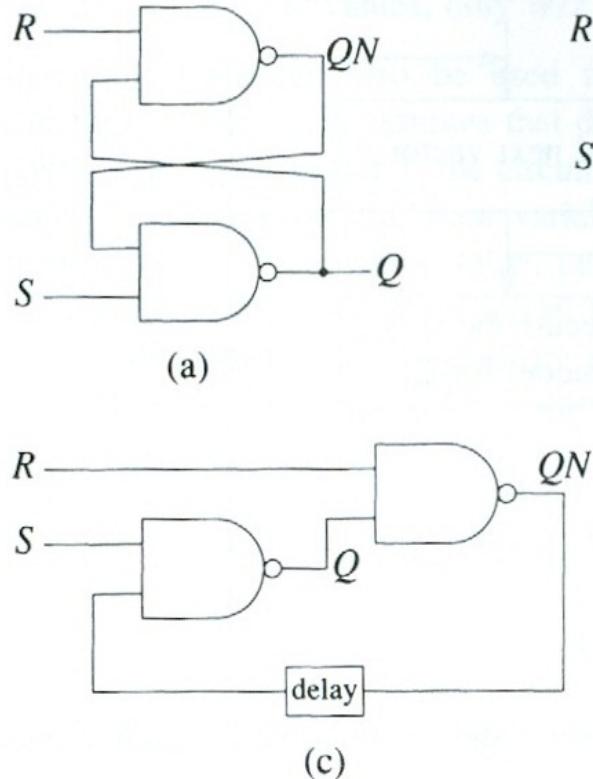


Figure 3.10 (a) Circuit used as pulse generator (b) Correct model for compiled simulation

Races/Hazards



- Models (b) and (c) differ in which feedback path has delay
- For same input $RS = 00 \rightarrow 11$, we will get different results!
- This example illustrates that compiled simulation cannot handle races

Figure 3.11 (a) Latch (b)&(c) Possible models for compiled simulation

Algorithm 3.2 - Updated Algorithm

```
Activated = Ø
for every event  $(i, v_i')$  pending at the current time  $t$ 
begin
     $v(i) = v_i'$ 
    for every  $j$  on the fanout list of  $i$ 
    begin
        update input values of  $j$ 
        add  $j$  to Activated
    end
end
for every  $j \in \text{Activated}$ 
begin
     $v_j' = \text{evaluate}(j)$ 
    if  $v_j' \neq lsv(j)$  then
        begin
            schedule  $(j, v_j')$  for time  $t+d(j)$ 
             $lsv(j) = v_j'$ 
        end
    end
end
```

*lsv: last
scheduled
value*

Figure 3.31 Algorithm 3.2 — guaranteed to schedule only events

CIS 4930. Digital System Testing

Fault Modeling

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

Overview

- Logic Fault Models
- Fault Detection & Redundancy
- Fault Equivalence & Fault Location
- Fault Dominance
- Single Stuck-Fault (SSF) Model
- Multiple Stuck-Fault (MSF) Model
- Summary

Recap: Testing Big Picture

- A circuit defect leads to a fault.
- A fault can cause a circuit error.
- A circuit error can result in a circuit failure.
- Testing a circuit:
 - Apply test vectors to the circuit inputs.
 - Compare circuit output responses to correct ones.
- Exhaustive testing
 - 2^n test vectors required for a n -input comb. circuit.
 - Difficult for comb. circuit, impossible for seq. circuit.

Recap: Testing Big Picture

- Goal: find a small set of test vectors that target specific faults.
 - Ideally, no redundant test vectors for the same fault.
 - The set contains enough test vectors to uncover all target faults.
- Impossible to achieve 100% fault coverage.
 - Due to undetectable faults.

Recap: Physical Faults

Recall that we have 4 types of errors

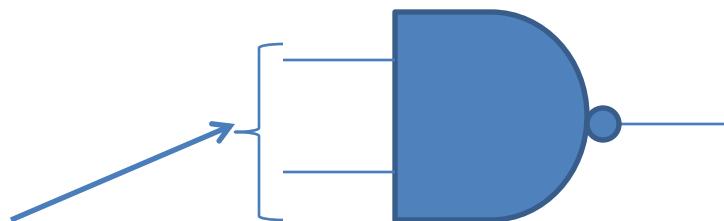
- Design Errors
 - Fabrication Errors
 - Fabrication Defects
 - Physical Failures
- 
- Physical Faults**

Physical Faults can be:

- Permanent
- Intermittent
- Transient

Recap: Logical Faults

- Physical faults are difficult to handle mathematically
- Therefore, for a physical fault we identify equivalent logical fault
 - Example:



Physical fault
- Inputs are shorted



Logical fault
- Gate now behaves as an inverter

4.1 Logic Fault Models

- Fault model: representation of physical faults and their natures at logic level.
- Recall that
Behavior = Function + Timing
- Therefore, we can talk about two types of faults:
 - **Logical Faults** – modify circuit logic function.
 - **Delay Faults** - modify circuit operating speed.
- Our focus will be on Logical Faults

Logical Fault Modeling - Advantages

- Fault analysis becomes a logical problem.
 - Test can start before silicon is available.
- Fault analysis become less complex.
 - *Many physical faults can be modeled by the same logical fault*
- Technology independent
- Tests derived for logical faults may be used for physical faults whose effect
 - not completely understood
 - or too complex to analyze

Structural and Functional Faults

→ Structural faults

- Faults defined on a structural circuit model.
- Effect: modify interconnections

→ Functional faults

- Faults defines on a functional circuit model
- Effect: modify truth table etc.

→ Intermittent & Permanent Faults

- Statistical data on probability of occurrence of transient/intermittent faults are difficult to have.
- Our focus in this discussion is on structural and permanent faults

Single Fault Assumption - Justification

- Assumption – one logical fault in the system.
- Justification
 - Frequent testing strategy (test often so that prob. of multiple faults developing in between too low)
 - Usually tests derived for individual single faults are applicable for detecting multiple faults composed of the single ones.

Structural Fault Models

→ Assumptions:

- Components are fault free and,
- Only interconnections are affected – shorts & opens.

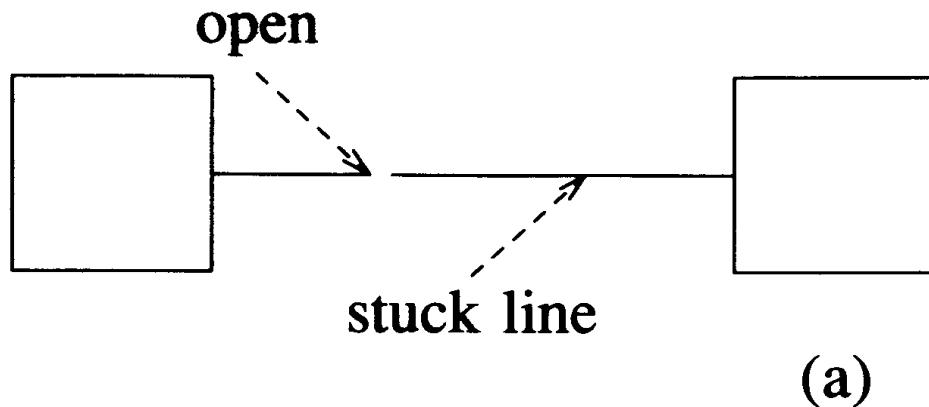
→ **Stuck-at-v Fault:**

- **Short** (with supply/gnd) or **Open** lines behave as “stuck at” fixed logic value v ($v \in \{0,1\}$)

→ **Bridging fault:**

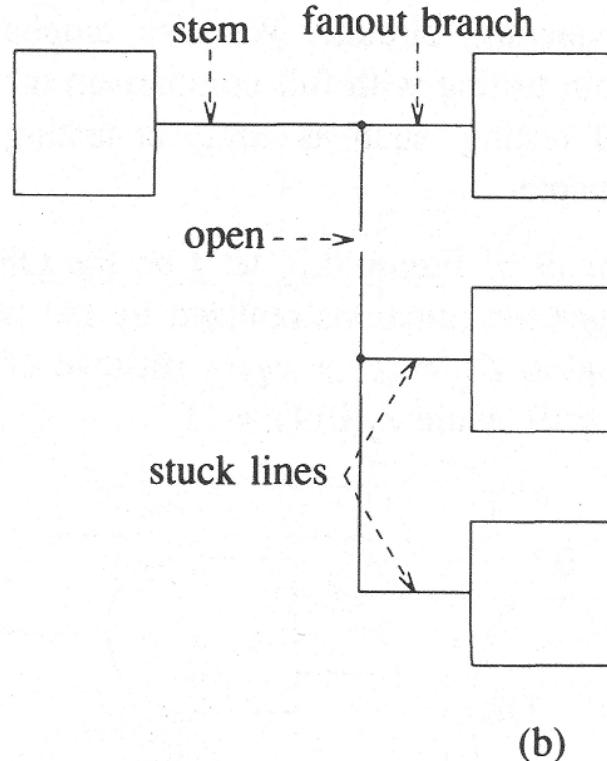
- Short between two lines → usually new logic function (AND or OR bridging)
- We will discuss bridging faults later

Single Stuck Fault – Open Line



- Open on an unidirectional line
- Unconnected input assumes a constant logic value
- Single logical fault, *i-stuck-at-a* can represent
 - Line *i* open
 - Line *i* shorted to Vdd or GND ($a=1$ or $a=0$)
 - Internal fault in component driving line *i*

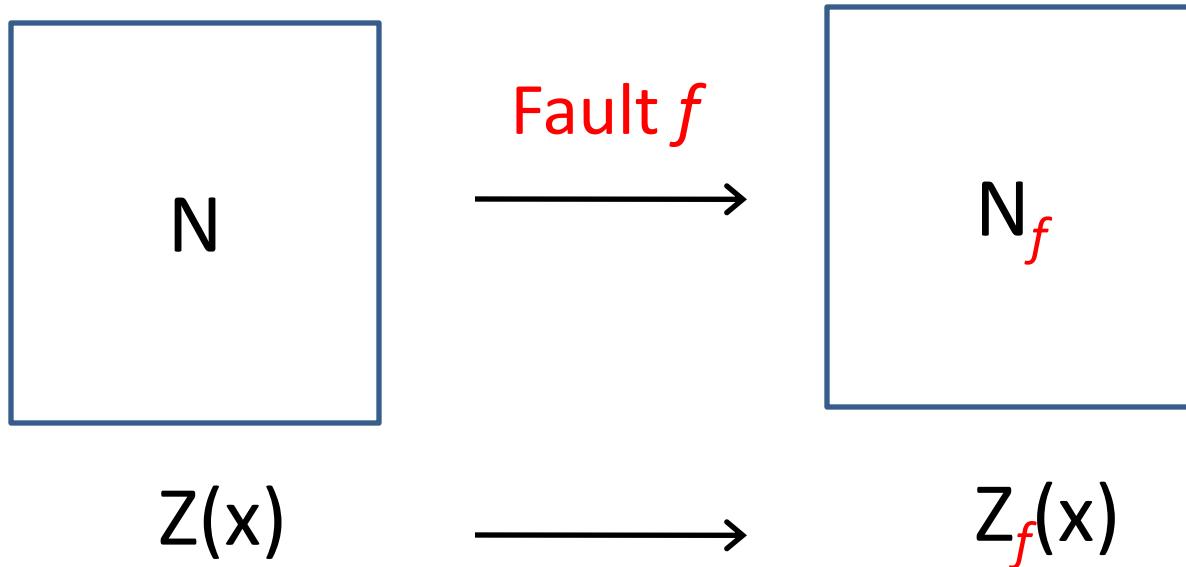
Multiple Stuck Fault



- Single open can result in multiple faults
- Under single fault model, we need to consider faults on all fanout branches separately.

4.2 Fault Detection and Redundancy

Fault Detection – Combinational Circuits



- Input vector t to N results in output response $Z(t)$.
- Test $T = \langle t_1, t_2 \dots t_m \rangle$ will yield $\langle Z(t_1), Z(t_2), \dots Z(t_m) \rangle$

Fault Detection – Comb Circuits

Definition 4.1 A test (vector) t **detects** a fault f
iff

$$z_f(t) \neq z(t)$$

- Note the above is applicable to comb. circuits only.
- Test vectors in T can applied in any order, so T is *set of tests*
- Applicable to *edge-pin testing*
 - Components are assumed to be fault free.

Example 1

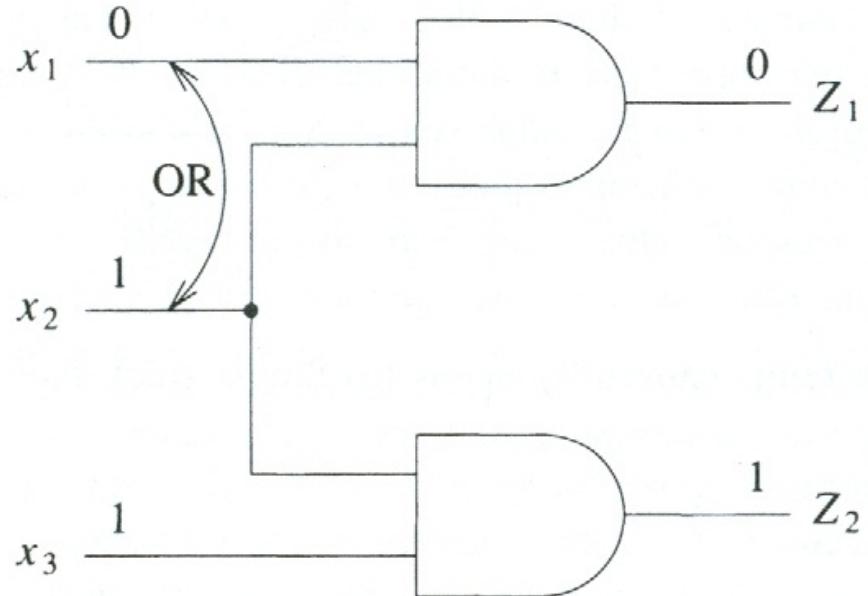


Figure 4.2

Let f = OR-bridging fault between x_1 and x_2

$$Z_1 = \underline{\hspace{2cm}} \quad Z_{1f} = \underline{\hspace{2cm}}$$

$$Z_2 = \underline{\hspace{2cm}} \quad Z_{2f} = \underline{\hspace{2cm}}$$

Give a test vector that detects the fault:

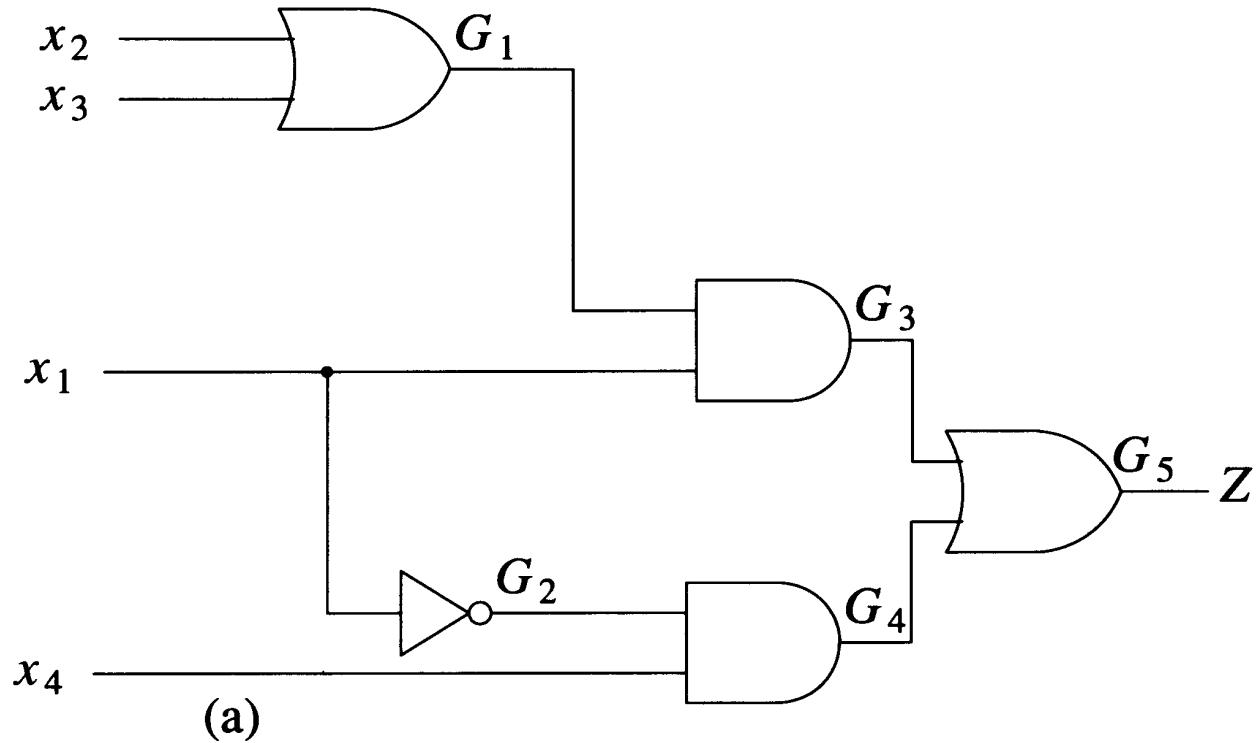
$$\langle x_1, x_2, x_3 \rangle = \underline{\hspace{2cm}}$$

Fault Detection and Test Vectors

- For a single-output circuit, a test t that detects a fault f makes
 - $Z(t) = 0$ and $Z_f(t) = 1$ or
 - $Z(t) = 1$ and $Z_f(t) = 0$
- Thus, the set of all tests that detect f is given by

$$Z_f(t) \oplus Z(t) = 1$$

Example 2

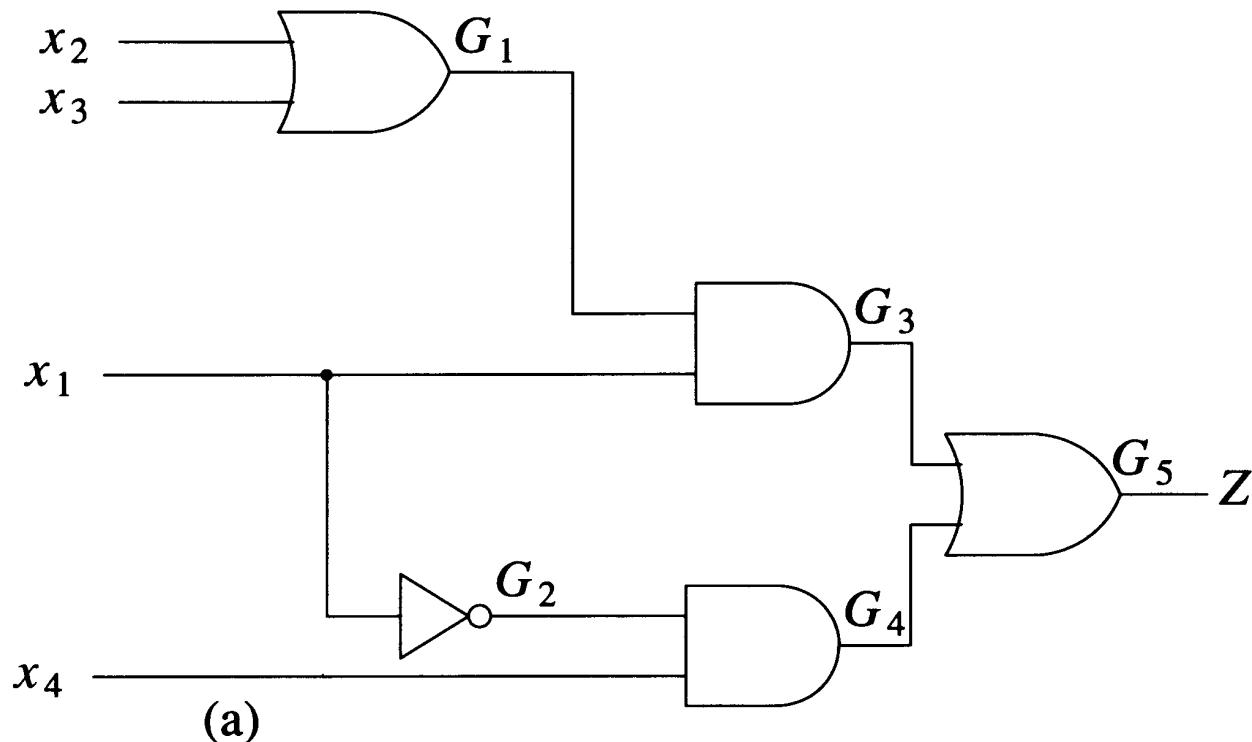


Let the fault f be x_4 s-a-0. Find all test vectors that detect f

Example 3

Let $f = x_4 \text{ s-a-0}$.

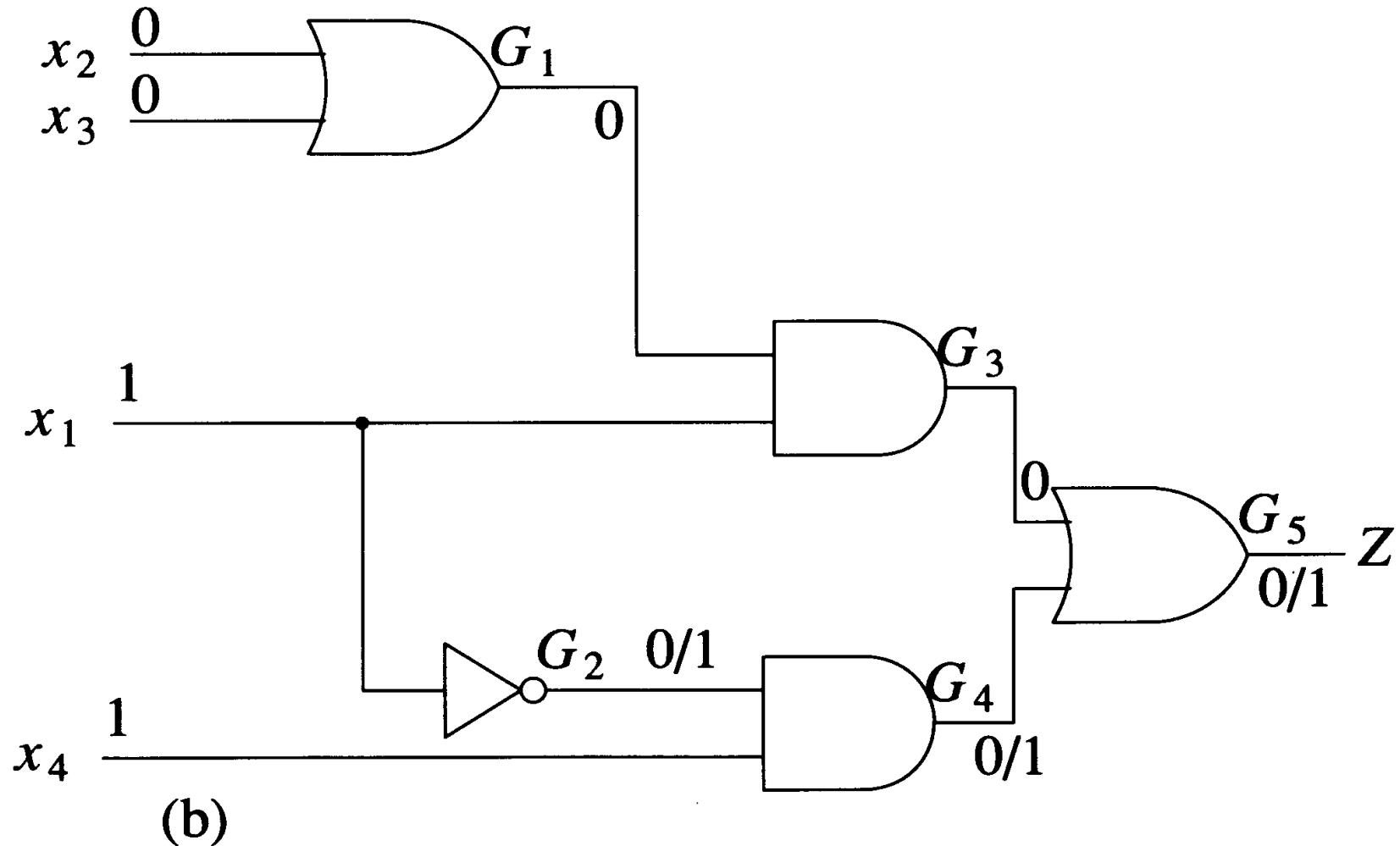
For test vector 1001 that detects f , simulate without and with fault f



Example 3

v/v_f : fault-free/faulty

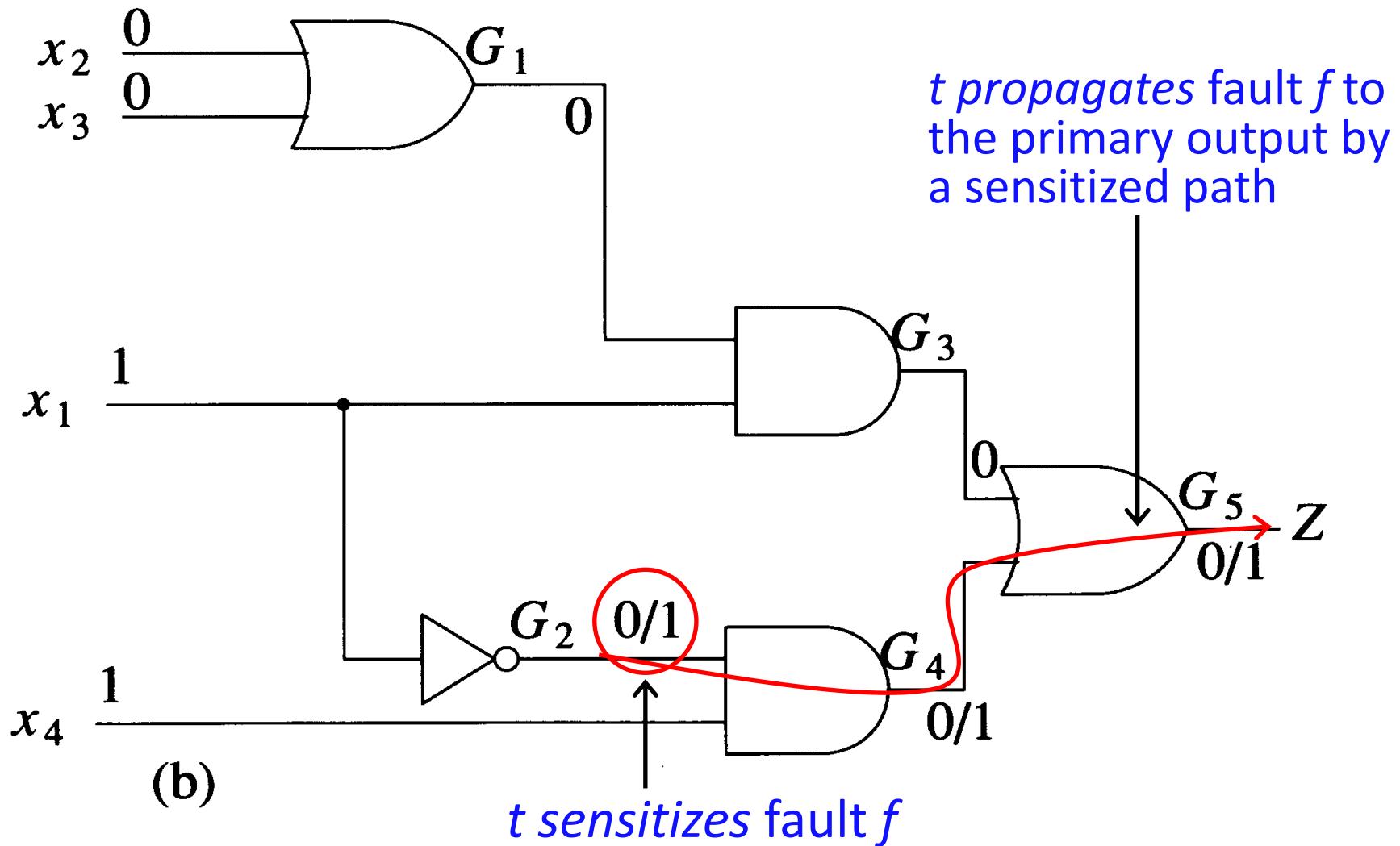
Fault: G2 s-a-1



(b)

Fault – Sensitization

$f = G2 \text{ s-a-1}$, Test vector $t = 1\ 0\ 0\ 1$



Fault Sensitization – Terminology

- **Fault Activation:** A test t activates a fault on a line if it generates an error at the site of the fault.
- **Fault Propagation:** A test t propagates the error to a primary output by creating at least one path from fault site to the primary output.
- **Line Sensitization:** A line whose value under the test t changes in the presence of the fault f is said to *be sensitized to the fault f by the test t*
- **Sensitized Path:** A path composed of sensitive lines

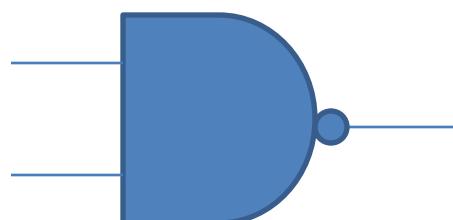
Gate Controlling and Enabling Values

→ **Controlling value c :** If at least one input assumes value c , then the gate's output assumes value

$$\cdot \quad c \oplus i$$

→ **Enabling value \bar{c} :** If all inputs of a gate have the enabling value, then the gate's output assumes the value $\bar{c} \oplus i$.

Example



Control value = _____
Enabling value = _____

NAND: $c=0, i=1$.

Lemma 4.1

Let G be a gate with inversion i and controlling value c , whose output is sensitized to a fault f (by a test t).

1. All inputs of G sensitized to f have the same value (say, a).
2. All inputs of G not sensitized to f (if any) have value \bar{c} .
3. The output of G has value $a \oplus i$.

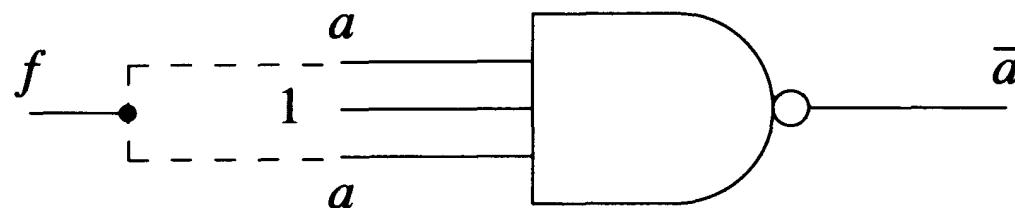


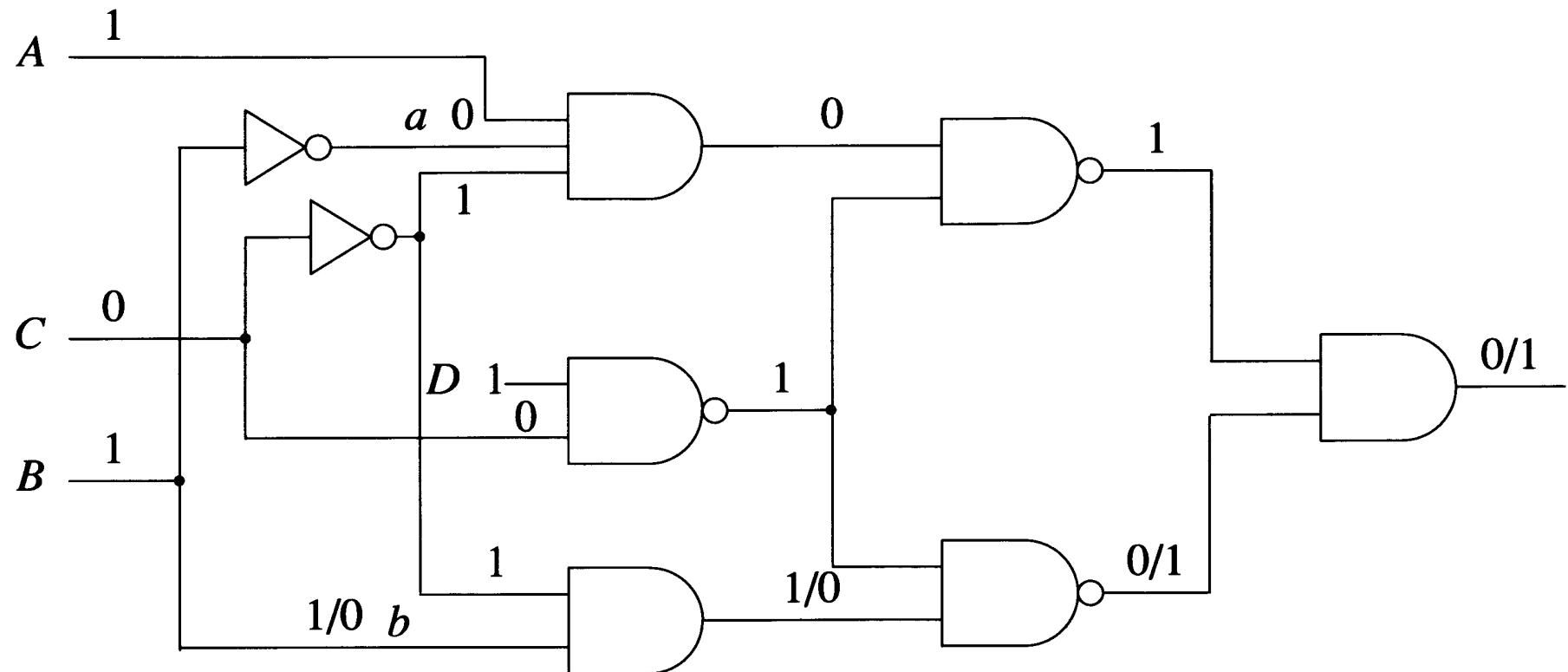
Figure 4.4 NAND gate satisfying Lemma 4.1 ($c=0$, $i=1$)

Faults – Detectability

A fault f is said to be **detectable** if there exists a test t that detects f
Otherwise, f is **undetectable**.

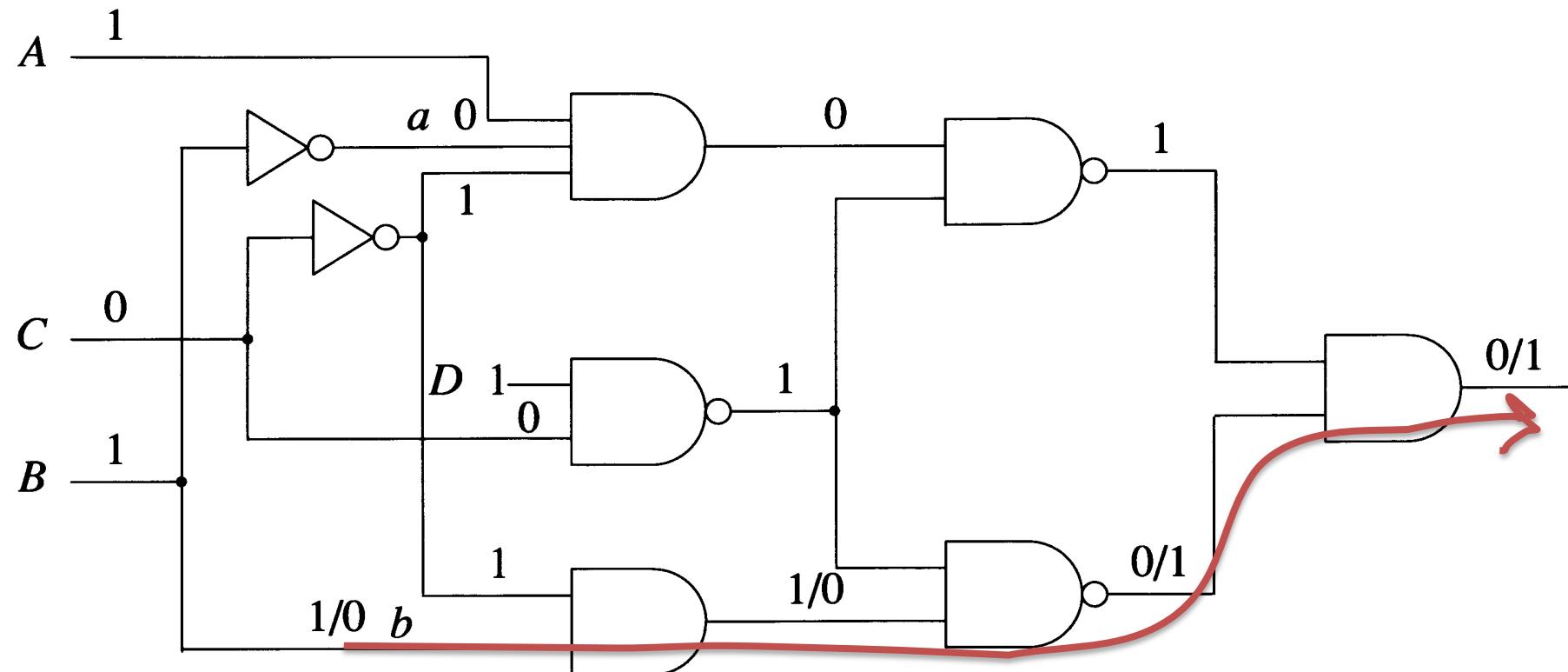
For undetectable fault, no test exists that can simultaneously activate and propagate the fault to the primary output.

Example: Undetectable Fault $a \text{ s-a-1}$



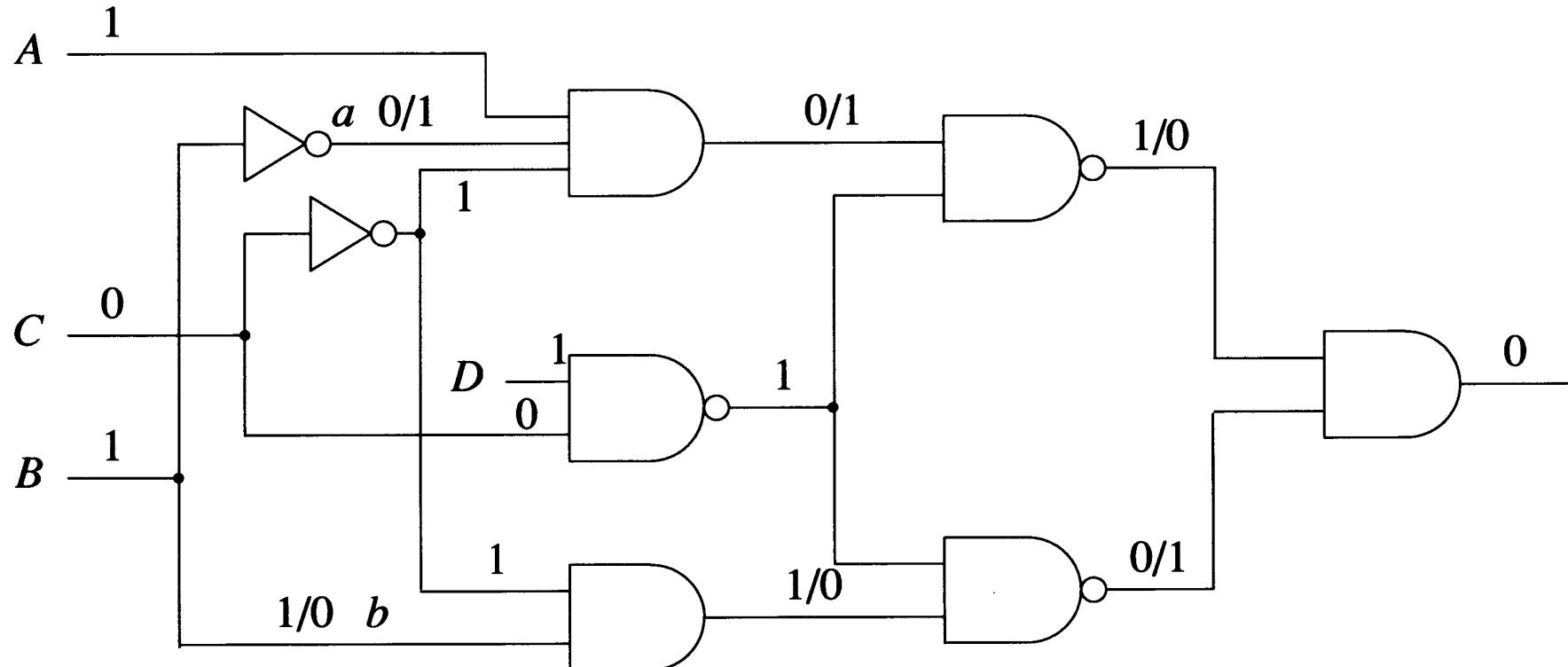
Example: Detectable Fault b $s-a-0$

b $s-a-0$ is detectable with $t = 1101$



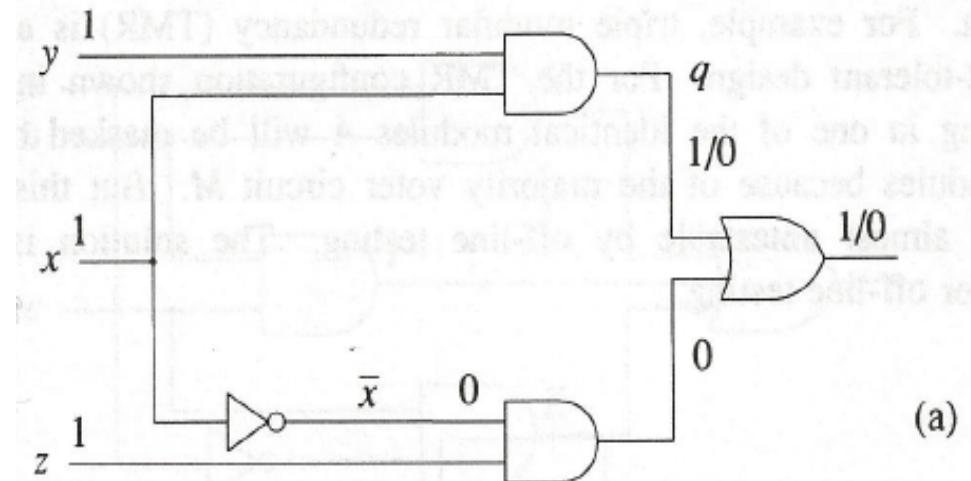
Example: Undetectable Fault

b s-a-0 becomes *undetectable* in the presence of
 a s-a-1 by test $t=1101$

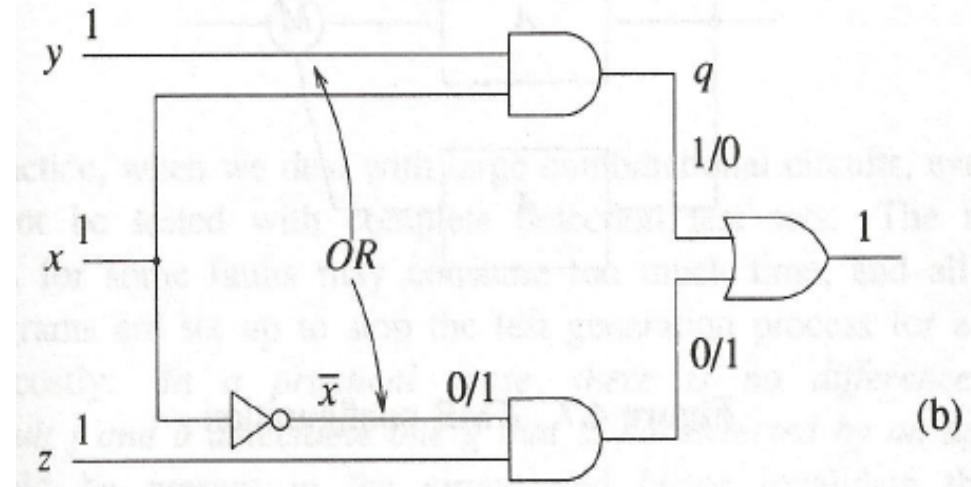


Example 2: Undetectable Fault

q s-a-0 is detectable with
 $t = 111$



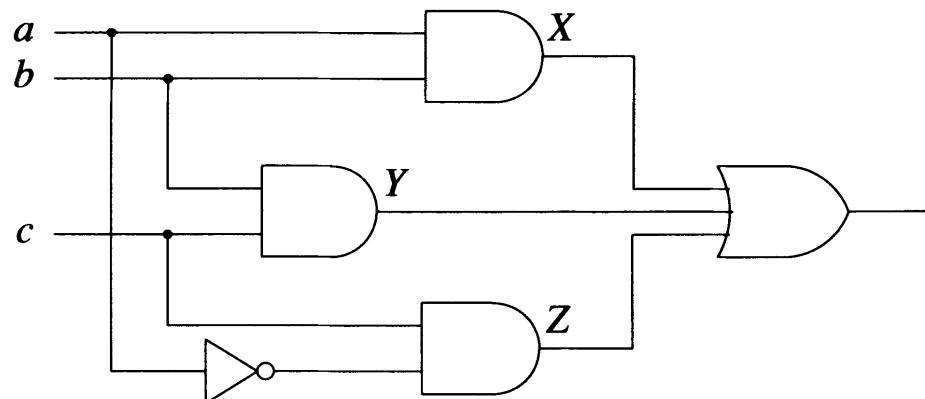
q s-a-0 is undetectable in the presence of OR Bridging fault between y and \bar{x} .



Fault – Redundancy

- A combinational circuit that contains an undetectable stuck fault is said to be **redundant**
- Such circuit can always be reduced by eliminating a gate or a gate input
- A combinational circuit in which all stuck faults are detectable is said to **irredundant**

Example: Y s-a-0 is undetectable. Gate Y can be dropped.



Fault Interaction

- If f is a detectable fault and g is an undetectable fault, then f may become undetectable in the presence of g . Such a fault f is called a *second-generation redundant fault*.
- Two undetectable single faults f and g may become detectable if simultaneously present in the circuit. In other words, the multiple fault $\{f, g\}$ may be detectable even if its single-fault components are not.

Detecting Redundancy

- To show a line is redundant => to prove that no test exists for the corresponding fault
- Detecting Redundancy Problem => Test Generation Problem
- Test generation problem is an *NP*-complete problem
- Practical test generation algorithms run in polynomial time
- Redundant faults make test generation algorithms exhibit worst-case behavior

Large Combinational Circuits

- Even if the circuit is *irredundant*, we may not have complete test set due to time limitations
- In such a case, the fault (say f) for which no test exists, is no different from an *undetectable fault* (say g)
- Undetectable fault g may be present in the circuit and invalidate the single fault assumption.

Sequential Circuits

- Testing more difficult than combinational circuits
- Need a test sequence
- Response is a function of initial state
 - Let T be a test sequence – a sequence of test vectors.
 - $R(q, T)$ be the response to T with initial state q
 - $R_f(q_f, T)$ be the response for faulty circuit

T Strongly Detects f

→ Definition 4.2: A test sequence T **strongly detects** the fault f

if and only if

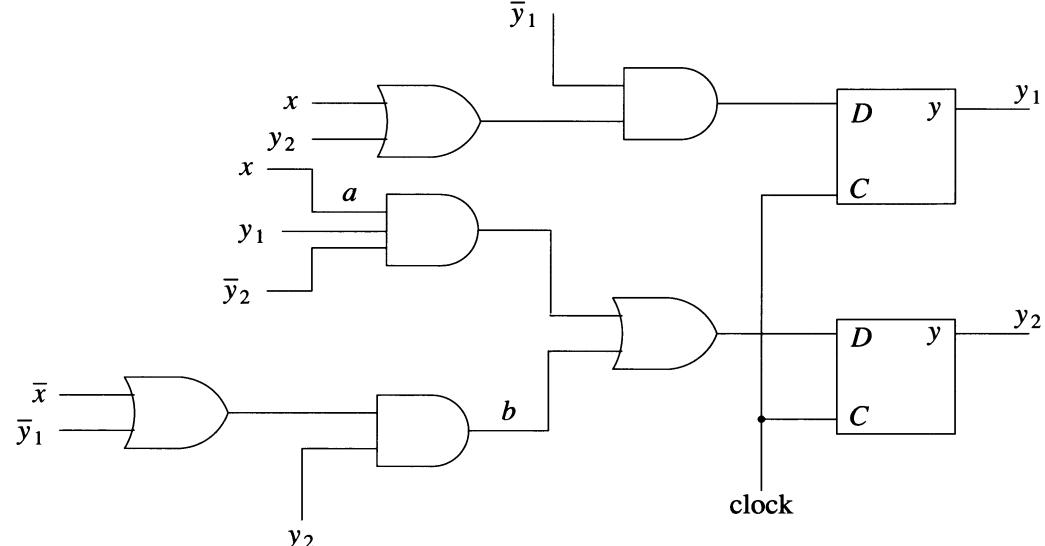
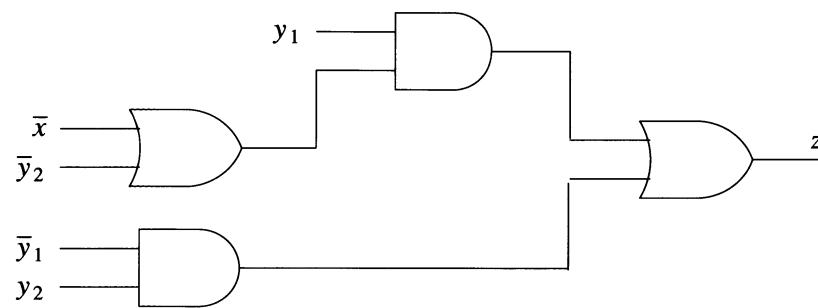
the output sequences $R(q, T) \neq R_f(q_f, T)$ for every possible pair of initial states q and q_f

Example

α line a $s-a-1$

β line b $s-a-0$

Test sequence T=10111



	x		y_1	y_2
	0	1		
A	A,0	D,0	0	0
	C,1	C,1	0	1
B	B,1	A,0	1	1
	A,1	B,1	1	0

Initial state	Output sequence		
	Fault-free	α (a $s-a-1$)	β (b $s-a-0$)
A	01011	01010	01101
B	11100	11100	11101
C	00011	00010	01010
D	11001	10010	11010

Figure 4.9 Output sequences as a function of initial state and fault

Example

- T does not strongly detect α
- T strongly detects β

	0	1	y_1	y_2	
A	$A,0$	$D,0$	0	0	
B	$C,1$	$C,1$	0	1	$: 10111$
C	$B,1$	$A,0$	1	1	
D	$A,1$	$B,1$	1	0	

Initial state	Output sequence		
	Fault-free	$\alpha (a \ s-a-1)$	$\beta (b \ s-a-0)$
A	01011	01010	01101
B	11100	11100	11101
C	00011	00010	01010
D	11001	10010	11010

T Detects f

→ Definition 4.3: A test sequence T **detects** the fault f

if and only if

for every possible pair of initial states q and q_f the output sequences $R(q, T) \neq R_f(q_f, T)$ for *some* specified vector $t_i \in T$

Testing with Initialization

- Phase I: Initialization sequence T , such that N and N_f are brought to known states q , and q_{If}
 - Output responses ignored during initialization
- Phase II: Apply T' (output responses are predictable)
 - t_i is first vector of T' for which an error is observed

Drawback

- Initialization may not be possible for faulty circuit
- Example:

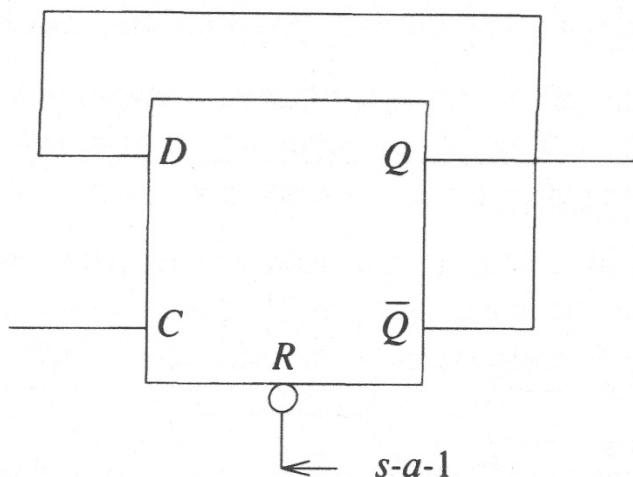


Figure 4.10 Example of a fault preventing initialization

4.3 Fault Equivalence and Fault Location

Fault Equivalence – Combinational Circuits

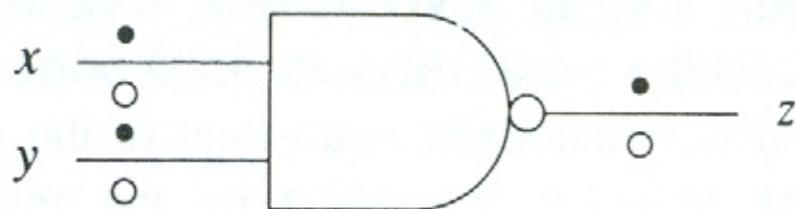
→ **Definition 4.4:** Two faults f and g are said to be **functionally equivalent** iff

$$Z_f(x) = Z_g(x)$$

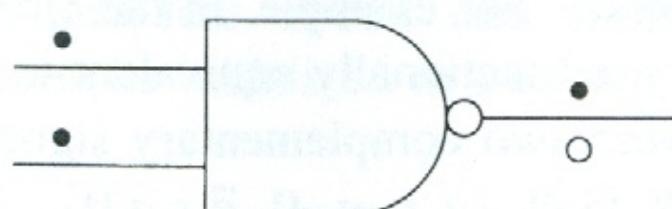
- A test t is said to **distinguish** between two faults f and g if $Z_f(t) \neq Z_g(t)$
- There exists no test that can distinguish functionally equivalent faults
- Faults are divided into equivalent classes.

Equivalence Fault Collapsing

- Reduce faults into equivalent classes
- For a NAND gate,
 - All input $s-a-0$ faults and the output $s-a-1$ are functionally equivalent



(a)



(b)

● : $s - a - 1$, ○ : $s - a - 0$

Equivalence Fault Collapsing

- In general, for a gate with controlling value c and inversion i ,
 - All input s-a-c faults are *functionally equivalent* to output s-a-($c \oplus i$) faults
 - Reduce $2(n+1)$ faults to $n+2$ faults for a n -input NAND gate.

Fault Location

- Goal of testing is to locate the fault besides detecting the fault
- A *complete location test* distinguishes between every pair of distinguishable faults in a circuit
- A fault-free circuit contains *empty fault*, denoted by Φ .
 - Therefore $Z_\Phi(x) = Z(x)$
- A fault detection is a particular case of fault location, since a test that detects f distinguishes between f and Φ .

Functional Equivalence Under a Test

- In practice, test sets are not complete
 - Affect diagnostic result
- **Definition 4.5:** Two f and g are **functionally equivalent under a test set T ,**

iff

$$Z_f(t) = Z_g(t) \text{ for every test } t \in T$$

Note:

Functional equivalence implies functional equivalence under any test set, but not vice versa.

Fault Equivalence – Sequential Circuits

→ **Definition 4.6:** Two f and g are **strongly functionally equivalent** iff their corresponding state tables are equivalent.

→ Impractical

→ **Definition 4.6:** Two f and g are **functionally equivalent** iff for any T' ,

$$R_f(q_{If}, T') = Z_g(q_{Ig}, T')$$

4.4 Fault Dominance

Fault Dominance – Combinational Circuits

- Another fault relation to reduce faults to be considered
- **Definition 4.8:** Let T_g be the set of all tests that detect a fault g . We say that a fault f **dominates** the fault g
 - iff
 - f and g are functionally equivalent under T_g .

Fault Dominance

- If f dominates g , then any test t that detects g , i.e., $Z_g(t) \neq Z(t)$, will also detect f (on the same primary inputs) because $Z_f(t) = Z_g(t)$
- Therefore, we can drop f from the fault set

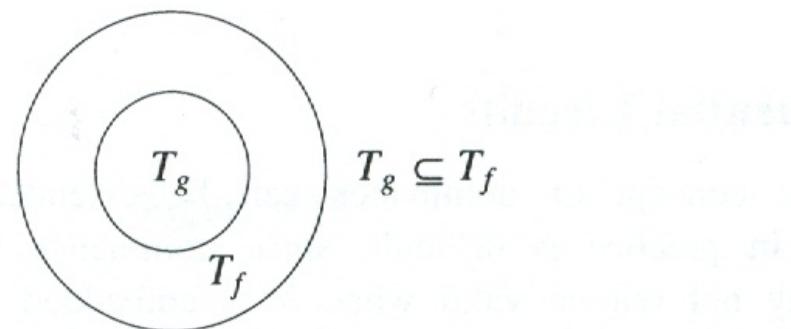
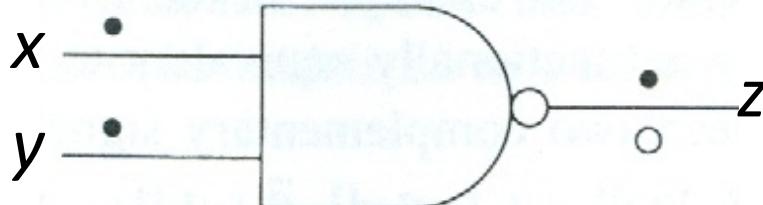
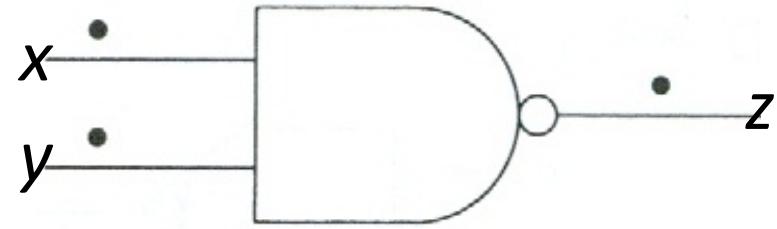


Figure 4.13 The sets T_f and T_g when f dominates g

Example



(b)



(c)

$$\bullet : s - a - 1, \quad \circ : s - a - 0$$

Fault g : y s-a-1

$$T_g = \{xy = 10\}$$

Fault f : z s-a-0

$$T_f = \{\text{??}\}$$

f dominates g

Dominant Fault Collapsing

- In general, for a gate with controlling value c and inversion i ,
 - Output $s-a-(c \oplus i)$ fault dominates input $s-a-\bar{c}$ faults.
- Better to choose a fault model dominated by other models
 - Tests detecting one model also detects other faults models

An Example

- Faults: $f: z_2 \text{ s-a-0}$, $g: y_1 \text{ s-a-1}$, test $t = 10$
 - t detects both f and g , but they do not dominate each other

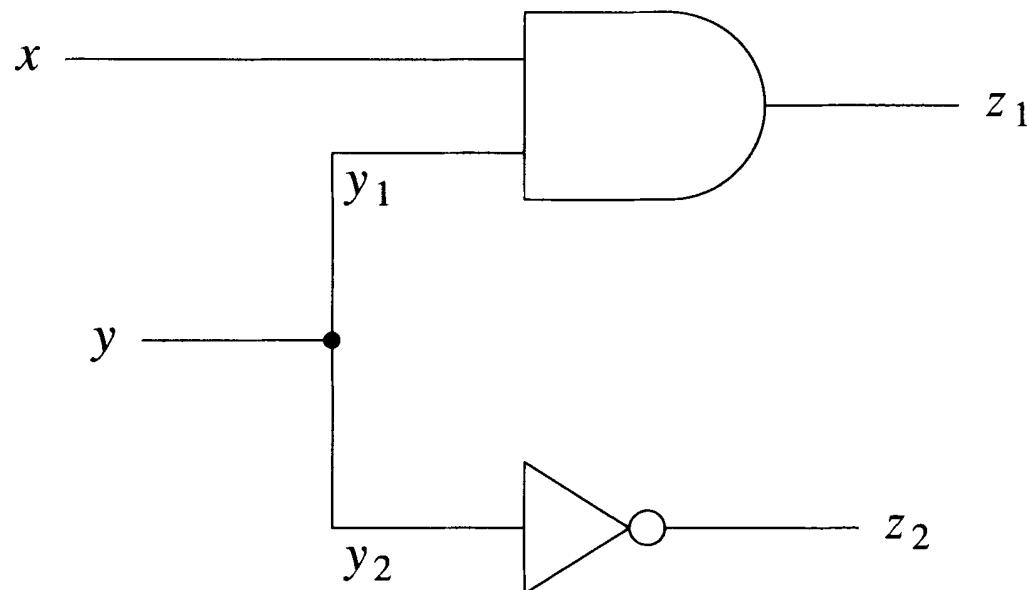


Figure 4.14

4.5 The Single Stuck-Fault Model

Fault Universe

- For n lines where SSFs can be defined, there are $2n$ stuck faults.
- For fanout-free circuits, the number of faults is $2(G+I)$.
 - G : gate count
 - I : number of primary inputs
- For circuits with fanout, the estimate is $2Gf$
 - f : average fanout count

Checking Functional Equivalence is Hard!

- In general, to determine whether two arbitrary faults are functionally equivalent is NP-complete
- Example: Are $c\ s-a-1$ and $d\ s-a-1$ functionally equivalent?

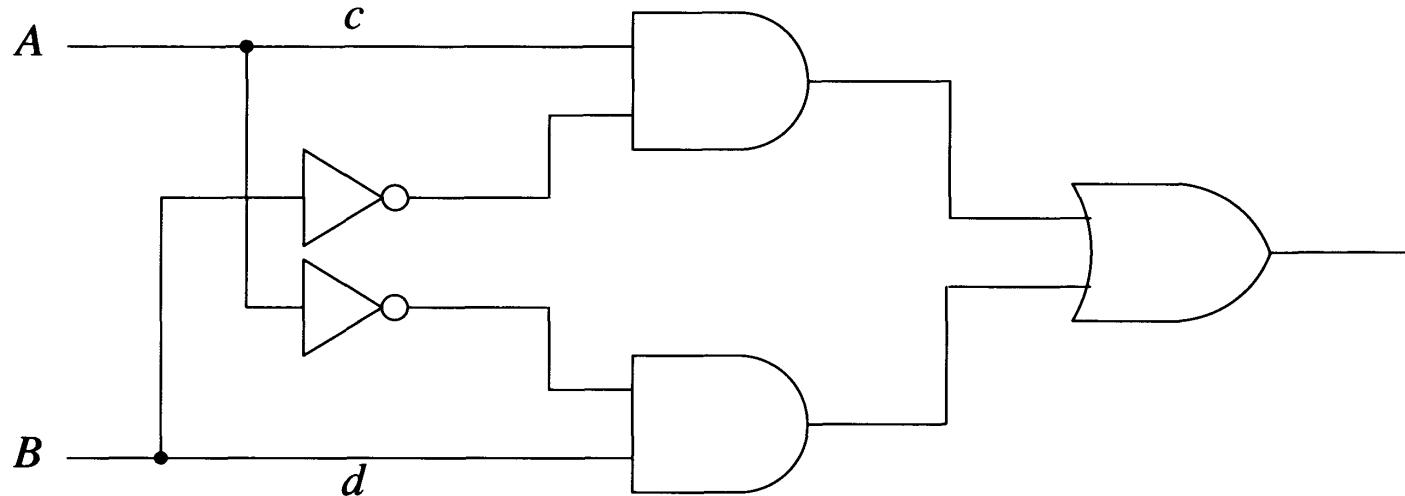


Figure 4.17

Structural Equivalence

- In a circuit N_f the fault f creates a set of lines with constant values.
- By removing all these lines (except POs), we obtain $S(N_f)$
- Two faults f and g are structurally equivalent if $S(N_f)$ and $S(N_g)$ are identical
- All structurally equivalent faults are functionally equivalent
- But not all functionally equivalent faults are structurally equivalent

Example: Structural Equivalence

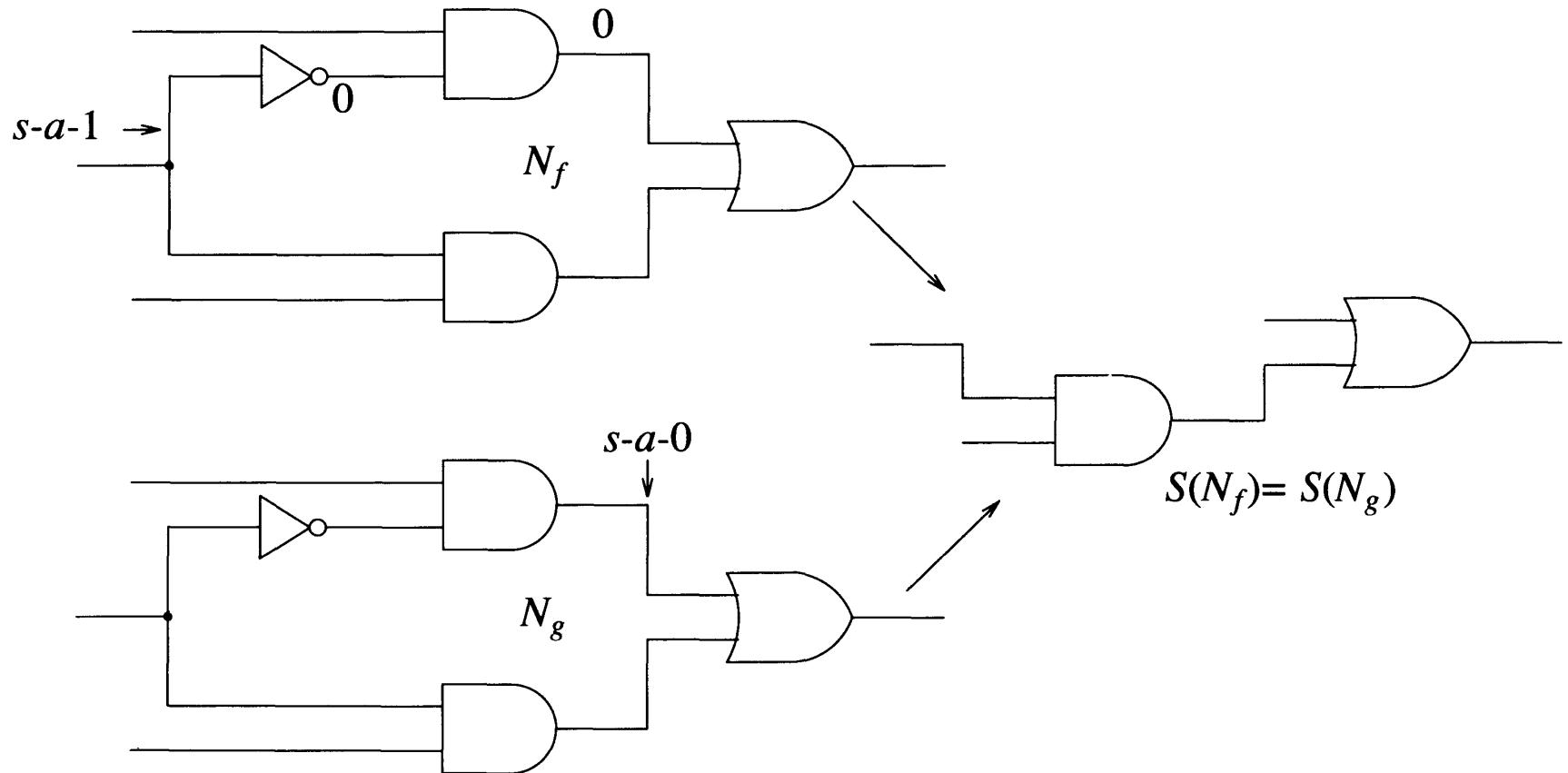


Figure 4.18 Illustration for structural fault equivalence

Structurally Equivalent Fault Collapsing

- For a line with fanout of 1, the faults at its sources are structurally eq. to those at its destination.
- For a gate, input $s-a-c$ is structurally eq. to output $s-a-(c \oplus i)$
- For each eq. class, retain only one fault.

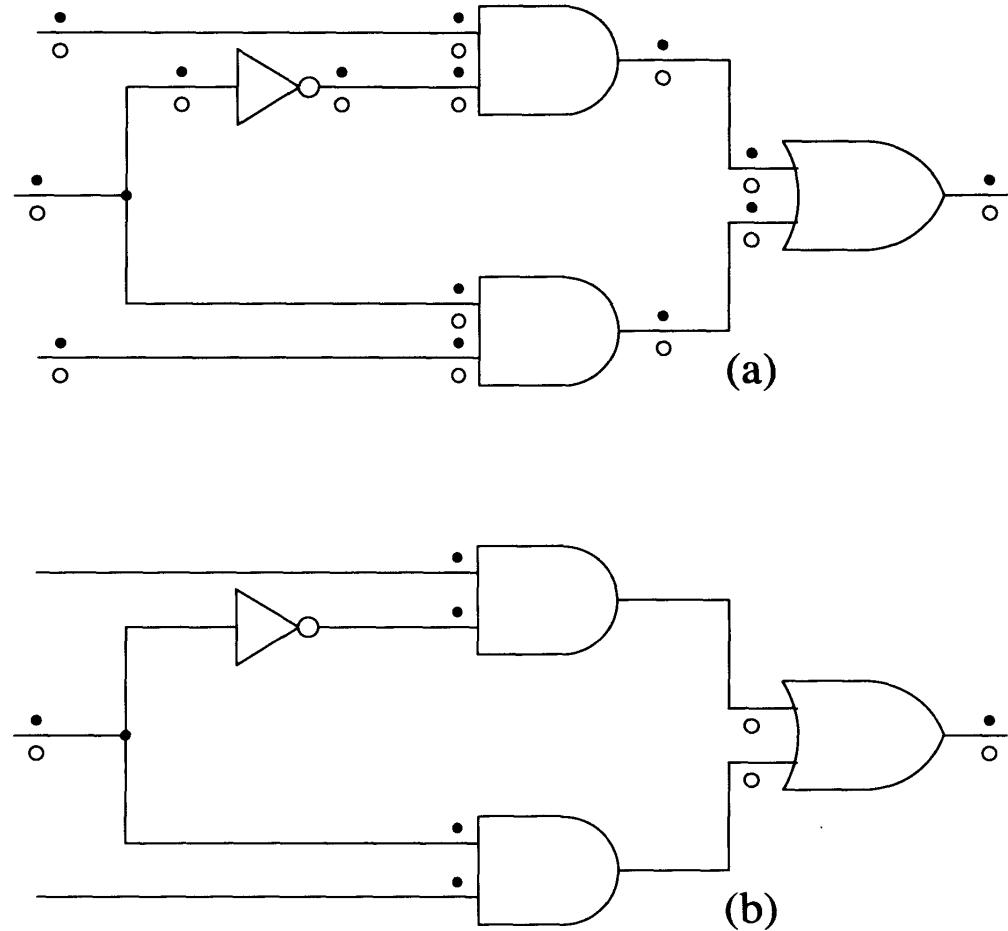
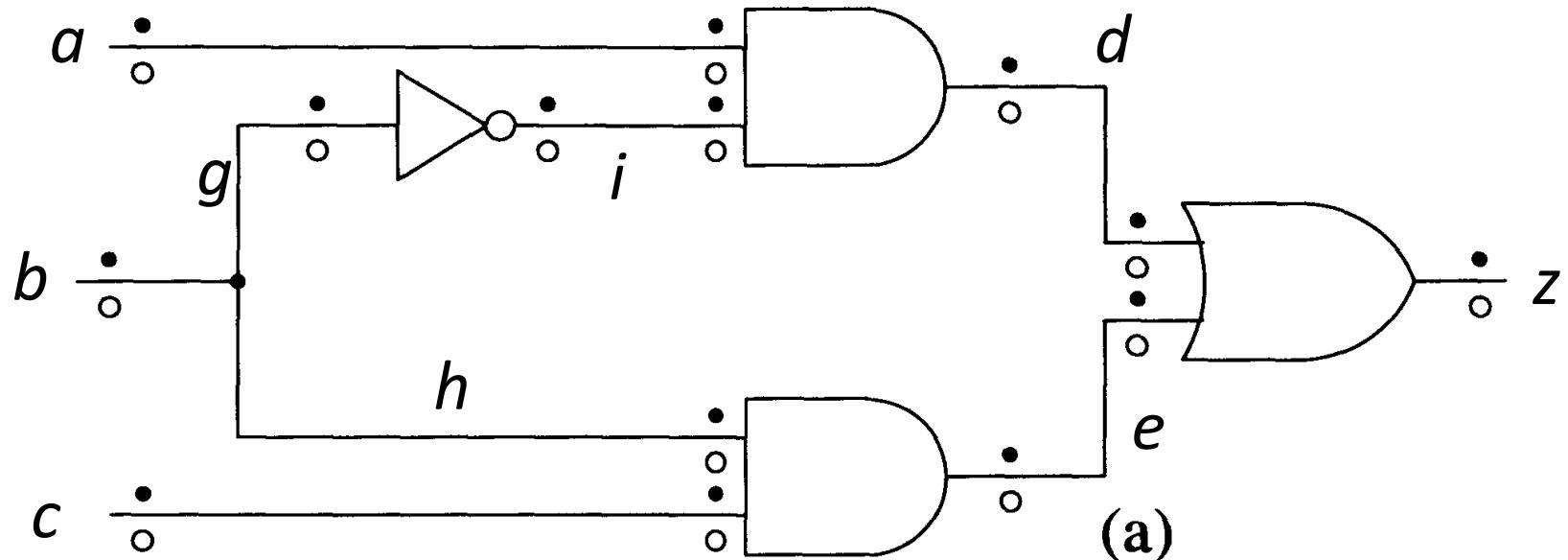


Figure 4.19 Example of structural equivalence fault collapsing

Structurally Equivalent Faults – Example



g s-a-1 and i s-a-0 ?

On average, about 50% SSFs can be reduced !

Fault Reduction by Dominance Relation

- **Theorem 4.1:** In a **fanout-free** combinational circuit C , any test set that detects all SSFs on the primary inputs of C detects all SSFs in C .
- **Theorem 4.2:** In a combinational circuit C any test set that detects all SSFs on the primary inputs and the fanout branches of C detects all SSFs in C .
 - These faults can be further reduced by structural equivalence and dominance relations.

Example

- How many faults after fault collapsing?
- Answer: 10 (verify this)

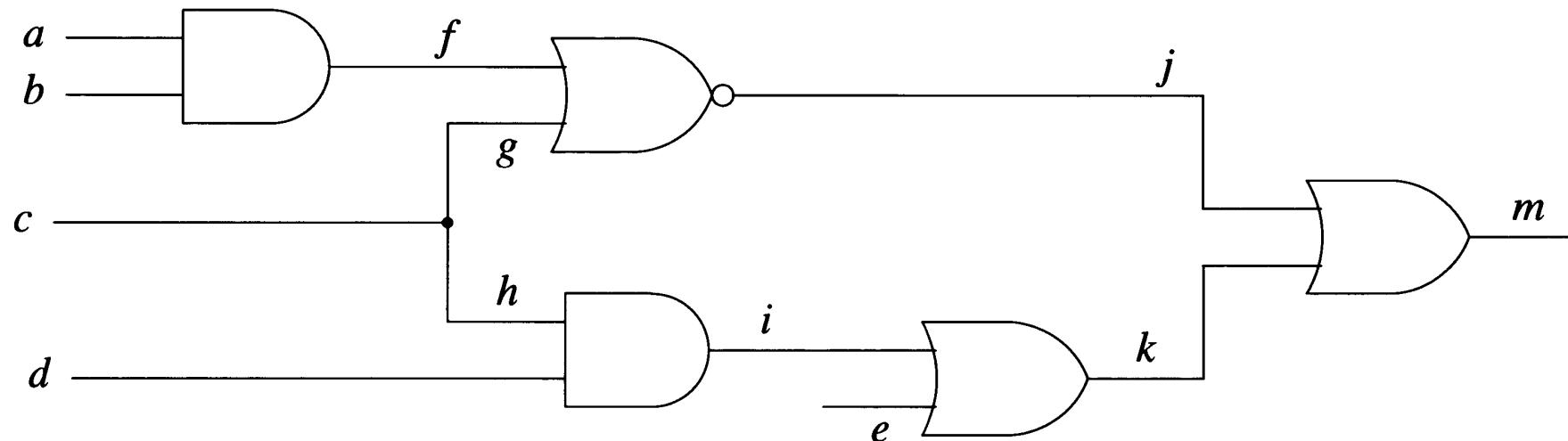


Figure 4.21

Stem and Fanout Faults – Example 1

- In general neither functional equivalence nor dominance relations exists between stem and fanout faults
- Stem fault j s-a-0 is detected but k s-a-0 and m s-a-0 are not

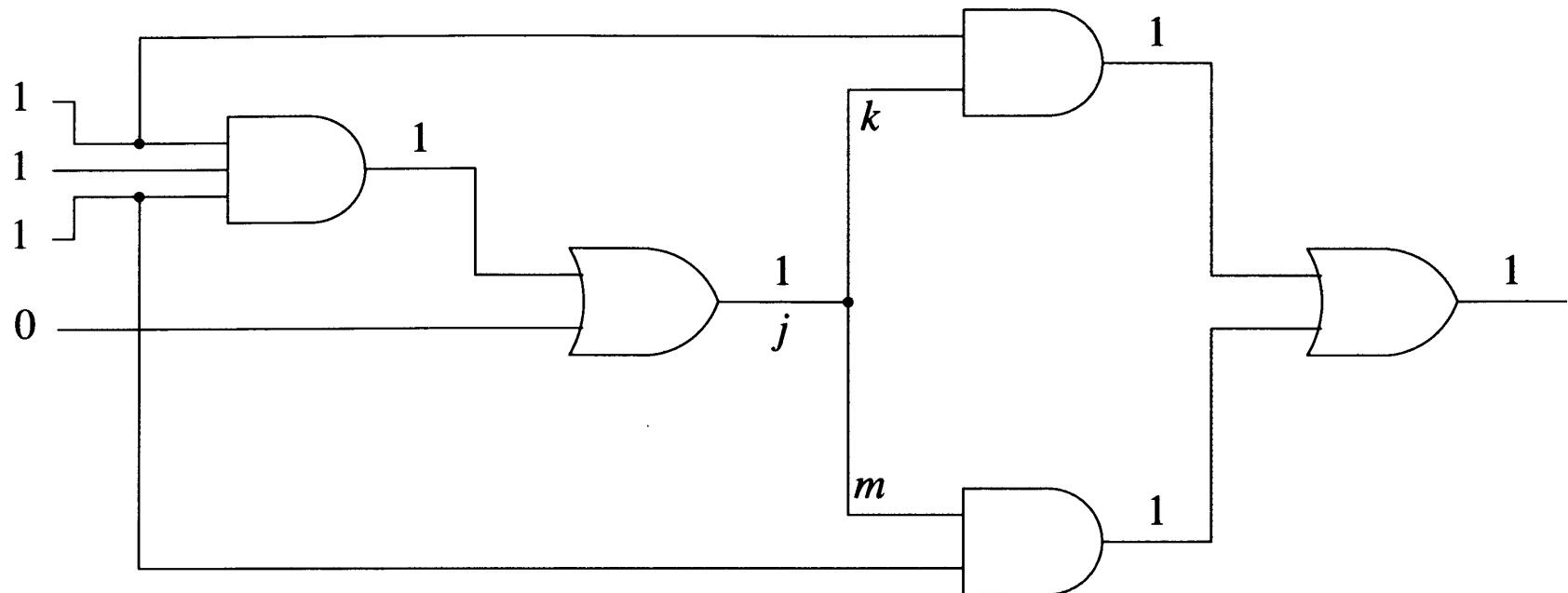


Figure 4.22

Stem and Fanout Faults – Example 2

→ Fanout faults x_1 s-a-0 and x_2 s-a-0 are detectable but stem fault x s-a-0 is not

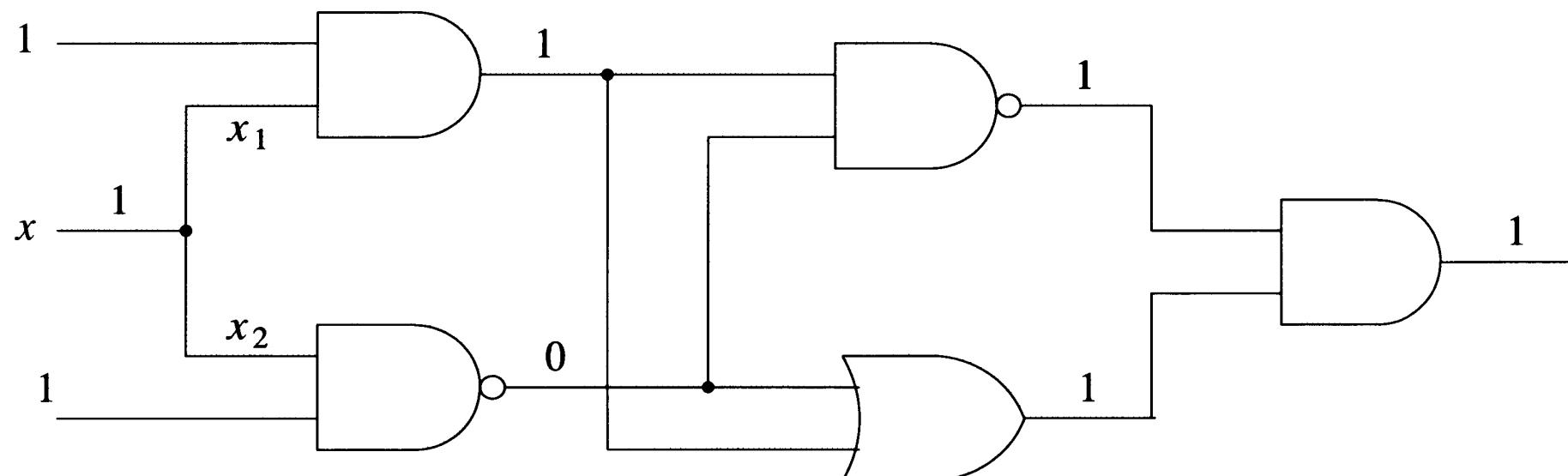


Figure 4.23

4.6 The Multiple Stuck-Fault Model

Multiple Stuck Fault (MSF) Model

- Several lines can be stuck simultaneously.
 - Straightforward extension of SSF
- For a n line circuit, there are
 - $2n$ SSFs, but
 - $3^n - 1$ possible MSFs
- $\sum_{i=1}^k \binom{n}{i} 2^i$ possible MSFs for multiplicity k .
 - Too large a number to handle.
- A MSF F can be viewed as $\{f_1, f_2, \dots, f_k\}$.
- Why do we need to consider MSF?
 - Because of masking relation between faults

Functional Masking

- **Definition 4.9:** Let T_g be the set of *all* tests that detect g . We say that f **functionally masks** g iff the multiple fault $\{f, g\}$ is not detected by any test in T_g
- Example: $a \text{ s-a-1}$ masks $c \text{ s-a-0}$
 - 011 only vector that detects $c \text{ s-a-0}$
 - $\{c \text{ s-a-0}, a \text{ s-a-1}\}$ is not detectable by 011

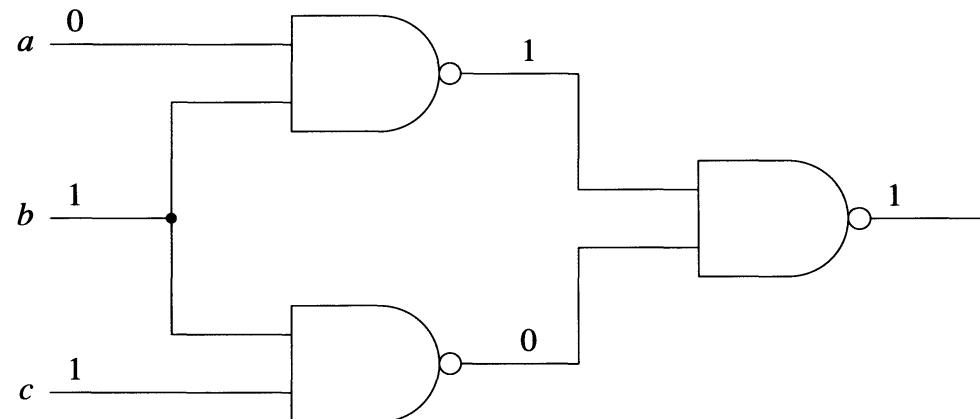


Figure 4.24

Masking under a Test Set

→ **Definition 4.10:** Let $T'_g \subseteq T$ be the set of all tests in T that detect a fault g . We say that a fault f **masks** the fault g **under a test set T** iff the multiple fault $\{f, g\}$ is not detected by any test in T'_g

→ Functional masking → masking under a test set



$$T'_g \subseteq T_g$$

Problem

→ If f masks g , then fault $\{f, g\}$ is not detected by tests that detect g alone. But can there be other tests to detect $\{f, g\}$?

$$\{c \text{ } s-a-0, \text{ } a \text{ } s-a-1\}$$

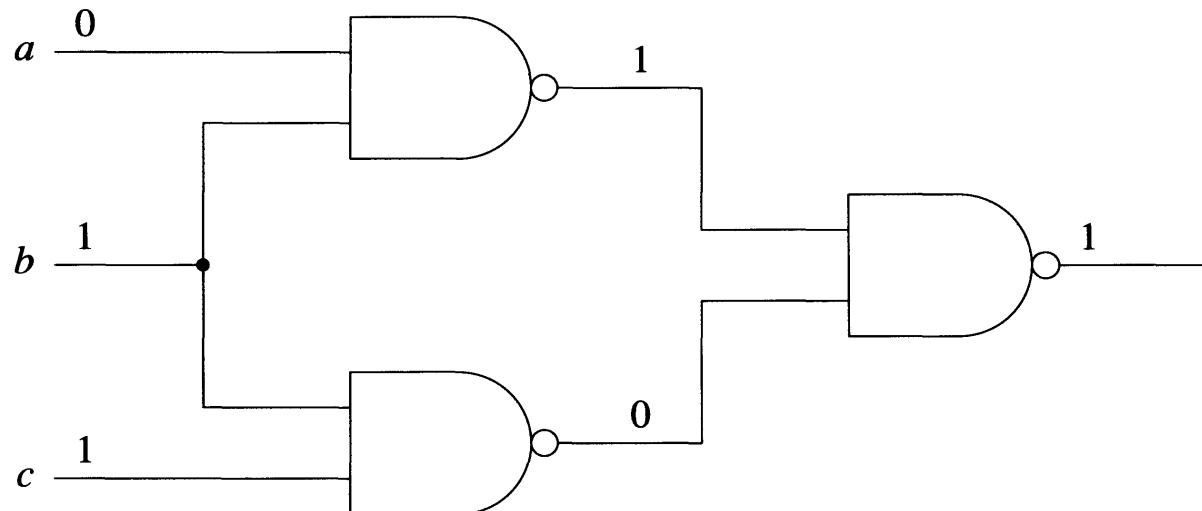


Figure 4.24

Problem – Detection of Multiple Faults

→ Given a complete test set T that detects all single faults, can there exist a multiple fault $F = \{f_1, f_2, \dots, f_k\}$ such that F is not detected by T ?

Example

- $T = \{1111, 0111, 1110, 1001, 1010, 0101\}$
- $f = B \text{ s-a-1}$ and $g = C \text{ s-a-1}$
- 1001 is the only vector that detects f and g SSFs, but...

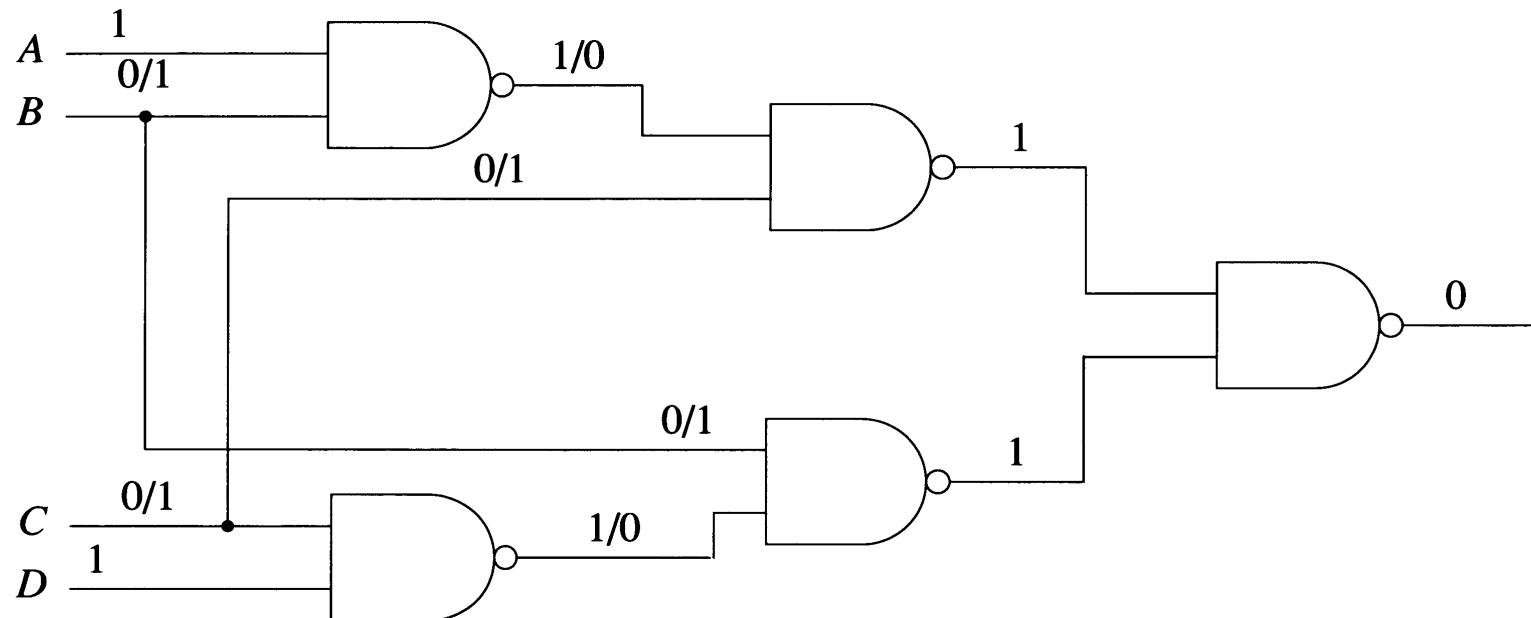


Figure 4.25

Example

- 1001 cannot detect $\{f, g\}$
 - f masks g , and g masks f – **circular masking**

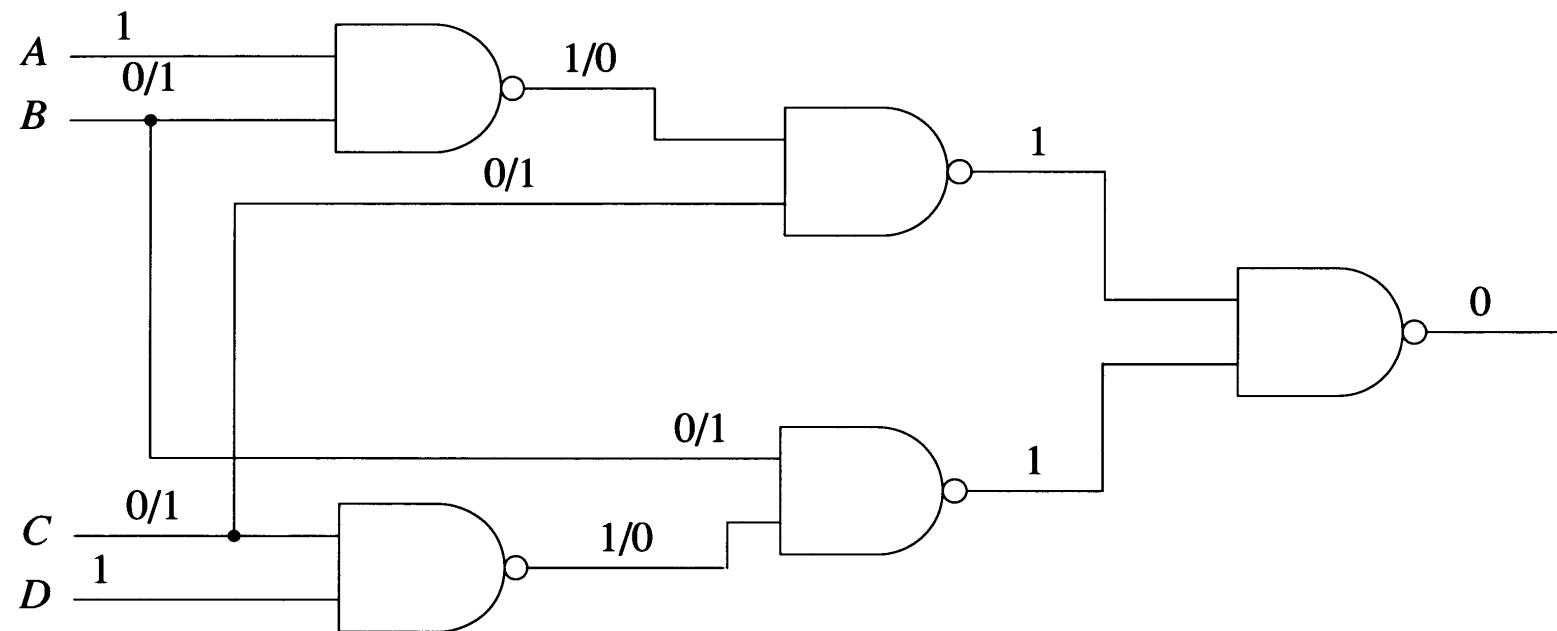


Figure 4.25

What % of MSFs can escape detection by a complete SSF test set (T)?

- In an irredundant 2-level circuit, T also detects all MSFs
- In a fanout-free circuit, any T detects all double and triple faults, and there exists a T for SSFs that detects all MSFs
- In an internal fanout free circuit, any T detects at least 98% of MSFs with multiplicity $k < 6$
- A test set that detects all MSFs defined on all primary inputs without fanout and all fanout branches of a circuit C detects all multiple faults in C .

MSFs or Not?

- A test set T for SSFs also detects most MSFs.
 - Typically focus on MSFs not detectable by T .
- Probability of MSFs is low.
 - Modern semi-manufacturing is highly reliable
- A SSF is **guaranteed to be detected (GTBD)** if it can be detected unconditionally.
 - Any MSF that includes a GTBD SSF is also GTBD.

Summary

- Logic faults can represent various physical faults
- Fault detection
 - Find a test that causes deviation in output responses
 - Undetectable faults → redundancy
- Fault collapsing
 - Fault equivalence & dominance
- Single stuck-fault model
 - Possible to find a complete test set
- Multiple fault model
 - A SSF detection test set can find many MSFs

Backup

Corollary 4.1

Let j be a line sensitized to the fault $/s-a-v$ (by a test t), and let p be the inversion parity of a sensitized path between $/$ and j .

1. The value of j in t is $v \oplus p$.
2. If there are several sensitized paths between $/$ and j , then all of them have the same inversion parity.

Redundancy in Circuits

→ Redundancy can be by design i.e., intentional

Fault tolerant design

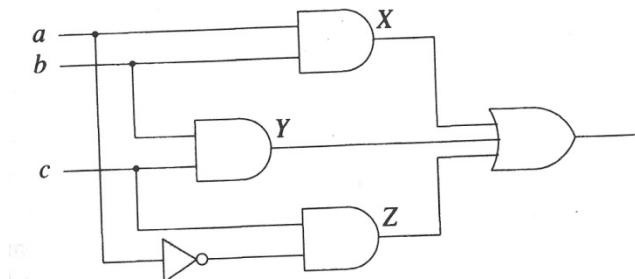


Figure 4.8

Hazard-free design

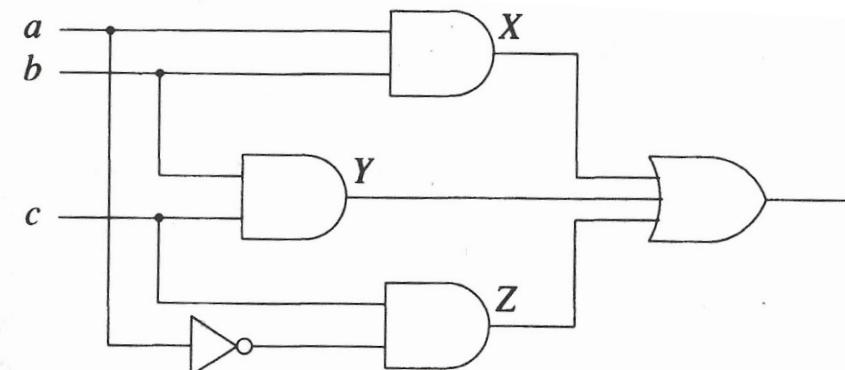


Figure 4.8

CIS 4930 Digital System Testing

Fault Simulation

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

Overview

- Fault simulation applications
- Fault simulation techniques
 - Serial
 - Parallel
 - Deductive
 - Concurrent
- **tentative**
 - Fault simulation for combinational circuits
 - Fault sampling
 - Statistical fault analysis

5.1 Applications

Fault Simulation

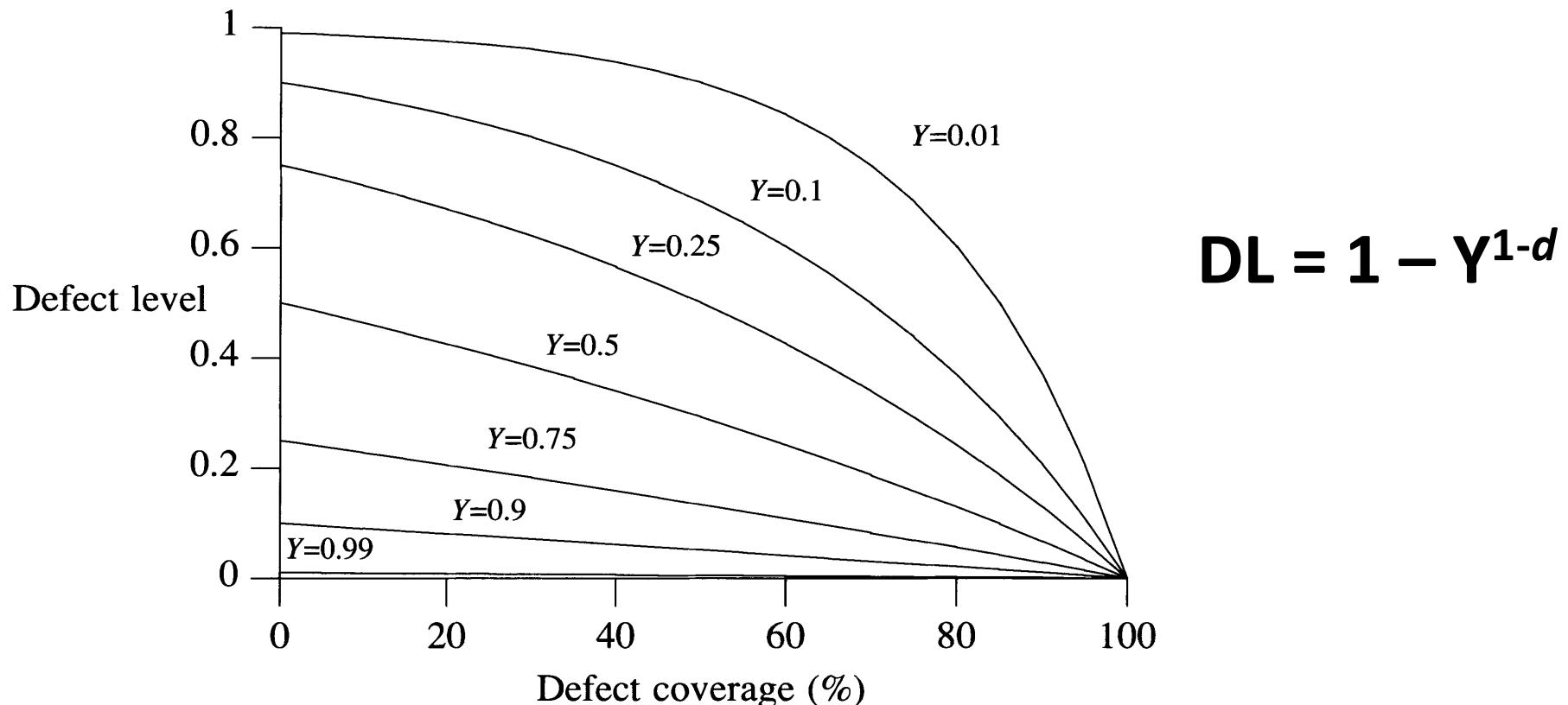
- Simulation of a circuit in the presence of faults
- Used to
 - Evaluate a test T wrt fault coverage.
 - Generate tests T to achieve certain fault coverage.
 - Construct fault dictionary
 - Analyze circuit operation in the presence of faults

1 – Evaluate a Test T

- Usual metric: fault coverage
- Fault coverage relevant to the fault model
 - 100% FC does not mean 100% defects are covered if the fault model is limited.
 - Other defects may still exist if not considered in a fault model.
 - Lower bound on **defect coverage**
- Defect coverage d = probability that T detect any physical fault.
 - Has a big impact on product quality.

Yield and Defect Level

- Defect level (DL) = prob. of shipping a defective product
- Yield (Y) = prob. that manufactured circuit is defect free



2 – Test Evaluation

→ Enhance T until adequate fault coverage is satisfactory

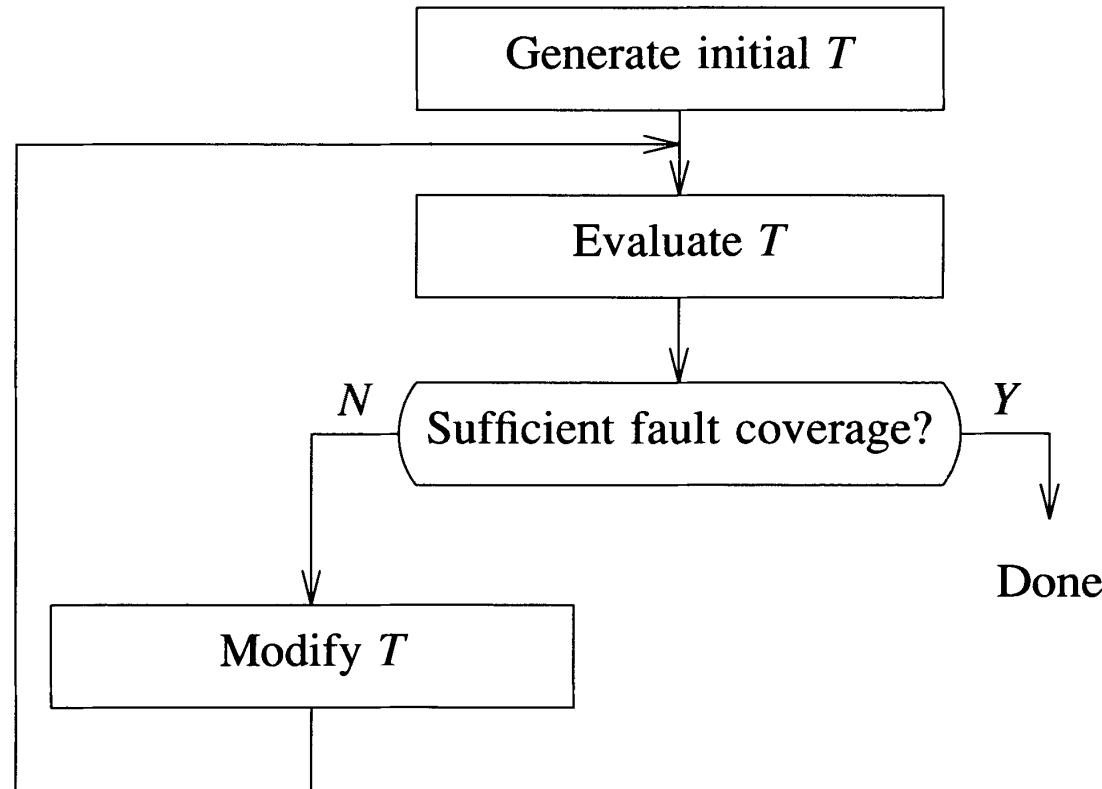
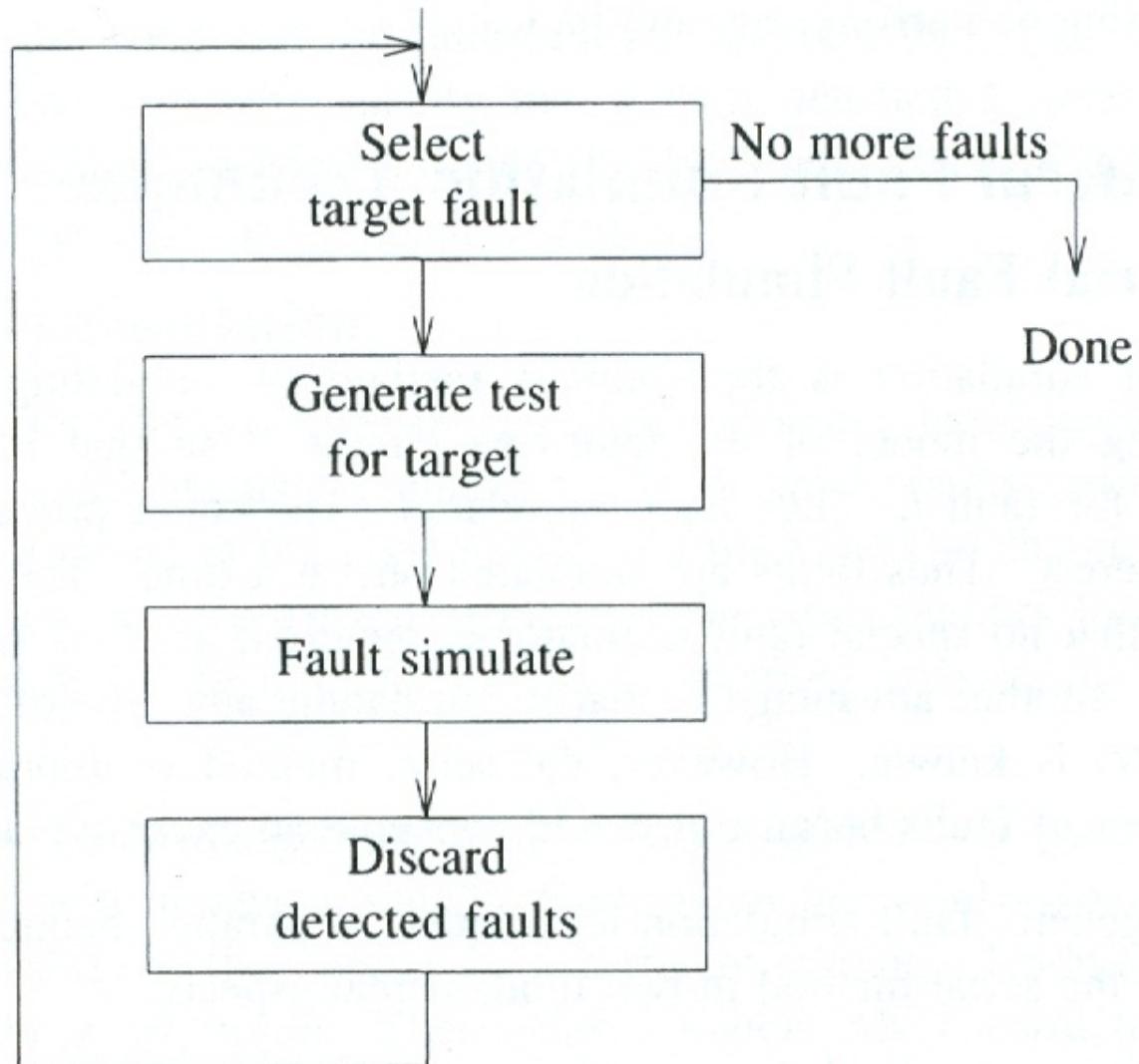


Figure 5.2 General use of fault simulation in test generation

Test Generation

Target fault
oriented
approach



3 – Construct Fault Dictionaries

→ Fault Dictionary – stores output response (R_f) or *signature* $S(R_f)$ to T of every faulty circuit N_f

	f1	f2	..	fn
T1	0	1	..	1
T2	1	0	..	1
:	:	:		:
Tm	1	1	..	0

4 – Circuit Analysis

- Analyze circuit operations in presence of faults
- Some effects introduced by faults may not present in fault-free circuit:
 - Races and/or hazards
 - Oscillation and/or deadlock
 - Inhibit proper initialization of seq. circuit
 - Transform combinational to sequential
 - Transform synchronous to asynchronous

5.2 Fault Simulation Techniques

General Fault Simulation Techniques

- Serial Fault Simulation
- Parallel Fault Simulation
- Deductive Fault Simulation
- Concurrent Fault Simulation

Serial Fault Simulation

- Simulate faults one at a time
- Given a fault f , do the following:
 - Transform N to N_f
 - Simulate N_f
- Repeat for other faults under consideration.
- **Advantage**
 - No need for a special fault simulator
- **Disadvantage**
 - Impractical for large number of faults

Other Three Techniques

- Common characteristics:
 - Do not change the circuit model
 - Can simultaneously simulate a *set of faults*(!)
 - Simultaneously simulate *good* and *bad* circuits
- **One-Pass** – If all faults are simulated simultaneously
- **Multi-Pass** – For large set of faults, need multiple simulation runs

Tasks in Fault Simulation

- **Fault specification:** define set of modeled faults and perform fault collapsing
- **Fault insertion:** select a fault subset and create data structures to indicate fault presence.
- **Fault effect generation:** Say line i has f_{s-a-1} then whenever value 0 propagates on line i , then simulator changes it to 1
- **Fault effect propagation:** Propagate v/v_f to primary output for fault detection
- **Fault discarding:** Inverse of fault insertion
 - Discard a fault if it is detected for k times.

Parallel Fault Simulation

- Simultaneously simulate the good circuit and W copies of faulty circuits
- Set F of faults needs $\lceil F/W \rceil$ number of passes
- Values of the same signal in different circuits are packed into one memory location (a word or multi-words).

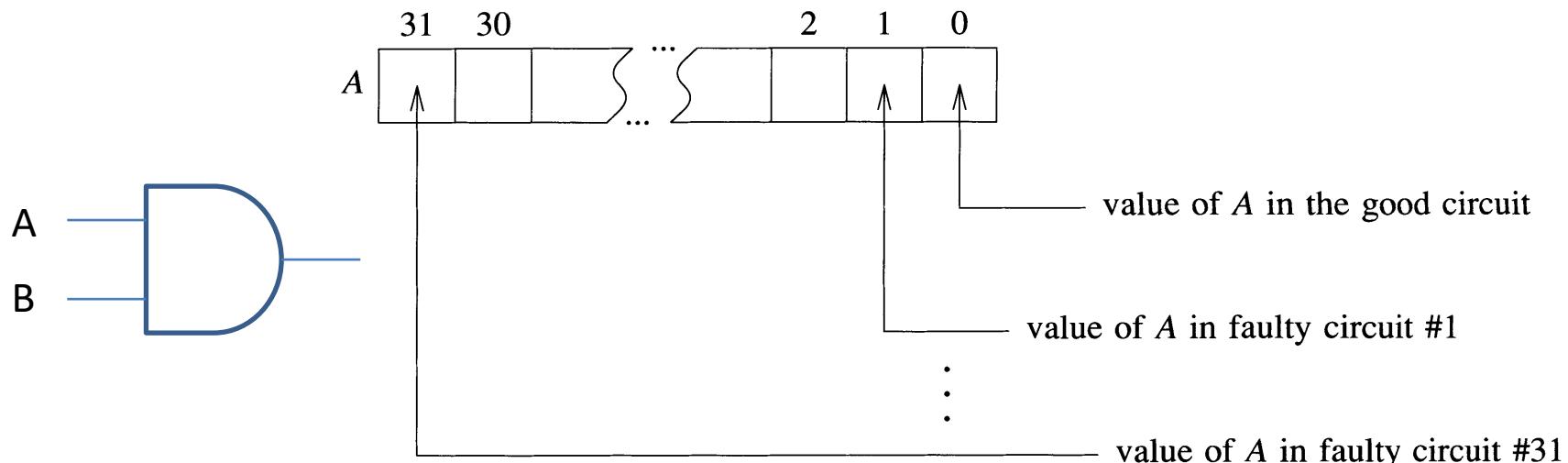


Figure 5.4 Value representation in parallel simulation

Function Evaluation

- Words for A and B are bitwise ANDed (for eg.) for logic AND.
- Similar for other Boolean operations.
- Sequential circuit: For eg., JK FF

$$Q^+ = J\bar{Q} + \bar{K}Q$$

$$Q = c / k \uparrow ? Q^+ : Q$$

The above expression is a Boolean expression consisting of AND, OR, and NOT

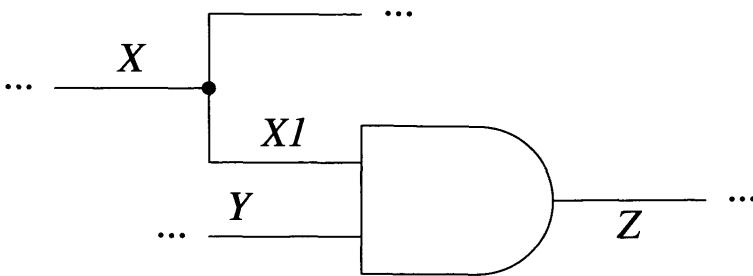
Bit Value Computation

- Let v'_i be the value on line i in the faulty circuit N_f where f is the fault j $s-a-c$
- Then,

$$v'_i = v_i \overline{\delta_{ij}} + c \delta_{ij}$$

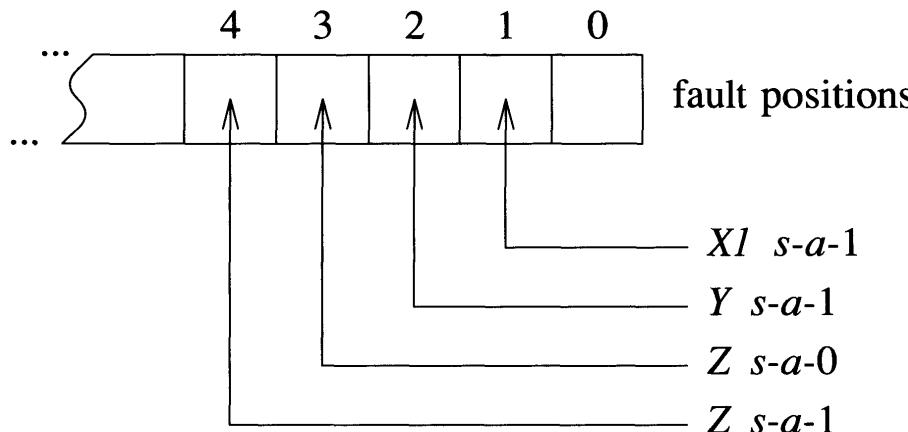
$$\text{where } \delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

Fault insertion for one fault



$$Z = X_1 \cdot Y$$

$$Z' = Z\overline{I_Z} + S_Z I_Z$$



To discard a fault simply make the it's mask bit = 0

I_z	faults to insert on Z
S_z	stuck values for faults defined by I_z
Z	before fault insertion
Z	after fault insertion

The table shows four rows of binary values. The first row is labeled I_z and contains the values 0, 1, 1, 0, 0, 0. The second row is labeled S_z and contains the values 1, 0, (empty), (empty), (empty), (empty). The third row is labeled Z before fault insertion and contains the values 0, 0, 0, 1, 0. The fourth row is labeled Z after fault insertion and contains the values 1, 0, 0, 1, 0. In the 'before' row, the first two columns are highlighted with a red box. In the 'after' row, only the first column is highlighted with a red box.

Mask

Stuck Values

Parallel FS - Limitations

- Parallel simulation is limited for functional level modeling
 - For example if we have to examine for a word value, we need to extract the bits and then re-pack
- Impractical for multi-valued logic
- Event on one bit position results in entire word evaluation => wasted computation
- Cannot take advantage of fault dropping
 - Even if all but one faults are dropped, we still evaluate W copies!

Deductive Fault Simulation

- Simulates good circuit and deduces the behavior of *all* faulty circuits (limited by memory)
- Maintains **Fault List**, L_i for each signal line i .
- L_i = List of all faults f that cause the values on i in N and N_f to be different at the current simulation time
- Difference with Parallel Simulation:

i	F	9	8	7	6	5	4	3	2	1	0
	1		1	1	0	1	1	0	1	1	1

$$L_i = \{4,7\}$$

Figure 5.8 Fault-effects representation in parallel and deductive fault simulation

How Deductive Simulation Works

- Given
 - Fault-free input values, and
 - Fault lists on inputs of an element
- Compute:
 - Fault-free output
 - Output fault list (i.e., fault list propagation)

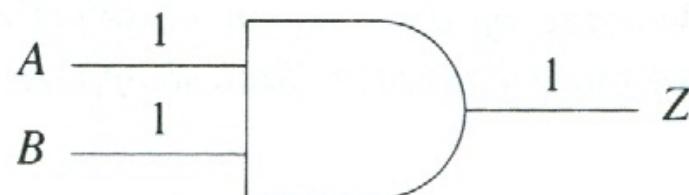
Two Valued Deductive Simulation

- Any fault that causes A or $B = 0$ will lead to $Z = 0$
- Therefore:

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

$$L_A = \{ A \text{ s-a-0} \}$$

$$L_B = \{ B \text{ s-a-0} \}$$

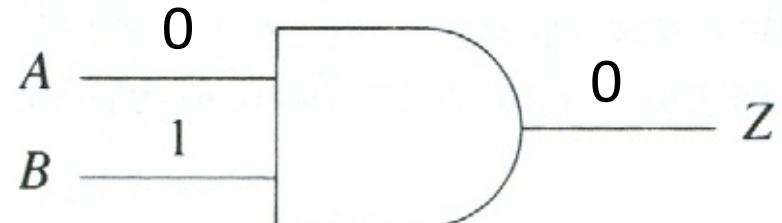


Use Ax to denote A s-a-x

Two Valued Deductive Simulation

- Any fault that causes $A = 1$ without changing B , will cause an error on Z
- Note -- A fault that propagates on both A and B will not affect Z
- Therefore:

$$\begin{aligned}L_Z &= (L_A \cap L_B) \cup \{Z_1\} \\&= (L_A - L_B) \cup \{Z_1\}\end{aligned}$$



General Formulae

→ Let I = set of inputs of gate Z

C = set of inputs with control value c

Then Fault List L_Z on Z is given by

if $C = \emptyset$ then

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

else

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

Example

After Fault Collapsing, the fault set is

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

Assume $T = 00110$ to $abcde$

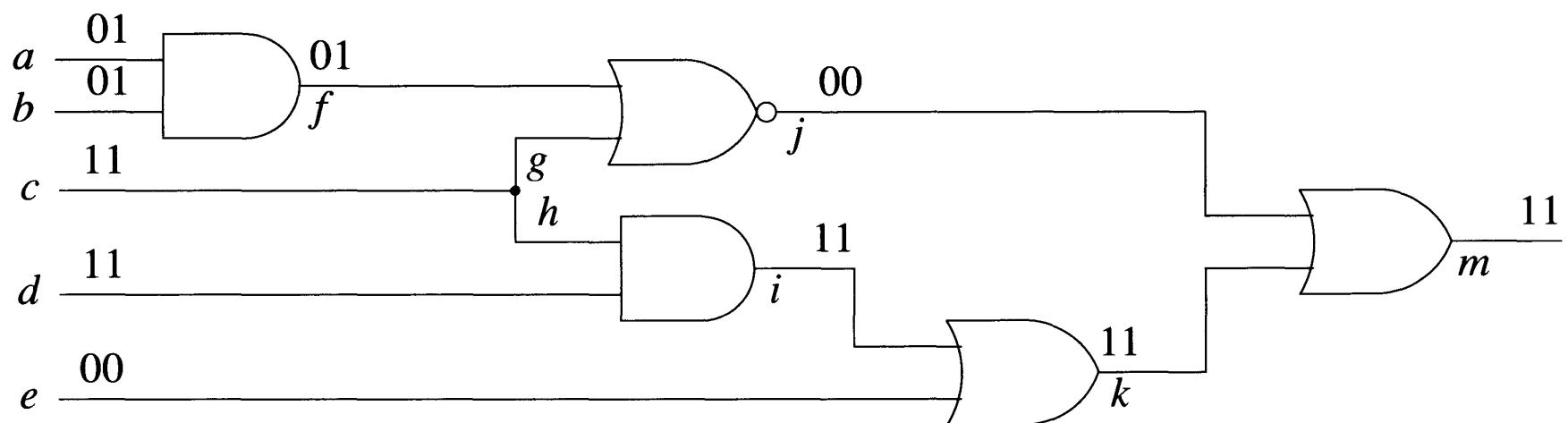


Figure 5.10

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

$$L_a = \{a_1\}, \quad L_b = \{b_1\}, \quad L_c = \{c_0\}, \quad L_d = \emptyset, \quad L_e = \emptyset$$

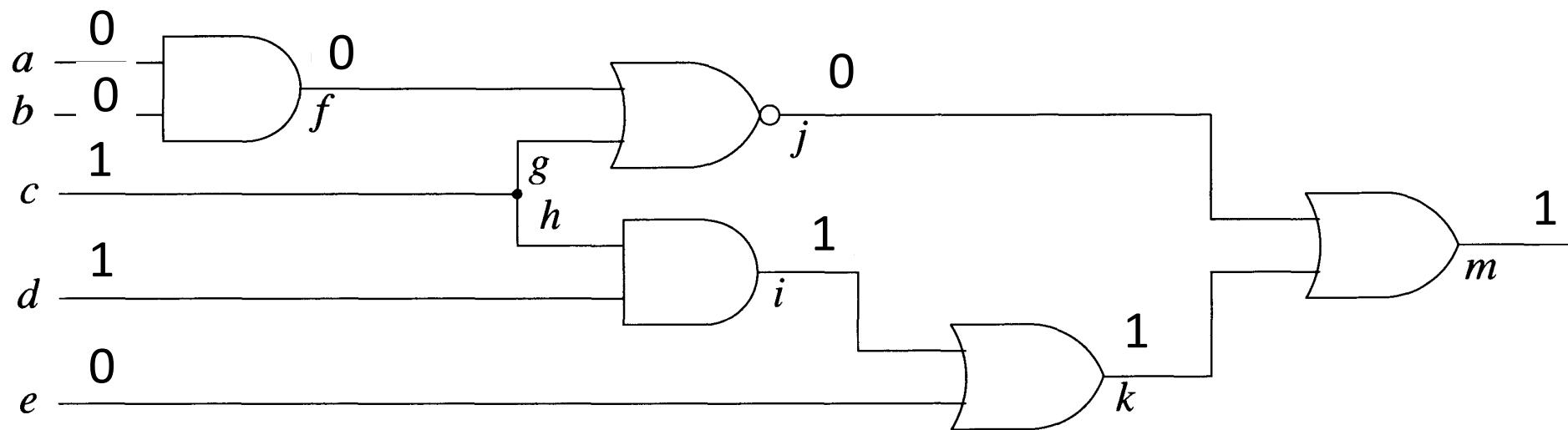


Figure 5.10

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

$$L_a = \{a_1\}, \quad L_b = \{b_1\}, \quad L_c = \{c_0\}, \quad L_d = \emptyset, \quad L_e = \emptyset$$

$$L_f = L_a \cap L_b = \emptyset,$$

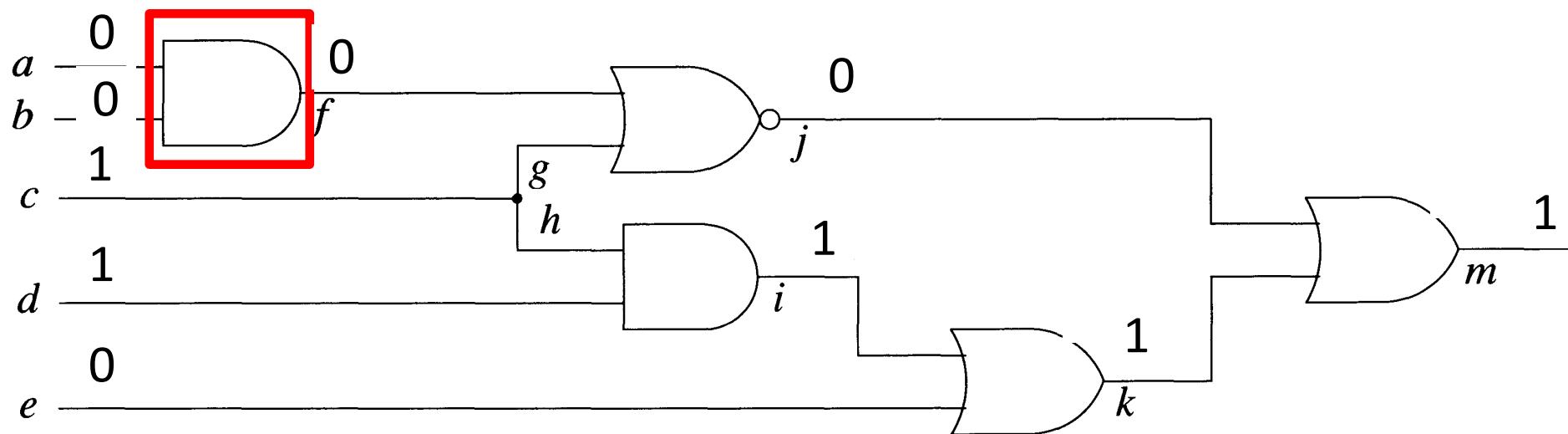


Figure 5.10

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

$$L_a = \{a_1\}, \quad L_b = \{b_1\}, \quad L_c = \{c_0\}, \quad L_d = \emptyset, \quad L_e = \emptyset$$

$$L_f = L_a \cap L_b = \emptyset, \quad L_g = L_c \cup \{g_0\} = \{c_0, g_0\}$$

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

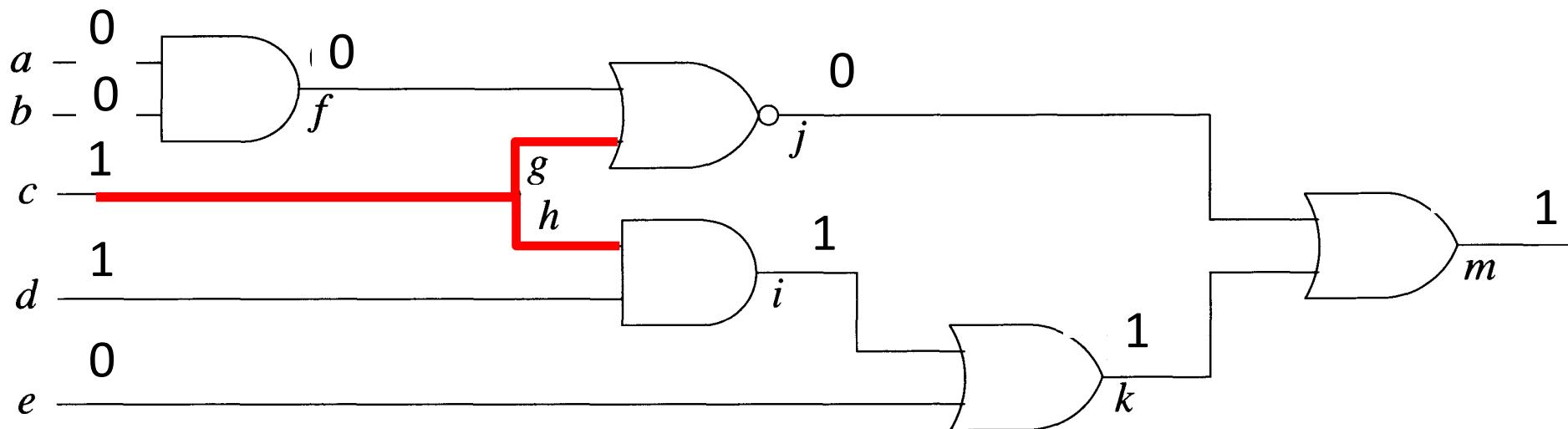


Figure 5.10

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

$$L_a = \{a_1\}, \quad L_b = \{b_1\}, \quad L_c = \{c_0\}, \quad L_d = \emptyset, \quad L_e = \emptyset$$

$$L_f = L_a \cap L_b = \emptyset, \quad L_g = L_c \cup \{g_0\} = \{c_0, g_0\}$$

$$L_h = L_c \cup \{h_0\} = \{c_0, h_0\}, \quad L_j = L_g \cup L_f = \{c_0, g_0\}$$

$$L_i = L_d \cup L_h = \{c_0, h_0\} \quad L_k = L_i - L_e = \{c_0, h_0\}$$

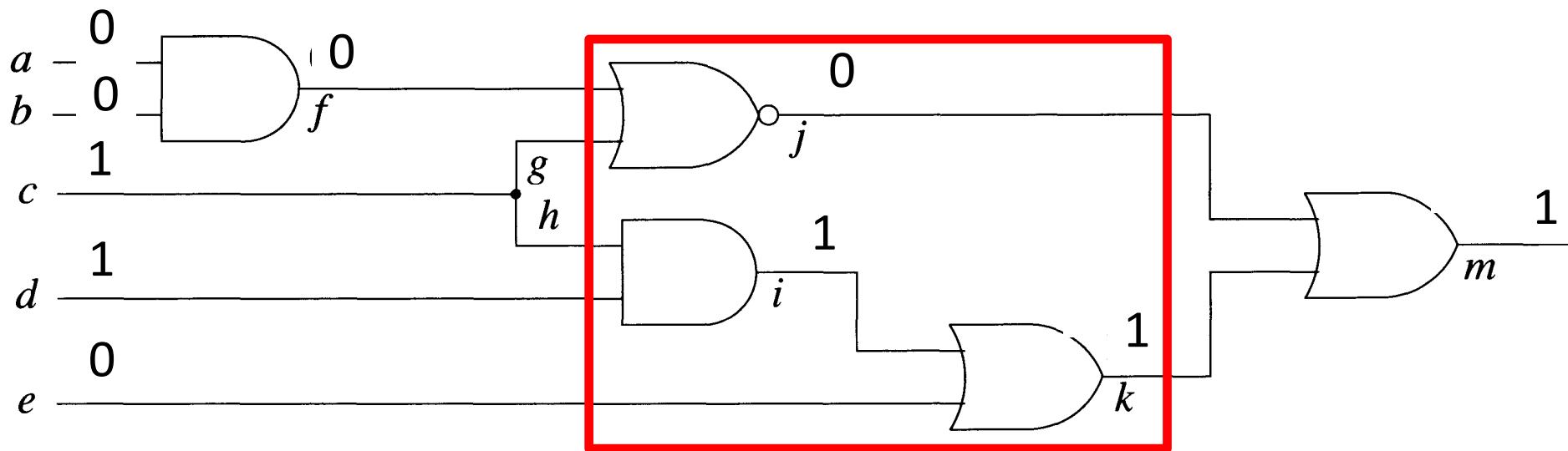


Figure 5.10

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

$$L_a = \{a_1\} \quad L_b = \{b_1\} \quad L_c = \{c_0\} \quad L_d = \emptyset \quad L_e = \emptyset$$

$$L_f = L_a \cap L_b = \emptyset \quad L_g = L_c \cup \{g_0\} = \{c_0, g_0\}$$

$$L_h = L_c \cup \{h_0\} = \{c_0, h_0\}. \quad L_j = L_g - L_f = \{c_0, g_0\}$$

$$L_i = L_d \cup L_h = \{c_0, h_0\} \quad L_k = L_i - L_e = \{c_0, h_0\}$$

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

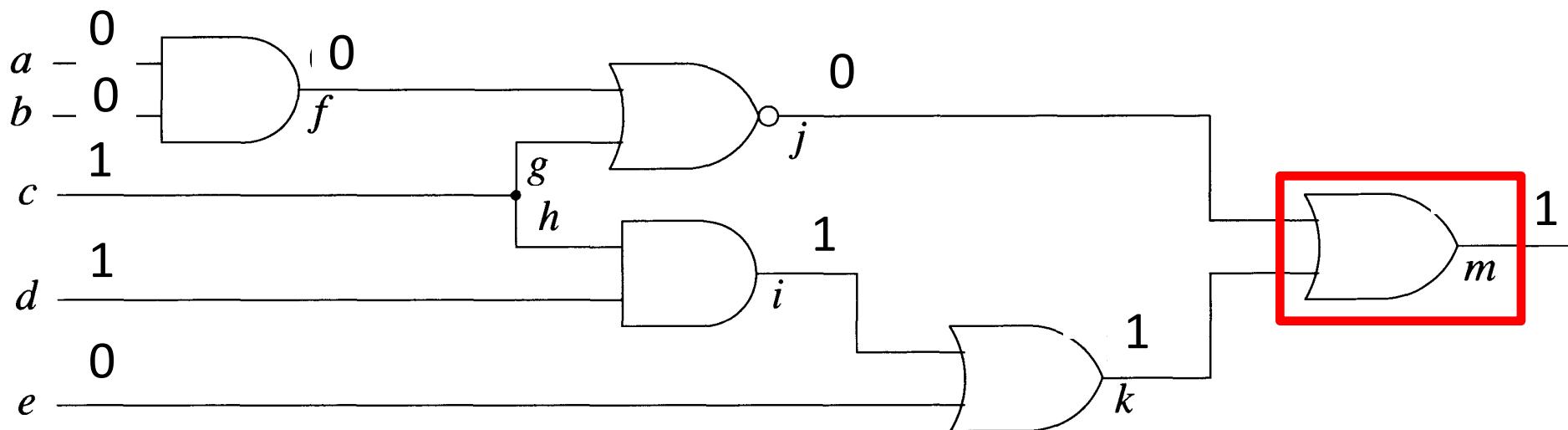


Figure 5.10

Now assume that next test vector is 11110. Redo the example.

$$L_a = \quad L_b =$$

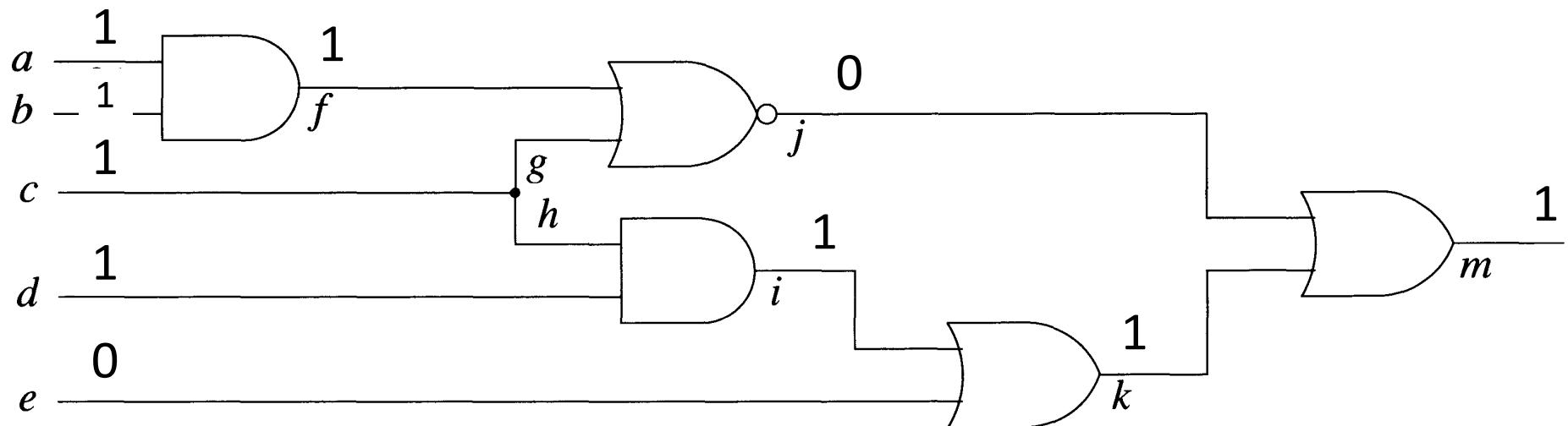
$$L_f = \quad L_g =$$

$$L_h =$$

$$L_j = \quad L_i =$$

$$L_k = \quad L_m =$$

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



Solution:

$$L_a = \{a_0\} \quad L_b = \emptyset$$

$$L_c = \{c_0\} \quad L_d = \emptyset \quad L_e = \emptyset$$

$$L_f = L_a \cup L_b = \{a_0\}$$

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

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

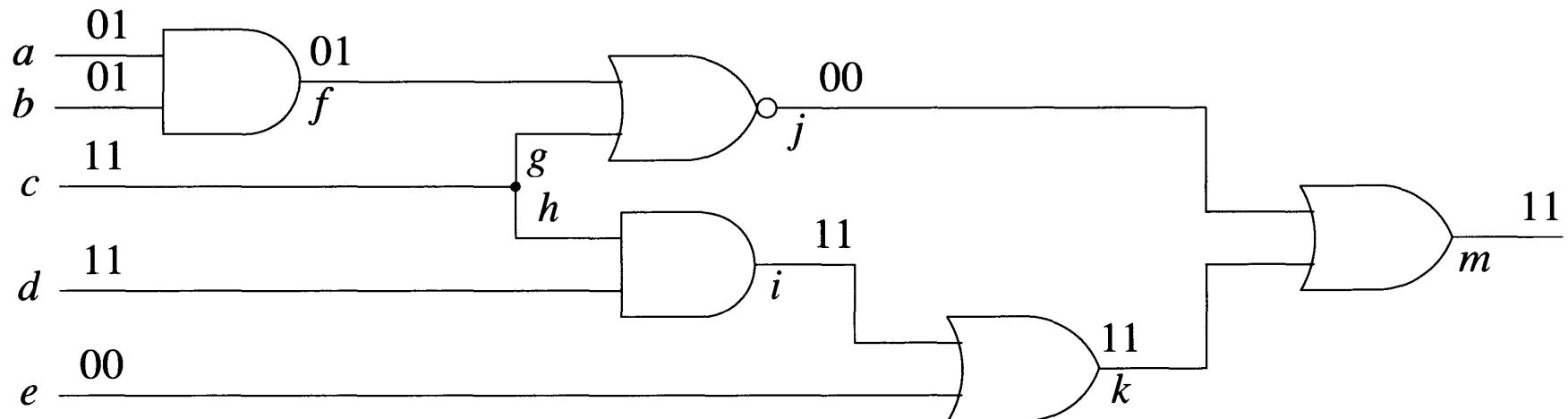
$$L_j = L_f \cap L_g = \emptyset$$

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

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

$$L_m = L_k - L_j = \{c_0, h_0\}$$

Fault c_0 is detected!



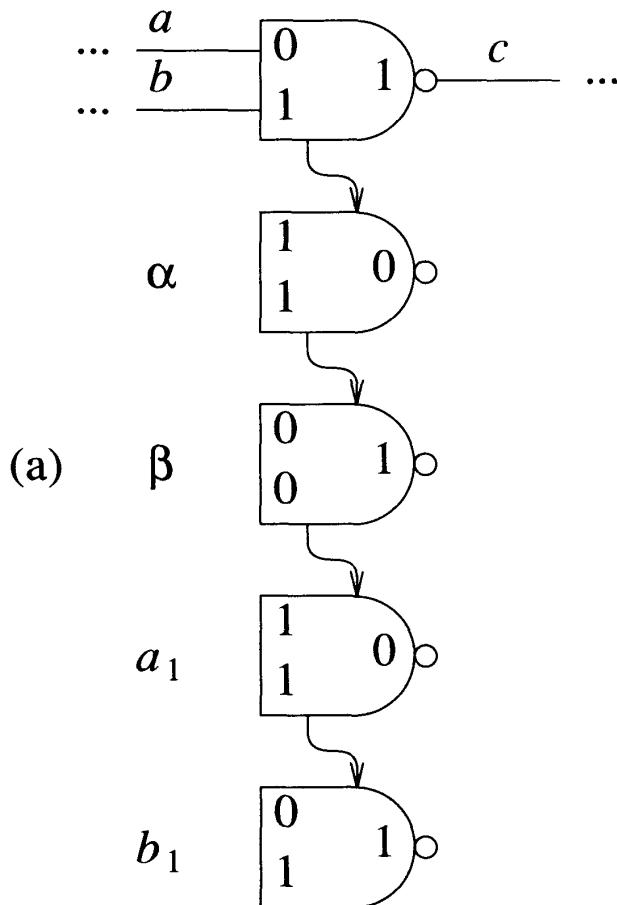
DS - Limitations

- Compatible only in part with functional level modeling
 - Applicable only to models with Boolean eqns.
- Limited to two or three logic values
- Cannot handle timing models
- Fault propagation mechanism cannot take full advantage of the concept of activity-directed simulation

Concurrent Fault Simulation

- Observation – Most of the time, most values in most fault circuits agree with those in the good circuit.
- Concurrent Method
 - simulates the good circuit N
 - For every faulty circuit N_f simulate *only those elements that differ with corresponding ones in N*
 - The differences of an element x in N is stored as a **concurrent fault list (CL_x)**

Concurrent List Example

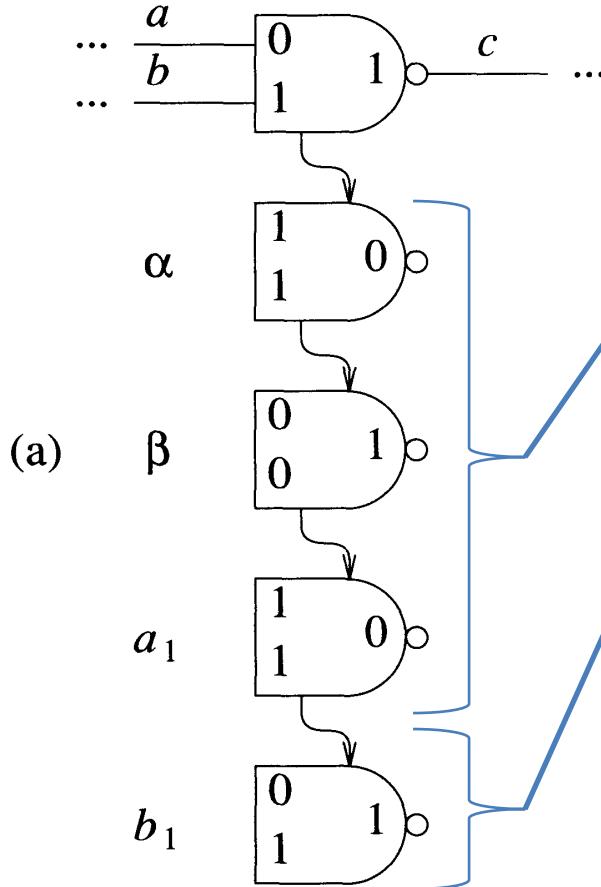


(b)

f	a	b	c
α	1	1	0
β	0	0	1
a_1	1	1	0
b_1	0	1	1

Figure 5.15 Concurrent fault list for gate c (a) Pictorial representation (b) Tabular representation

Two Cases of Differences



Let x_f be replica of x in N_f
 V_{xf} and V_x be <inputs, output>

Case 1: $V_{xf} \neq V_x$

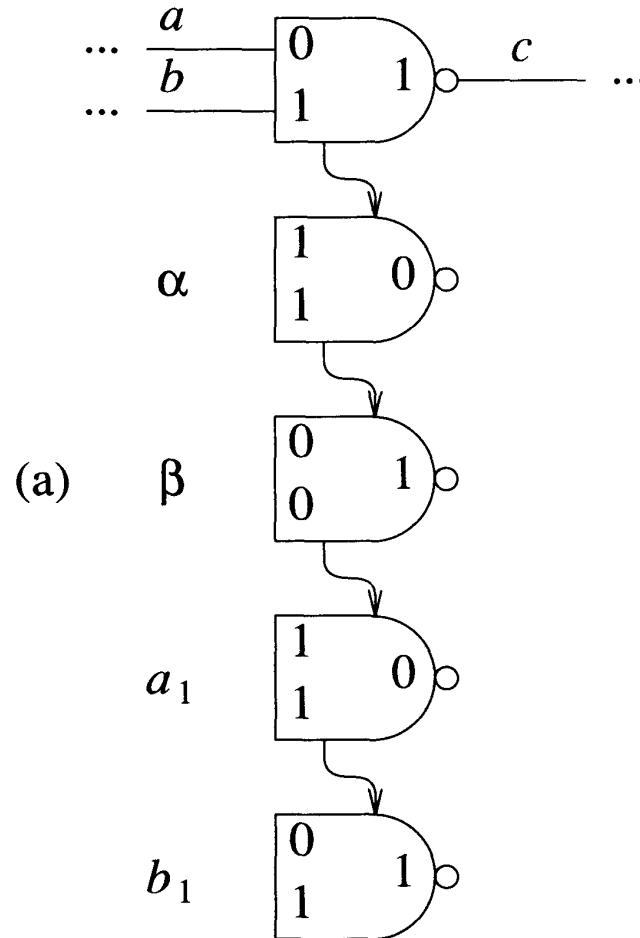
This happens when fault effect of f propagates to x

Case 2: $V_{xf} = V_x$

This happens when f is a local fault
(i.e., input/output fault)

Note: Even if the V_x and V_{xf} are equal,
the elements are different because
of the local fault.

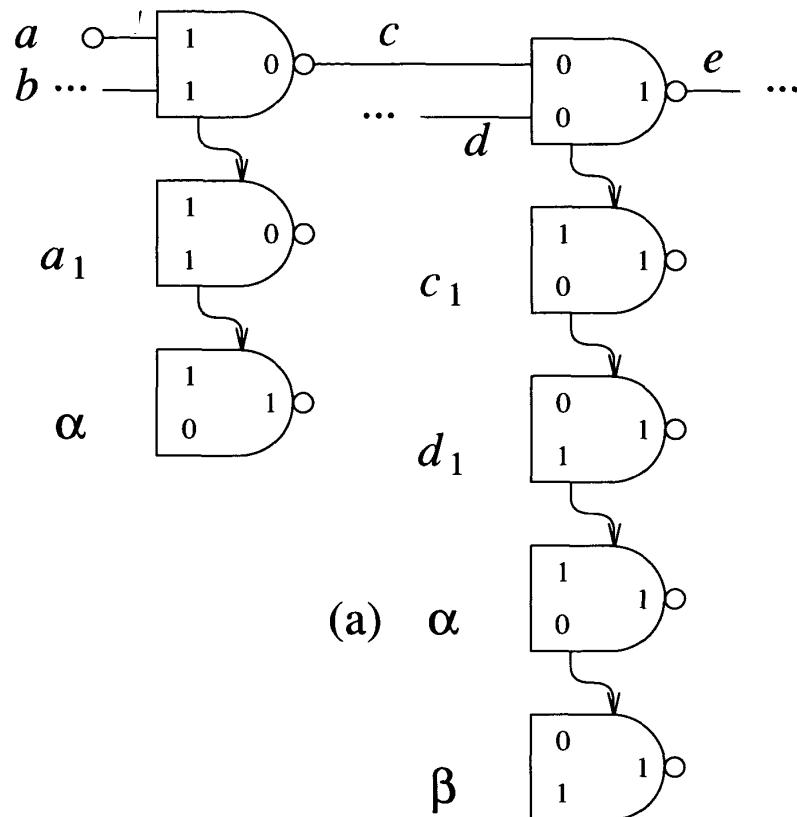
Visible Faults



A fault is **visible** on line i when the values of i in N and N_f are different.

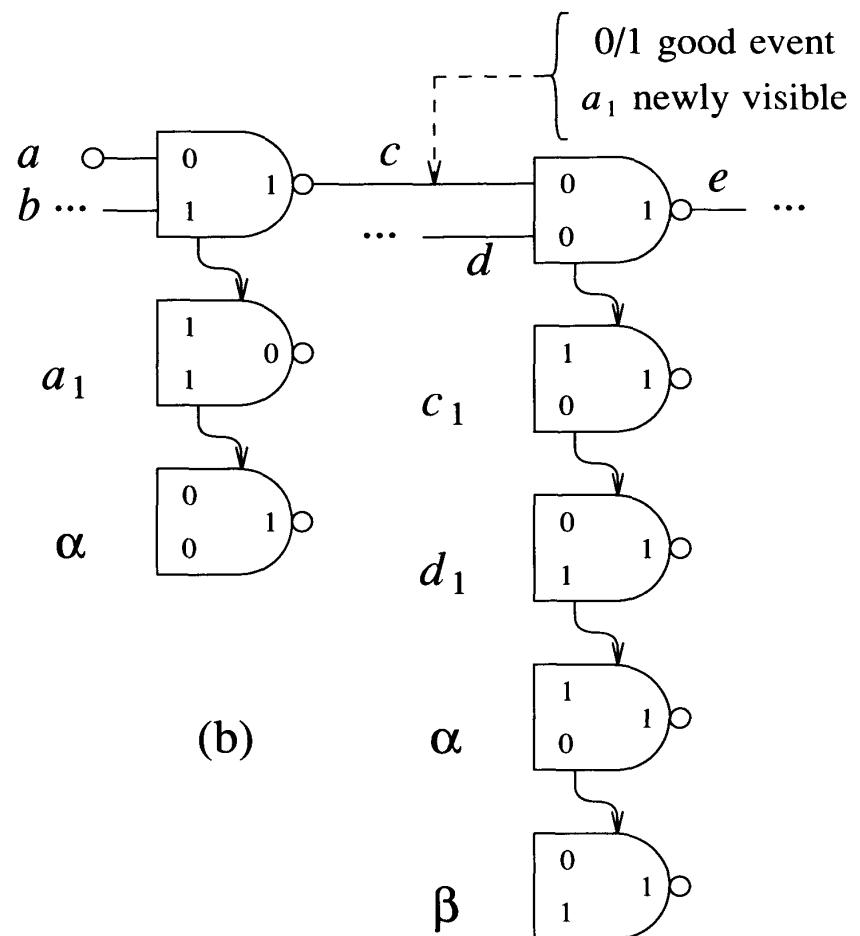
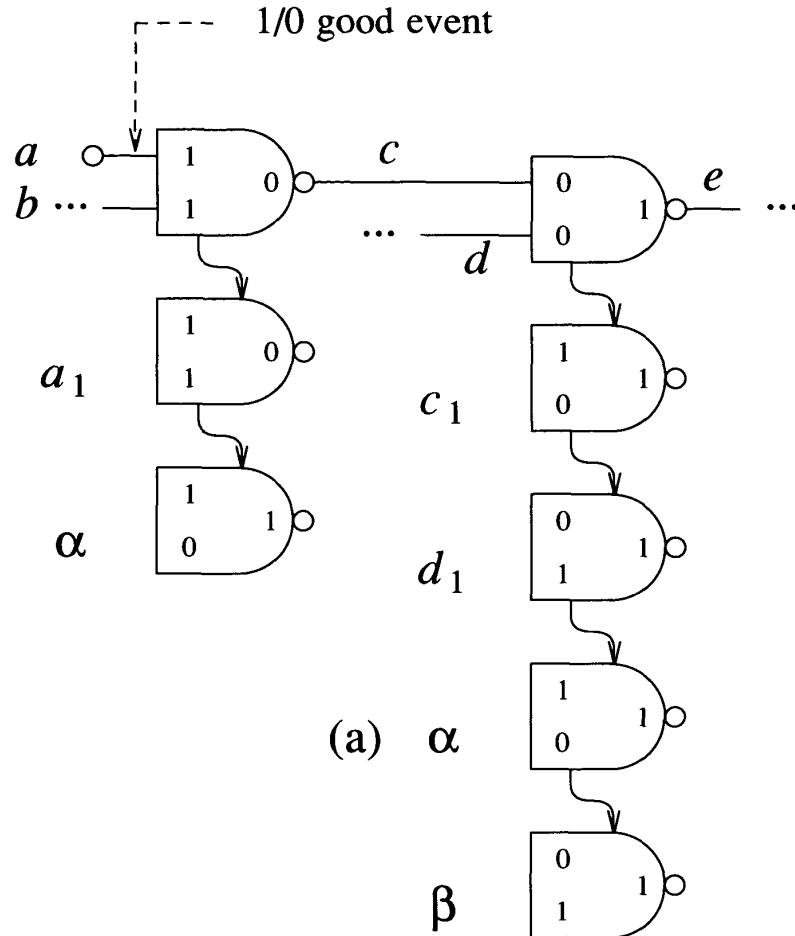
A deductive fault list includes all visible faults, which is subset of the concurrent fault list.

CFS - Example

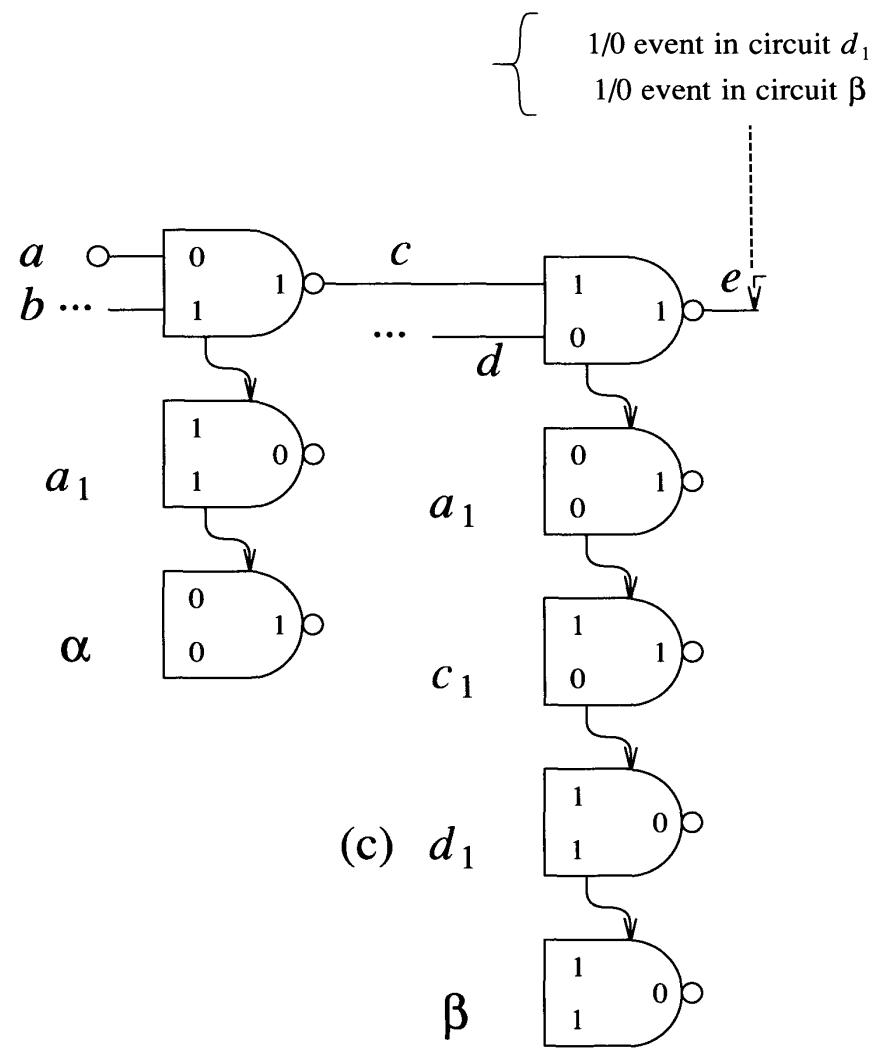
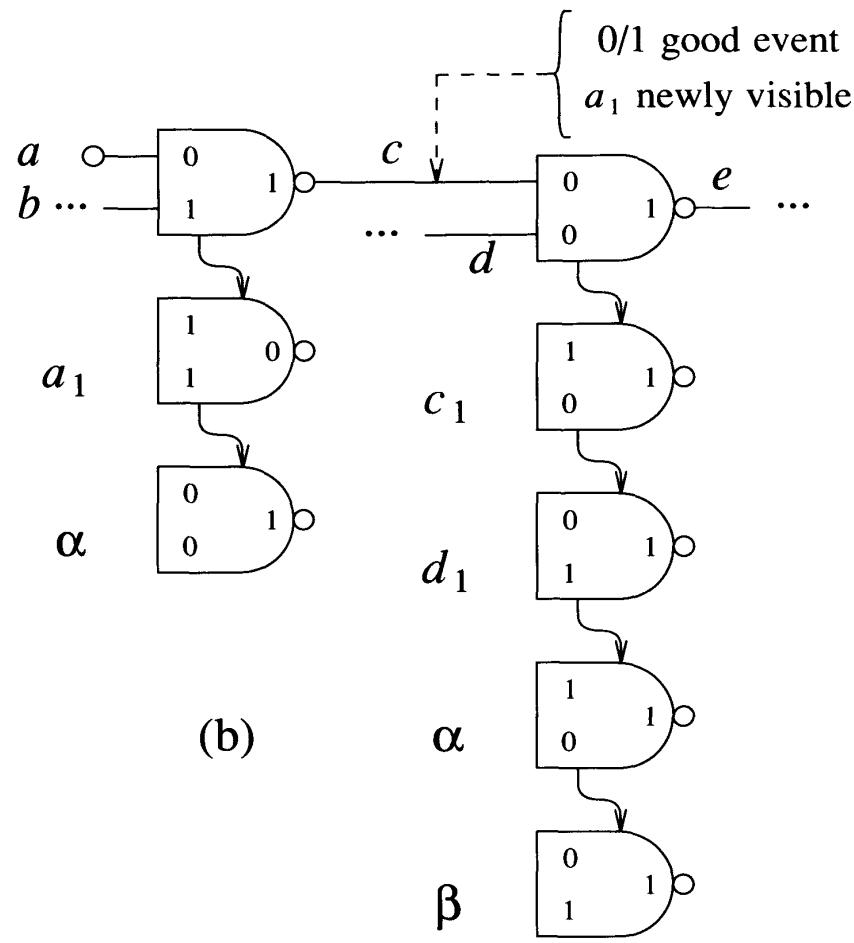


What are those faults in the initial state?

CFS - Example



CFS Example – Contd.



Concurrent Simulation

- Individually evaluates elements in both good and faulty circuits
- A line i may change even if i is stable in good circuit (see gate d_1 in previous example)
- A line i in the good circuit and some faulty circuits may also have simultaneous but different events

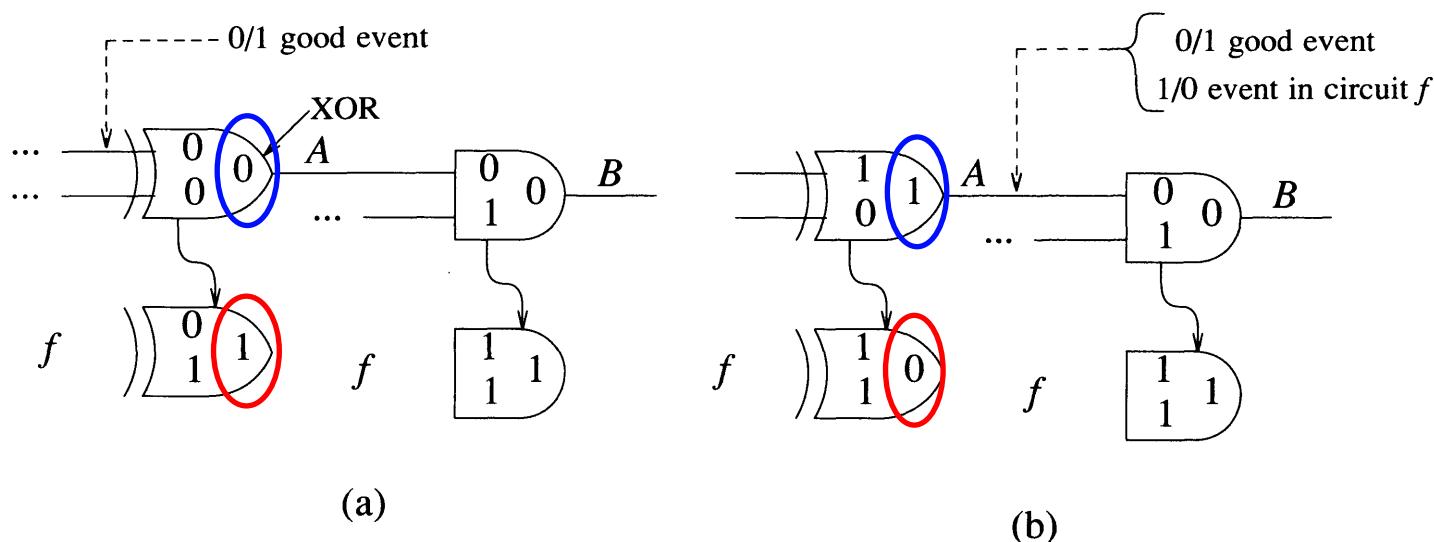
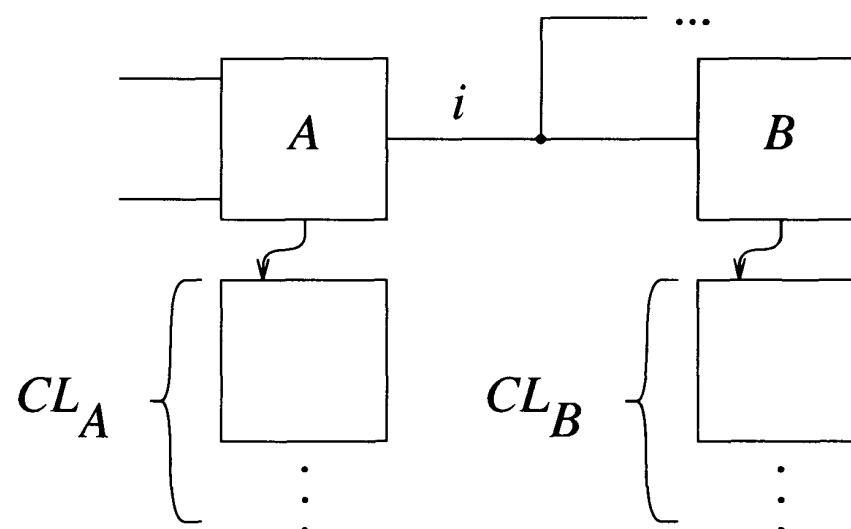


Figure 5.17

Composed Event

- For a given input event on A, we compute the outputs in all copies of A in the fault list
- Let the output list be $L = \langle (f_0, v'_{f0}), (f_1, v'_{f1}), \dots (f_n, v'_{fn}) \rangle$
- Composed Event:
 - A set of simultaneous events occurring on a line
 - Represented as (i, L)

Figure 5.18



Processing of a composed event (i, L) at element A

$NV = \emptyset$

if i changes in the good circuit **then**

begin

 set i to v' in the good circuit

for every $f \in CL_A$

begin

if $f \in L$ **then**

begin

 set i to v'_f in circuit f

if $V_{Af} = V_A$ **then** delete f from CL_A

end

else /* no event in circuit f */

if $v_f = v$ **then** add newly visible fault f to NV

else if $V_{Af} = V_A$ **then** delete f from CL_A

end

end

else /* no good event for i */

for every $f \in L$

begin

 set i to v'_f in circuit f

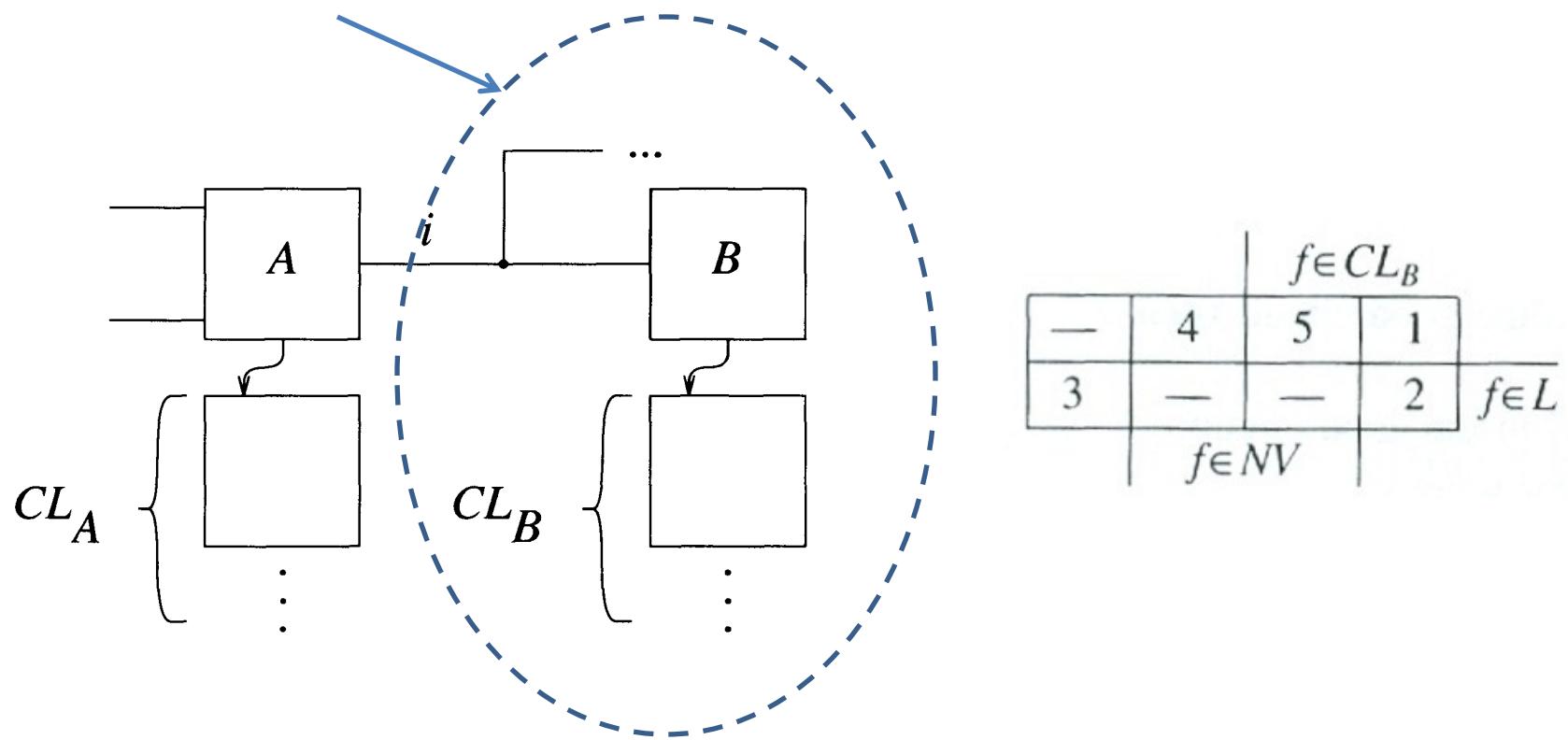
if $V_{Af} = V_A$ **then** delete f from CL_A

end

NV : newly visible faults

Processing of element $B_f \in CL_B$

→ After updating the CL_A of source element A , we need to update values and CL_B of every element B on the fanout list of i and evaluate activated elements



Case 1: $f \in CL_B, f \notin L, f \notin NV$

$f \in CL_B$				
—	4	5	1	
3	—	—	2	$f \in L$
			$f \in NV$	

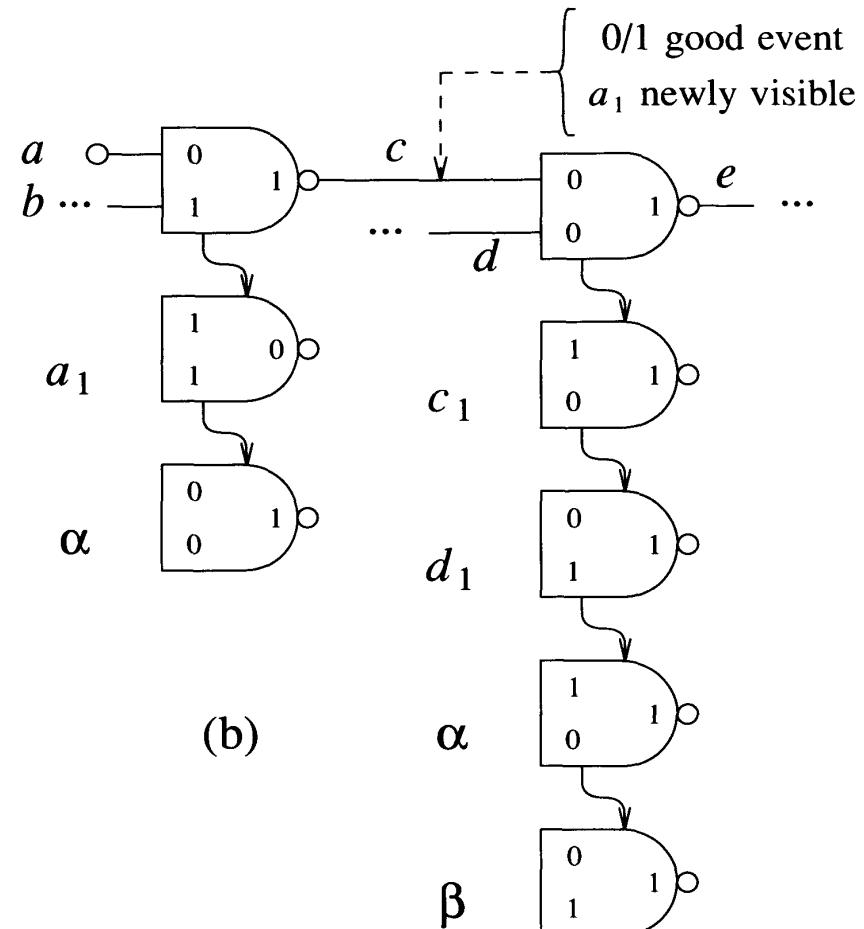
Remark: B_f exists in CL_B and no independent event on i occurs in N_f

Action

If good event exists and can propagate in N_f then activate B_f

Example:

Change c 0/1 propagates in d_1 and β but not in c_1 and α



Case 2: $f \in CL_B$, $f \in L$, $f \notin NV$

$f \in CL_B$	—	4	5	1
—	3	—	—	2
$f \in NV$				$f \in L$

Remark: B_f exists in CL_B and an independent event on i occurs in N_f

Action

Activate B_f

Example:

f in CL_B

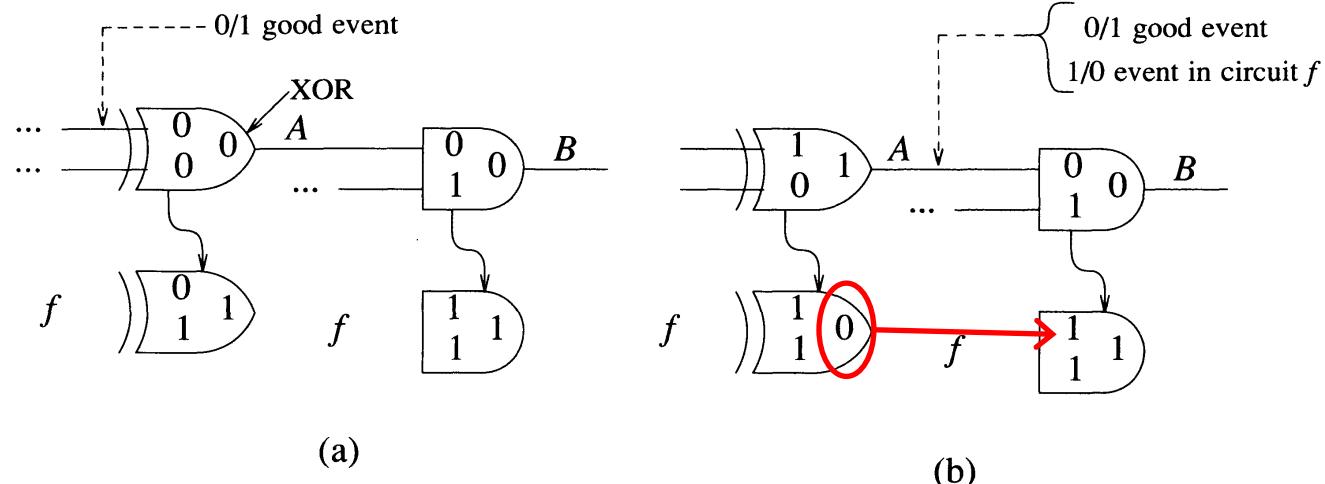


Figure 5.17

Case 3: $f \notin CL_B, f \in L, f \notin NV$

				$f \in CL_B$
$f \in NV$	—	4	5	1
$f \in L$	3	—	—	2

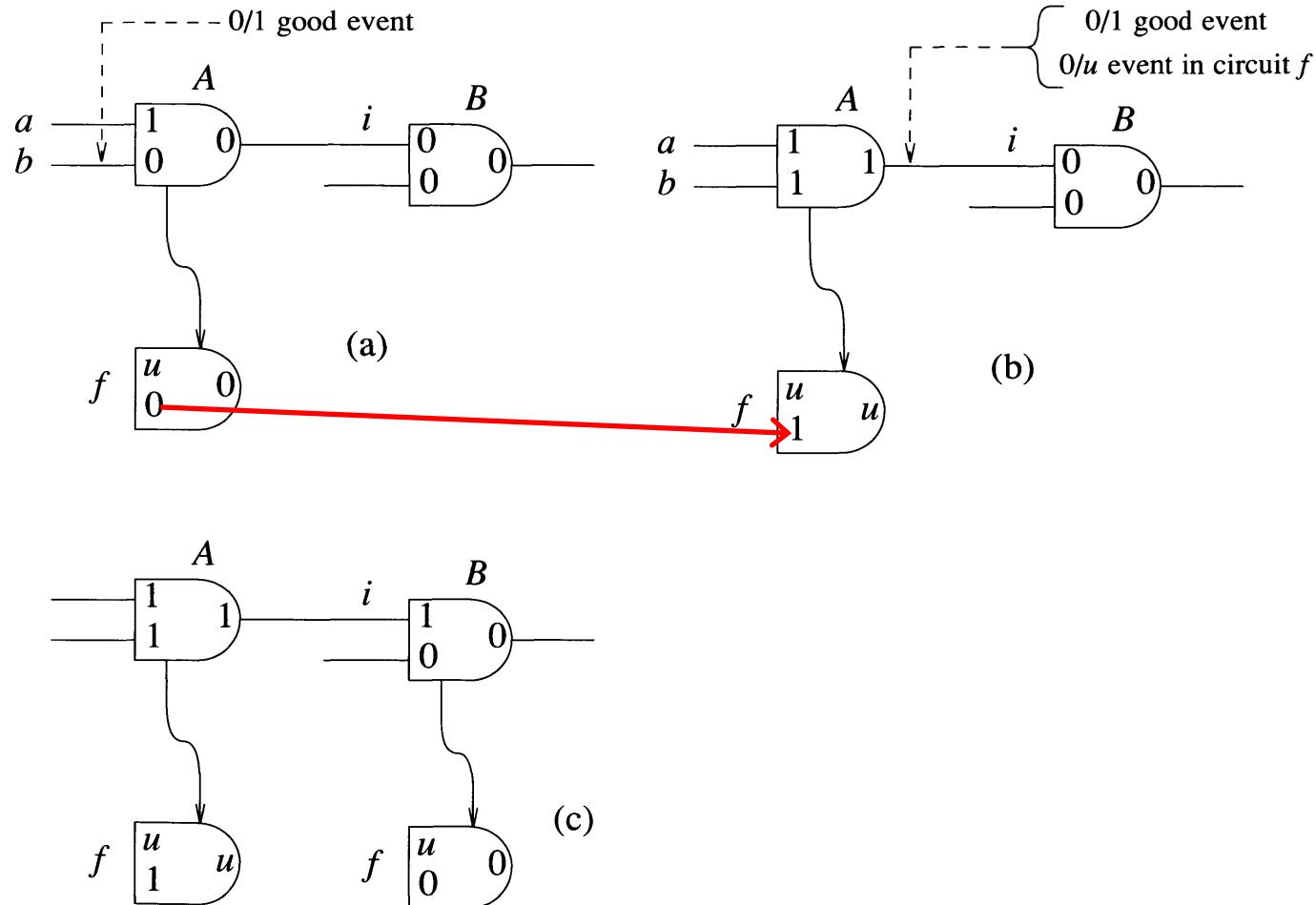
Remark: An independent event on i occurs in N_f but f does not appear in CL_B

Action: Add an entry for f to CL_B and activate B_f

Case 3: $f \notin CL_B$, $f \in L$, $f \notin NV$

$f \in CL_B$	—	4	5	1	
3	—	—	—	2	$f \in L$
$f \in NV$					

Example:



Case 4: $f \notin CL_B, f \notin L, f \in NV$

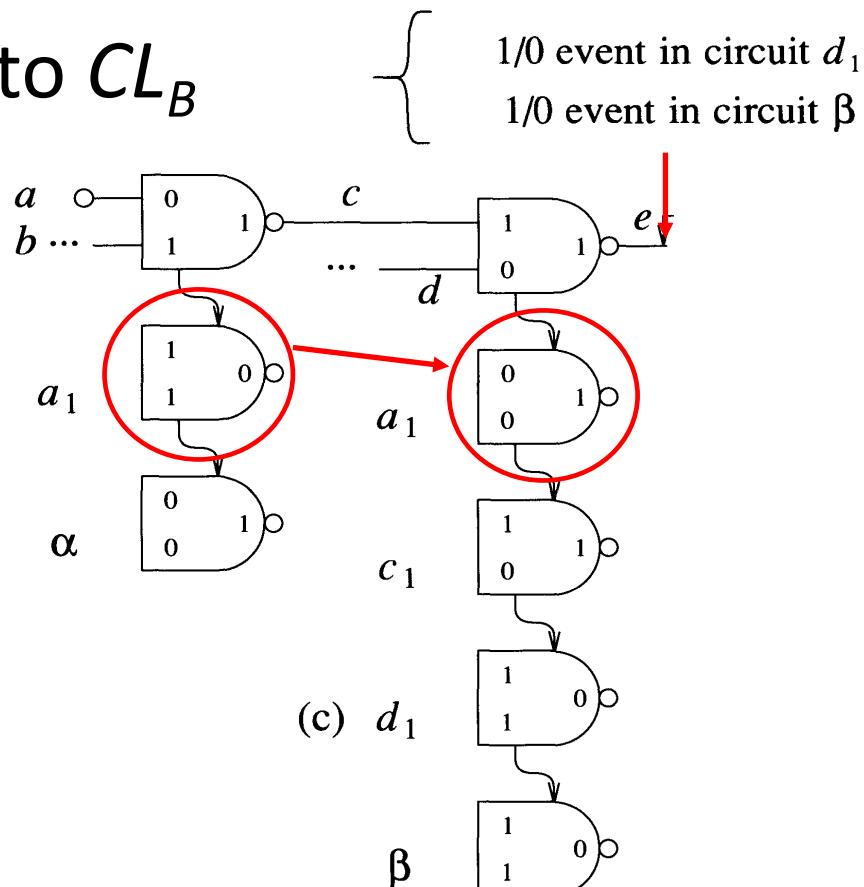
$f \in CL_B$	—	4	5	1	—
—	3	—	—	2	$f \in L$
$f \in NV$	—	—	—	—	—

Remark: f is newly visible on line i and does not appear in CL_B

Action: Add an entry for f to CL_B

Example:

Add a_1 to CL_e



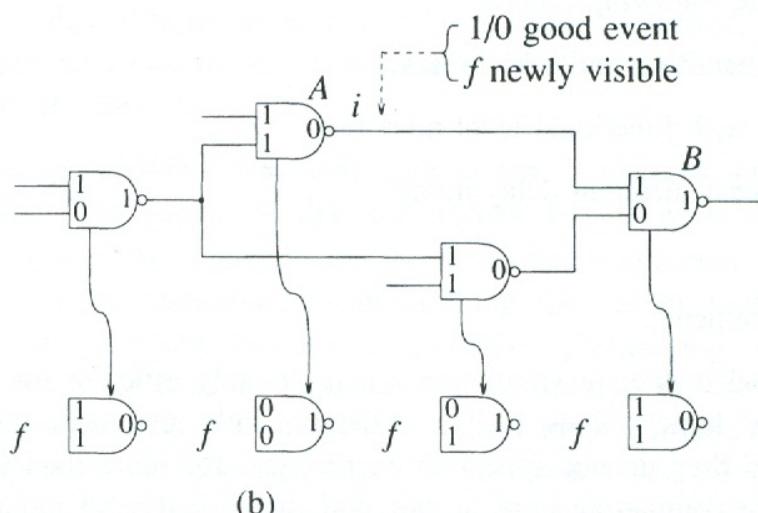
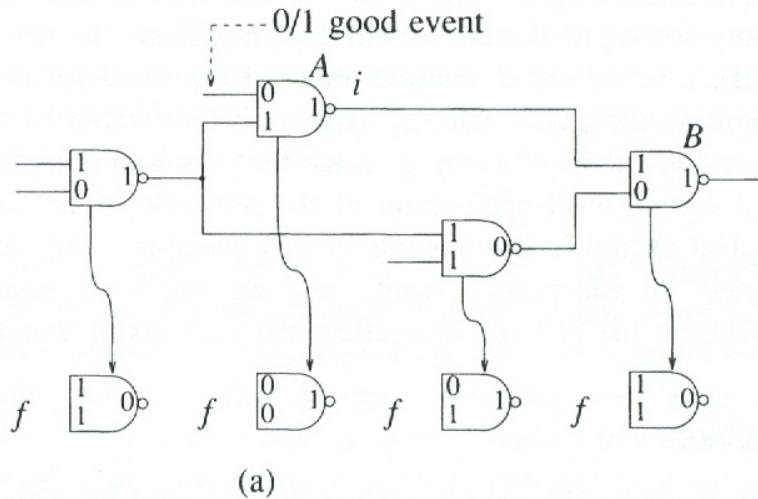
Case 5: $f \in CL_B, f \notin L, f \in NV$

$f \in CL_B$	—	4	5	1	
—	4	5	1	2	$f \in L$
3	—	—	2		$f \in NV$

Remark: f is newly visible on line i but an entry is already present in CL_B

Action: No Action.

Example: In a comb. circuit this occurs with reconv. fanout



Comparison

Criteria	Parallel	Deductive	Concurrent
Multiple Logic Values	Impractical for more than 3 logic values	Impractical for more than 3 logic values	No limit
Functional level Modeling	Partially compatible	Partially compatible	Fully compatible
Different Delay Models	No	No	Yes
Speed*	n^3	n^2	faster?
Storage Reqs.	Medium	Medium	Large

* Comparison for large combinational circuit with n gates. No comparison between deductive and concurrent reported.

Backup

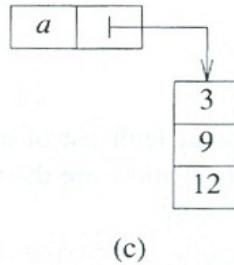
Fault Storage

$$L_a = \{3, 9, 12\}$$

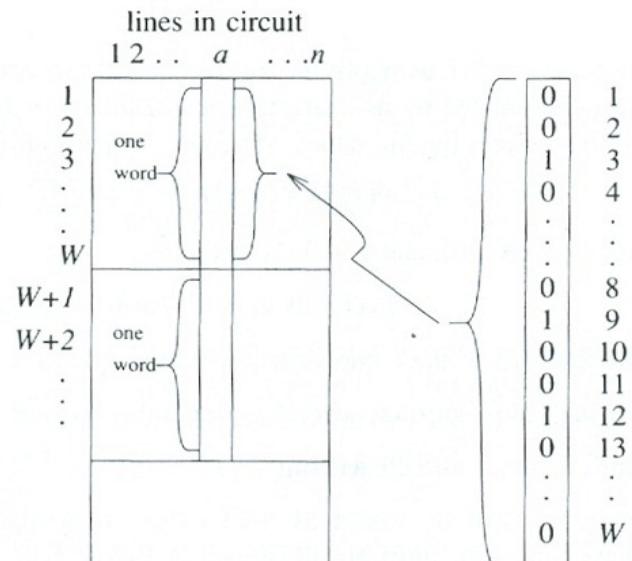
(a)



(b)



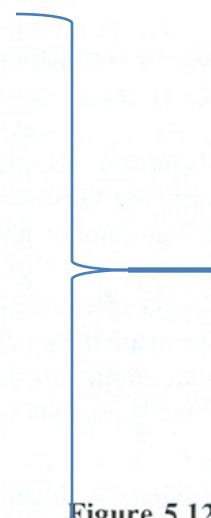
(c)



(d)

Figure 5.12 Three storage structures for lists (a) Fault list (b) Linked list (c) Sequential table (d) Characteristic vector

- Fault insertion Bit = 1
- Fault deletion Bit = 0
- Union – Bit-wise OR
- Intersection – Bit-wise AND
- Memory Intensive



CIS 4930 Digital System Testing

Testing for Single Stuck-at Faults (SSFs)

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

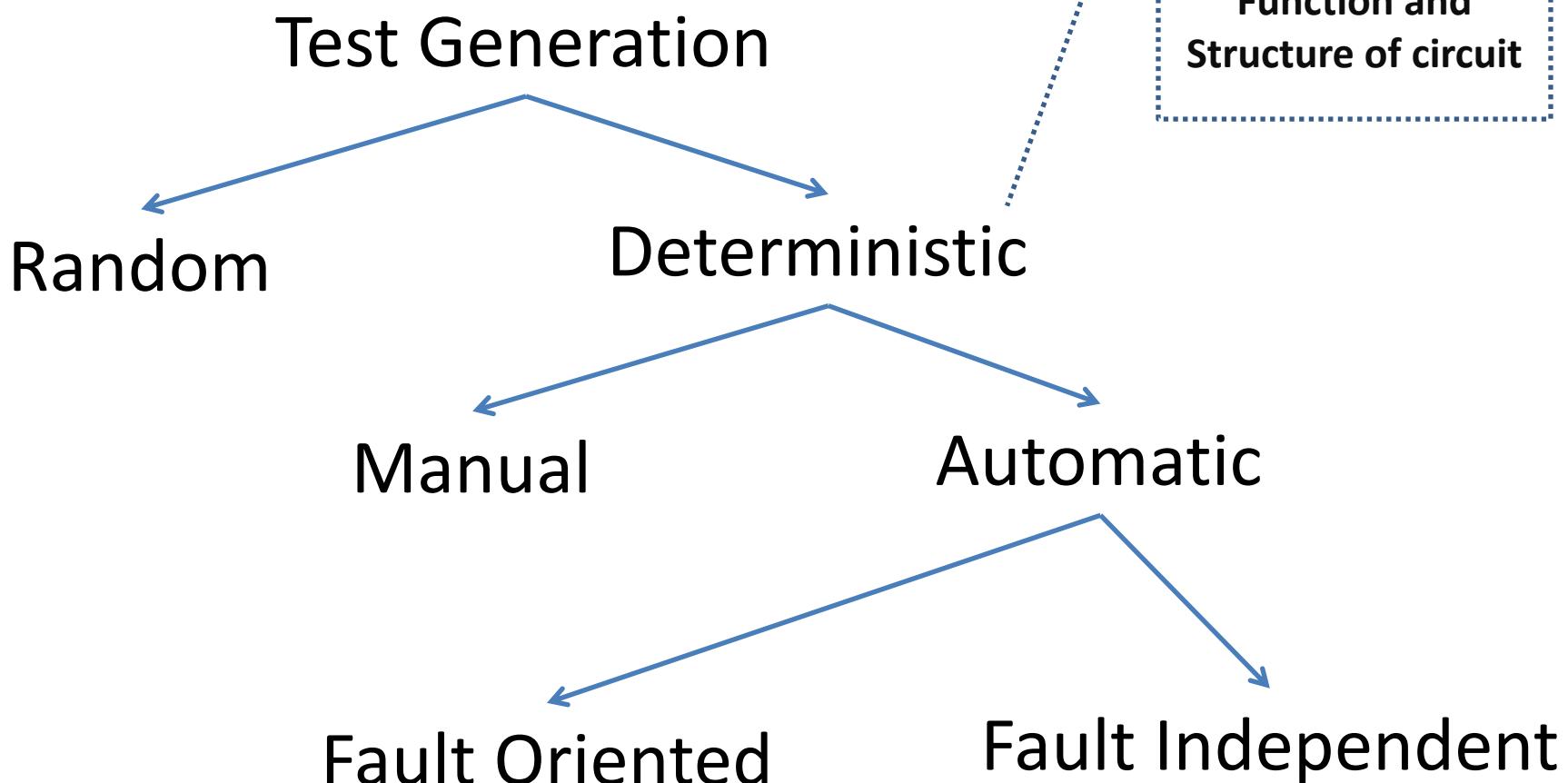
Testing Generation

Testing Generation (TG) is a complex problem

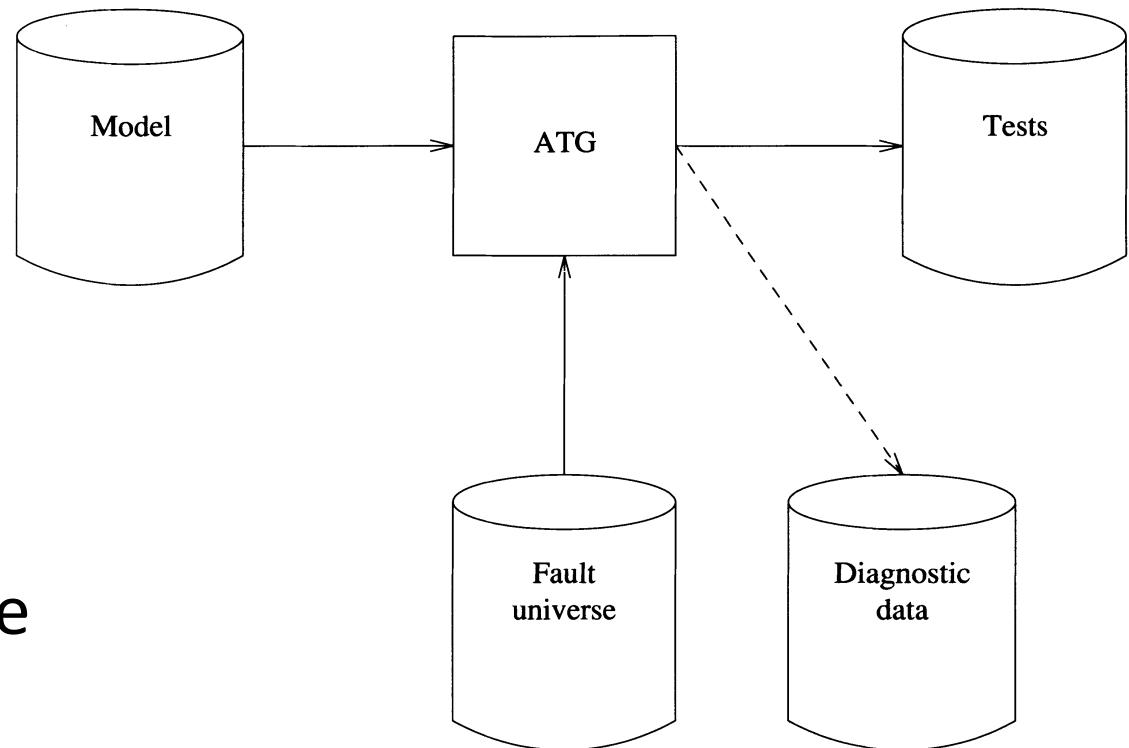
We are interested in:

- 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 \bar{D} represent 0/1

v/v_f	
0/0	0
1/1	1
1/0	D
0/1	\bar{D}

AND	0	1	D	\bar{D}	X
0	0	0	0	0	0
1	0	1	D	\bar{D}	X
D	0	D	D	0	X
\bar{D}	0	D'	0	\bar{D}	X
X	0	X	X	X	X

OR	0	1	D	\bar{D}	X
0	0	1	D	D'	0
1	1	1	1	1	1
D	D	1	D	1	X
\bar{D}	\bar{D}	1	1	\bar{D}	X
X	X	1	X	X	X

Fig 6.3 TG for $l \leftarrow s-a-v$ in Fanout Free circuit

begin

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

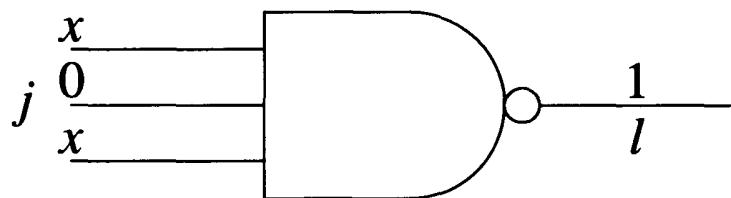
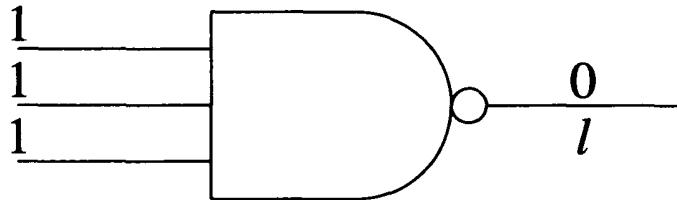
Justify(l , \bar{v}) // justification of line l

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

else *Propagate* (l , \bar{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

$intval = val \oplus i$

if ($intval = \bar{c}$)

then for every input j **of** l

Justify (j, intval)

else

begin

 select one input (j) of l

Justify (j, intval)

end

end

Error Propagation – Fanout Free circuit

Propagate (l , err)

/ err is D or \bar{D} */*

begin

 set l to err

if l is PO **then return**

k = the fanout of l

c = controlling value of k

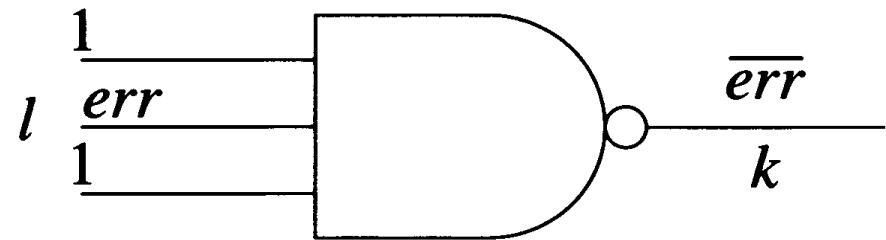
i = inversion of k

for every input j of k other than l

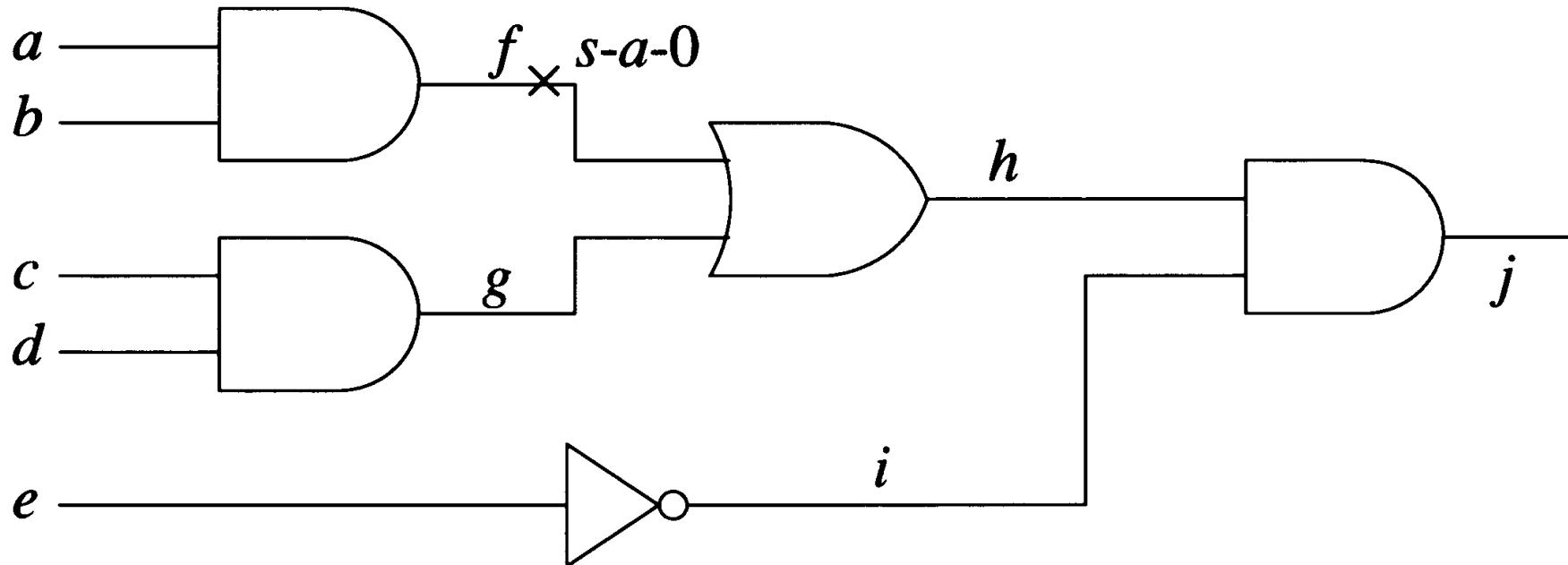
Justify (j , \bar{c})

Propagate (k , $err \oplus i$)

end

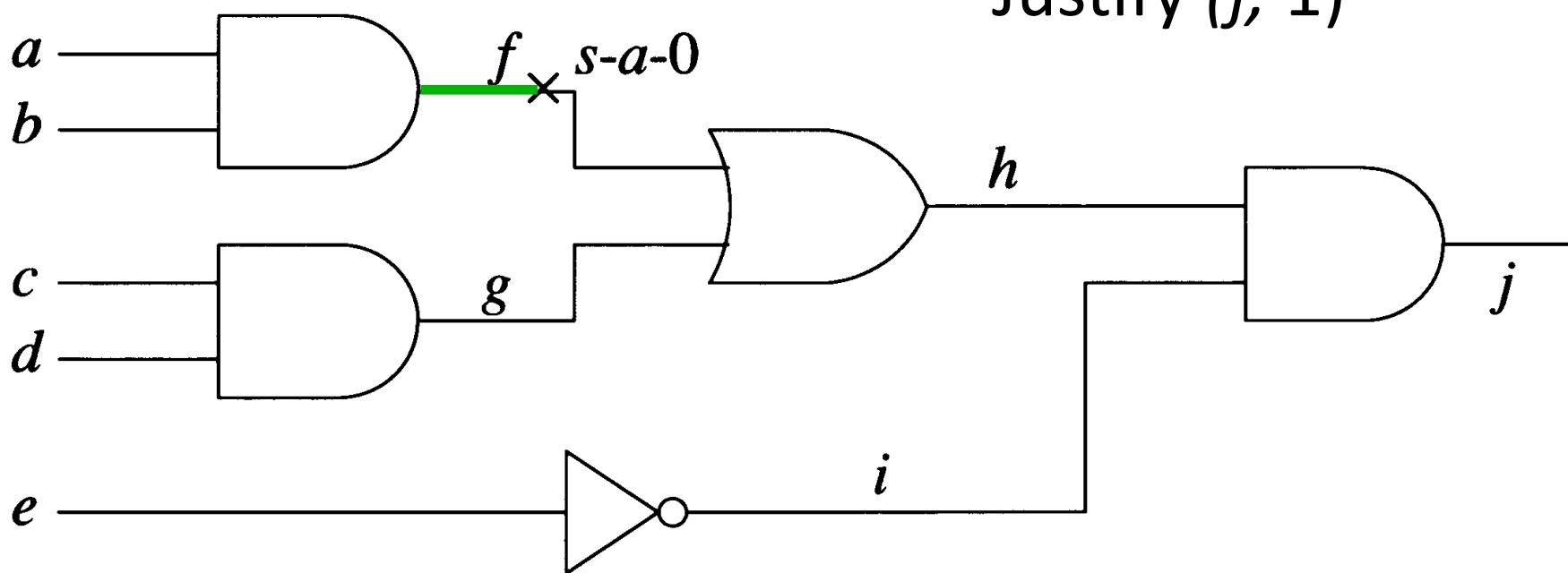


Example 6.1

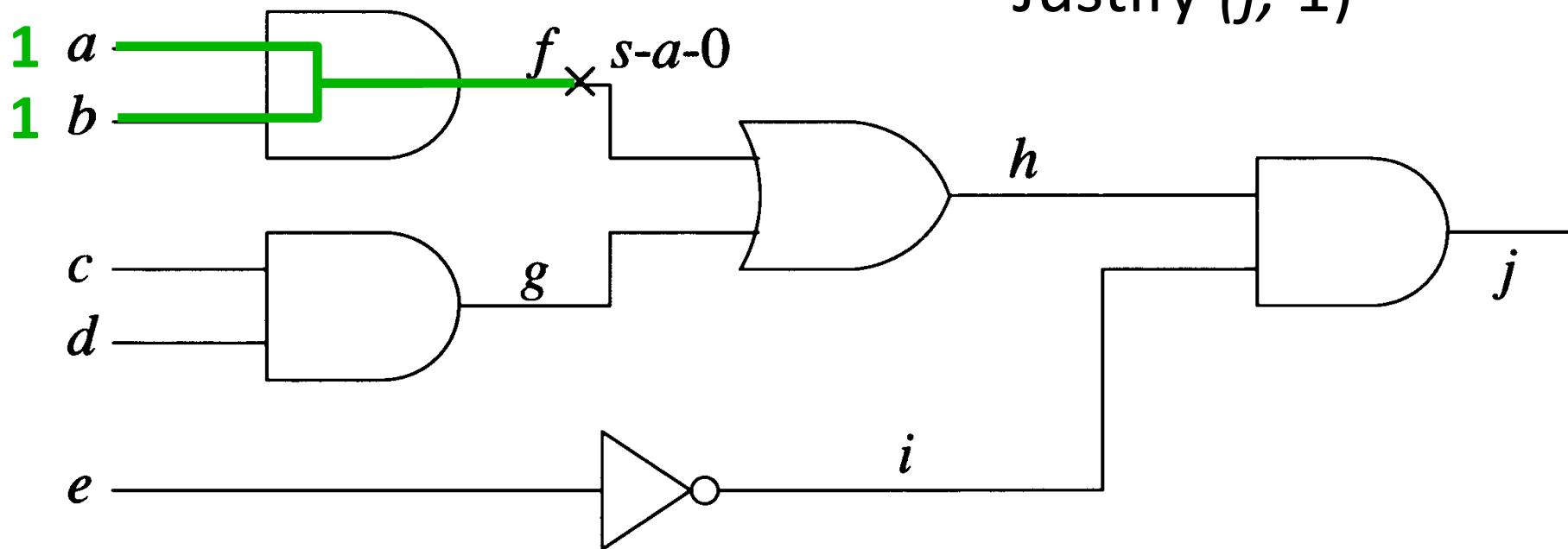


Find an input vector such that $f \times s-a-0$ is observable on j

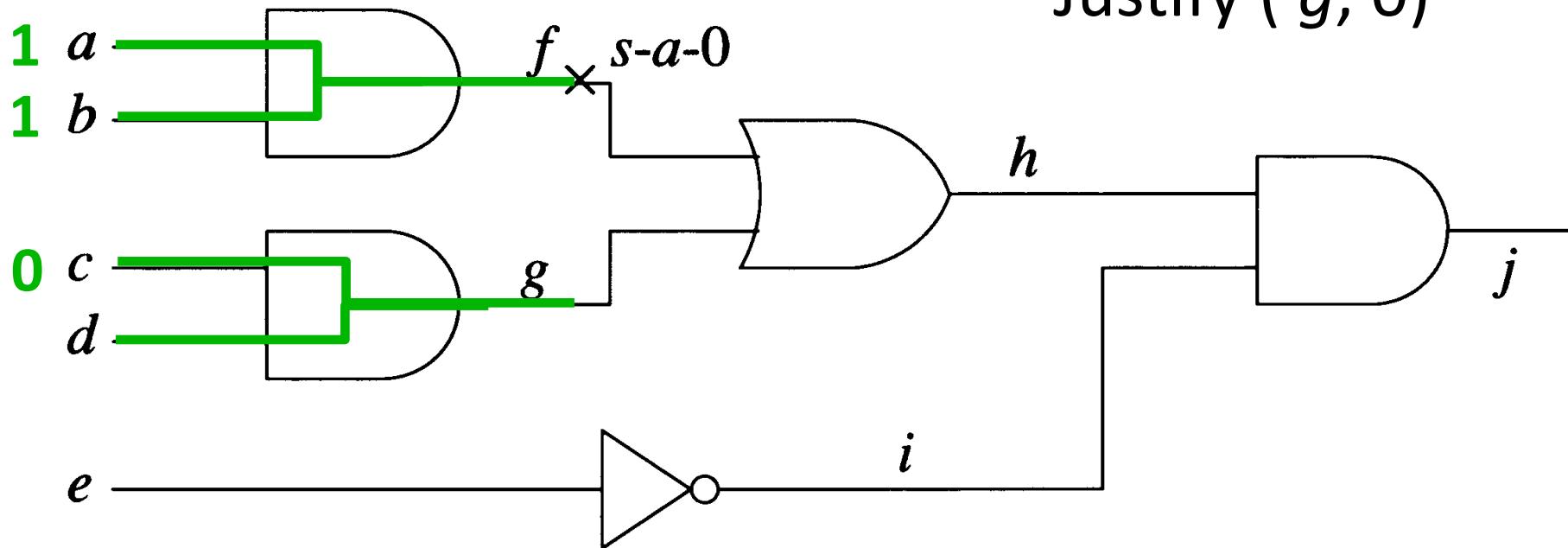
Example 6.1



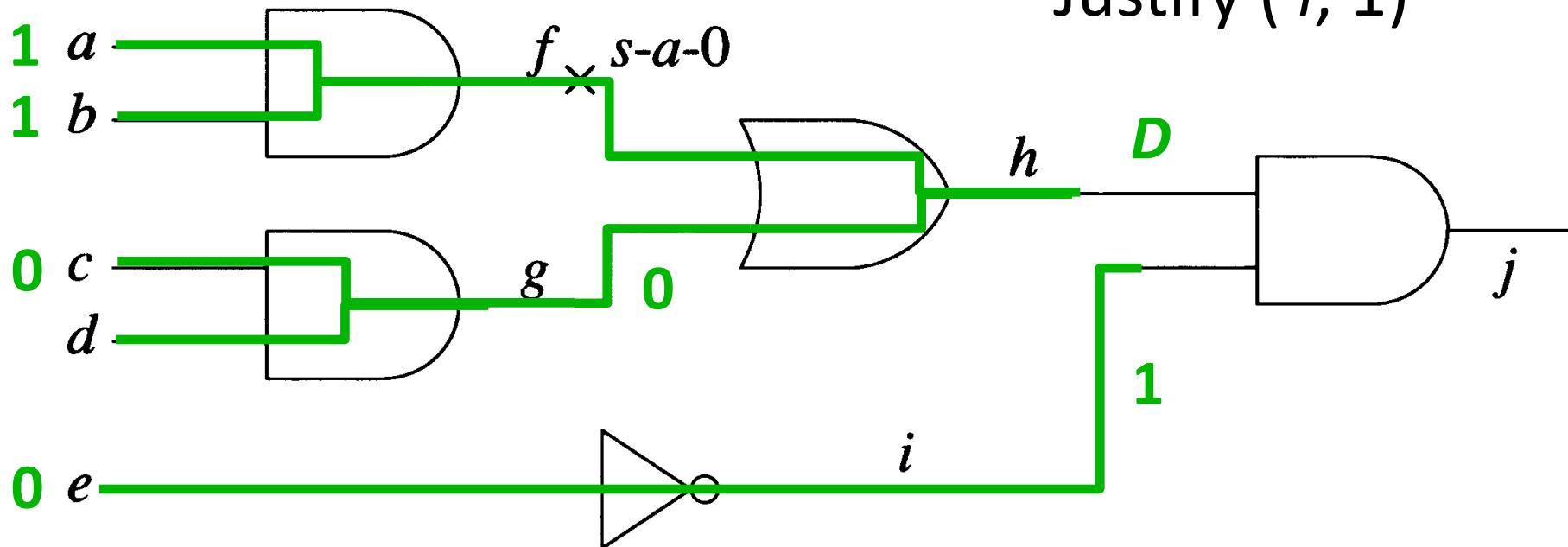
Example 6.1



Example 6.1

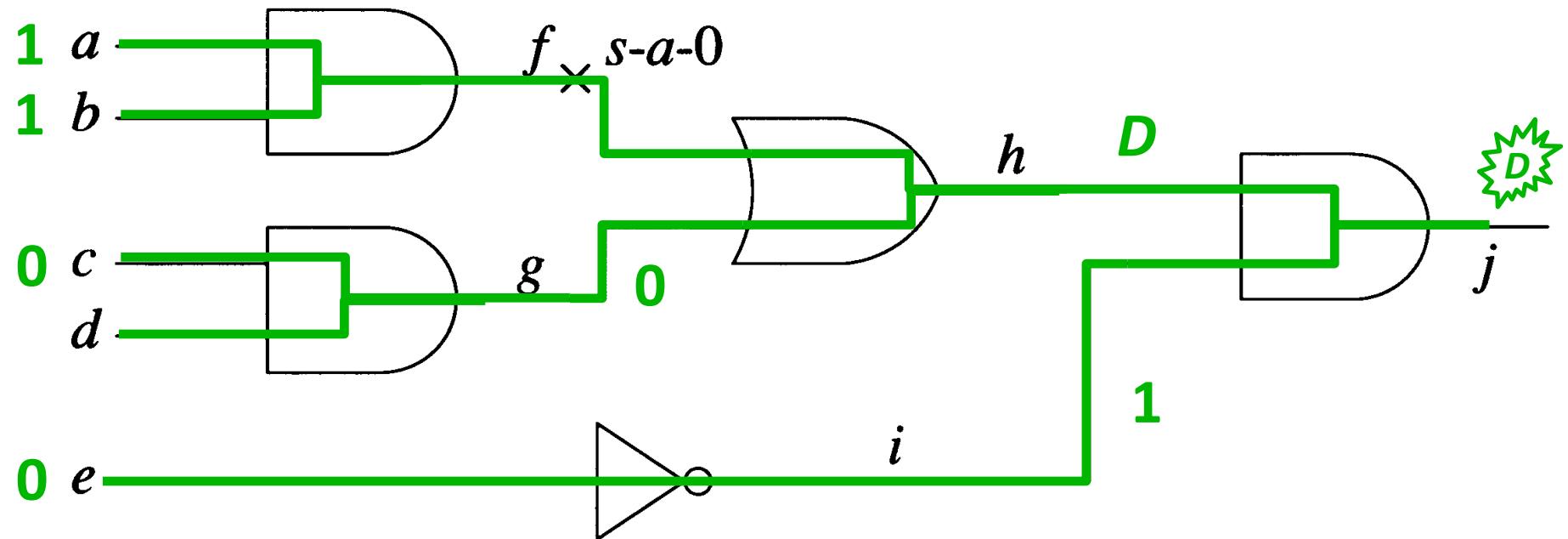


Example 6.1

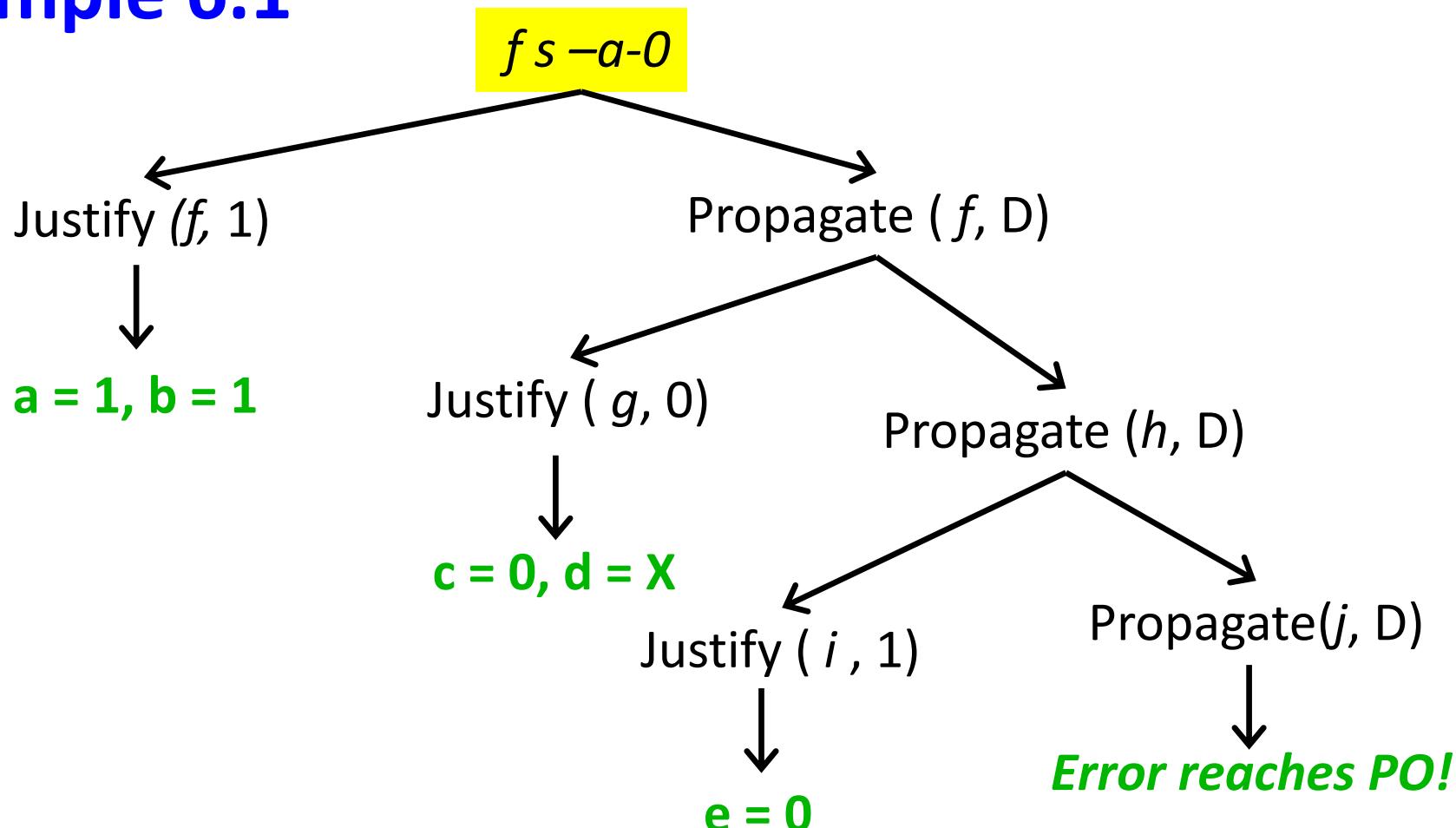


Example 6.1

Propagate (j , D)



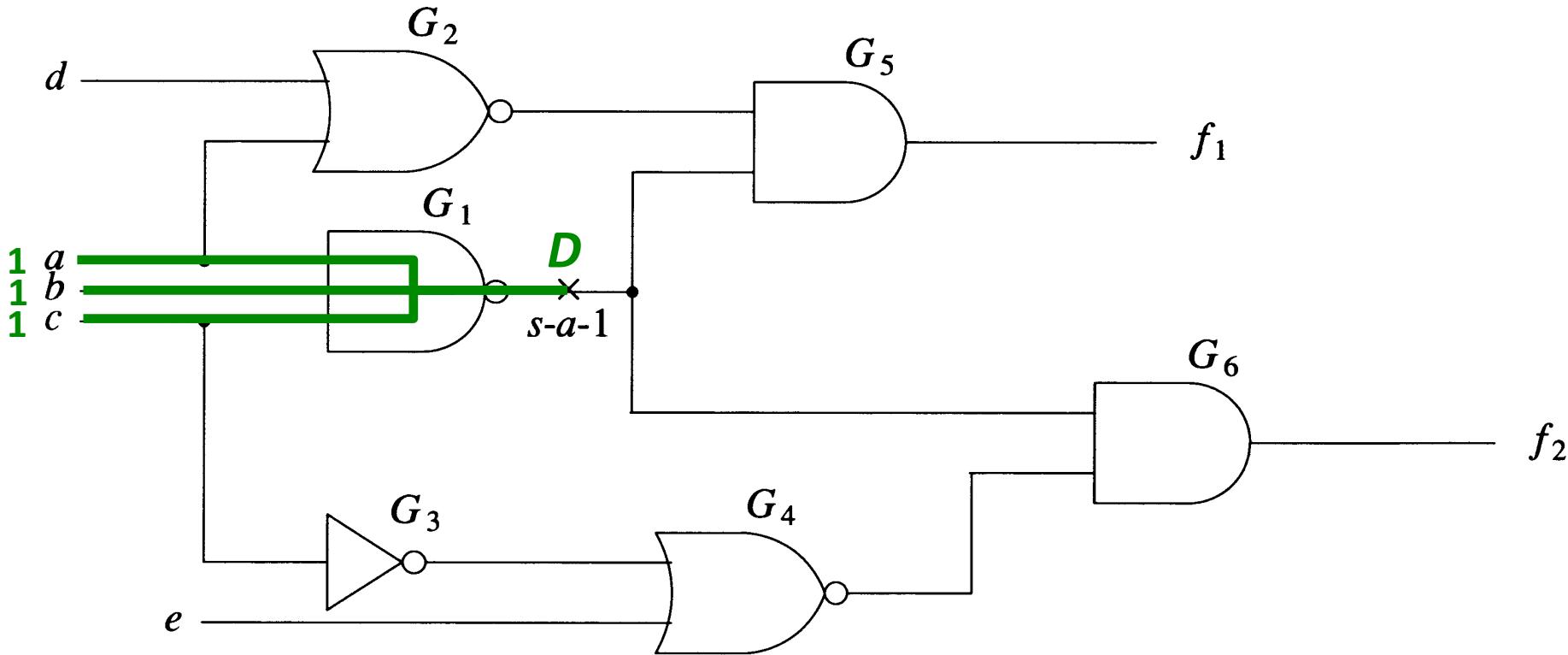
Example 6.1



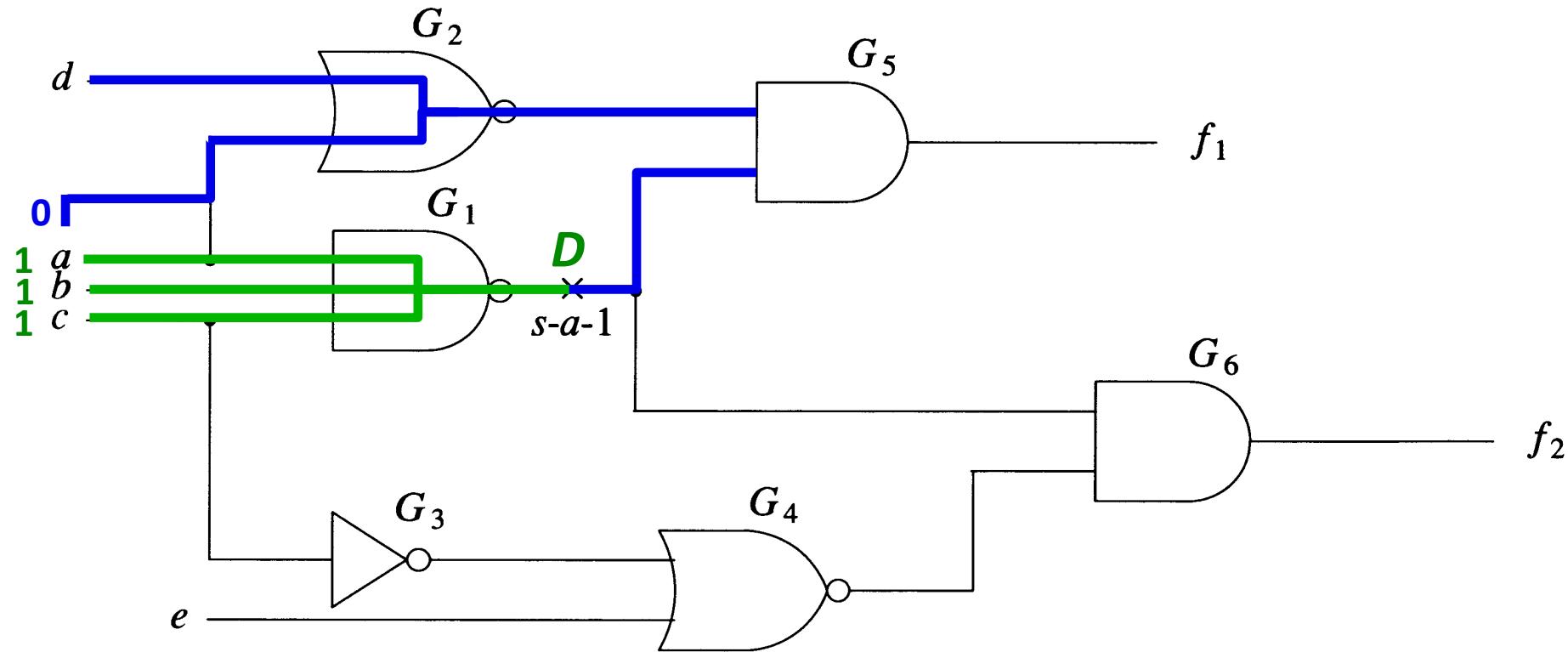
Fanout Free vs. Fanout

- 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

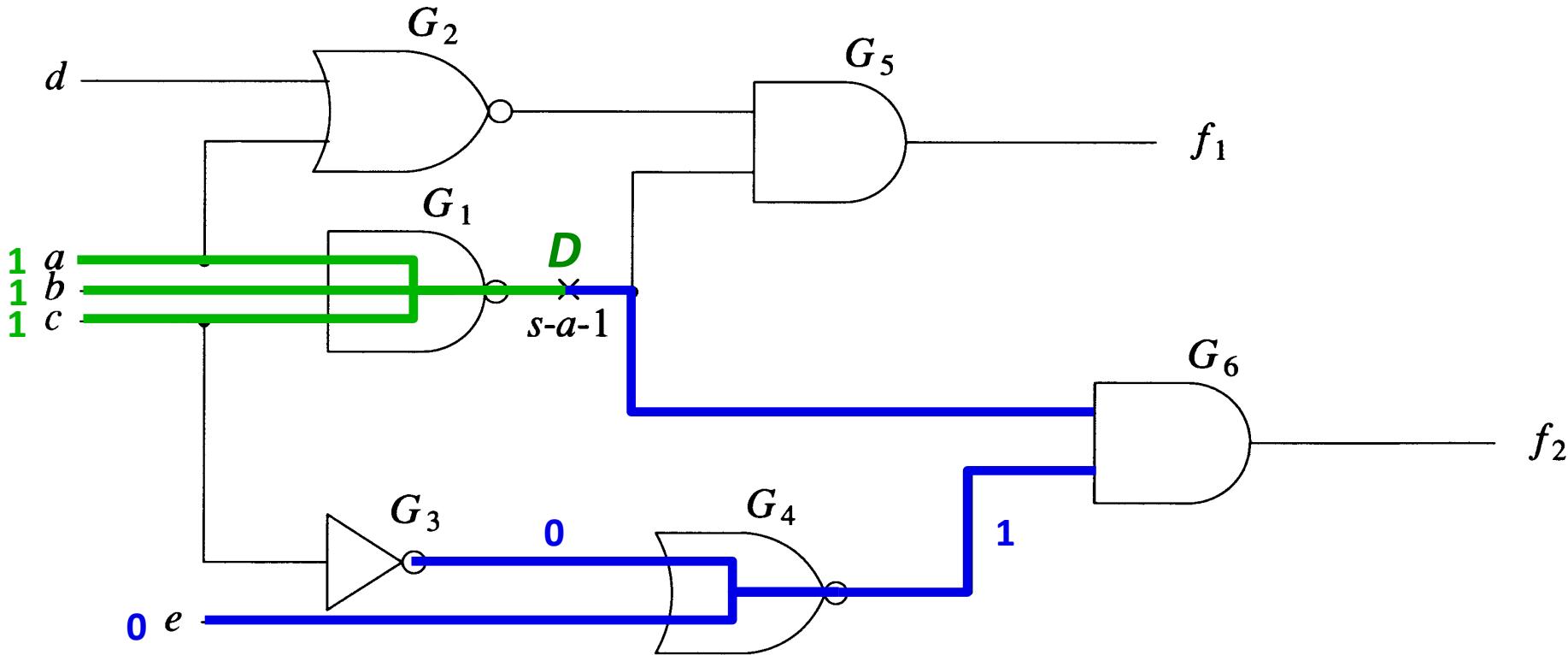
Example 6.2



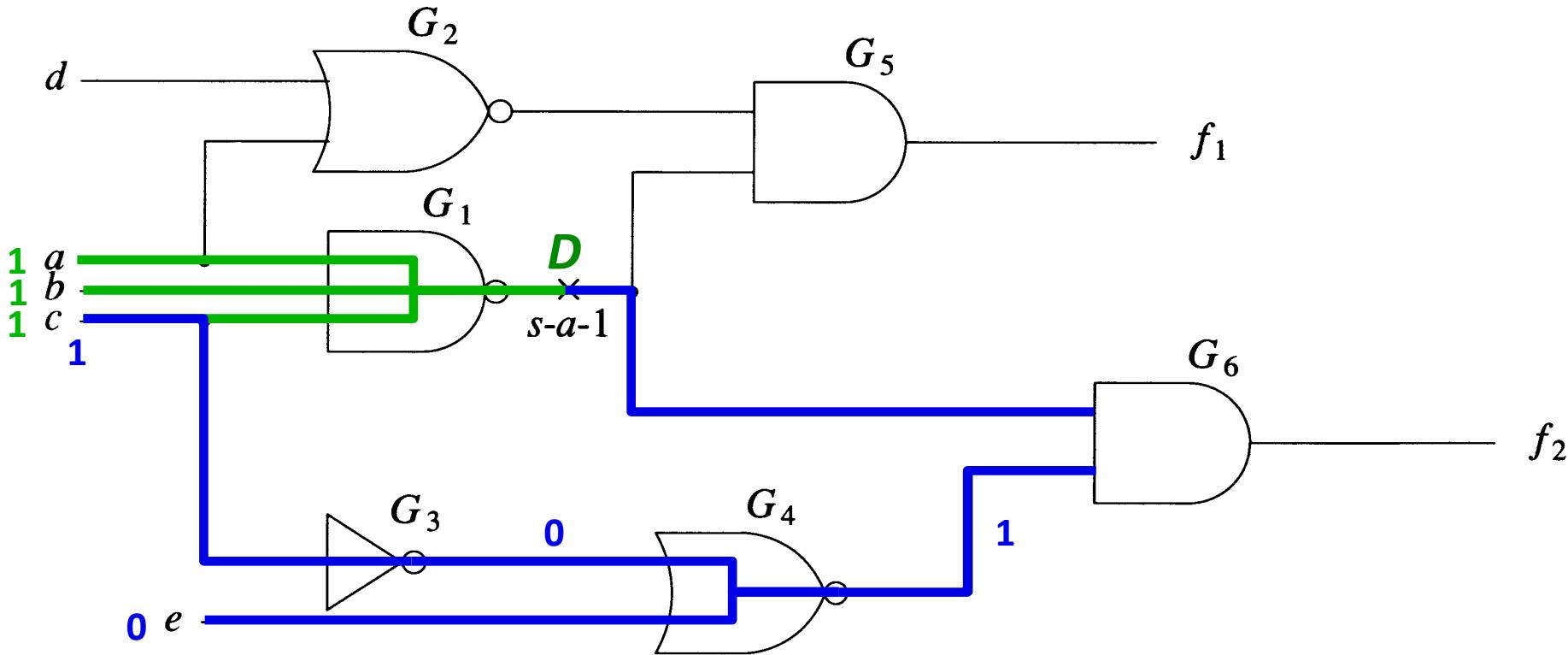
Example 6.2



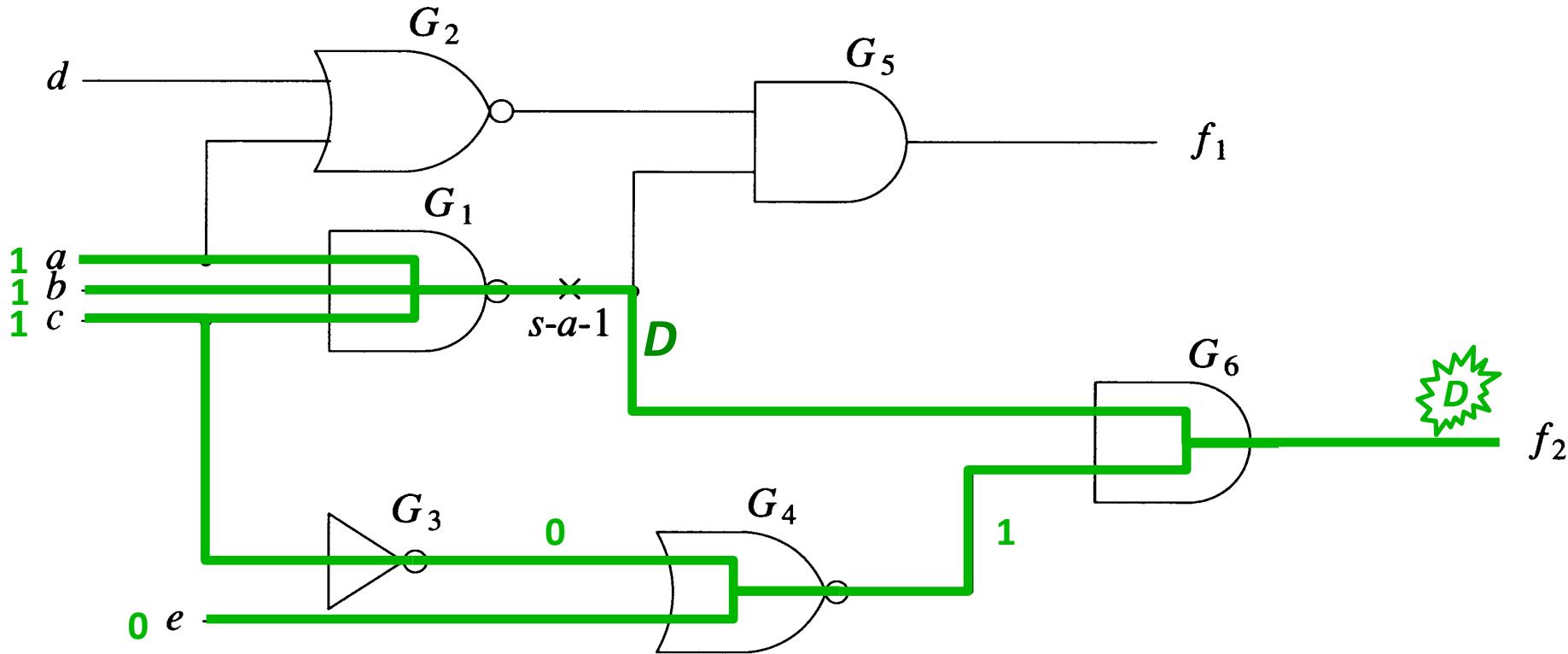
Example 6.2



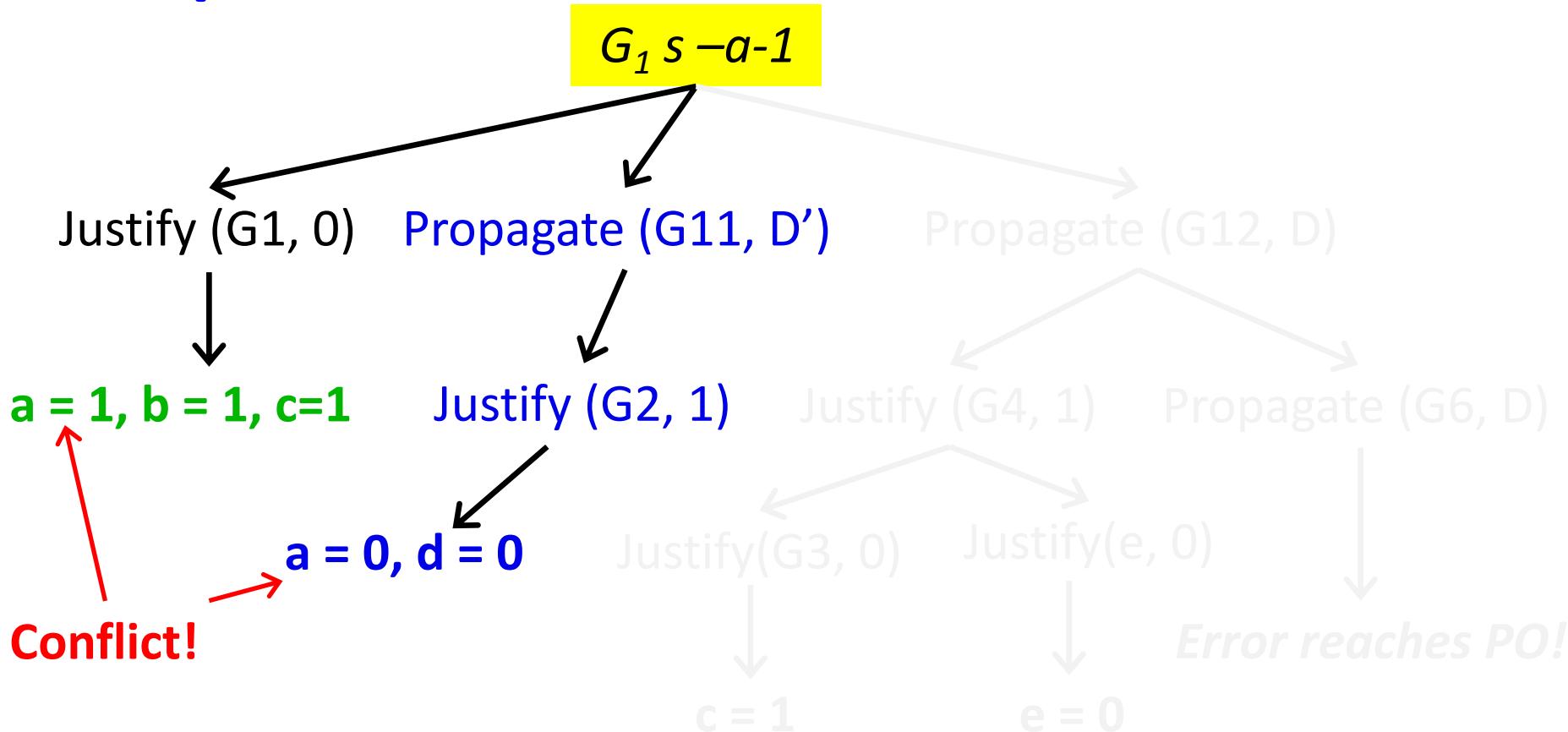
Example 6.2



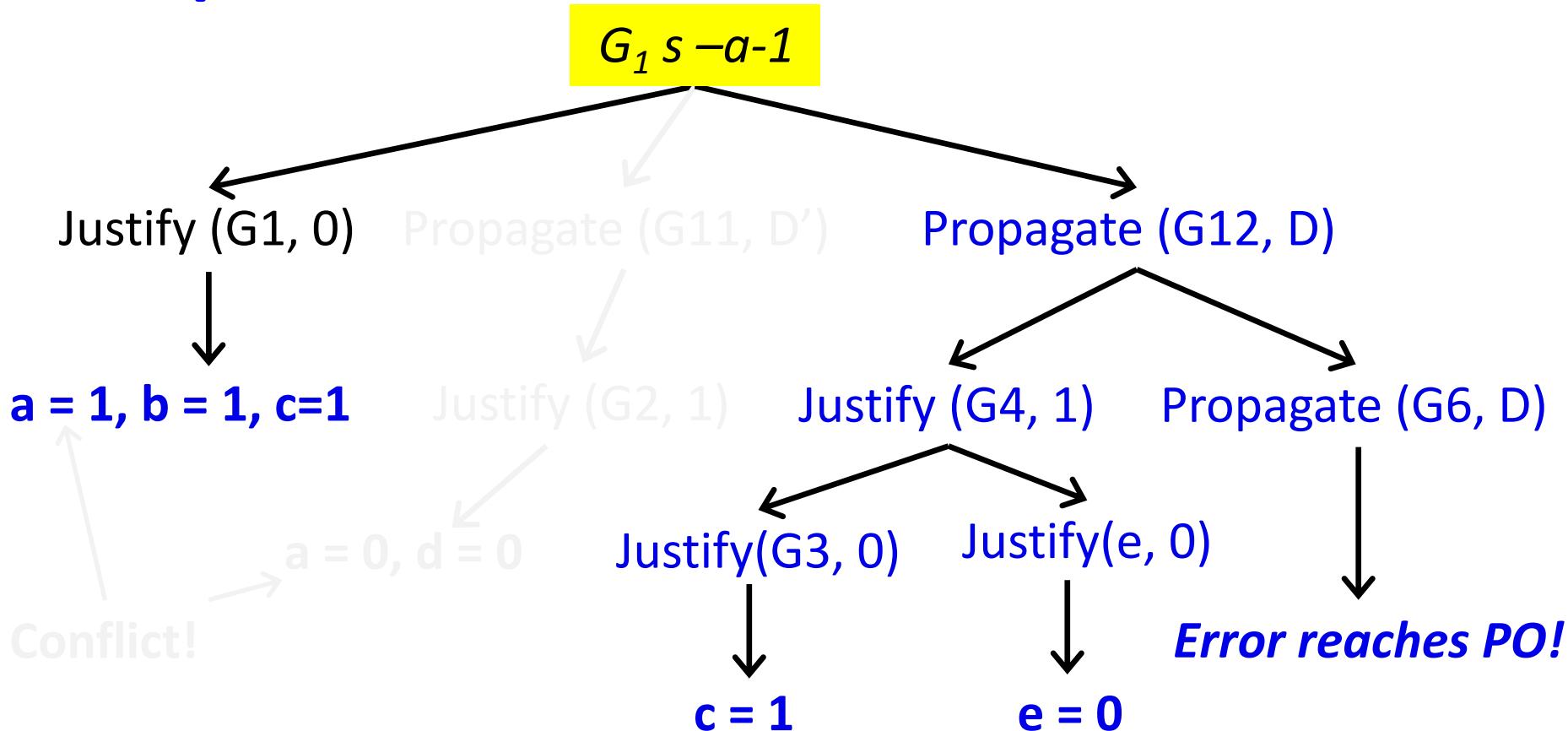
Example 6.2



Example 6.2



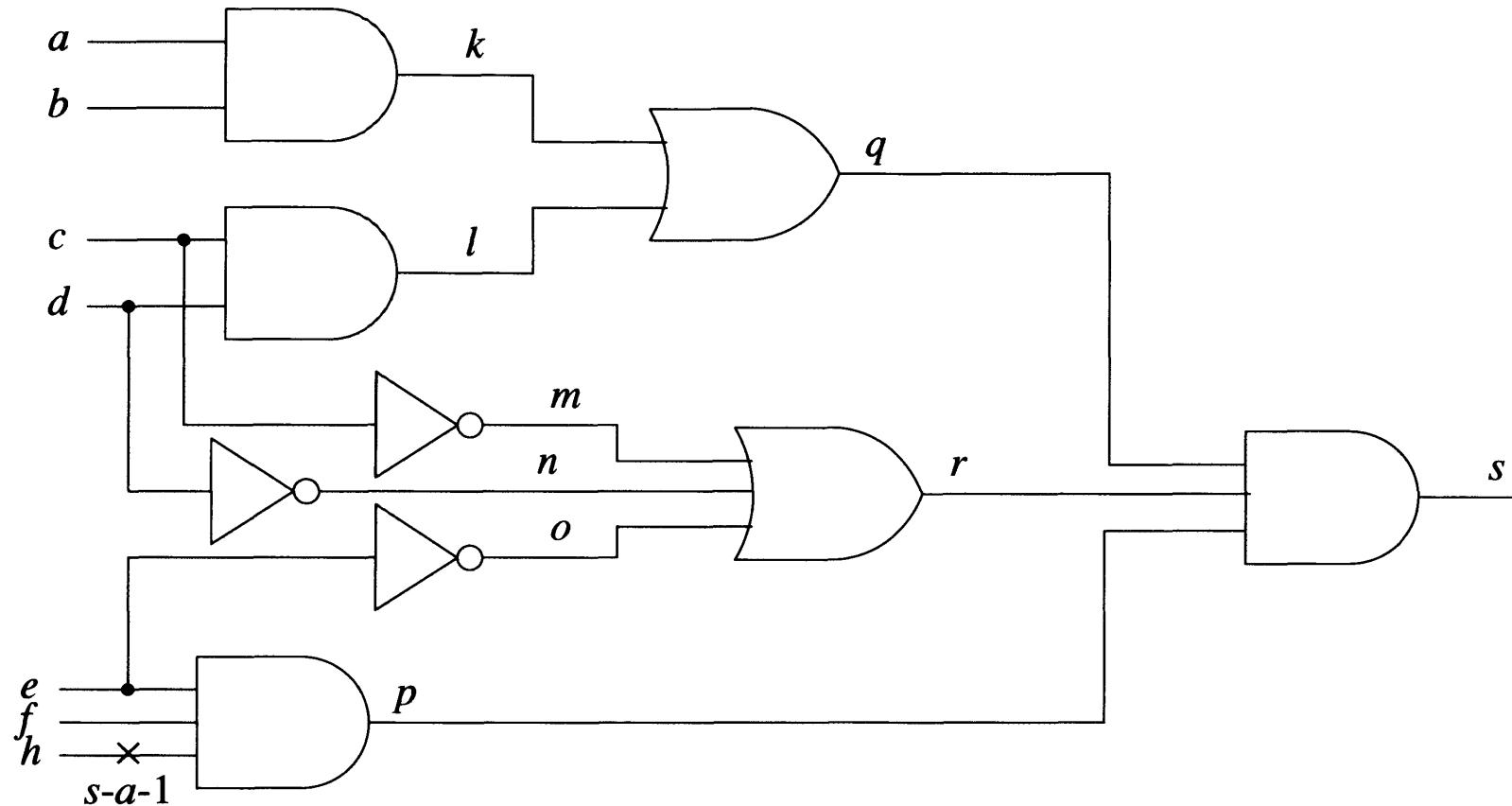
Example 6.2



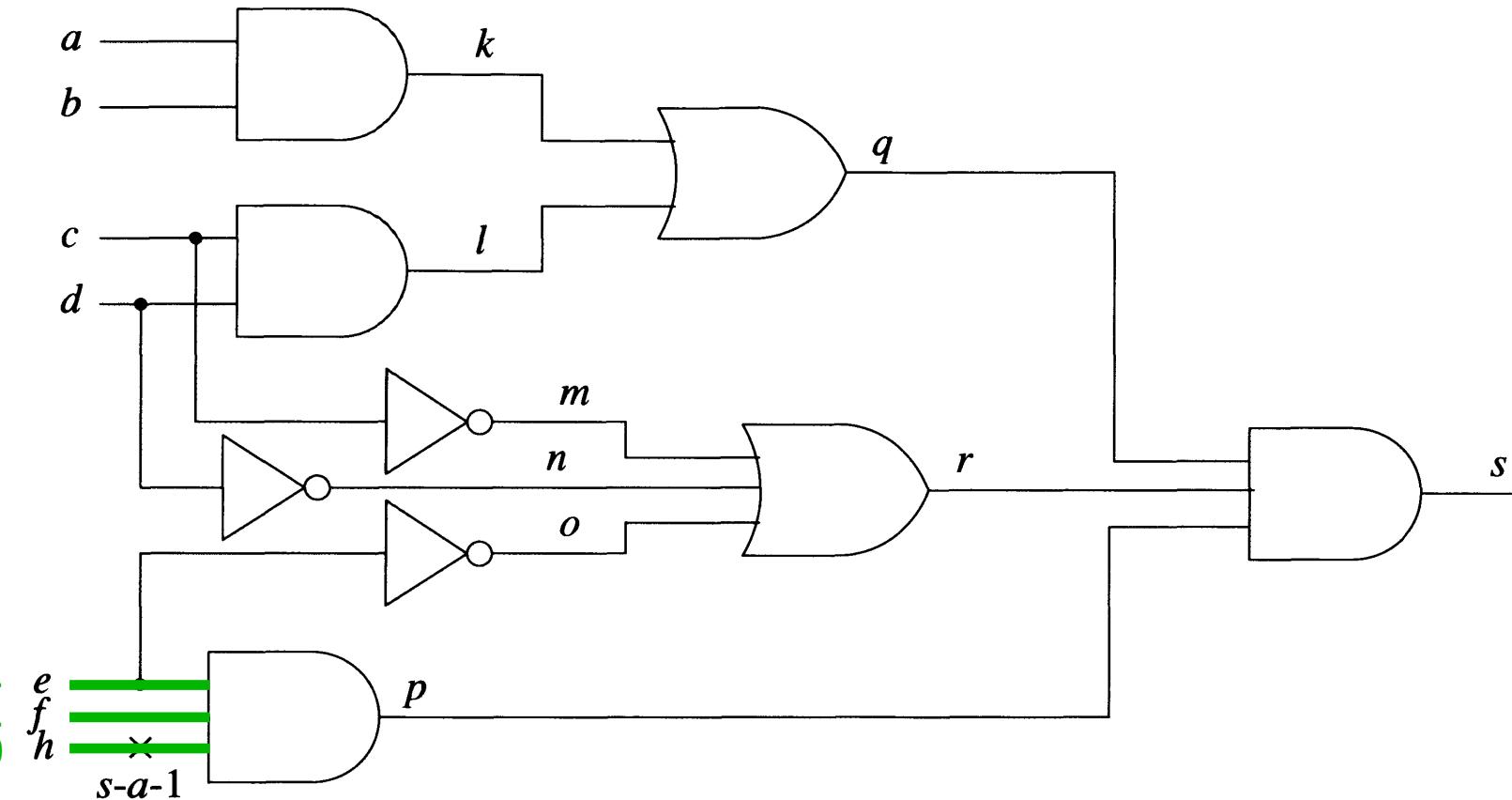
Backtracking Strategy

- Search for a test vector → 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

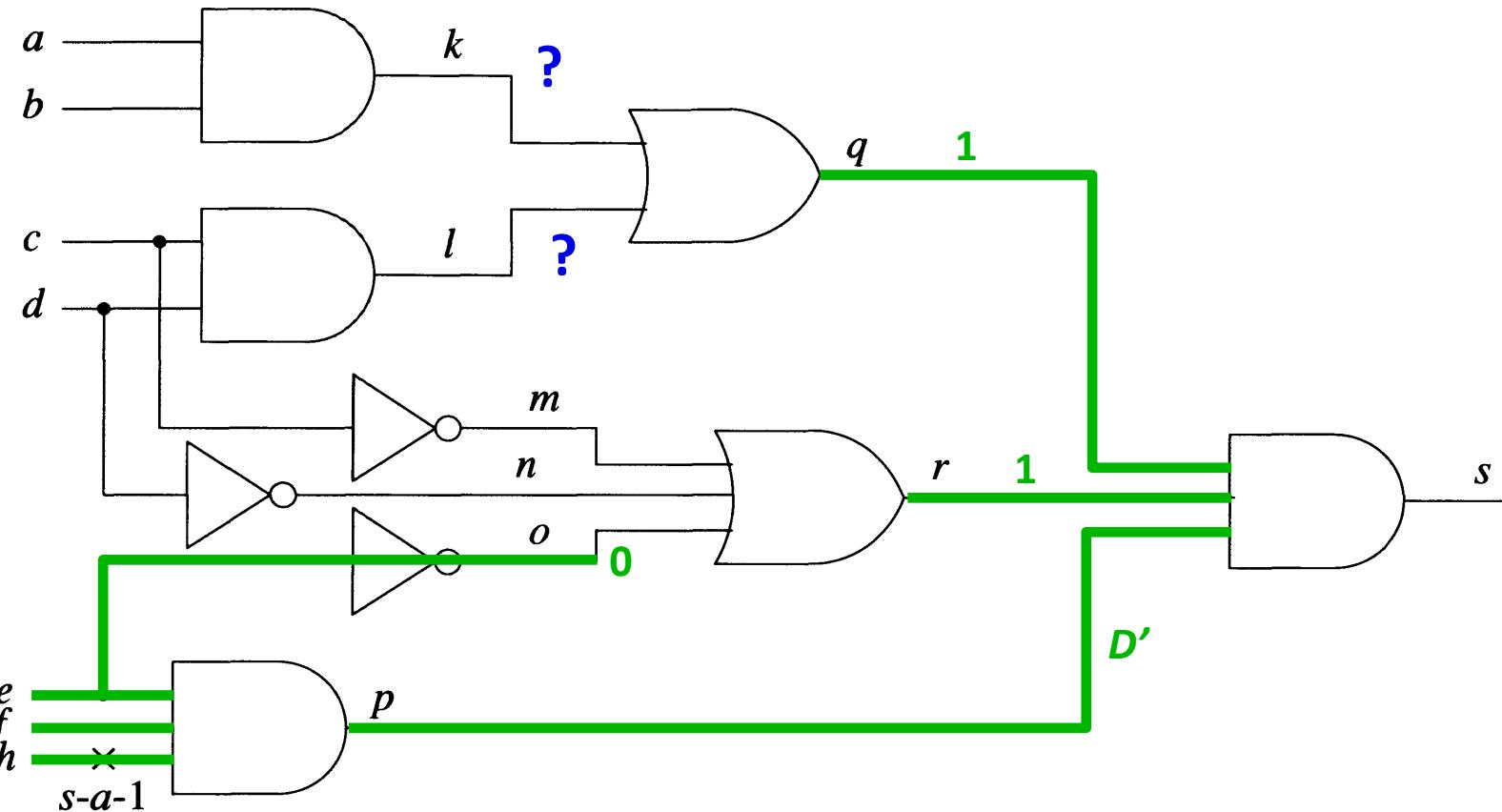
Example 6.3



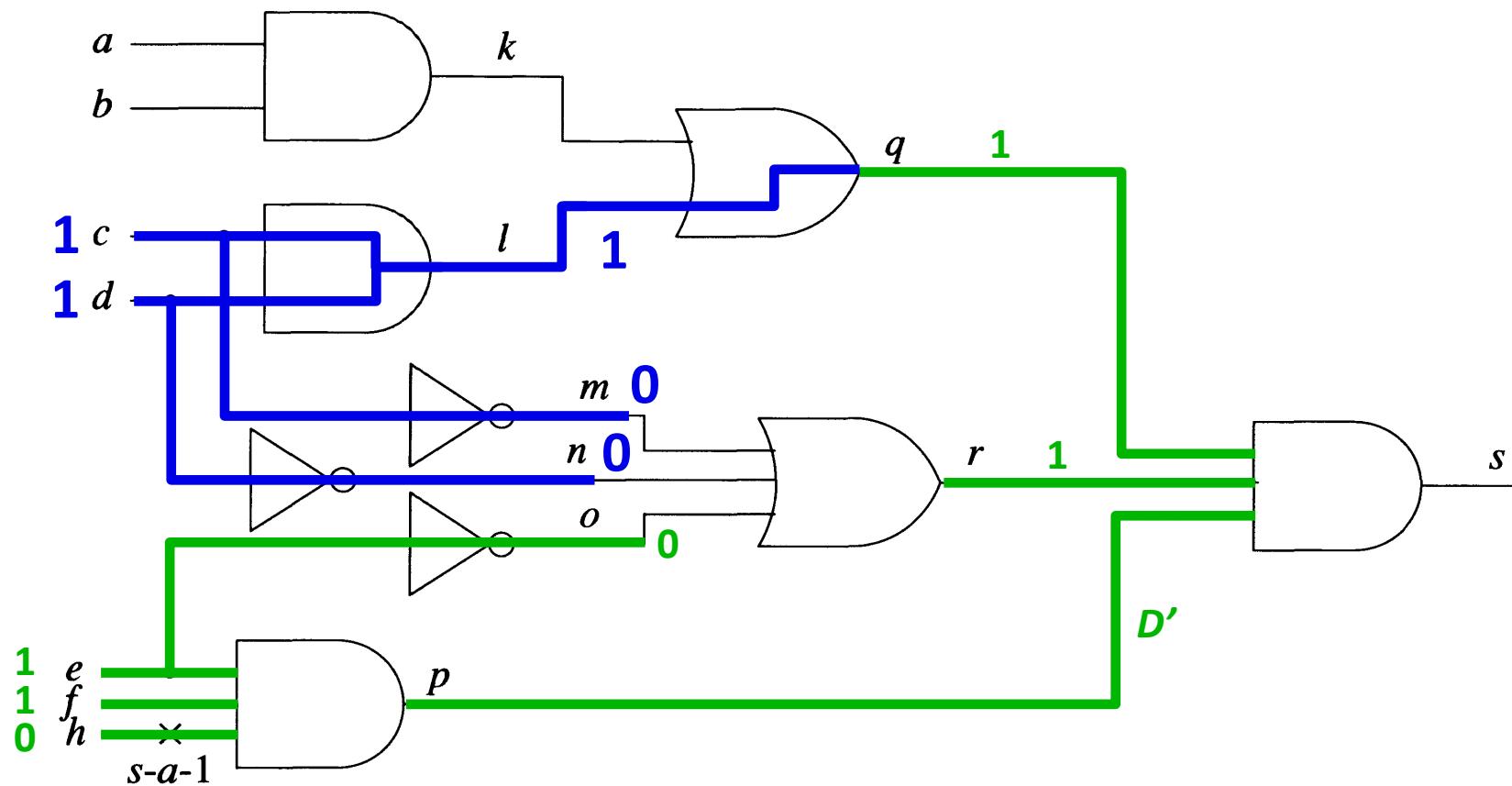
Example 6.3



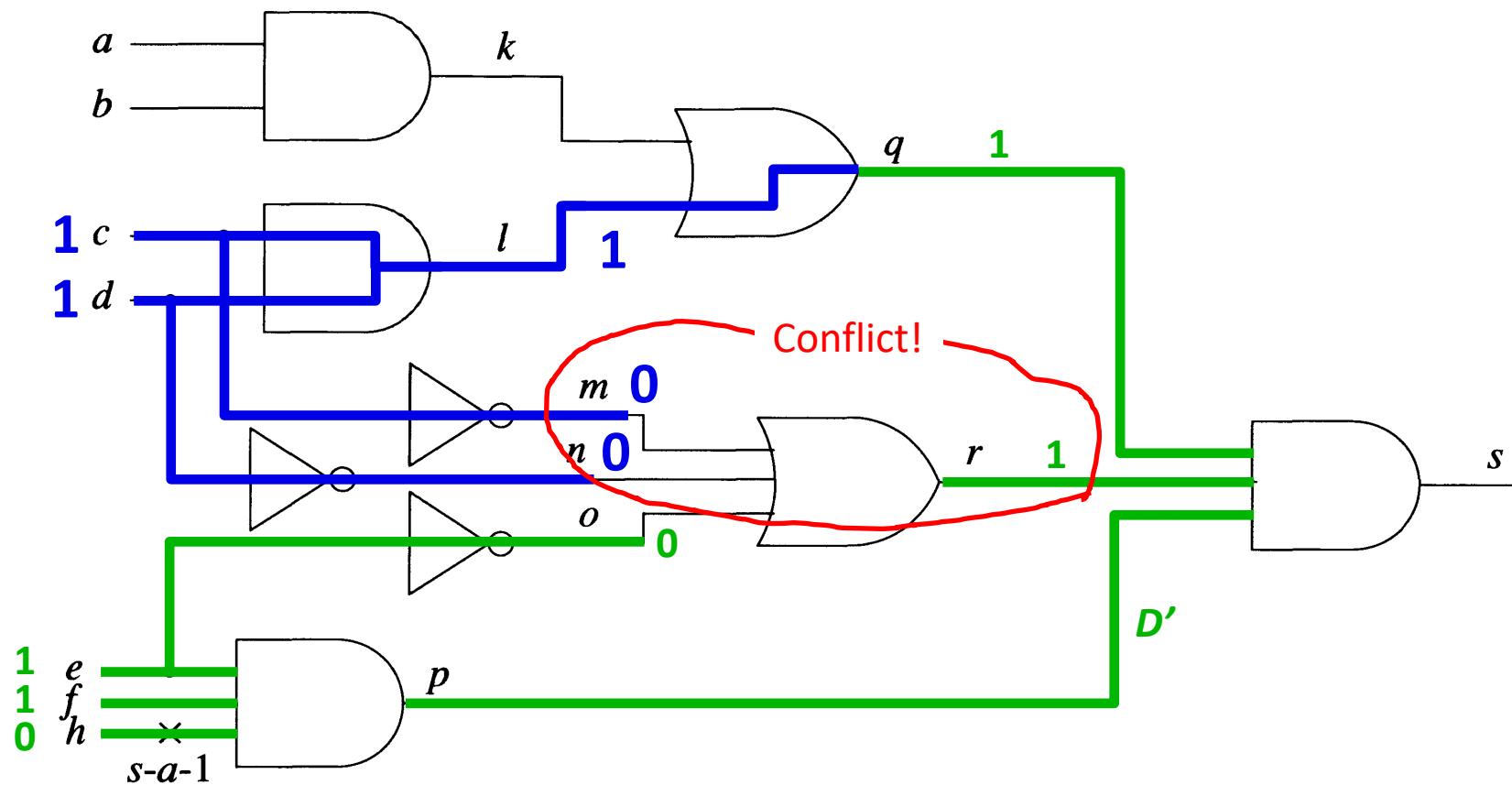
Example 6.3



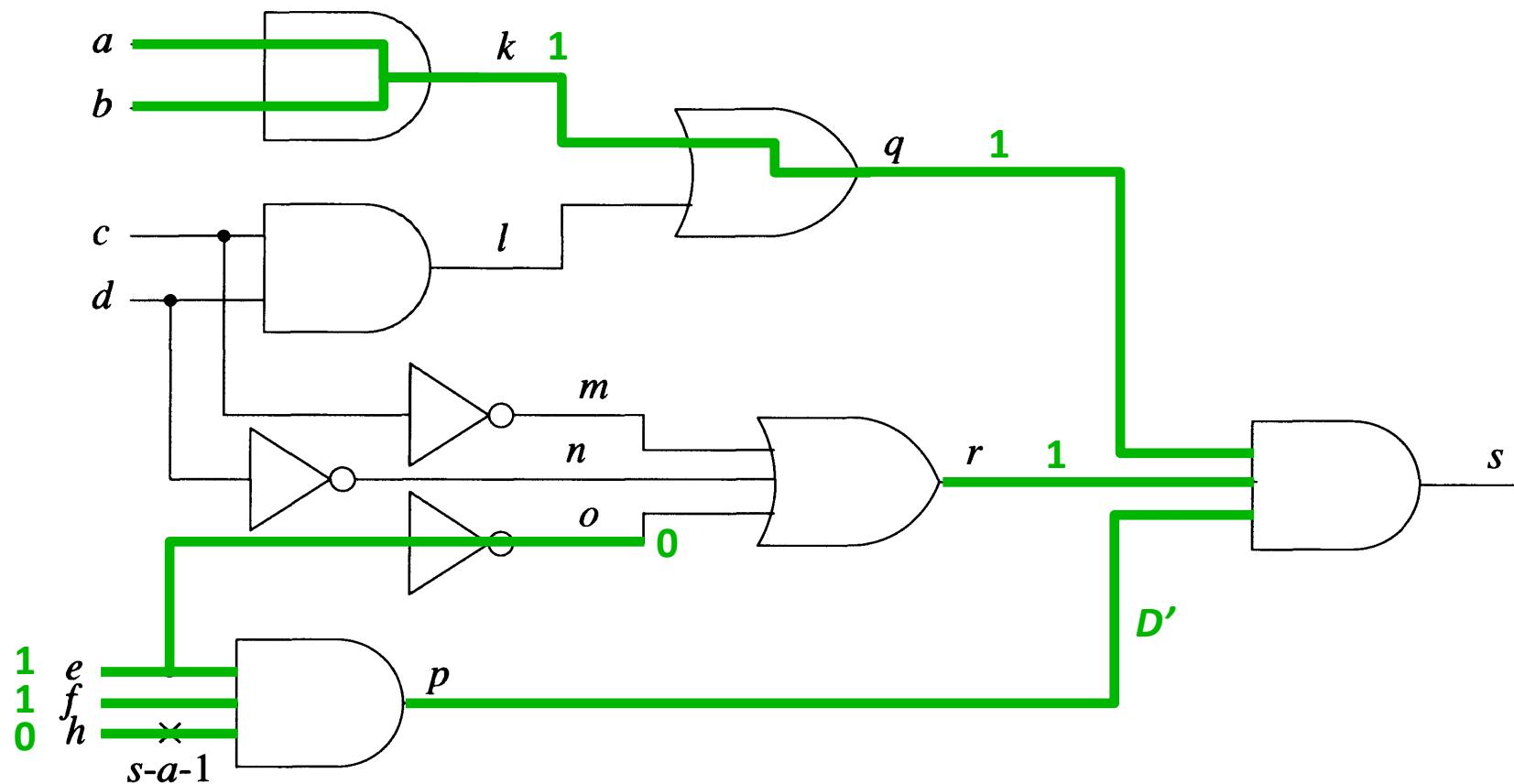
Example 6.3



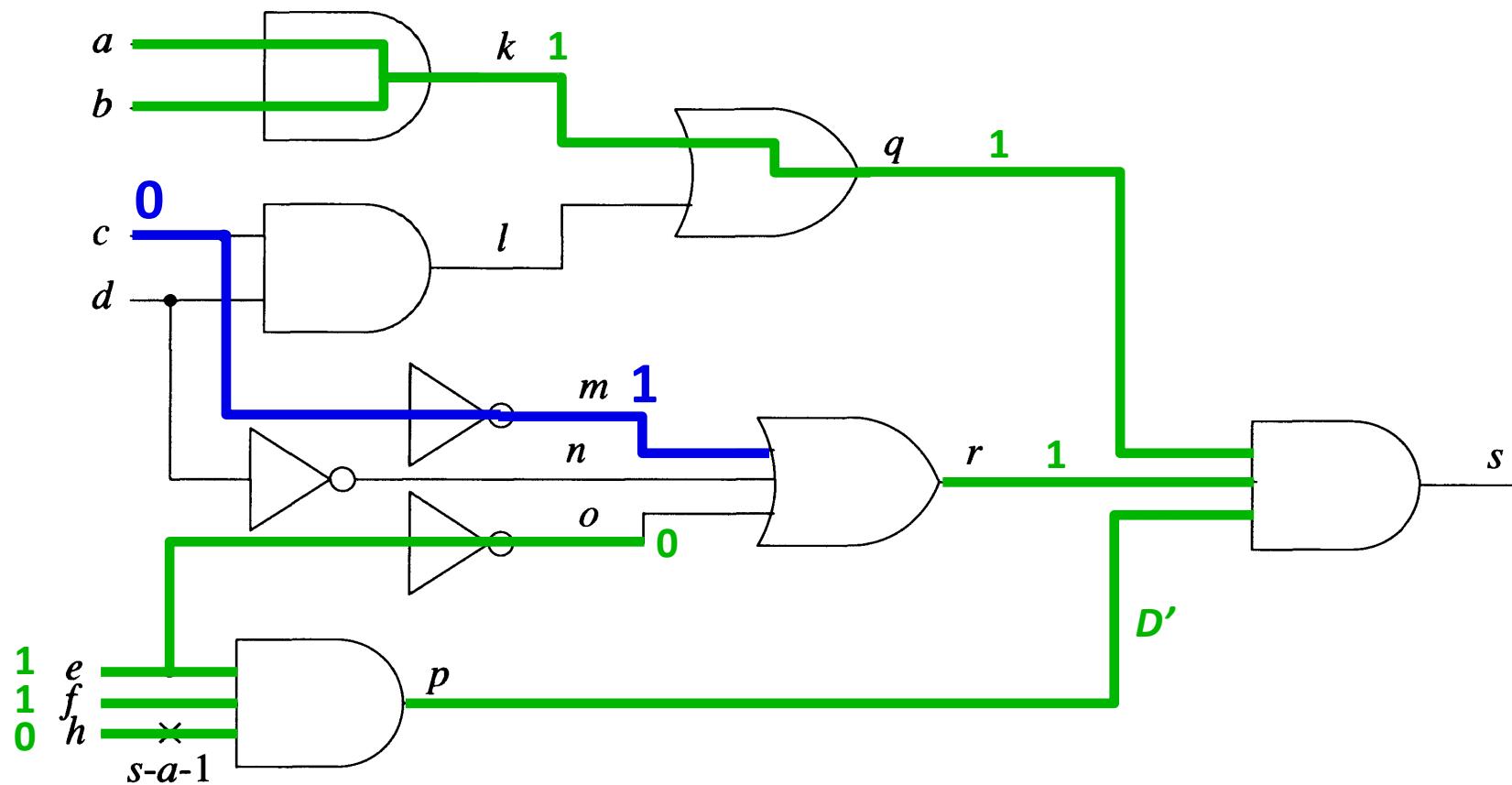
Example 6.3



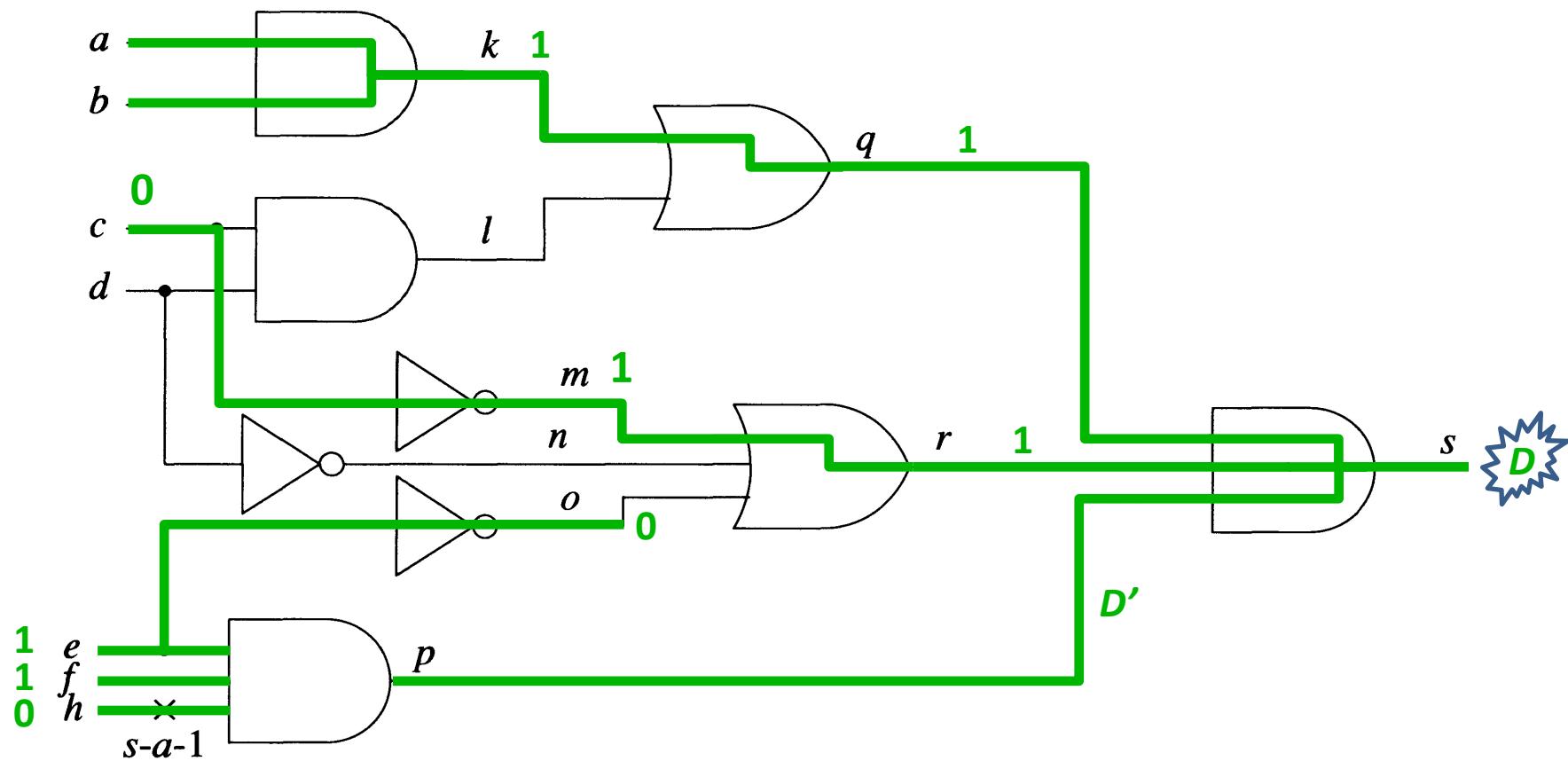
Example 6.3



Example 6.3



Example 6.3



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

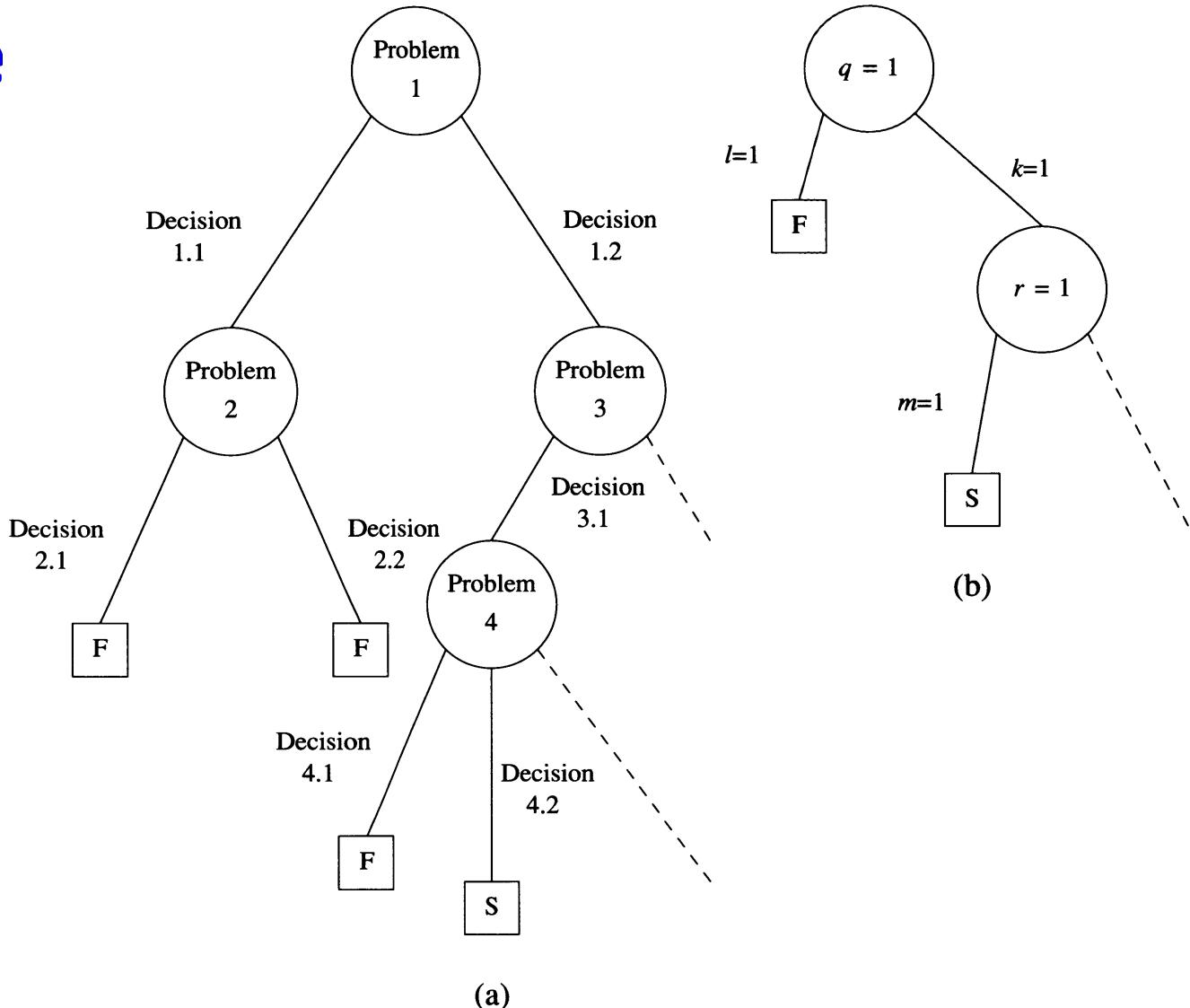
Implication: compute new values as a result of **decision**, and check inconsistencies.

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



TG Failure for an Undetectable Fault

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

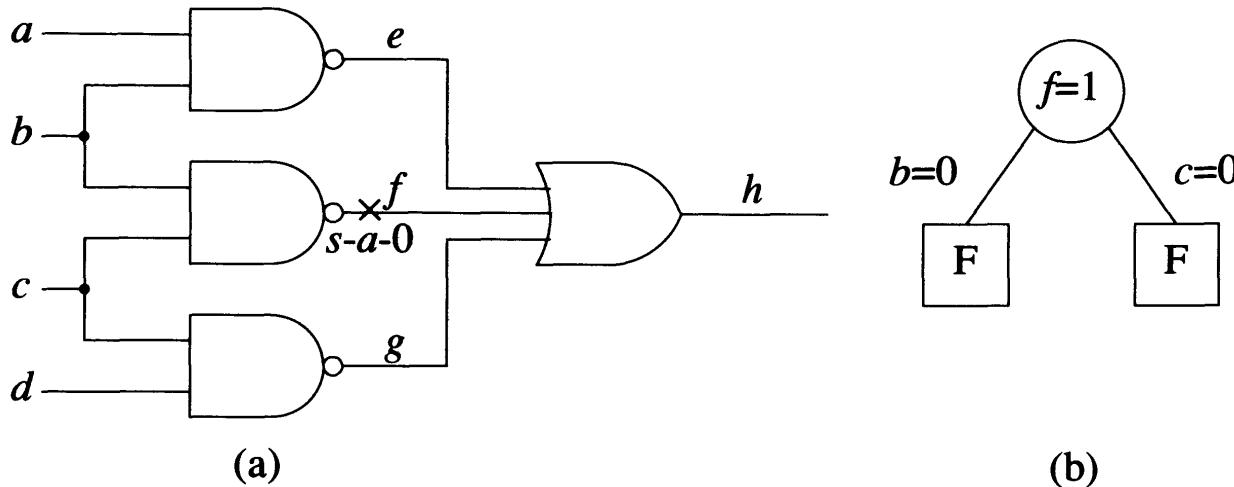


Figure 6.12 TG failure for an undetectable fault (a) Circuit (b) Decision tree

D-Frontier

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

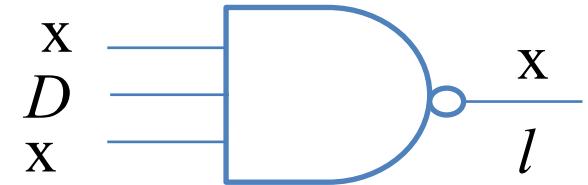
→ **D-drive** operation –

Pick a gate and try to propagate error

→ If **D-frontier** becomes empty

⇒ No error can be propagated to PO

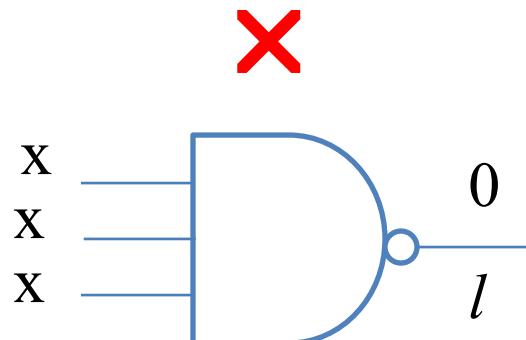
⇒ Backtracking should occur



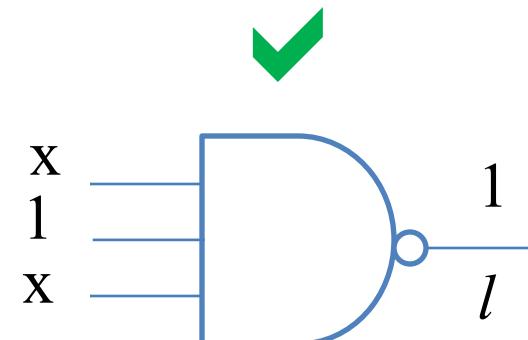
Gates in D-frontier indicate necessary decisions in order to proceed.

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

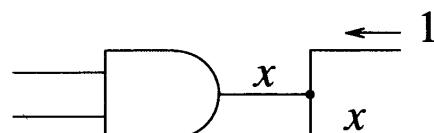
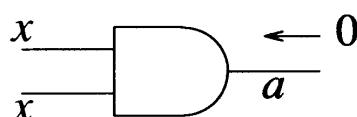
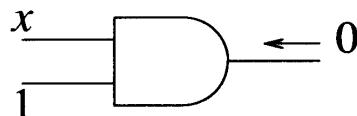
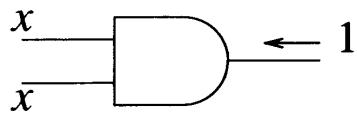
3 Steps:

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*

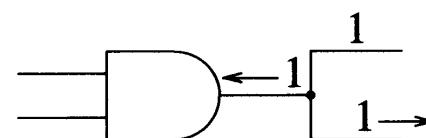
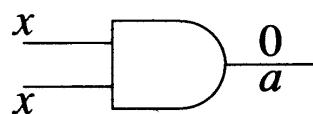
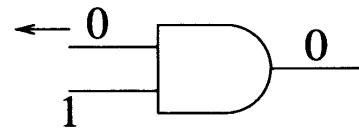
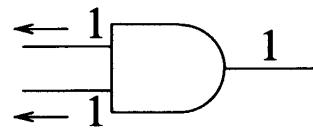
Implication can be forward or backward.

Backward Implication Propagation

Before



After



Gate a gets added to J -
frontier
after $a = 0$

(a)

(b)

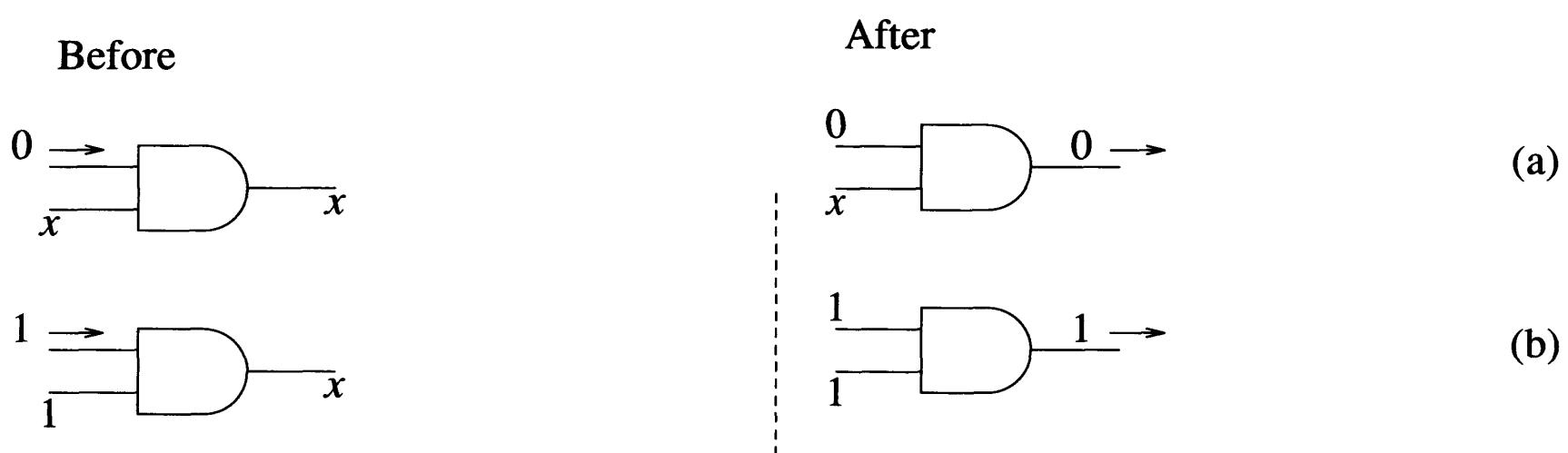
(c)

J -frontier = { ..., a }

(d)

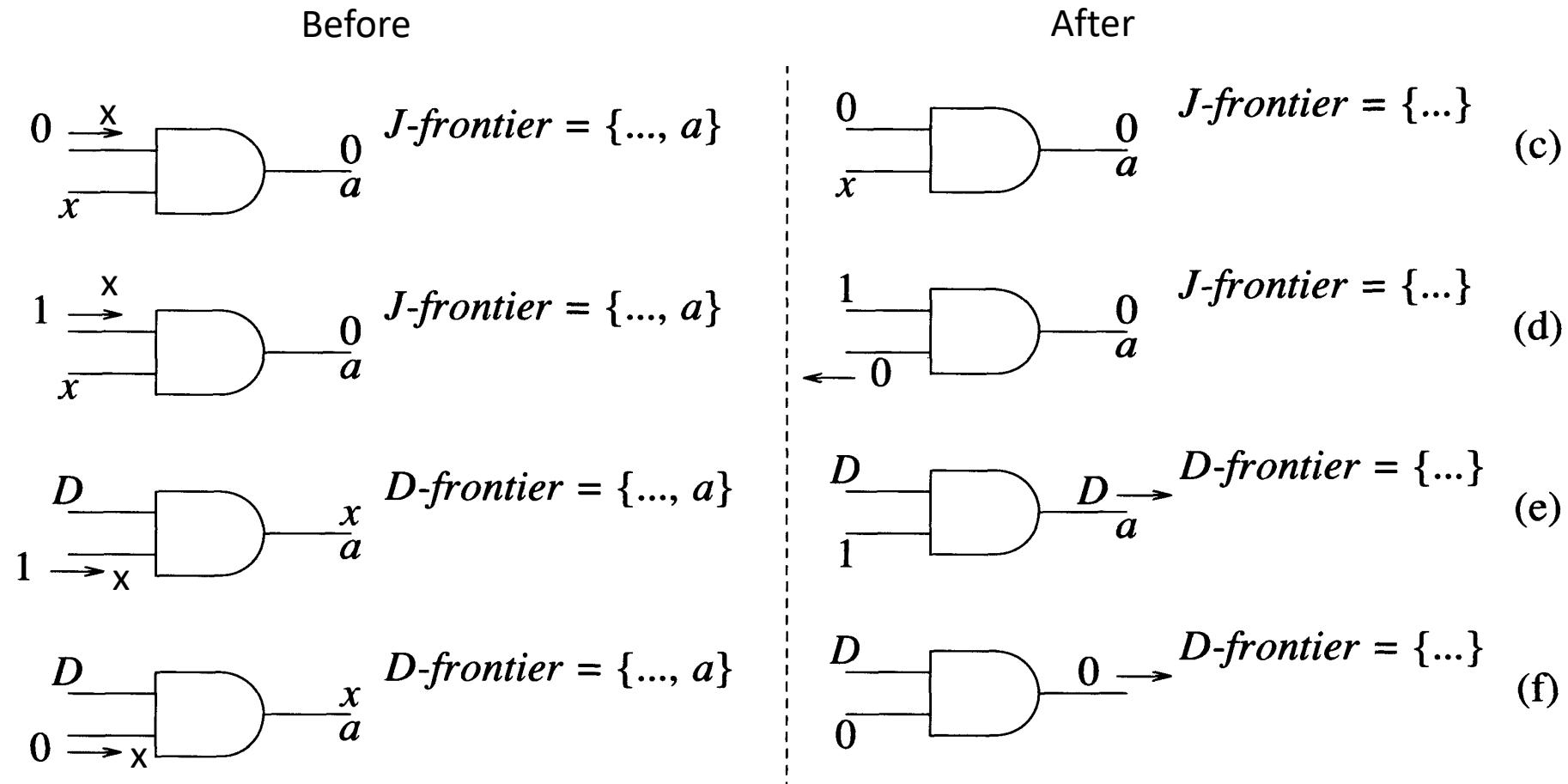
Forward Implication Propagation

Figure 6.15



Forward Implication Propagation

Figure 6.15



Forward Implication Propagation – cont'd

Figure 6.16

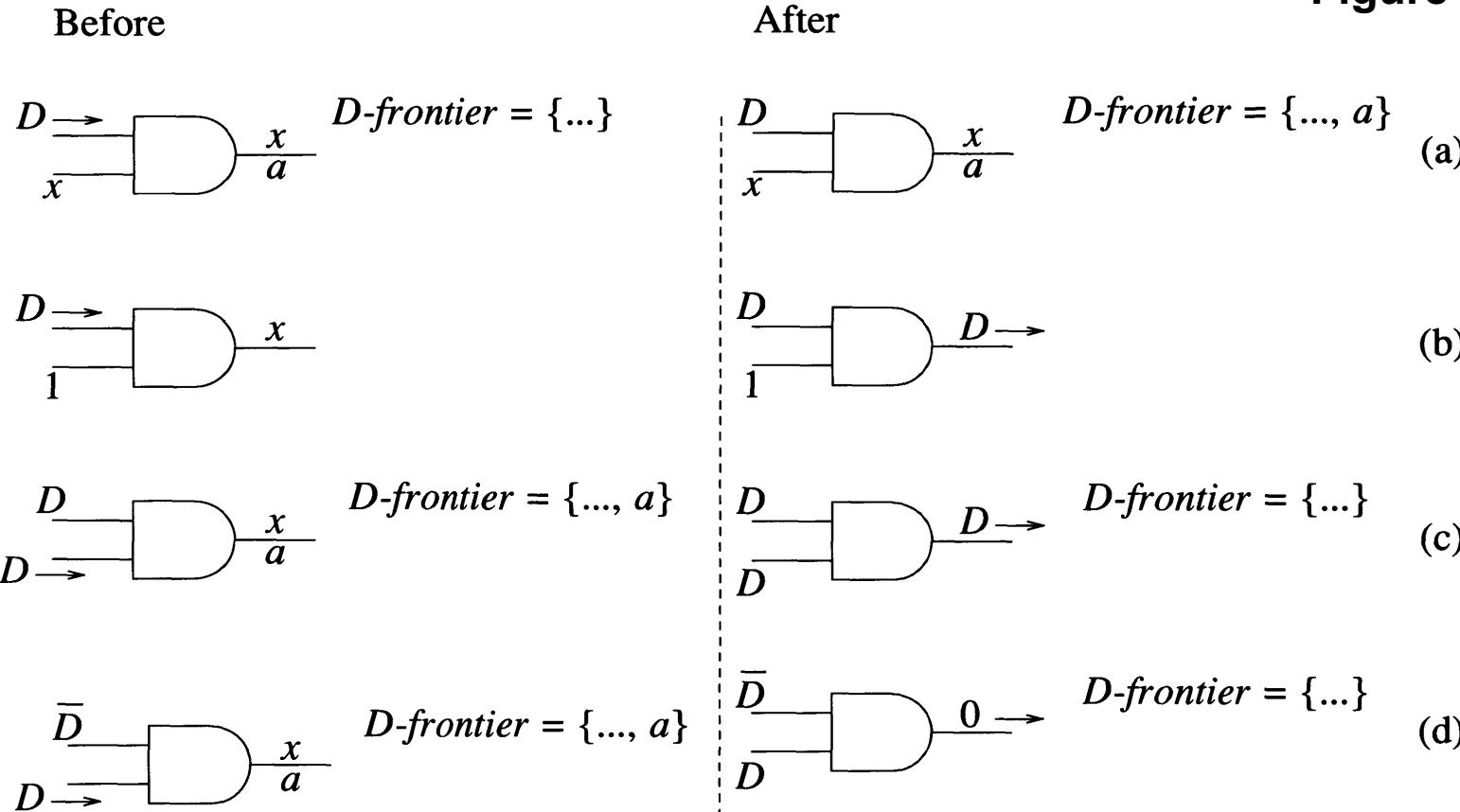
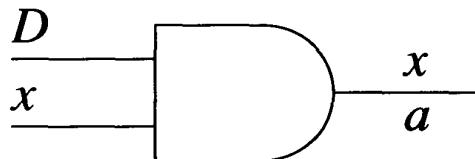


Figure 6.17 Unique D-Drive

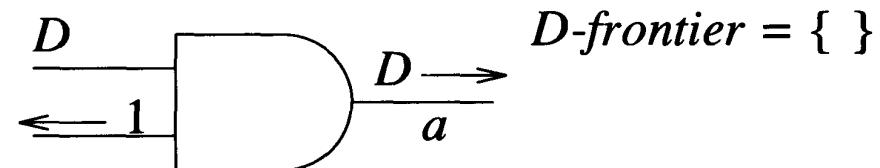
- When only gate remains in the D -frontier.
- There is only one way to propagate D .

Before



$$D\text{-frontier} = \{a\}$$

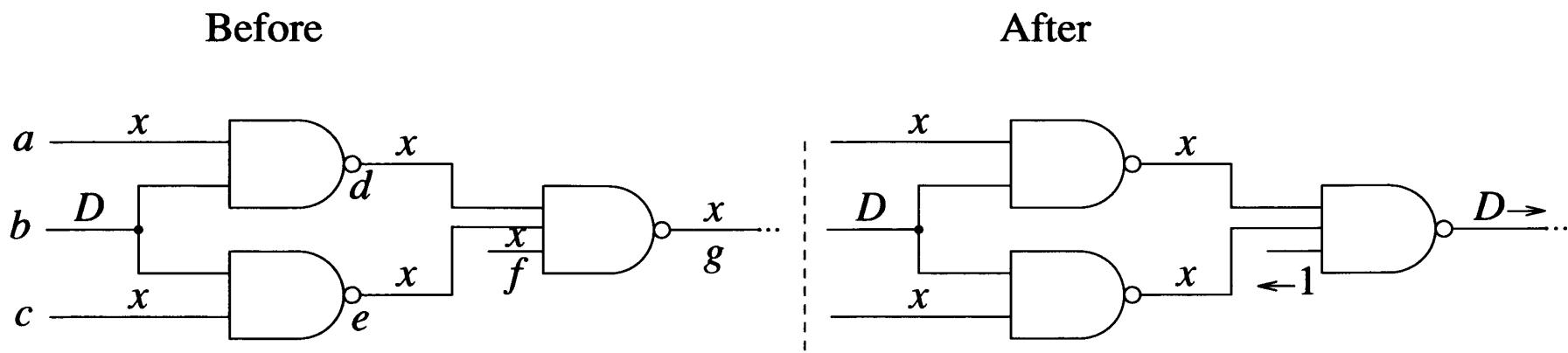
After



$$D\text{-frontier} = \{ \}$$

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!

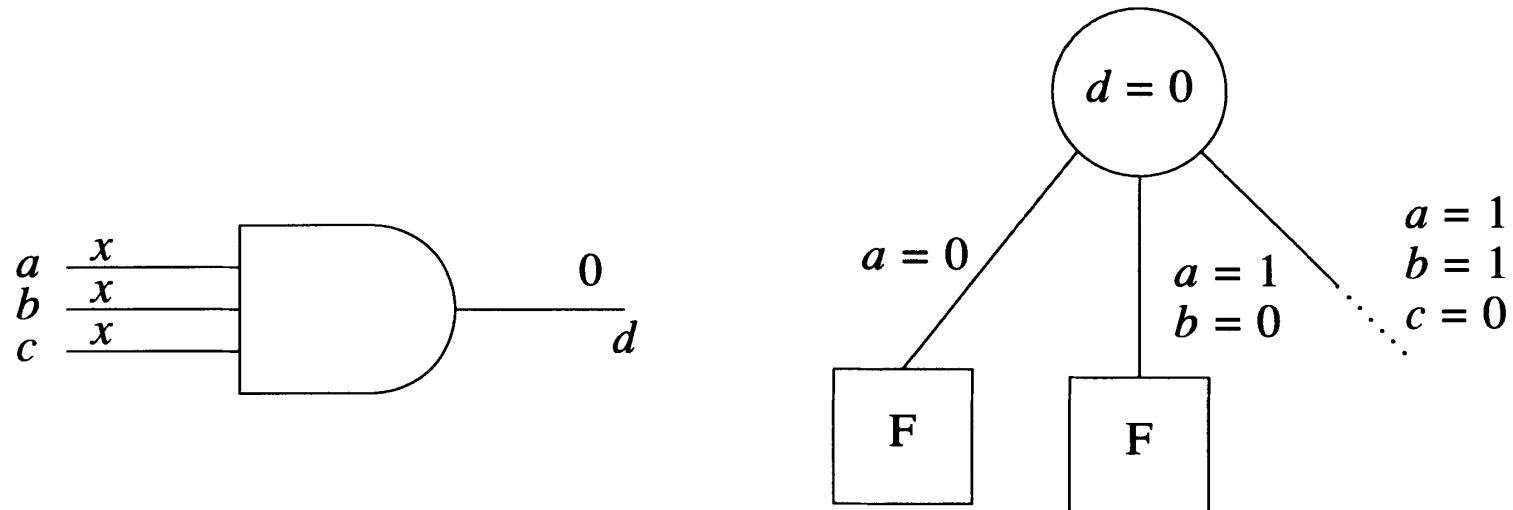
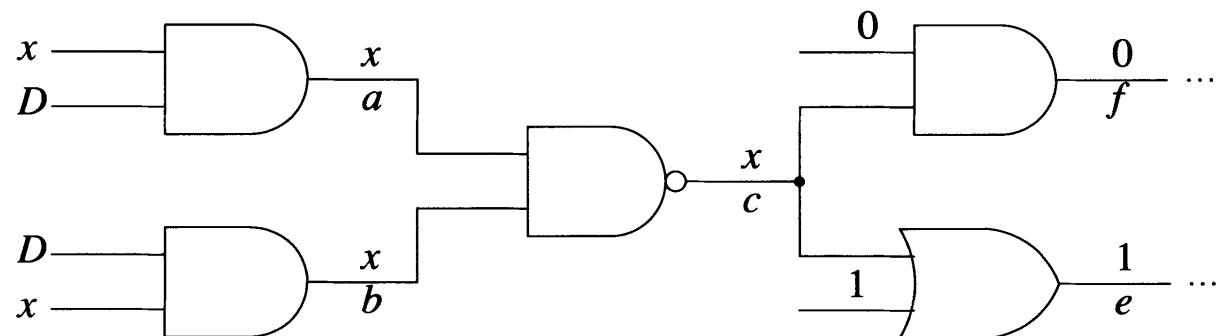


Figure 6.21 Reversing incorrect decisions

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.



D-Algorithm

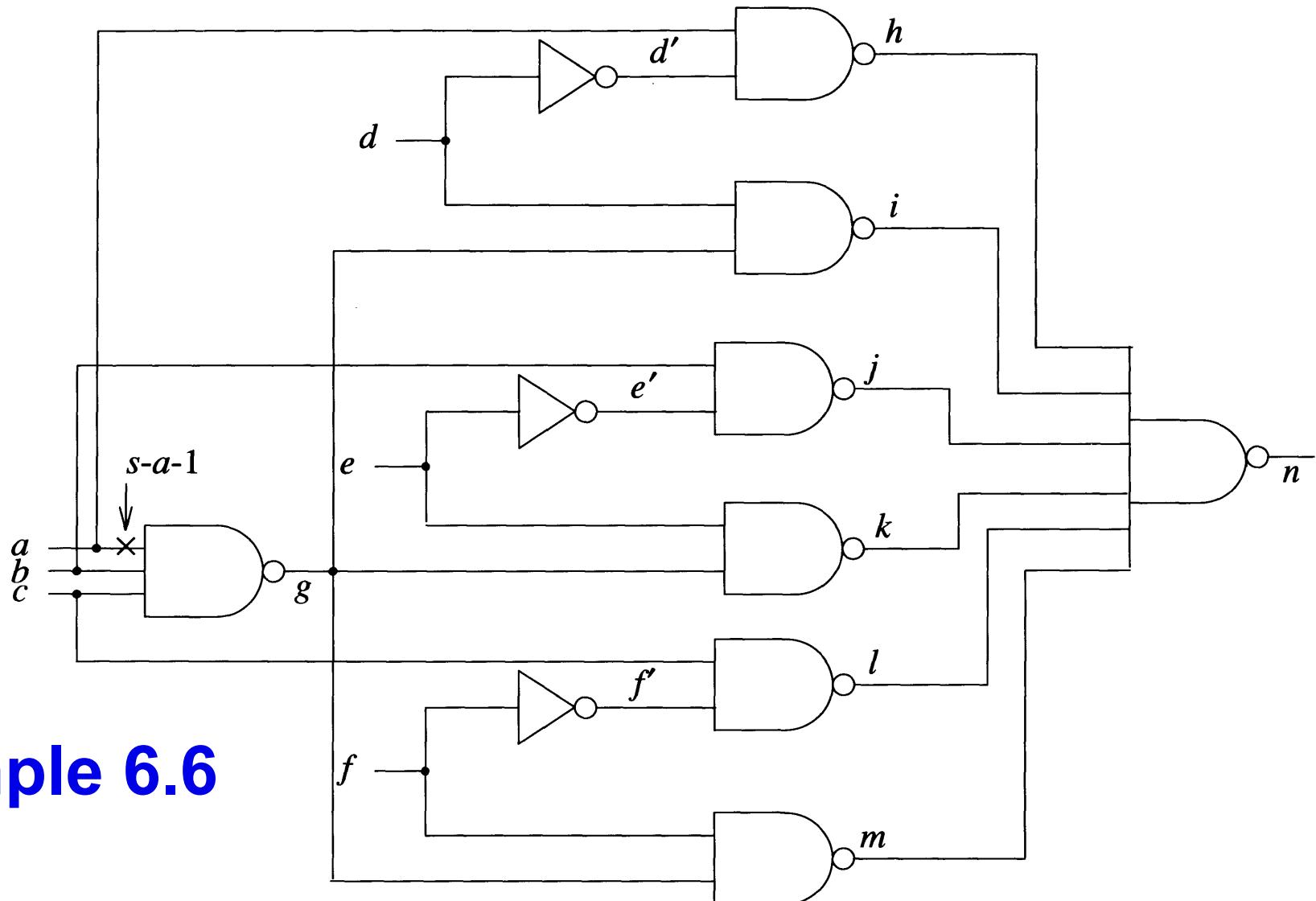
- Ability to propagate errors on several reconvergent fanouts
- We assume error propagation is given priority over justification problems (simplifying assumption)
- “assign” means “add the value to the assignment queue”
- *Implify_and_check()* handles the assignments

```

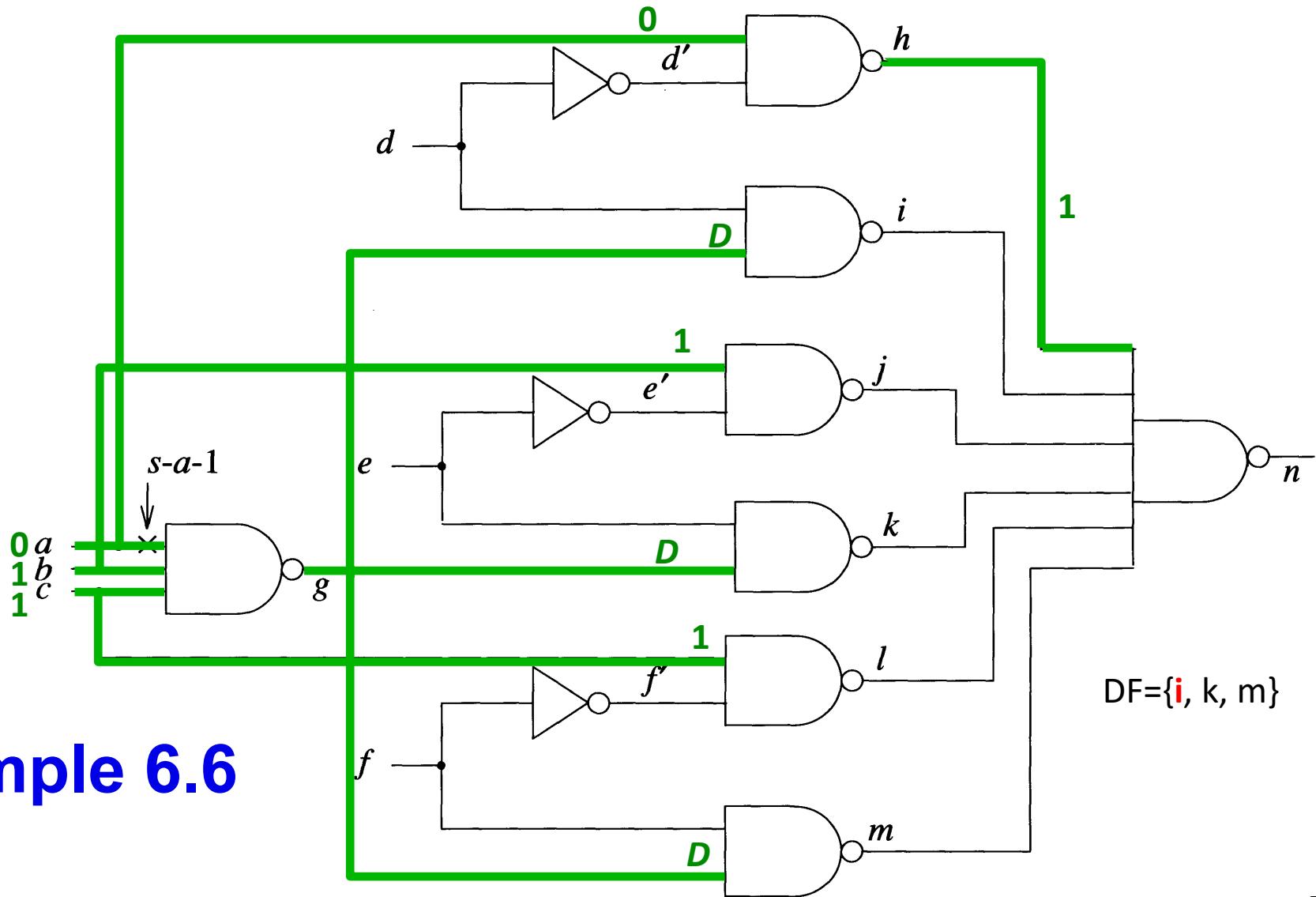
1.  D-alg()
2.  begin
3.      if Impl_y_and_check() = FAIL then return FAIL
4.      if (error not at PO) then /* error propagation */
5.          begin
6.              if D-frontier =  $\emptyset$  then return FAIL
7.              repeat
8.                  begin
9.                      select an untried gate (G) from D-frontier
10.                     c = controlling value of G
11.                     assign c' to every input of G with input x
12.                     if D-alg() = SUCCESS then return SUCCESS
13.                 end
14.             until all gates from D-frontier have been tried
15.             return FAIL
16.         end

```

```
17. /* error propagated to a PO */  
18. if  $J$ -frontier =  $\emptyset$  then return SUCCESS  
19. select a gate (G) from the  $J$ -frontier  
20.  $c$  = controlling value of G  
21. repeat  
22.   begin  
23.     select an input ( $j$ ) of G with value x  
24.     assign  $c$  to  $j$   
25.     if  $D$ -alg() = SUCCESS then return SUCCESS  
26.     assign  $c'$  to  $j$  /* reverse decision */  
27.   end  
28. until all inputs of G are specified  
29. return FAIL  
30. end
```



Example 6.6



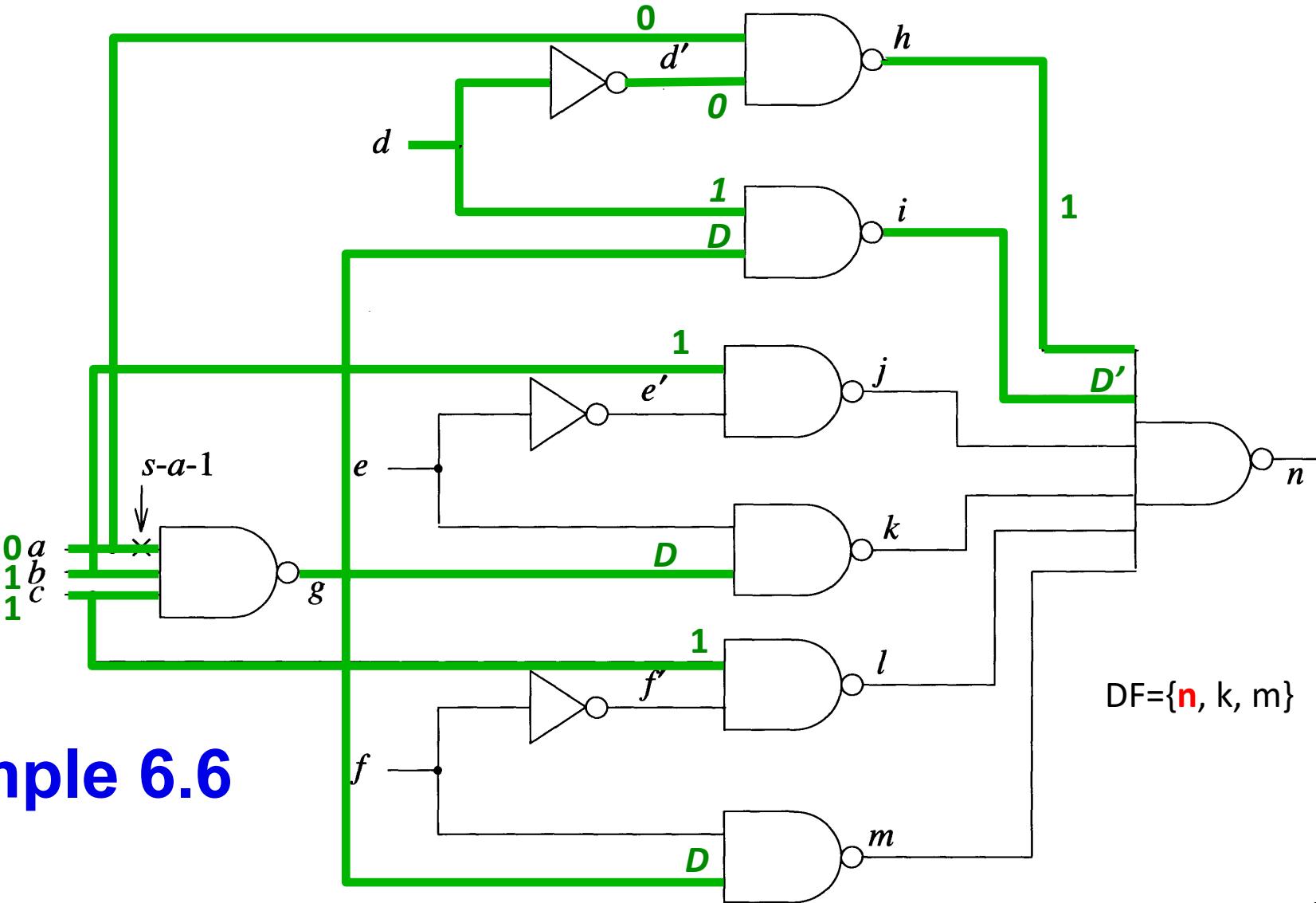
Example 6.6

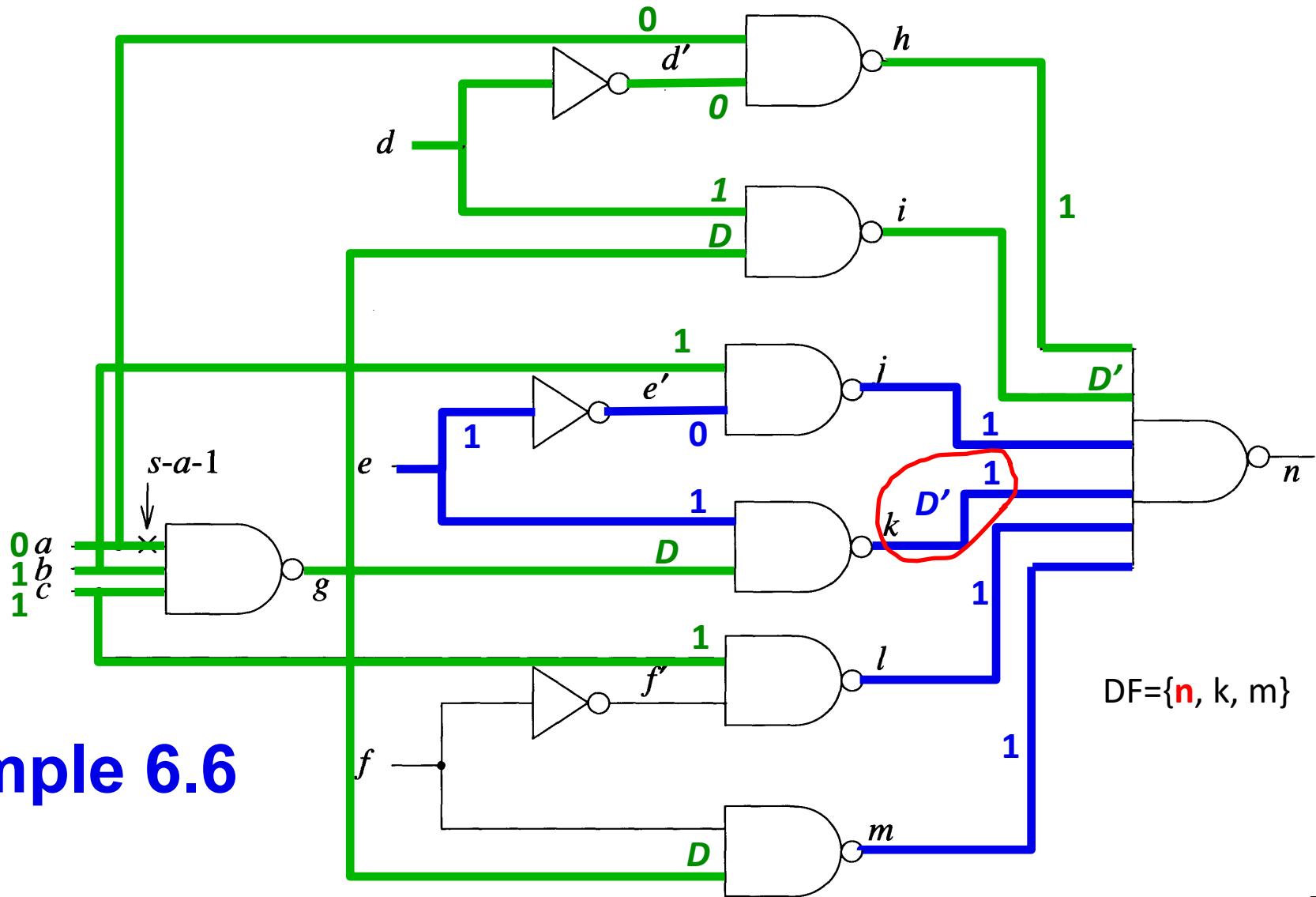
0 *a*
1 *b*
1 *c*

3

10

s-a-1





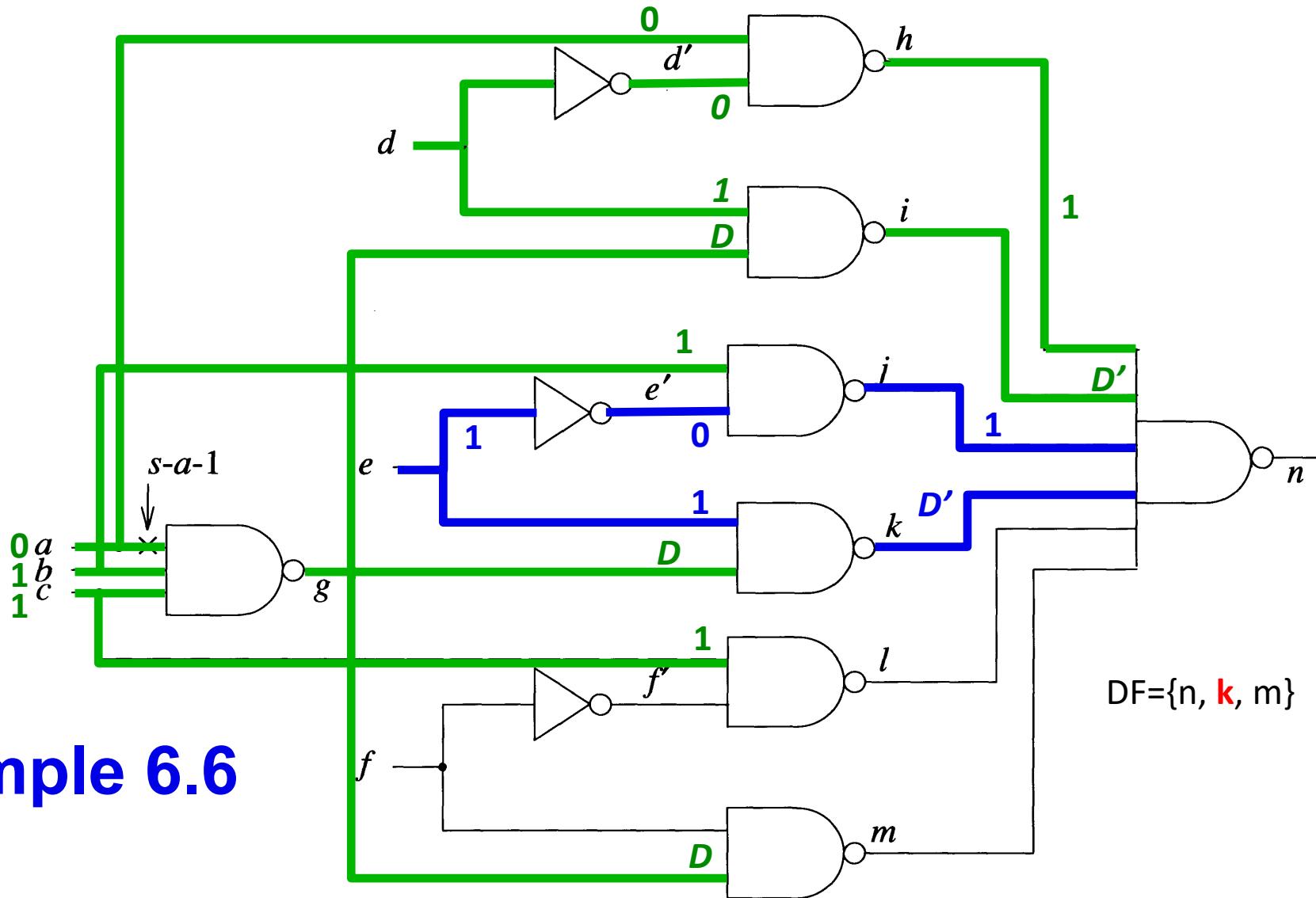
Example 6.6

0 *a*
1 *b*
1 *c*

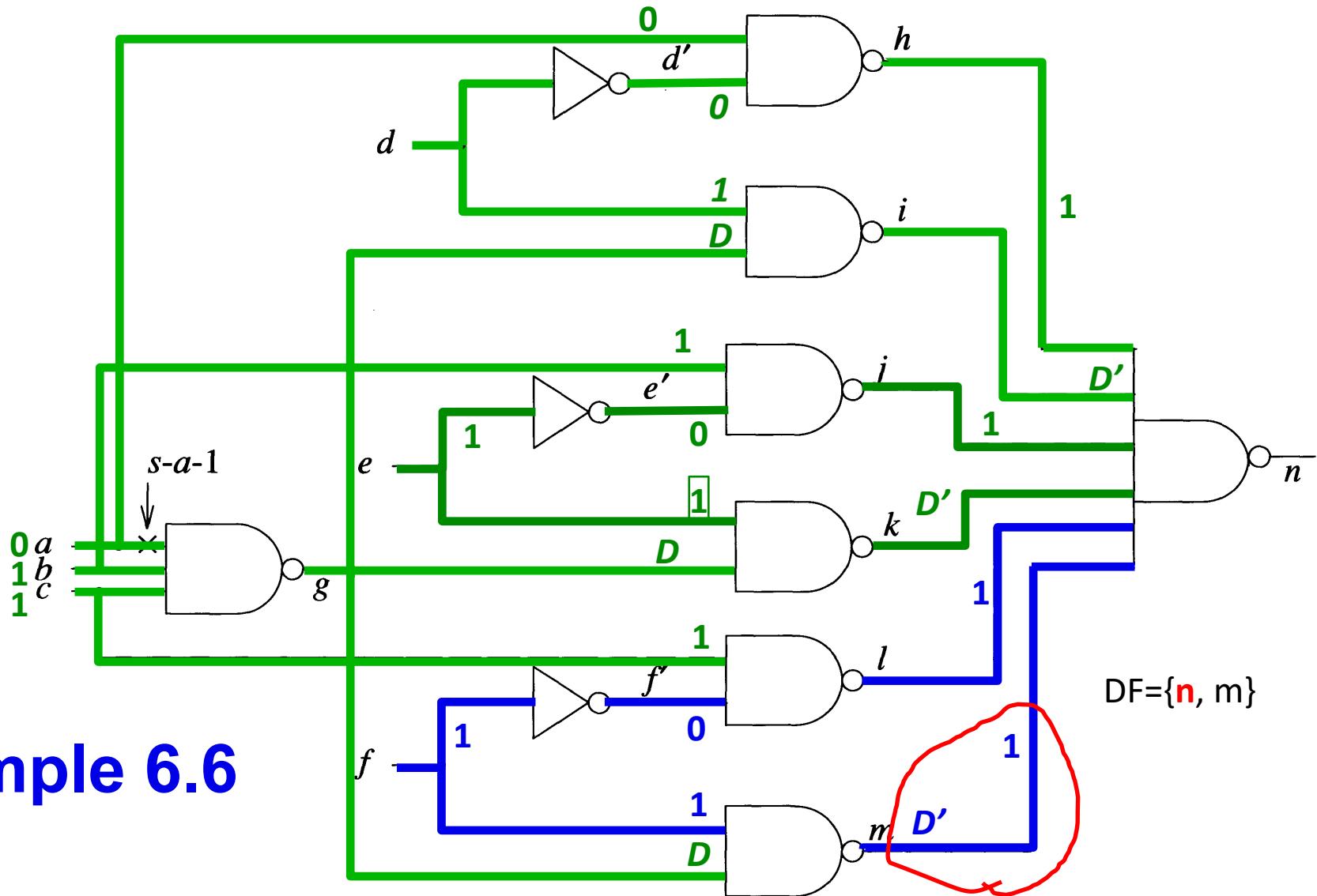
011

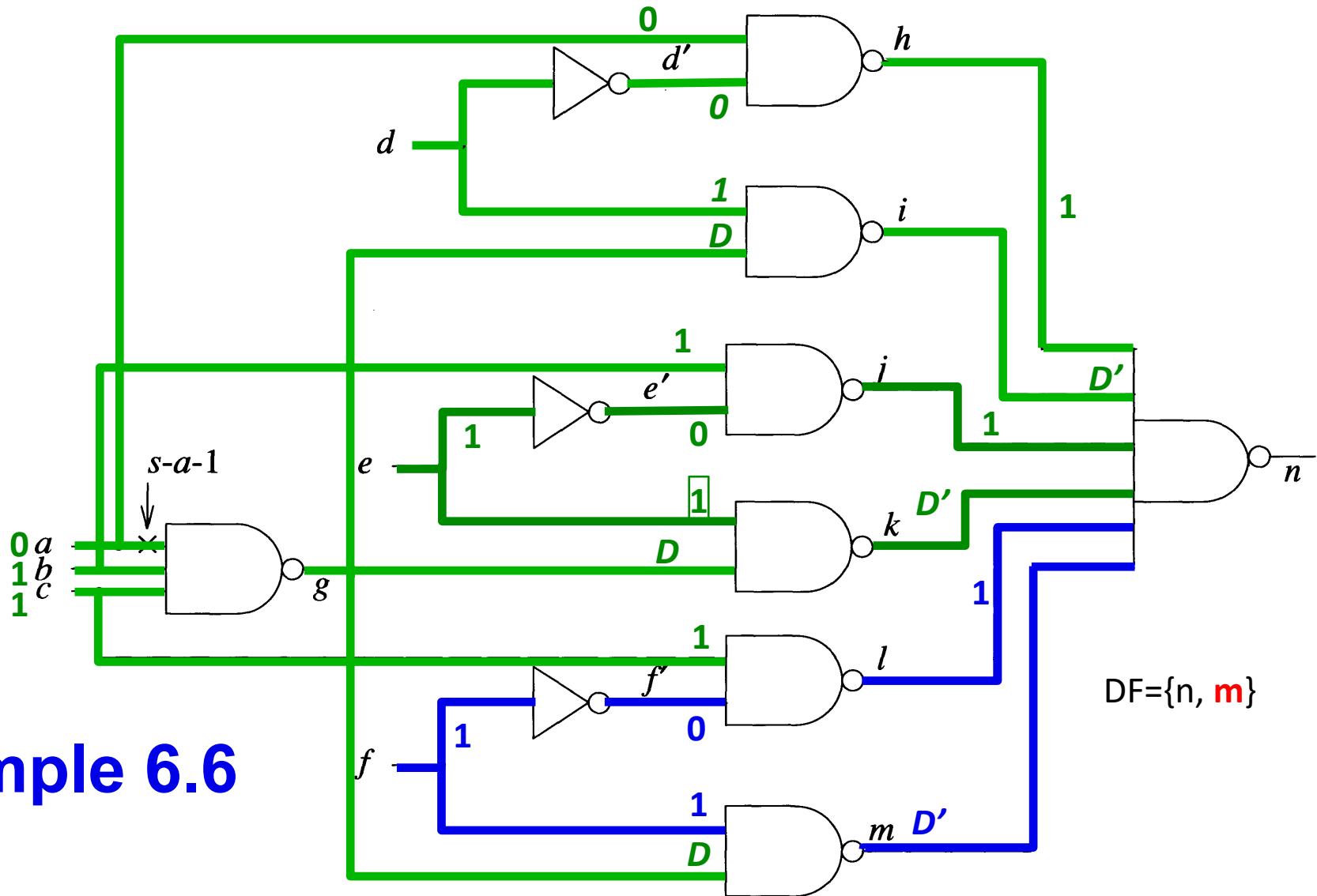
011

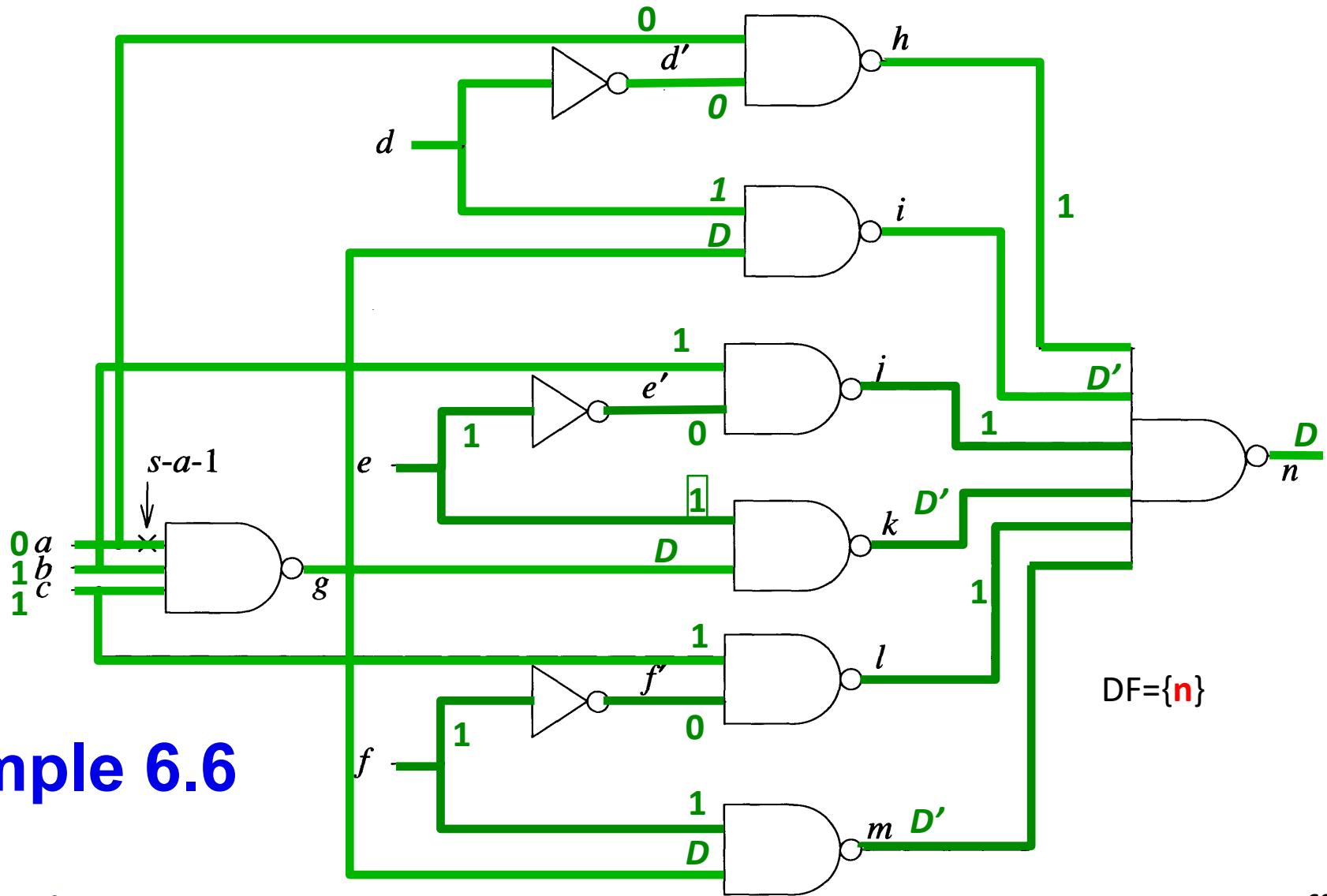
S-a-1



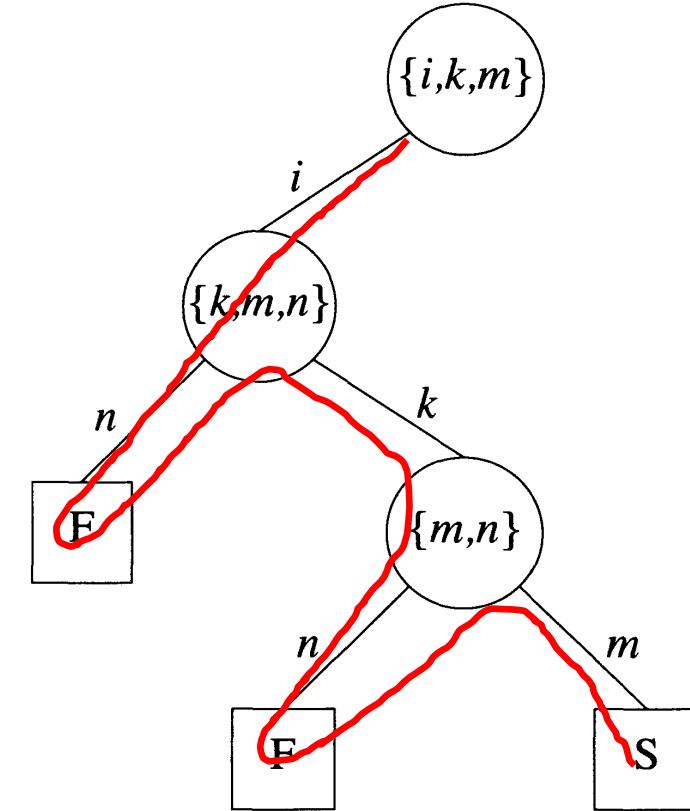
$$DF=\{n, k, m\}$$







Decisions	Implications
	$a=0$ $h=1$ $b=1$ $c=1$ $g=D$
$d=1$	$i=\bar{D}$ $d'=0$
$j=1$ $k=1$ $l=1$ $m=1$	$n=D$ $e'=0$ $e=1$ $k=\bar{D}$
$e=1$	$k=\bar{D}$ $e'=0$ $j=1$
$l=1$ $m=1$	$n=D$ $f'=0$ $f=1$ $m=\bar{D}$
$f=1$	$m=\bar{D}$ $f'=0$ $l=1$ $n=D$



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

```
             $i = \text{inversion of } k$ 
```

```
            select an input ( $j$ ) of  $k$  with value  $x$ 
```

```
             $v = v \oplus i$ 
```

```
             $k = j$ 
```

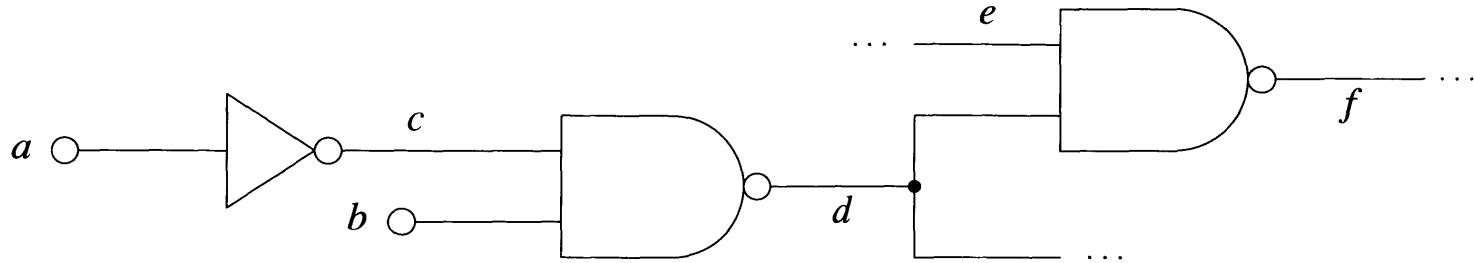
```
        end
```

```
    /*  $k$  is a PI */
```

```
    return ( $k, v$ )
```

```
end
```

Backtrace – An Example



- 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* $(f, 1)$

Figure 6.28

Selecting an Objective

Objective()

begin

/* the target fault is l s-a-v */

if (the value of l is x) **then return** (l, \bar{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, \bar{c})

end

Activate fault



Find the
necessary
inputs to
propagate fault



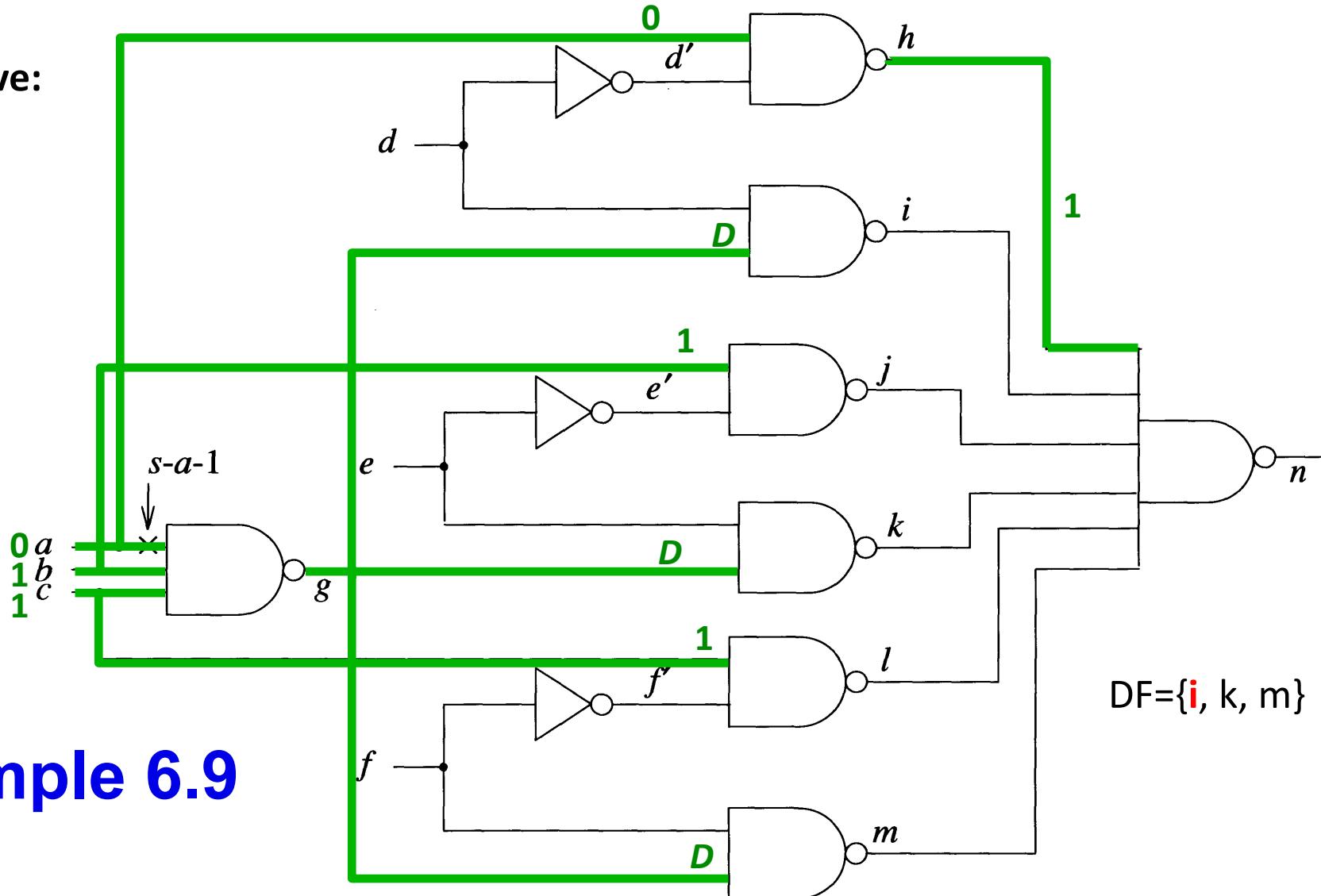
```

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 ( $j, \bar{v}_j$ )
    if PODEM() = SUCCESS then return SUCCESS
    Imply ( $j, x$ )
    return FAILURE // D-frontier becomes empty
end

```

Objective:

$d=1$



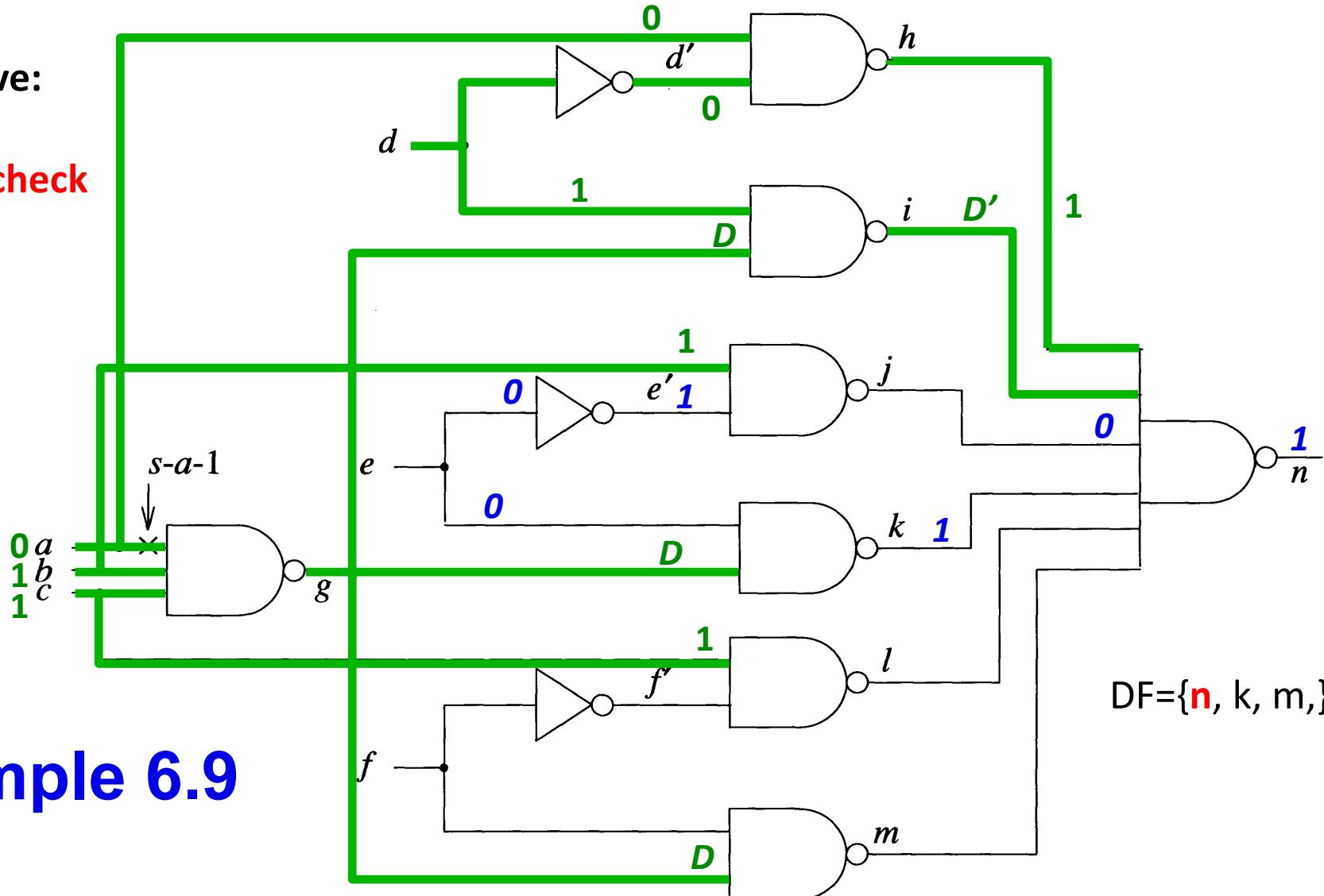
Example 6.9

Objective:

k=1

X-path check

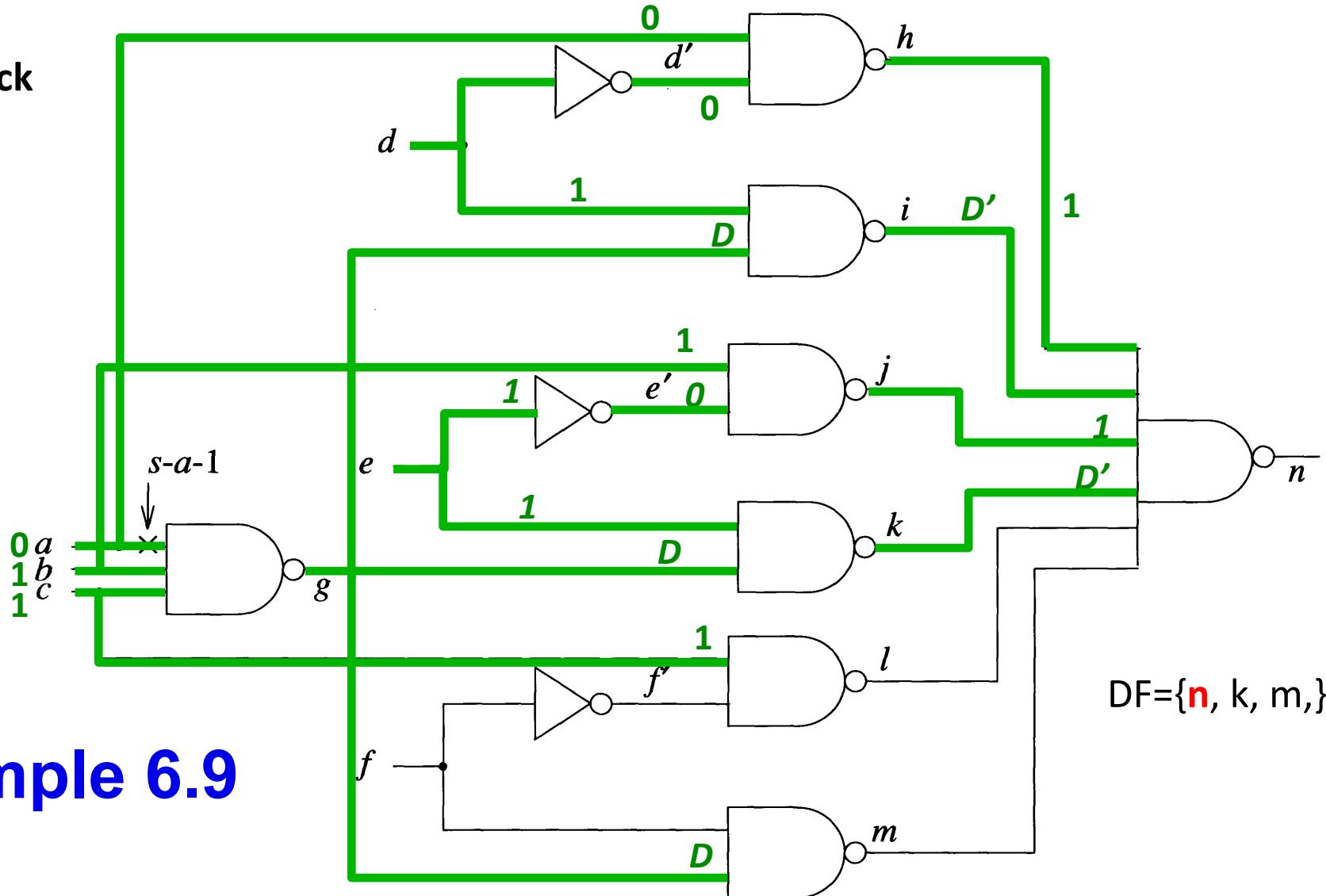
Failed!



$$DF=\{n, k, m,\}$$

Example 6.9

Backtrack

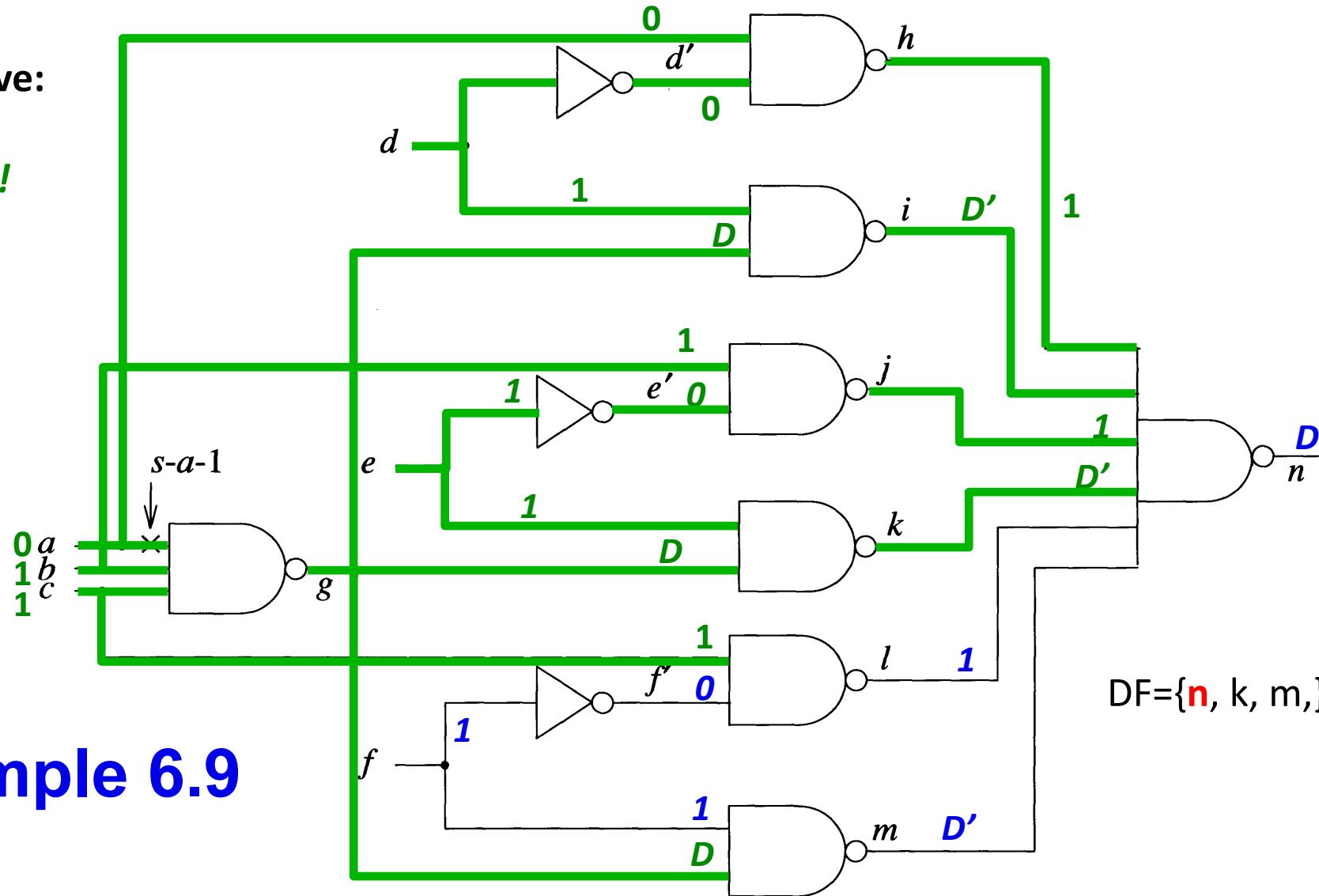


Example 6.9

Objective:

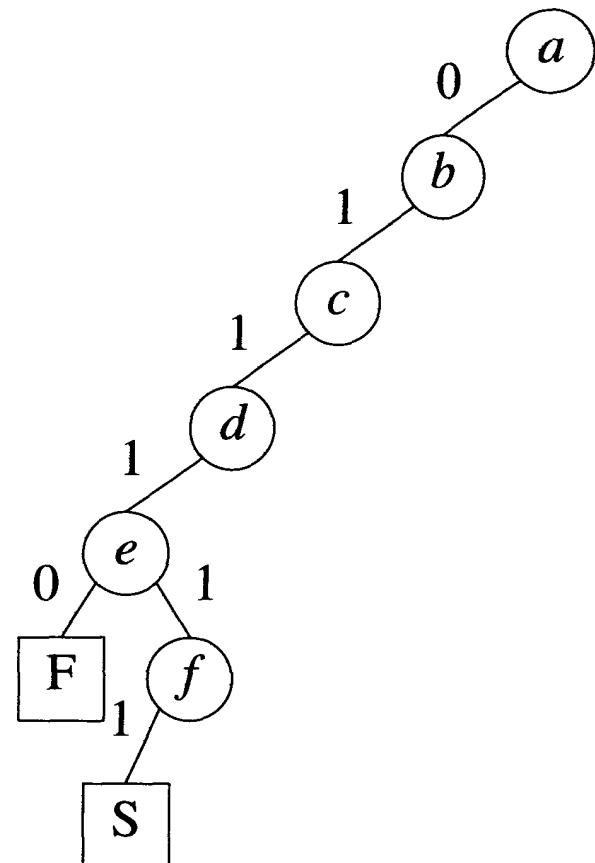
I=1

Success!



Example 6.9

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



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

$C_0(l) = \text{Relative difficulty of setting line } l \text{ to 0}$

$C_1(l) = \text{Relative difficulty of setting line } l \text{ to 1}$

Assume we know C_0 and C_1 costs of all inputs of the AND gate,

To set X to 0:

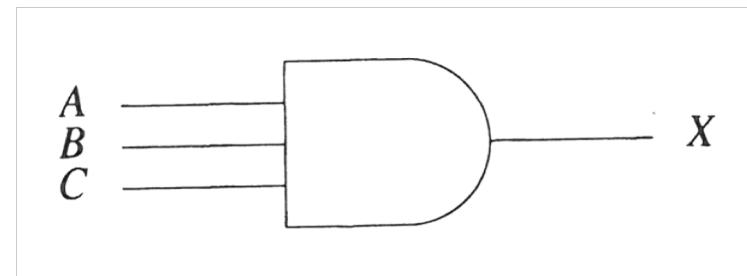
$$C_0(X) = \min \{C_0(A), C_0(B), C_0(C)\}$$

To set X to 1:

$$C_1(X) = C_1(A) + C_2(B) + C_3(C)$$

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

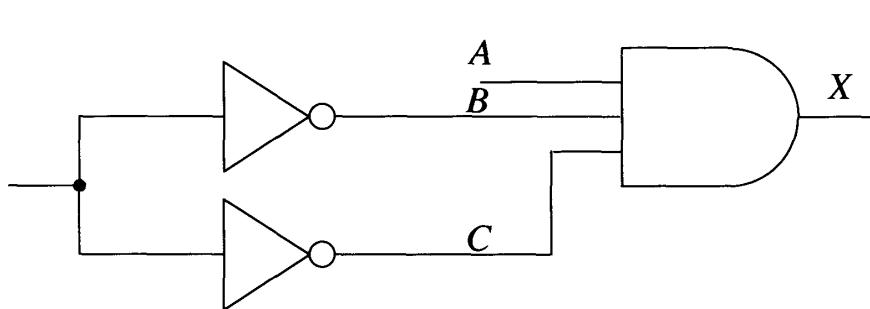
- Set C0 and C1 for every primary input to 1
- Compute C0's and C1' level by level
 - Cost are computed only after predecessor costs are known
- Costs can be computed in one forward traversal
- Linear in number of gates

Issues

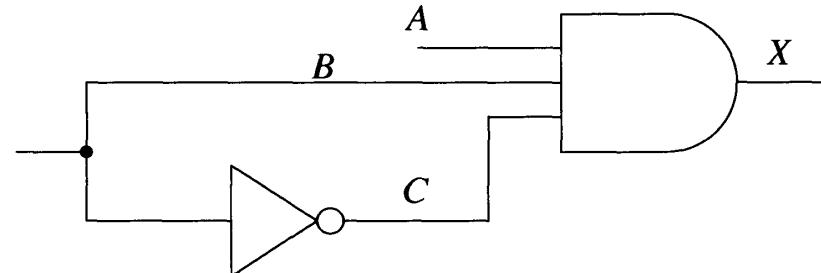
If inputs of a gate are not independent, it can lead to incorrect results

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 $O(l)$

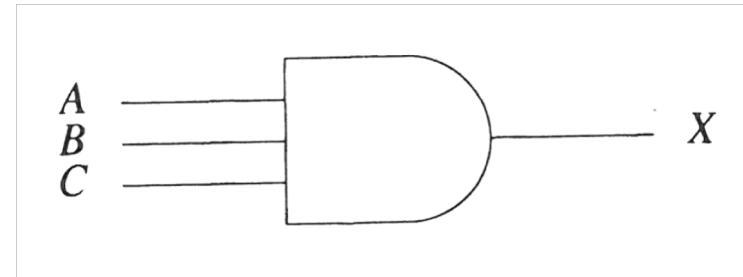
Cost of observing the input A?

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

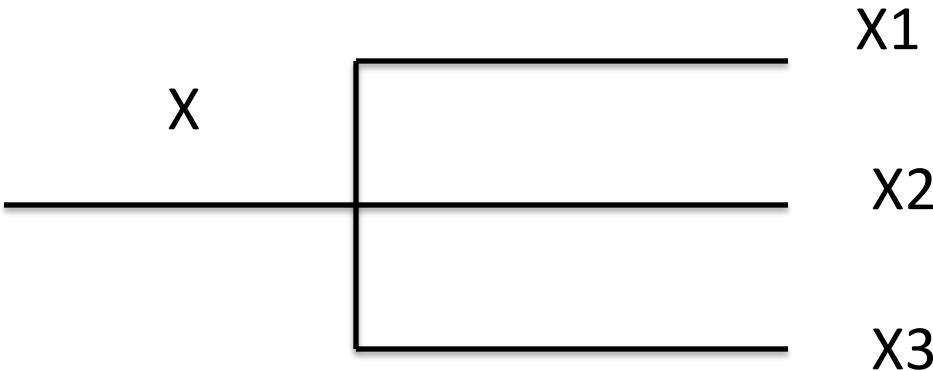
$$O(A) = C_1(B) + C_1(C) + O(X) \dots \text{Eq (3)}$$

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

What about OR gate?



Observability of a Stem X



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

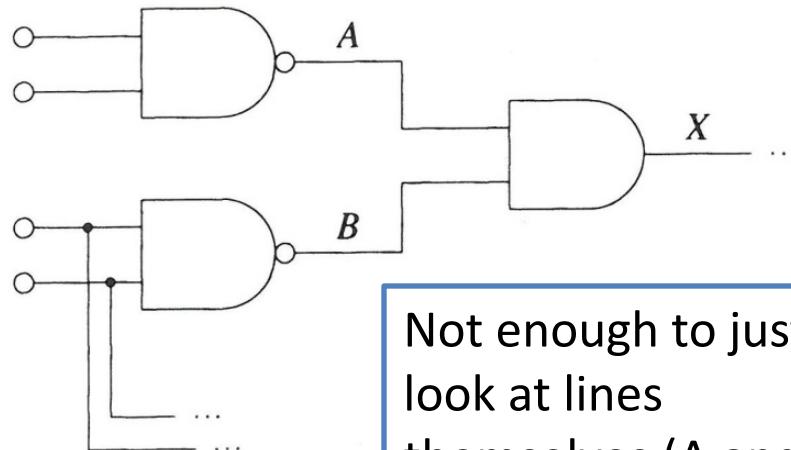
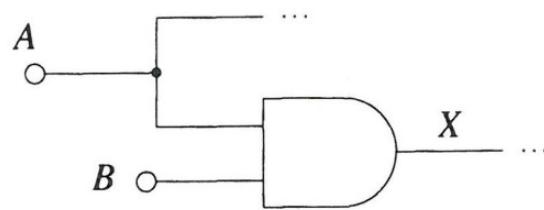
Assuming single path propagation is possible

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

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

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

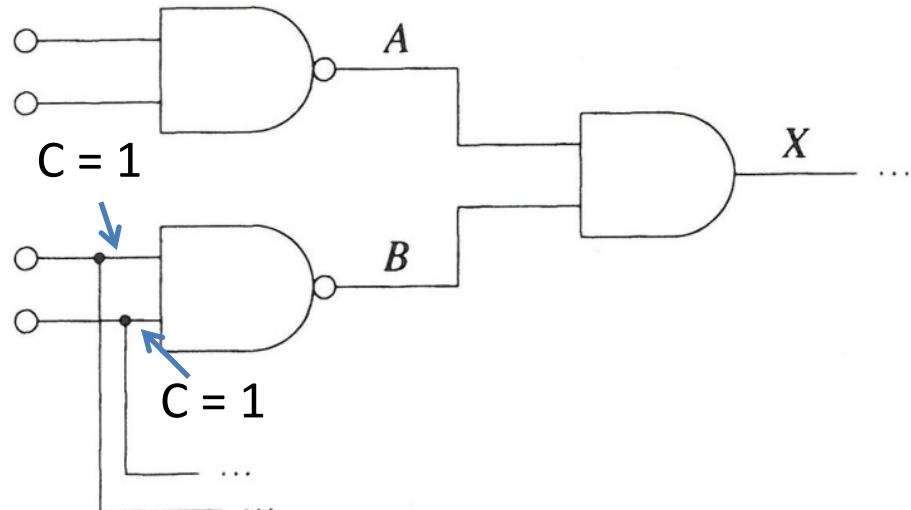
$$C(l) = \sum_i C(i) + f_l - 1$$

$$C(A) = 0$$

$$C(B) = 2$$

$$C(X) = 2$$

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

$$C_0(l) = \min \{C_0(i)\} + f_l - 1$$

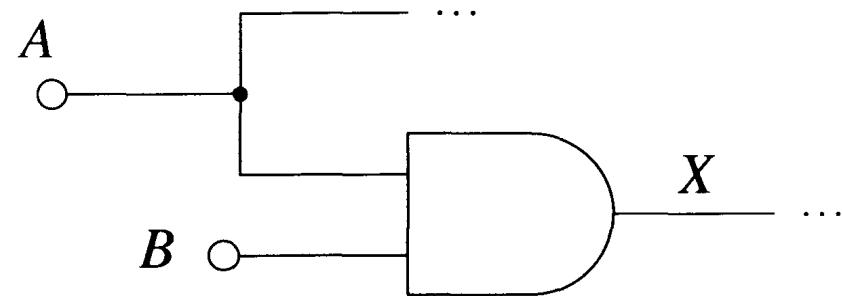
$$C(l) = \sum_i C(i) + f_l - 1$$

$$C_0(A) = C_1(A) = 1$$

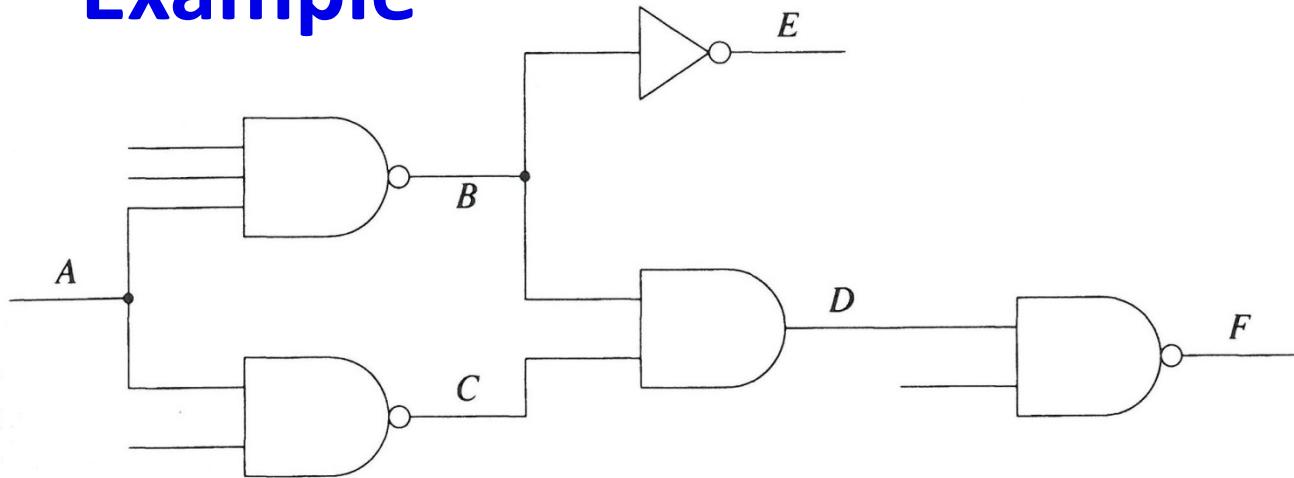
$$C_0(B) = C_1(B) = 0$$

$$C_0(X) = 0,$$

$$C_1(X) = 1.$$



Side Effects – Example

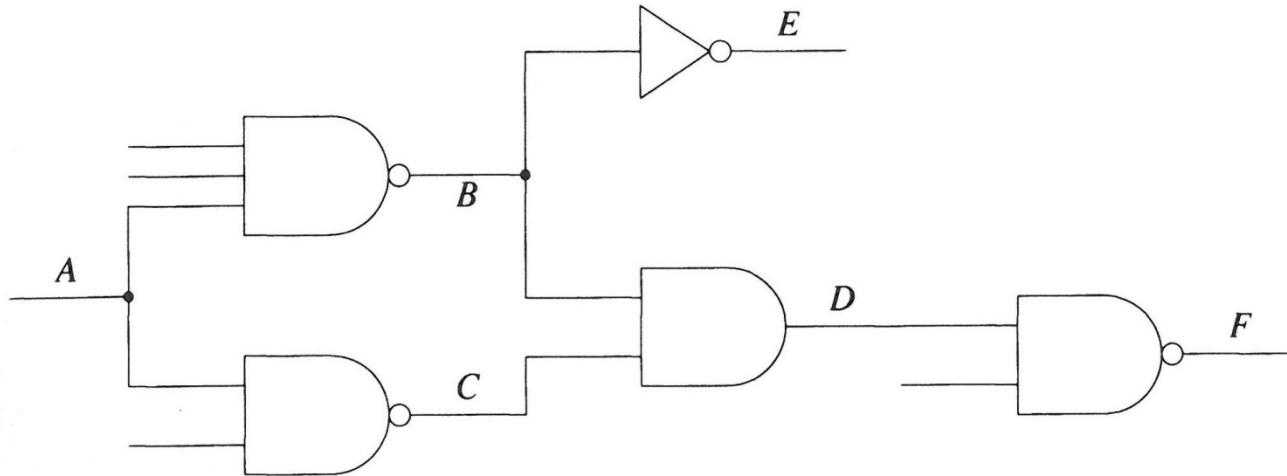


- $C_0(A)$ and $C_1(A)$ both have corrective terms =1
- $A = 0$ has greater potential of conflicts than $A = 1$
 - $A = 0$ results in B, C, D, E being set to binary values
 - Less x-paths for error propagation.

Side Effects Cost Function

- Side-Effects Cost Functions: $\text{CS0}(l)$ and $\text{CS1}(l)$ 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



- $\text{CS}_0(A) = 4(1/2)$
- $\text{CS}_1(A) = (1/3) + (1/2) = 5/6$

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

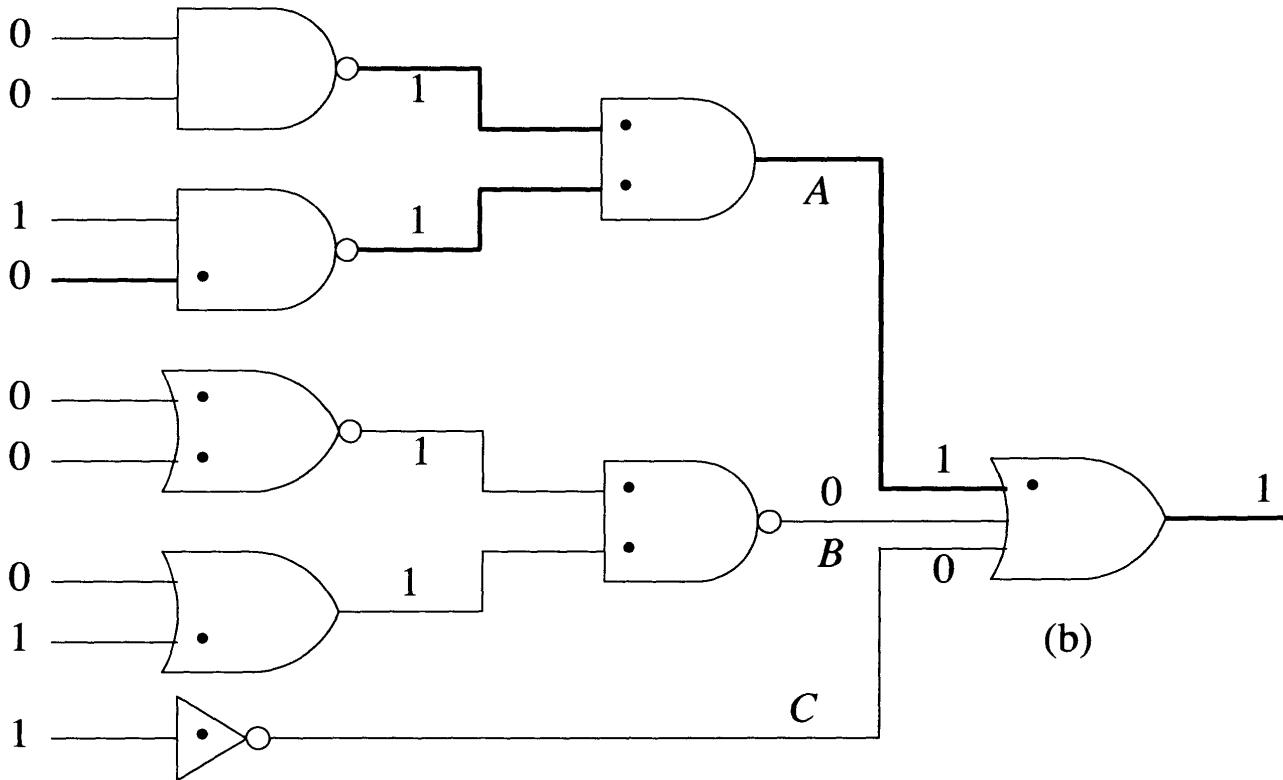
- Complexity of cost function computation must be low.
- Cost functions are based heuristics.
- Dynamic cost functions may lead to better performance.

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

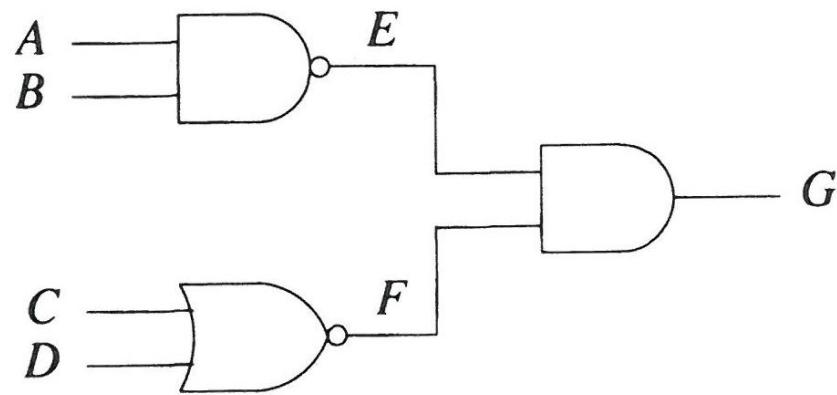
(a)

By Critical Cubes

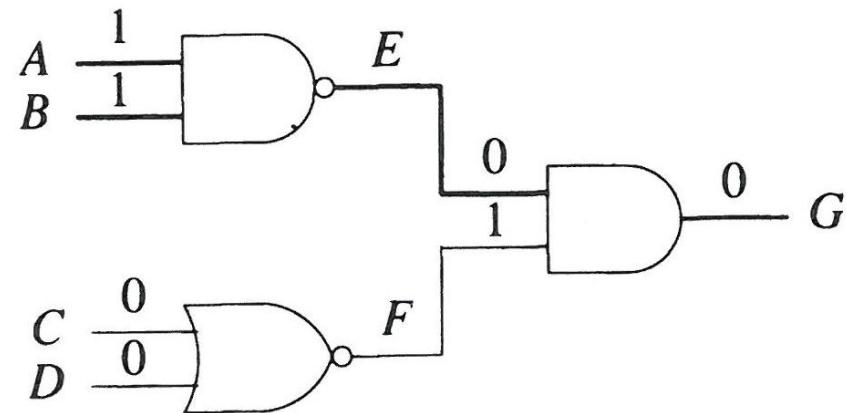
A	B	C	Z
1	1	1	1
0	1	1	0
1	0	1	0
1	1	0	0

(b)

Critical-path TG - Example



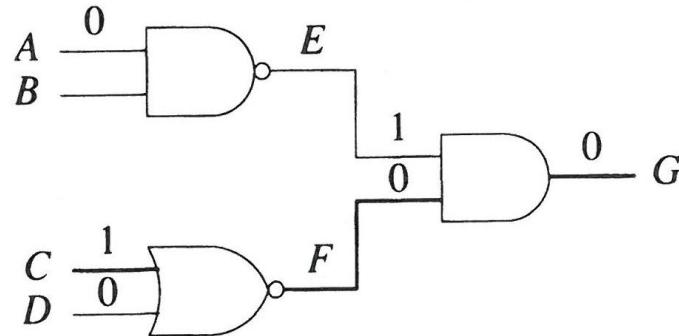
(a)



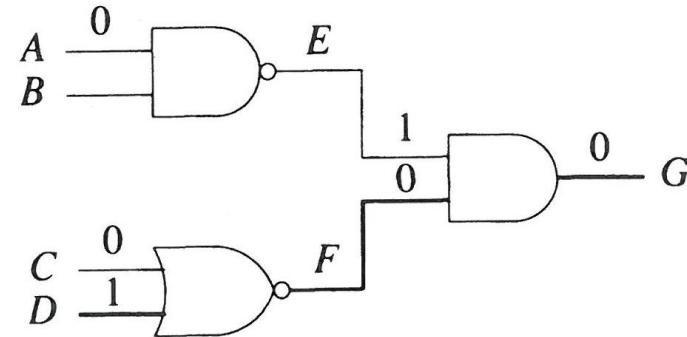
(b)

What SSFs can be detected by this input vector?

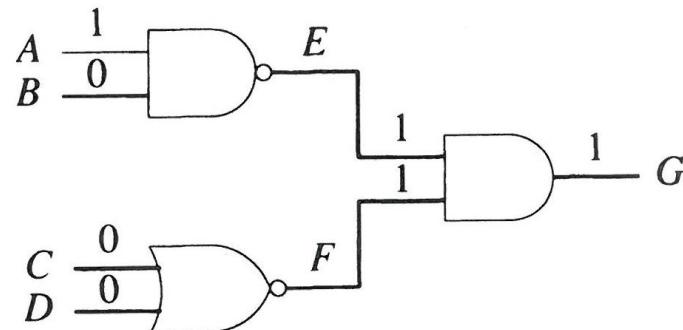
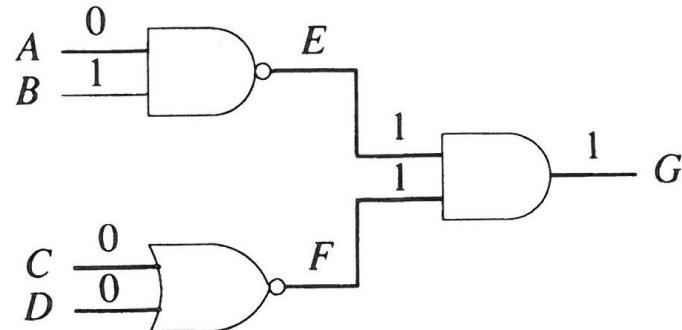
Critical-path TG – Example ...contd.



(c)



(d)



```

CPTGFF()
begin
  while (Critical ≠ Ø)
    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 =  $\bar{c}$ )
            then for every input j of l
              add (j, $\bar{c}$ ) 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, $\bar{c}$ )
                      CPTGFF()
                  end
                end
              return
            end
          end
        end
      end
    /* Critical = Ø */
    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

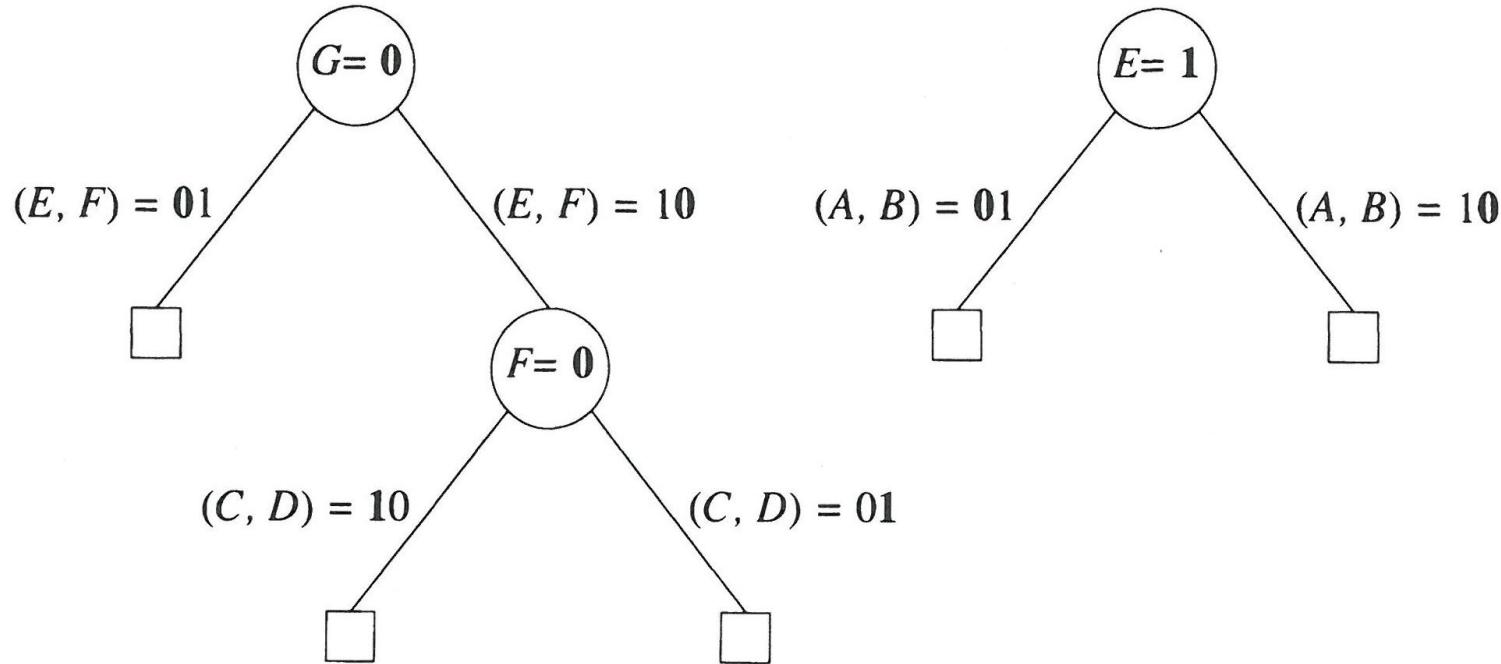
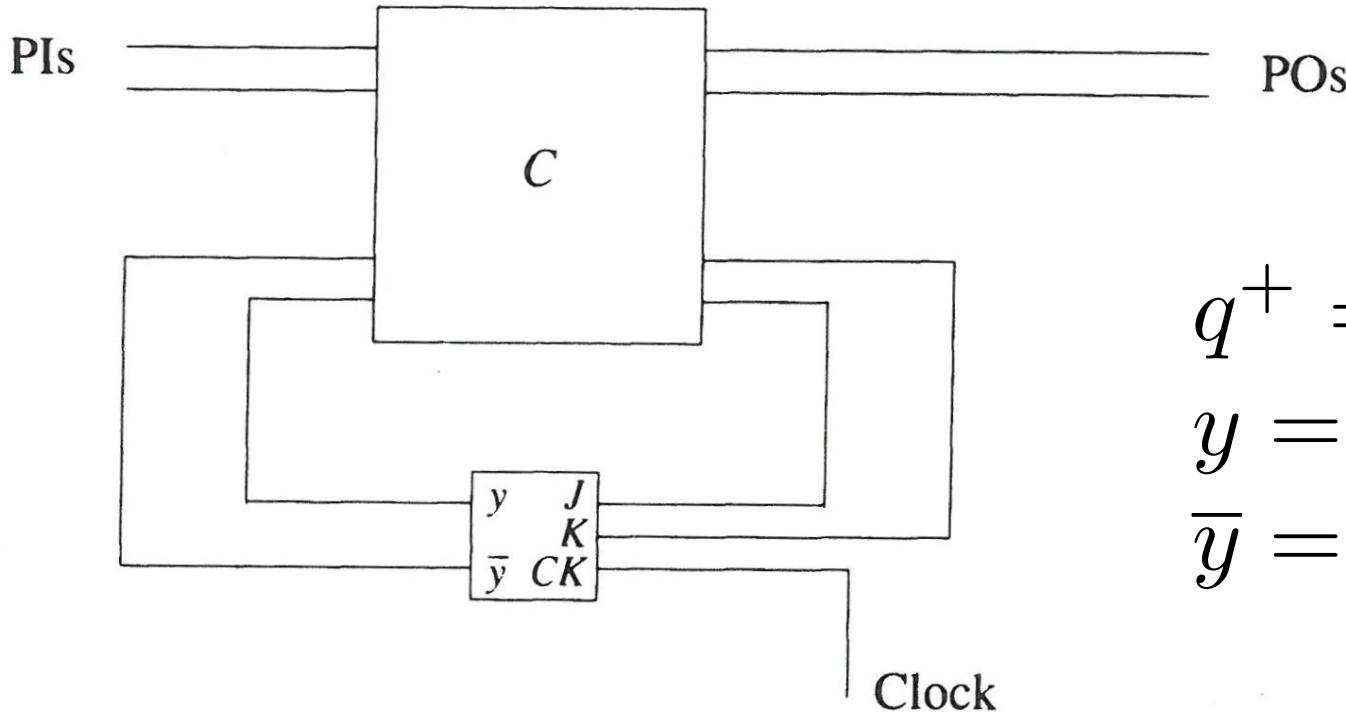


Figure 6.47 Decision trees for Example 6.12
The number of terminal nodes equals the number of tests generated. 101

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

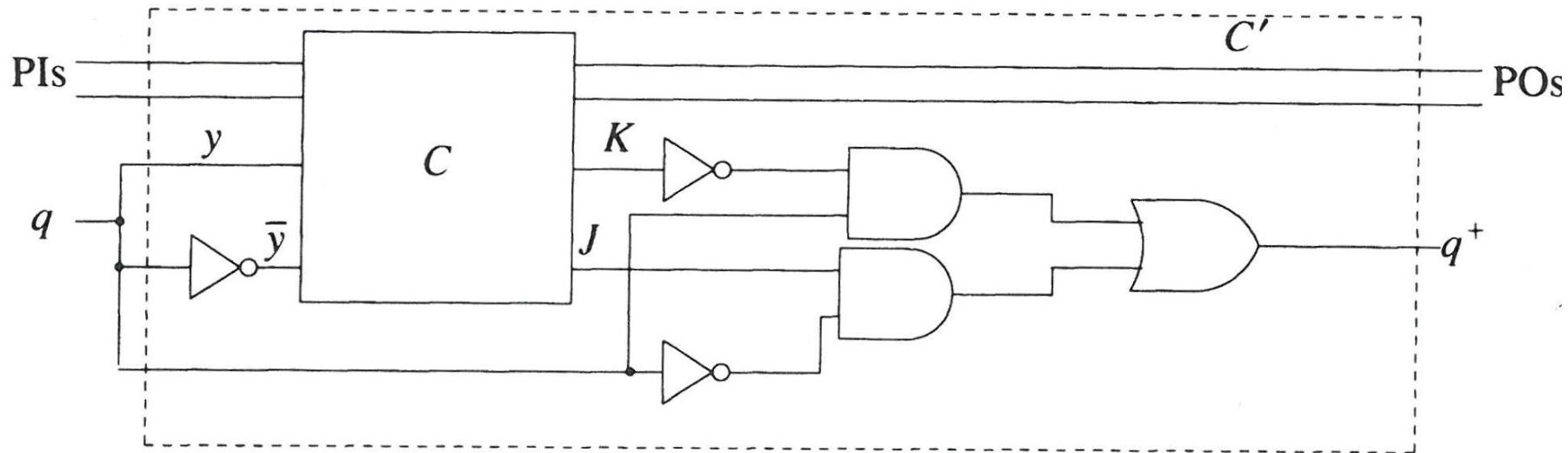


$$q^+ = J\bar{q} + \bar{K}q$$

$$y = q$$

$$\bar{y} = \bar{q}$$

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
\bar{D}	$s-a-0$	0
\bar{D}	$s-a-1$	\bar{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

$q(1) = \text{given initial state}$

if (test generation is successful) **then return** SUCCESS

/* no solution with r frames */

$r = r + 1$

end

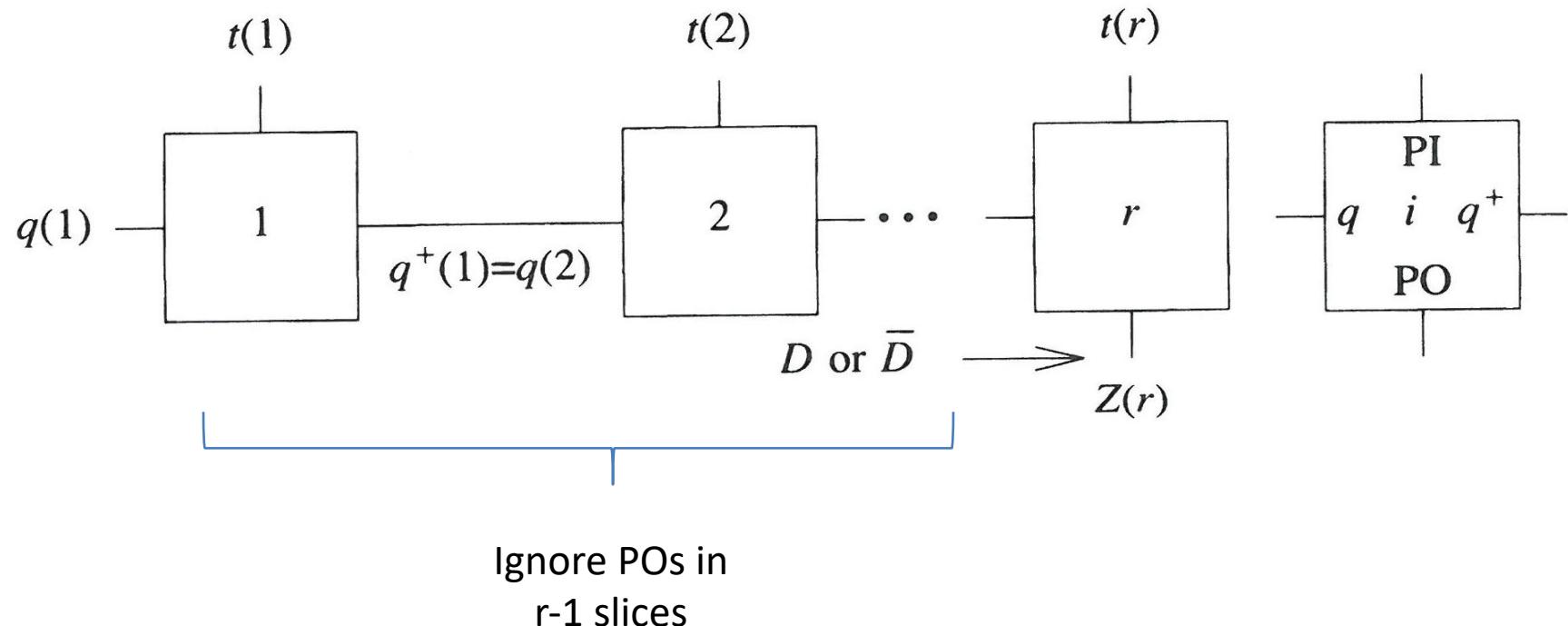
until $r = f_{\max}$

return FAILURE

Maximum Unroll factor

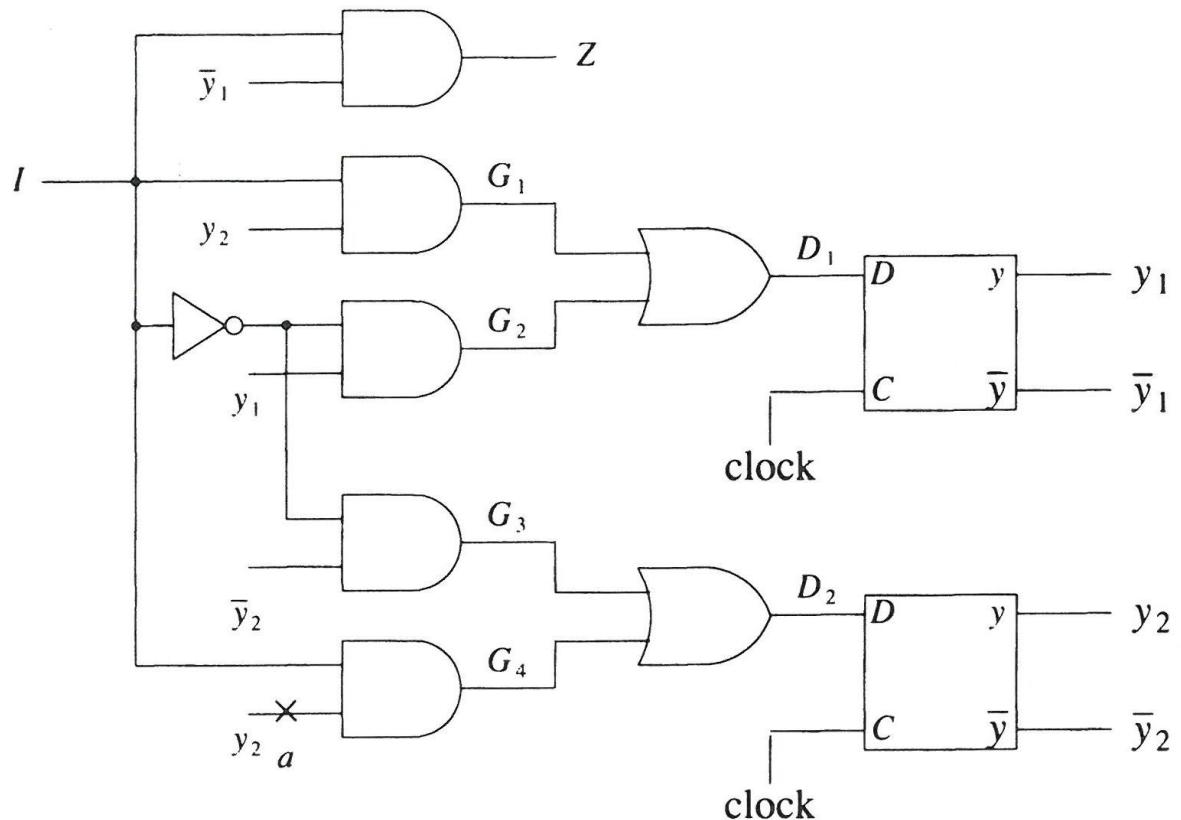
Once circuit is unrolled, we can use any of the test generation algorithm we studied for combinational circuits, such as D-alg(), PODEM, etc.

Iterative Array Model

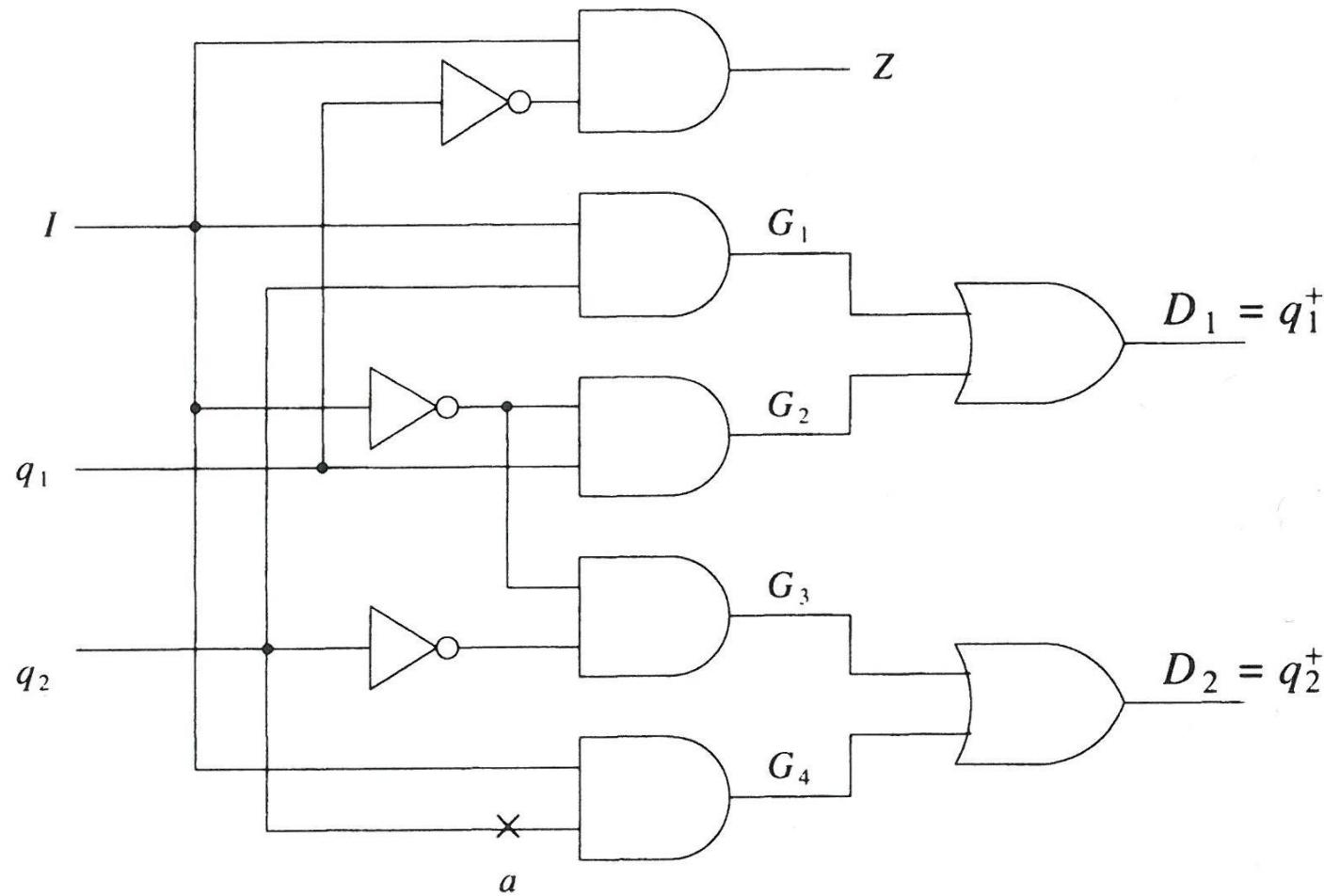


Example

→ Assume $q_1 = q_2 = 0$

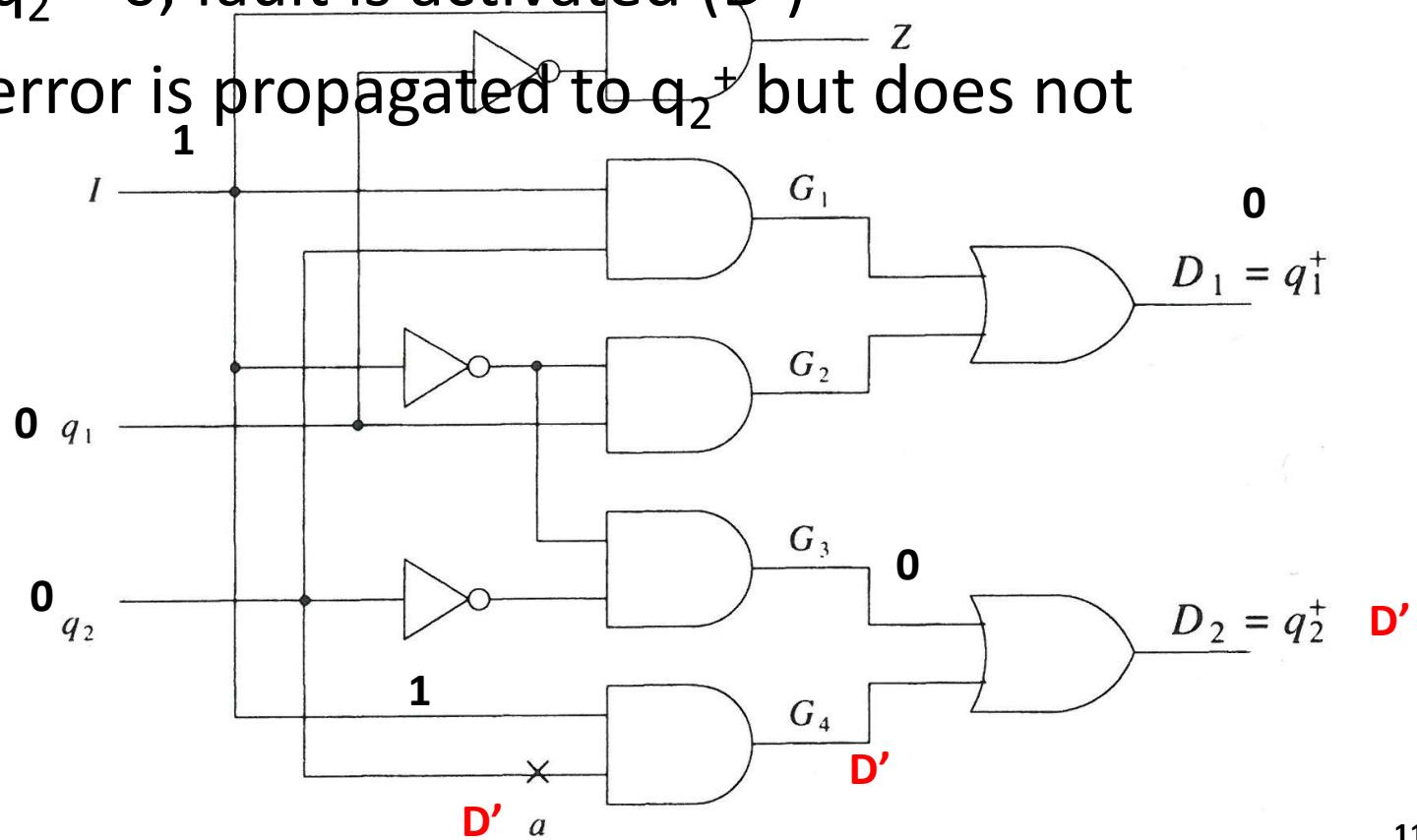


Time Frame



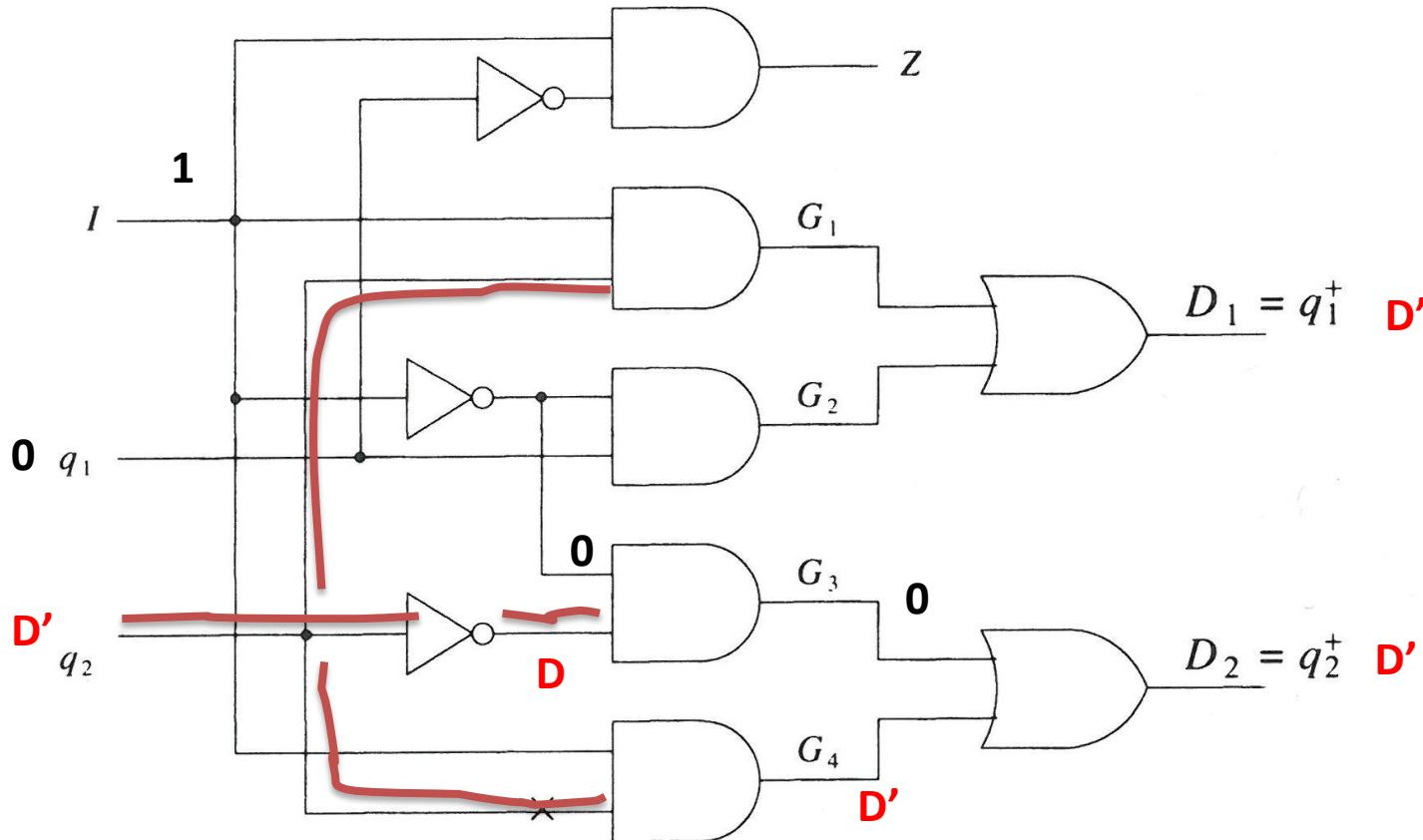
Time Frame 1

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



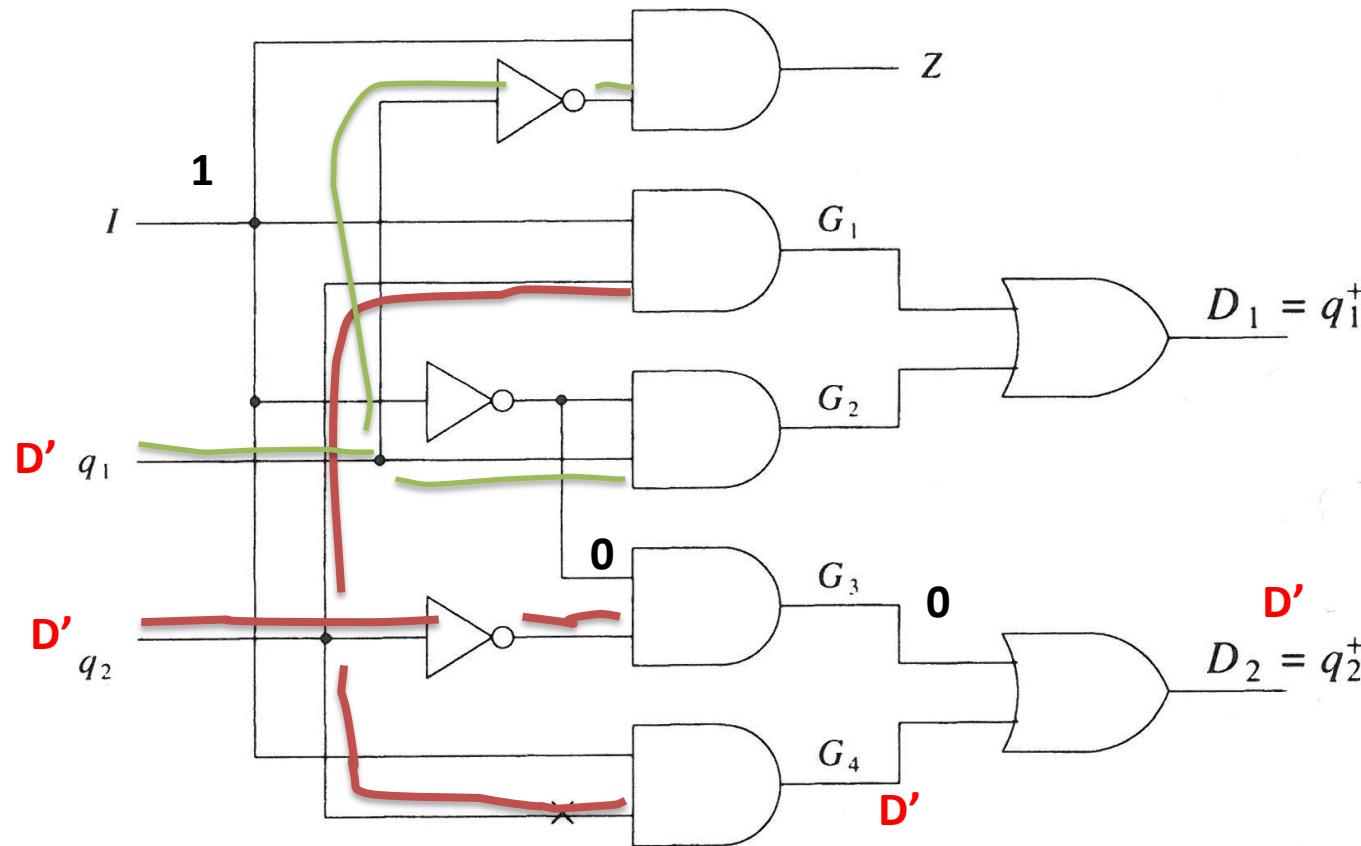
Time Frame 2

- 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 $q_2^+ = D$



Time Frames [cont]

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

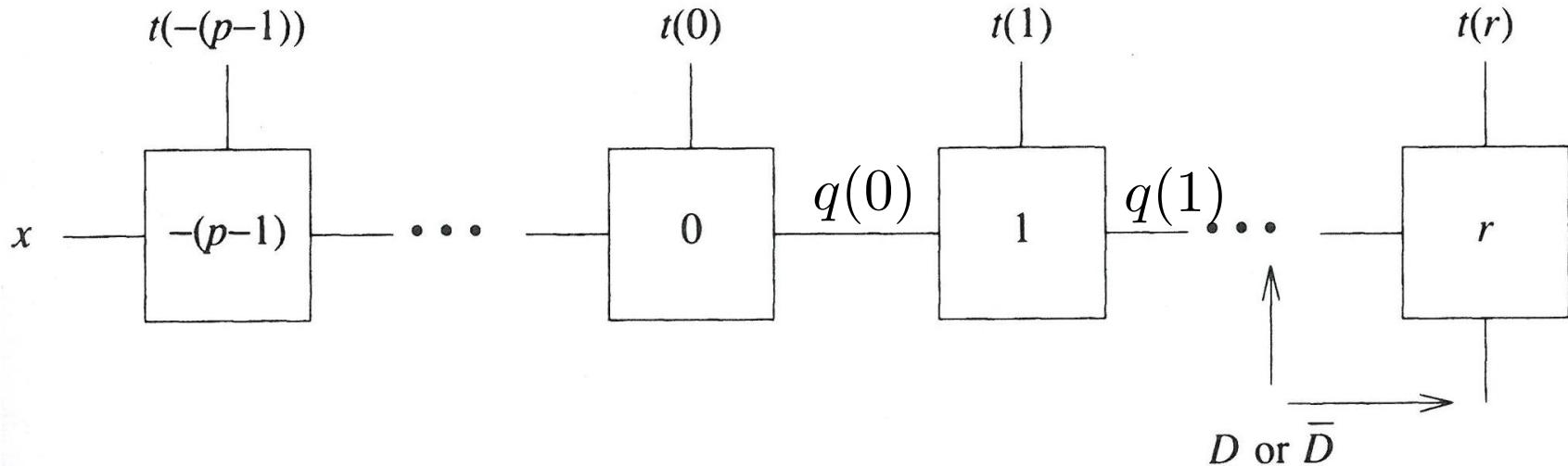


Generation of Self-initializing Test Sequences

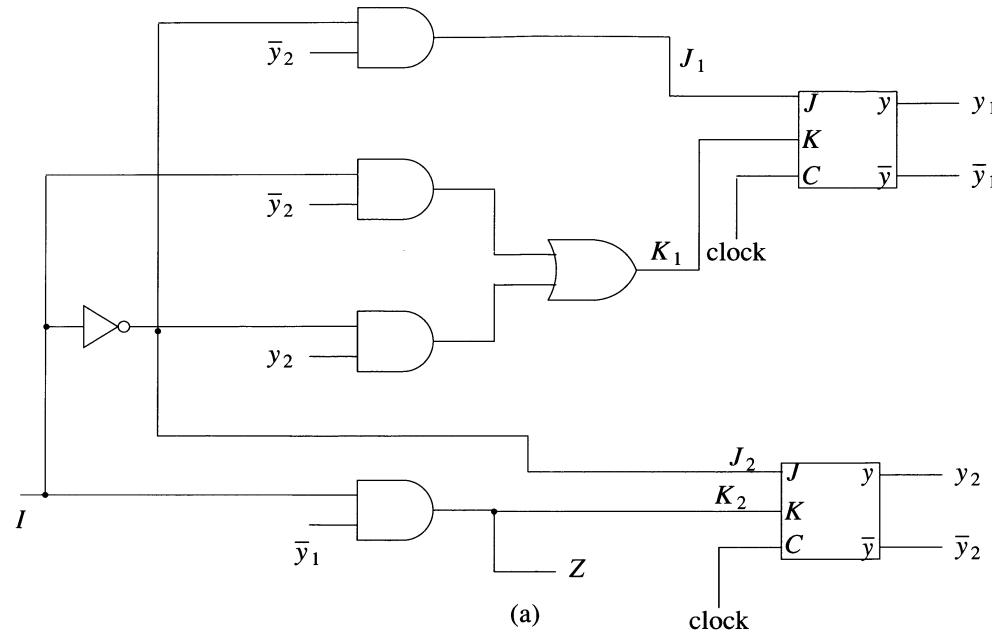
```
r = 1
p = 0
repeat
    begin
        build model with  $p + r$  time frames
        ignore the POs in the first  $p + r - 1$  frames
        ignore the  $q^+$  outputs in the last frame
        if (test generation is successful and every  $q$  input in the first frame has
            value  $x$ )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

Generation of Self-initializing Test Sequences

(a)

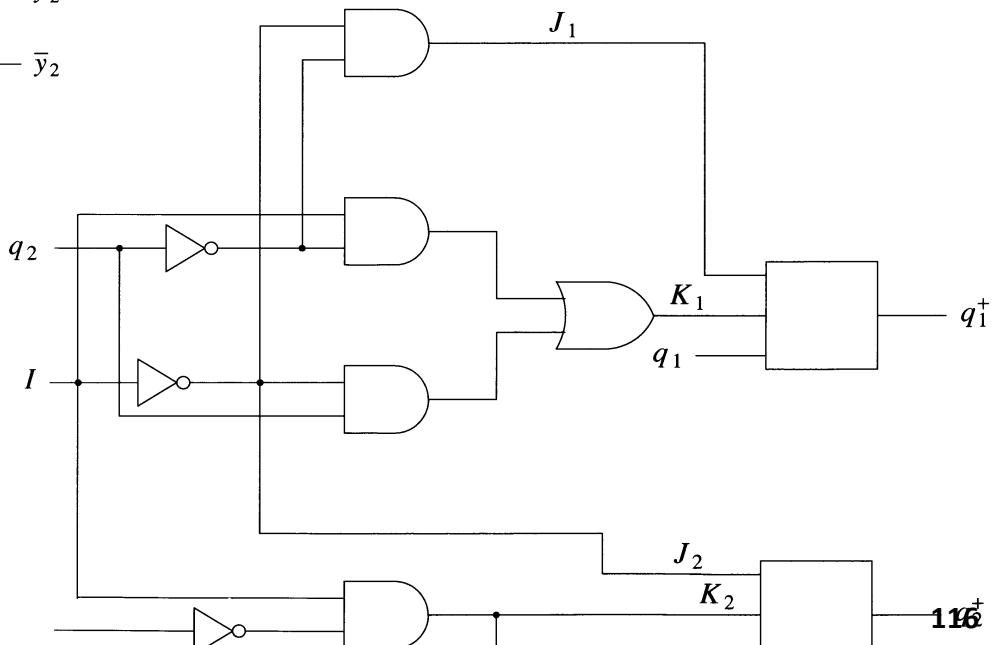


1. Activate fault in frame 1, and propagate it to PO using r frames.
2. If $q(0)$ is not all x , justify $q(0)$ by backward propagation of p frames.

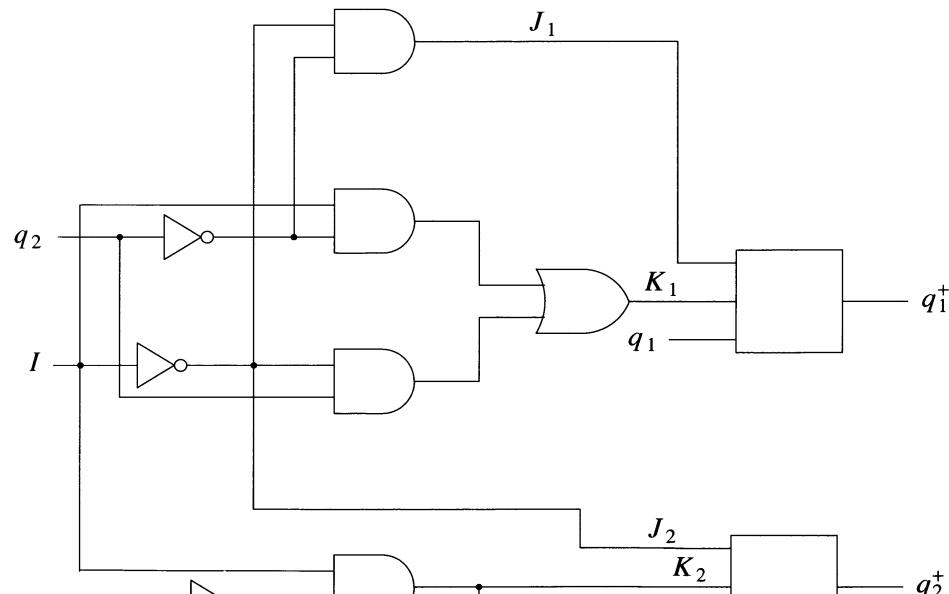
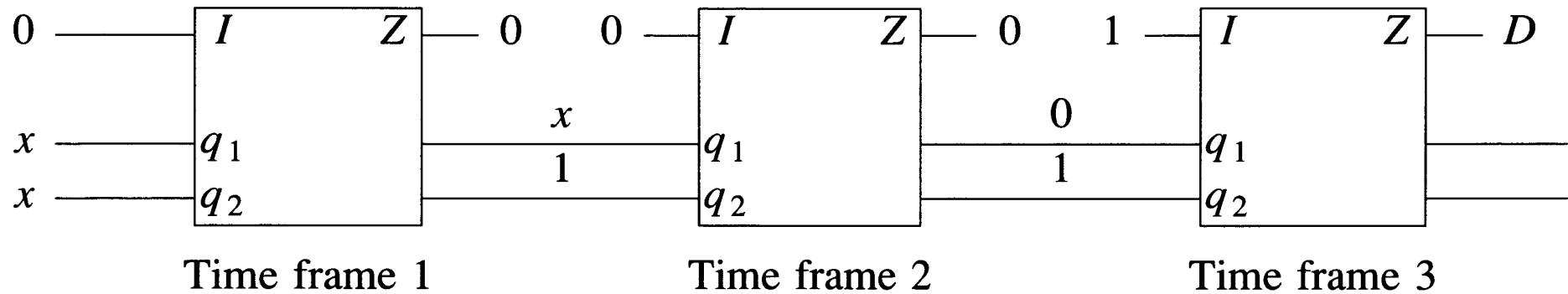


(a)

$$q^+ = J\bar{q} + \bar{K}q \quad \rightarrow$$

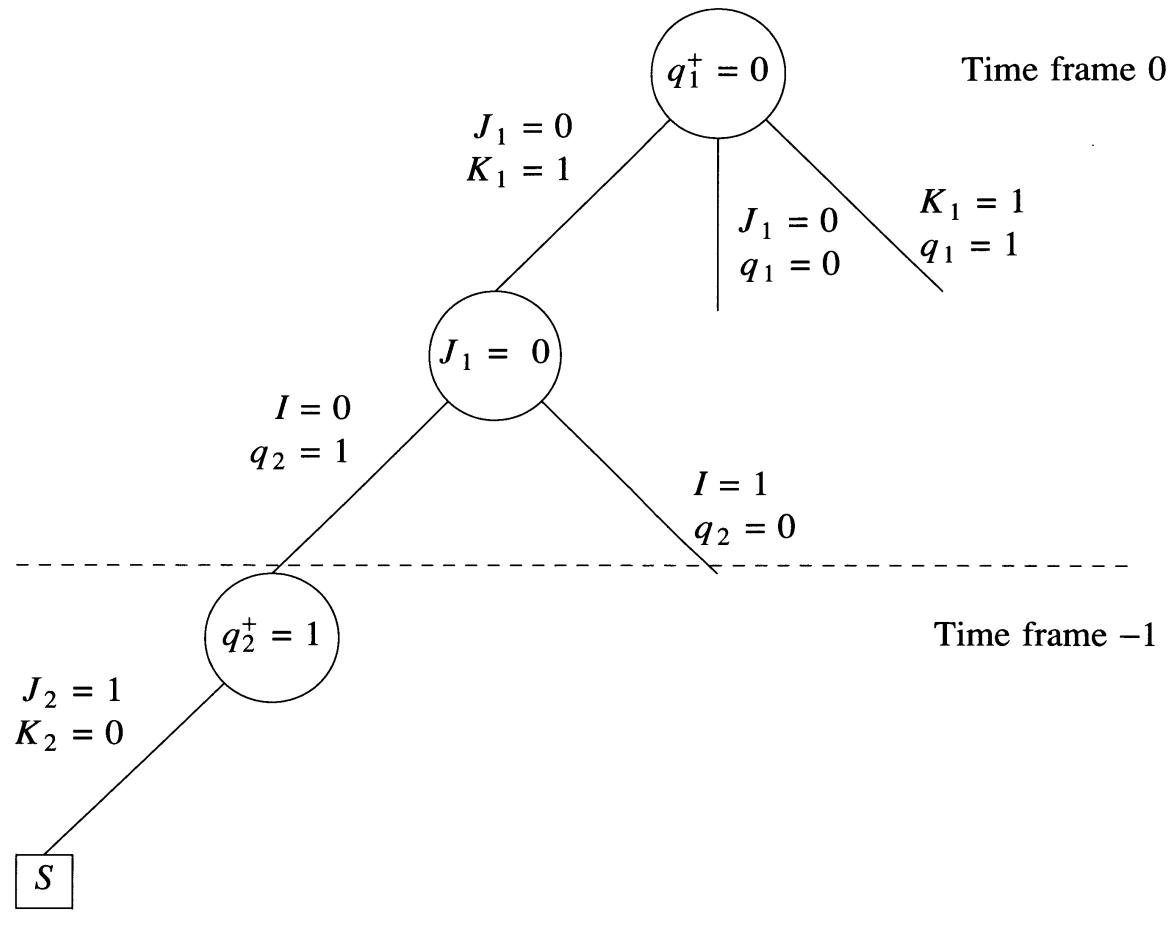


Example: Iterative Array: Detect Z s-a-0



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

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



CDCL SAT Solvers & SAT-Based Problem Solving

Joao Marques-Silva^{1,2} & Mikolas Janota²

¹University College Dublin, Ireland

²IST/INESC-ID, Lisbon, Portugal

SAT/SMT Summer School 2013
Aalto University, Espoo, Finland

The Success of SAT

- Well-known NP-complete decision problem
 - In practice, SAT is a success story of Computer Science
 - Hundreds (even more?) of practical applications

A word cloud visualization representing various research topics in computer science, categorized by color:

- Red Words (Top Row):** Network Security Management, Fault Localization, Binate Covering, Noise Analysis, Pedigree Consistency, Function Decomposition, Model-Based Diagnosis, Technology Mapping Games.
- Yellow Words (Second Row):** Maximum Satisfiability, Configuration, Termination Analysis.
- Orange Words (Third Row):** Software Testing, Filter Design, Switching Network Verification, Equivalence Checking, Resource Constrained Scheduling, Abstract Argumentation.
- Dark Red Words (Fourth Row):** Satisfiability Modulo Theories, Package Management, Symbolic Trajectory Evaluation.
- Dark Orange Words (Fifth Row):** Quantified Boolean Formulas, Constraint Programming, FPGA Routing.
- Light Orange Words (Sixth Row):** Minimum Satisfiability, Cryptanalysis, Telecom Feature Subscription, Timetabling.
- Dark Brown Words (Seventh Row):** Software Model Checking, Haplotyping, Test Pattern Generation.
- Medium Orange Words (Eighth Row):** Model Finding, Planning, Logic Synthesis, Design Debugging.
- Light Brown Words (Ninth Row):** Hardware Model Checking, Power Estimation, Circuit Delay Computation, Genome Rearrangement.
- Dark Brown Words (Tenth Row):** Test Suite Minimization, Lazy Clause Generation.
- Medium Orange Words (Eleventh Row):** Pseudo-Boolean Formulas.

Part I

CDCL SAT Solvers

Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

Preliminaries

- **Variables:** $w, x, y, z, a, b, c, \dots$
- **Literals:** $w, \bar{x}, \bar{y}, a, \dots$, but also $\neg w, \neg y, \dots$
- **Clauses:** disjunction of literals **or** set of literals
- **Formula:** conjunction of clauses **or** set of clauses
- **Model (satisfying assignment):** partial/total mapping from variables to $\{0, 1\}$
- Formula can be **SAT/UNSAT**

Preliminaries

- **Variables:** $w, x, y, z, a, b, c, \dots$
- **Literals:** $w, \bar{x}, \bar{y}, a, \dots$, but also $\neg w, \neg y, \dots$
- **Clauses:** disjunction of literals **or** set of literals
- **Formula:** conjunction of clauses **or** set of clauses
- **Model (satisfying assignment):** partial/total mapping from variables to $\{0, 1\}$
- Formula can be **SAT/UNSAT**
- Example:

$$\mathcal{F} \triangleq (r) \wedge (\bar{r} \vee s) \wedge (\bar{w} \vee a) \wedge (\bar{x} \vee b) \wedge (\bar{y} \vee \bar{z} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d)$$

- Example models:
 - ▶ $\{r, s, a, b, c, d\}$
 - ▶ $\{r, s, \bar{x}, y, \bar{w}, z, \bar{a}, b, c, d\}$

Resolution

- Resolution rule:

[DP60,R65]

$$\frac{(\alpha \vee x) \quad (\beta \vee \bar{x})}{(\alpha \vee \beta)}$$

- Complete proof system for propositional logic

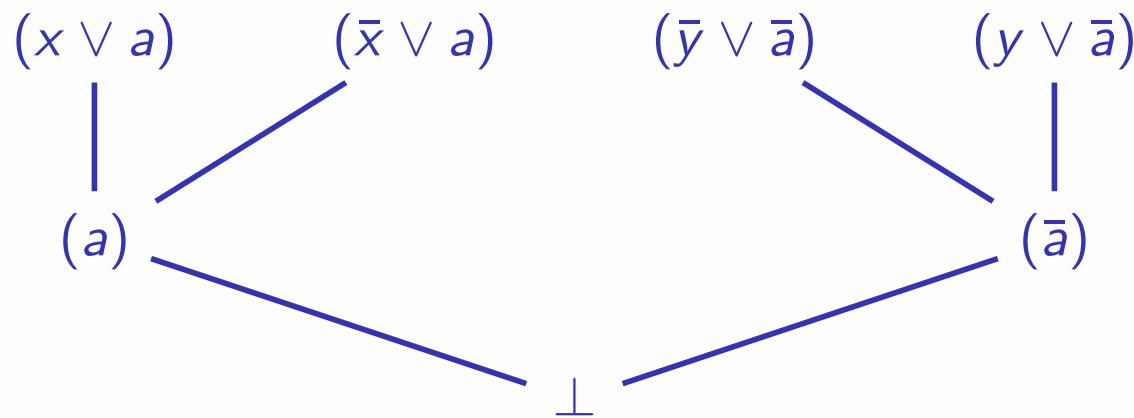
Resolution

- Resolution rule:

[DP60,R65]

$$\frac{(\alpha \vee x) \quad (\beta \vee \bar{x})}{(\alpha \vee \beta)}$$

- Complete proof system for propositional logic



- Extensively used with (CDCL) SAT solvers

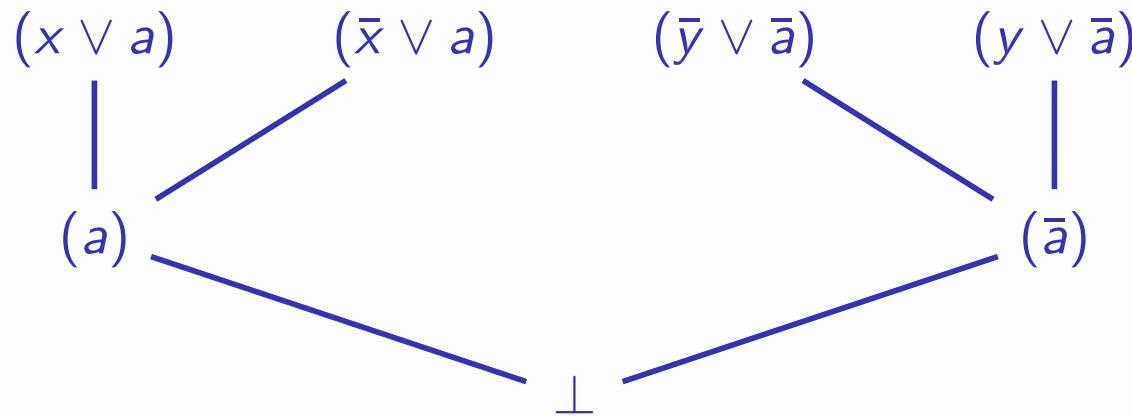
Resolution

- Resolution rule:

[DP60,R65]

$$\frac{(\alpha \vee x) \quad (\beta \vee \bar{x})}{(\alpha \vee \beta)}$$

- Complete proof system for propositional logic



- Extensively used with (CDCL) SAT solvers

- Self-subsuming resolution (with $\alpha' \subseteq \alpha$):

[e.g. SP04,EB05]

$$\frac{(\alpha \vee x) \quad (\alpha' \vee \bar{x})}{(\alpha)}$$

- (α) subsumes $(\alpha \vee x)$

Unit Propagation

$$\begin{aligned}\mathcal{F} = & (r) \wedge (\bar{r} \vee s) \wedge \\ & (\bar{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \\ & (\bar{y} \vee \bar{z} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d)\end{aligned}$$

Unit Propagation

$$\begin{aligned}\mathcal{F} = & (r) \wedge (\bar{r} \vee s) \wedge \\ & (\bar{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \\ & (\bar{y} \vee \bar{z} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d)\end{aligned}$$

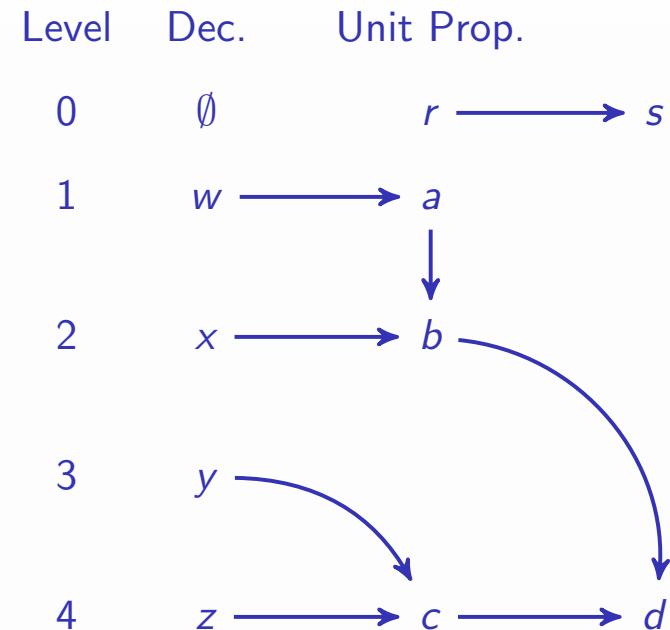
- Decisions / Variable Branchings:

$$w = 1, x = 1, y = 1, z = 1$$

Unit Propagation

$$\begin{aligned}\mathcal{F} = & (r) \wedge (\bar{r} \vee s) \wedge \\ & (\bar{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \\ & (\bar{y} \vee \bar{z} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d)\end{aligned}$$

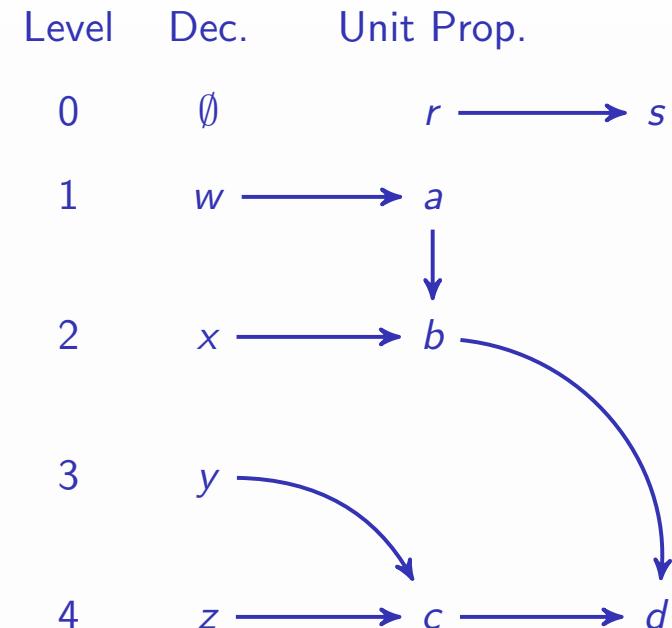
- Decisions / Variable Branchings:
 $w = 1, x = 1, y = 1, z = 1$



Unit Propagation

$$\begin{aligned}\mathcal{F} = & (r) \wedge (\bar{r} \vee s) \wedge \\ & (\bar{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \\ & (\bar{y} \vee \bar{z} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d)\end{aligned}$$

- Decisions / Variable Branchings:
 $w = 1, x = 1, y = 1, z = 1$



- Additional definitions:
 - Antecedent (or reason) of an implied assignment
 - $(\bar{b} \vee \bar{c} \vee d)$ for d
 - Associate assignment with decision levels
 - $w = 1 @ 1, x = 1 @ 2, y = 1 @ 3, z = 1 @ 4$
 - $r = 1 @ 0, d = 1 @ 4, \dots$

Outline

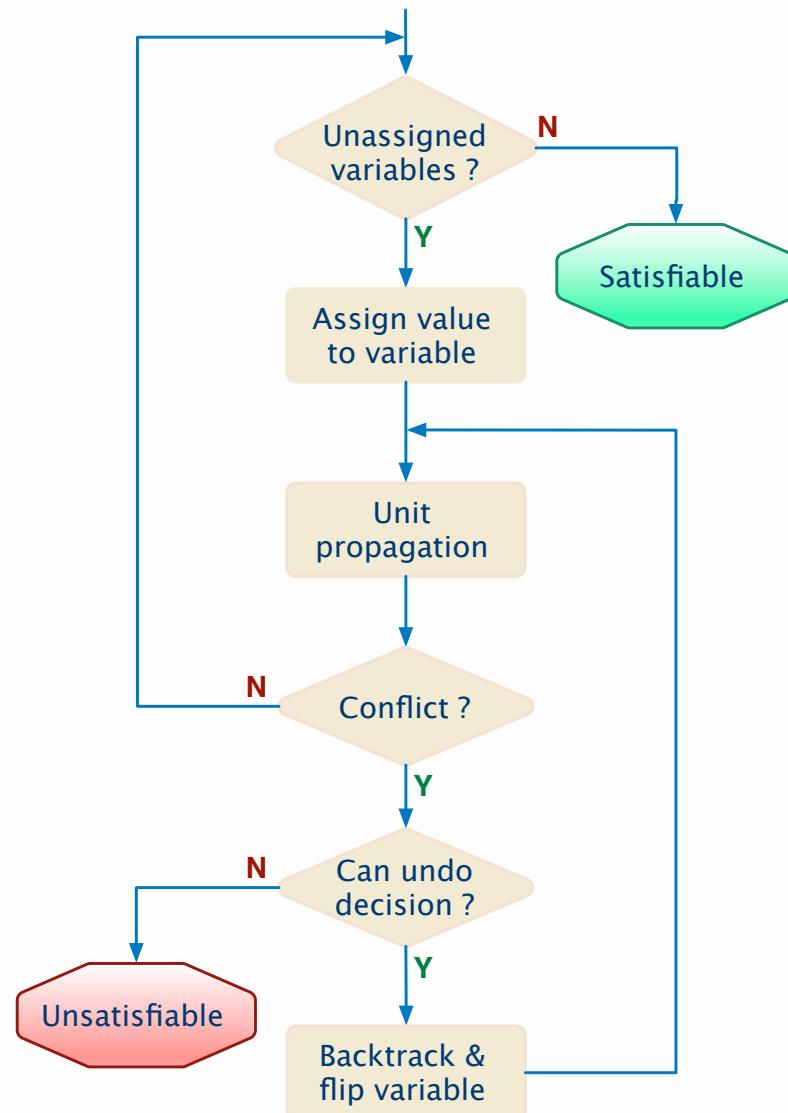
Basic Definitions

DPLL Solvers

CDCL Solvers

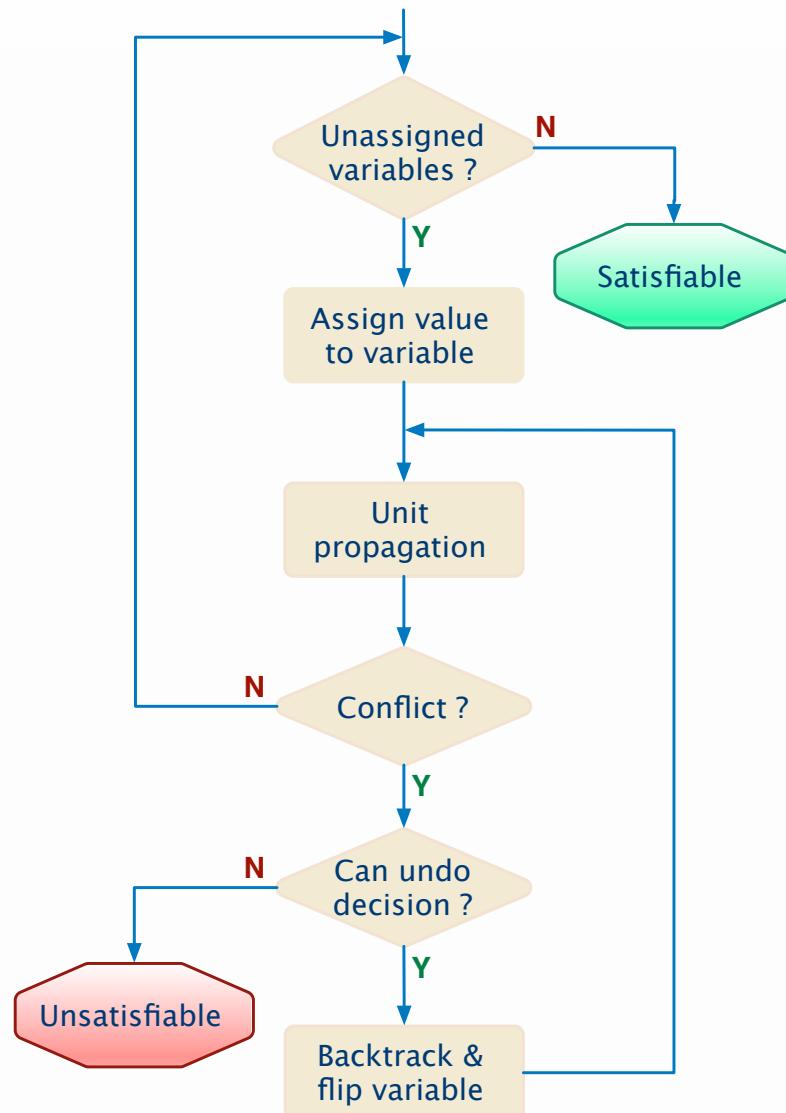
What Next in CDCL Solvers?

The DPLL Algorithm



- Optional: pure literal rule

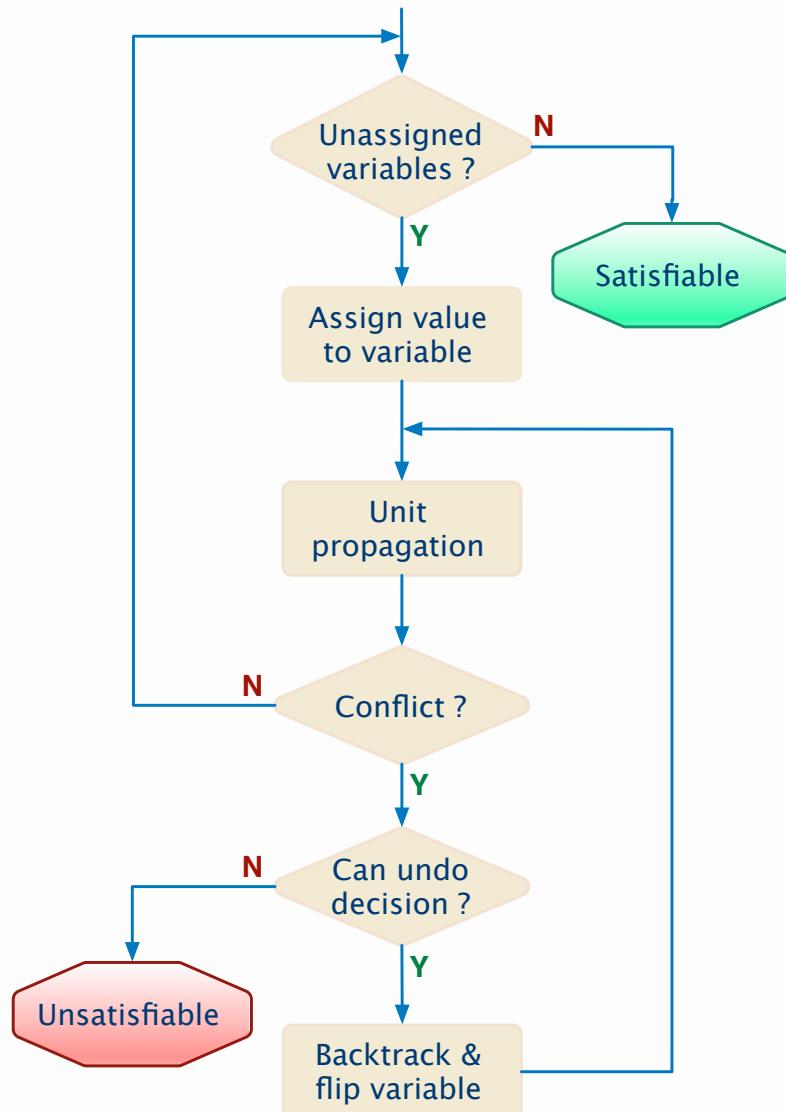
The DPLL Algorithm



$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

- Optional: pure literal rule

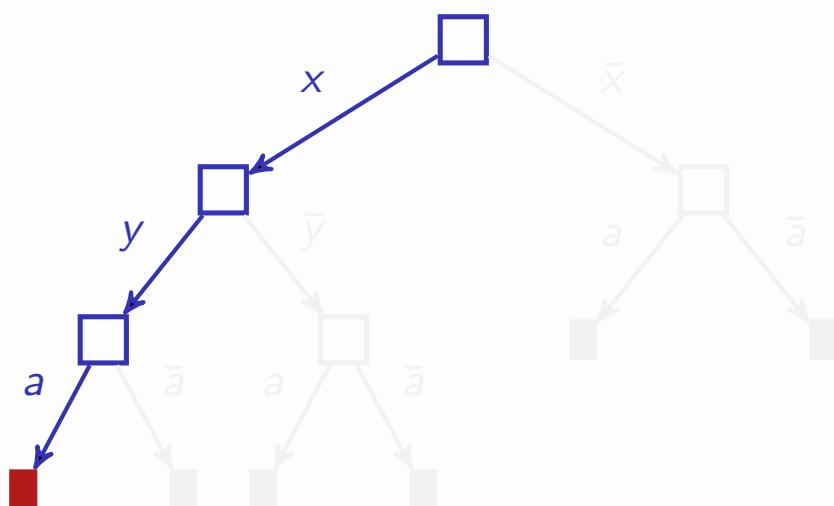
The DPLL Algorithm



$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

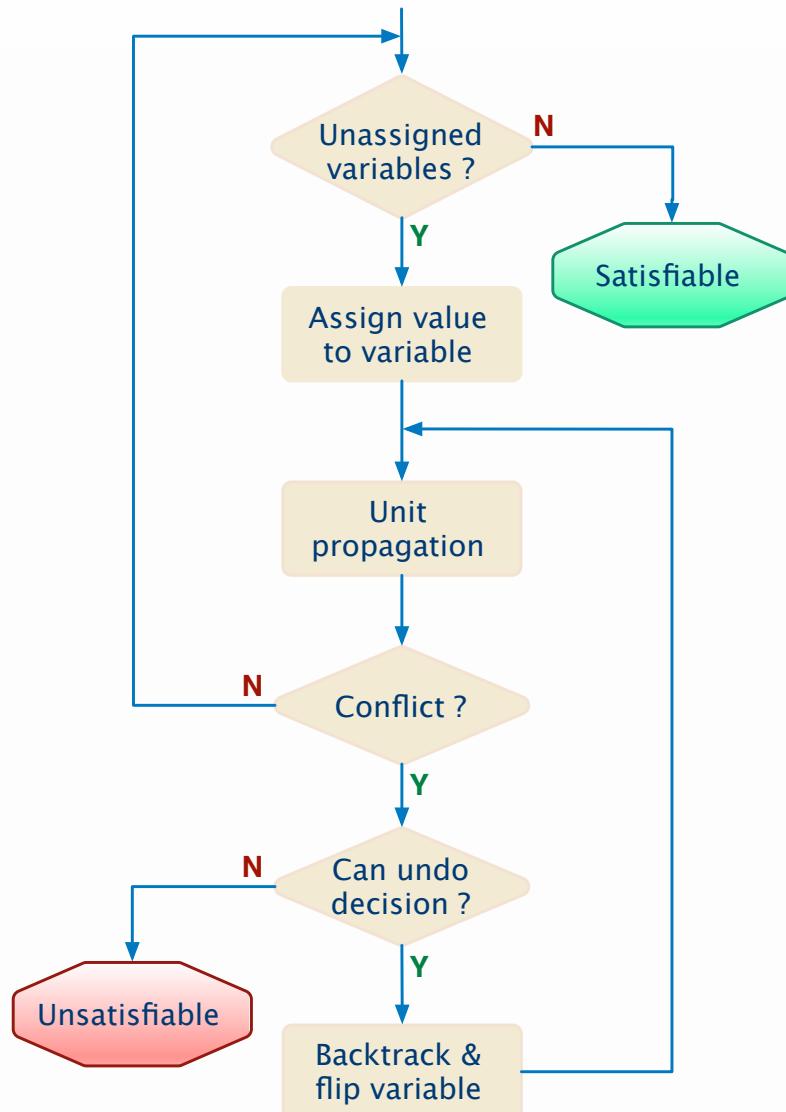
Level Dec. Unit Prop.

0	\emptyset	
1	x	
2	y	
3	a	$a \rightarrow b \rightarrow \perp$



- Optional: pure literal rule

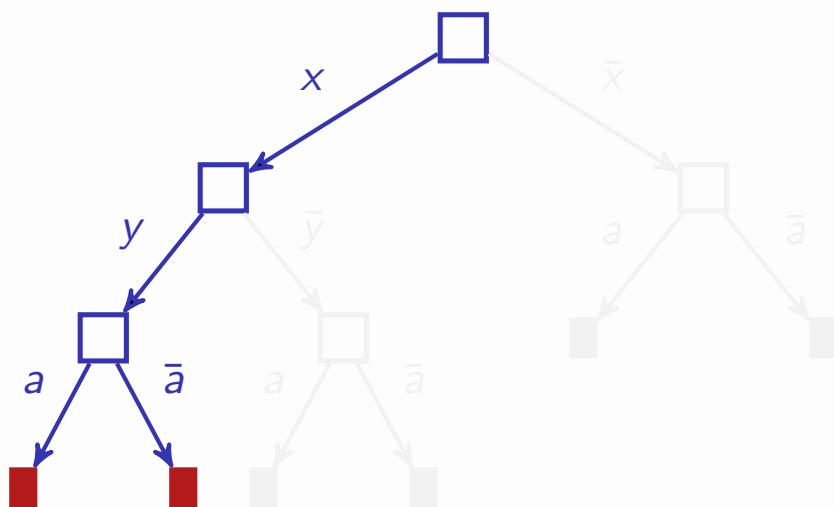
The DPLL Algorithm



$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

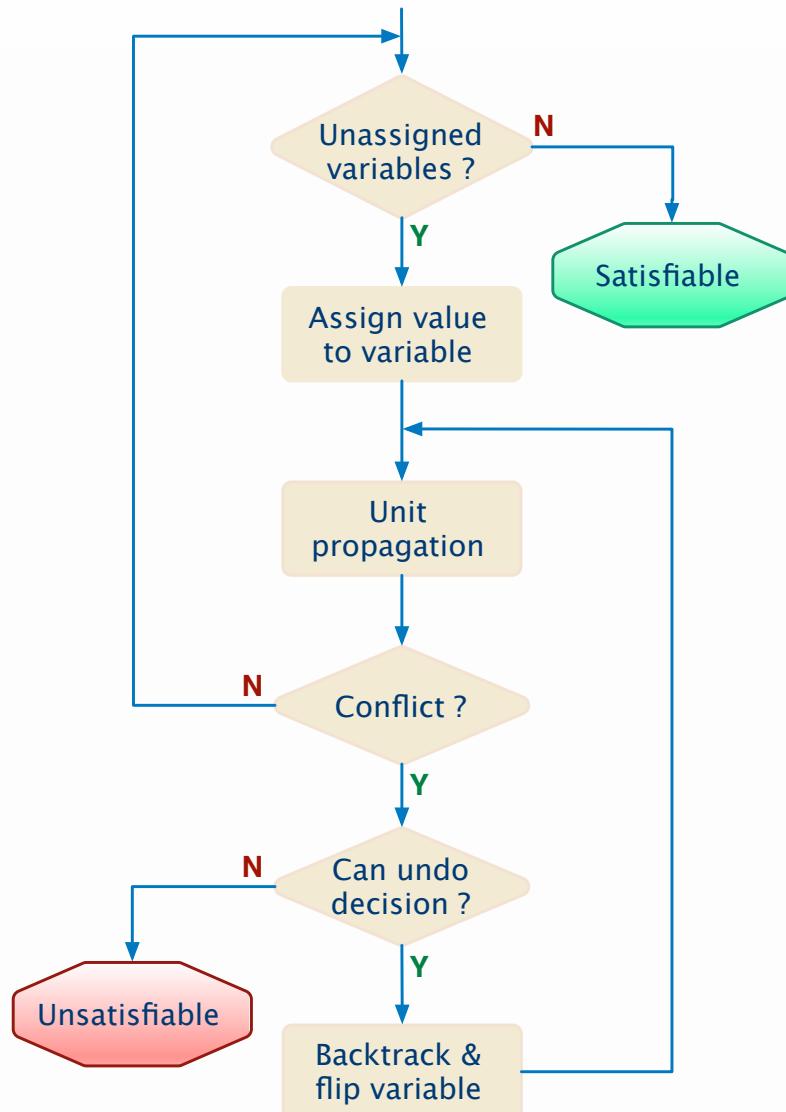
Level Dec. Unit Prop.

0	\emptyset	
1	x	
2	y	
3	\bar{a}	$\bar{b} \rightarrow \perp$



- Optional: pure literal rule

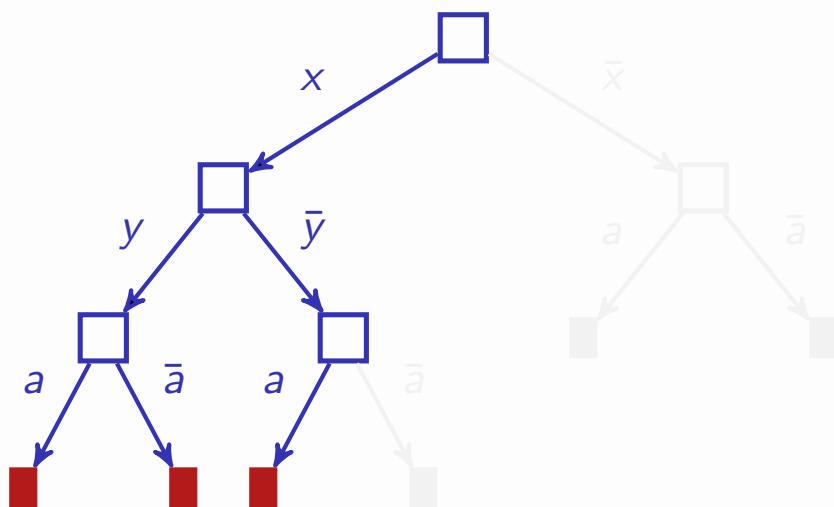
The DPLL Algorithm



$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

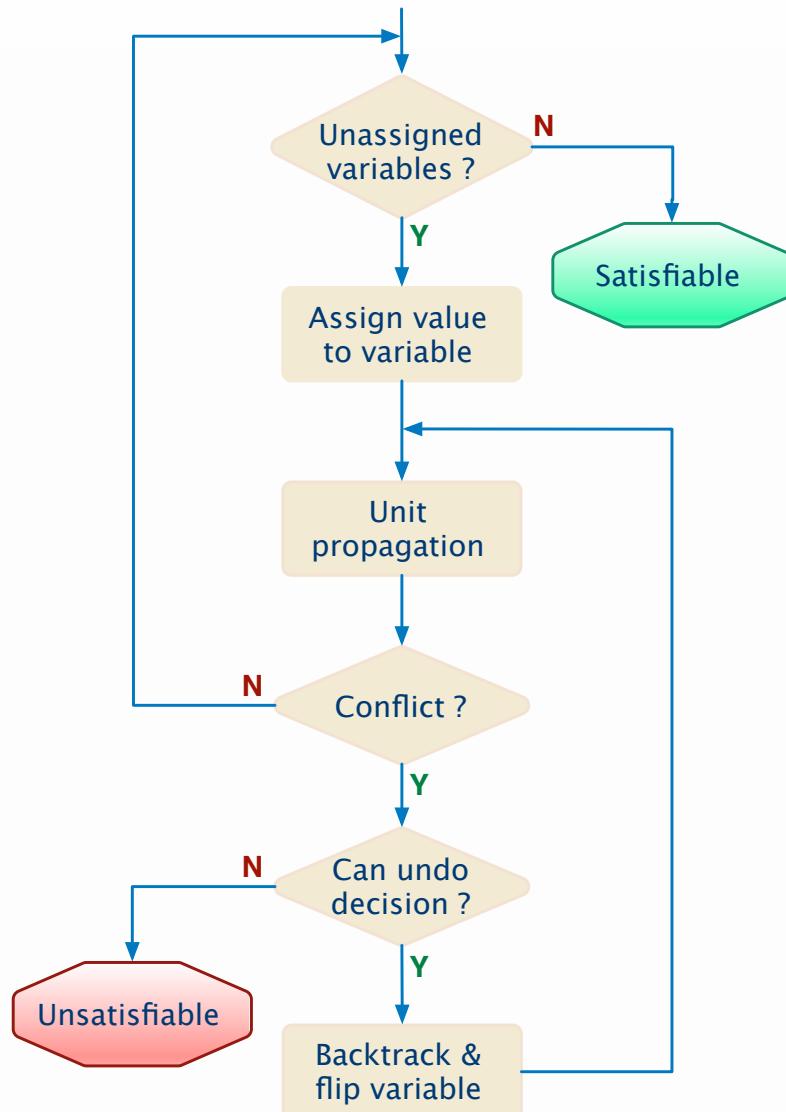
Level Dec. Unit Prop.

0	\emptyset	
1	x	
2	\bar{y}	
3	a	$b \rightarrow \perp$



- Optional: pure literal rule

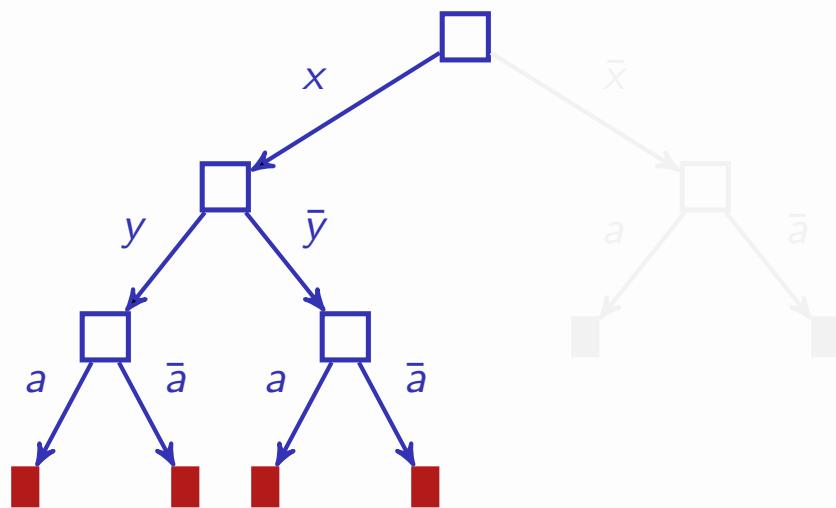
The DPLL Algorithm



$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

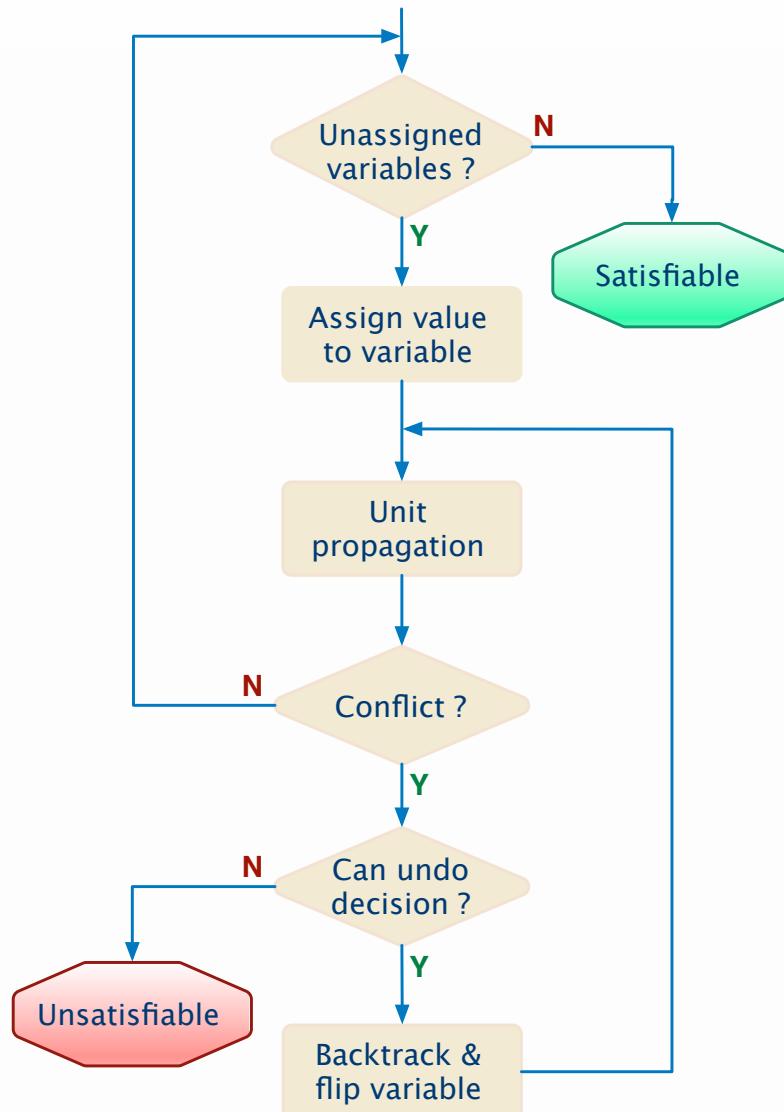
Level Dec. Unit Prop.

0	\emptyset	
1	x	
2	\bar{y}	
3	\bar{a}	$\bar{b} \rightarrow \perp$



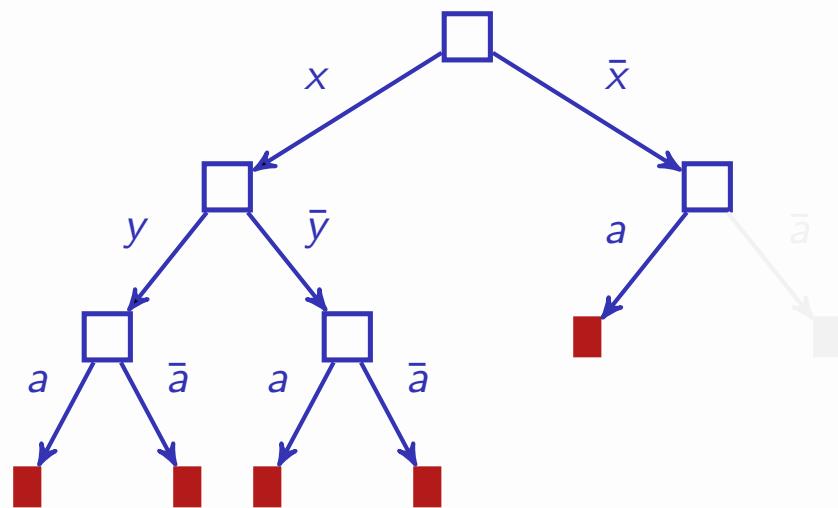
- Optional: pure literal rule

The DPLL Algorithm



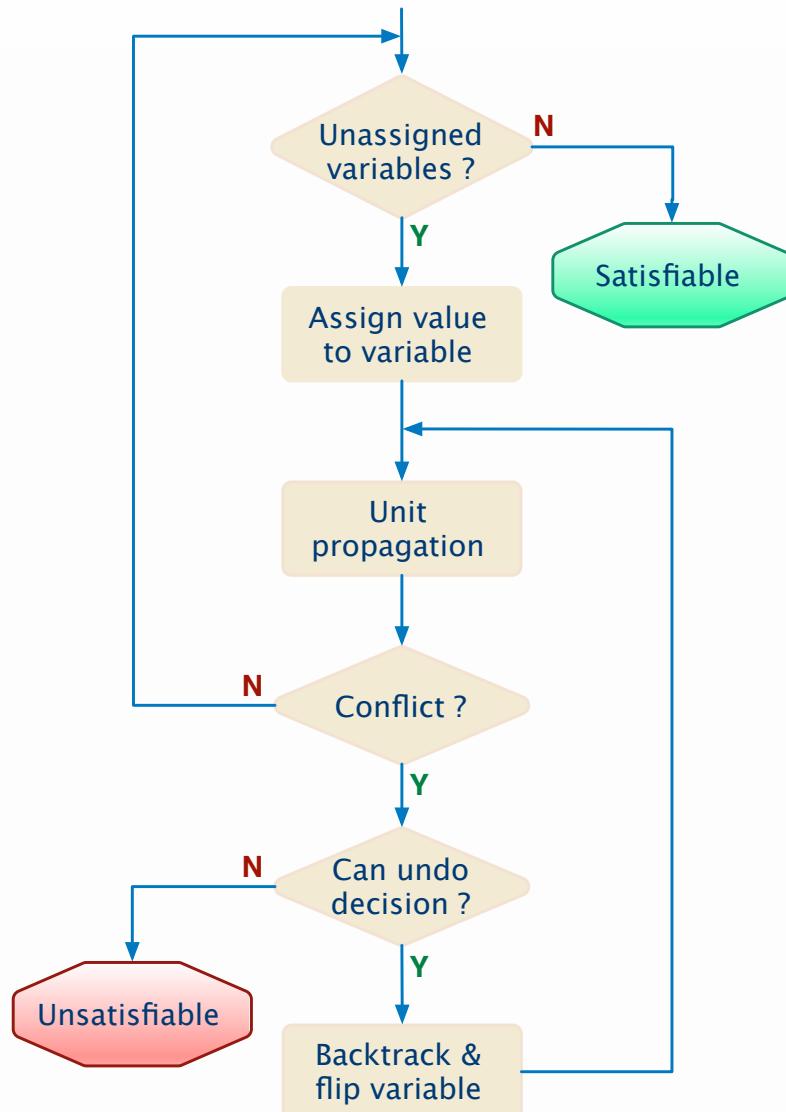
$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

Level	Dec.	Unit Prop.
0	\emptyset	
1	\bar{x}	$\bar{x} \rightarrow y$
2	a	$a \rightarrow b \rightarrow \perp$



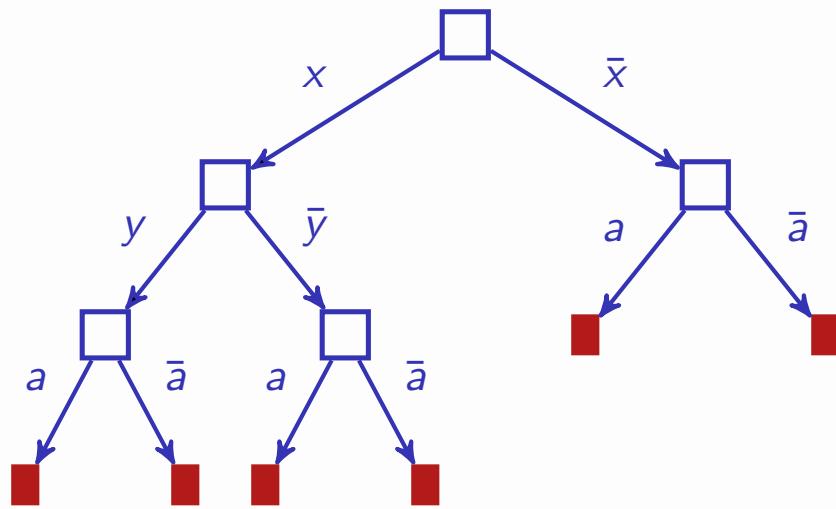
- Optional: pure literal rule

The DPLL Algorithm



$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

Level	Dec.	Unit Prop.
0	\emptyset	
1	\bar{x}	$\bar{x} \rightarrow y$
2	\bar{a}	$\bar{a} \rightarrow \bar{b}$ $\bar{b} \rightarrow \perp$



- Optional: pure literal rule

Outline

Basic Definitions

DPLL Solvers

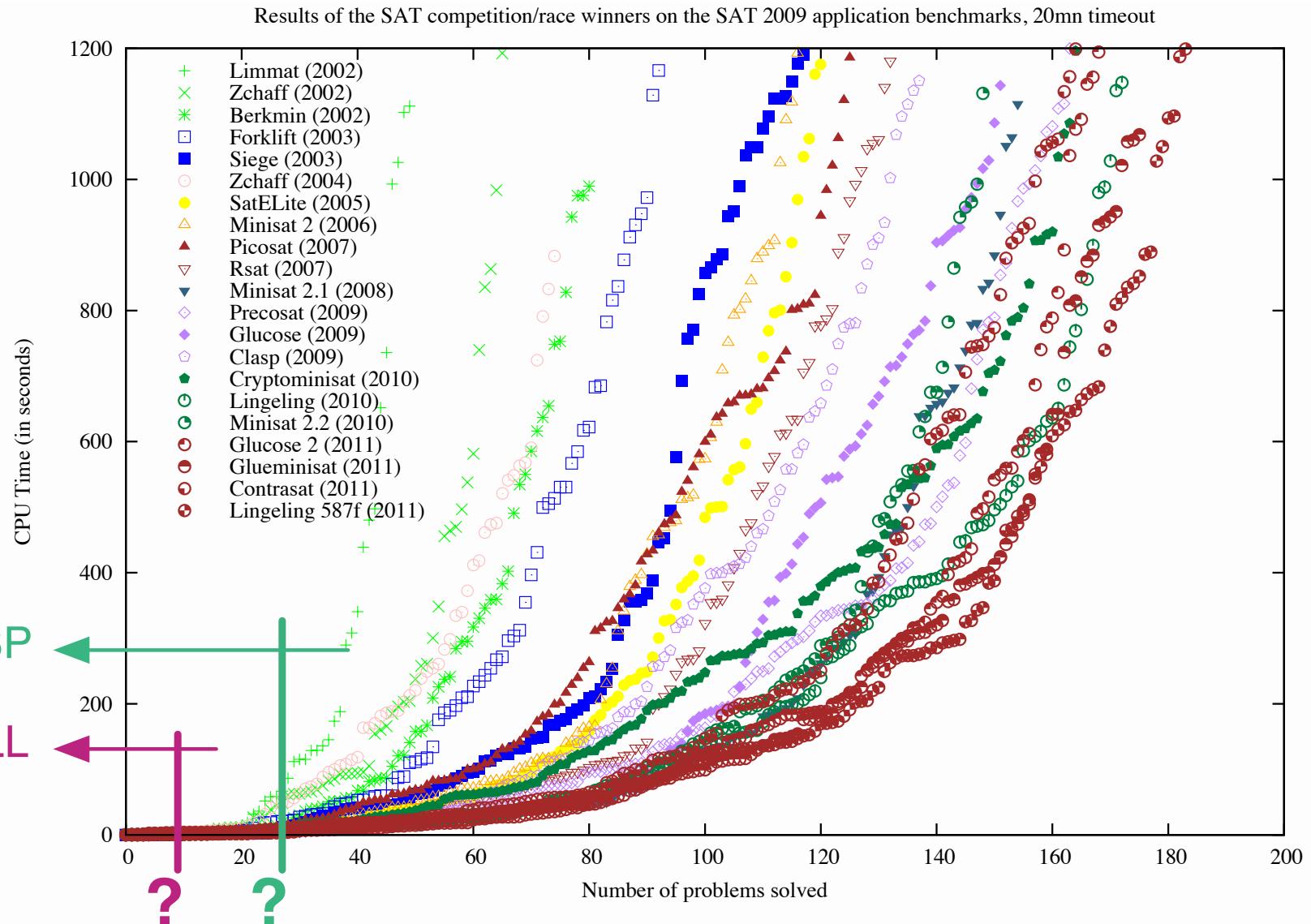
CDCL Solvers

What Next in CDCL Solvers?

What is a CDCL SAT Solver?

- Extend DPLL SAT solver with:
 - Clause learning & non-chronological backtracking
 - ▶ Exploit UIPs [MSS96, SSS12]
 - ▶ Minimize learned clauses [SB09, VG09]
 - ▶ Opportunistically delete clauses [MSS96, MSS99, GN02]
 - Search restarts [GSK98, BMS00, H07, B08]
 - Lazy data structures
 - ▶ Watched literals [MMZZM01]
 - Conflict-guided branching
 - ▶ Lightweight branching heuristics [MMZZM01]
 - ▶ Phase saving [PD07]
 - ...

How Significant are CDCL SAT Solvers?



Outline

Basic Definitions

DPLL Solvers

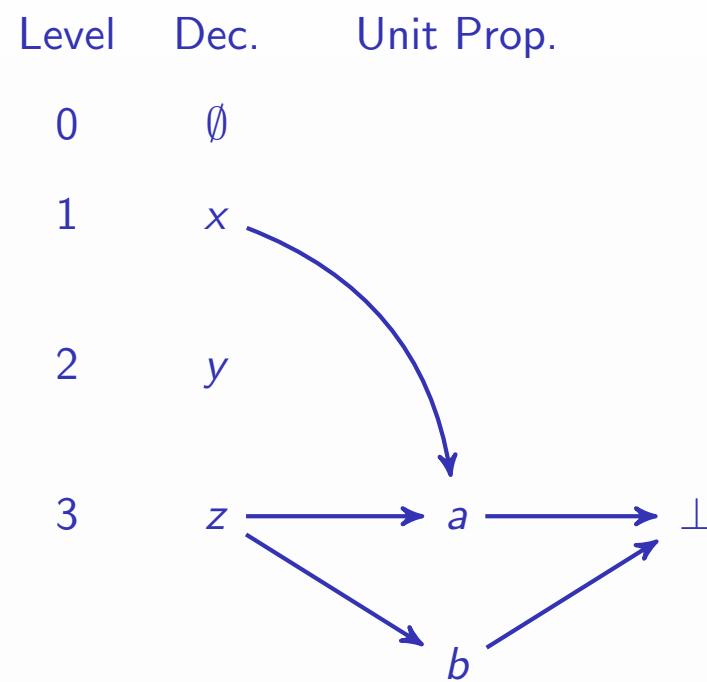
CDCL Solvers

Clause Learning, UIPs & Minimization

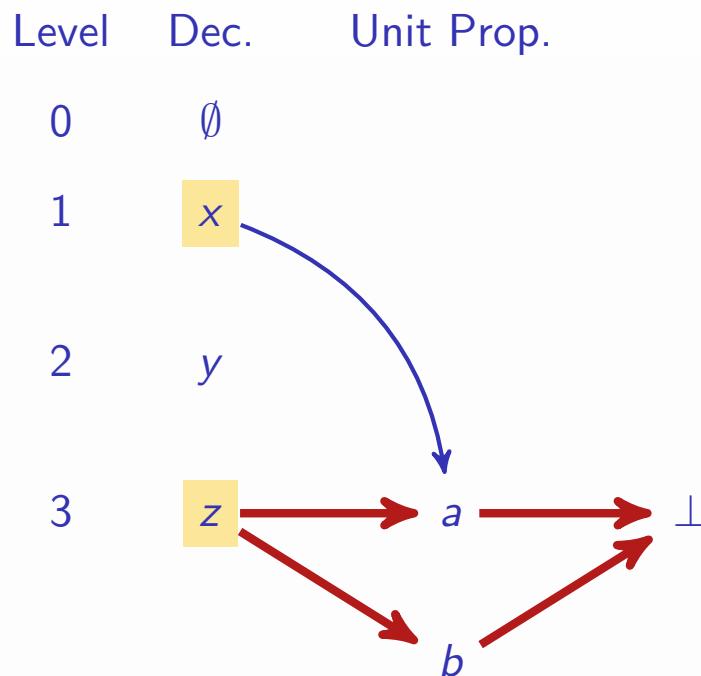
Search Restarts & Lazy Data Structures

What Next in CDCL Solvers?

Clause Learning



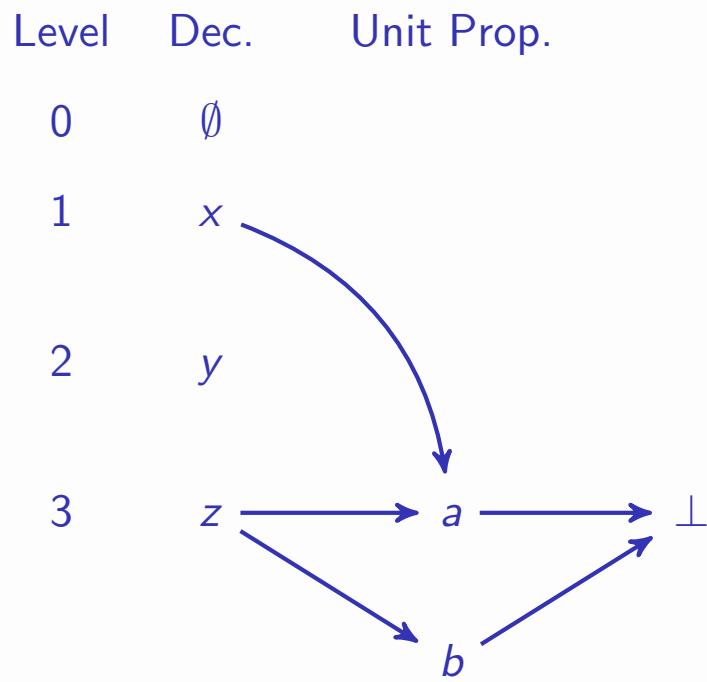
Clause Learning



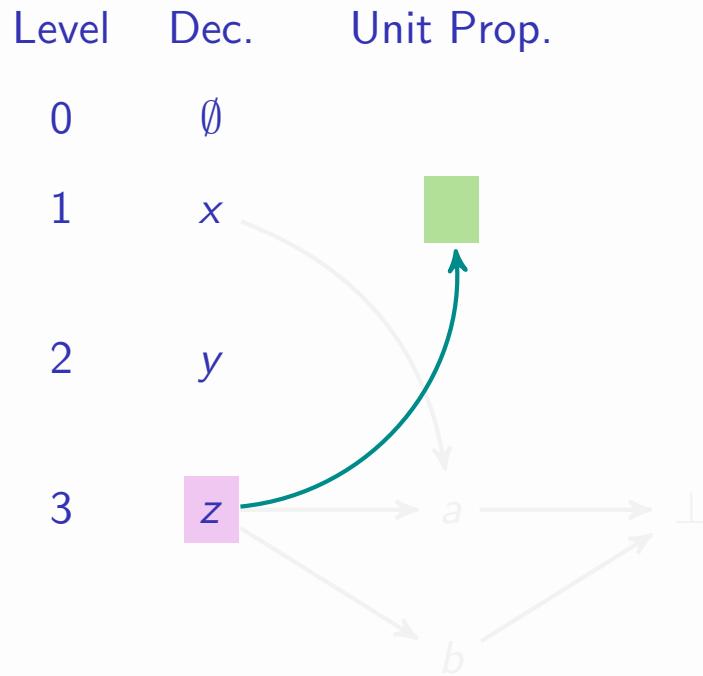
$$\begin{aligned} ab \rightarrow \text{false} &= !(ab) + \text{false} = !a + !b \\ z \rightarrow b &= !z + b \\ xz \rightarrow a. &= !(xz) + a = !x + !z + a \end{aligned}$$

- Analyze conflict
 - Reasons: x and z
 - ▶ Decision variable & literals assigned at lower decision levels
 - Create new clause: $(\bar{x} \vee \bar{z})$
- Can relate clause learning with resolution

Clause Learning – After Bracktracking

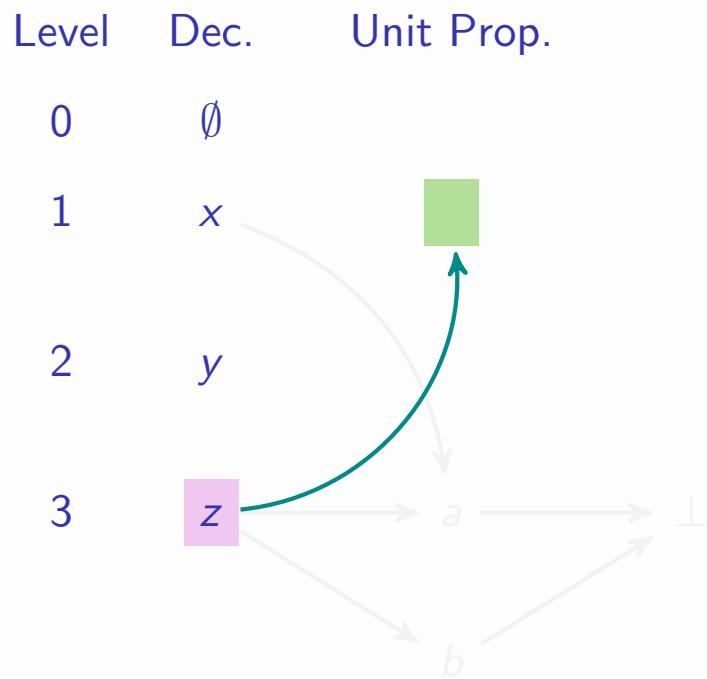


Clause Learning – After Backtracking



- Clause $(\bar{x} \vee \bar{z})$ is **asserting** at decision level 1

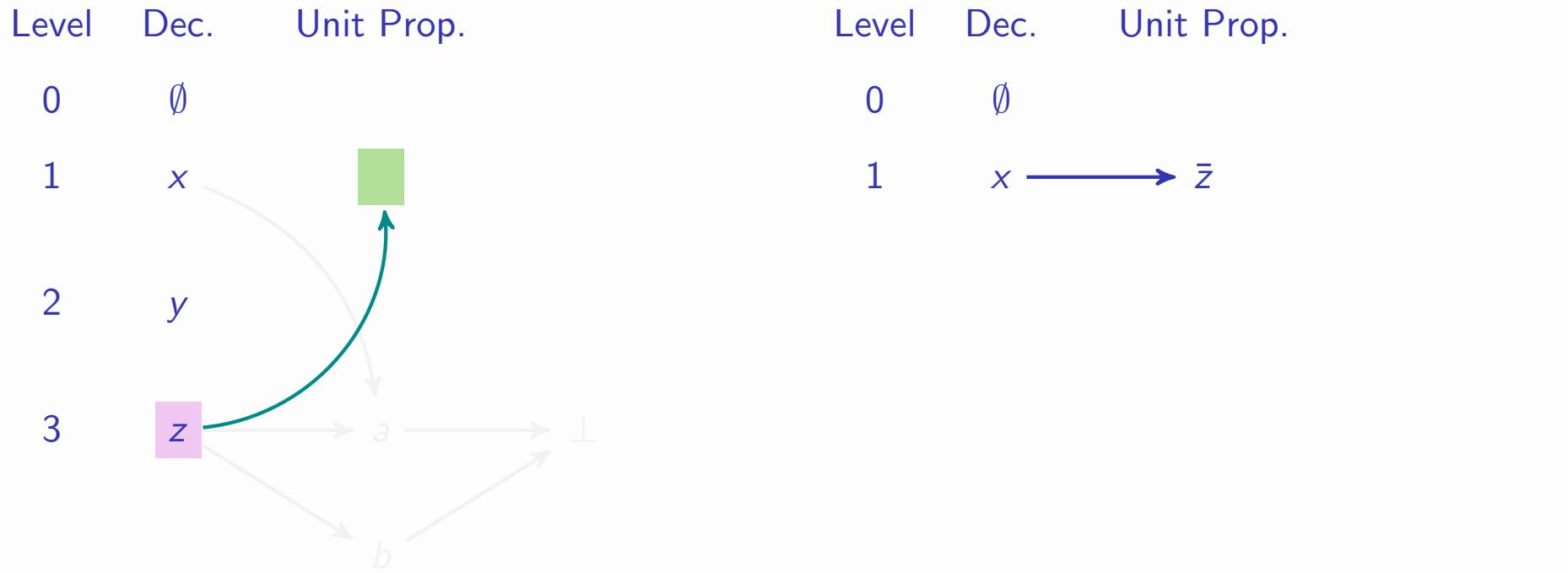
Clause Learning – After Backtracking



Level	Dec.	Unit Prop.
0	\emptyset	
1	x	\bar{z}

- Clause $(\bar{x} \vee \bar{z})$ is **asserting** at decision level 1

Clause Learning – After Backtracking



- Clause $(\bar{x} \vee \bar{z})$ is **asserting** at decision level 1
- Learned clauses are **always** asserting
- Backtracking differs from plain DPLL:
 - Always backtrack after a conflict

[MSS96, MSS99]

[MMZZM01]

CIS 4930 Digital Circuit Testing

Functional Testing

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

Introduction

- Testing for SSFs is based on structural model.
 - May not be available, or
 - May be too complex.
 - Physical fault testing cannot check design errors.
- Functional testing is based on functional model.
 - Specifies **design functionality**.
 - Check **physical faults + design errors**.

Introduction – cont'd

- **Objective:** validate system implementation wrt its functional specification.
- Functional testing without fault models
 - Generate tests wrt fault free behavior
- Functional testing using specific fault models
 - Directed tests
- Exhaustive and pseudoexhaustive testing

Functional Testing without Fault Models

- Exercise specified functions.
- Ex: to test D-FFs, need to validate that
 - It can set and reset, and
 - It can hold values.
- Coverage Problem: more difficult to evaluate the quality of tests with fault models.

Heuristics for Coverage

→ Operation activation

if x **then** operation1 **else** operation2

→ A “complete” test should exercise both branches.

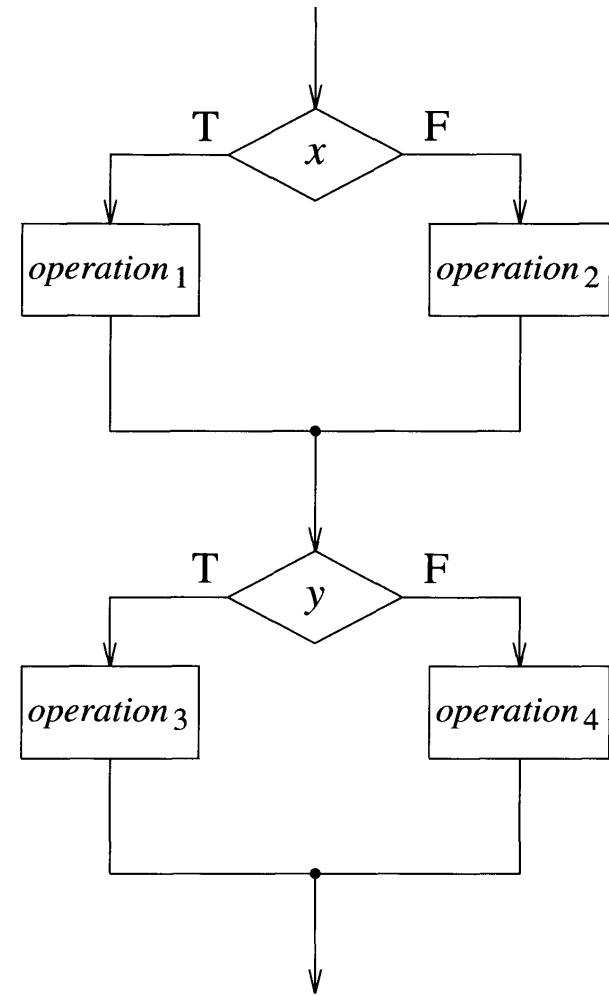
→ Decision path tracing

if y **then** operation3 **else** operation4

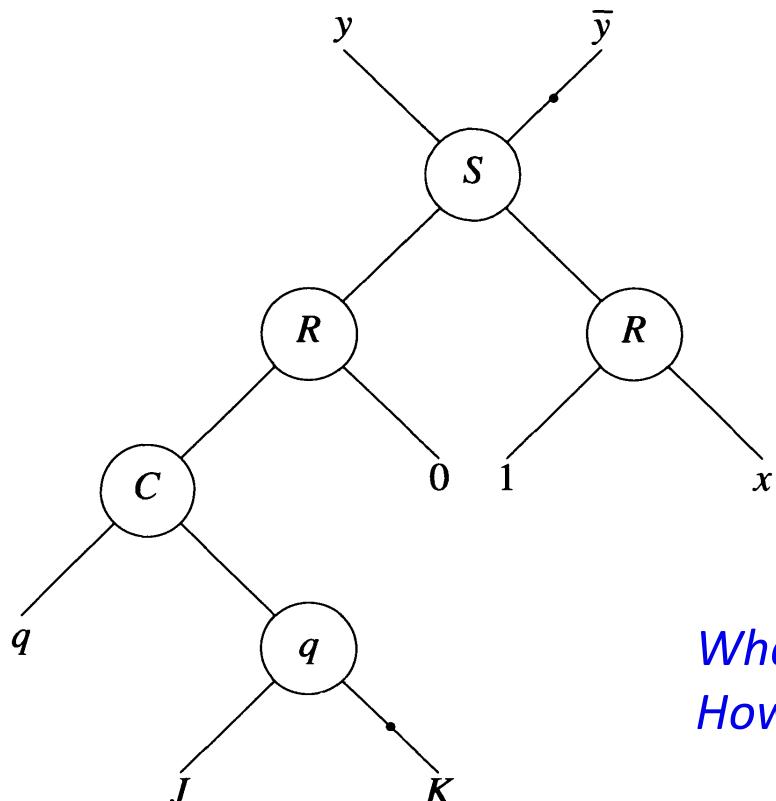
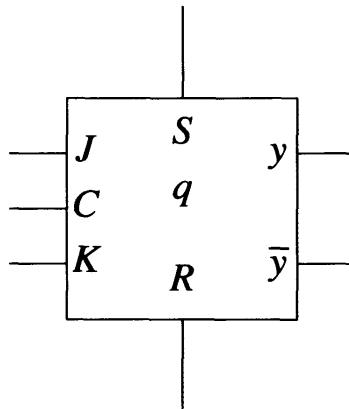
→ There are 4 paths considering those two statement

Heuristics for Coverage

- **Decision path tracing:**
 - coverage measure by the ratio of paths traversed vs total # of paths.
- **Checking unintended behavior:**
 - Possibility of writing data to R2 in addition to R1?



Functional Testing with BDDs



S	R	C	q	J	K	y
0	1	x	x	x	x	0
1	0	x	x	x	x	1
0	0	0	q	x	x	q
0	0	1	0	J	x	J
0	0	1	1	x	K	\bar{K}

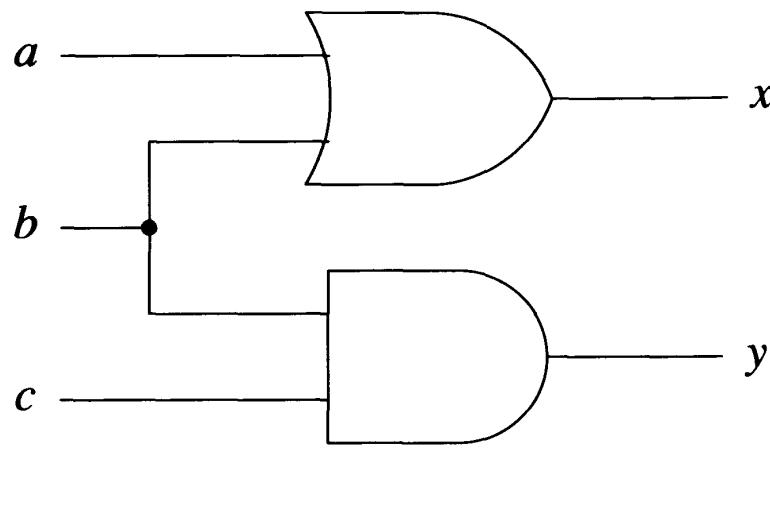
*What functions to test?
How to generate tests from BDDs?*

Exhaustive & PseudoExhaustive Testing

- Universal fault model
 - Any fault in a circuit is possible.
 - Any faults that changes a circuit's function
 - Need to apply all 2^n input vectors for n PIs.
 - only practical for small circuits.
- **PseudoExhaustive** Testing
 - Consider certain structural information
 - Significantly reduce the input vectors by circuit partitioning wrt POs.

Partial-Dependence Combinational Circuits

- Definition: No PO depends on all PIs.
- Only need 2^{n_i} input vectors for PO O_i with n_i PIs.



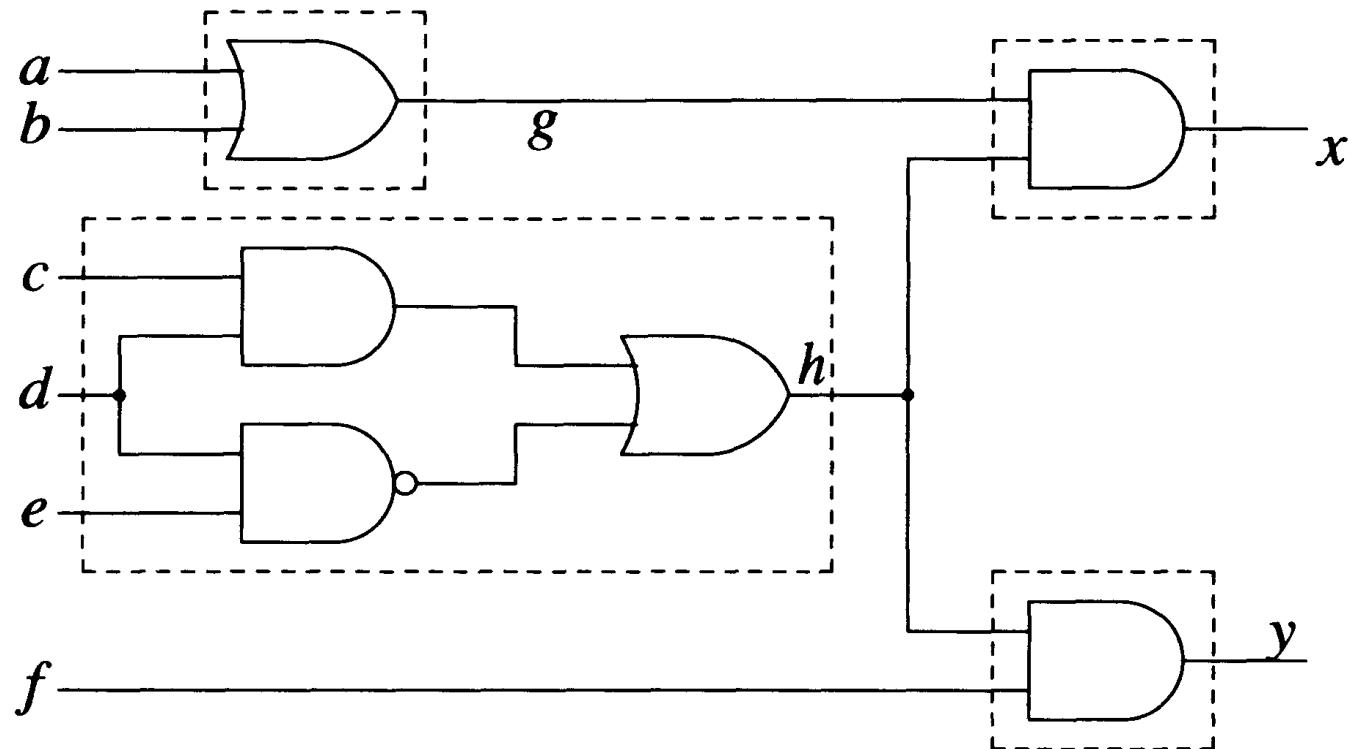
a	b	c
0	0	0
0	1	0
1	0	1
1	1	1

(b)

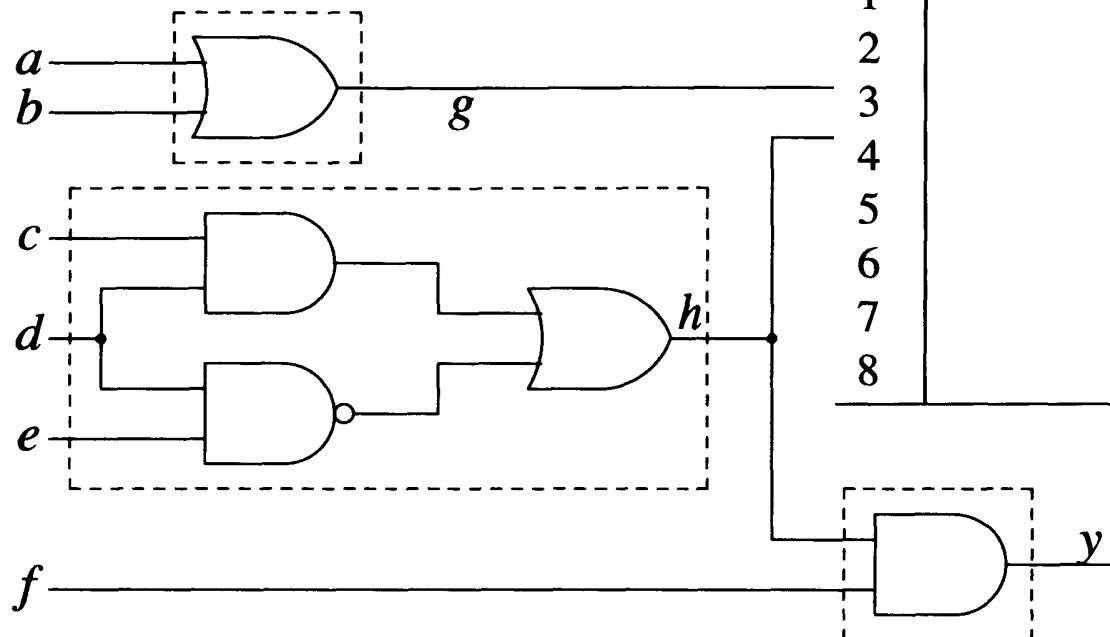
Circuit Partitioning

- Pseudo-exhaustive testing still not practical for large n ;
 - Or total dependence circuits
- Circuit partitioned into *segments* with limited # of inputs.
- If inputs/outputs of a segment are not PIs/POs,
 - Need to control segment's inputs from PIs,
 - Need to observe segment's outputs on POs.

Circuit Partitioning



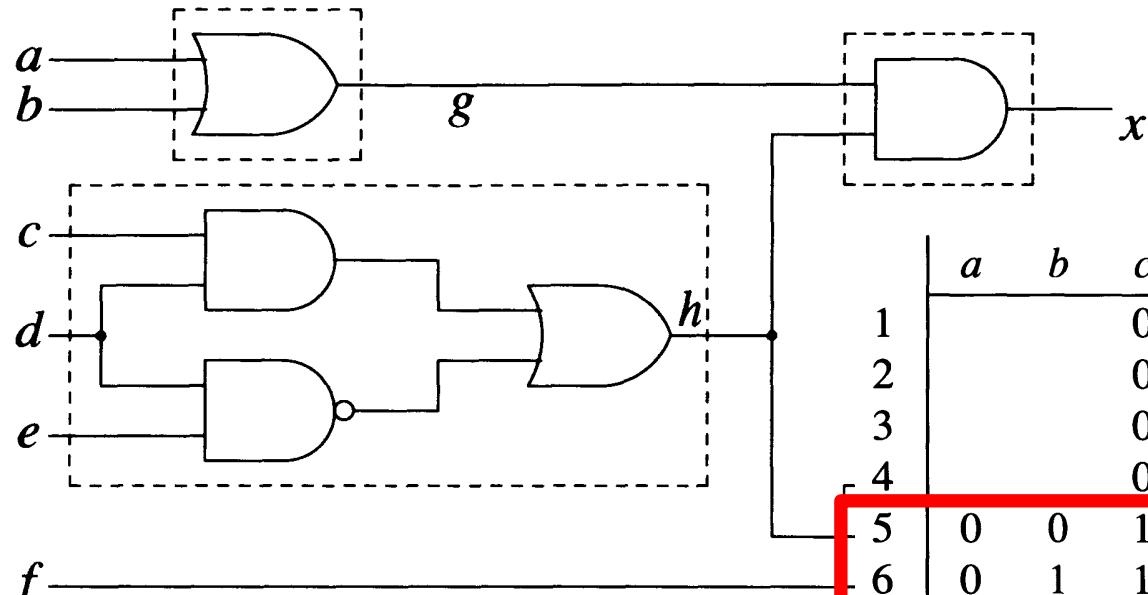
Circuit Partitioning – Vectors for Testing h



	a	b	c	d	e	f	g	h	x	y
1							0	0	0	1
2							0	0	1	1
3							0	1	0	1
4							0	1	1	1
5							1	0	0	1
6							1	0	1	1
7							1	1	0	1
8							1	1	1	1

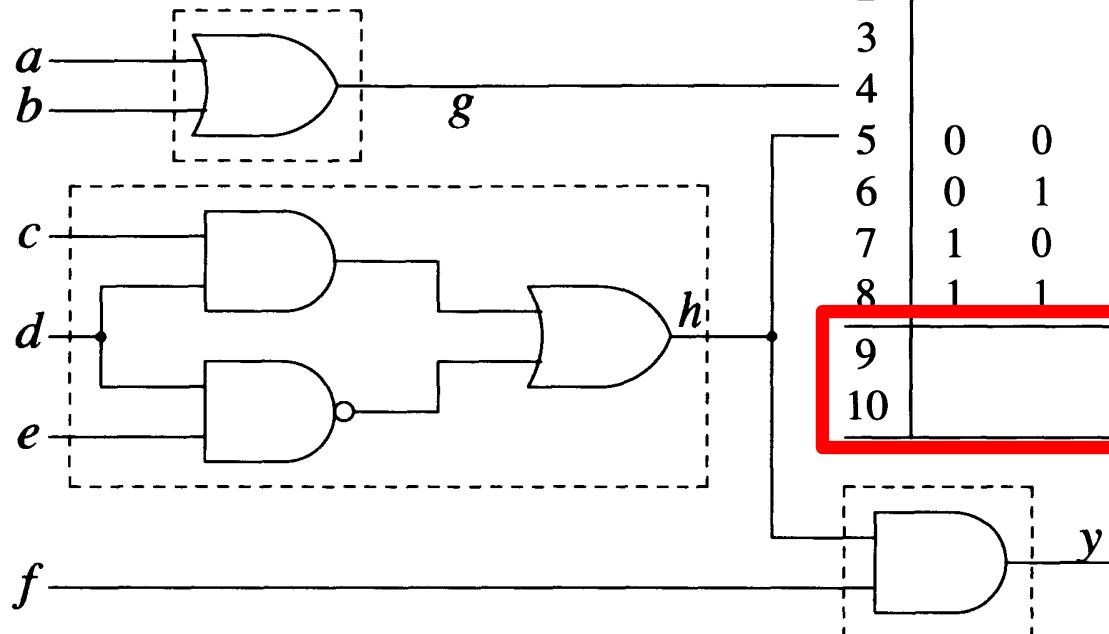
Test vectors for h which is observable at y

Circuit Partitioning – Vectors for Testing g



	a	b	c	d	e	f	g	h	x	y
1	1			0	0	0	1		1	1
2	2			0	0	1	1		1	1
3	3			0	1	0	1		1	1
4	4			0	1	1	1		0	0
5	5	0	0	1	0	0	1	0	1	0
6	6	0	1	1	0	1	1	1	1	1
7	7	1	0	1	1	0	1	1	1	1
8	8	1	1	1	1	1	1	1	1	1
9	9			0	1	1	0		0	0
10	10			0	0	0	0		1	0

Circuit Partitioning – Vectors for Testing y



	a	b	c	d	e	f	g	h	x	y
1				0	0	0	1		1	1
2				0	0	1	1		1	1
3			0	1	0	1			1	1
4			0	1	1	1		0	0	0
5	0	0	1	0	0	1	0	1	0	1
6	0	1	1	0	1	1	1	1	1	1
7	1	0	1	1	0	1	1	1	1	1
8	1	1	1	1	1	1	1	1	1	1
9				0	1	1	0		0	0
10				0	0	0	0	1		0

Testing Sequential Circuits

- **Fault assumption:** state table modified w/o increasing # of states
 - Add/remove state transitions
- **Problem:** finding input sequences that distinguish a circuit with n states from all other n -state circuits
 - Such sequences exist for reduced and strong connected sequential circuits

Testing Sequential Circuits – Three Phases

- 1: **initialization** – bring circuit to a known state
- 2: Verify that circuit has n states
- 3: Verify that every entry in the state table

Functional Fault Models

- Functional faults represent effects of physical faults on the functions of a system.
- Behavior due to functional faults should match the behavior due to physical faults.
- Tests to detect functional faults have high coverage for the SSFs in the structural model.

Example – Addressing Faults

- **Addressing decoding – Functions**
 - Addressing a word in memory
 - Selecting a register in processor
 - Decoding an instruction to determine operations to perform
- **Functional faults**
 - Selecting no item
 - selecting item i instead of j
 - Selecting item j in addition to item i .
- Test generation concerns generating a program that produces wrong results.

Summary

- Test if circuit functions are implemented correctly
- Functional Fault Model: effects of physical faults on functional behavior
- Functional testing:
 - Pseudo-exhaustive
 - Test generation w or w/o fault models

CIS 4930 Digital Circuit Testing

Design For Testability

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

Introduction

- Testing cost
 - Test gen., fault sim., test equipment, test process (fault detection and location), etc
- Testing should be considering during the design process
 - enhances “testability” & design quality
 - reduces test cost
- Test complexity determined by three factors
 - Controllability
 - Observability
 - Predictability – ability to obtain know outputs
- *Design for Testability (DFT)* techniques are design efforts to ensure that a device is testable

Introduction

- Testing cost
 - Test gen., fault sim., test equipment, test process (fault detection and location), etc
- Testing should be considering during the design process
- **Design for testability (DFT)** modifies design to
 - enhances “testability” & design quality
 - reduces test cost
- Testability involves
 - **Controllability**
 - **Observability**
 - **Predictability** – ability to obtain known outputs

Testability - Controllability

- **Controllability**
 - ability to establish a specific signal value at each node in the circuit by setting values on PIs
- Circuits difficult to control:
 - Decoders
 - Circuits with feedback
 - Oscillators
 - Clock generators
 - Counters (eg., 16 bit counter, how clock cycles will it take to force MSB to 1?)

Testability - Observability

- **Observability**
 - Ability to determine the signal value at any node in a circuit by controlling circuit's inputs and observing its outputs
- Circuits difficult to observe:
 - Sequential circuits
 - Circuits with global feedback
 - Embedded RAMs, ROMs, or PLAs
 - Circuits with redundant nodes

Some General Observations

- Sequential logic is more difficult to test than combinational logic
- Control logic is more difficult to test than data-path logic
- Random logic is more difficult to test than structured, bus-oriented designs
- Asynchronous designs is more difficult to test than synchronous designs

Tradeoffs

- DFT techniques often reduce costs to test.
 - Improved controllability and observability.
 - Reduced test time, test generation cost,
 - improved test quality -> product quality
- At meantime, they increase product cost.
 - Silicon area, I/O pins, power consumption, and circuit delay.
- Need to balance gain from DFT and its cost

Ad Hoc DFT Techniques

1. Test Points
2. Initialization
3. Monostable multivibrators (one-shots)
4. Oscillators and clocks
5. Counters/Shift Registers
6. Partitioning Large Circuits
7. Logical Redundancy
8. Breaking Global Feedback Paths

1 – Test Points

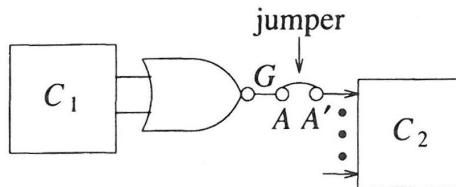
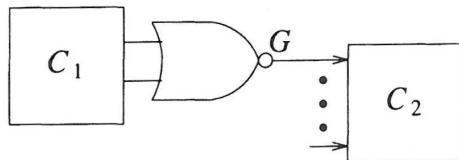
Employ test points to enhance controllability and observability

Two types of test points:

- Control points (CP) = PIs used to enhance controllability
- Observation points (OP) = POs used to enhance observability

Employing Test Points

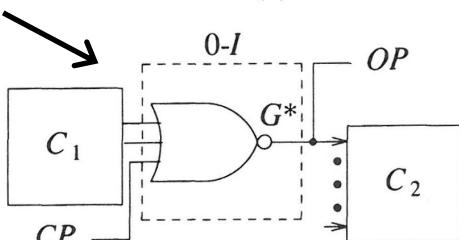
A is an OP
A' is a CP



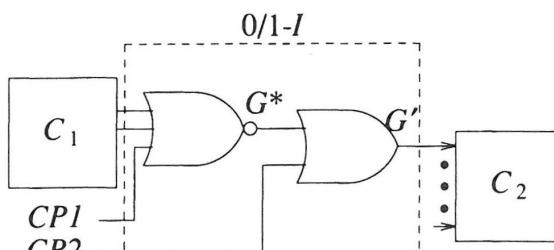
0 Injection Ckt

(a)

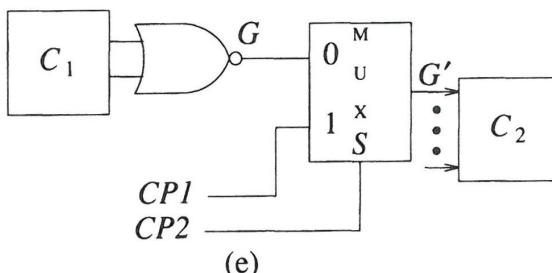
(b)



(c)



(d)

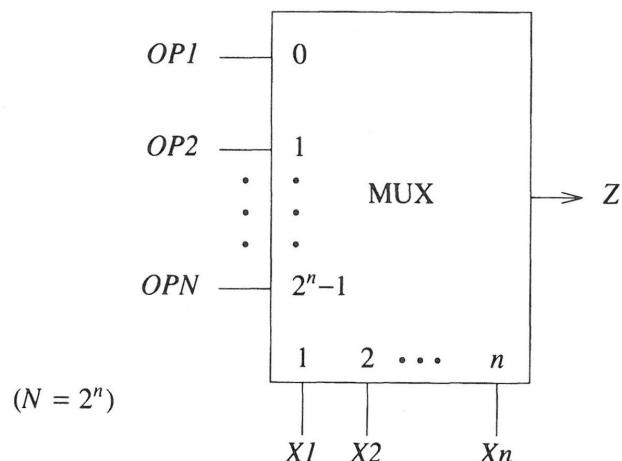


(e)

Demand of large # of
IO pins!

Multiplexing Monitor Points

- For limited IO pins, we can use multiplexer
- Drawback – can monitor only one OP at a time -> increases test time
- Select lines can be driven by a counter

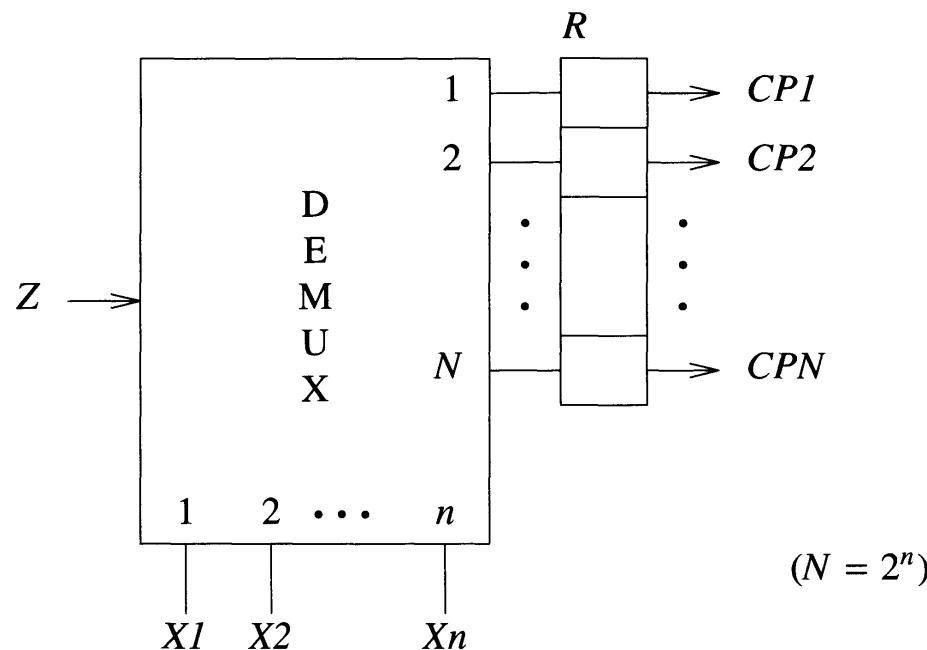


Tradeoff between test time and IO pins.

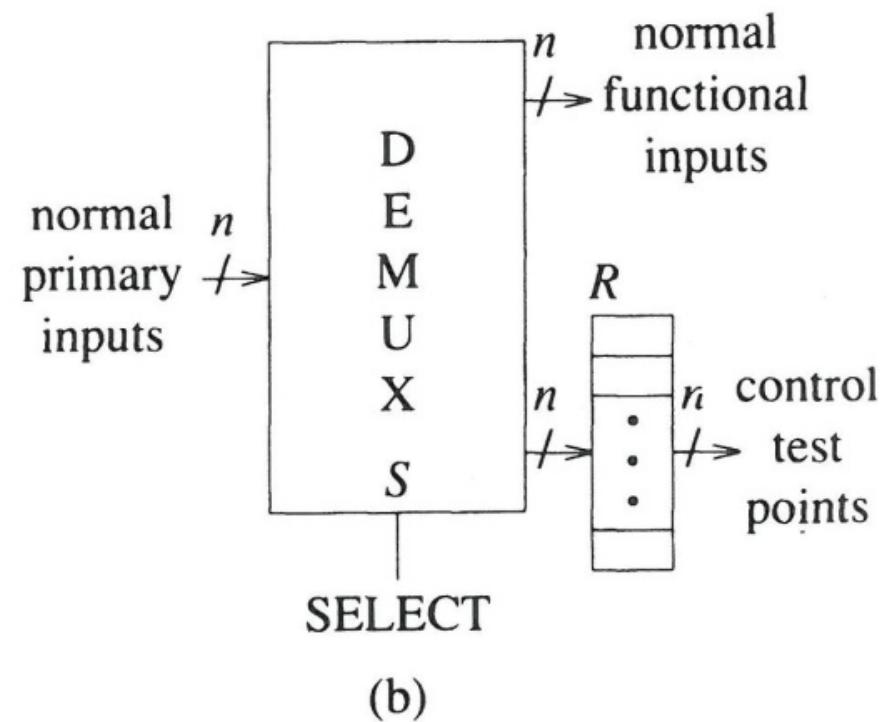
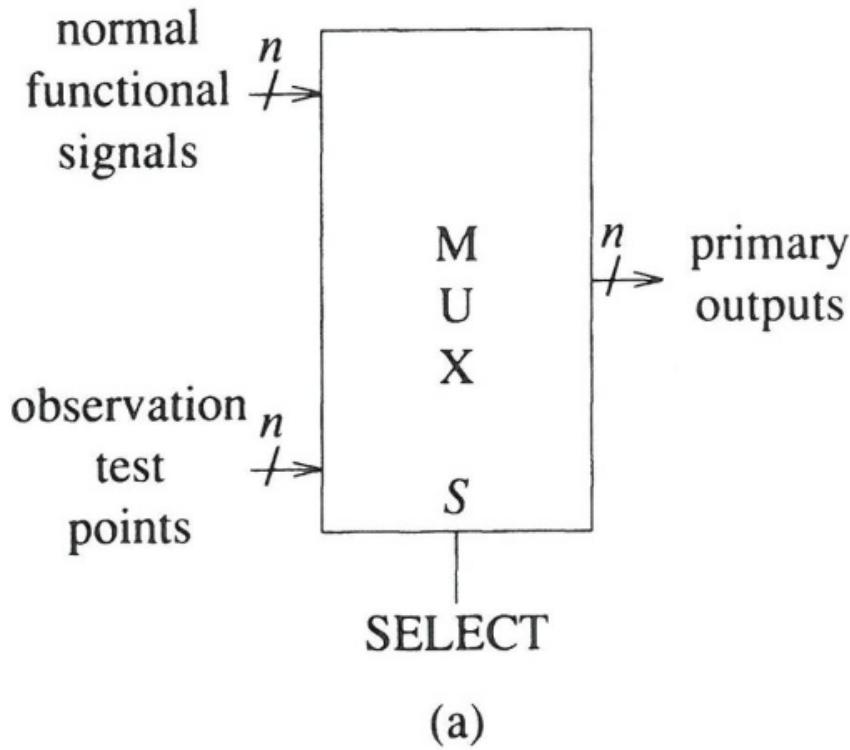
N clock cycles are required between test vectors

Demux/Latch for Multiple CPs

- Values of 2^n control points are serially applied to input Z
- Stored in N bit-wide latch
- Need N cycles to set up the control values



Time-Sharing Normal I/O Pins



Signal Selection – Control Points

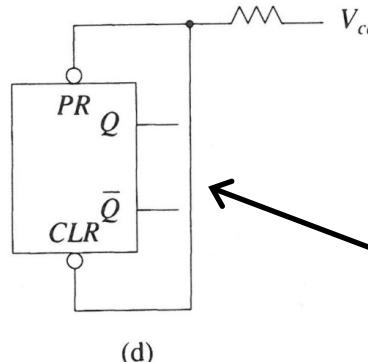
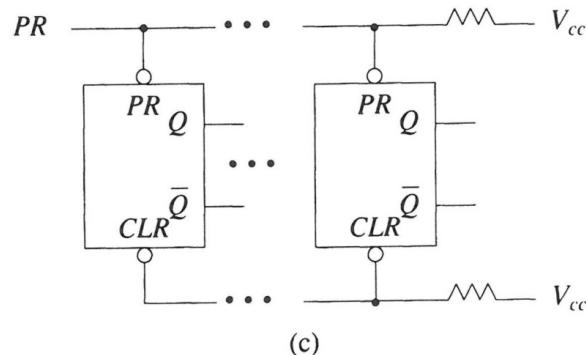
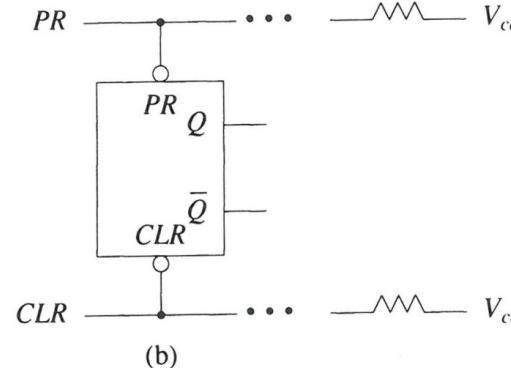
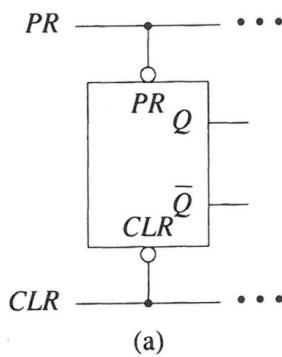
- Control, address, and data bus lines on bus-structured designs
- Enable/hold inputs to microprocessors
- Enable and read/write inputs to memory devices
- Clock and preset/clear inputs to memory devices
- Data select inputs to muxes/demuxes
- Control lines on tri-state devices

Signal Selection – Observation Points

- Stem lines associated with signals having high fanout
- Global feedback paths
- Redundant signal lines
- Outputs of logic devices having many inputs (muxes, parity generators)
- Outputs from state devices (FFs, Counters, Shift Registers)
- Address, control, and data busses

2 – Initialization

Design circuits to be easily initializable



Should be avoided!

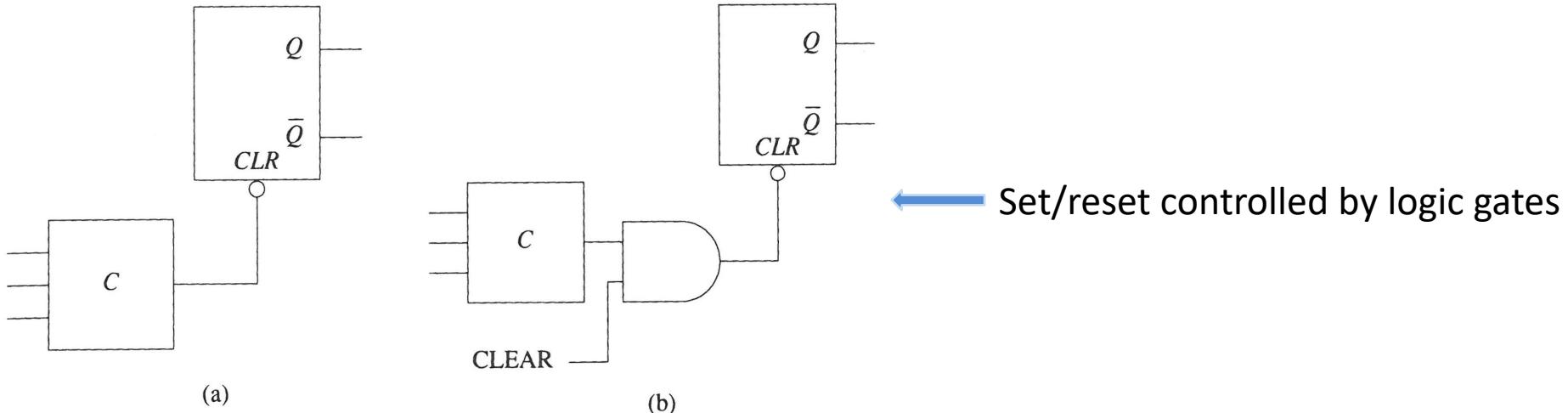


Figure 9.6 (a) Flip-flop without explicit clear (b) Flip-flop with explicit clear

Set/reset signal controlled internally →

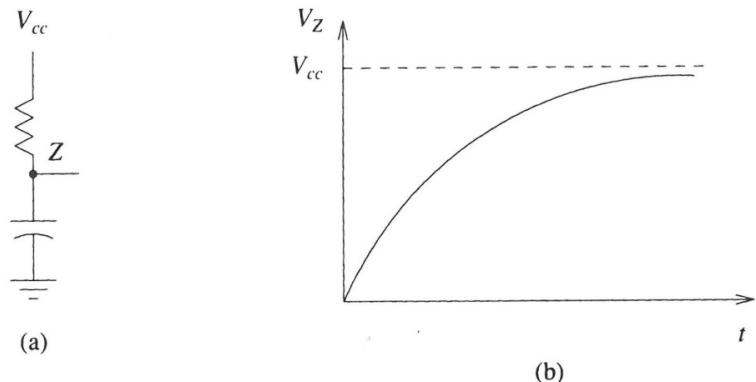
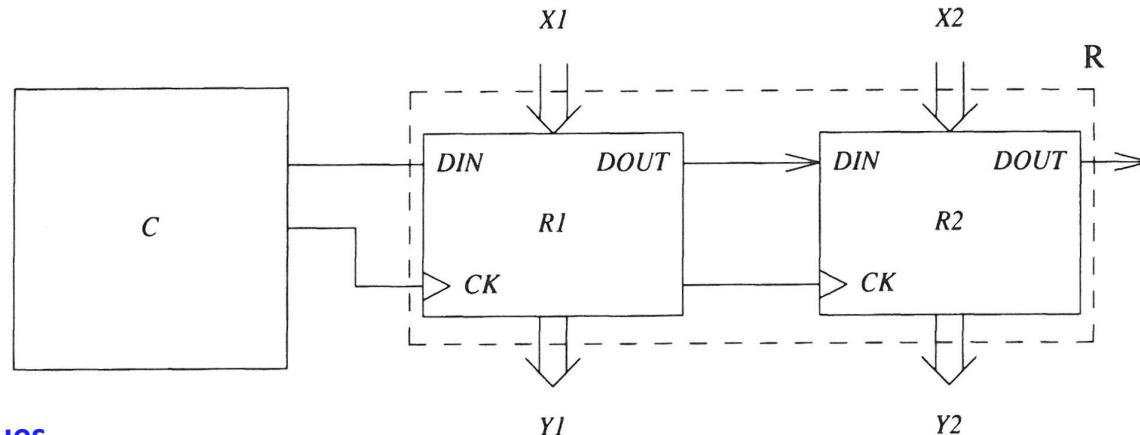


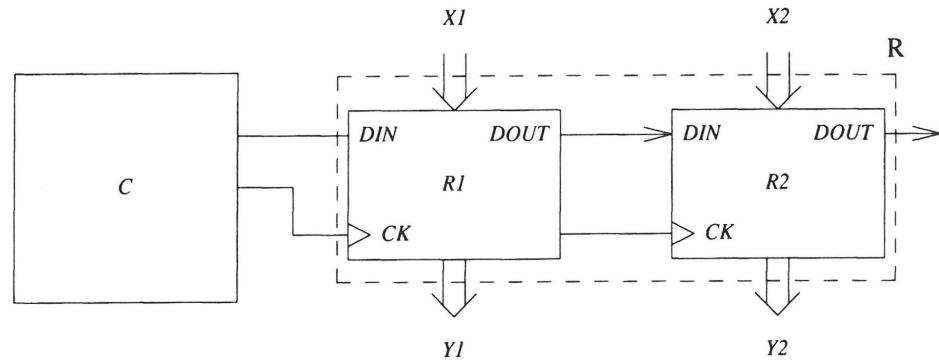
Figure 9.7 Built-in initialization signal generator

5 – Partitioning Counters and Shift Registers

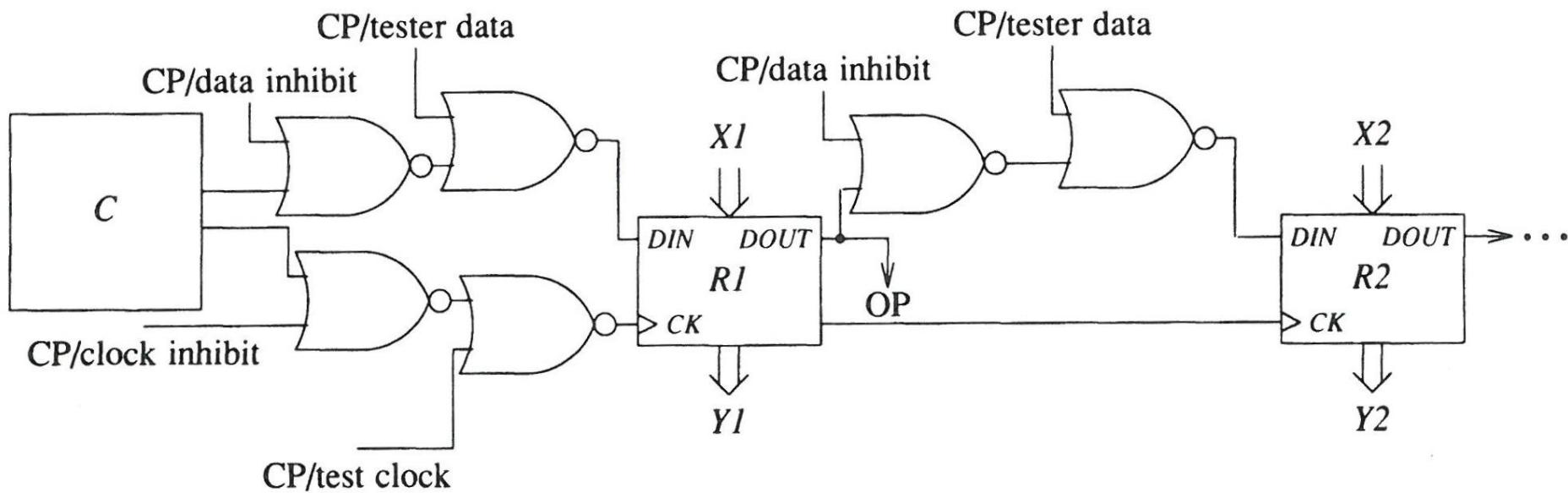
Partition large counters and shift registers into smaller units

- Counters/SR are difficult to test because test sequences usually require many clock cycles
- Partition the register for better control/obs.





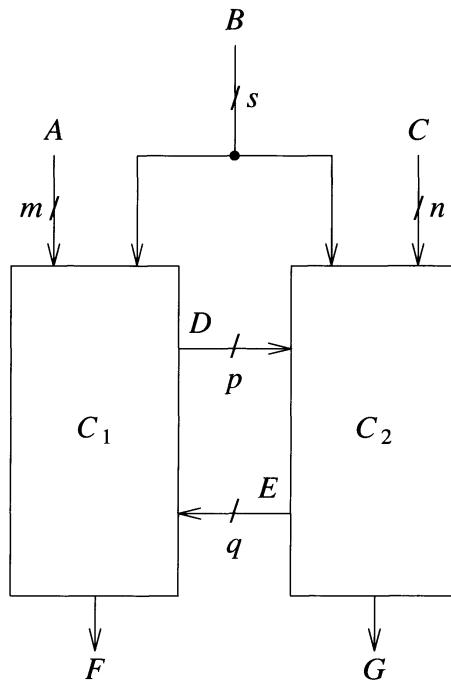
(a)

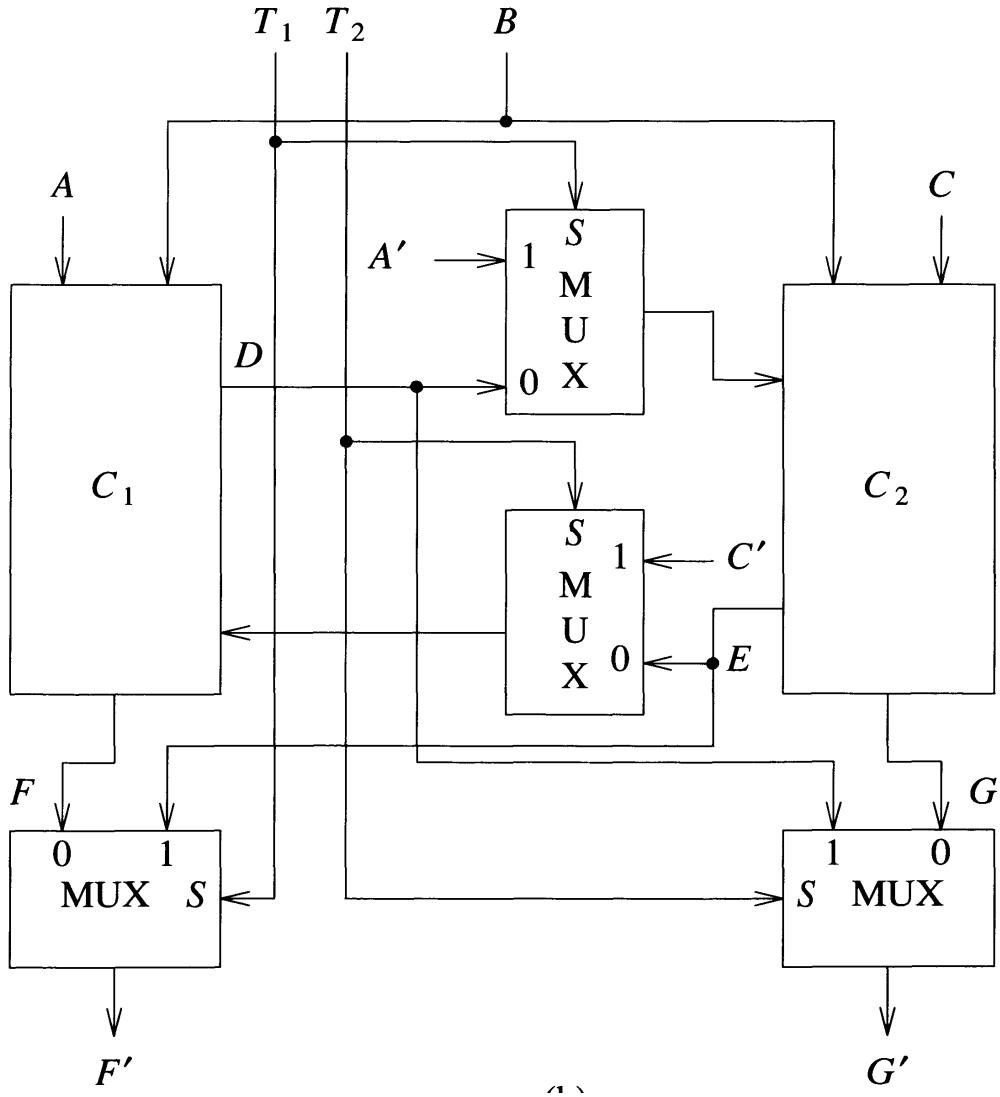


6. Partitioning of Large Comb. Circuits

Partition large circuits into small subcircuits to reduce test generation costs.

- Time complexity is a linear function of $m + n$, where m and n are the number of primary inputs and outputs respectively. This grows faster than a linear function of the number of nodes in the circuit.
- Partitioning can reduce test generation cost by dividing the circuit into smaller, more manageable subcircuits.





T_1	T_2	Mode
0	0	normal
0	1	test C_1
1	0	test C_2

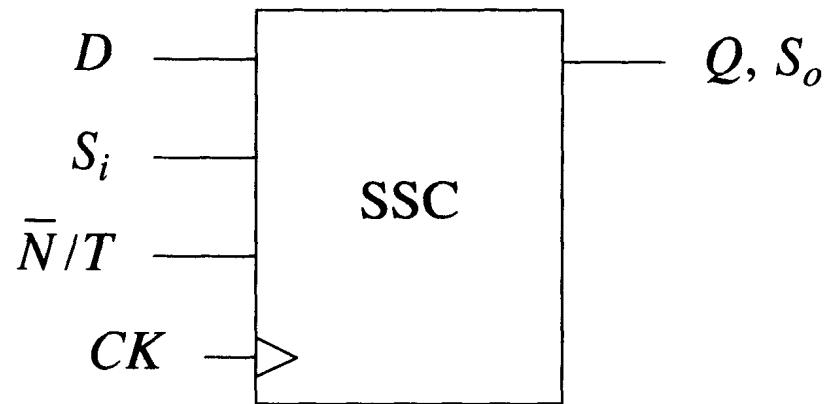
7 – Logical Redundancy

- **Rule:** *Avoid the use of redundant logic.*
- Redundancy makes faults undetectable
- It may invalidate some test for nonredundant faults
- Can cause difficulty in fault coverage calculations
- Redundancy can be introduced inadvertently
 - Maybe difficult to remove.
 - Test points can be added to remove redundancy during testing

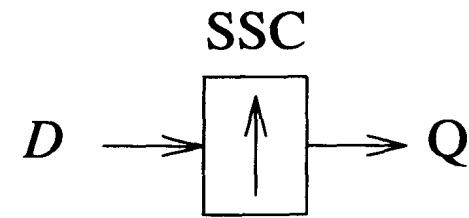
Scan Registers

- Test points are costly in terms of I/O pins
- Scan Register is an alternative – tradeoff between test time, area, and I/O pins
- Scan Register (SR) = Register with both shift and parallel-load capability
- Storage cells in SR can be used as observation and control points

Scan Storage Cell (SSC)



(a)

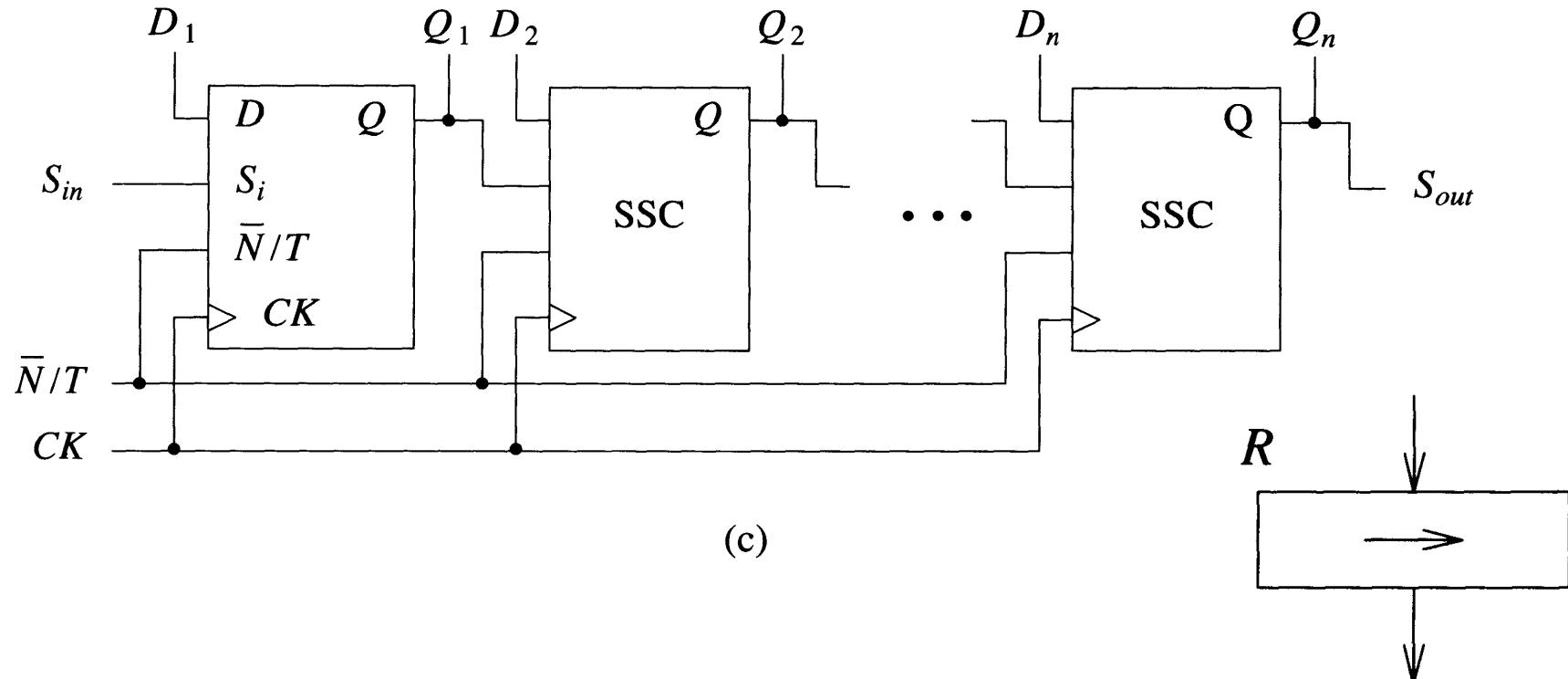


(b)

$\bar{N}/T = 0$ Normal mode: D is loaded

$\bar{N}/T = 1$ Testing mode: S is loaded

Scan Register



Normal mode ($\bar{N}/T = 0$)

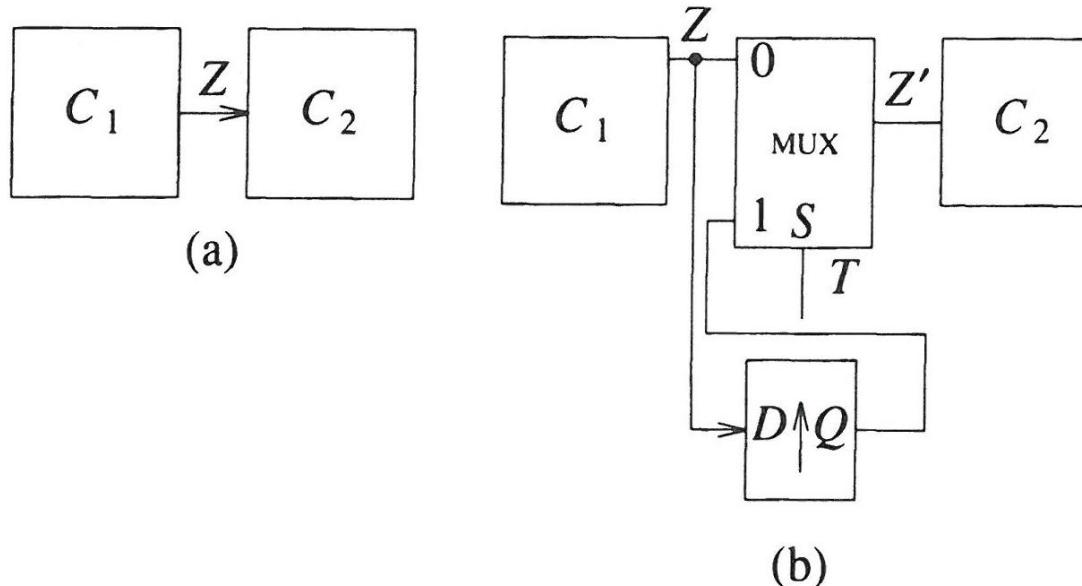
load data (D_x) in parallel

Test mode ($\bar{N}/T = 1$)

shift data (from S_{in} to S_{out})

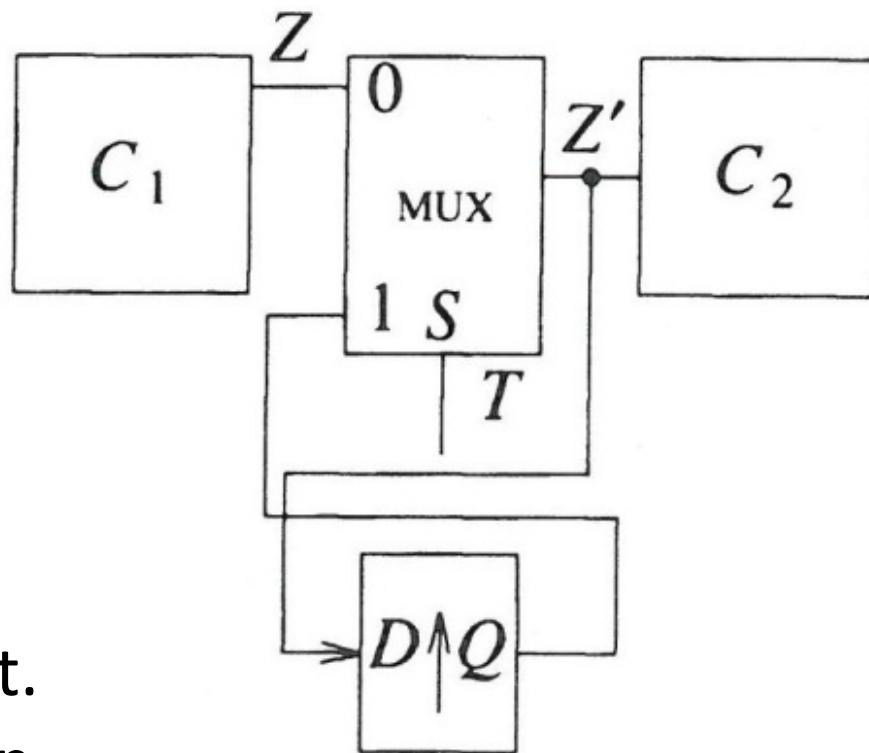
(d)

Simultaneous Controllability and Observability



- C_1 and C_2 can be sequential/combinational
- Z can be loaded into SSC via scan-in and observed by scan-out operation
- Data can be loaded in SSC via D -input and injected onto line Z'

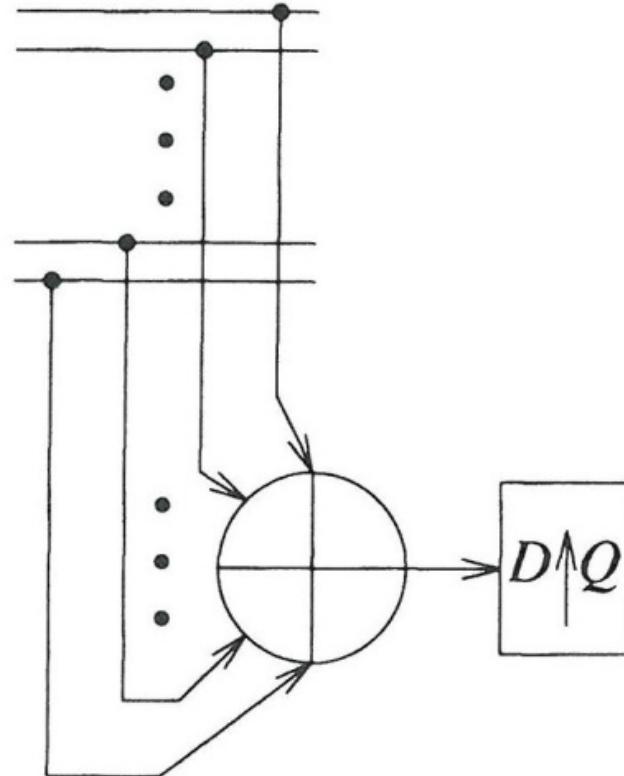
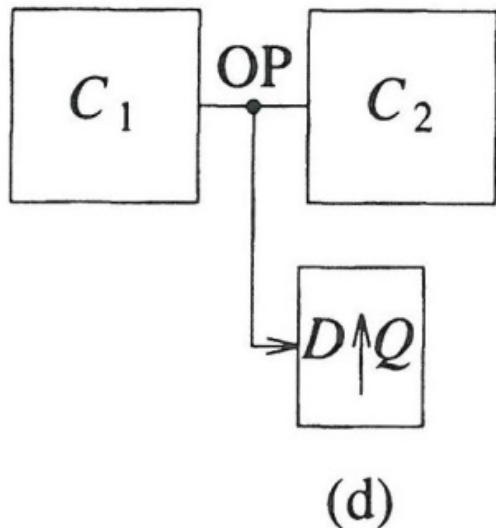
Separate Controllability and Observability



Z' is connected to D -input.
 CP is connected to Scan-in.

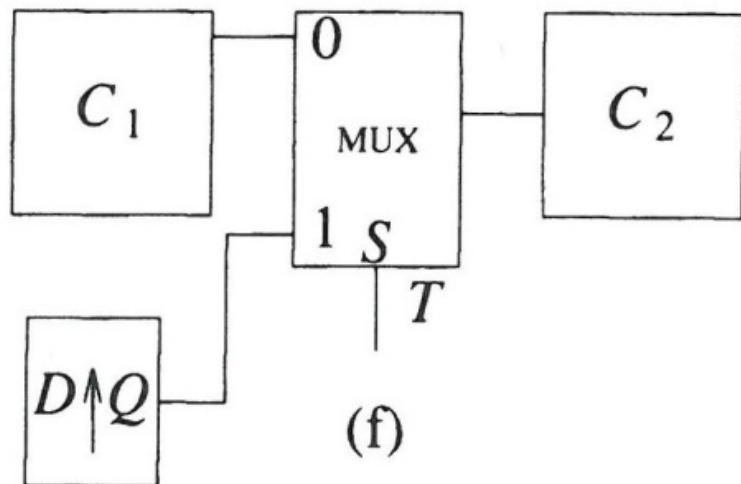
(c)

Observability Only



(e)

Controllability Only



Making Undetectable Faults Detectable

- X' = Control points Z' = Observation points
- Lets say f is an undetectable fault
- Choose X' and Z' such that f becomes detectable
- R1 and R2 can be combined as a single register

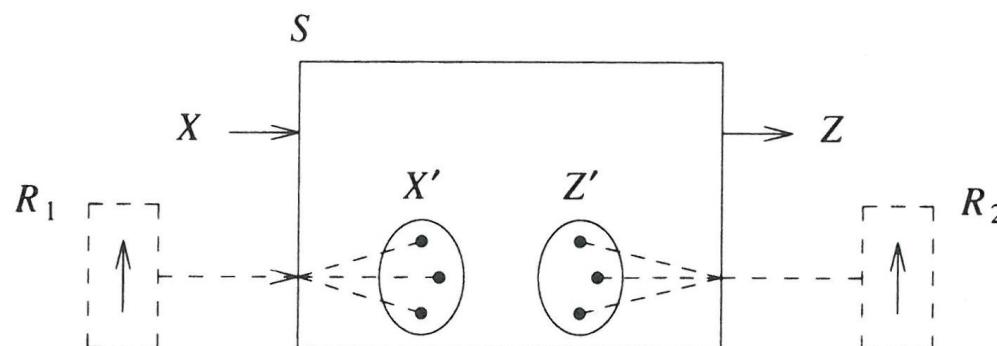
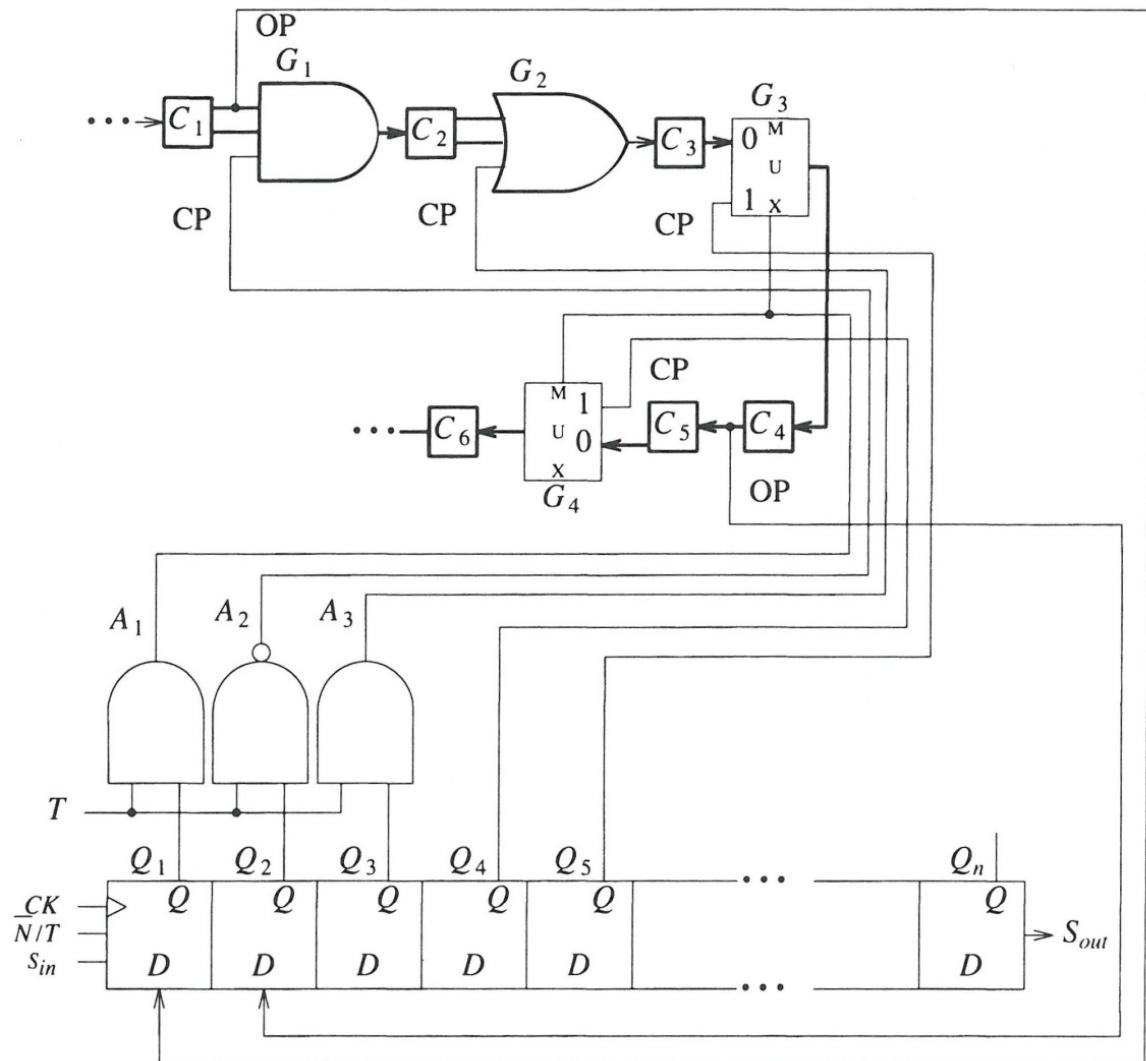


Figure 9.15 General architecture using test points tied to scan registers

Example 1 – Enhancing Testability

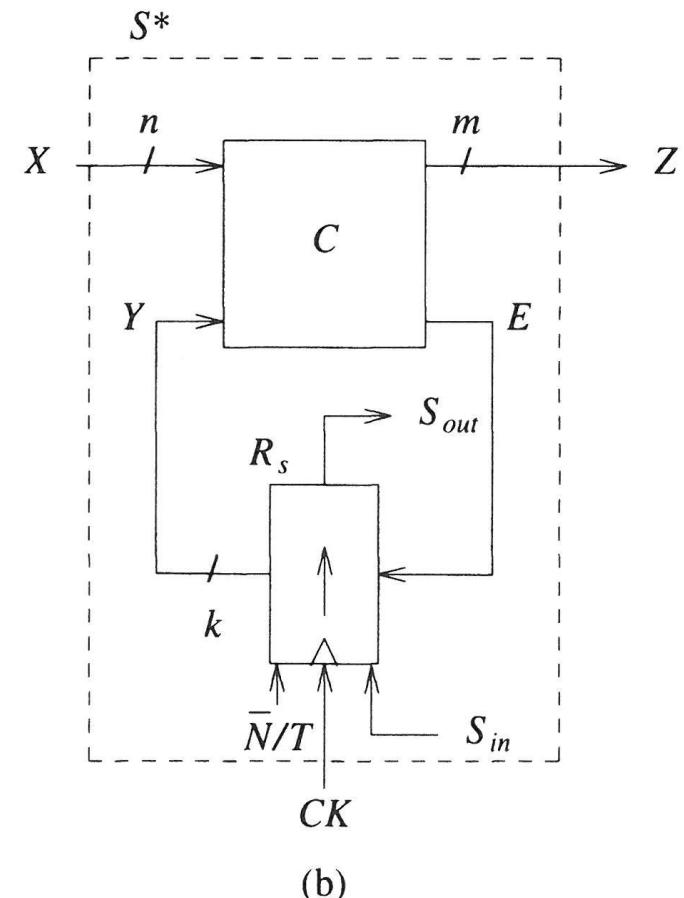
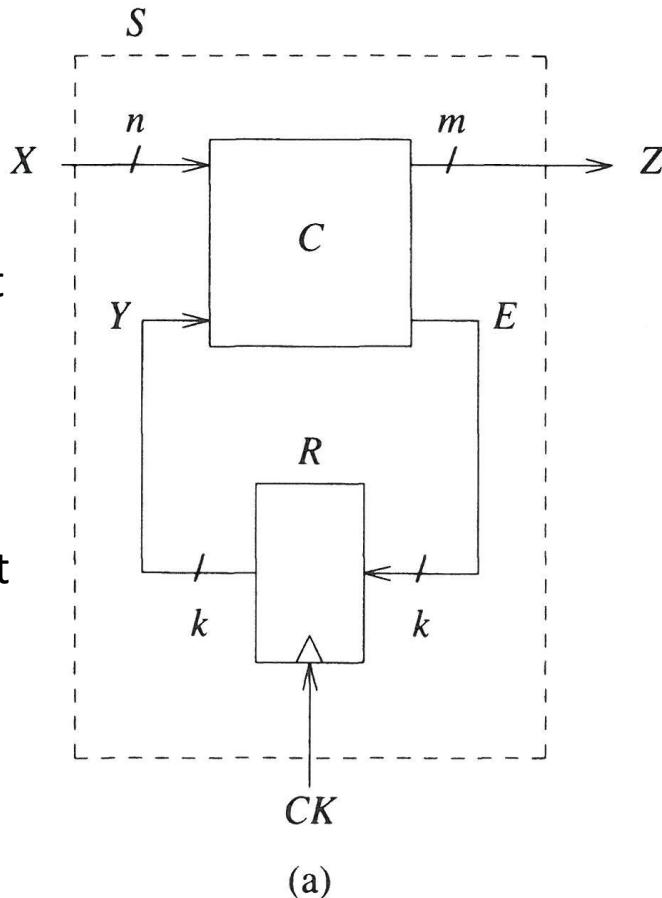
- C1, C2, ... C6 are Complex Seq/Comb blocks.
- # of CPs and OPs decides the length of scan register.
- *How does it work?*



Generic Scan-Based Designs

- Scan Design – most popular structured DFT technique, employs a scan register
- Several forms of scan designs – differ primarily in the scan cell design
- Three generic forms of scan design
 - Full Serial Integrated Scan
 - Isolated Serial Scan
 - Non-serial Scan

Full Serial Integrated Scan



All storage elements in the design become part of scan register.

Instead of testing circuit in (a) as a sequential circuit, now C can be tested using a series of test vectors.

Figure 9.19 (a) Normal sequential circuit S (b) Full serial integrated scan

Isolated Serial Scan

- Unlike Full Serial, Scan register is not part of the data path

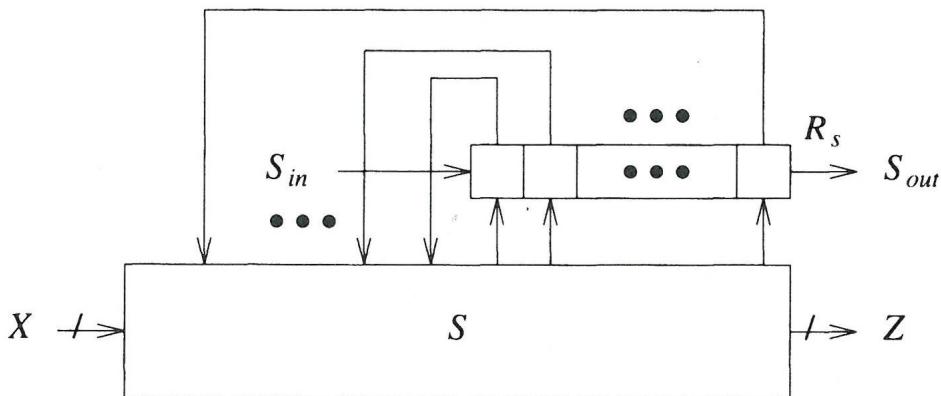


Figure 9.20 Isolated serial scan (scan/set)

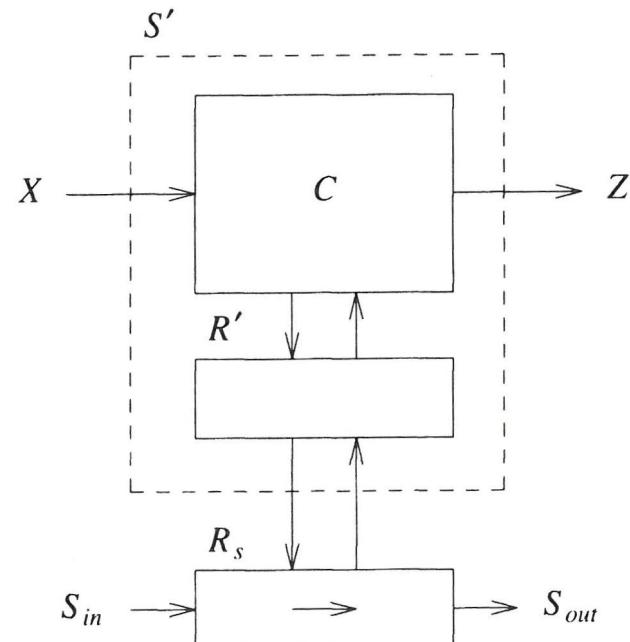


Figure 9.21 Full isolated scan

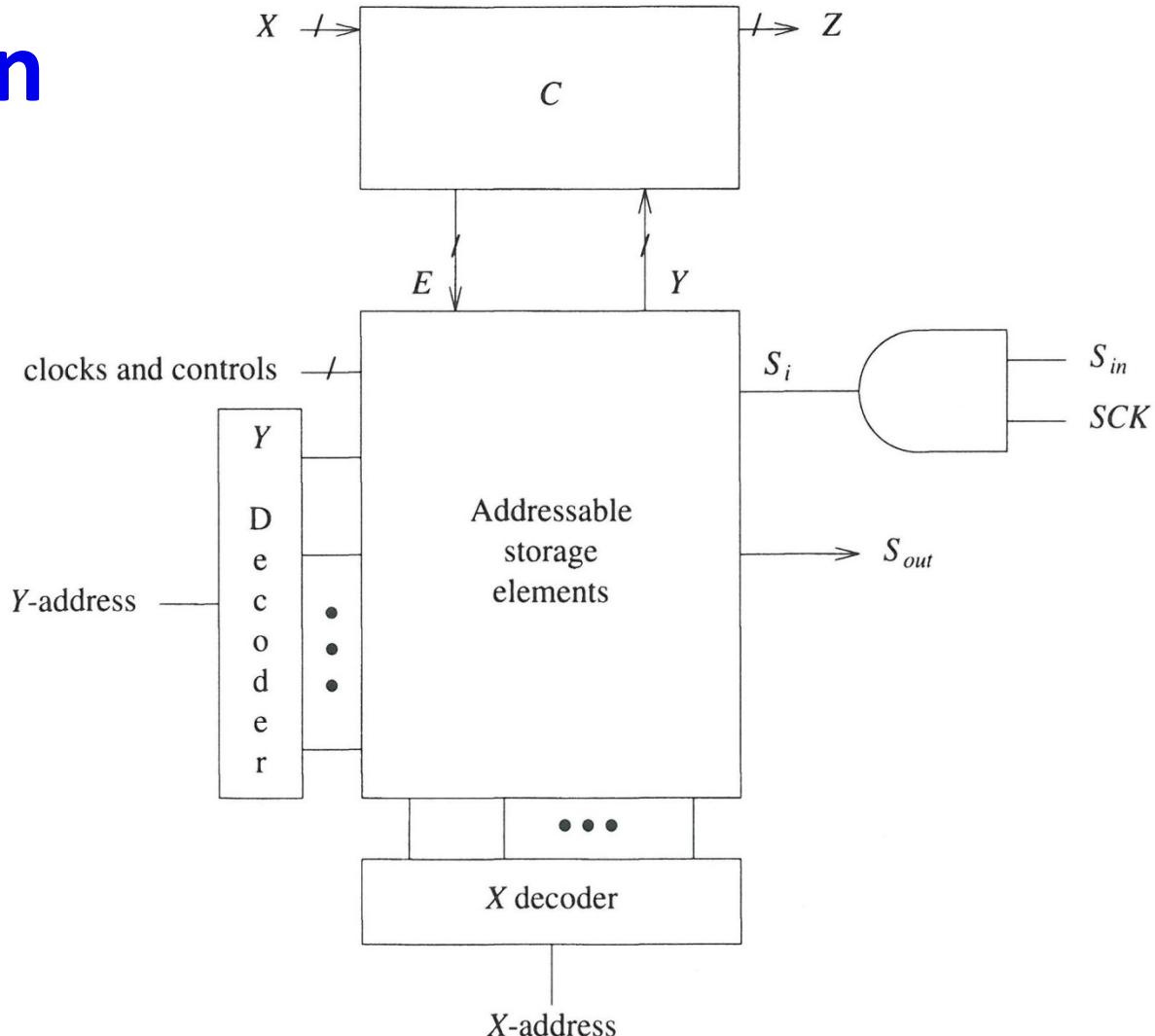
Shadow register R_s : does not interfere the normal operation,

Non-Serial Scan

RAM is used instead of shift register.

Individual bits can be modified for selected CPs or OPs.

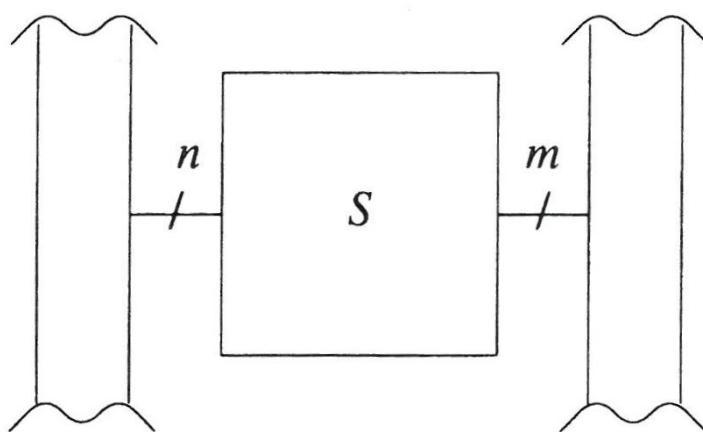
Area overhead is high.



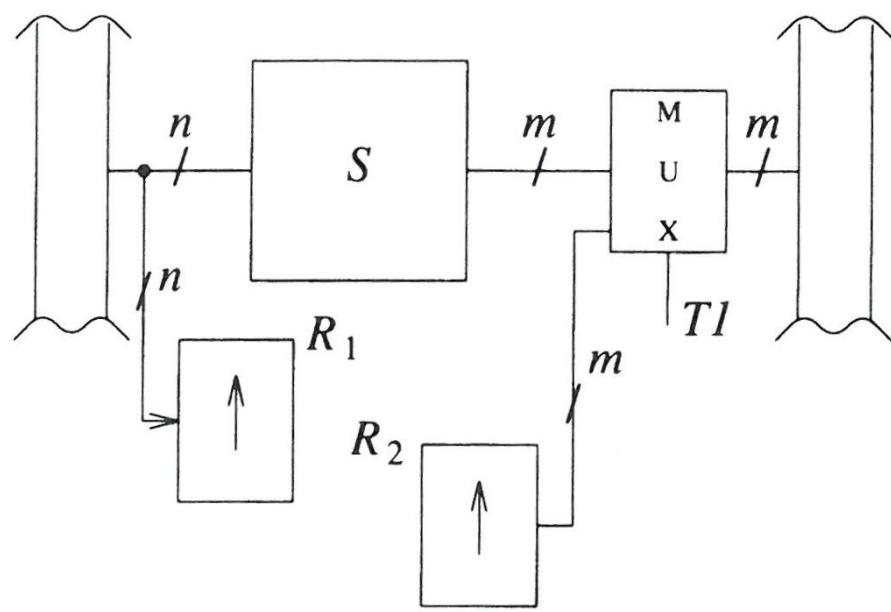
Generic Boundary Scan

- Concept – in designing modules such as complex chips or PCBs, for local testing and fault isolation, we should be able to isolate one module from another
- All chips on board are designed using boundary scan architecture
- Boundary scan registers are on the periphery; not part of the function
- Test vectors can be scanned in and responses saved and scanned out
- Internal clock must be disabled

Generic Boundary Scan



(a)



(b)

Figure 9.18 Boundary scan architecture (a) Original circuit (b) Modified circuit

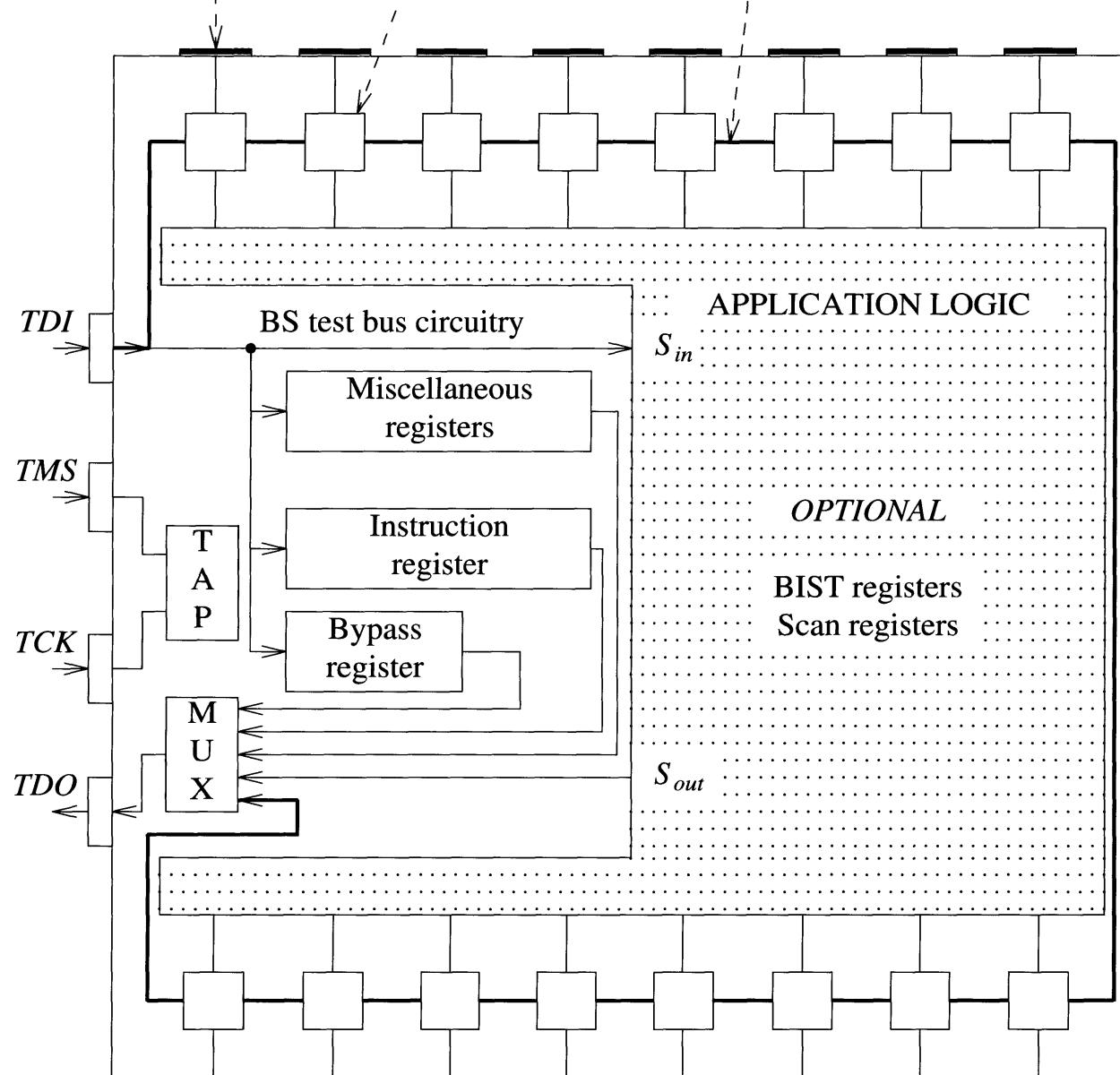
Boundary-Scan Standards

- Goal – to ensure chips of VLSI complexity contain standard DFT circuitry to make test development effective and less costly
- Some initiatives
 - Joint Test Action Group (JTAG) Boundary Scan Std
 - VHSIC Element Test and Maintenance (IBM Std)
 - IEEE 1149.1 Testability Bus Standard
- Primarily deal with
 - Test Bus (resides on the board)
 - Bus Protocol
 - Interface logic between test bus ports and DFT hardware
- JTAG Boundary Scan and IEEE 1149.1 require a bound-scan register exist on the chip

TAP = Test Access Port

- TDI = Test Data Input
- TDO = Test Data Output
- TMS = Test Mode Signal
- TCK = Test Clock

TAP Controller = a FSM
control operation of test
bus

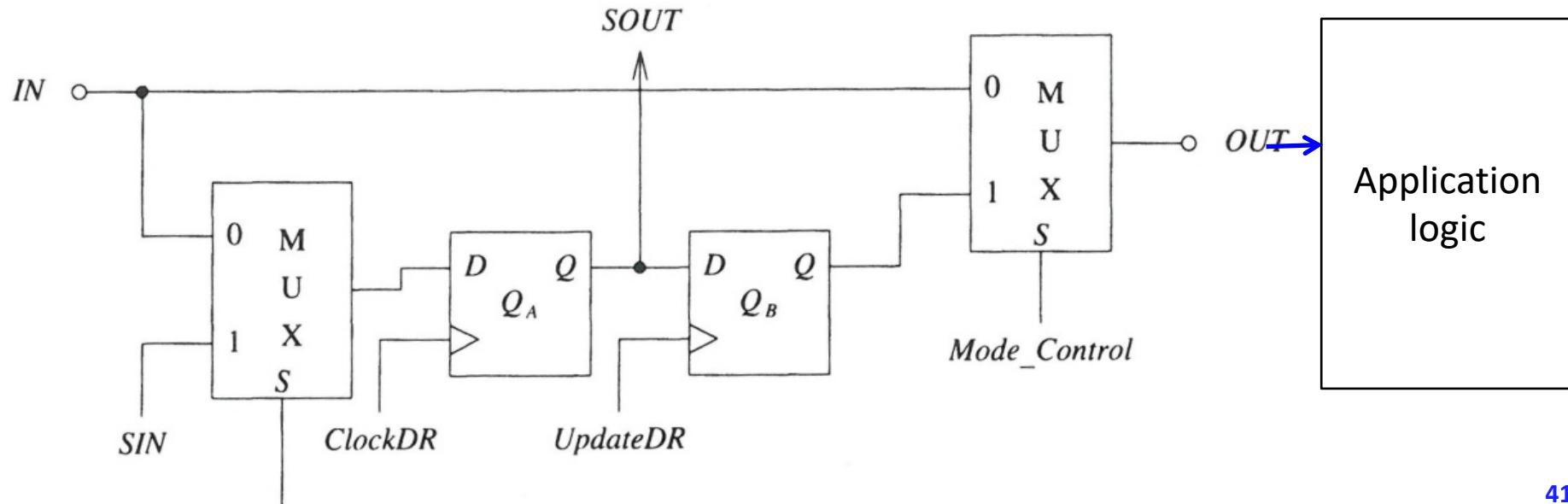


Test Bus Operation

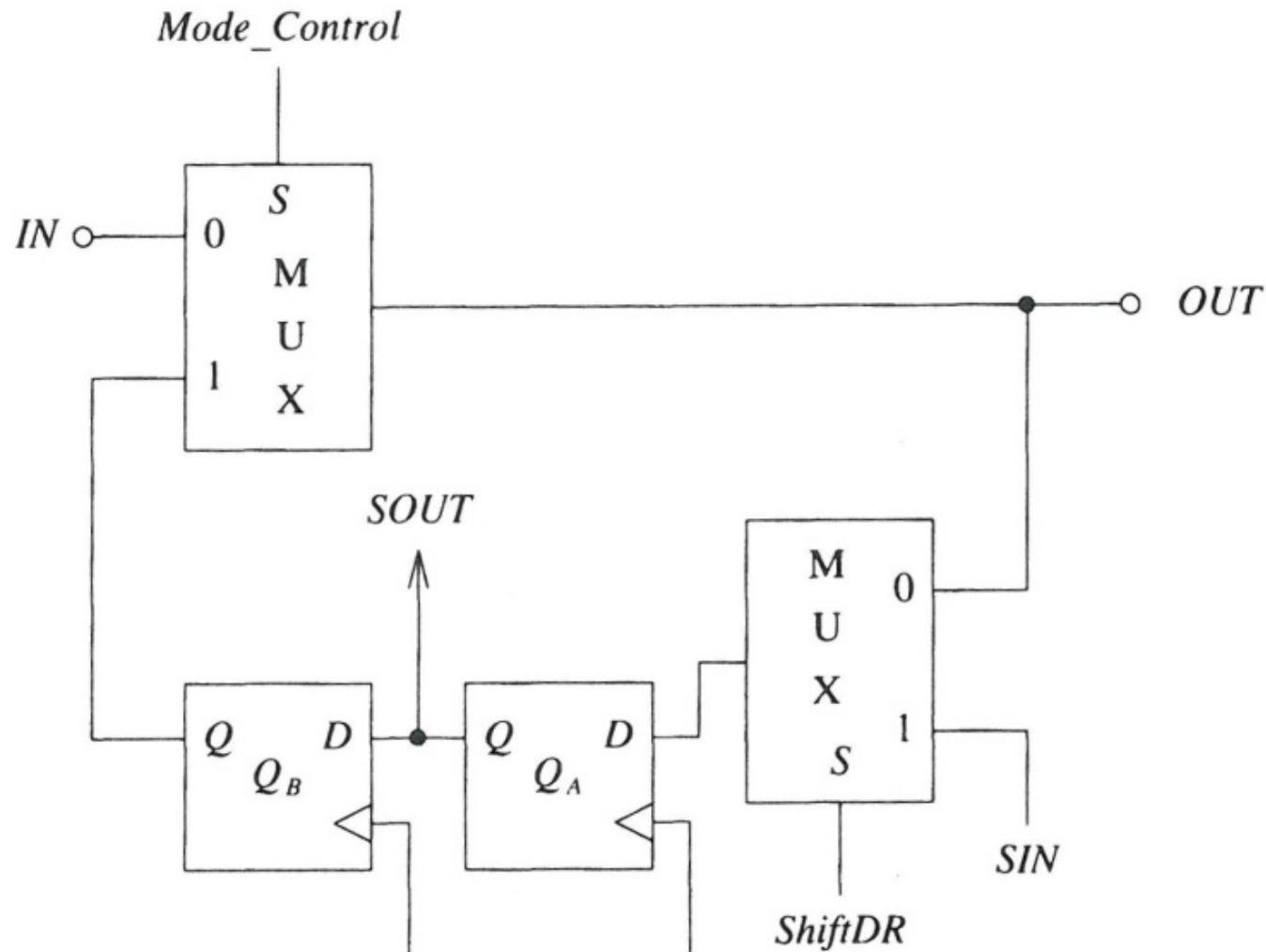
1. Instruction sent serially over the TDI line into the instruction register
2. Selected test circuitry is configured to respond to the instruction
 - More data needed to configure data registers
3. The test instruction is executed. Test results can be shifted out of selected registers and transmitted over TDO line to the bus master. Data can be shifted in while results are shifted out.

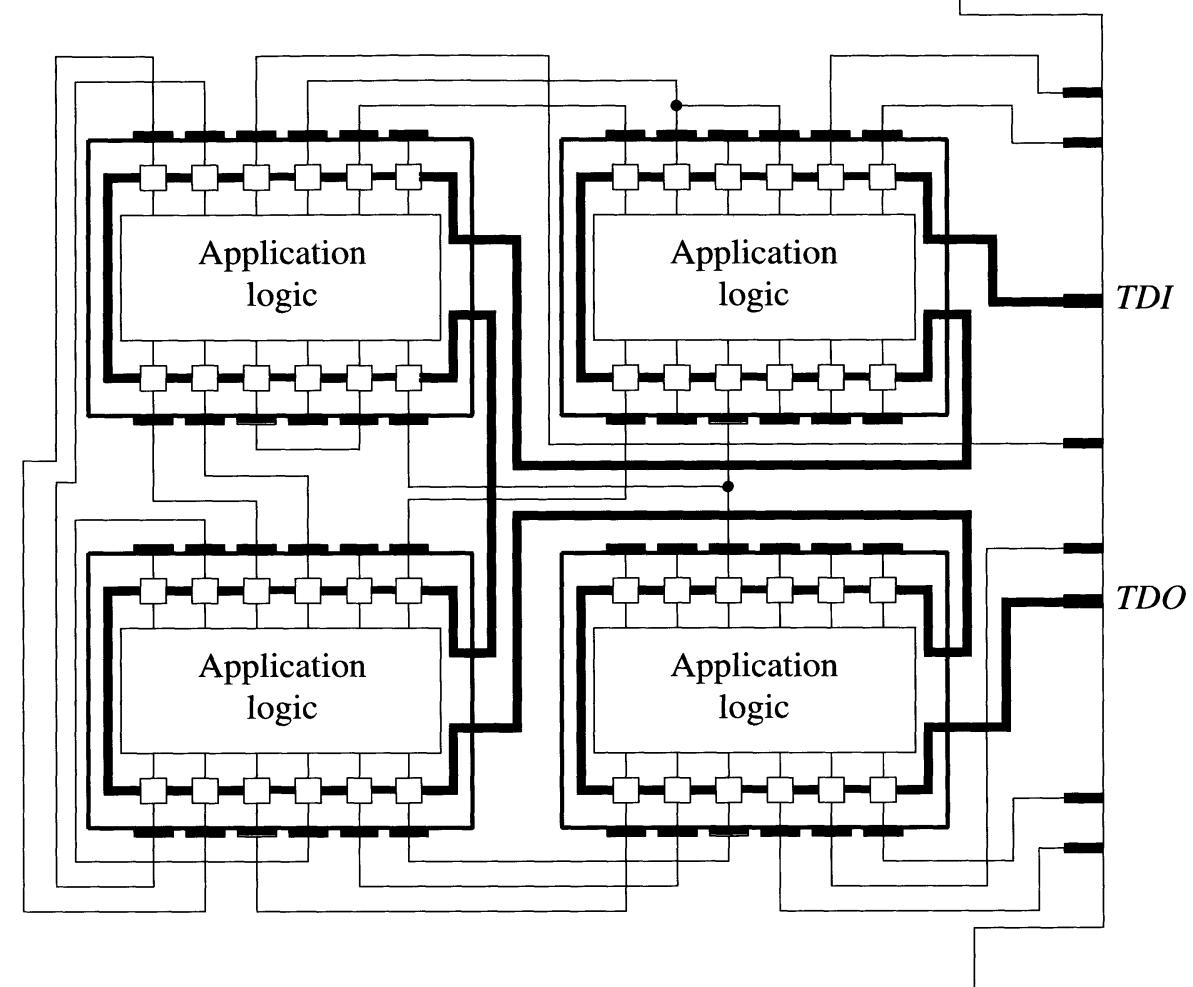
Boundary Scan Cell

- **Normal Mode:** *Mode_Control* = 0, cell is transparent
- **Scan Mode** – Boundary cells are interconnected into a scan path (TDI input, TDO output) *ShiftDR* = 1 and Clock pulses applied to *ClockDR*
- **Capture Mode** – *ShiftDR* = 0 => input IN is *captured*
- **Update Mode** – Once Q_A is loaded (by scan/capture), set *Mode_control* = 1 and apply a clock pulse to *UpdateDR* for the value in Q_A applied to OUT



Boundary Scan Cell – Another Design



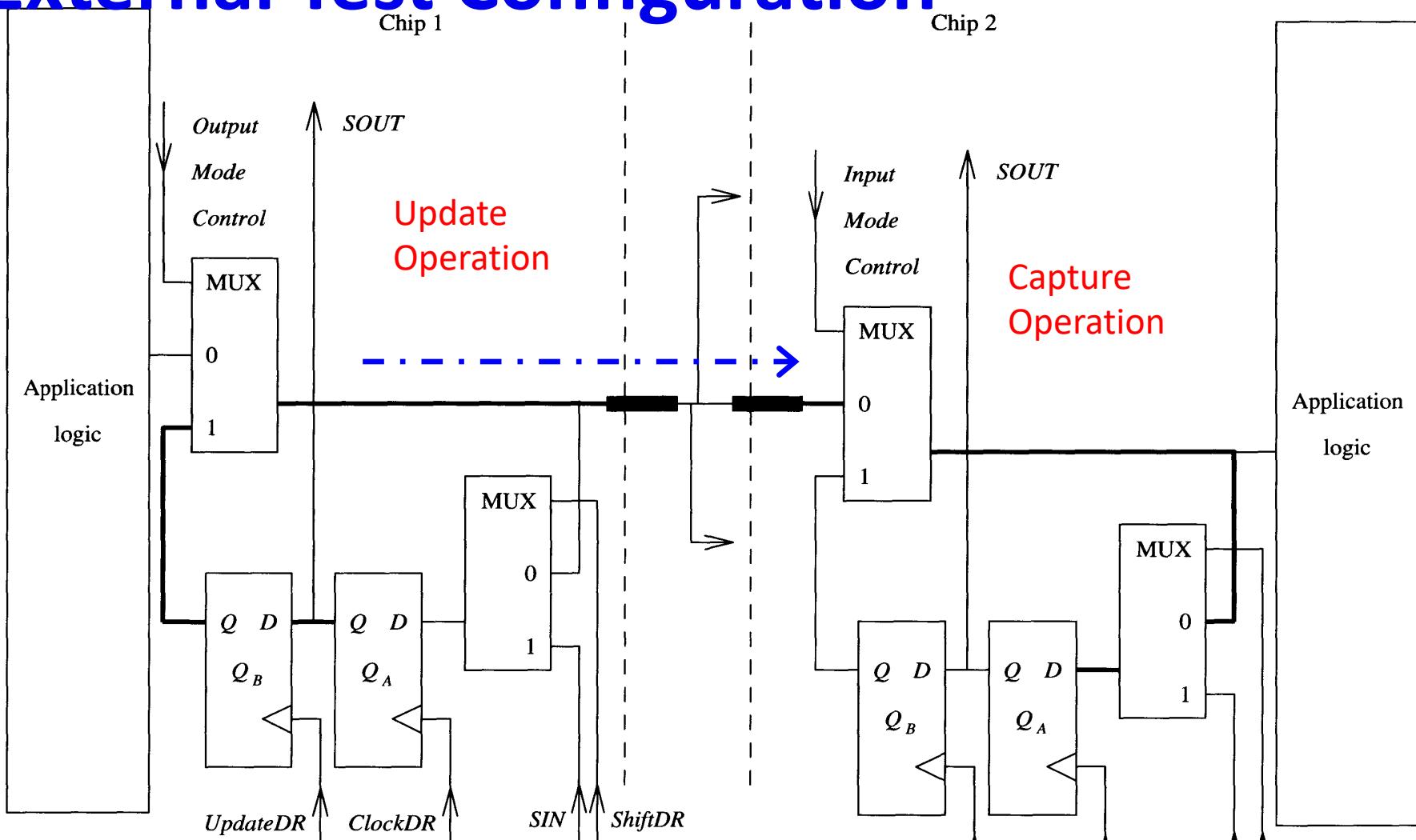


- Interconnect Test
- System Snapshot
- Chip Test

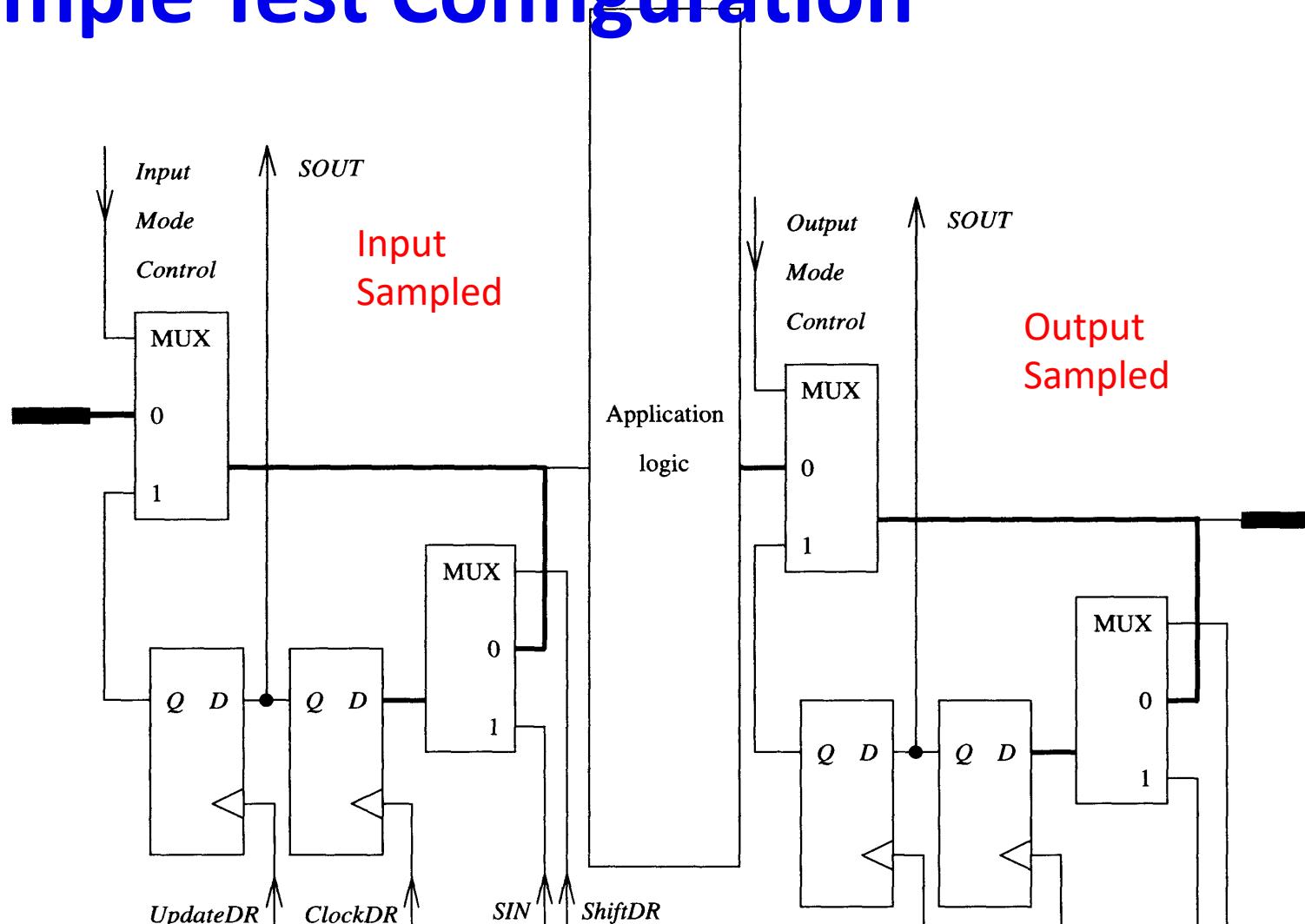
PCB Test – Three modes

- External Test Mode
 - Test interconnects between chips
- Sample Test Mode
 - I/O data of a chip can be sampled during normal system operation – **snapshots of chip IO data**
 - Sampled data can be scanned out while board is in normal operation
- Internal Test Mode
 - Inputs to the application logic is driven by the input boundary-scan cells and response captured in output boundary-scan cells

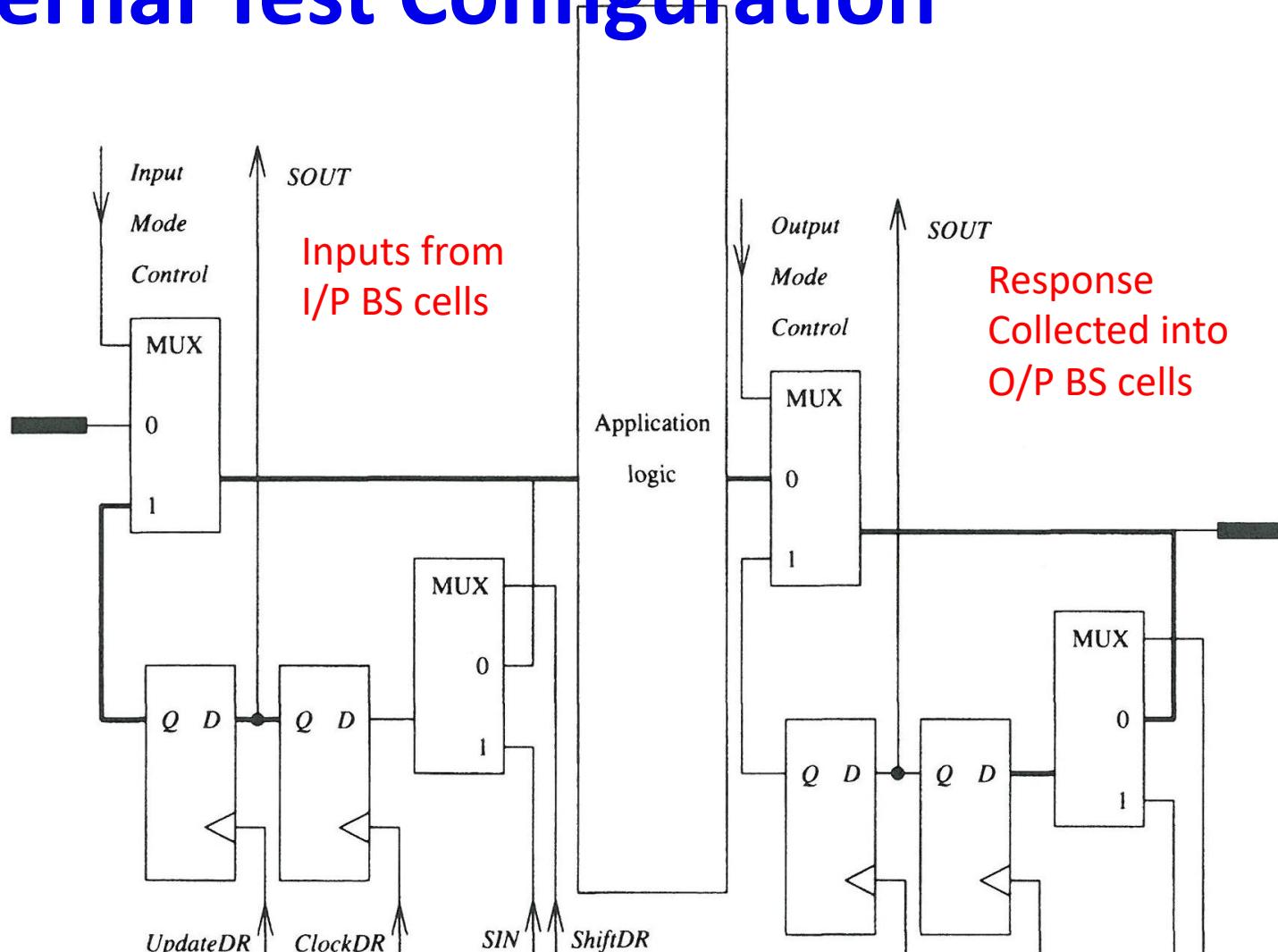
External Test Configuration

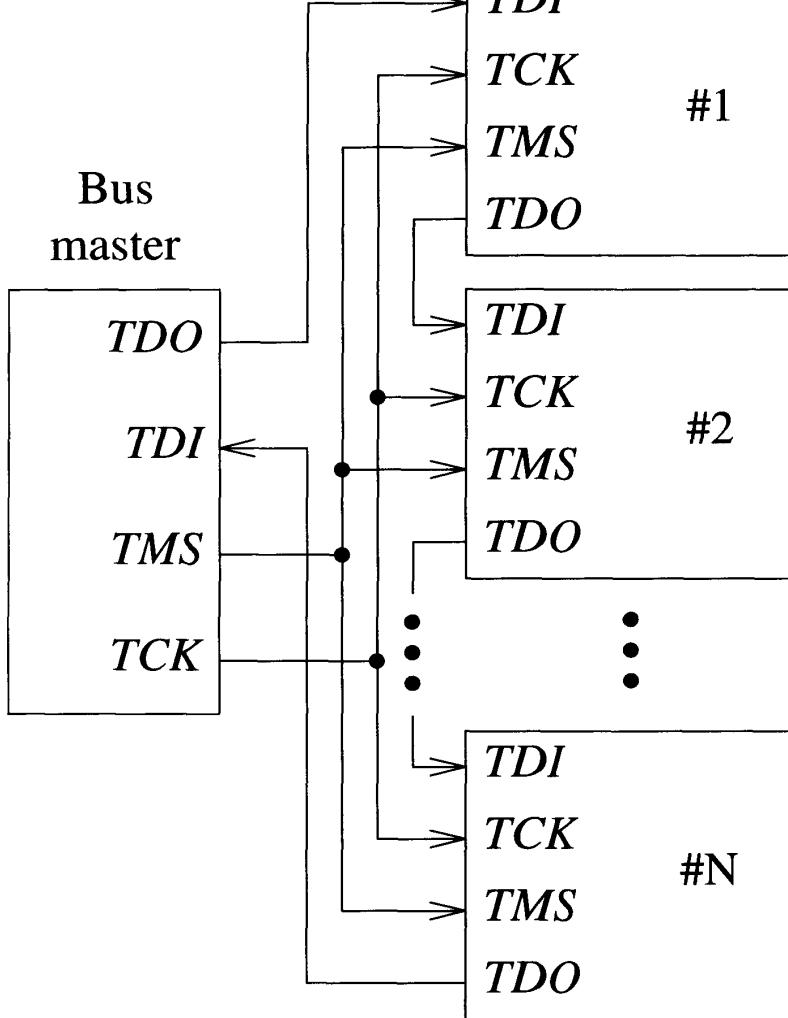


Sample Test Configuration

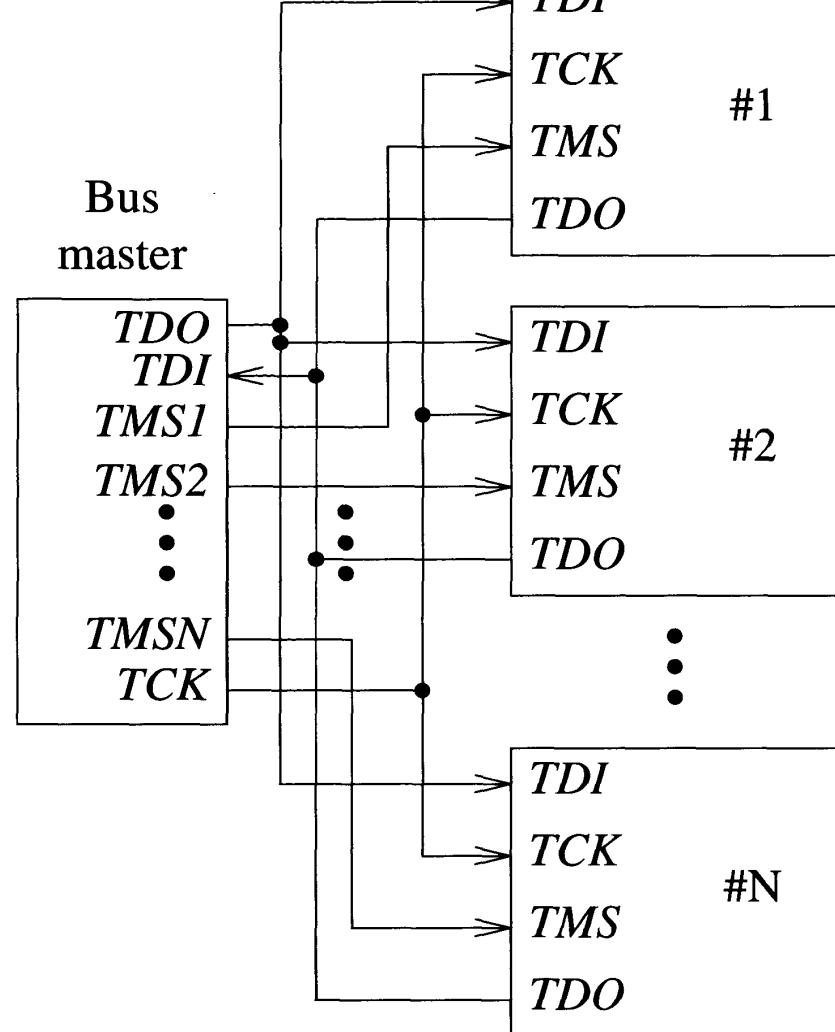


Internal Test Configuration





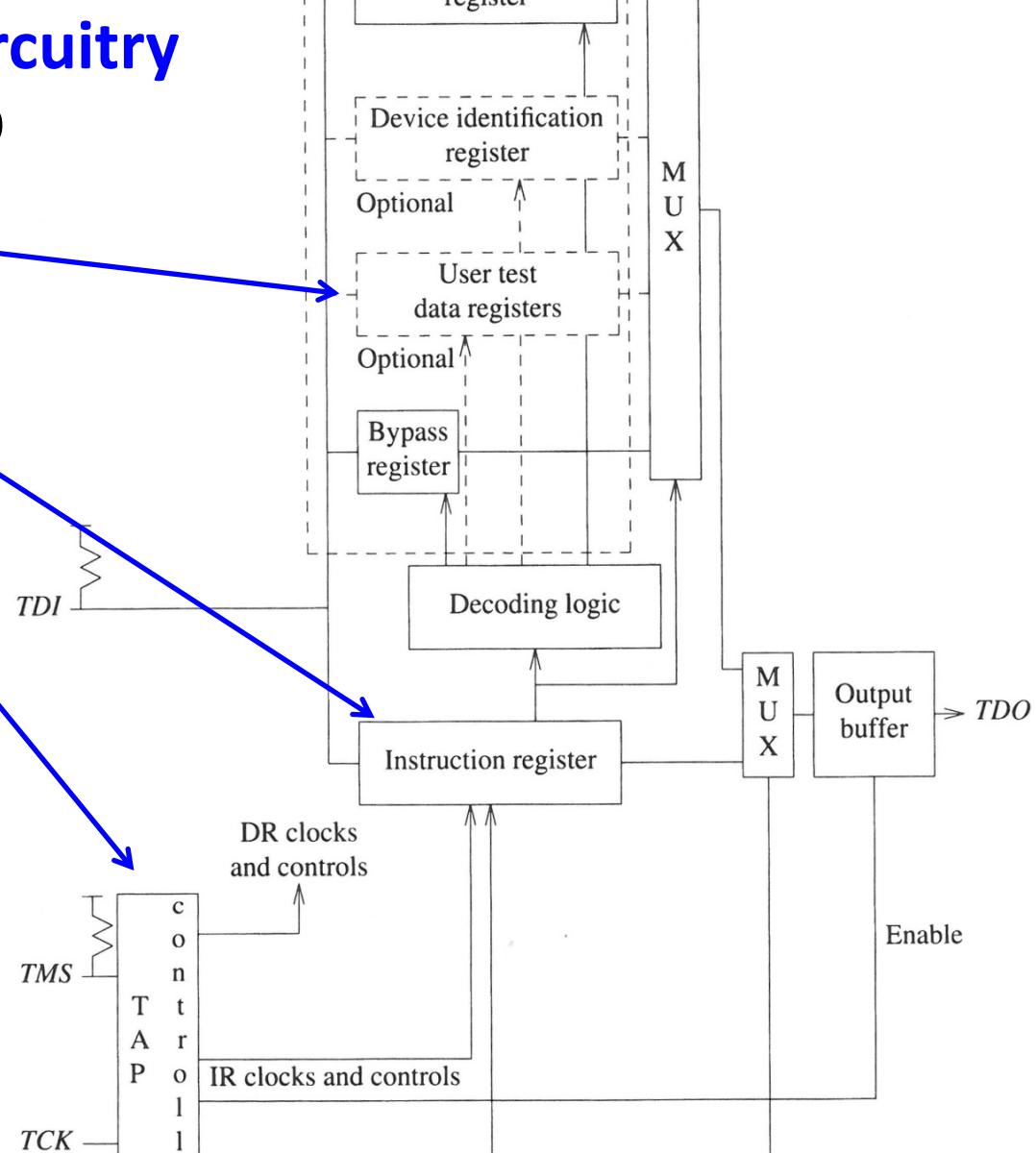
(a)



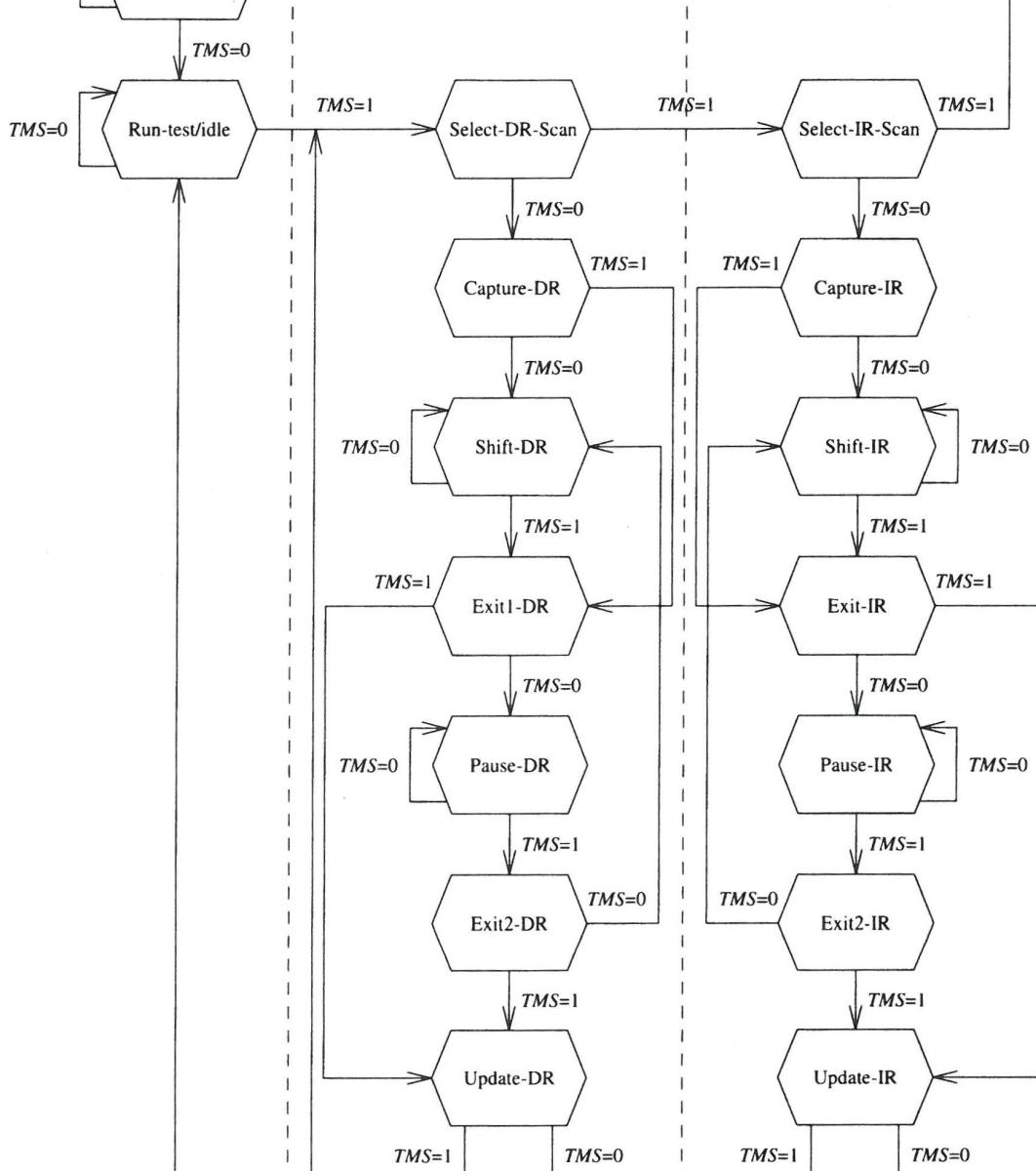
(b)

IEEE 1149.1 Test Bus Circuitry

- TAP – TCK, TMS, TDI, TDO
- Test DR
- Scannable IR
- TAP controller



State Diagram of TAP Controller



Instruction Register and Commands

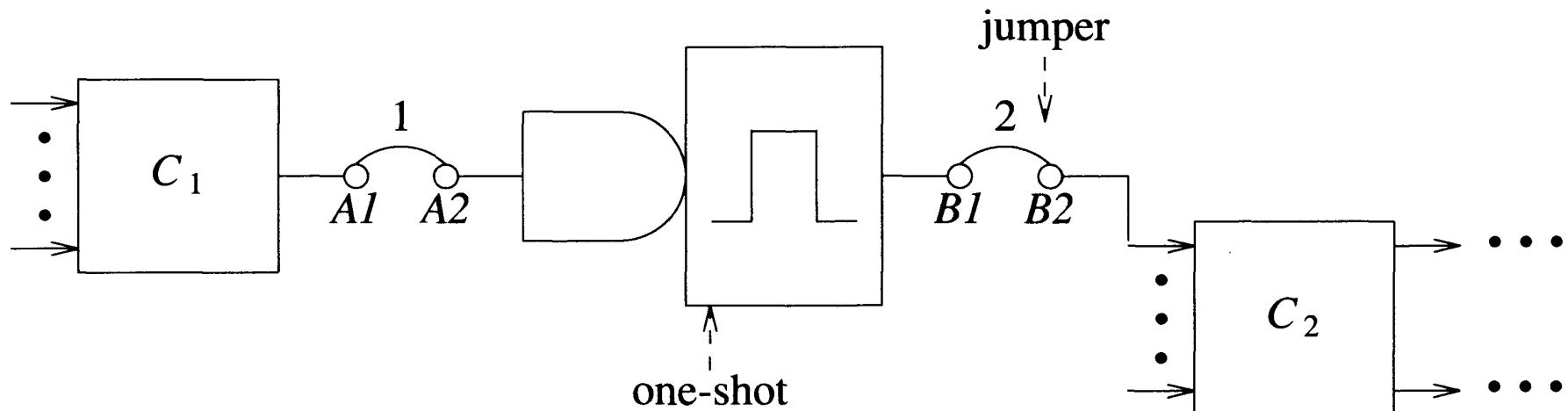
- Commands can be shifted into IR from TDI.
- Commands specify operations and selection of DR.
 - **BYPASS** – exclude a chip from scan path.
 - **EXTEST** – test intra-chip interconnect
 - **SAMPLE** – capture chip IO and store data on boundary scan registers.
 - **INTEST** – test a chip itself.
 - **RUNBIST** – support a self-testing.

Backup

3 – Monostable Multivibrators

Rule: *Disable internal one-shots during test*

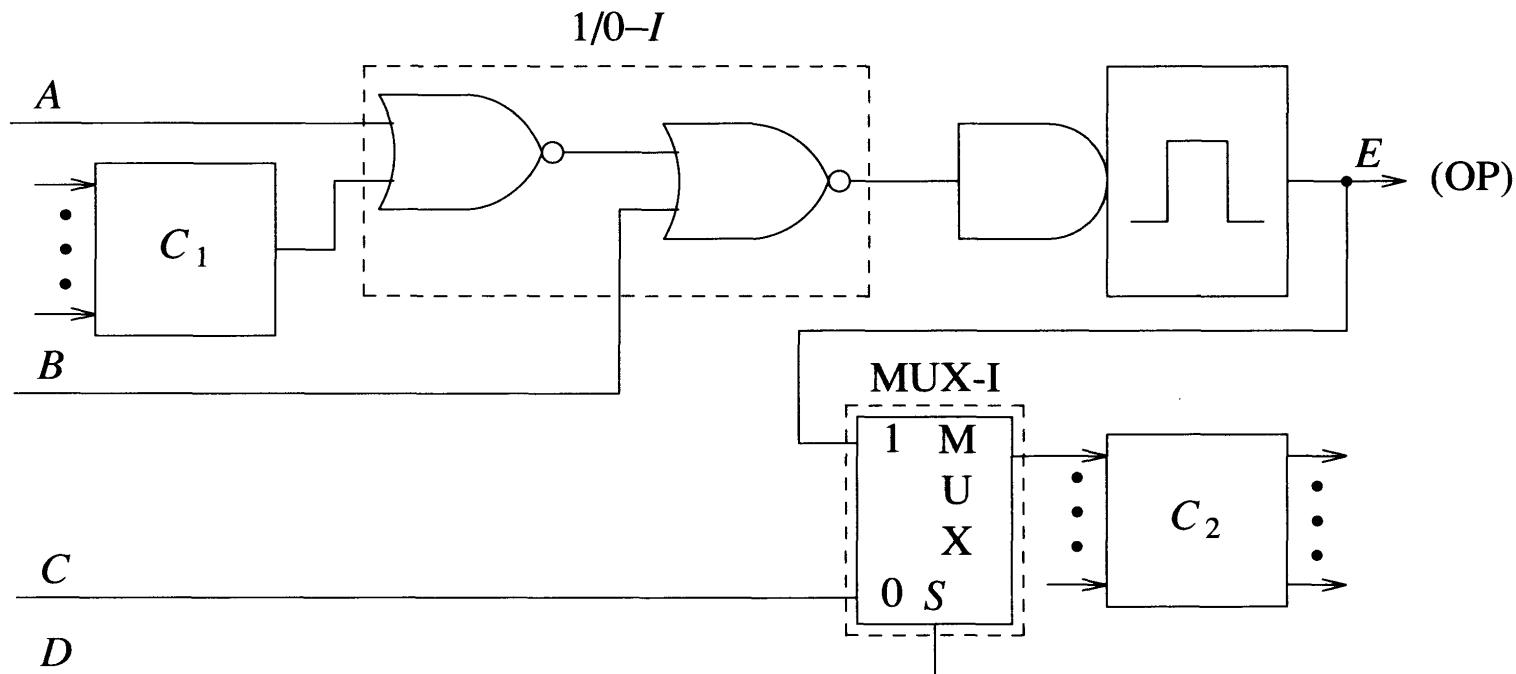
- One-shots provide pulses internal to circuit
- Difficult for ATE to remain in synchronization with the circuit



3 – Monostable Multivibrators

Rule: *Disable internal one-shots during test*

- One-shots provide pulses internal to circuit
- Difficult for ATE to remain in synchronization with the circuit



4 – Oscillators and Clocks

Rule: *Disable internal oscillators/clocks during test*

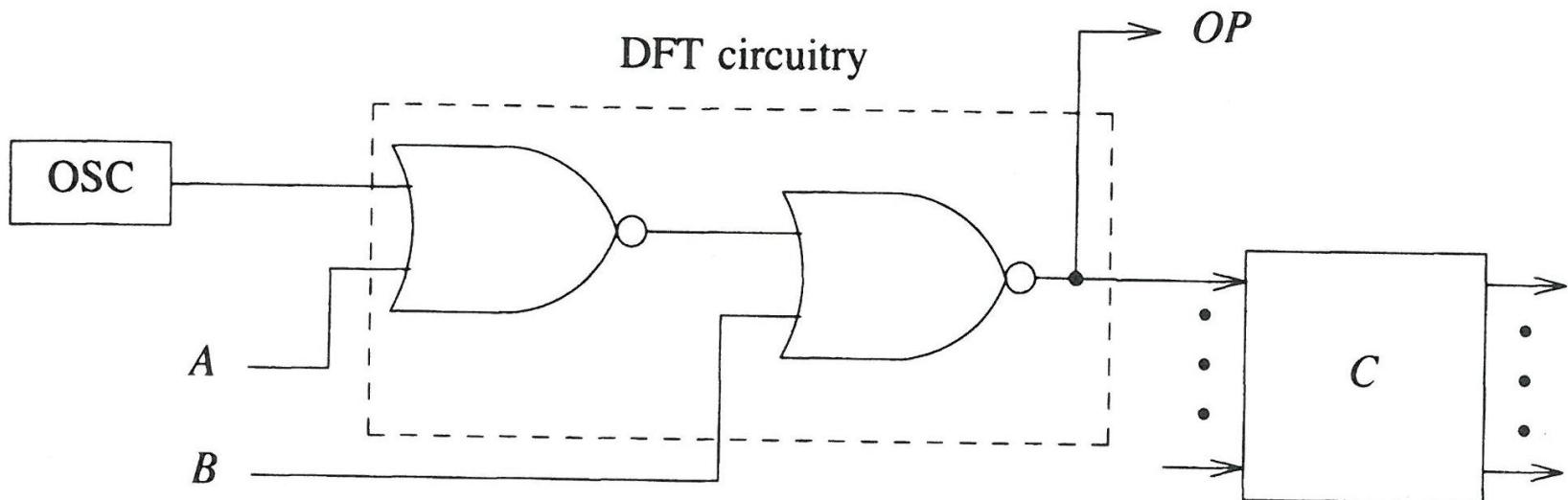


Figure 9.9 Testability logic for an oscillator

CIS 4930 Digital System Testing

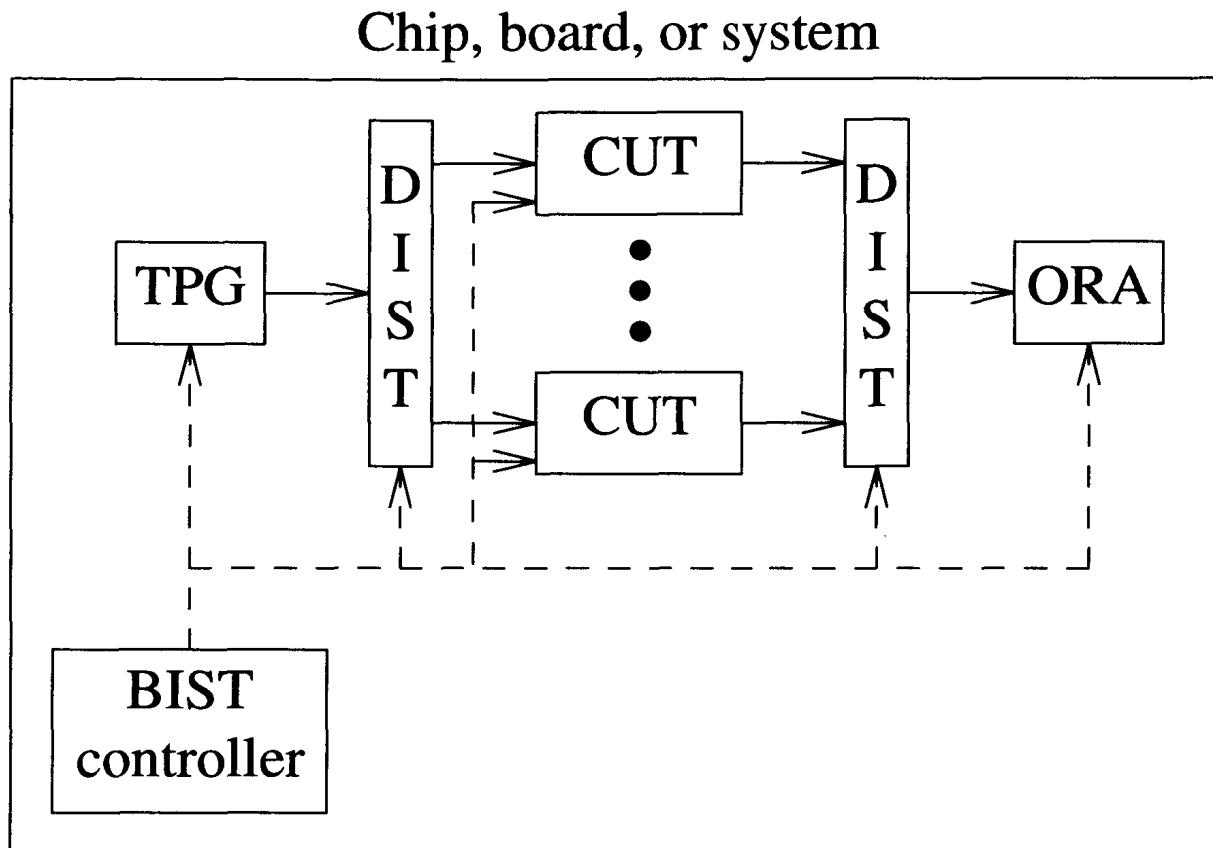
Built-In Self Test (BIST)

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

Introduction

Built-In Self-Test (BIST)

- BIST is the capability of a circuit (chip, board, or system) to test itself



Forms of Built-In Self-Test (BIST)

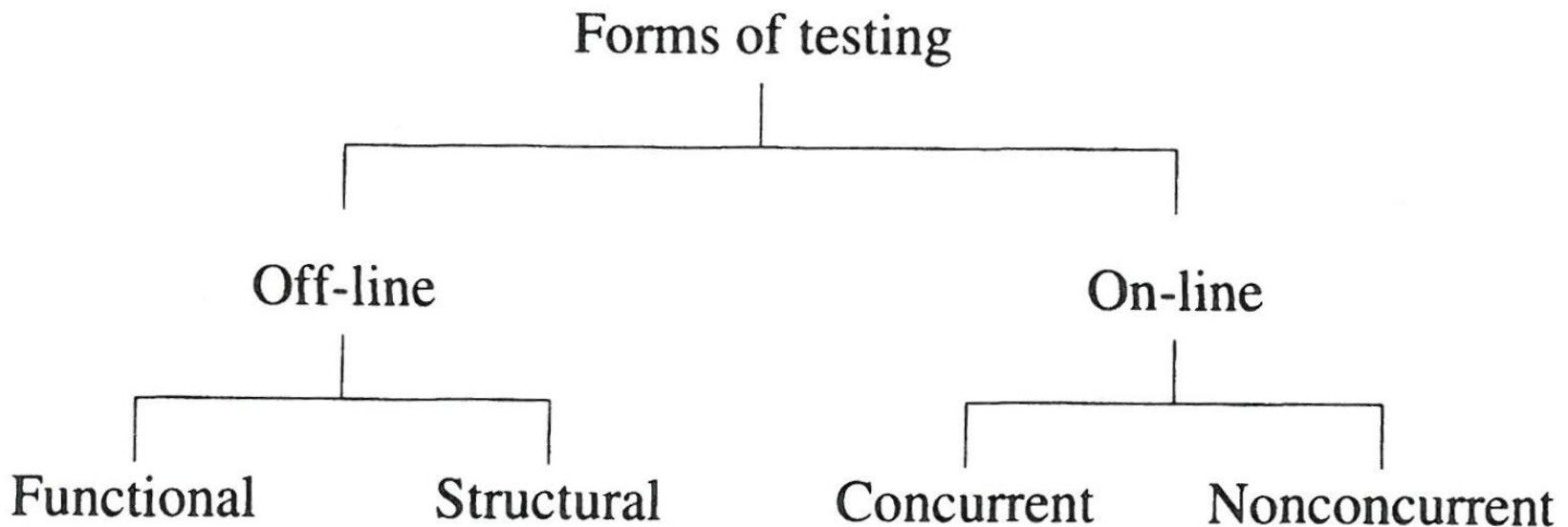


Figure 11.1 Forms of testing

On-line BIST

- Testing occurs during normal functional operating conditions
 - Circuit Under Test (CUT) is *not* put in test mode
- **Concurrent online BIST**
 - Testing occurs *simultaneously with normal functional operation*
- **Non-concurrent online BIST**
 - Testing while system is in **idle state**
 - Executing diagnostic software
 - Test process can be interrupted so that normal operation can resume

Off-line BIST

- Testing a system when it is not carrying out its normal functions
- Systems, boards, and chips can be tested
- Applicable at the manufacturing, field, depot, and operational stages
- Usually employs test-pattern generators (TPGs) and output response analyzers (ORAs)
- Errors cannot be detected in real time

Off-line BIST – cont'd

- **Functional off-line BIST**
 - Test based on functional description
 - Employs a functional fault model
- **Structural off-line BIST**
 - Explicit structural fault model may be used
 - Fault coverage based on **structural fault detection**
 - Usually tests are generated and responses are compressed

Our discussion is primarily on **Structural Off-line BIST**

Glossary of key BIST Architectures

BILBO — built-in logic block observer (register)

LFSR — linear feedback shift register

MISR — multiple-input signature register

ORA — (generic) output response analyzer

PRPG — pseudorandom pattern generator, often referred to as a pseudorandom number generator

SISR — single-input signature register

SRSG — shift-register sequence generator; also a single-output PRPG

TPG — (generic) test-pattern generator

Hardcore

- Parts of circuit that must be operational (correct) to execute a self-test
- At a minimum it consists of Power, Ground, and Clock Distribution
- Easy to detect, but hard to diagnose
 - Faults may be in CUT or hardcore
- Usually tested by external test equipment
- Designer attempts to minimize complexity of hardware

Levels of Test

- **Production Test**
 - Newly manufactured components
 - Performed at Chip, Board, System levels
 - Reduces the need for expensive ATE (Automated Test Equipment)
- **Field Testing**
 - Eliminates the need for expensive special test equipment.
 - Improve maintainability,
 - Reduce life-cycle costs.

Test-Pattern Generation for BIST

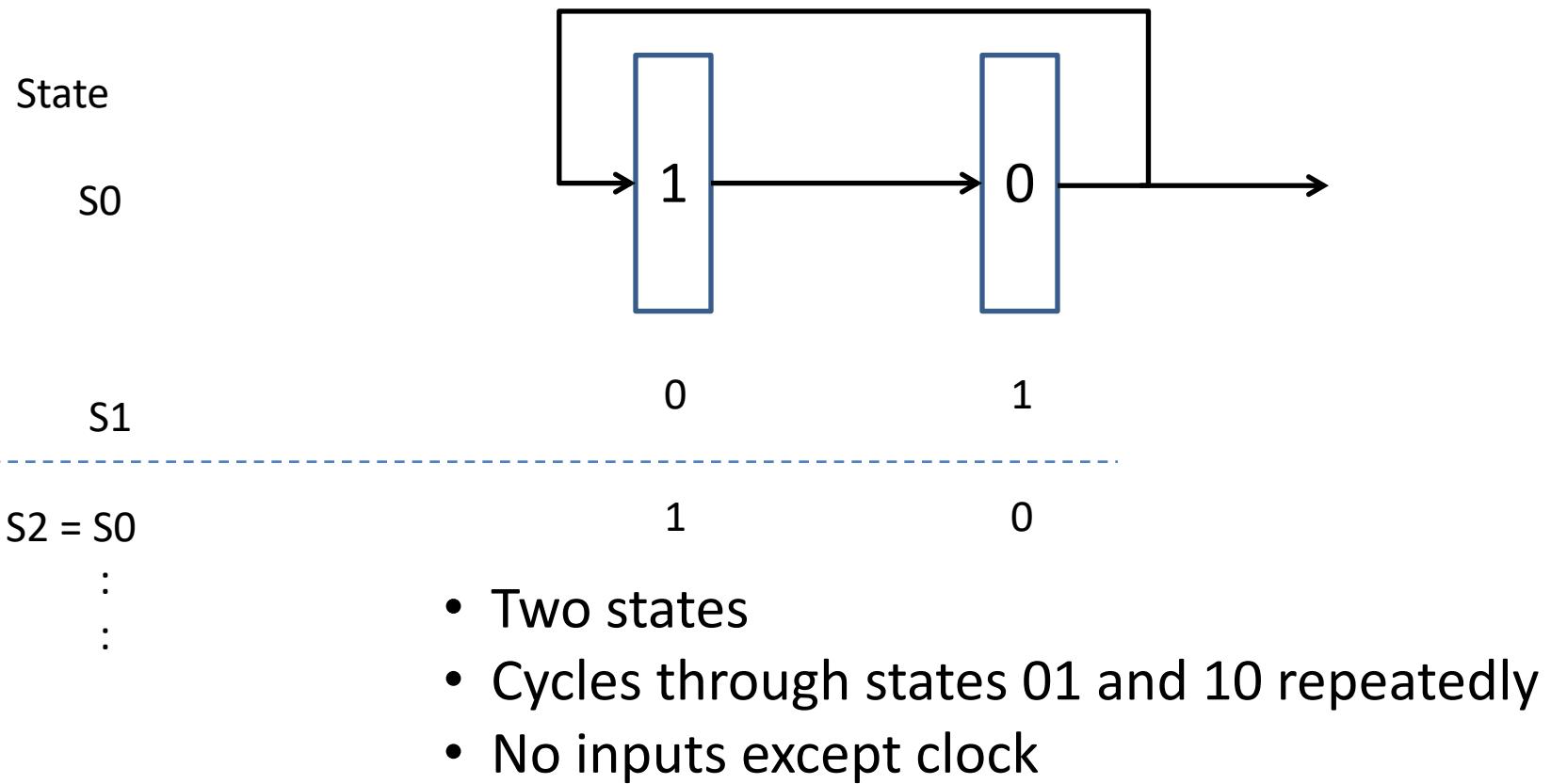
Test Pattern Generation for BIST

Assume CUT = n -input, m -output combinational circuit

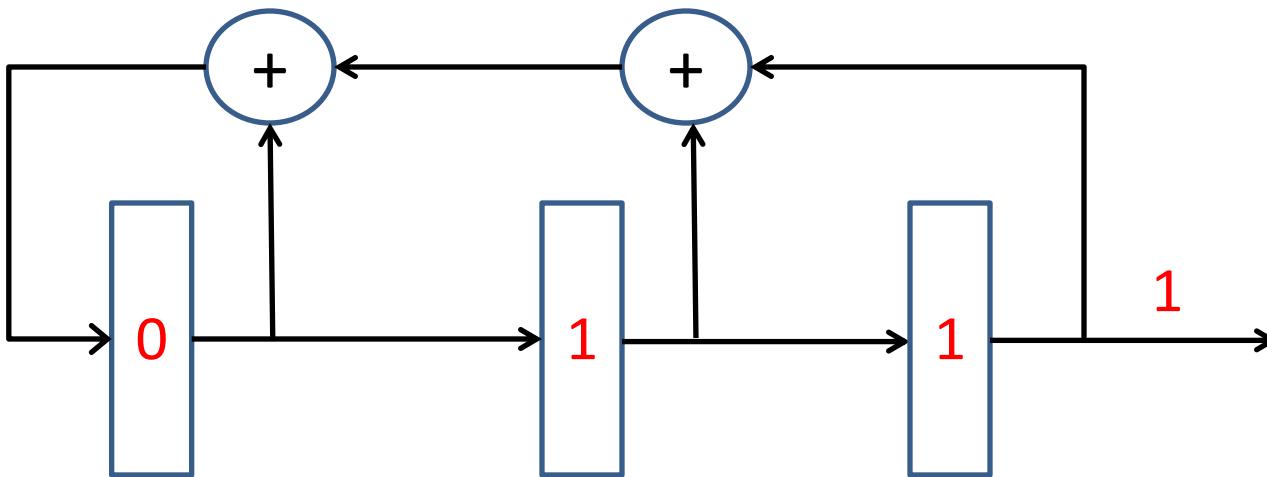
- **Exhaustive Testing**
 - Exhaustive test-pattern generators – expensive
- **Pseudo-random Testing**
 - Weighted test generator
 - Adaptive test generator
- **Pseudo-exhaustive testing (cf. 8.3)**
 - Syndrome driver counter
 - Constant-weight counter
 - Combined LFSR and shift register
 - Combined LFSR and XOR gates
 - Condensed LFSR
 - Cyclic LFSR

Linear Feedback Shift Register (LFSR)

- LFSRs used for pseudo-random test vector generation and signature analysis



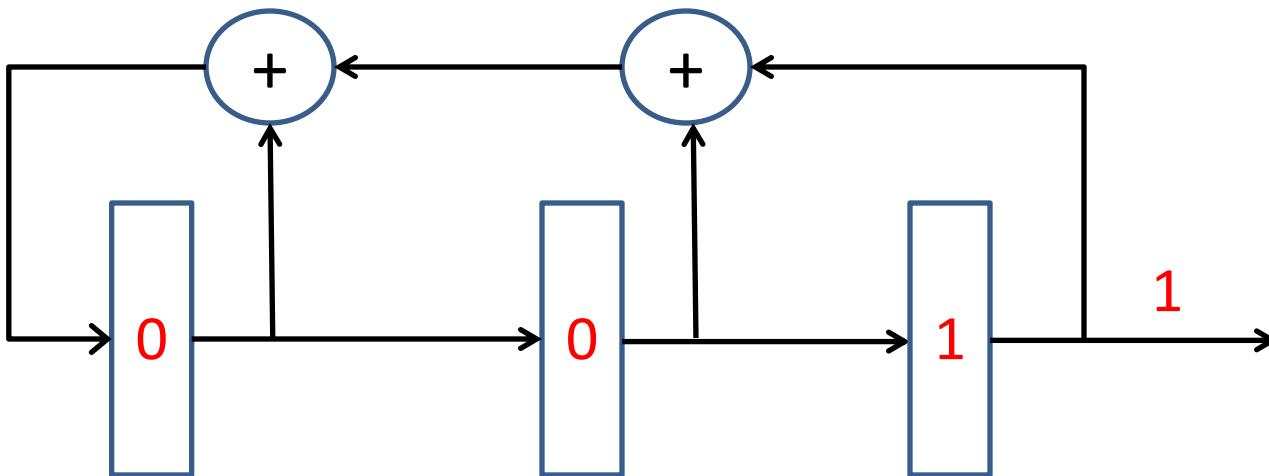
XOR Gates in Feedback



S0	0	1	1	
S1	0	0	1	
S2	1	0	0	
S3	1	1	0	
S4 = S0	0	1	1	
:				
:				

Cycles through
4 states

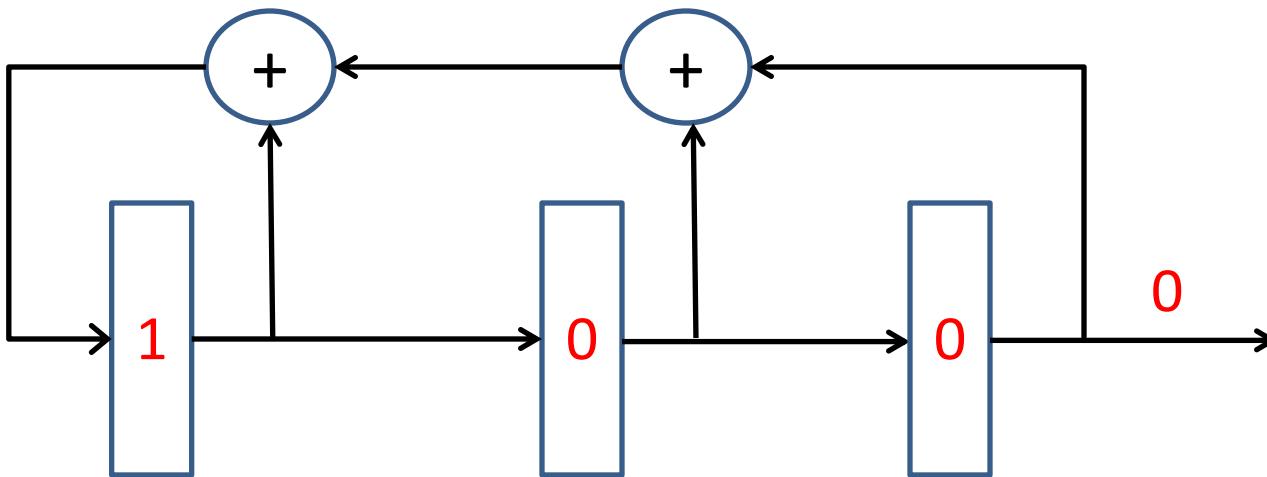
XOR Gates in Feedback



S0	0	1	1
S1	0	0	1
S2	1	0	0
S3	1	1	0
S4 = S0	0	1	1
:			
:			

Cycles through
4 states

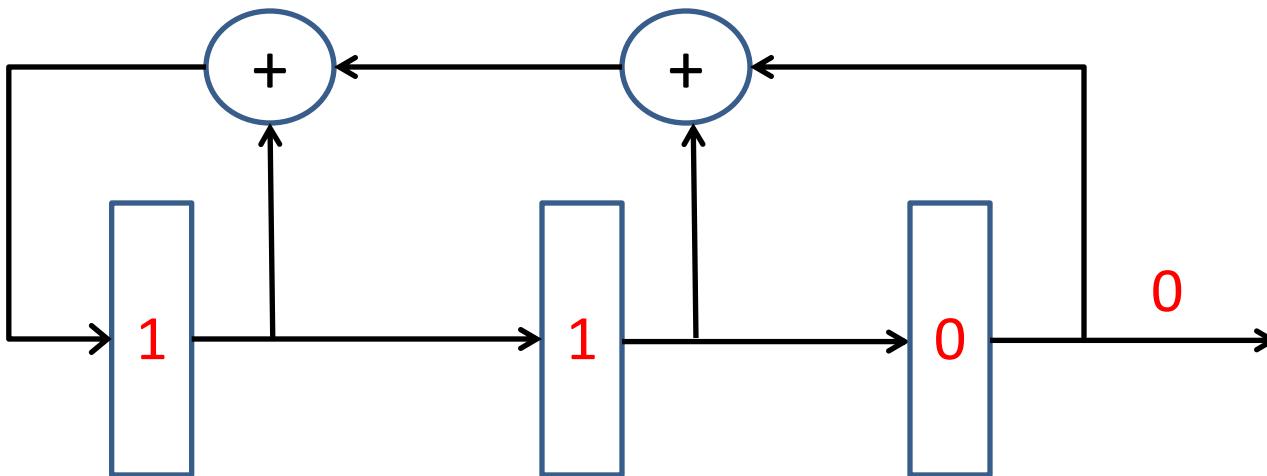
XOR Gates in Feedback



S0	0	1	1
S1	0	0	1
S2	1	0	0
S3	1	1	0
S4 = S0	0	1	1
:			
:			

Cycles through
4 states

XOR Gates in Feedback



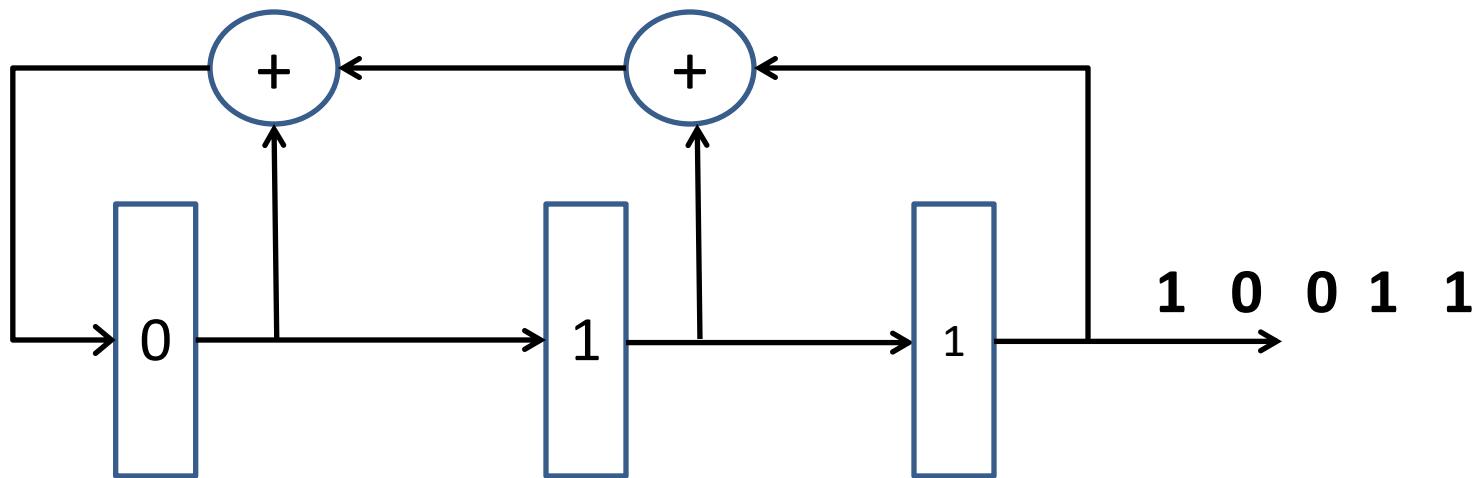
S0	0	1	1
S1	0	0	1
S2	1	0	0
S3	1	1	0
S4 = S0	0	1	1

Cycles through
4 states

:

:

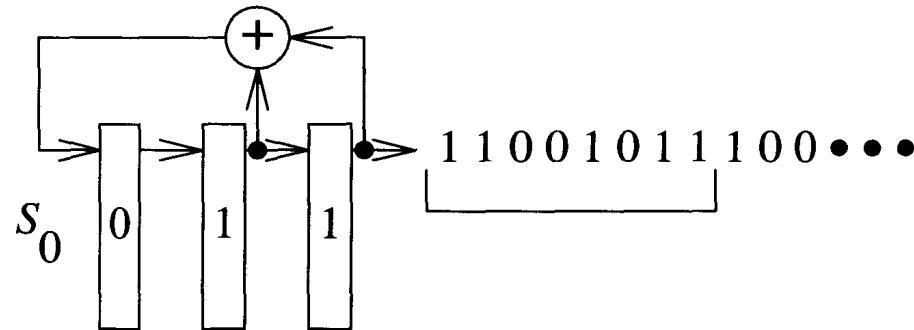
XOR Gates in Feedback



S0	0	1	1	}
S1	0	0	1	
S2	1	0	0	
S3	1	1	0	
S4 = S0	0	1	1	
:				
:				

Cycles through
4 states

Maximal Length LFSR



s_1	0	0	1
s_2	1	0	0
s_3	0	1	0
s_4	1	0	1
s_5	1	1	0
s_6	1	1	1
<hr/>			
$s_7 = s_0$	0	1	1
•			
•			
•			

Generates a cyclic sequence of length $2^n - 1$

All-0 initial state leads to a sequence of length 1.

Exhaustive Testing

- Test the n -input comb. circuit with 2^n inputs
- Binary counter can be used as TPG.
- Autonomous LFSR can also be used.
- Guarantees that all detectable faults that *do not* introduce sequential behavior will be detected
 - i.e. no bridging faults.
- Depending on clock rate, $n > 22$ is impractical
- Not used for sequential circuits

Pseudo-Random Testing

- Many characteristics of random patterns
- Generated deterministically => Repeatable
- With or without replacement
- **With replacement** = patterns can repeat
- **Without replacement** = unique patterns
(autonomous LFSR can be a source)
- Applicable to both comb. and seq. circuits

Bias in Pattern Generation

- Autonomous LFSR: 0's and 1's balanced in the output
- Sometimes we want a bias (say more 1's than 0's)
- Example: 4-input AND gate
 - Probability of an input set to 0 is $15/16$
 - With random inputs, hard to test other input s-a-0 or s-a-1 fault

Weighted & Adaptive Test Generation

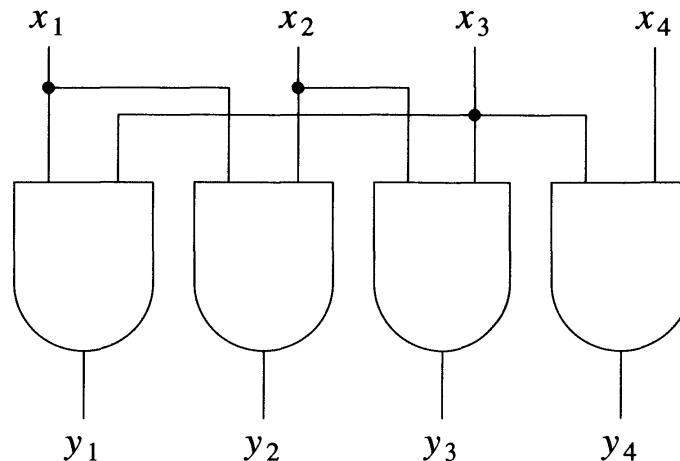
- **Weighted Test Generator**
 - Distribution of 0s and 1s -> not uniform
 - Can be constructed by LFSR + a combinational Circuit
 - When testing a circuit using WTG, preprocessing is carried out to determine weights
 - Therefore, each part of circuit can be tested with different distributions
- **Adaptive Test Generator**
 - Uses a WTG
 - Results of fault simulation used to modify weights
 - Efficient in terms of test length
 - Requires complex TPG hardware

Pseudo-Exhaustive Testing

- Achieves benefits of exhaustive testing *but with far fewer test patterns*
- Relies on circuit segmentation
- A segment = subcircuit of the CUT
- Attempts testing each segment exhaustively
- Segments need not be disjoint
- Forms of Segmentation
 1. Logical Segmentation
 - a. Cone Segmentation
 - b. Sensitized Path Segmentation
 2. Physical Segmentation

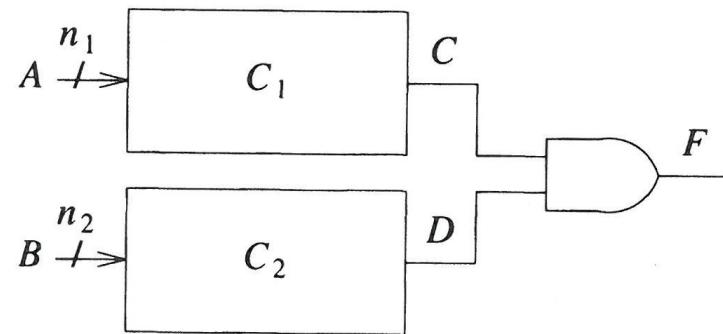
Cone Segmentation

- Cone segmentation of a m output circuit is logically segmented into m cones
- Cone = all logic associated with one output
- Each cone tested exhaustively
- All cones tested concurrently



Sensitized Path Segmentation

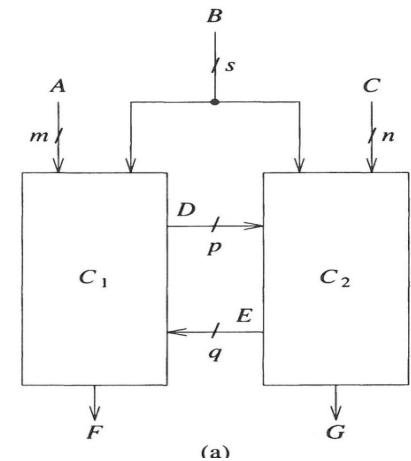
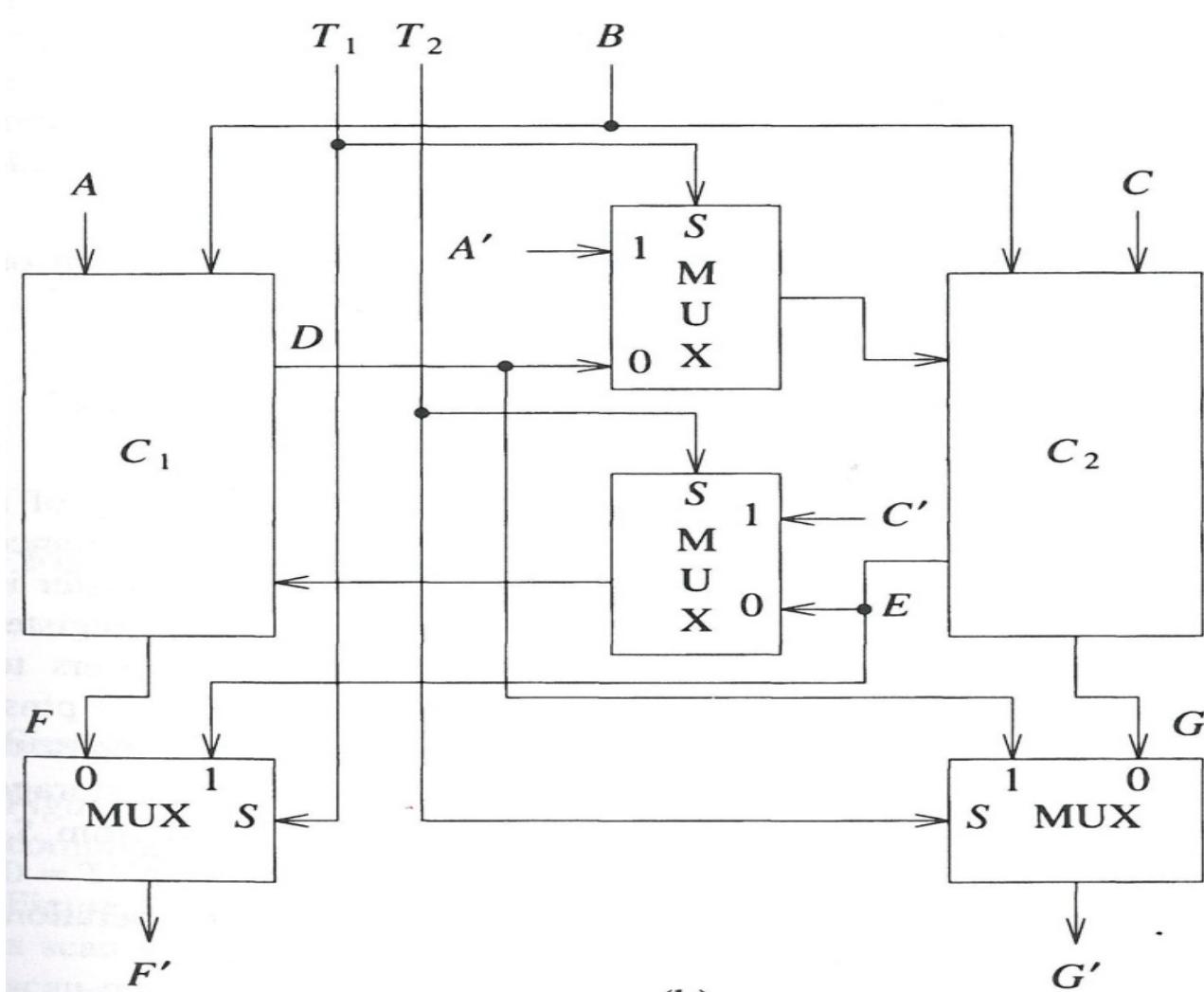
- Example:
 - C partitioned into C_1 and C_2
 - Set inputs to B such that D=1 and apply 2^{n_1} patterns to test C_1
 - Similarly test C_2
 - Need $2^{n_1} + 2^{n_2} + 1$ patterns instead of $2^{n_1+n_2}$



Physical Segmentation

- In large circuits, pseudo-exhaustive testing leads to large test sets
- Can employ physical segmentation
 - Partitioning: Circuit is divided into sub-circuits
 - Bypass Storage Cell
 - Normal mode: acts as a wire
 - Test mode: part of an LFSR

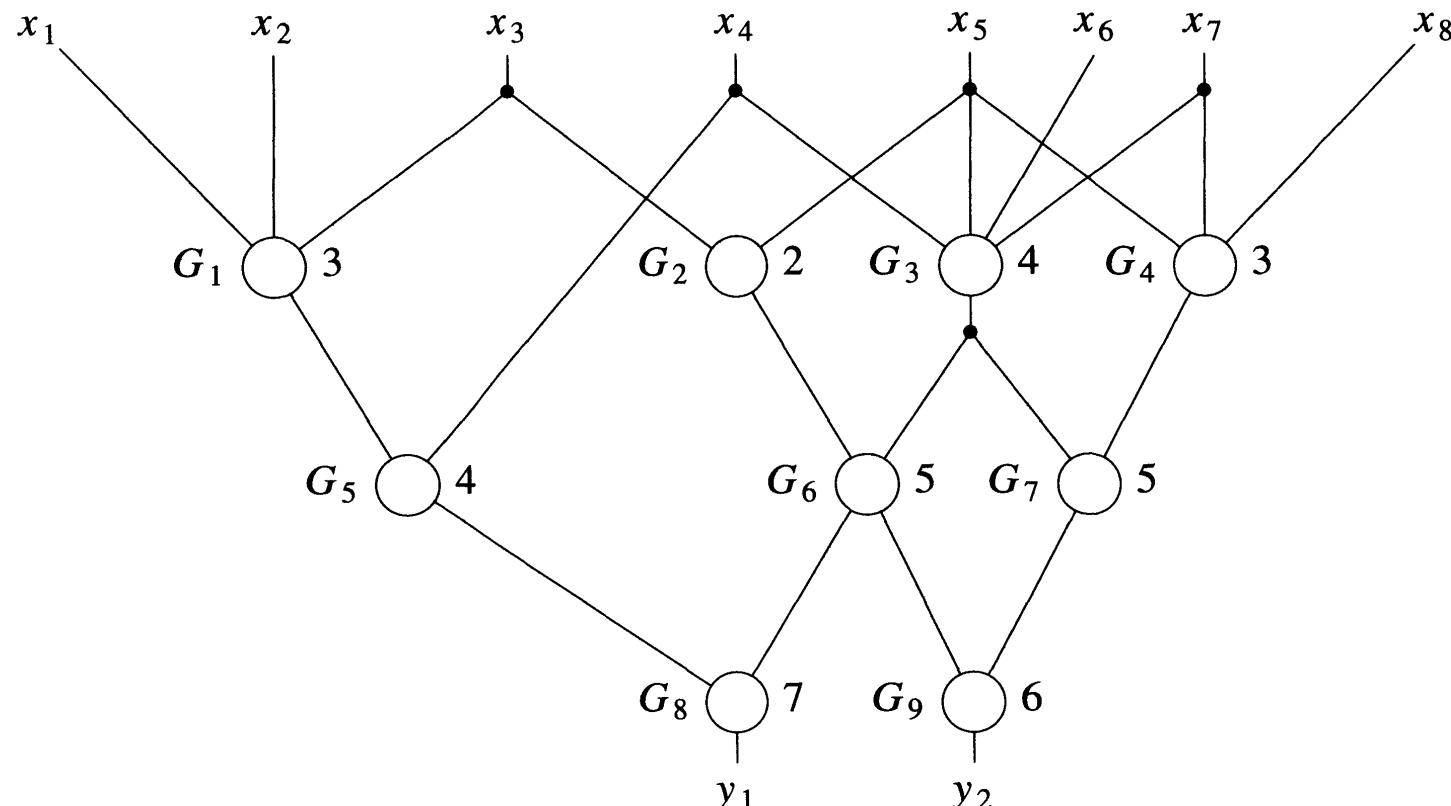
Physical Segmentation by Partitioning

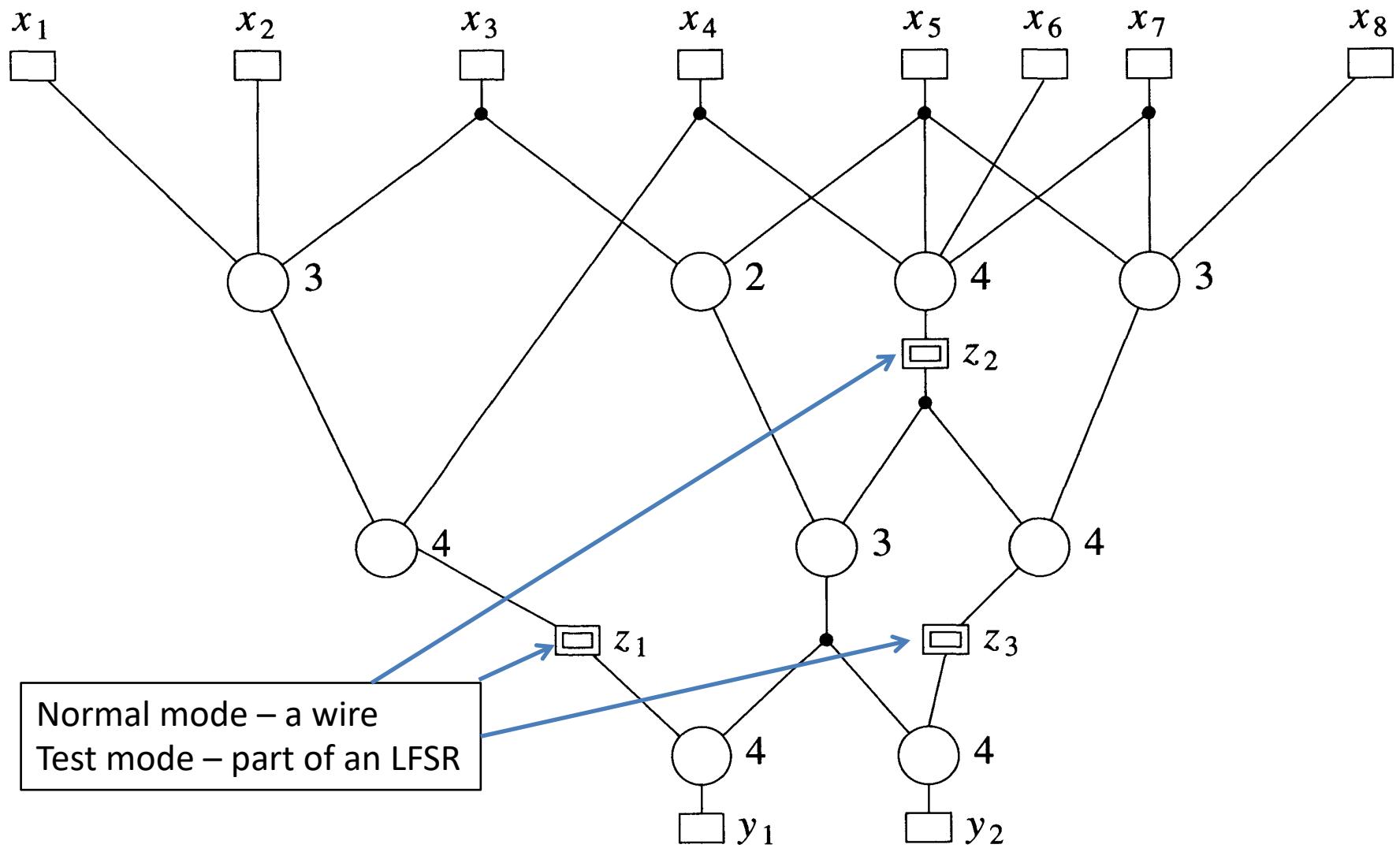


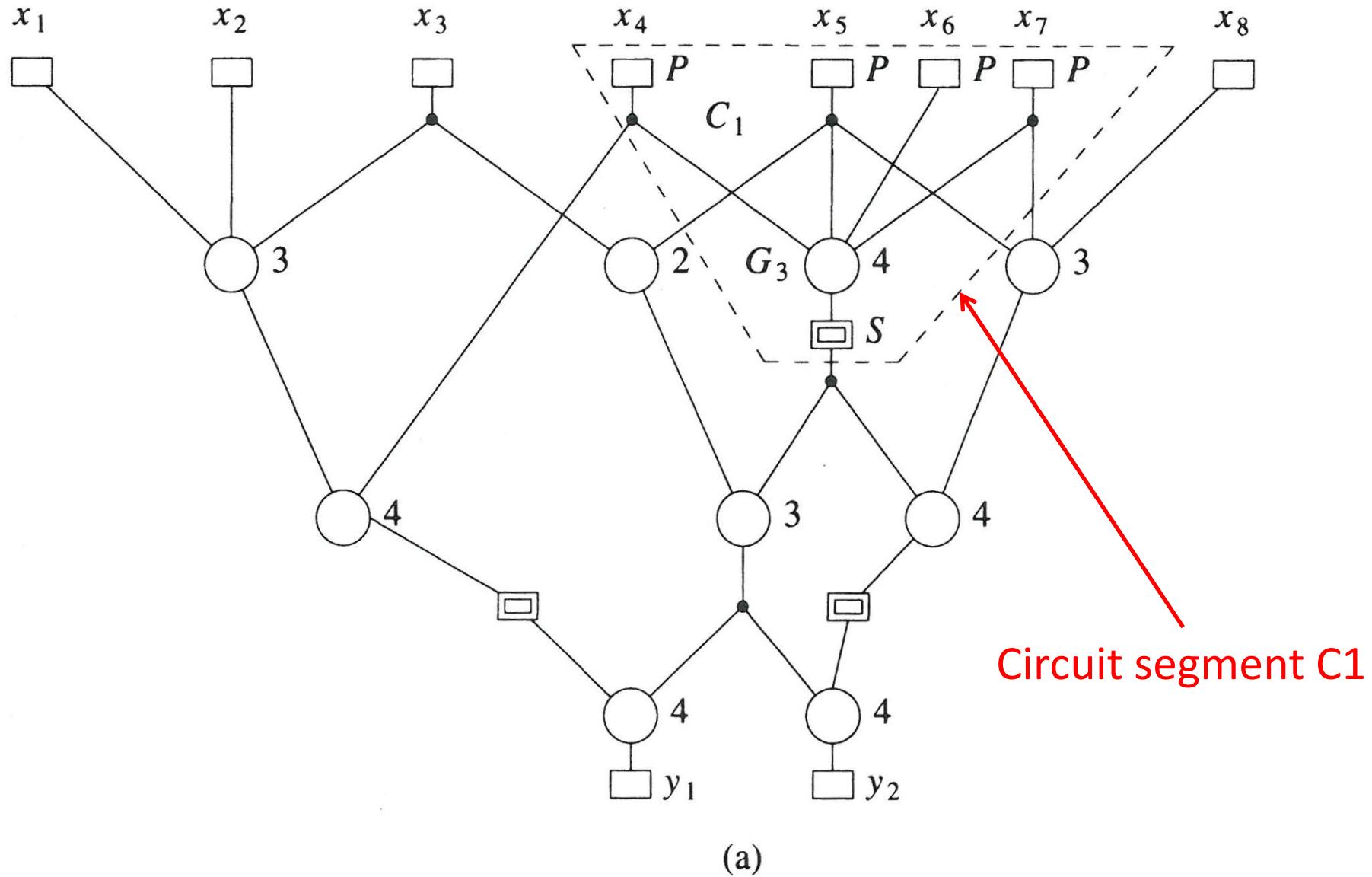
T_1	T_2	Mode
0	0	normal
0	1	test C_1
1	0	test C_2

Physical Segmentation by Storage Cells

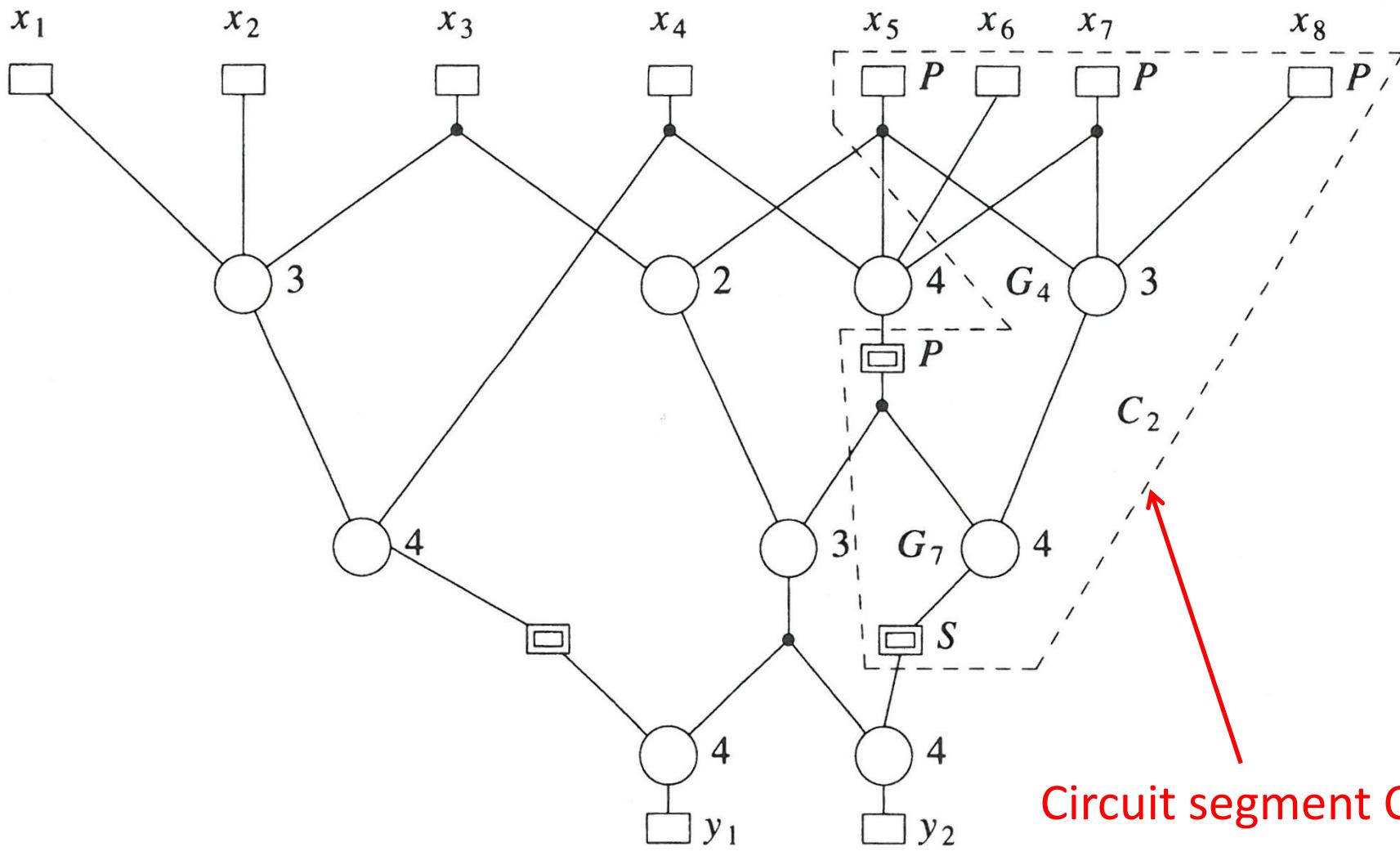
- Let us say we want to segment the following such that no signal is a function of more than 4 variables





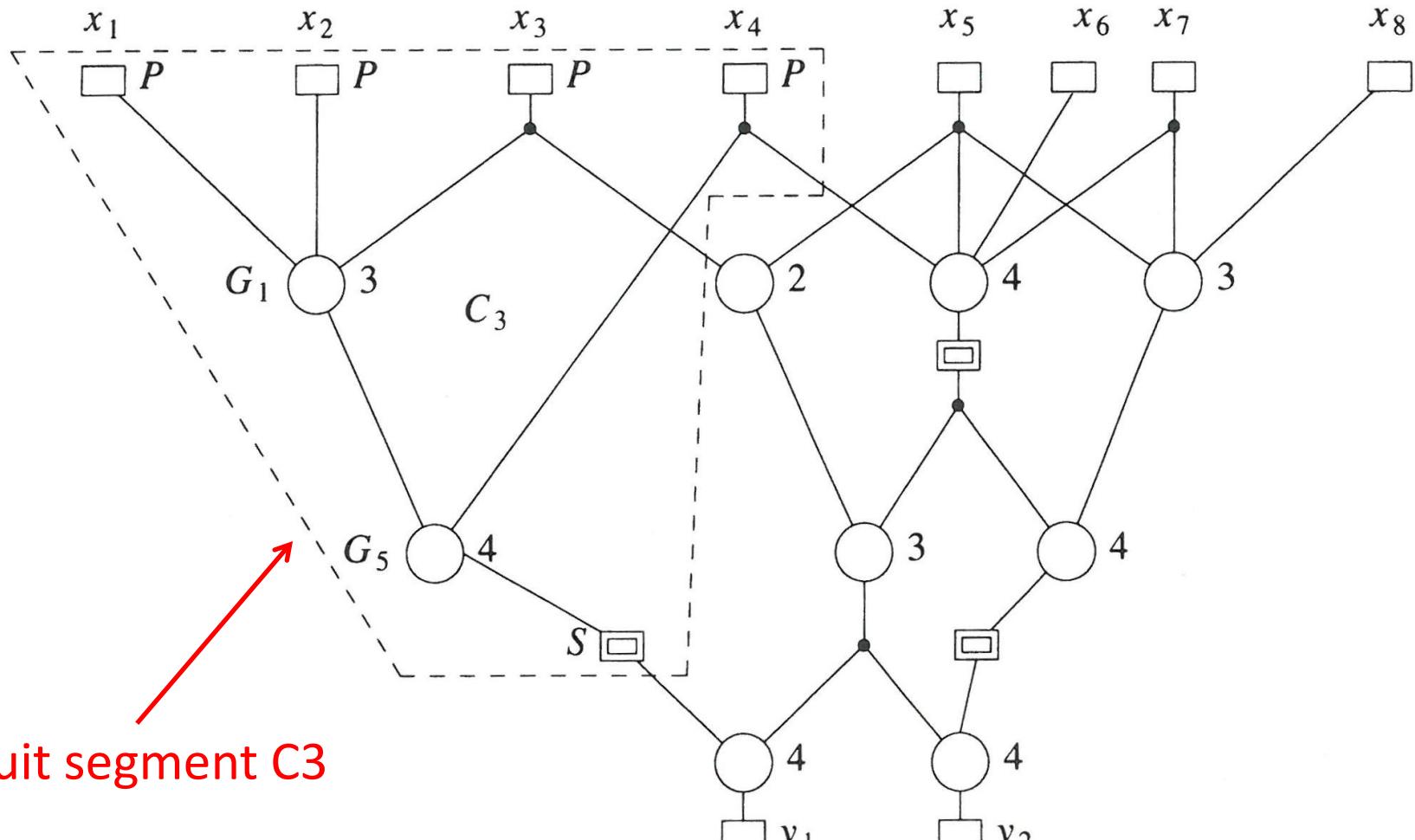


(a)

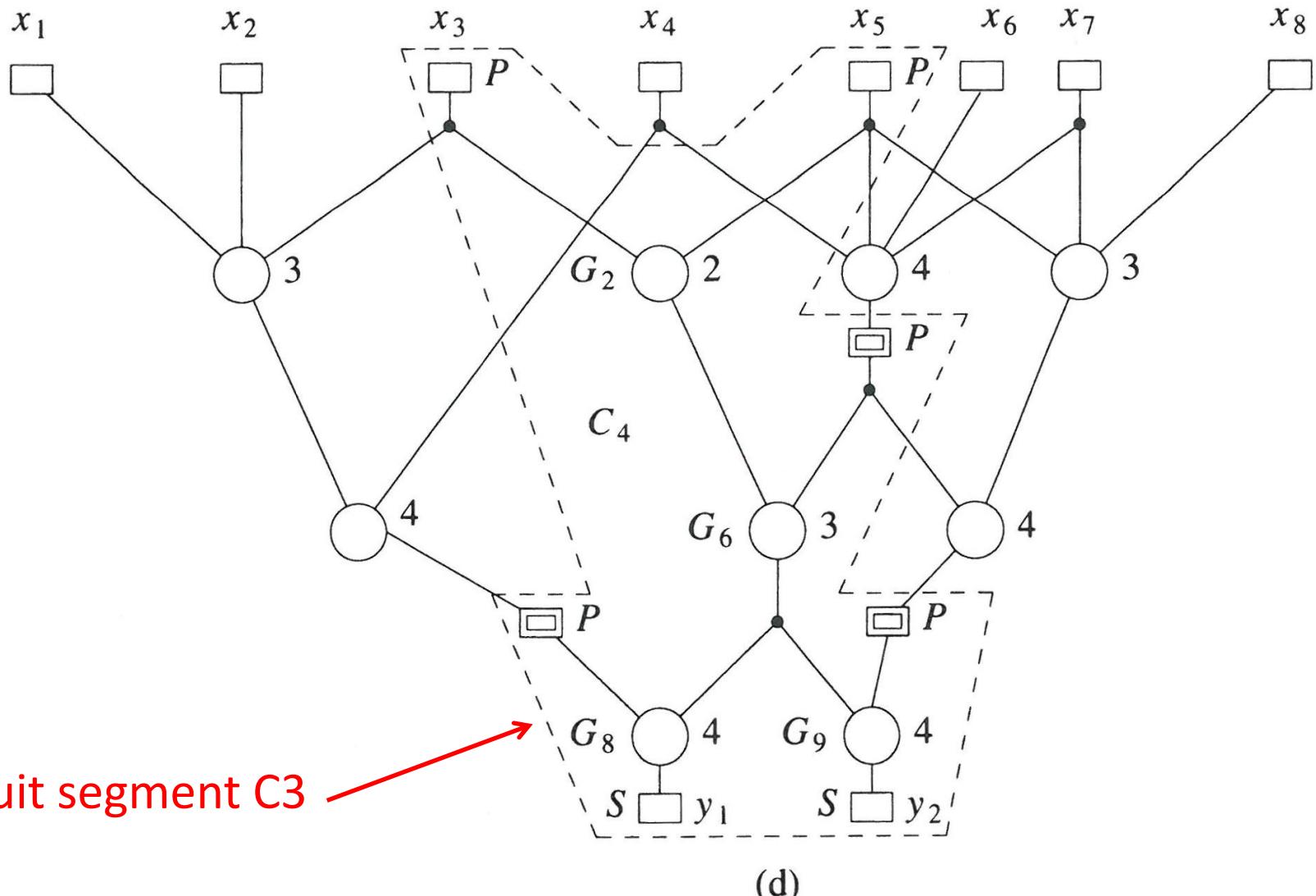


(b)

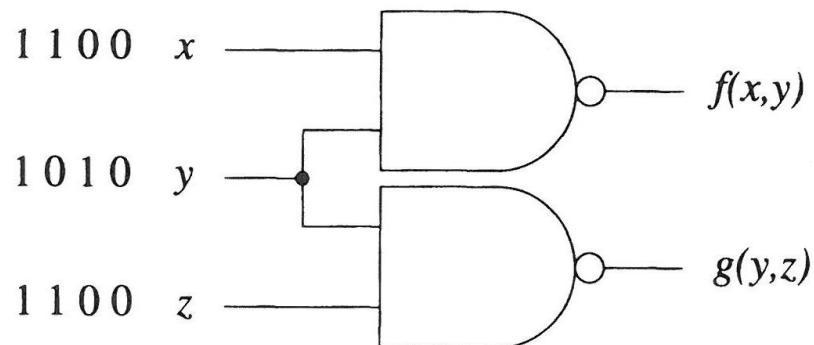
Circuit segment C2



(c)



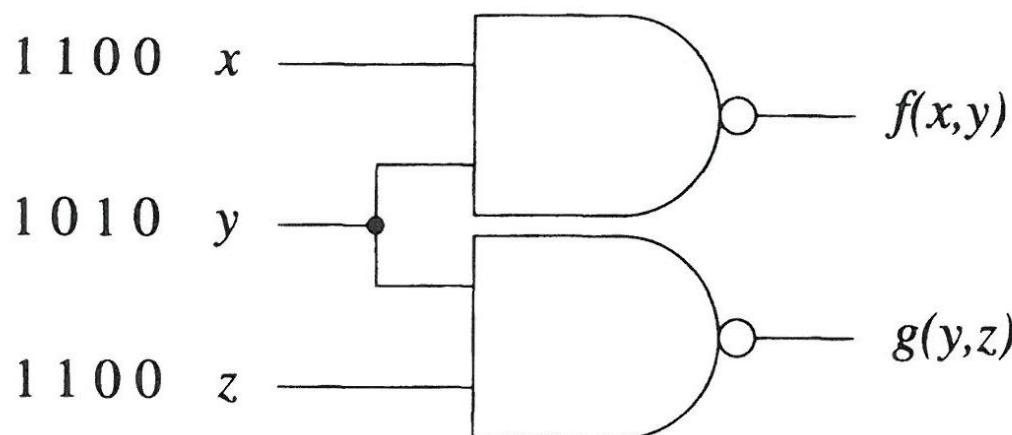
Identification of Test Signal Inputs



- f and g are functions of only two inputs each
- To exhaustively test the multiple function (f, g), we need 8 vectors
- Since no output is function of both x and z , same test data can be applied to both these lines
 - 2 test signals
 - 4 test vectors are sufficient

Maximal-Test-Concurrency (MTC) circuit

- A circuit is said to be a *maximal-test-concurrency* (MTC) circuit, if the minimal number of required test signals is equal to the maximum number of inputs upon which any output depends.



Non-MTC circuit

- All three signals are required, can still be tested exhaustively by just four test patterns

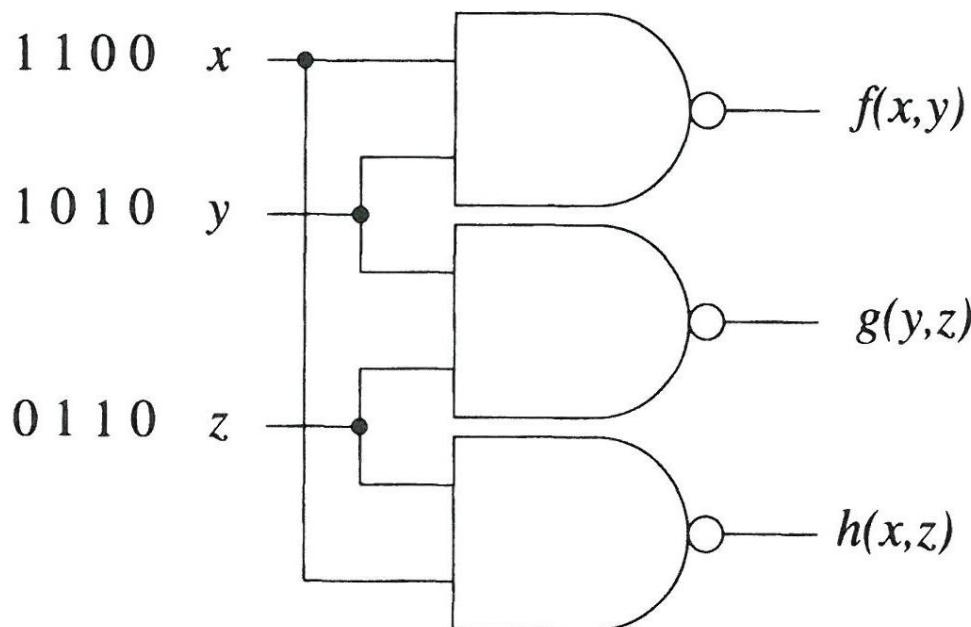


Figure 11.6 A nonmaximal-test-concurrency circuit with verification test input

TPG – Syndrome-Driver Counter

- If $(n-p)$ input share test signals with p other inputs, at most 2^p tests are required.
 - n : # of inputs
- $n = 4$, $p = 3$, $w=2$
 - w = # of inputs of a segment
- At most 8 tests are needed.
- 0000 & 1111 are not needed

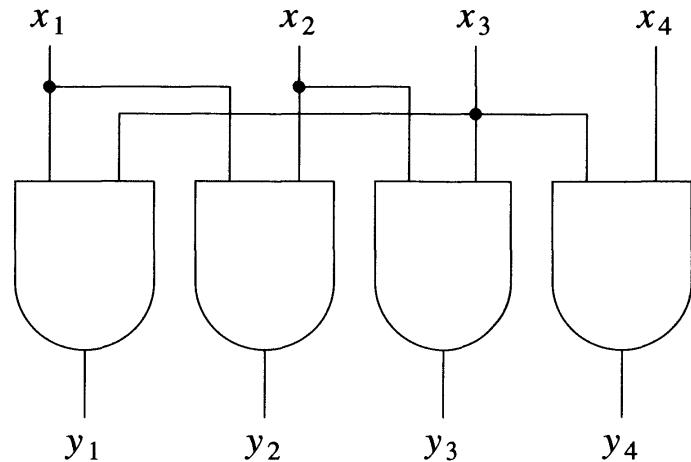


Figure 11.3 A (4,2)-CUT

TPG – Constant-Weight Counter

- A (n, w) circuit can be tested by a counter implementing by *w-out-of-K* 1100
1010
- Complexity of the counter can be high for large w 1001
0110
010 1
0011

TPG – Combined LFSR/SR

- (n, w) circuit
- Lower cost
- May generate more tests
- # of tests near minimal when $w \ll n/2$

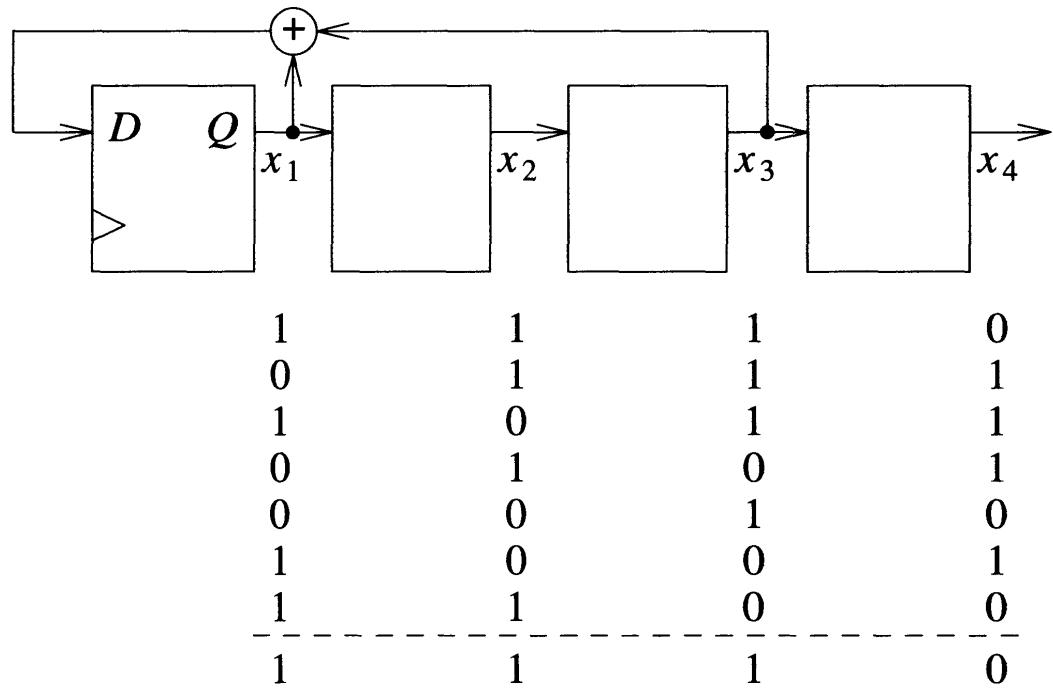


Figure 11.12 A 4-stage LFSR/SR for a $(4,2)$ -CUT

TPG – Condensed LFSR

- (n, w) circuit
- Can produce efficient test set when $w \geq n/2$
- But produce more test than combined LFSR/SR
- What patterns does it generate?

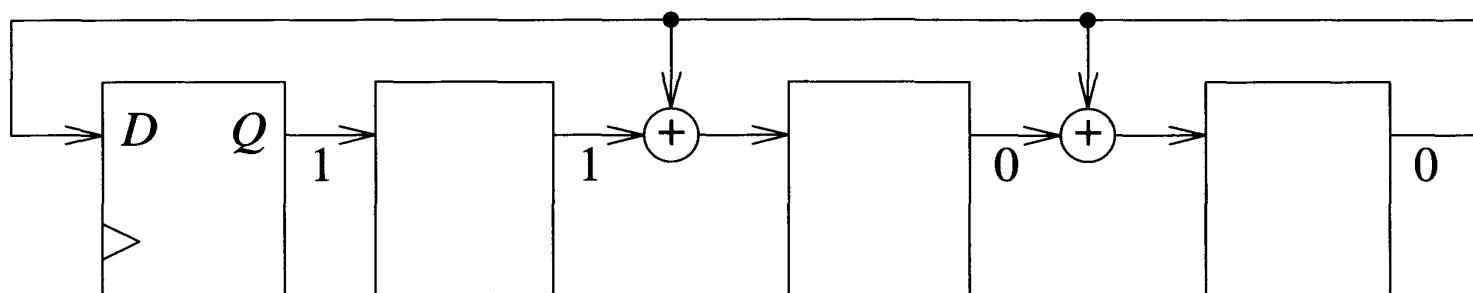


Figure 11.15 A condensed LFSR for a (4,2)-CUT

Generic Off-line BIST Architectures

- Off-line BIST Architectures
 1. Centralized or Distributed
 2. Embedded or Separate
- BIST architecture elements:
 1. Test pattern generators
 2. Output response analyzers
 3. Circuit under test
 4. Distribution system (DIST) for transmitting date from TPGs to CUTs and from CUTs to ORAs
 5. BIST Controller

BIST Controller

During testing BIST Controller can carry out one or more functions:

- 1. Single-step** the CUTs through some test sequence
- 2. Inhibit** system clocks and control test clocks
- 3. Communicate** with other test controllers
- 4. Control** the operation of self-test (seeding of registers, number of test patterns processed, etc.)

Centralized and BIST Architecture

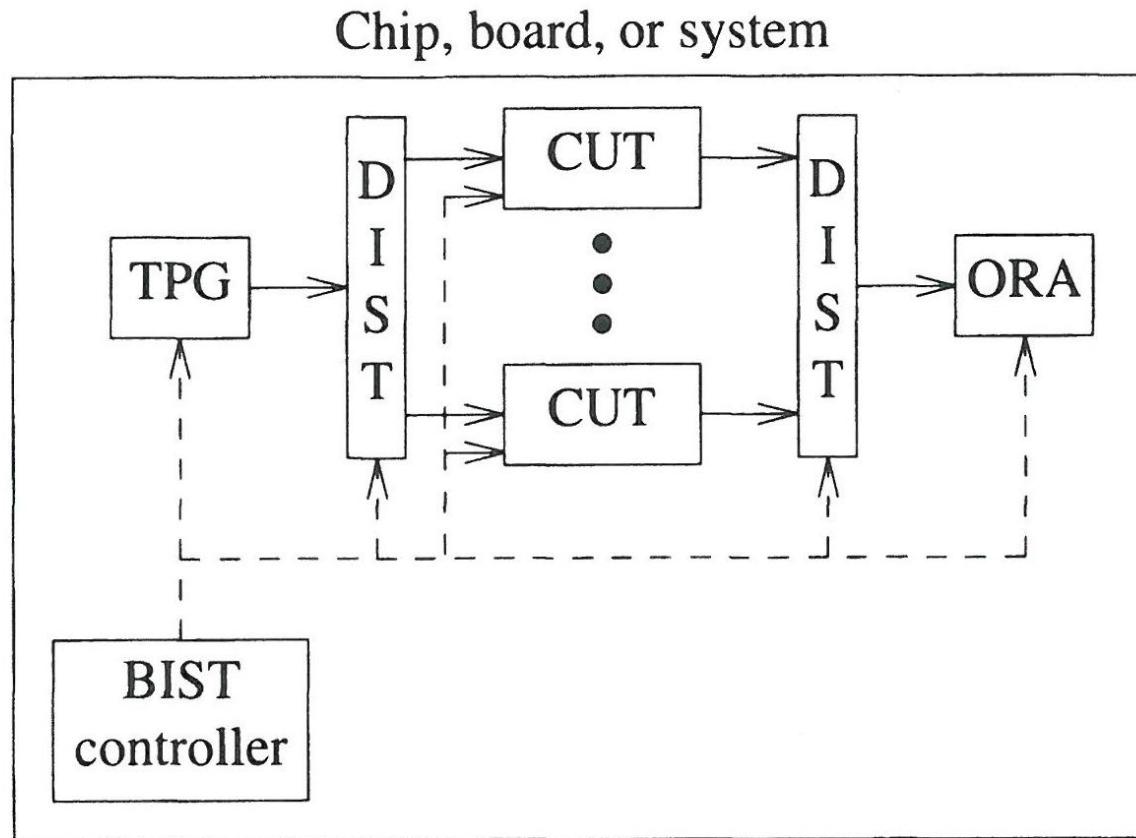


Figure 11.18 Generic form of centralized and separate BIST architecture

Distributed and Separate BIST

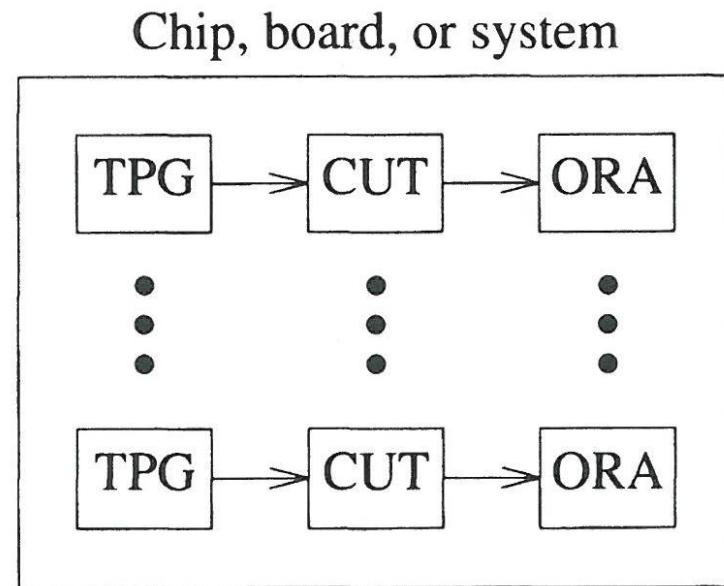


Figure 11.19 Generic form of distributed and separate BIST architecture

Distributed and Embedded BIST

- TPG and ORA configured from within CUT
- Complex design to control

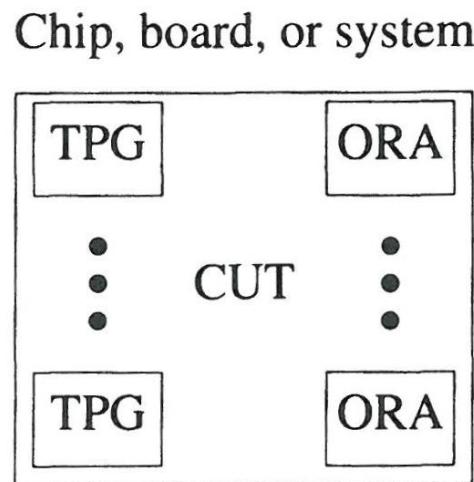


Figure 11.20 Generic form of distributed and embedded BIST architecture

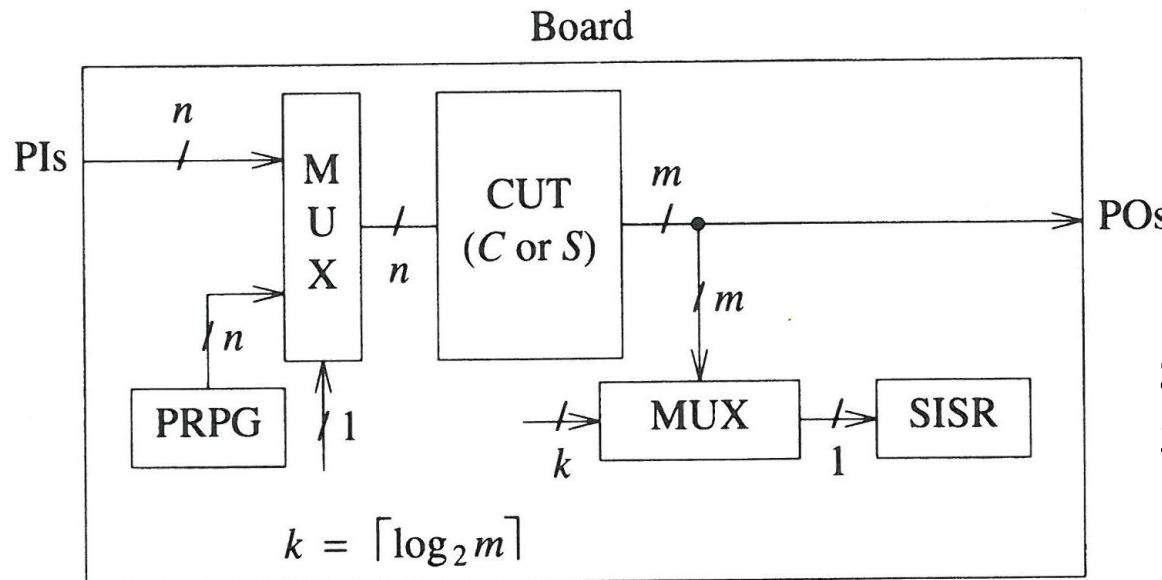
BIST Architecture

When choosing BIST architecture, following factors need to be considered:

1. Degree of test parallelism
2. Fault coverage
3. Level of packaging
4. Test time
5. Physical constraints
6. Complexity of replaceable units
7. Factory and field test-and-repair strategy
8. Performance degradation

Some Example BIST Architectures

1. Centralized and Separate Board-Level BIST



SISR: Single Input Signature Analyzer
PRPG: Pseudorandom Pattern Generator

Figure 11.21 A centralized and separate BIST architecture (CSBL)

Some Example BIST Architectures

2. LSSD On-Chip Self-Test (LOCST)

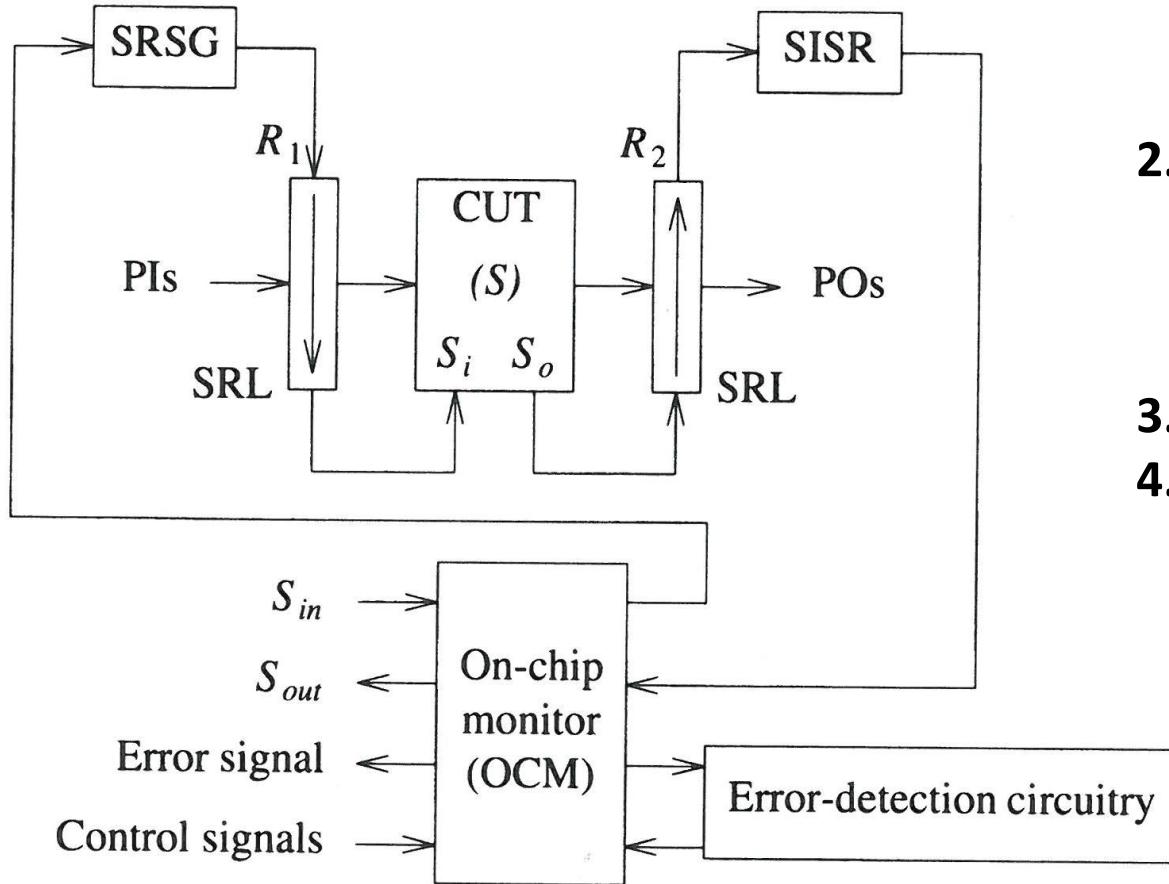


Figure 11.25 The LOCST architecture

- | | <u>Test Process</u> |
|-----------------------------------|--|
| 1. Initialize | Scan path loaded with seed via S_{in} |
| 2. Activate Self-test mode | a) Disable sys clks on R_1 and R_2
b) Enable LFSR operation |
| 3. Execute Self-test | |
| 4. Check Result | Compare final value of SISR with known good signature |
- LSSD:** Level Sensitive Scan Design
SRSG: Shift Register Sequence Generator

Some Example BIST Architectures

3. Random Test Data (RTD) BIST

- Previous archs – entire scan path be loaded with new data to apply a single test pattern to CUT; RTD overcomes this
- Test process:

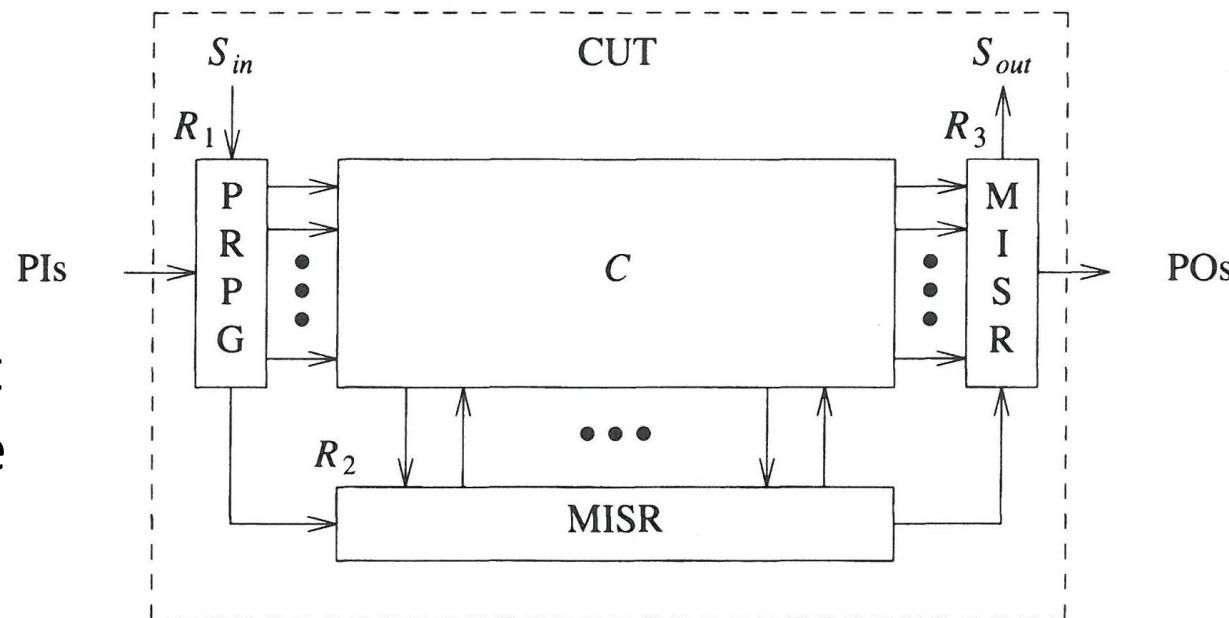
a) R1, R2, and R3

set to scan mode
and a seed pattern
is loaded

b) Registers put to test
mode and held while
circuit is tested

c) For each clock cycle,

R1 and R2 generate a new
test pattern, and R2 and R3 operate as a MISR



Built-In Logic Block Observation (BILBO) Register

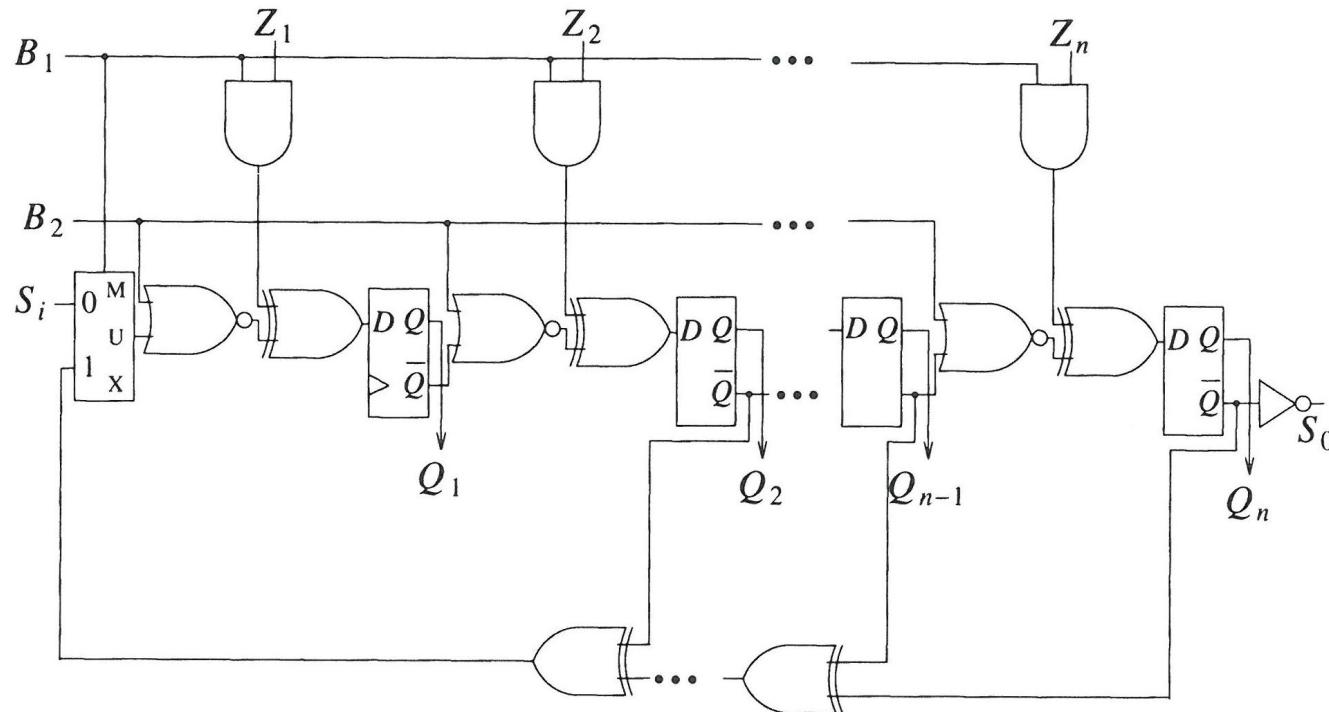
Operates in four modes:

$B_1 = B_2 = 1$ - Normal Mode (parallel load register)

$B_1 = B_2 = 0$ – Shift Register Mode

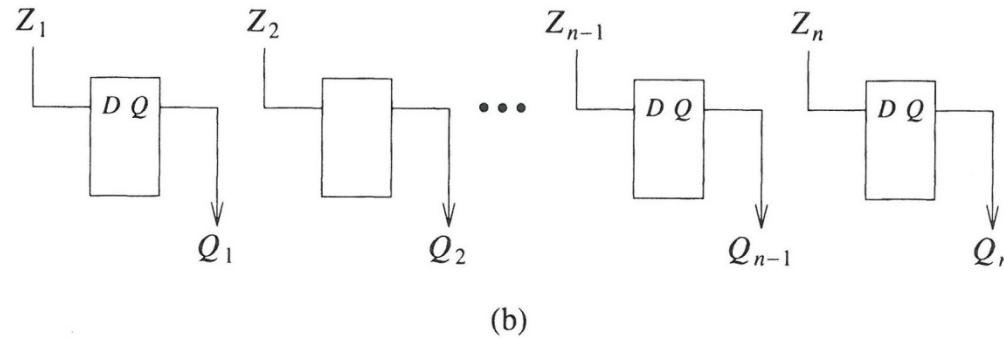
$B_1 = 1, B_2 = 0$ -- LFSR (test) mode

$B_1 = 0, B_2 = 1$ -- all storage cells - reset



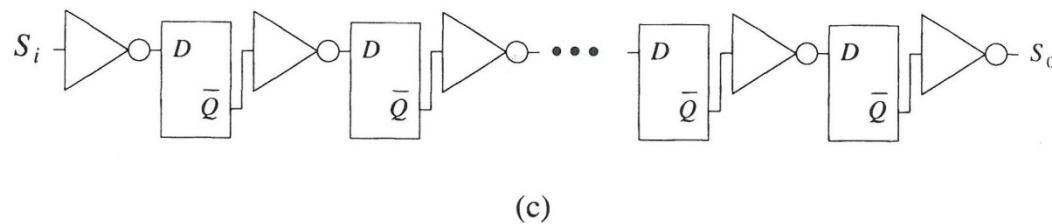
BIBLO Register Modes

B1 = B2 = 1
Normal Mode



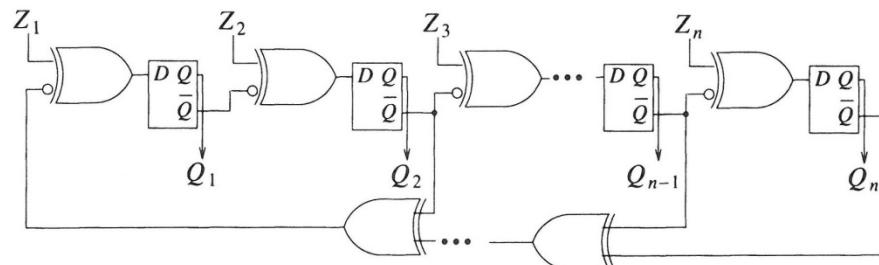
(b)

B1 = B2 = 0
**Shift Register
Mode**



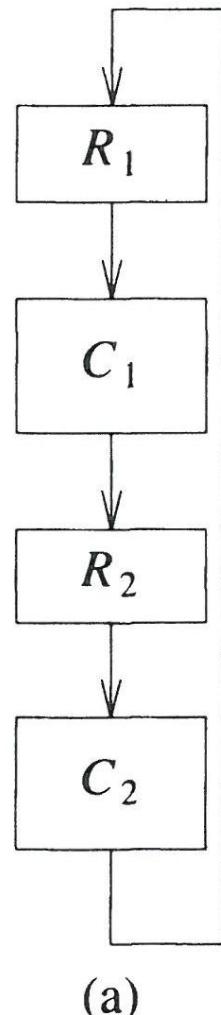
(c)

B1 = 1, B2 = 0
LFSR Mode



(d)

BIST Design with BILBO Registers



- To test C_1
 1. R_1 and R_2 are seeded
 2. R_1 into PRPG mode, R_2 into MISR mode
 3. Hold inputs to R_1 to value 0 so that LFSR (R_1) acts as a PRPG
 4. Run for N clock cycles
- If C_1 is not too large, C_1 can be tested exhaustively (except for all-zero pattern)
- At the end of test session, R_2 scanned out and signature checked
- Need two test sessions, one for C_1 and other for C_2

Summary

- Built-In Self Test – can be offline or online
- Needs test pattern generators (TPGs) and output response analyzers (ORAs)
- Linear Feedback Shift Registers (LFSRs) can be used as both as a TPG and as an ORA
- Offline BIST architectures can be centralized or distributed, embedded or separate