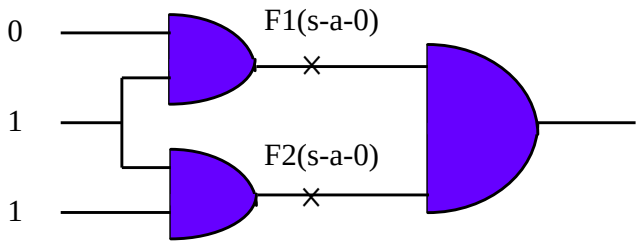
Characteristics of Fault Simulation

- Fault activity with respect to fault-free circuit
  - is often sparse both in time and space.
- For example
  - F1 is not activated by the given pattern, while F2 affects only the lower part of this circuit.



Fault Simulation Techniques

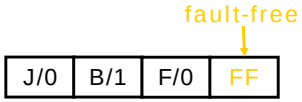
- Parallel Fault Simulation
- Deductive Fault Simulation

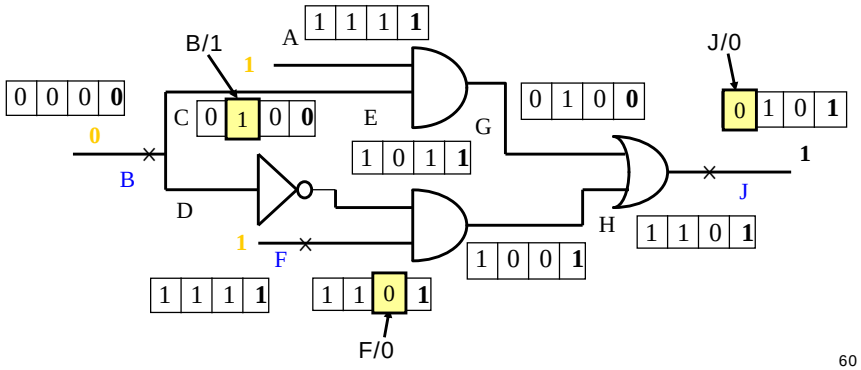
Parallel Fault Simulation

- Simulate multiple circuits simultaneously
  - The inherent parallel operation of computer words to simulate faulty circuits in parallel with fault-free circuit
  - The number of faulty circuits or faults can be processed simultaneously is limited by the word length, e.g., 32 circuits for a 32-bit computer
- Complication
  - An event or a value change of a single faulty or fault-free circuit leads to the computation of an entire word
  - The fault-free logic simulation is repeated for each pass

Parallel Fault Simulation

- Example
  - Consider three faults: (J s-a-0, B s-a-1, and F s-a-0)
  - Bit-space: (FF denotes fault-free)



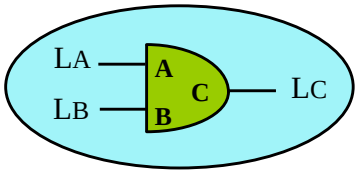


Deductive Fault Simulation

- Simulate all faulty circuits in one pass
  - For each pattern, sweep the circuit from PIs to POs.
  - During the process, a list of faults is associated with each wire
  - The list contains faults that would produce a fault effect on this wire
  - The union fault list at every PO contains the detected faults by the simulated input vector
- Main operation is fault list propagation
  - Depending on gate types and values
  - The size of the list may grow dynamically, leading to the potential memory explosion problem

Illustration of Fault List Propagation

Consider a two-input AND-gate:



Non-controlling case:

Case 1: A=1, B=1, C=1 at fault-free,
$L_C = L_A \cup L_B \cup \{C/0\}$

Controlling cases:

Case 2: A=1, B=0, C=0 at fault-free,
$L_C = (\overline{L_A} \cap L_B) \cup \{C/1\}$

Case 3: A=0, B=0, C=0 at fault-free,
$L_C = (L_A \cap L_B) \cup \{C/1\}$

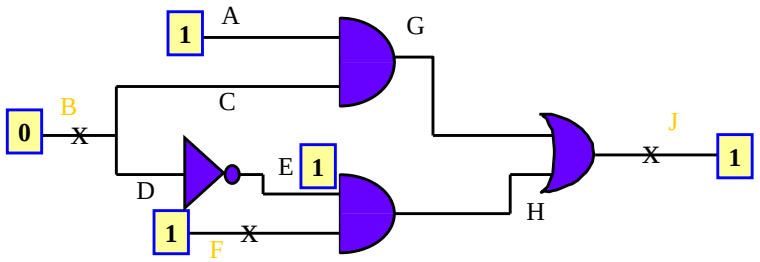
$\overline{L_A}$ is the set of all faults not in L_A

Rule of Fault List Propagation

	a	b	z	Output fault list
AND	0	0	0	$\{L_a \cap L_b\} \cup z_1$
	0	1	0	$\{L_a - L_b\} \cup z_1$
	1	0	0	$\{L_b - L_a\} \cup z_1$
	1	1	1	$\{L_a \cup L_b\} \cup z_0$
OR	0	0	0	$\{L_a \cup L_b\} \cup z_1$
	0	1	1	$\{L_b - L_a\} \cup z_0$
	1	0	1	$\{L_a - L_b\} \cup z_0$
	1	1	1	$\{L_a \cap L_b\} \cup z_0$
NOT	0		1	$L_a \cup z_0$
	1		0	$L_a \cup z_1$

Deductive Fault Simulation

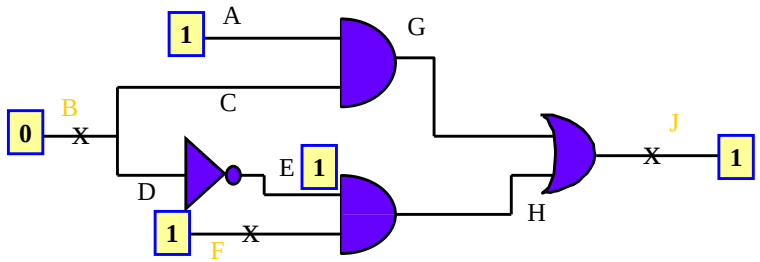
- Example (1/4)
  - Consider 3 faults: B/1, F/0, and J/0 under (A,B,F) = (1,0,1)



Fault list at PIs:
$L_B = \{B/1\}$, $L_F = \{F/0\}$, $L_A = \varnothing$, $L_C = L_D = \{B/1\}$

Deductive Fault Simulation

- Example (2/4)
  - Consider 3 faults: B/1, F/0, and J/0 under (A,B,F) = (1,0,1)



$L_B = \{B/1\}$, $L_F = \{F/0\}$, $L_A = \varnothing$, $L_C = L_D = \{B/1\}$

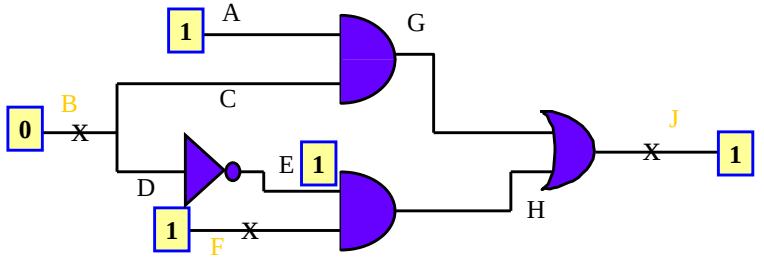
Fault lists at G and E:

$L_G = (L_A \cap L_C) \cup G/1 = \{B/1, G/1\}$

$L_E = (L_D) \cup E/0 = \{B/1, E/0\}$

Deductive Fault Simulation

- Example (3/4)
  - Consider 3 faults: B/1, F/0, and J/0 under (A,B,F) = (1,0,1)



$L_B = \{B/1\}$, $L_F = \{F/0\}$, $L_A = \varnothing$, $L_C = L_D = \{B/1\}$,

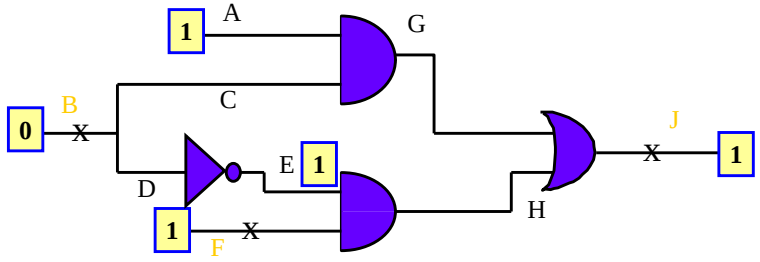
$L_G = \{B/1, G/1\}$, $L_E = \{B/1, E/0\}$

Fault list at H:

$L_H = (L_E \cup L_F) \cup L_H = \{B/1, E/0, F/0, H/0\}$

Deductive Fault Simulation

- Example (4/4)
  - Consider 3 faults: B/1, F/0, and J/0 under (A,B,F) = (1,0,1)



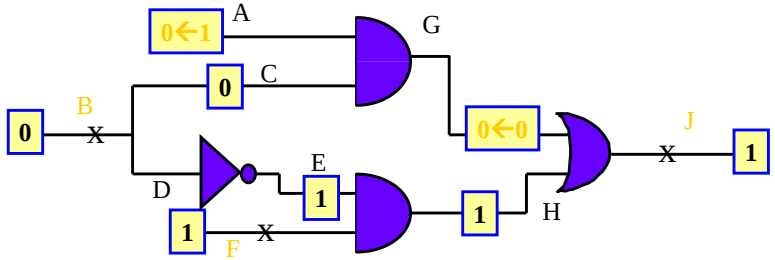
$L_B = \{B/1\}$, $L_F = \{F/0\}$, $L_A = \varnothing$, $L_C = L_D = \{B/1\}$, $L_G = \{B/1, G/1\}$, $L_E = \{B/1, E/0\}$, $L_H = \{B/1, E/0, F/0, H/0\}$

Final fault list at PO J:

$L_J = (L_H - L_G) \cup L_J = \{E/0, F/0, J/0\}$

Deductive Fault Simulation

- Example (cont'd)
  - Consider 3 faults: B/1, F/0, and J/0 under (A,B,F) = (0,0,1)



Event driven updates:

$L_B = \{B/1\}$, $L_F = \{F/0\}$, $L_A = \varnothing$, $L_C = L_D = L_E = \{B/1\}$,

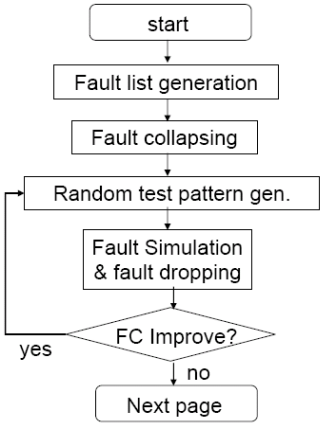
$L_G = \{G/1\}$, $L_H = \{B/1, F/0\}$, $L_J = \{B/1, F/0, J/0\}$

Outline

- Fault Modeling
- Fault Simulation
- Automatic Test Pattern Generation (ATPG)
  - Functional approach
    - Boolean difference
  - Structural approach
    - D-algorithm
    - PODEM
- Design for Testability

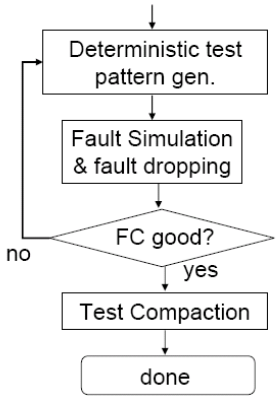
Typical ATPG Flow

- 1st phase: random test pattern generation



Typical ATPG Flow (cont'd)

- 2nd phase: deterministic test pattern generation



Test Pattern Generation

- The test set T of a fault α with respect to some PO z can be computed by

  $$T(x) = z(x) \oplus z_\alpha(x)$$

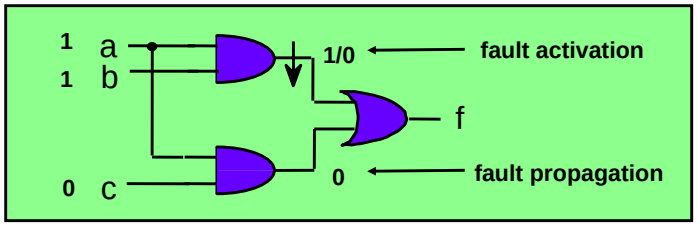
- A test pattern can be fully specified or partially specified depending on whether the values of PIs are all assigned
  - Example

abc	z	z_α
000	0	0
001	0	0
010	0	0
011	0	0
100	0	0
101	1	1
110	1	0
111	1	0

Input vectors (1,1,0) and (1,1,-) are fully and partially specified test patterns of fault α, respectively.

Structural Test Generation
D-Algorithm

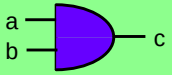
- Test generation from circuit structure
- Two basic goals
  - (1) Fault activation (FA)
  - (2) Fault propagation (FP)
  - Both of which requires Line Justification (LJ), i.e., finding input combinations that force certain signals to their desired values
- Notations:
  - 1/0 is denoted as D, meaning that good-value is 1 while faulty value is 0
  - Similarly, 0/1 is denoted D'
  - Both D and D' are called fault effects (FE)



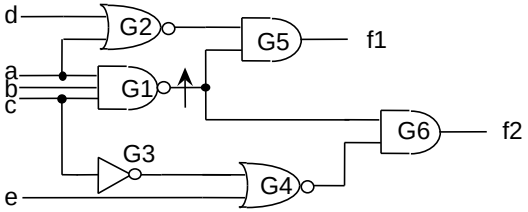
Structural Test Generation
D-Algorithm

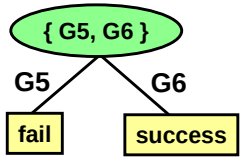
- Fault activation
  - Setting the faulty signal to either 0 or 1 is a Line Justification problem
- Fault propagation
  - (1) select a path to a PO → decisions
  - (2) Once the path is selected → a set of line justification (LJ) problems are to be solved
- Line justification
  - Involves decisions or implications
  - Incorrect decisions: need backtracking

To justify c=1 → a=1 and b=1 (implication)
To justify c=0 → a=0 or b=0 (decision)



Structural Test Generation
D-Algorithm: Fault Propagation



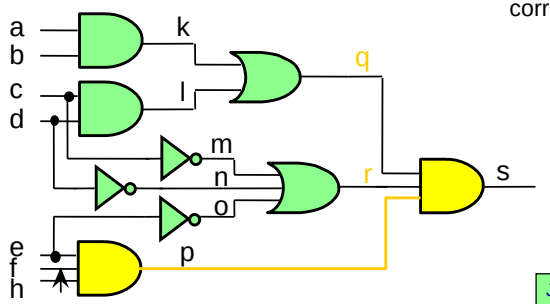


decision tree

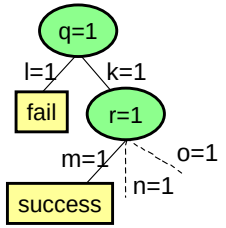
- Fault activation
  - G1=0 → { a=1, b=1, c=1 } → { G3=0 }
- Fault propagation: through G5 or G6
- Decision through G5:
  - G2=1 → { d=0, a=0 } → inconsistency at a → backtrack !!
- Decision through G6:
  - → G4=1 → e=0 → done !! The resulting test is (111x0)

D-frontiers: are the gates whose output value is x, while one or more Inputs are D or D'. For example, initially, the D-frontier is { G5, G6 }.

Structural Test Generation
D-Algorithm: Line Justification



corresponding decision tree



J-frontier: is the set of gates whose output value is known (i.e., 0 or 1), but is not implied by its input values.
Ex: initially, J-frontier is {q=1, r=1}

- FA → set h to 0
- FP → e=1, f=1 (→o=0) ; FP → q=1, r=1
- To justify q=1 → l=1 or k=1
- Decision: l =1 → c=1, d=1 → m=0, n=0 → r=0 → inconsistency at r → backtrack !
- Decision: k=1 → a=1, b=1
- To justify r=1 → m=1 or n=1 (→c=0 or d=0) → Done ! (J-frontier is ϕ)

Test Generation

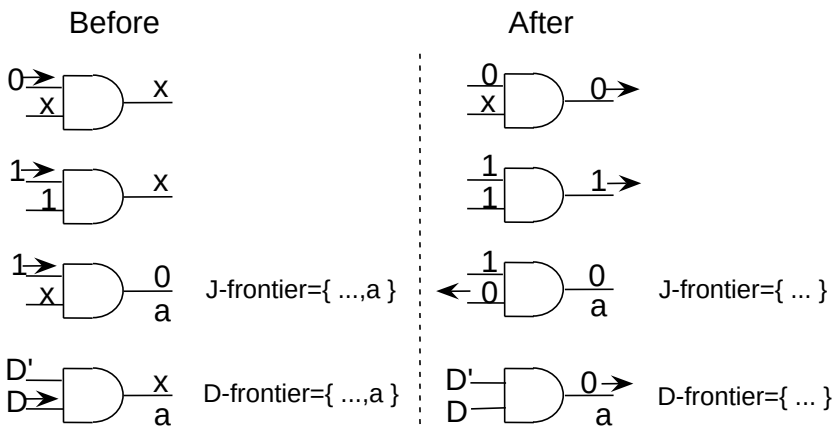
- A branch-and-bound search
- Every decision point is a branching point
- If a set of decisions lead to a conflict, a backtrack is taken to explore other decisions
- A test is found when
  - (1) fault effect is propagated to a PO, and
  - (2) all internal lines are justified
- No test is found after all possible decisions are tried → Then, target fault is undetectable
- Since the search is exhaustive, it will find a test if one exists

For a combinational circuit, an undetectable fault is also a redundant fault
→ Can be used to simplify circuit.

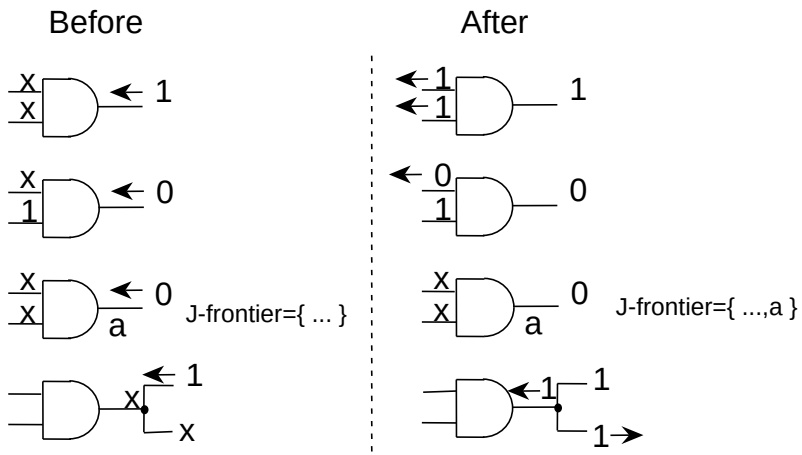
Implication

- Implication
  - Compute the values that can be uniquely determined
    - Local implication: propagation of values from one line to its immediate successors or predecessors
    - Global implication: the propagation involving a larger area of the circuit and re-convergent fanout
- Maximum implication principle
  - Perform as many implications as possible
  - It helps to either reduce the number of problems that need decisions or to reach an inconsistency sooner

Forward Implication

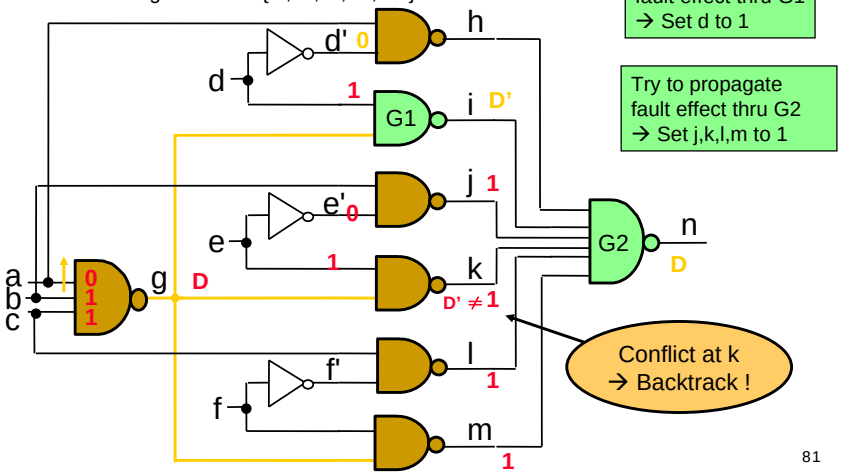


Backward Implication



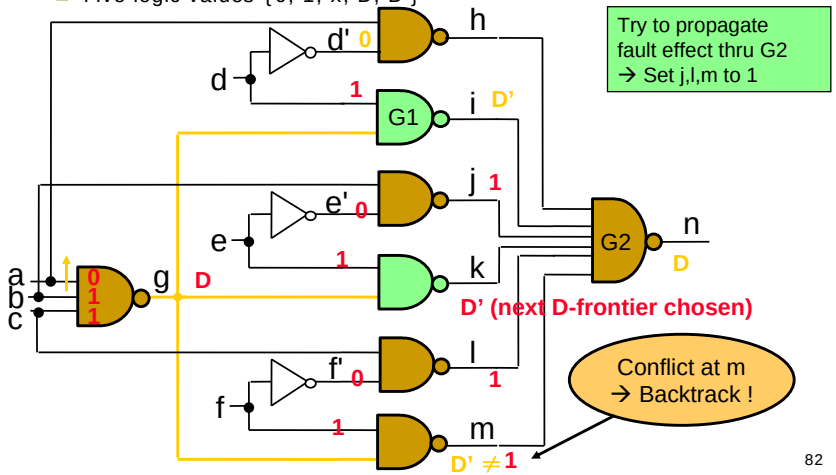
D-Algorithm (1/4)

- Example
  - Five logic values {0, 1, x, D, D'}



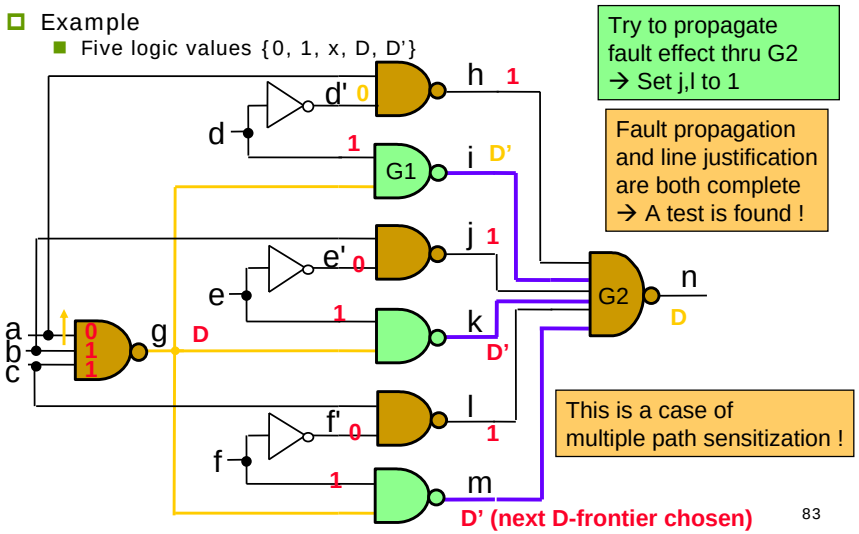
D-Algorithm (2/4)

- Example
  - Five logic values {0, 1, x, D, D'}



D-Algorithm (3/4)

- Example
  - Five logic values {0, 1, x, D, D'}



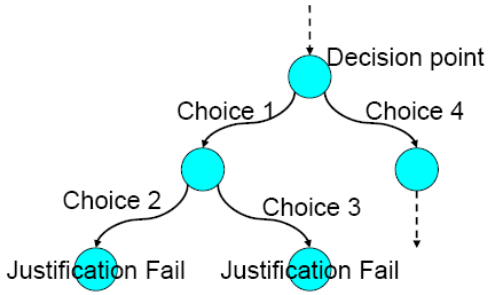
D-Algorithm (4/4)

Decision	Implication	Comments
	a=0 h=1 b=1 c=1 g=D	Active the fault Unique D-drive
d=1	i=D' d'=0	Propagate via i
j=1 k=1 l=1 m=1	n=D e'=0 e=1 k=D'	Propagate via n Contradiction

e=1	k=D' e'=0 j=1	Propagate via k
l=1 m=1	n=D f'=0 f=1 m=D'	Propagate via n Contradiction
f=1	m=D' f'=0 l=1 n=D	Propagate via m

Decision Tree on D-Frontier

- The decision tree
  - Node → D-frontier
  - Branch → decision taken
  - A Depth-First-Search (DFS) strategy is often used

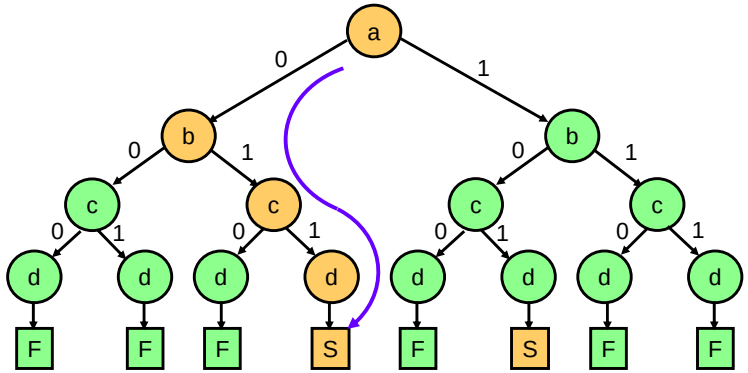


PODEM Algorithm

- PODEM: Path-Oriented DEcision Making
- Fault Activation (FA) and Propagation (FP)
  - lead to sets of Line Justification (LJ) problems. The LJ problems can be solved via value assignments.
- In D-algorithm
  - TG is done through indirect signal assignment for FA, FP, and LJ, that eventually maps into assignments at PI's
  - The decision points are at internal lines
  - The worst-case number of backtracks is exponential in terms of the number of decision points (e.g., at least 2^k for k decision nodes)
- In PODEM
  - The test generation is done through a sequence of direct assignments at PI's
  - Decision points are at PIs, thus the number of backtracking might be fewer

PODEM Algorithm
Search Space of PODEM

- Complete search space
  - A binary tree with 2^n leaf nodes, where n is the number of PIs
- Fast test generation
  - Need to find a path leading to a SUCCESS terminal quickly



PODEM Algorithm
Objective and Backtrace

- PODEM
  - Also aims at establishing a sensitization path based on fault activation and propagation like D-algorithm
  - Instead of justifying the signal values required for sensitizing the selected path, objectives are setup to guide the decision process at PIs
- Objective
  - is a signal-value pair (w, v_w)
- Backtrace
  - Backtrace maps a desired objective into a PI assignment that is likely to contribute to the achievement of the objective
  - Is a process that traverses the circuit back from the objective signal to PIs
  - The result is a PI signal-value pair (x, v_x)
  - No signal value is actually assigned during backtrace (toward PI) !

PODEM Algorithm
Objective

- Objective routine involves
  - selection of a D-frontier, G
  - selection of an unspecified input gate of G

```
Objective() {
   /* The target fault is w s-a-v */
   /* Let variable obj be a signal-value pair */
   if (the value of w is x) obj = ( w, v' );      <- fault activation
   else {
        select a gate (G) from the D-frontier;    <- fault propagation
        select an input (j) of G with value x;
        c = controlling value of G;
        obj = (j, c');
   }
   return (obj);
}
```

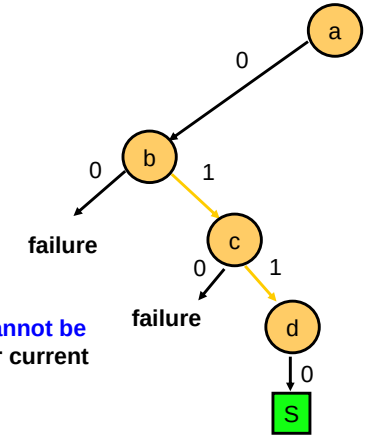

PODEM Algorithm
Backtrace

- Backtrace routine involves
  - finding an all-x path from objective site to a PI, i.e., every signal in this path has value x

```
Backtrace(w, v_w) {
   /* Maps objective into a PI assignment */
   G = w;  /* objective node */
   v = v_w; /* objective value */
   while (G is a gate output) { /* not reached PI yet */
        inv = inversion of G;
        select an input (j) of G with value x;
        G = j;          /* new objective node */
        v = v⊕inv;  /* new objective value */
   }
   /* G is a PI */    return (G, v);
}
```


PODEM Algorithm
PI Assignment

PIs: { a, b, c, d }
Current Assignments: { a=0 }
Decision: b=0 → objective fails
Reverse decision: b=1
Decision: c=0 → objective fails
Reverse decision: c=1
Decision: d=0



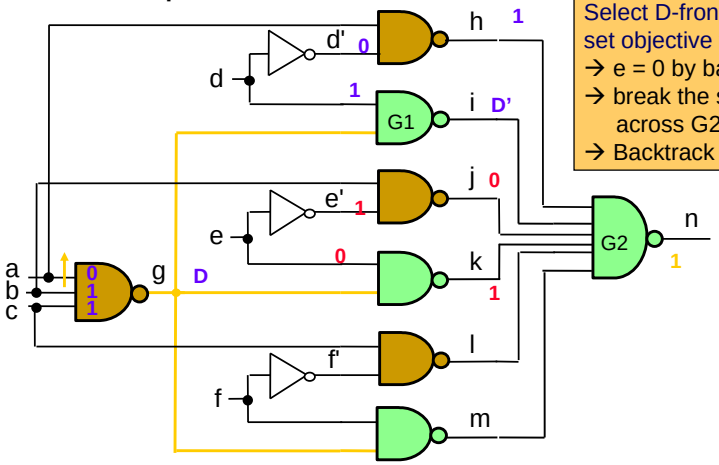
Failure means fault effect cannot be propagated to any PO under current PI assignments

PODEM Algorithm

```
PODEM () /* using depth-first-search */
begin
    If(error at PO) return(SUCCESS);
    If(test not possible)      return(FAILURE);
    (k, v_k) = Objective();            /* choose a line to be justified */
    (j, v_j) = Backtrace(k, v_k);      /* choose the PI to be assigned */
    Imply (j, v_j);                    /* make a decision */
    If ( PODEM()==SUCCESS )            return (SUCCESS);
    Imply (j, v_j');                   /* reverse decision */
    If ( PODEM()==SUCCESS )            return(SUCCESS);
    Imply (j, x);
    Return (FAILURE);
end
```

PODEM Algorithm (1/4)

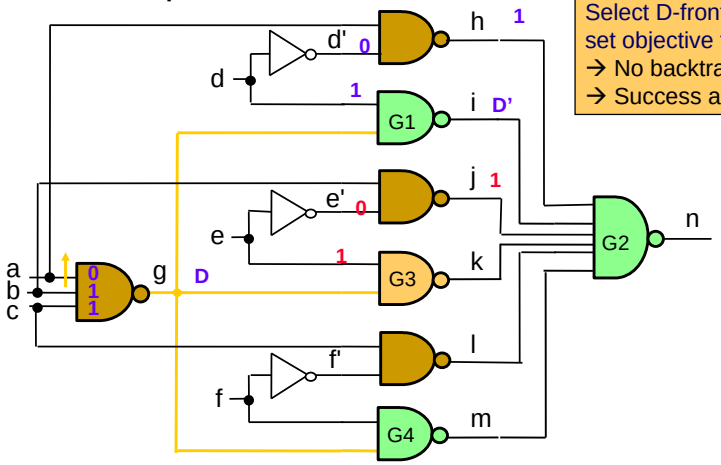
- Example



Select D-frontier G2 and set objective to (k,1)
→ e = 0 by backtrace
→ break the sensitization across G2 (j=0)
→ Backtrack !

PODEM Algorithm (2/4)

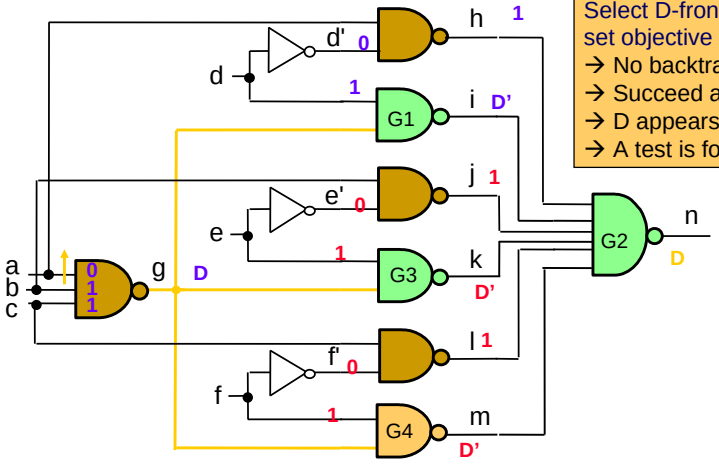
- Example



Select D-frontier G3 and set objective to (e,1)
→ No backtrace is needed
→ Success at G3

PODEM Algorithm (3/4)

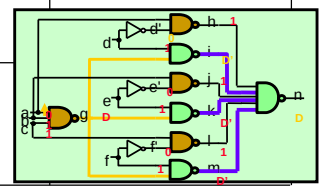
- Example



Select D-frontier G4 and set objective to (f,1)
→ No backtrace is needed
→ Succeed at G4 and G2
→ D appears at one PO
→ A test is found !!

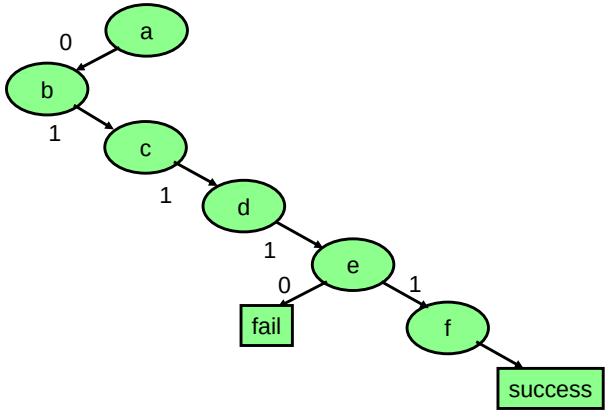
PODEM Algorithm (4/4)

Objective	PI assignment	Implications	D-frontier	Comments
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=D'	k,m,n	
k=1	e=0	e'=1 j=0 k=1 n=1	m	Assignments need to be reversed during backtracking no solutions! → backtrack
	e=1	e'=0 j=1 k=D'	m,n	flip PI assignment
l=1	f=1	f'=0 l=1 m=D' n=D		



PODEM Algorithm Decision Tree

- Decision node:
  PI selected through backtrace for value assignment
- Branch:
  value assignment to the selected PI



Termination Conditions

- D-algorithm
  - Success:
    (1) Fault effect at an output (D-frontier may not be empty)
    (2) J-frontier is empty
  - Failure:
    (1) D-frontier is empty (all possible paths are false)
    (2) J-frontier is not empty
- PODEM
  - Success:
    - Fault effect seen at an output
  - Failure:
    - Every PI assignment leads to failure, in which D-frontier is empty while fault has been activated

PODEM Overview

- PODEM
  - examines all possible input patterns implicitly but exhaustively (branch-and-bound) for finding a test
  - complete like D-algorithm (i.e., will find a test if exists)
- Other key features
  - No J-frontier, since there are no values that require justification
  - No consistency check, as conflicts can never occur
  - No backward implication, because values are propagated only forward
  - Backtracking is implicitly done by simulation rather than by an explicit and time-consuming save/restore process
  - Experiments show that PODEM is generally faster than D-algorithm

Outline

- Fault Modeling
- Fault Simulation
- Automatic Test Pattern Generation
- Design for Testability

Why DFT ?

- Direct testing is way too difficult !
  - Large number of FFs
  - Embedded memory blocks
  - Embedded analog blocks

Design for Testability

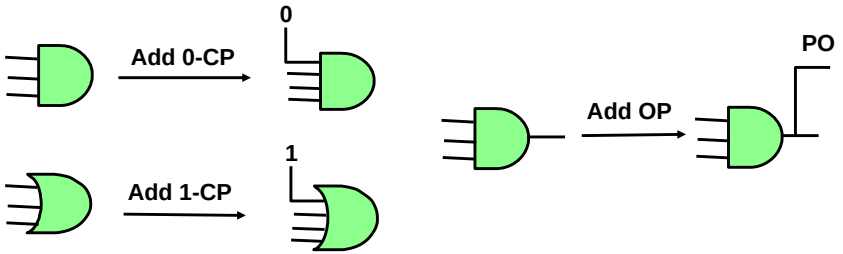
- Definition
  - Design for testability (DFT) refers to those design techniques that make test generation and testing cost-effective
- DFT methods
  - Ad-hoc methods, full and partial scan, built-in self-test (BIST), boundary scan
- Cost of DFT
  - Pin count, area, performance, design-time, test-time, etc.

Important Factors

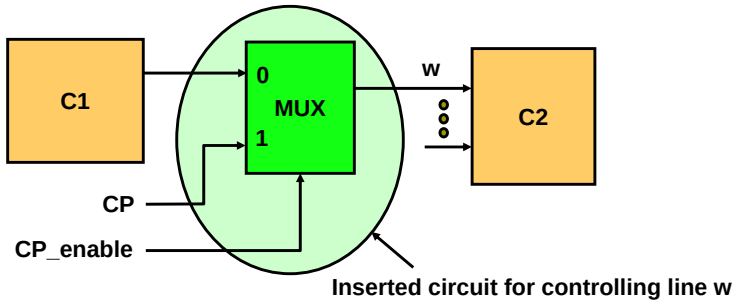
- Controllability
  - Measure the ease of controlling a line
- Observability
  - Measure the ease of observing a line at PO
- DFT deals with ways of improving
  - Controllability and observability

Test Point Insertion

- Employ test points to enhance **controllability** and **observability**
  - CP: Control Points
    - Primary inputs used to enhance controllability
  - OP: Observability Points
    - Primary outputs used to enhance observability



Control Point Insertion



- Normal operation:
  When CP_enable = 0
- Inject 0:
  Set CP_enable = 1 and CP = 0
- Inject 1:
  Set CP_enable = 1 and CP = 1

Control Point Selection

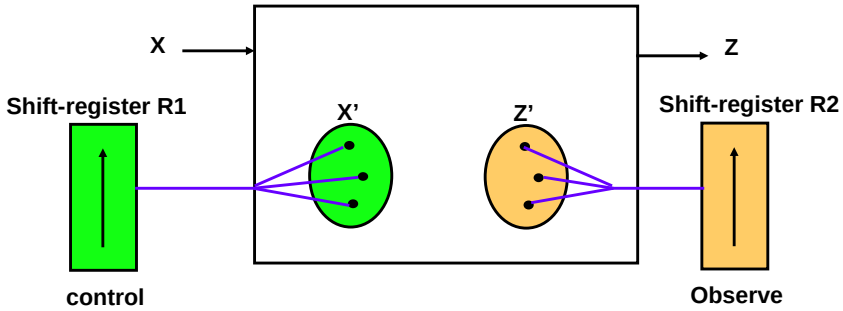
- Goal
  - Controllability of the fanout-cone of the added point is improved
- Common selections
  - Control, address, and data buses
  - Enable/hold inputs
  - Enable and read/write inputs to memory
  - Clock and preset/clear signals of flip-flops
  - Data select inputs to multiplexers and demultiplexers

Observation Point Selection

- Goal
  - Observability of the transitive fanins of the added point is improved
- Common choice
  - Stem lines with more fanouts
  - Global feedback paths
  - Redundant signal lines
  - Output of logic devices having many inputs
    - MUX, XOR trees
  - Output from state devices
  - Address, control and data buses

Problems with Test Point Insertion

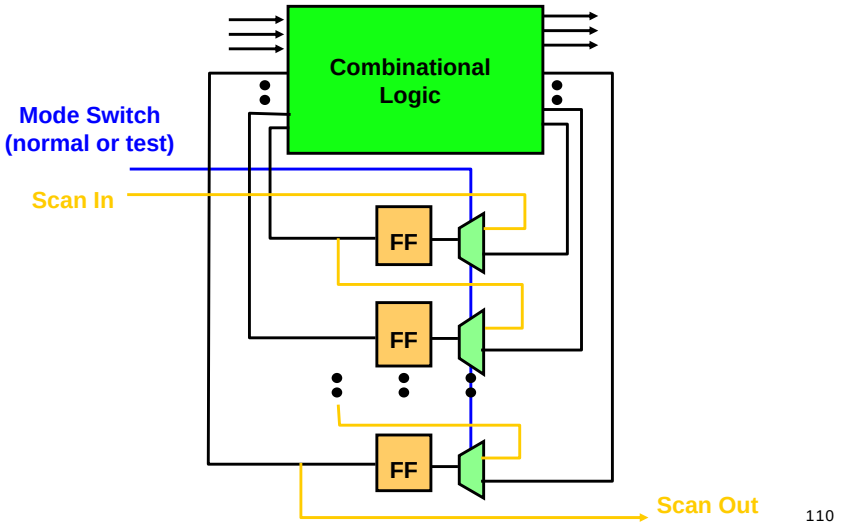
- Large number of I/O pins
  - Can be resolved by adding MUXs to reduce the number of I/O pins, or by adding shift-registers to impose CP values



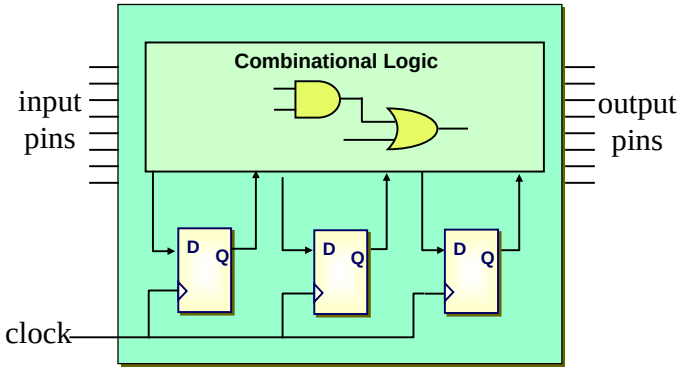
What Is Scan ?

- Objective
  - To provide controllability and observability at internal state variables for testing
- Method
  - Add test mode control signal(s) to circuit
  - Connect flip-flops to form shift registers in test mode
  - Make inputs/outputs of the flip-flops in the shift register controllable and observable
- Types
  - Internal scan
    - Full scan, partial scan, random access
  - Boundary scan

Scan Concept

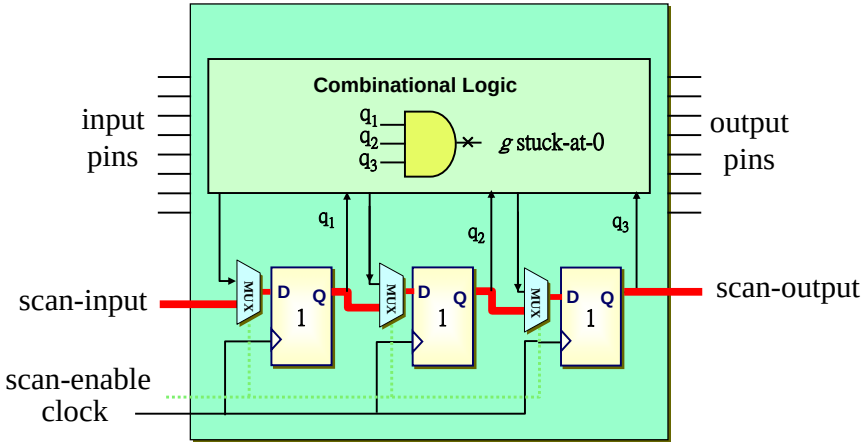


Logic Design before Scan Insertion



Sequential ATPG is extremely difficult:
due to the lack of controllability and observability at flip-flops.

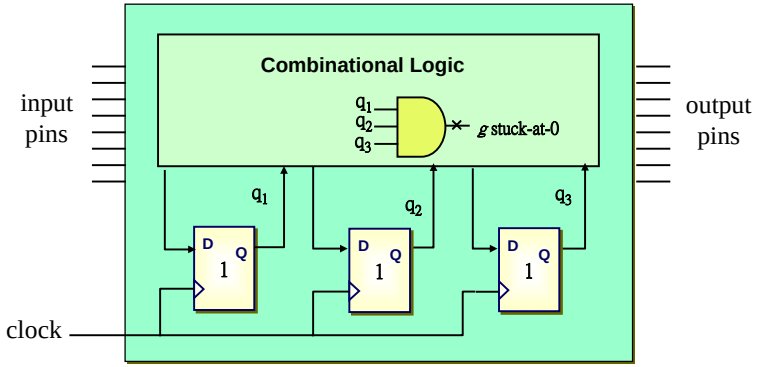
Logic Design after Scan Insertion



Scan Chain provides an easy access to flip-flops
⟶ Pattern generation is much easier !!

Scan Insertion

- Example
  - 3-stage counter



It takes 8 clock cycles to set the flip-flops to be (1, 1, 1), for detecting the target fault *g* stuck-at-0 fault (2^{20} cycles for a 20-stage counter !)

Overhead of Scan Design

- Case study
  - #CMOS gates = 2000
  - Fraction of flip-flops = 0.478
  - Fraction of normal routing = 0.471

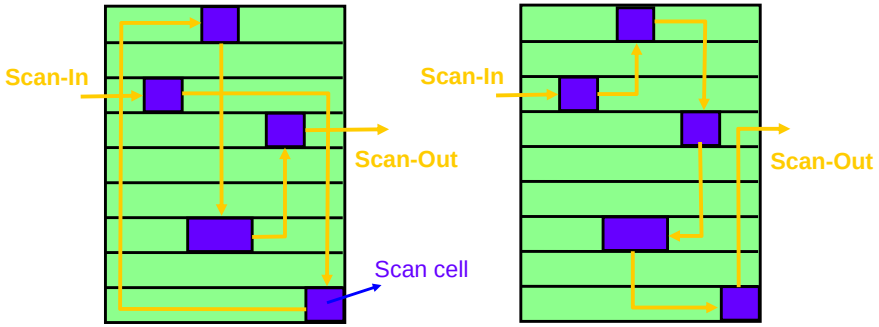
Scan implementation	Predicted overhead	Actual area overhead	Normalized operating frequency
None	0	0	1.0
Hierarchical	14.05%	16.93%	0.87
Optimized	14.05%	11.9%	0.91

Full Scan Problems

- Problems
  - Area overhead
  - Possible performance degradation
  - High test application time
  - Power dissipation
- Features of commercial tools
  - Scan-rule violation check (e.g., DFT rule check)
  - Scan insertion (convert a FF to its scan version)
  - ATPG (both combinational and sequential)
  - Scan chain reordering after layout

Scan-Chain Reordering

- Scan-chain order is often decided at gate-level without knowing the cell placement
- Scan-chain consumes a lot of routing resources, and could be minimized by re-ordering the flip-flops in the chain after layout is done



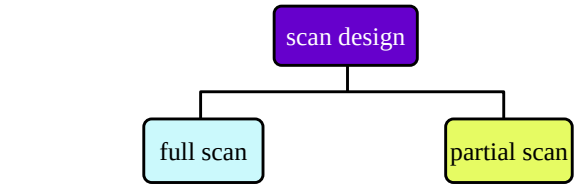
Layout of a cell-based design

A better scan-chain order

Partial Scan

- Basic idea
  - Select a subset of flip-flops for scan
  - Lower overhead (area and speed)
  - Relaxed design rules
- Cycle-breaking technique
  - Cheng & Agrawal, IEEE Trans. On Computers, April 1990
  - Select scan flip-flops to simplify sequential ATPG
  - Overhead is about 25% off than full scan
- Timing-driven partial scan
  - Jou & Cheng, ICCAD, Nov. 1991
  - Allow optimization of area, timing, and testability simultaneously

Full Scan vs. Partial Scan

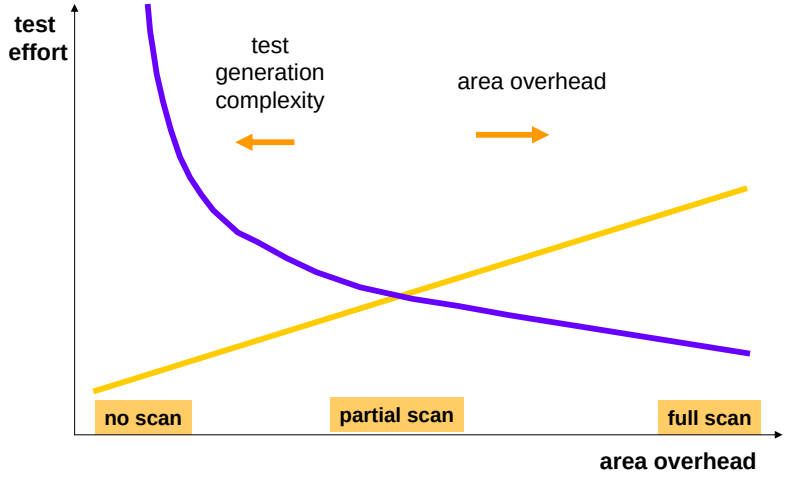


every flip-flop is a scan-FF

NOT every flip-flop is a scan-FF

scan time	longer	shorter
hardware overhead	more	less
fault coverage	~100%	unpredictable
ease-of-use	easier	harder

Area Overhead vs. Test Effort



Conclusions

- Testing
  - Conducted after manufacturing
  - Must be considered during the design process
- Major fault models
  - Stuck-at, bridging, stuck-open, delay fault, ...
- Major tools needed
  - Design-for-Testability
    - By scan chain insertion or built-in self-test
  - Fault simulation
  - ATPG
- Other Applications in CAD
  - ATPG is a way of Boolean reasoning and is applicable to may logic-domain CAD problems