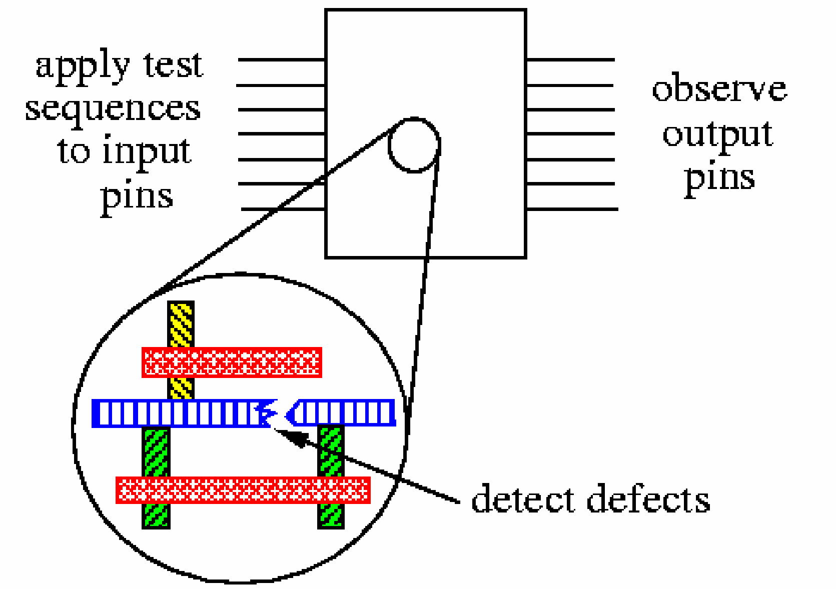
Unit 6: Testing

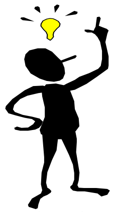
- Course contents
  - Fault Modeling
  - Fault Simulation
  - Test Generation
  - Design For Testability
- Reading
  - Supplementary readings



Outline

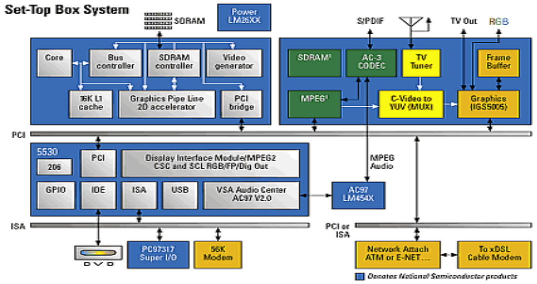
- Introduction
- Fault Modeling
- Fault Simulation
- Test Generation
- Design For Testability

Chip Design & Manufacturing Flow



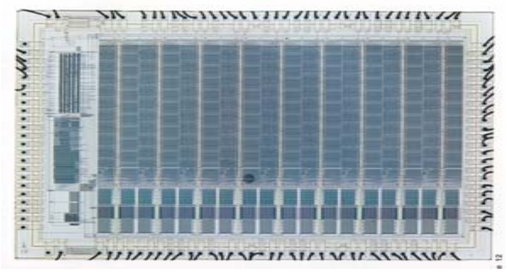
Idea

Architecture Design



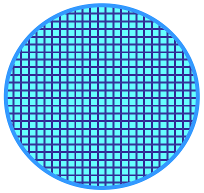
Block diagram

Circuit & Layout Design



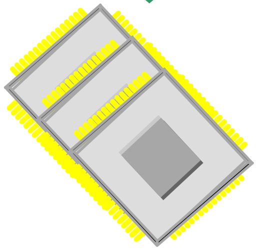
Layout

IC Fabrication



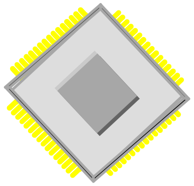
Wafer (hundreds of dies)

Sawing & Packaging

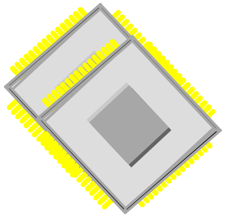


Final chips

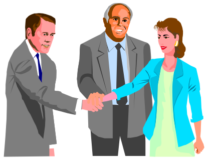
Final Testing



Bad chips



Good chips



customers

Design Verification, Testing and Diagnosis

- Design Verification:
  - Ascertain the design perform its specified behavior
- Testing:
  - Exercise the system and analyze the response to ascertain whether it behaves correctly after manufacturing
- Diagnosis:
  - To locate the cause(s) of misbehavior after the incorrect behavior is detected

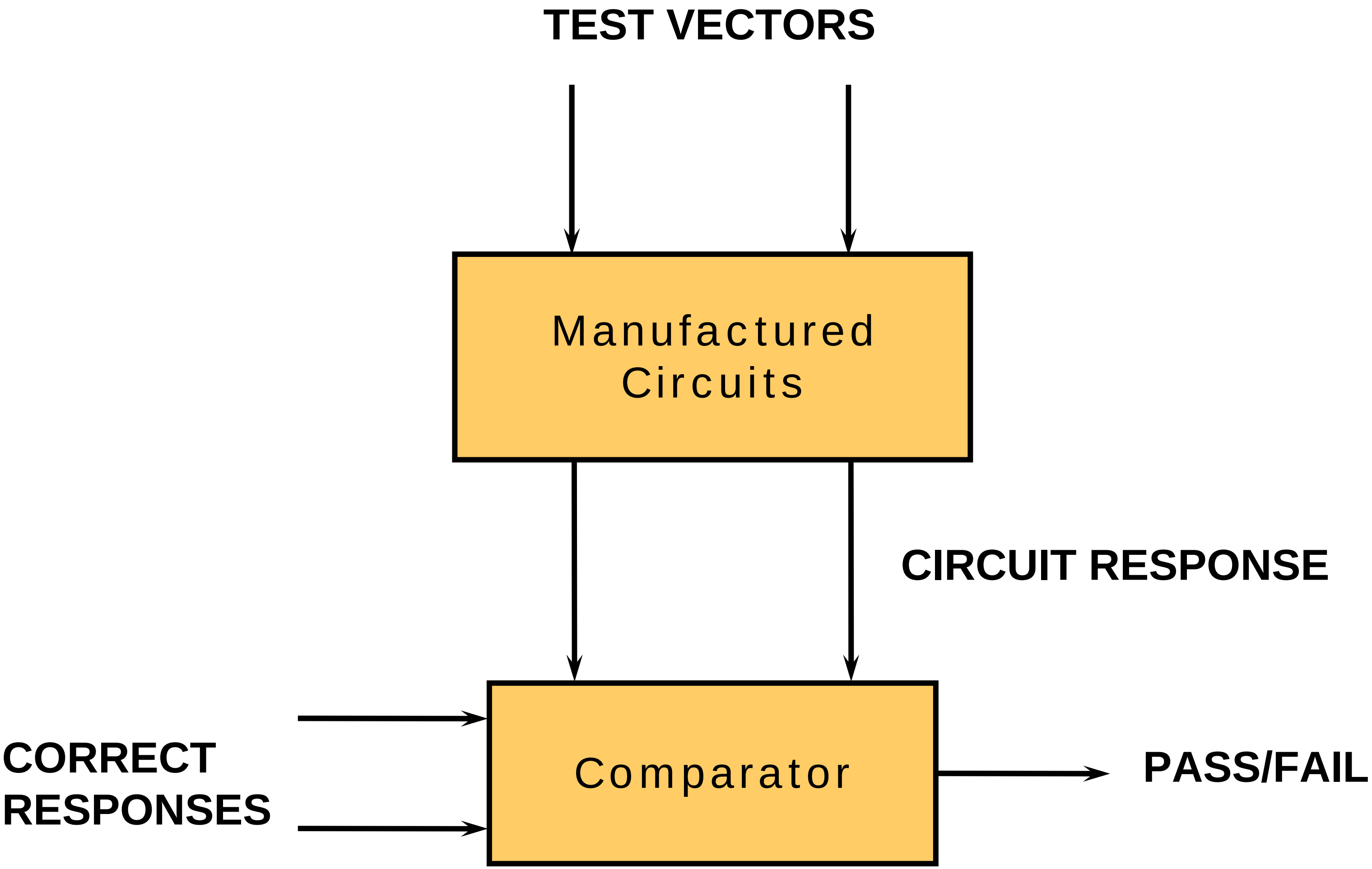
Manufacturing Defects

- Processing Faults
  - missing contact windows
  - parasitic transistors
  - oxide breakdown
- Material Defects
  - bulk defects (cracks, crystal imperfections)
  - surface impurities
- Time-Dependent Failures
  - dielectric breakdown
  - electro-migration
- Packaging Failures
  - contact degradation
  - seal leaks

Faults, Errors and Failures

- Fault:
  - A physical defect within a circuit or a system
  - May or may not cause a system failure
- Error:
  - Manifestation of a fault that results in incorrect circuit (system) outputs or states
  - Caused by faults
- Failure:
  - Deviation of a circuit or system from its specified behavior
  - Fails to do what it should do
  - Caused by an error
- Fault ---> Error ---> Failure

Scenario of Manufacturing Test



TEST VECTORS

Manufactured Circuits

CIRCUIT RESPONSE

CORRECT RESPONSES

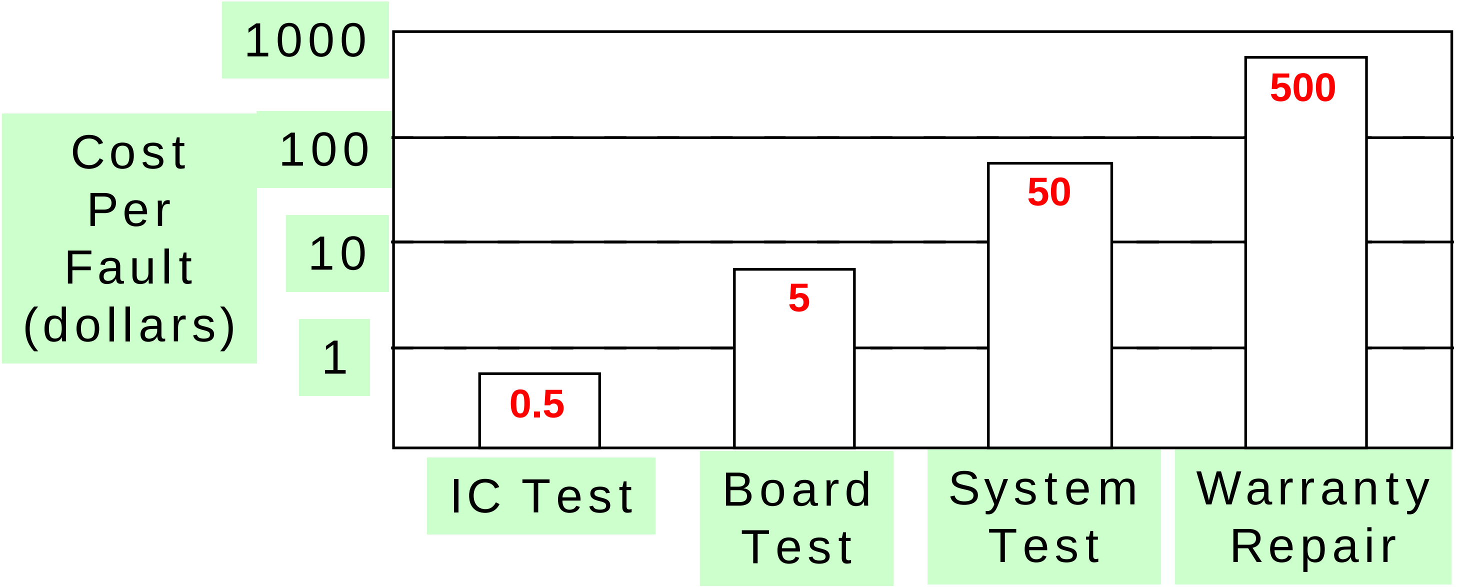
Comparator

PASS/FAIL

Tester: Advantest T6682

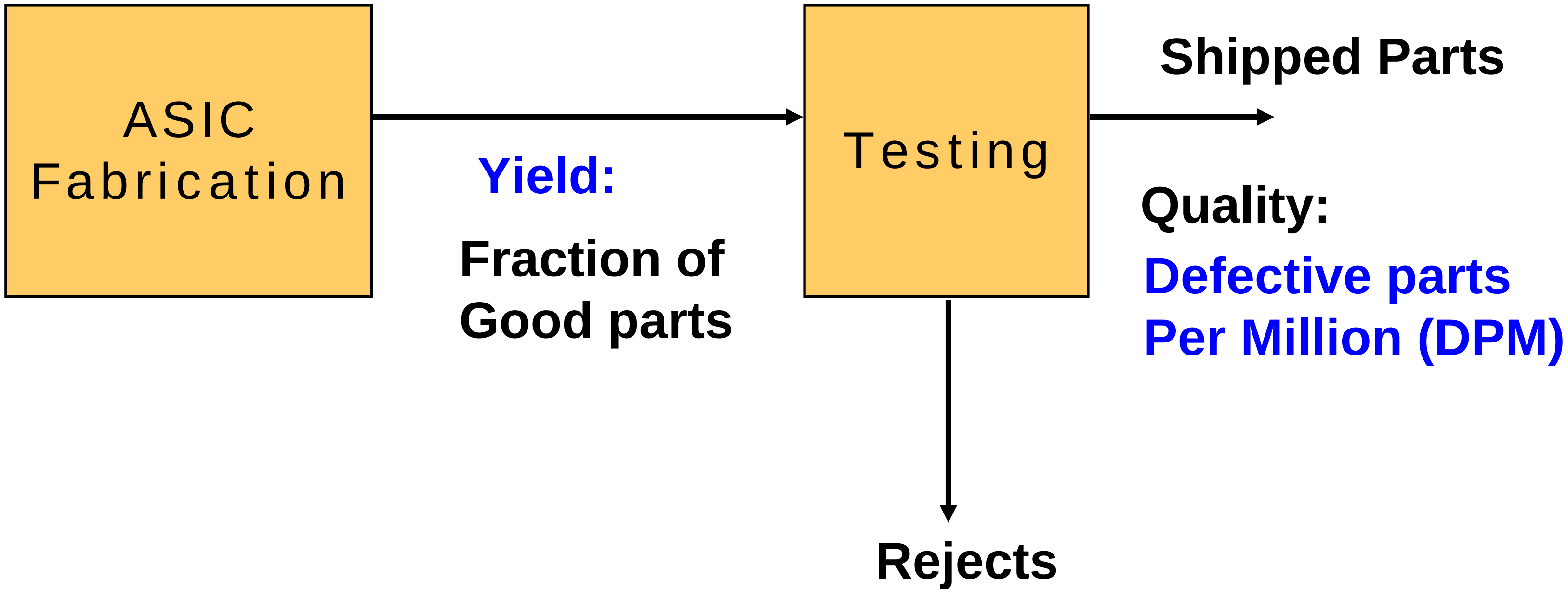
Purpose of Testing

- Verify Manufacturing of Circuit
  - Improve System Reliability
  - Diminish System Cost
- Cost of repair
  - goes up by an order of magnitude each step away from the fab. line



B. Davis, "The Economics of Automatic Testing" McGraw-Hill 1982

Testing and Quality



Yield:

Fraction of Good parts

Quality:

Defective parts Per Million (DPM)

Quality of shipped part is a function of yield Y and the test (fault) coverage T.

Fault Coverage

- Fault Coverage T
  - Is the measure of the ability of a set of tests to detect a given class of faults that may occur on the device under test (DUT)

$$T = \frac{\text{No. of detected faults}}{\text{No. of all possible faults}}$$

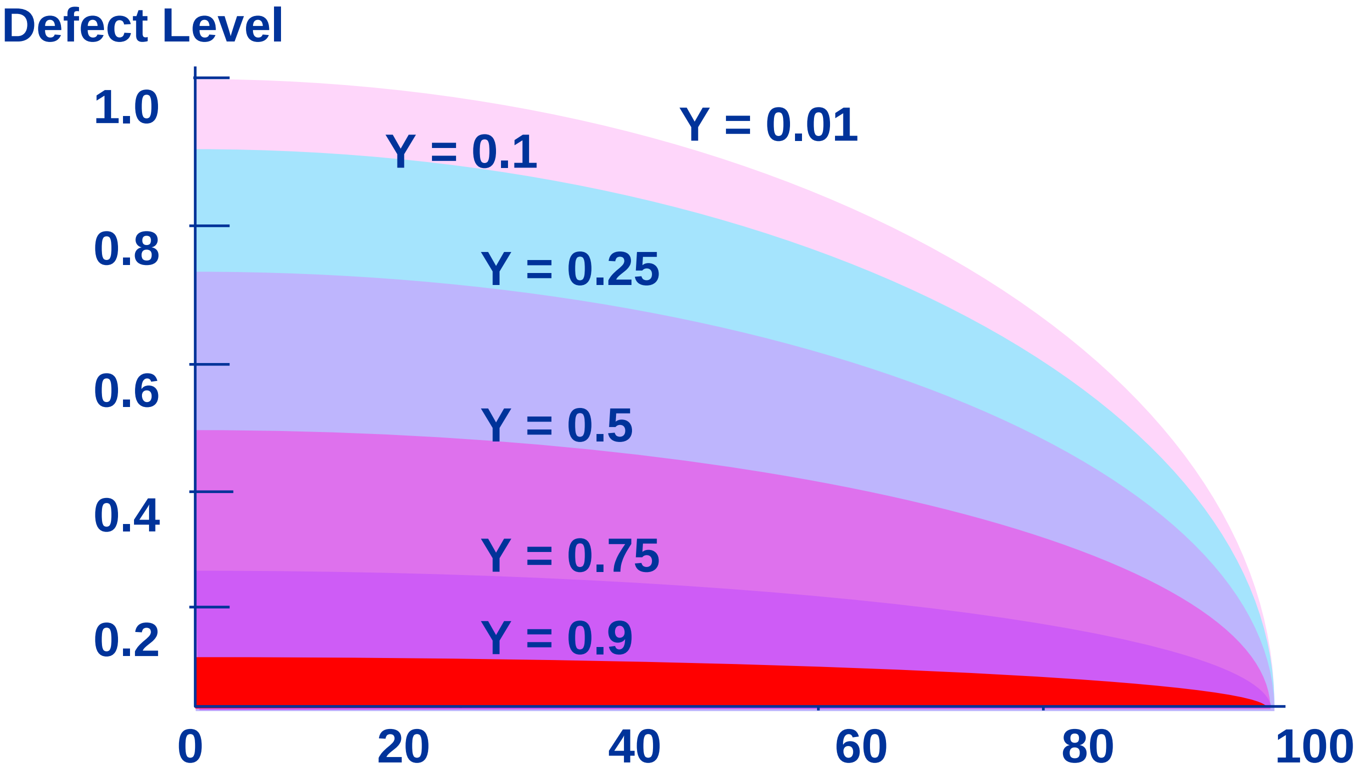
Defect Level

- Defect Level
  - Is the fraction of the shipped parts that are defective

$$\boxed{DL = 1 - Y^{(1-T)}}$$

Y: yield
T: fault coverage

Defect Level v.s. Fault Coverage



(Williams IBM 1980)

Fault Coverage (%)

High fault coverage ⟶ Low defect level

DPM v.s. Yield and Coverage

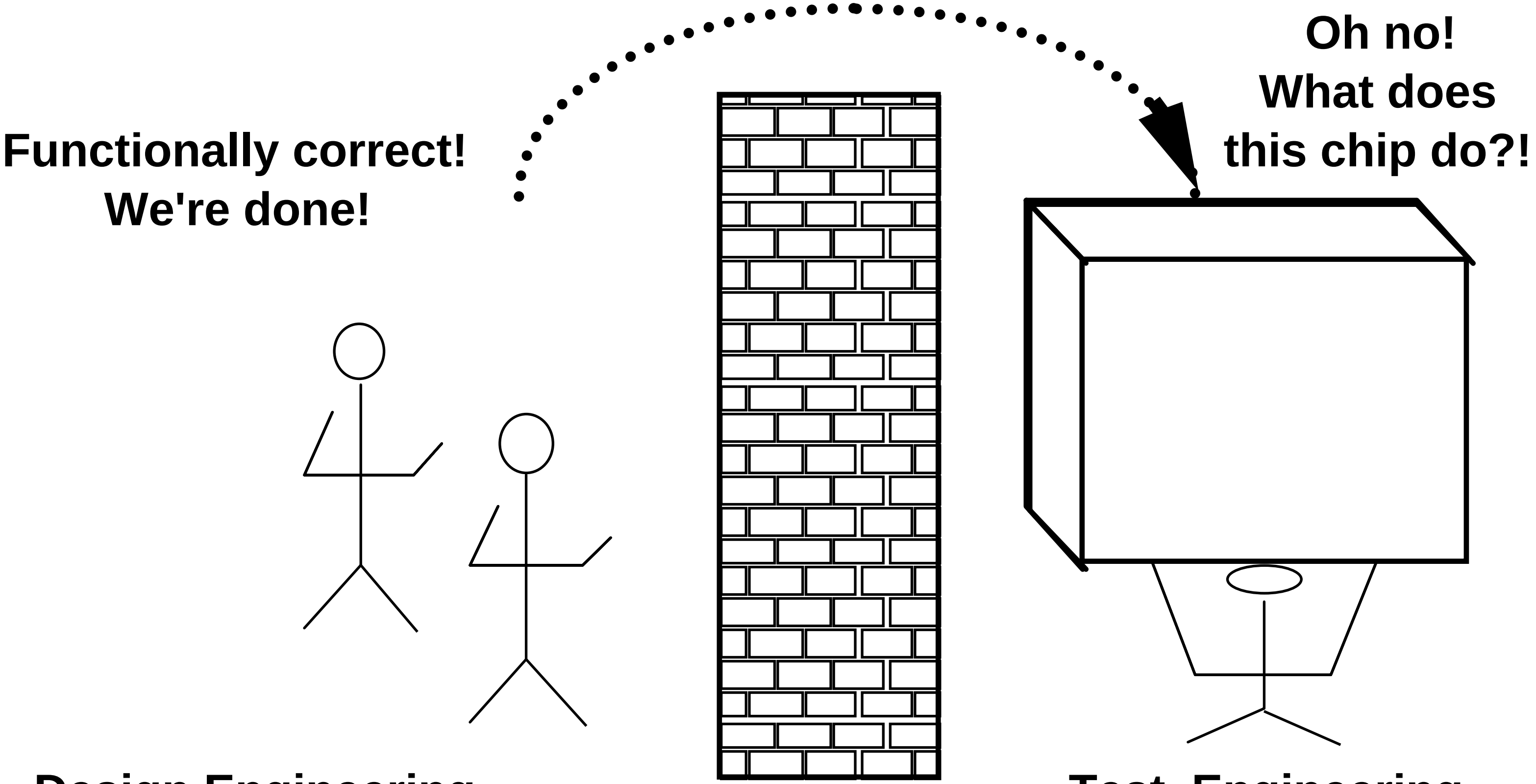
Yield	Fault Coverage	DPM
50%	90%	67,000
75%	90%	28,000
90%	90%	10,000
95%	90%	5,000
99%	90%	1,000
90%	90%	10,000
90%	95%	5,000
90%	99%	1,000
90%	99.9%	100

Why Testing Is Difficult ?

- Test application time can be exploded for exhaustive testing of VLSI
  - For a combinational circuit with 50 inputs, we need 2^{50} = 1.126**x**10^{15} test patterns.
  - Assume one test per 10^{-7}sec, it takes 1.125**x**10^{8}sec = 3.57yrs. to test such a circuit.
  - Test generation for sequential circuits are even more difficult due to the lack of controllability and observability at flip-flops (latches)
- Functional testing
  - may NOT be able to detect the physical faults

The Infamous Design/Test Wall

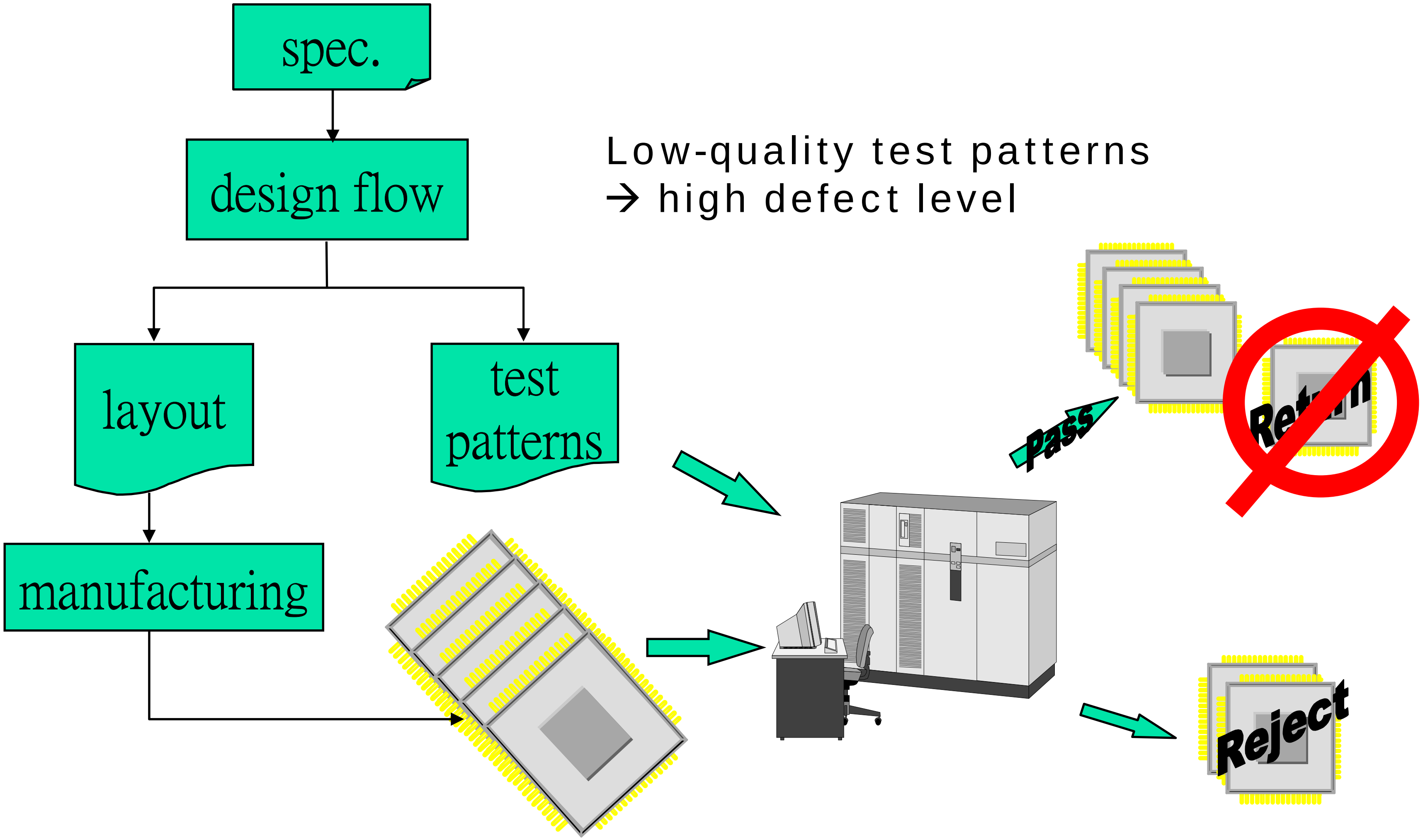
30 years of experience proves that test after design does not work!



Design Engineering

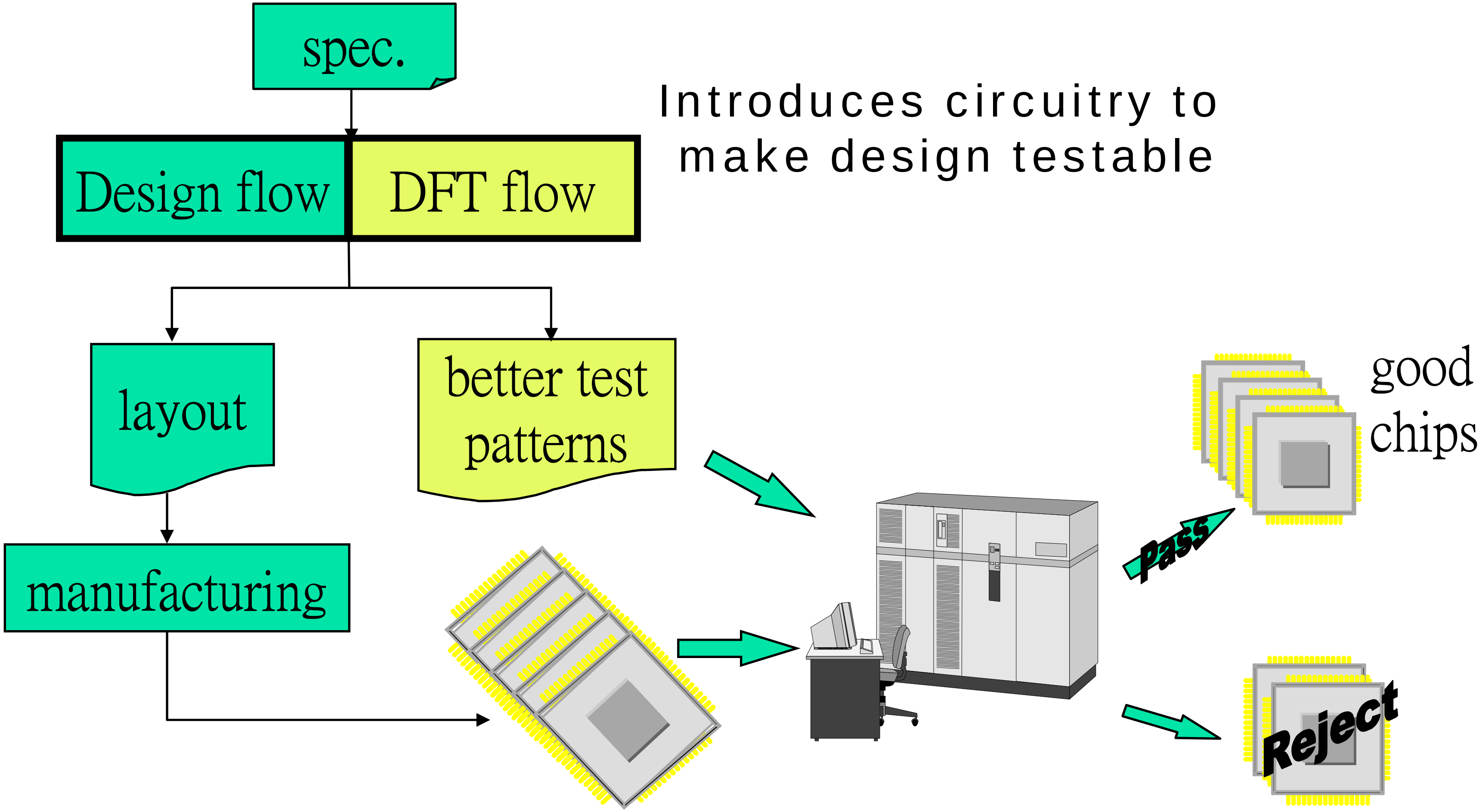
Test Engineering

Old Design & Test Flow



Low-quality test patterns
→ high defect level

New Design and Test Flow



Outline

- Introduction
- Fault Modeling
- Fault Simulation
- Test Generation
- Design For Testability

Functional v.s. Structural Testing

- I/O function tests inadequate for manufacturing
  - Functionality vs. component & interconnection testing
- Exhaustive testing is Prohibitively expensive

Why Fault Model ?

- Fault model identifies target faults
  - Model faults most likely to occur
- Fault model limits the scope of test generation
  - Create tests only for the modeled faults
- Fault model makes effectiveness measurable by experiments
  - Fault coverage can be computed for specific test patterns to reflect its effectiveness
- Fault model makes analysis possible
  - Associate specific defects with specific test patterns

Fault Modeling

- Fault Modeling
  - Model the effects of physical defects on the logic function and timing
- Physical Defects
  - Silicon Defects
  - Photolithographic Defects
  - Mask Contamination
  - Process Variation
  - Defective Oxides

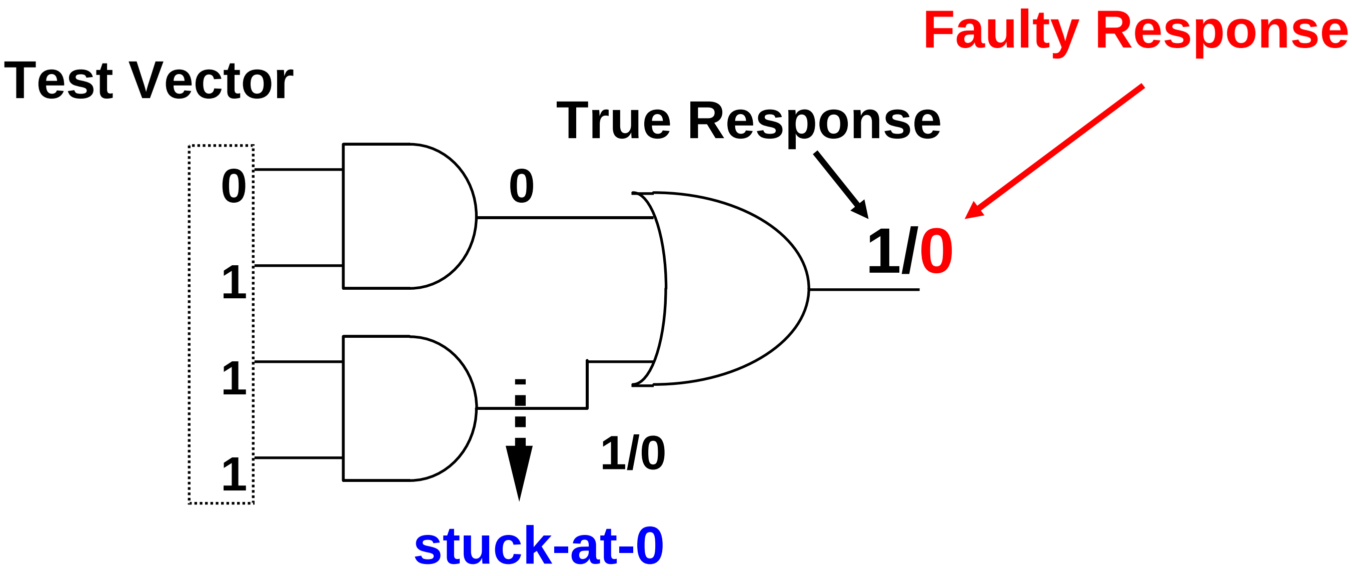
Fault Modeling (cont’d)

- Electrical Effects
  - Shorts (Bridging Faults)
  - Opens
  - Transistor Stuck-On/Open
  - Resistive Shorts/Opens
  - Change in Threshold Voltages
- **Logical Effects**
  - Logical Stuck-at 0/1
  - Slower Transition (Delay Faults)
  - AND-bridging, OR-bridging

Fault Types Commonly Used To Guide Test Generation

- Stuck-at Faults
- Bridging Faults
- Transistor Stuck-On/Open Faults
- Delay Faults
- IDDQ Faults
- State Transition Faults (for FSM)
- Memory Faults
- PLA Faults

Single Stuck-At Fault



Assumptions:

- Only One line is faulty
- Faulty line permanently set to 0 or 1
- Fault can be at an input or output of a gate

Multiple Stuck-At Faults

- Several stuck-at faults occur at the same time
  - Important in high density circuits
- For a circuit with k lines
  - there are 2k single stuck-at faults
  - there are 3^k-1 multiple stuck-at faults
    - A line could be stuck-at-0, stuck-at-1, or fault-free
    - One out of 3^k resulting circuits is fault-free

Why Single Stuck-At Fault Model

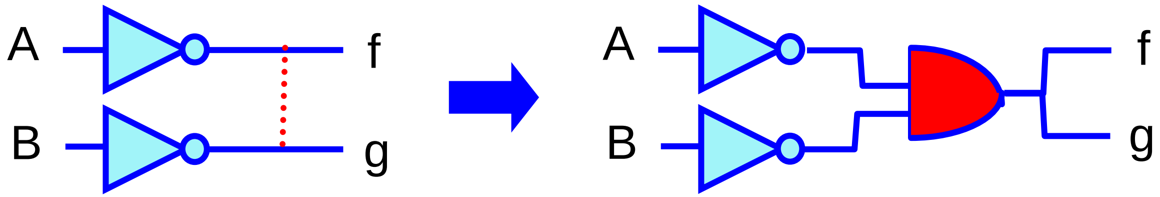
- Complexity is greatly reduced
  - Many different physical defects may be modeled by the same logical single stuck-at fault
- Stuck-at fault is technology independent
  - Can be applied to TTL, ECL, CMOS, BiCMOS etc.
- Design style independent
  - Gate array, standard cell, custom VLSI
- Detection capability of un-modeled defects
  - Empirically many defects accidentally detected by test derived based on single stuck-at fault
- Cover a large percentage of multiple stuck-at faults

Multiple Faults

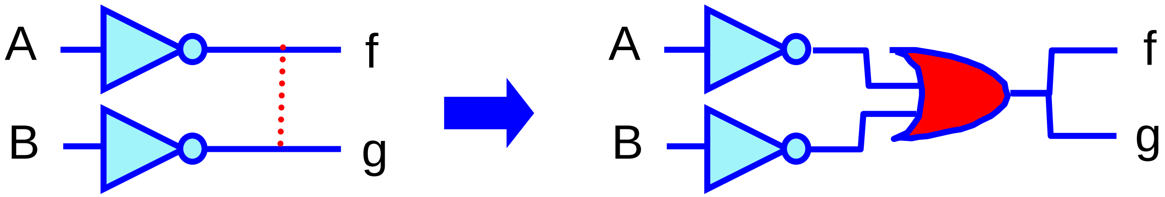
- Multiple stuck-fault coverage by single-fault tests of combinational circuit:
  - 4-bit ALU (Hughes & McCluskey, ITC-84)
    All double and most triple-faults covered.
  - Large circuits (Jacob & Biswas, ITC-87)
    Almost 100% multiple faults covered for circuits with 3 or more outputs.
- No results available for sequential circuits.

Bridging Faults

- Two or more normally distinct points (lines) are shorted together
  - Logic effect depends on technology
  - Wired-AND for TTL



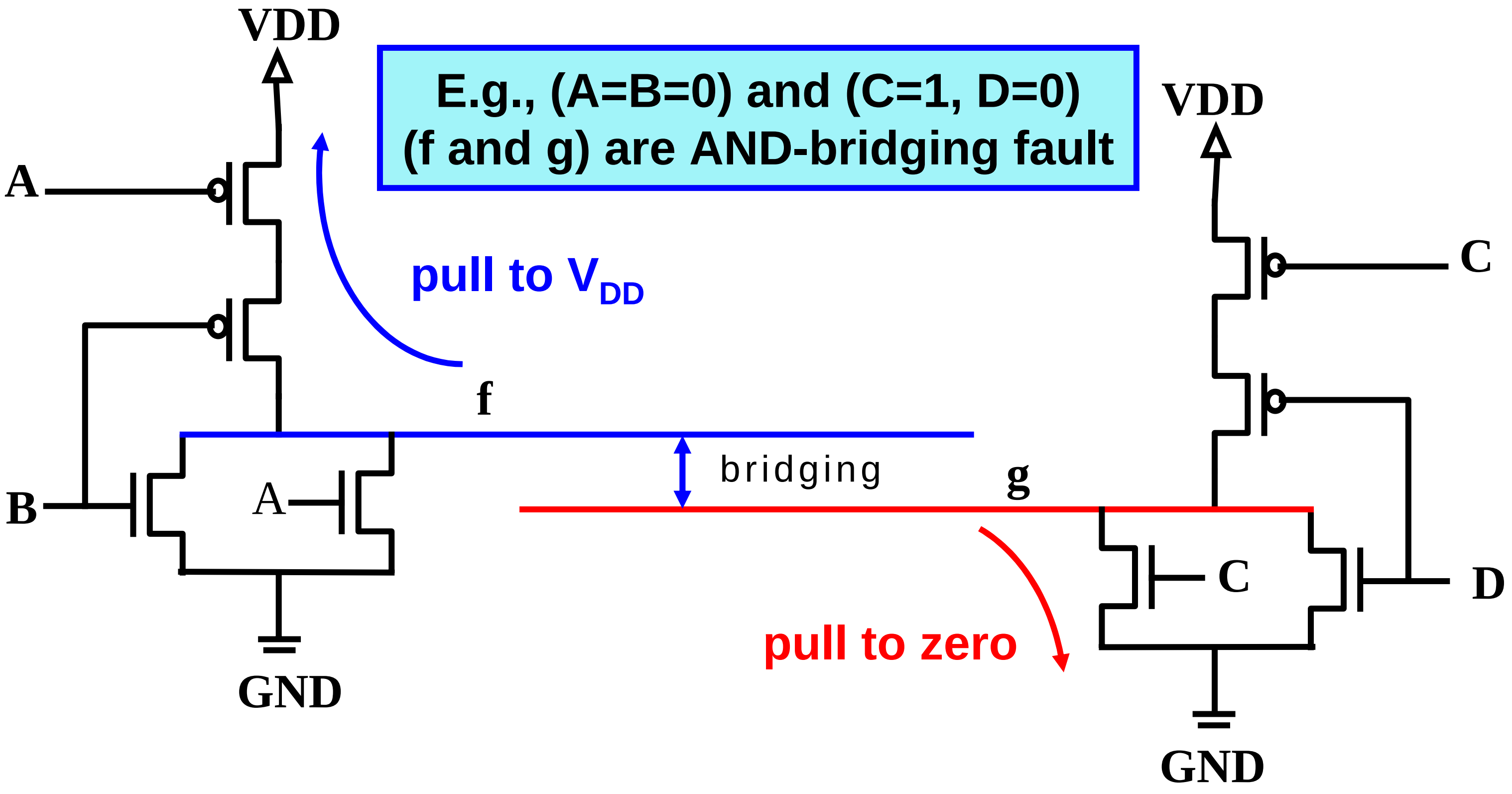
  - Wired-OR for ECL



  - CMOS ?

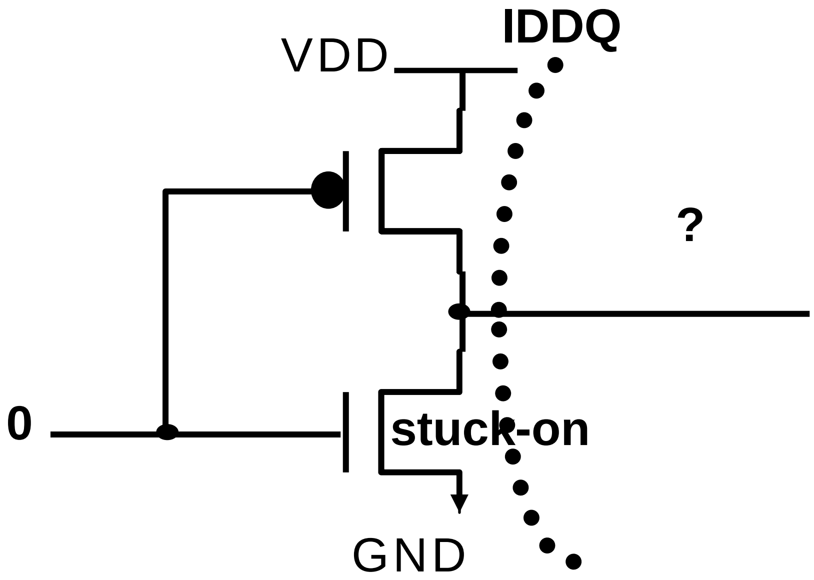
Bridging Faults For CMOS Logic

- The result
  - could be AND-bridging or OR-bridging
  - depends on the the inputs



E.g., (A=B=0) and (C=1, D=0)
(f and g) are AND-bridging fault

CMOS Transistor Stuck-On

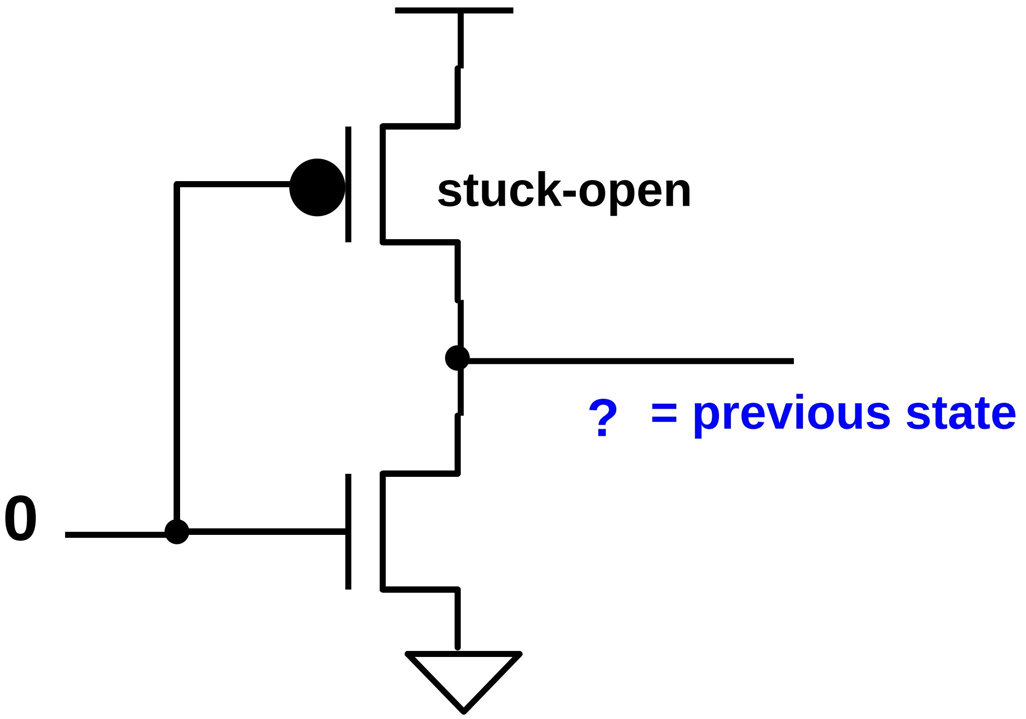


Example:
N transistor
is always ON

- Transistor Stuck-On
  - May cause ambiguous logic level
  - Depends on the relative impedances of the pull-up and pull-down networks
- When Input Is Low
  - Both P and N transistors are conducting, causing increased quiescent current, called IDDQ fault

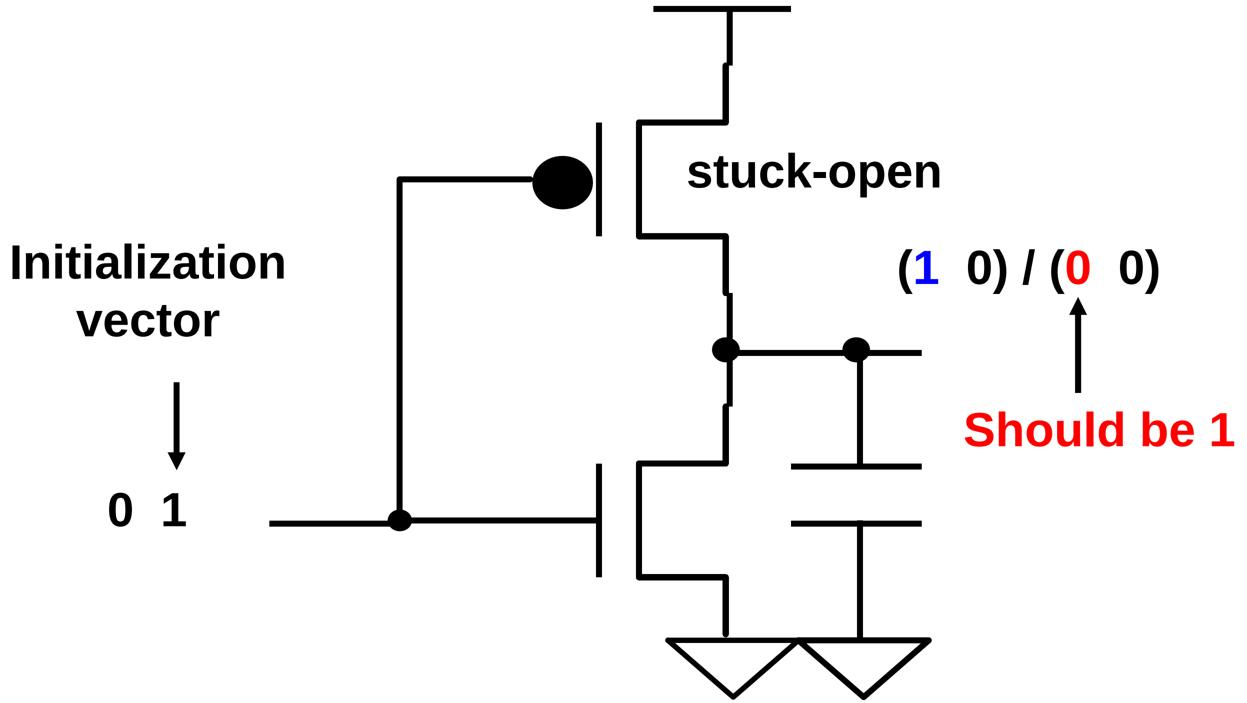
CMOS Transistor Stuck-Open (I)

- Transistor stuck-open
  - May cause the output to be floating
  - The faulty becomes exhibits sequential behavior

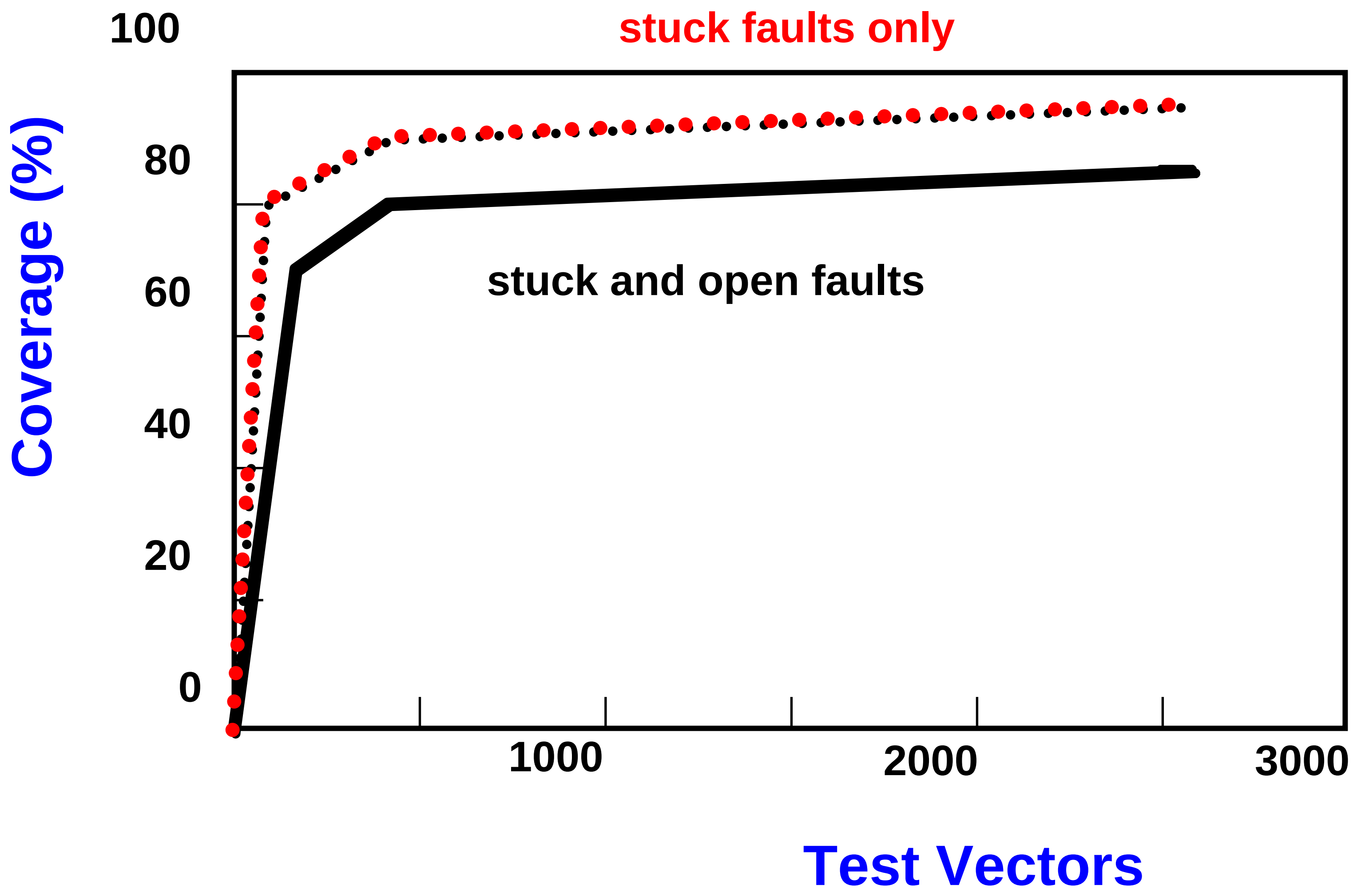


CMOS Transistor Stuck-Open (II)

- The circuit may turn into a sequential one
- Stuck-open requires two vector tests



Fault Coverage in a CMOS Chip



Summary of Stuck-Open Faults

- First Report:
  - Wadsack, Bell System Technology, J., 1978
- Recent Results
  - Woodhall et. al, ITC-87 (1-micron CMOS chips)
  - 4552 chips passed parametric test
  - 1255 chips (27.57%) failed tests for stuck-at faults
  - 44 chips (0.97%) failed tests for stuck-open faults
  - 4 chips with stuck-open faults passed tests for stuck-at faults
- Conclusion
  - Stuck-at faults are about 20 times more frequent than stuck-open faults
  - About 91% of chips with stuck-open faults may also have stuck-at faults
  - Faulty chips escaping tests for stuck-at faults = 0.121%

Memory Faults

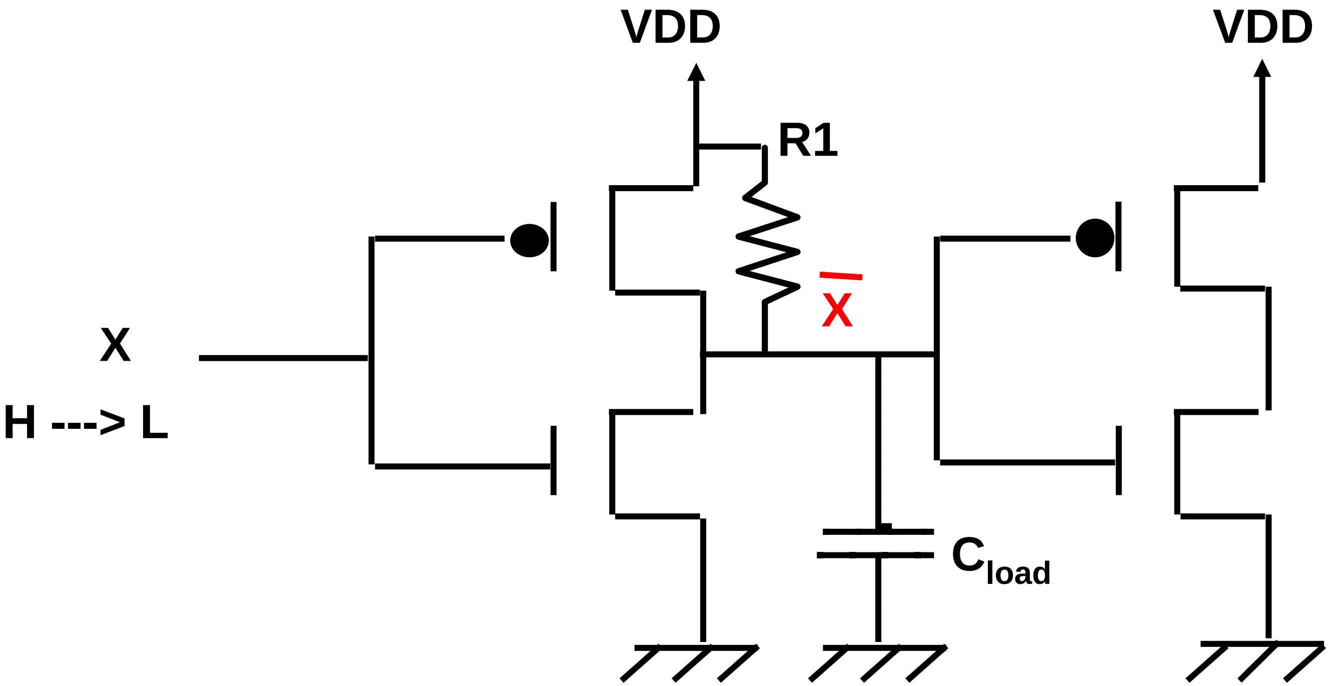
- Parametric Faults
  - Output Levels
  - Power Consumption
  - Noise Margin
  - Data Retention Time
- Functional Faults
  - Stuck Faults in Address Register, Data Register, and Address Decoder
  - Cell Stuck Faults
  - Adjacent Cell Coupling Faults
  - Pattern-Sensitive Faults

Delay Testing

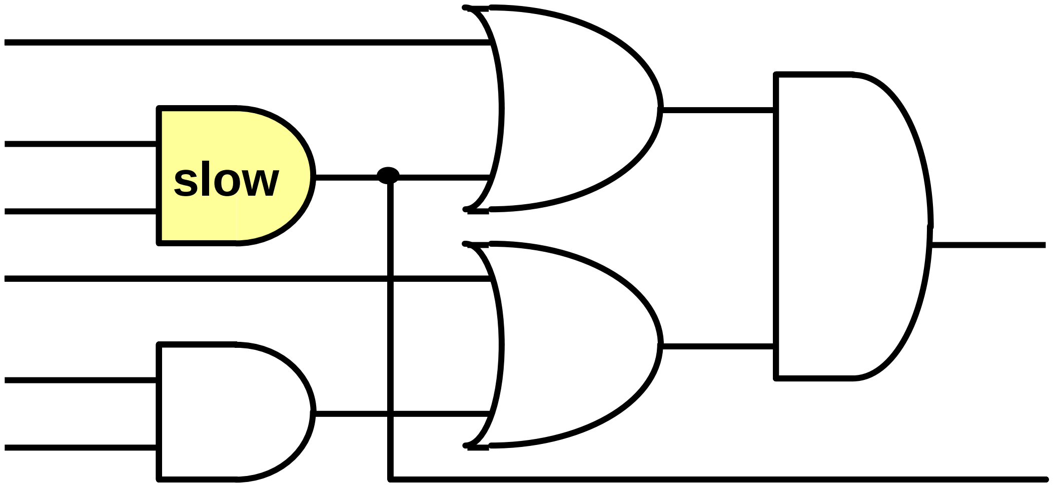
- Chip with timing defects
  - may pass the DC stuck-fault testing, but fail when operated at the system speed
  - For example, a chip may pass testing under 10 MHz operation, but fail under 100 MHz
- Delay Fault Models
  - Gate-Delay Fault
  - Path-Delay Fault

Gate-Delay Fault

- Slow to rise, slow to fall
  - $\overline{x}$ is slow to rise when channel resistance R1 is abnormally high



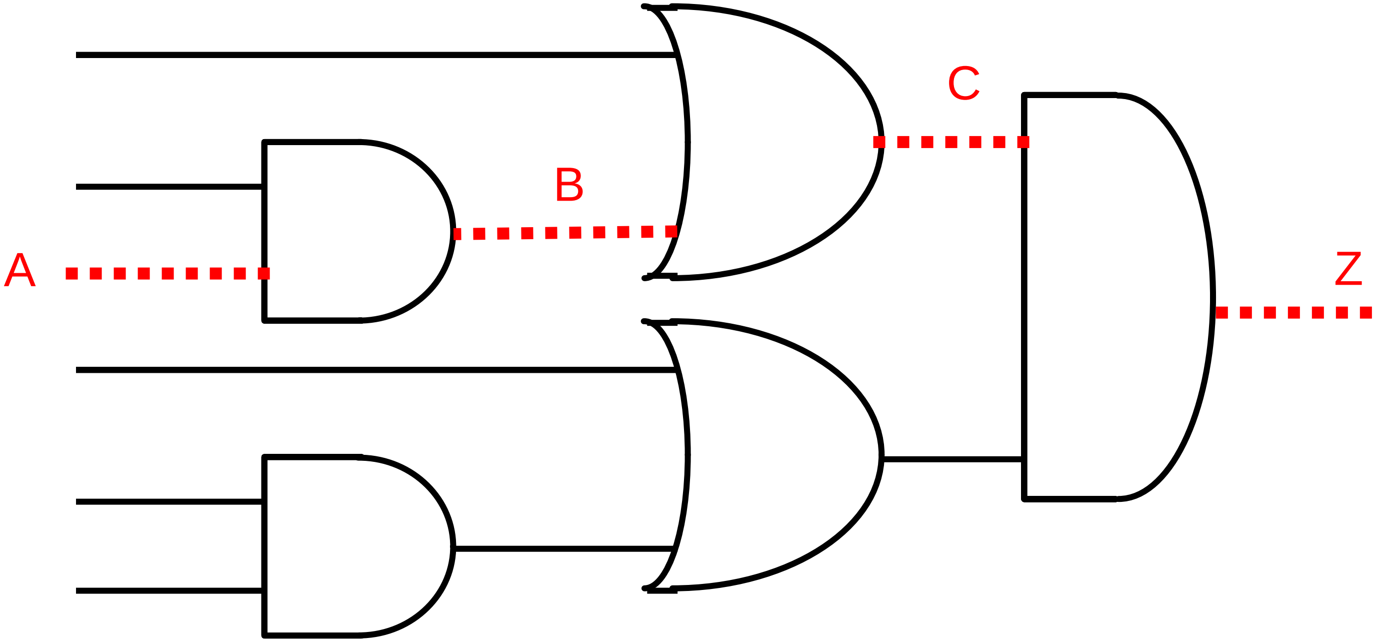
Gate-Delay Fault (cont'd)



- Test Based on Gate-Delay Fault
  - May not detect those delay faults that result from the accumulation of a number of small incremental delay defects along a path !! (Disadvantage)

Path-Delay Fault

- Associated with a Path (e.g. A-B-C-Z)
  - Whose delay exceeds the clock interval
- More complicated than gate-delay fault
  - Because the number of paths grows exponentially

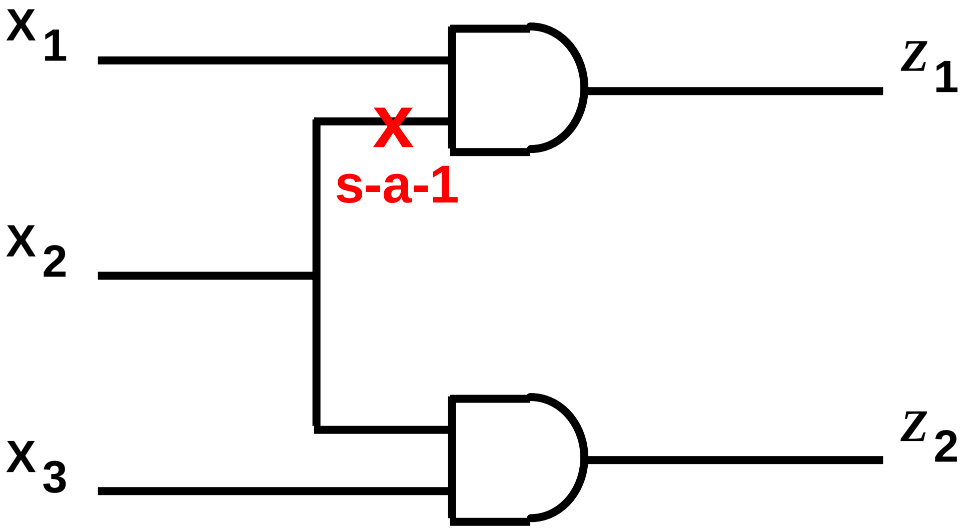


Why Logical Fault Modeling ?

- Fault analysis on logic rather than physical problem
  - Complexity is reduced
- Technology independent
  - Same fault model is applicable to many technologies
  - Testing and diagnosis methods remain valid despite changes in technology
- Tests derived
  - may be used for physical faults whose effect on circuit behavior is not completely understood or too complex to be analyzed
- Stuck-at fault is
  - The most popular logical fault model

Definition Of Fault Detection

- A test (vector) *t* detects a fault *f* iff
  - *t* detects *f* ⇔ $\mathbf{z}(t) \neq \mathbf{z}_f(t)$
- Example



$Z_1 = X_1X_2$ $Z_2 = X_2X_3$

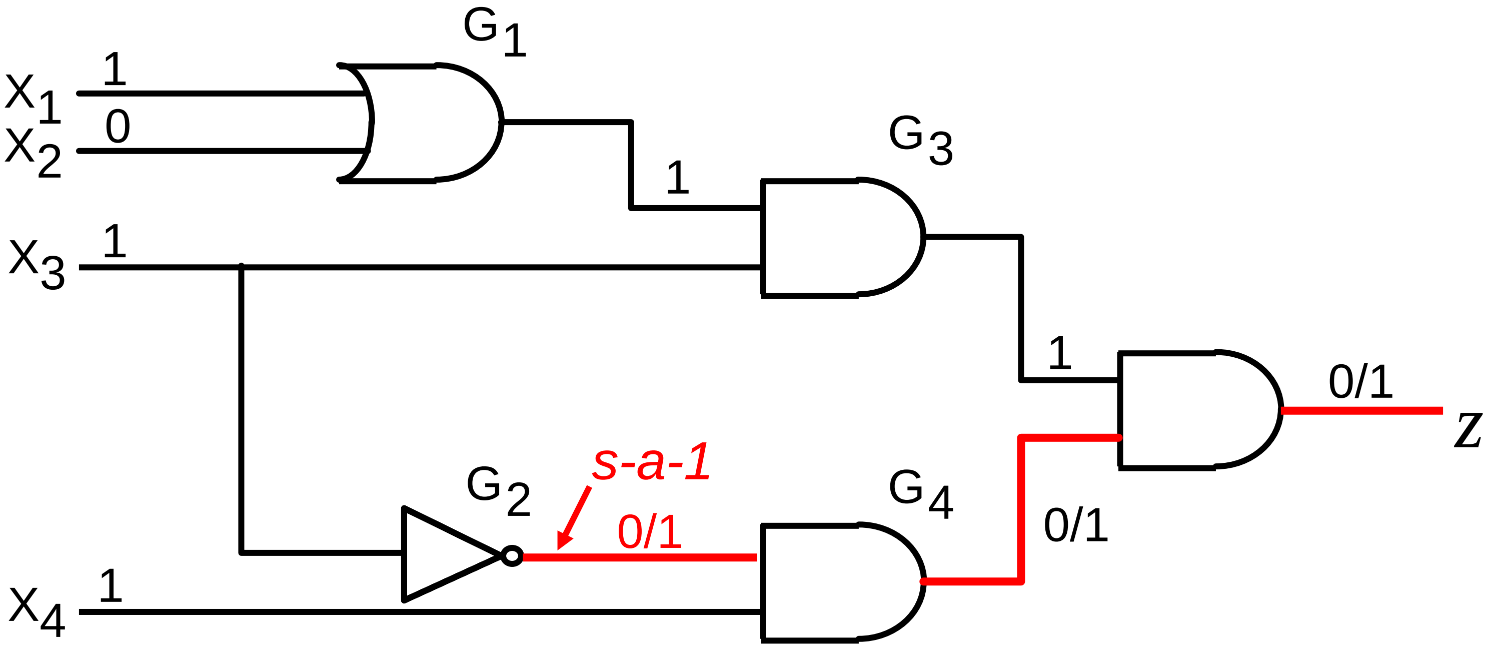
$Z_{1f} = X_1$ $Z_{2f} = X_2X_3$

The test (x1,x2,x3) = (100) detects *f* because z_1(100)=0 while z_{1f}(100)=1

Fault Detection Requirement

- A test *t* that detects a fault *f*
  - Activates *f* (or generate a fault effect) by creating different *v* and v_f values at the site of the fault
  - Propagates the error to a primary output *w* by making all the lines along at least one path between the fault site and *w* have different *v* and v_f values
- Sensitized Line:
  - A line whose value in response to the test changes in the presence of the fault *f* is said to be sensitized by the test in the faulty circuit
- Sensitized Path:
  - A path composed of sensitized lines is called a sensitized path

Fault Sensitization



z (1011)=0 z_f (1011)=1

1011 detects the fault *f* (G$_2$ stuck-at 1)

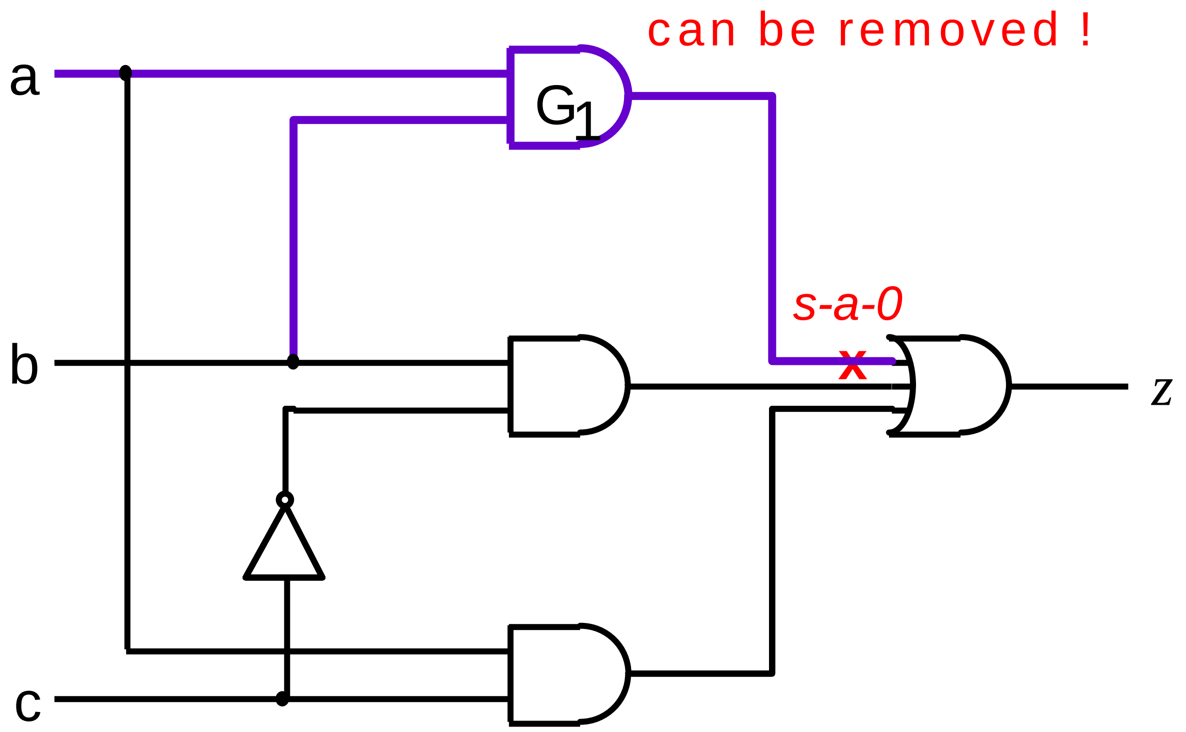
v/v$_f$: *v* = signal value in the fault free circuit

***v$_f$* = signal value in the faulty circuit**

Detectability

- A fault *f* is said to be detectable
  - if there exists a test *t* that detects *f* ; otherwise, *f* is an undetectable fault
- For an undetectable fault *f*
  - No test can simultaneously activate *f* and create a sensitized path to a primary output

Undetectable Fault

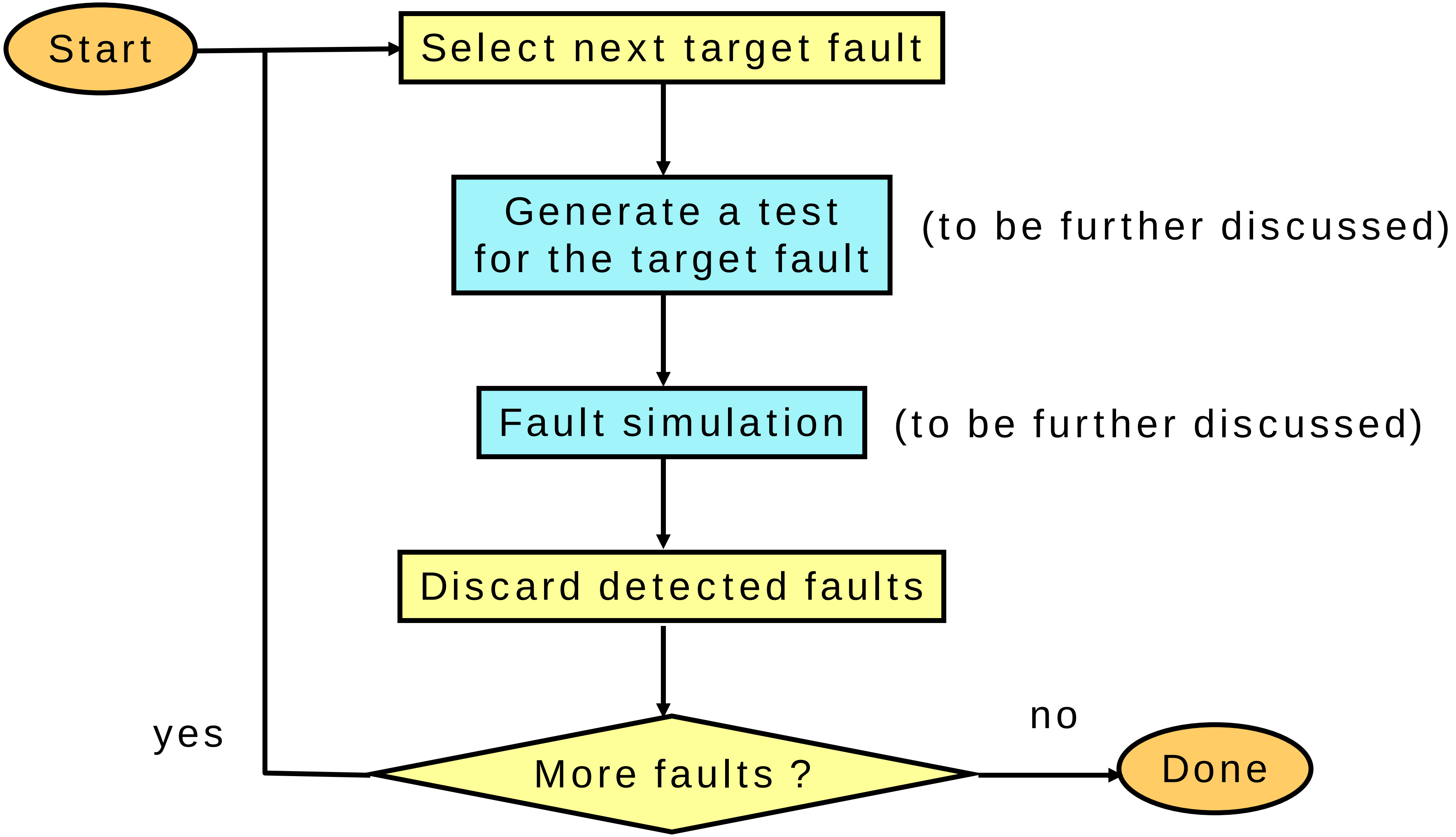


- G_1 output stuck-at-0 fault is undetectable
  - Undetectable faults do not change the function of the circuit
  - The related circuit can be deleted to simplify the circuit

Test Set

- Complete detection test set:
  - A set of tests that detect any detectable faults in a class of faults
- The quality of a test set
  - is measured by fault coverage
- Fault coverage:
  - Fraction of faults that are detected by a test set
- The fault coverage
  - can be determined by fault simulation
  - >95% is typically required for single stuck-at fault model
  - >99.9% in IBM

Typical Test Generation Flow



Fault Equivalence

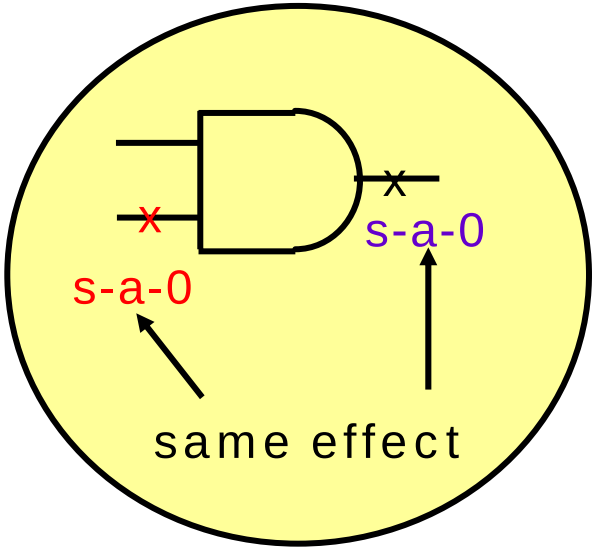
- Distinguishing test
  - A test *t* distinguishes faults α and β if

$$Z_{\alpha}(t) \oplus Z_{\beta}(t) = 1$$

- Equivalent Faults
  - Two faults, α & β are said to be equivalent in a circuit , iff the function under α is equal to the function under β for any input combination (sequence) of the circuit.
  - No test can distinguish between α and β
  - In other words, test-set(α) = test-set(β)

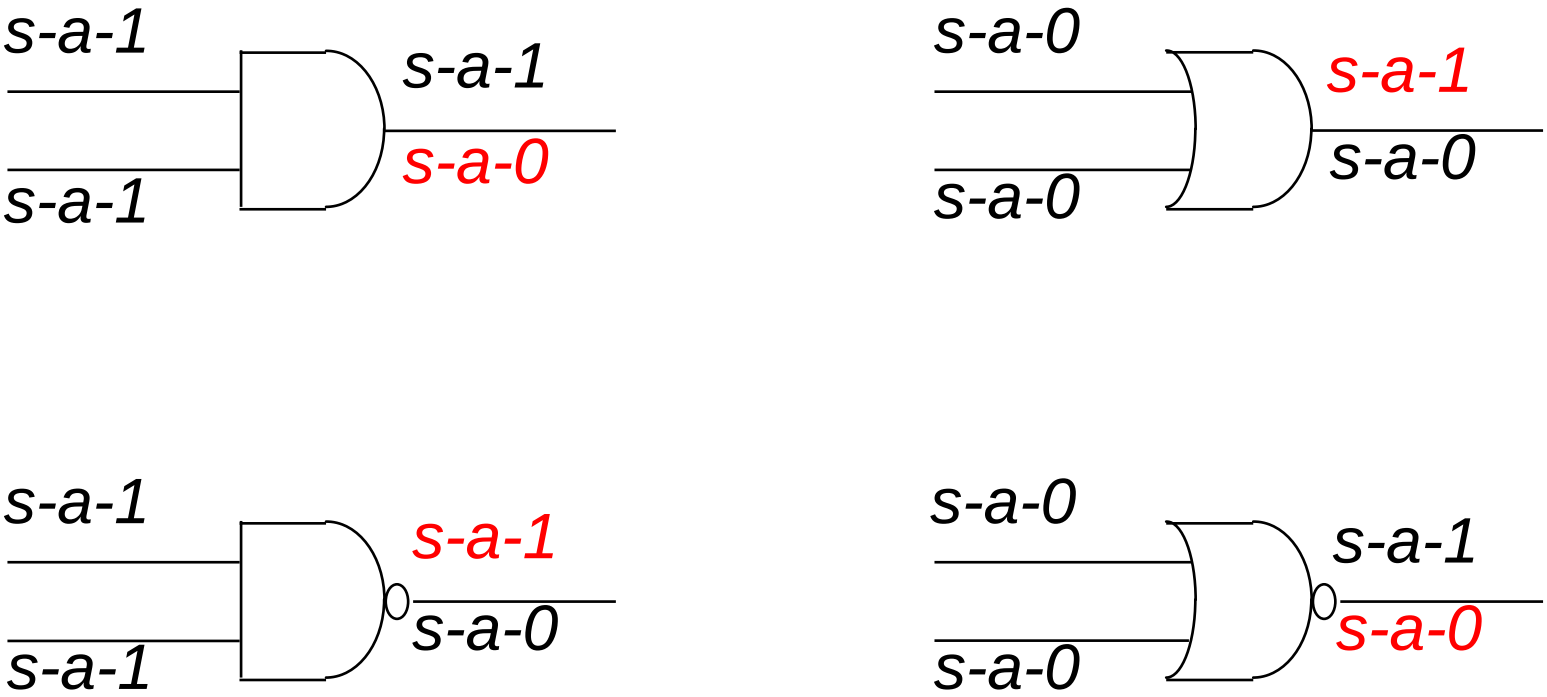
Fault Equivalence

- AND gate:
  - all *s-a-0* faults are equivalent
- OR gate:
  - all *s-a-1* faults are equivalent
- NAND gate:
  - all the input *s-a-0* faults and the output *s-a-1* faults are equivalent
- NOR gate:
  - all input *s-a-1* faults and the output *s-a-0* faults are equivalent
- Inverter:
  - input *s-a-1* and output *s-a-0* are equivalent
    input *s-a-0* and output *s-a-1* are equivalent



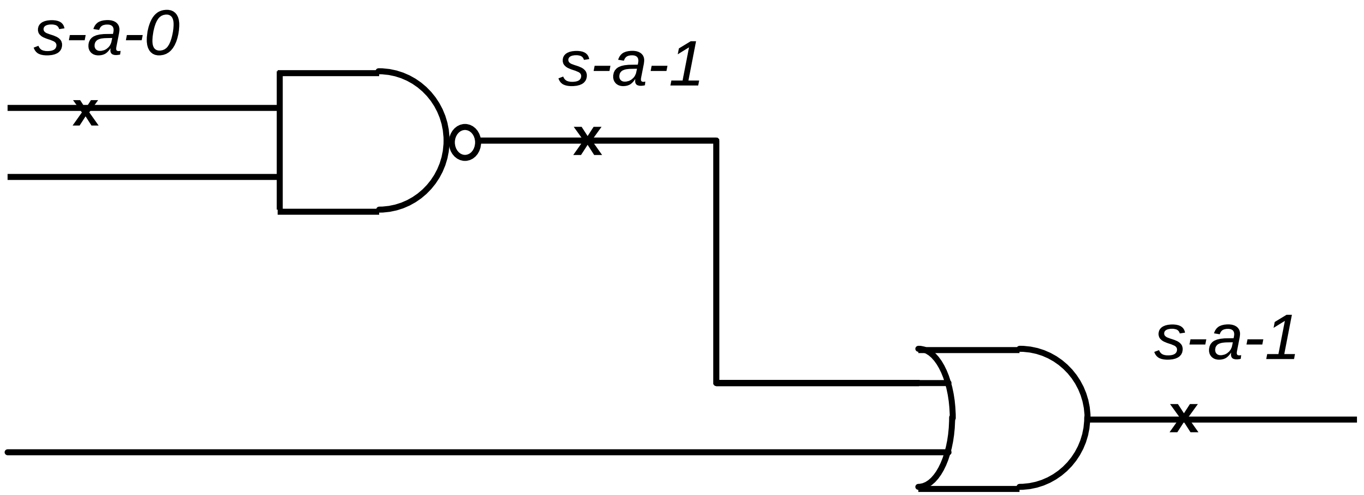
Equivalence Fault Collapsing

- *n*+2 instead of *2(n+1)* faults need to be considered for *n*-input gates



Equivalent Fault Group

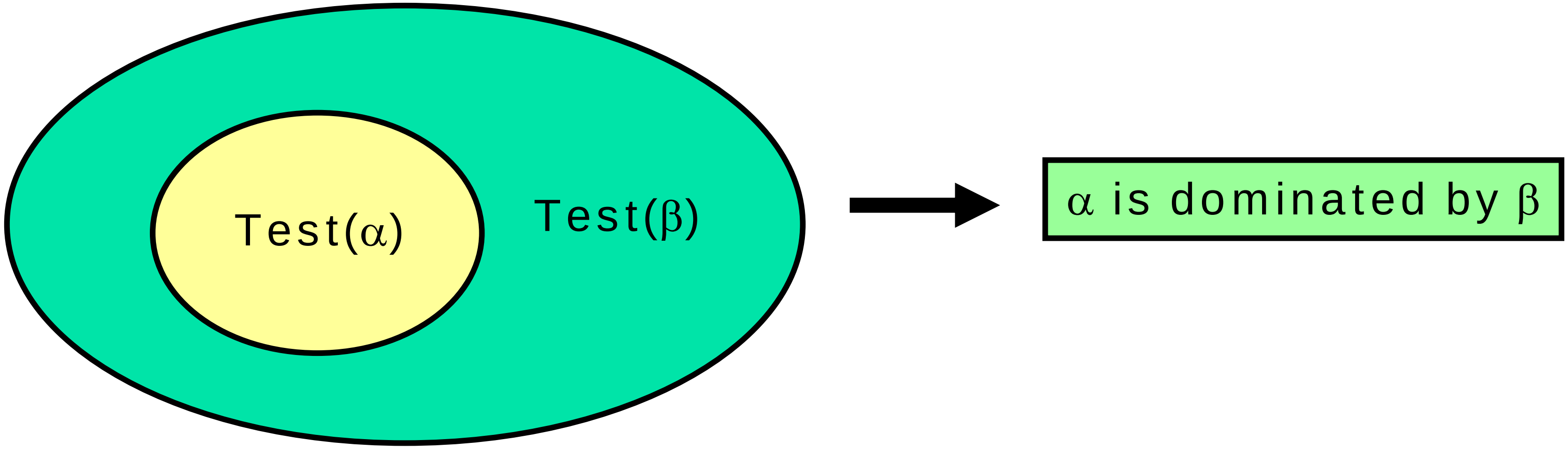
- In a combinational circuit
  - Many faults may form an equivalent group
  - These equivalent faults can be found by sweeping the circuit from the primary outputs to the primary inputs



Three faults shown are equivalent !

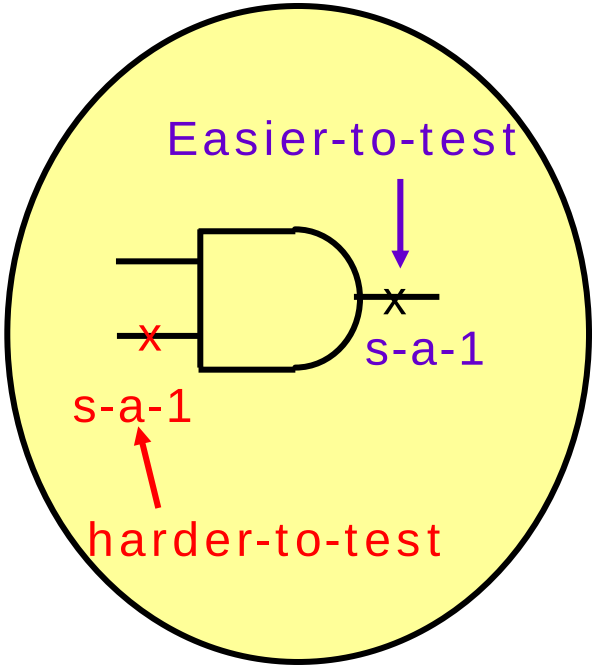
Fault Dominance

- Dominance Relation
  - A fault β is said to *dominate* another fault α in an irredundant circuit, iff every test (sequence) for α is also a test (sequence) for β.
  - I.e., test-set(β) > test-set(α)
  - No need to consider fault β for fault detection



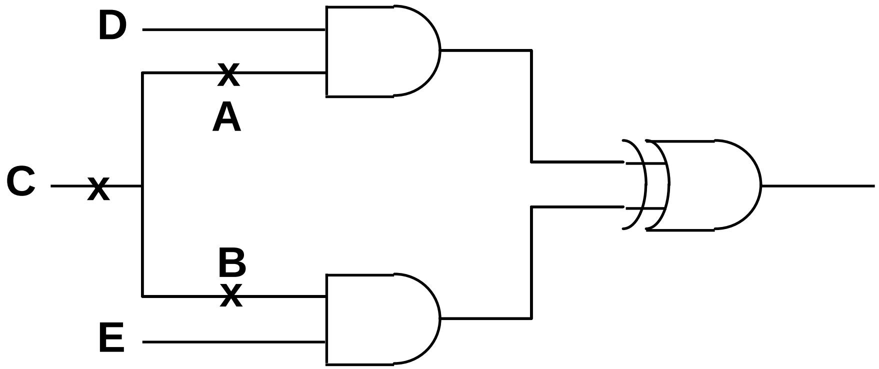
Fault Dominance

- AND gate:
  - Output *s-a-1* dominates any input *s-a-1*
- NAND gate:
  - Output *s-a-0* dominates any input *s-a-1*
- OR gate:
  - Output *s-a-0* dominates any input *s-a-0*
- NOR gate:
  - Output *s-a-1* dominates any input *s-a-0*
- Dominance fault collapsing:
  - The reduction of the set of faults to be analyzed based on dominance relation



Stem v.s. Branch Faults

C: stem of a multiple fanout
A & B: branches



- Detect A sa1:

$$z(t) \oplus z_f(t) = (\mathbf{CD} \oplus \mathbf{CE}) \oplus (\mathbf{D} \oplus \mathbf{CE}) = \mathbf{D} \oplus \mathbf{CD} = \mathbf{1}$$

$$\Rightarrow (\mathbf{C} = \mathbf{0},\ \mathbf{D} = \mathbf{1})$$

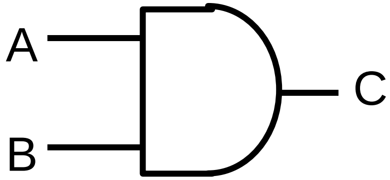
- Detect C sa1:

$$z(t) \oplus z_f(t) = (\mathbf{CD} \oplus \mathbf{CE}) \oplus (\mathbf{D} \oplus \mathbf{E}) = \mathbf{1}$$

$$\Rightarrow (\mathbf{C} = \mathbf{0},\ \mathbf{D} = \mathbf{1}) \text{ or } (\mathbf{C} = \mathbf{0},\ \mathbf{E} = \mathbf{1})$$

- Hence, C sa1 dominates A sa1
- Similarly
  - C sa1 dominates B sa1
  - C sa0 dominates A sa0
  - C sa0 dominates B sa0
- In general, there might be no equivalence or dominance relations between stem and branch faults

Analysis of a Single Gate

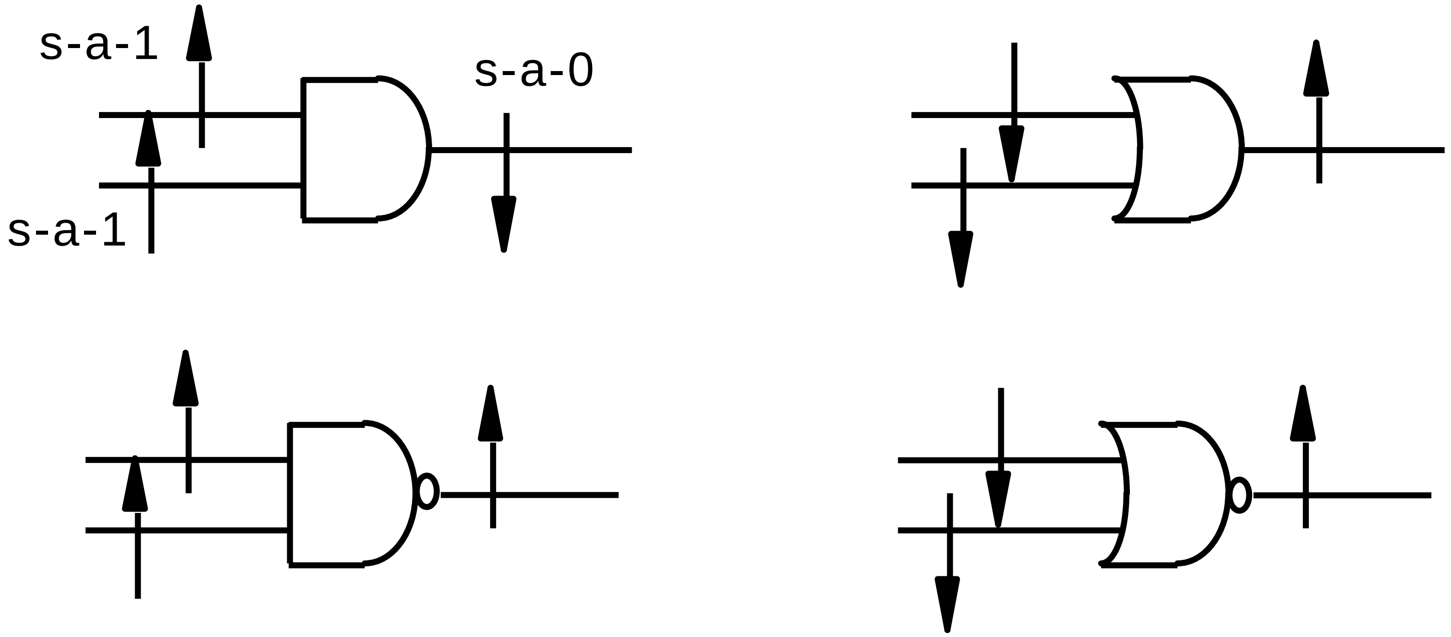


AB	C	A sa1	B sa1	C sa1	A sa0	B sa0	C sa0
00	0			1			
01	0	1		1			
10	0		1	1			
11	1				0	0	0

- Fault Equivalence Class
  - (A s-a-0, B s-a-0, C s-a-0)
- Fault Dominance Relations
  - (C s-a-1 > A s-a-1) and (C s-a-1 > B s-a-1)
- Faults that can be ignored:
  - A s-a-0, B s-a-0, and C s-a-1

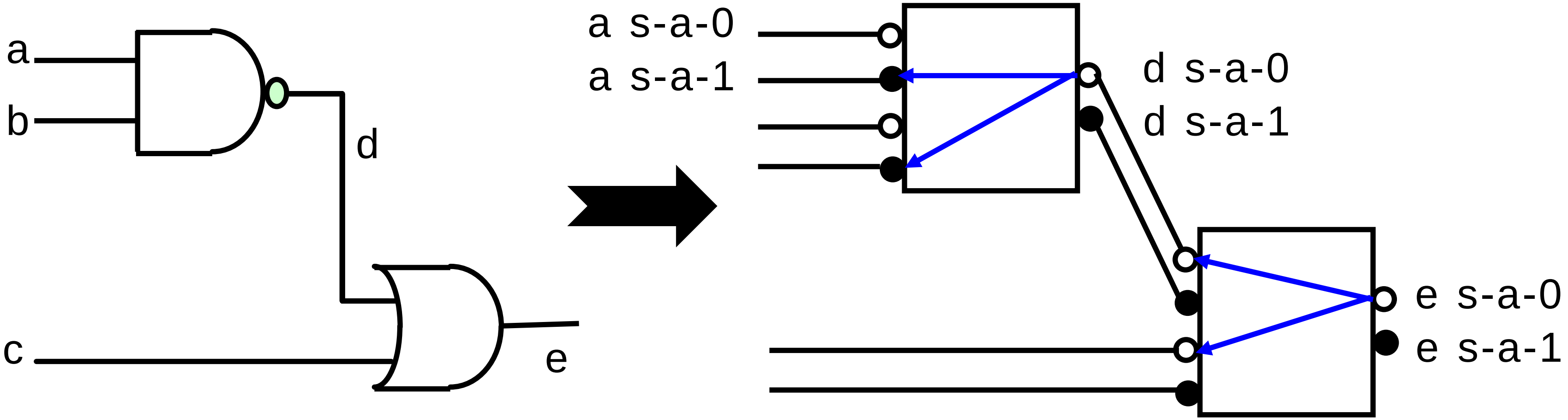
Fault Collapsing

- Equivalence + Dominance
  - For each *n*-input gate, we only need to consider *n+1* faults during test generation

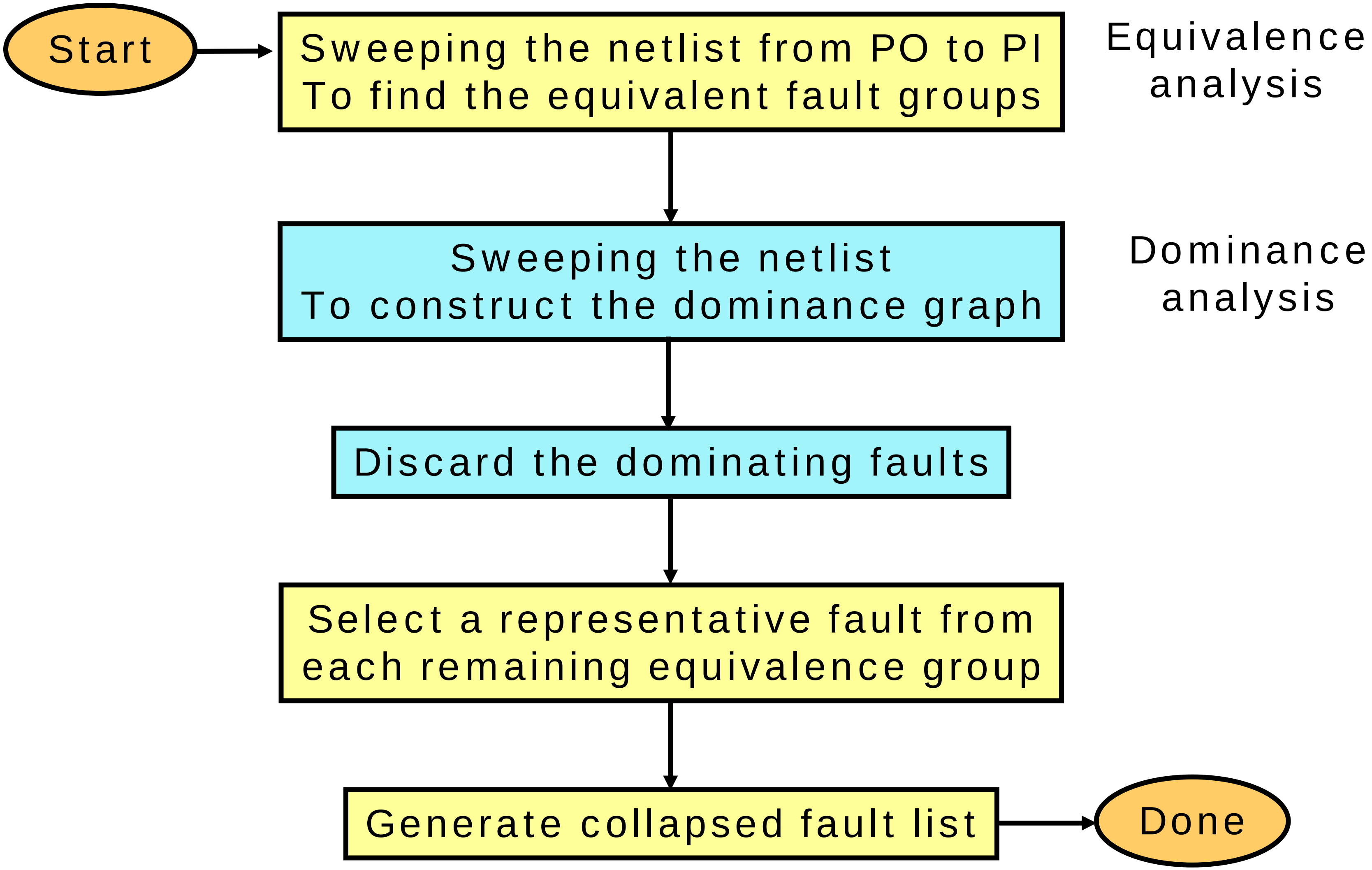


Dominance Graph

- Rule
  - When fault α dominates fault β, then an arrow is pointing from α to β
- Application
  - Find out the transitive dominance relations among faults



Fault Collapsing Flow

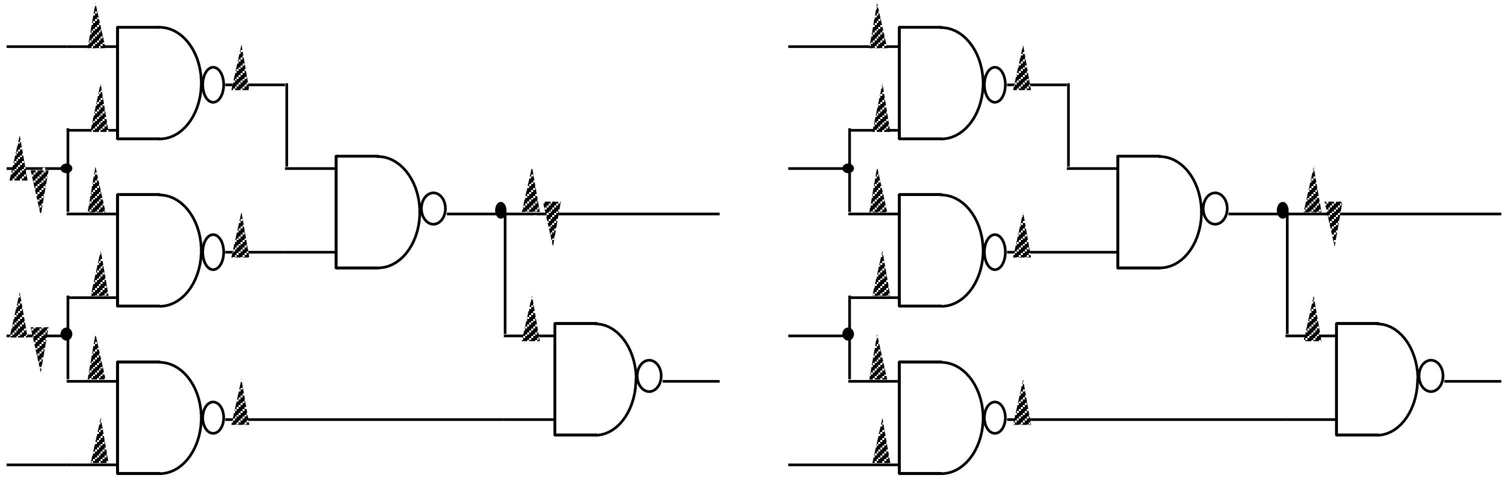

Start
Sweeping the netlist from PO to PI
To find the equivalent fault groups
Equivalence analysis
Sweeping the netlist
To construct the dominance graph
Dominance analysis
Discard the dominating faults
Select a representative fault from each remaining equivalence group
Generate collapsed fault list
Done

Prime Fault

- α is a prime fault if every fault that is dominated by α is also equivalent to α
- Representative Set of Prime Fault (RSPF)
  - A set that consists of exactly one prime fault from each equivalence class of prime faults
  - True minimal RSPF is difficult to find

Why Fault Collapsing ?

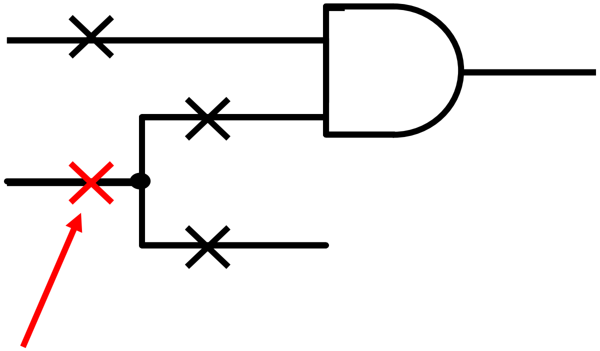
- Memory and CPU-time saving
- Ease testing generation and fault simulation



*** 30 total faults → 12 prime faults**

Checkpoint Theorem

- Checkpoints for test generation
  - A test set detects every fault on the primary inputs and fanout branches is complete
  - I.e., this test set detects all other faults too
  - Therefore, primary inputs and fanout branches form a *sufficient* set of checkpoints in test generation
  - In fanout-free combinational circuits, primary inputs are the checkpoints

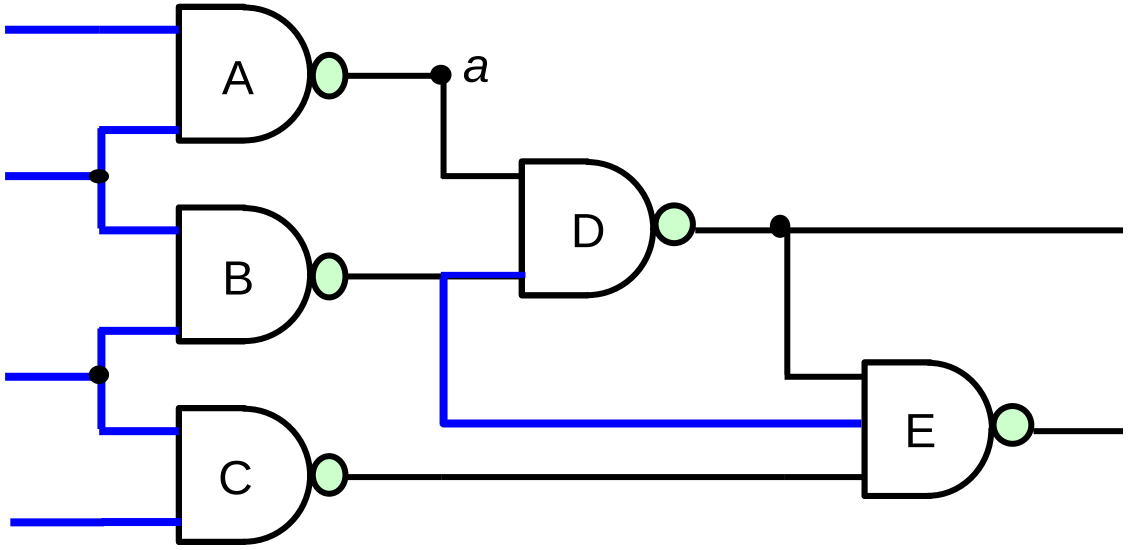


Stem is not a checkpoint !

Why Inputs + Branches Are Enough ?

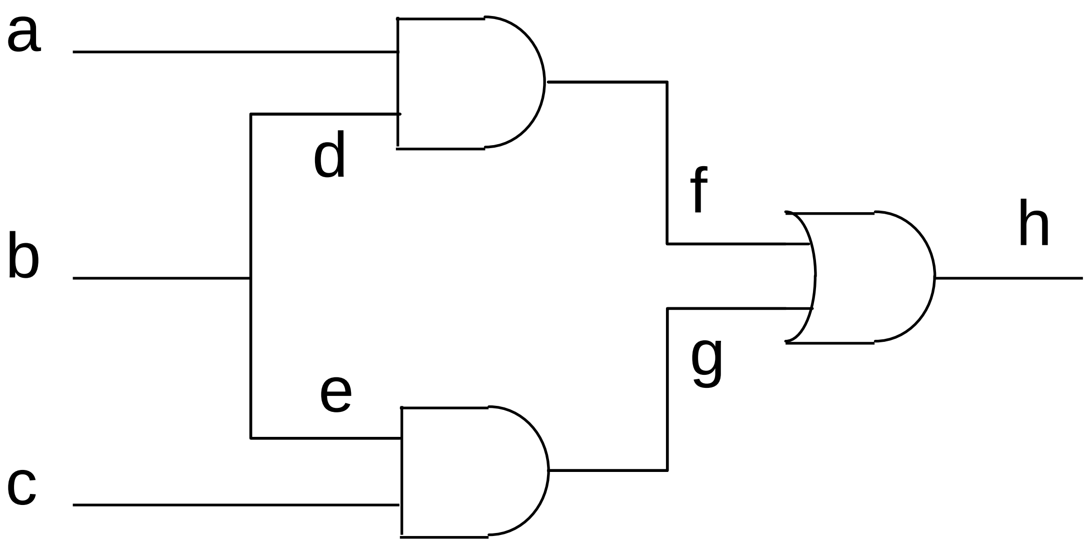
- Example
  - Checkpoints are marked in blue
  - Sweeping the circuit from PI to PO to examine every gate, e.g., based on an order of (A->B->C->D->E)
  - For each gate,

output faults are detected if every input fault is detected



Fault Collapsing + Checkpoint

- Example:
  - 10 checkpoint faults
  - a s-a-0 <=> d s-a-0 , c s-a-0 <=> e s-a-0
    b s-a-0   > d s-a-0   , b s-a-1 > d s-a-1
  - 6 tests are enough



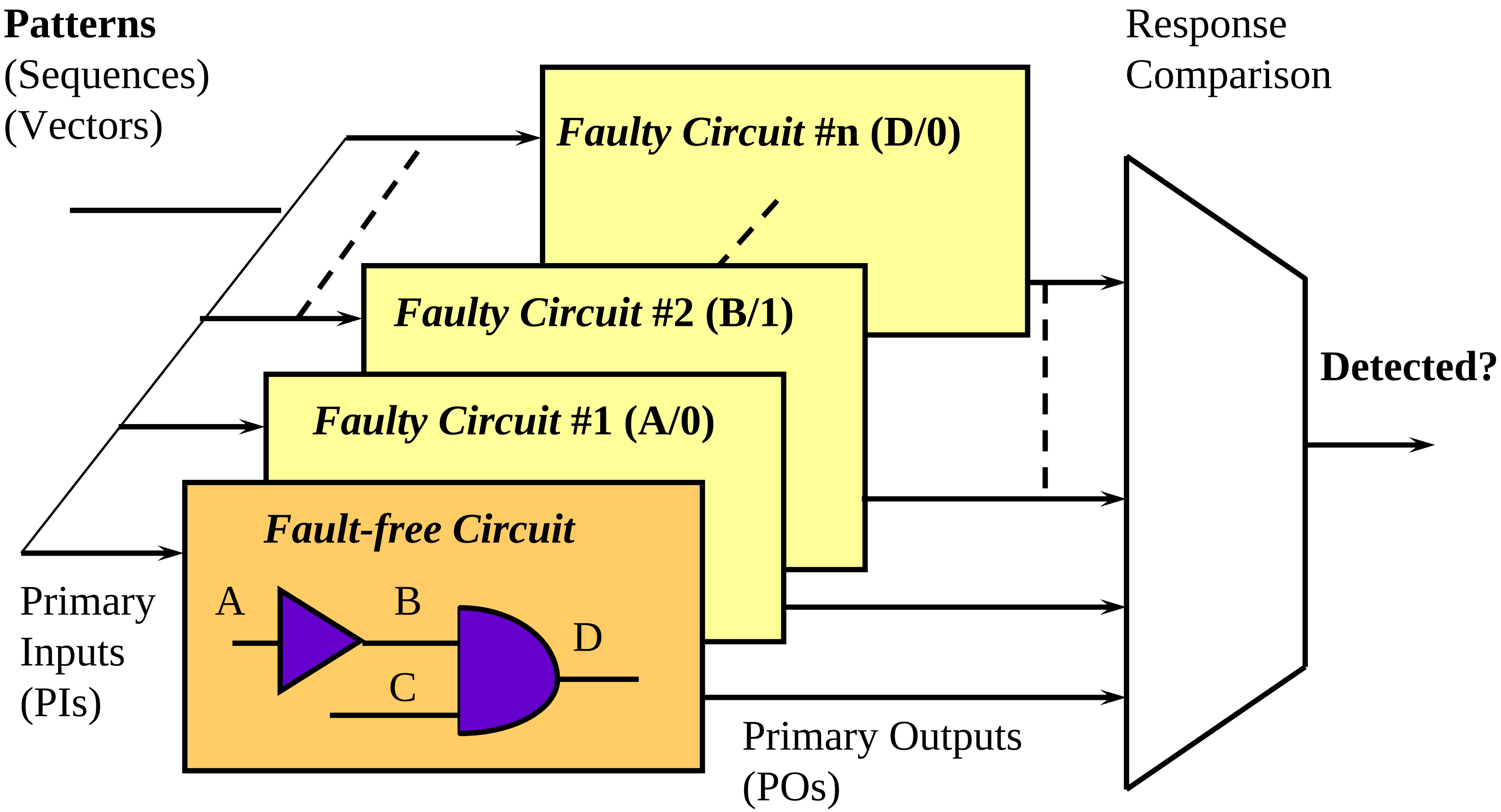
Outline

- Introduction
- Fault Modeling
- Fault Simulation
- Test Generation
- Design For Testability

Why Fault Simulation ?

- To evaluate the quality of a test set
  - I.e., to compute its fault coverage
- Part of an ATPG program
  - A vector usually detected multiple faults
  - Fault simulation is used to compute the faults accidentally detected by a particular vector
- To construct fault-dictionary
  - For post-testing diagnosis

Conceptual Fault Simulation

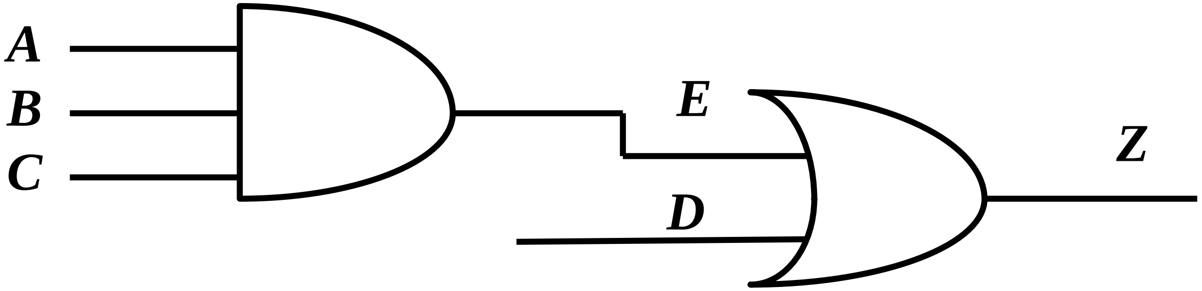


Logic simulation on both good (fault-free) and faulty circuits

Some Basics for Logic Simulation

- For fault simulation purpose,
  - mostly the gate delay is assumed to be zero unless the delay faults are considered. Our main concern is the functional faults
- The logic values
  - can be either two (0, 1) or three values (0, 1, X)
- Two simulation mechanisms:
  - Oblivious compiled-code:
    - circuit is translated into a program and all gates are executed for each pattern. (may have redundant computation)
  - Interpretive event-driven:
    - Simulating a vector is viewed as a sequence of value-change events propagating from the PI's to the PO's
    - Only those logic gates affected by the events are re-evaluated

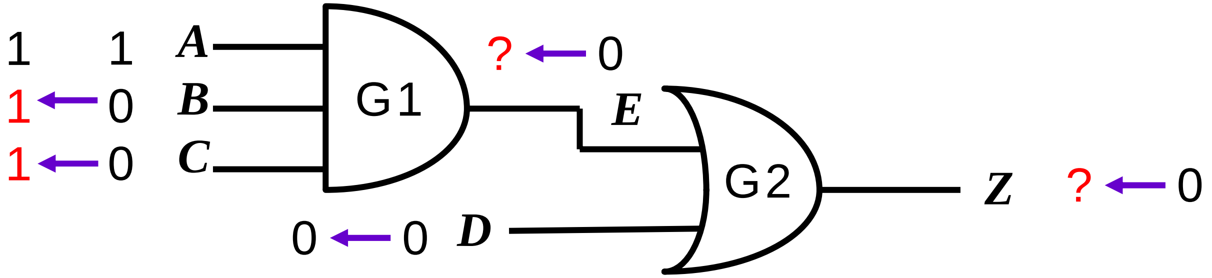
Compiled-Code Simulation

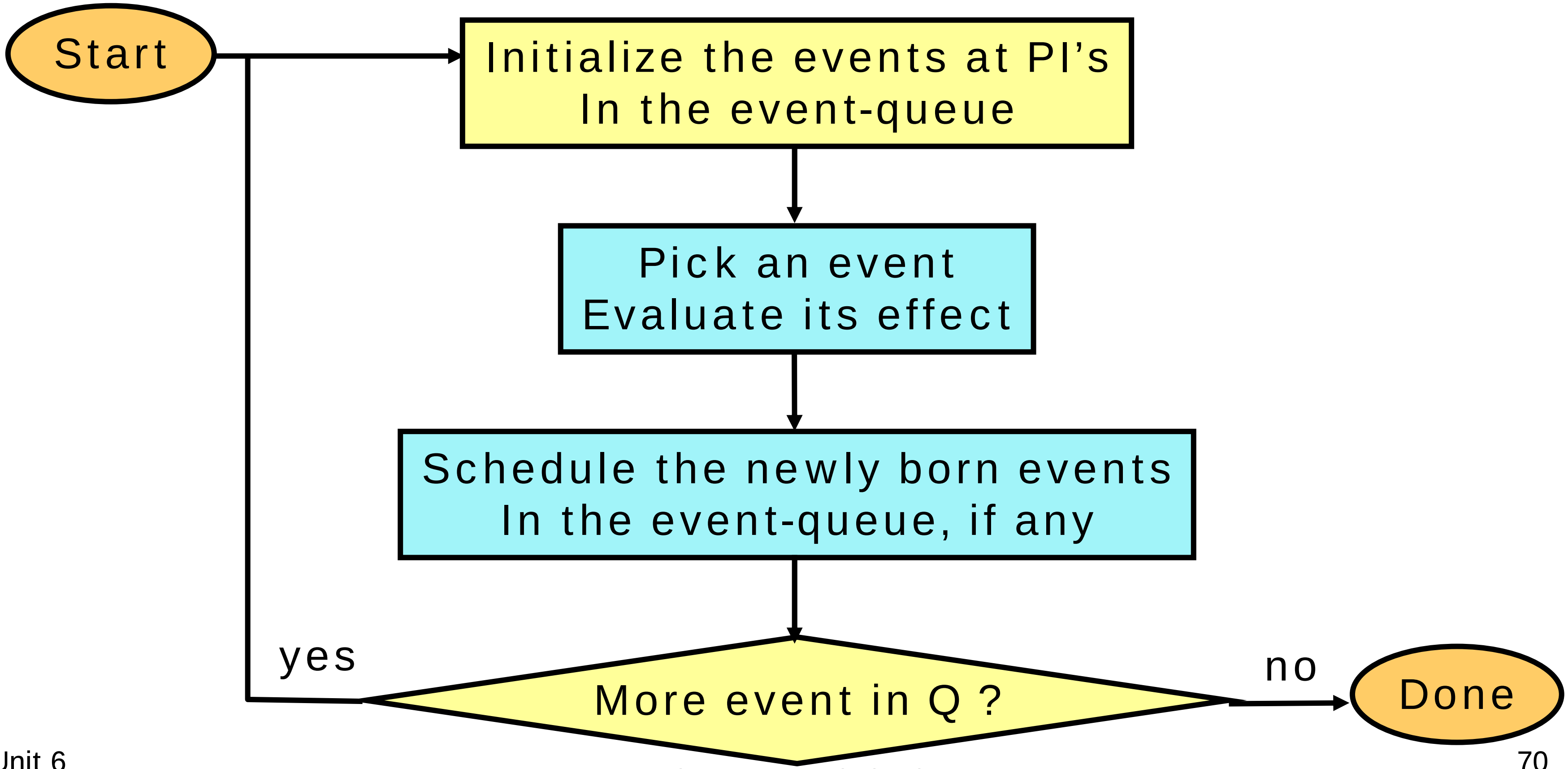


- Compiled code

```
LOAD    A    /* load accumulator with value of  A */
AND     B    /* calculate A and B */
AND     C    /* calculate  E = AB and C */
OR      D    /* calculate  Z = E or D */
STORE   Z    /* store result of  Z */
```

Event-Driven Simulation




Start

Initialize the events at PI’s
In the event-queue

Pick an event
Evaluate its effect

Schedule the newly born events
In the event-queue, if any

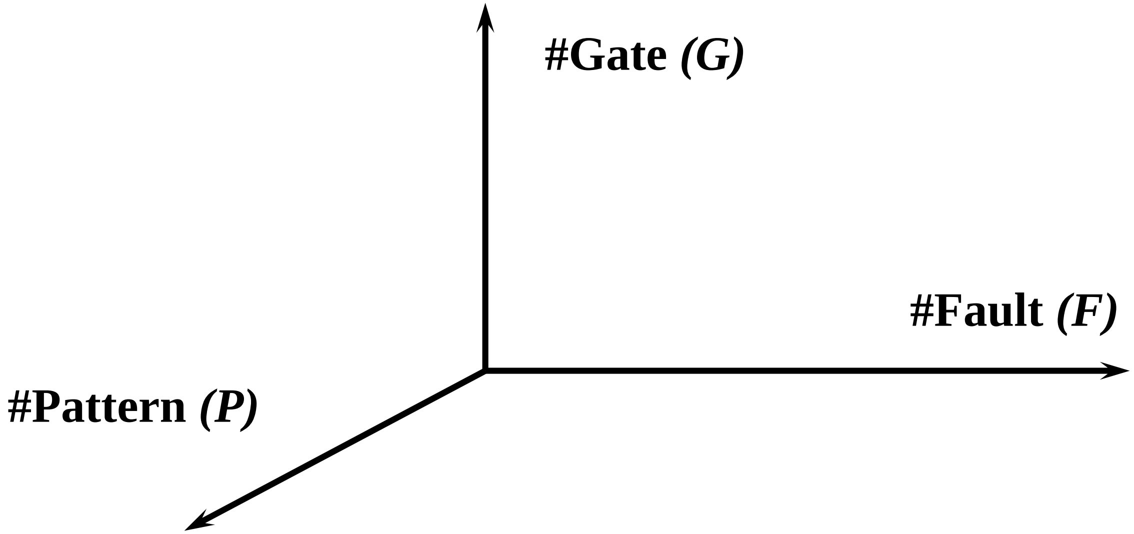
More event in Q ?

yes

no

Done

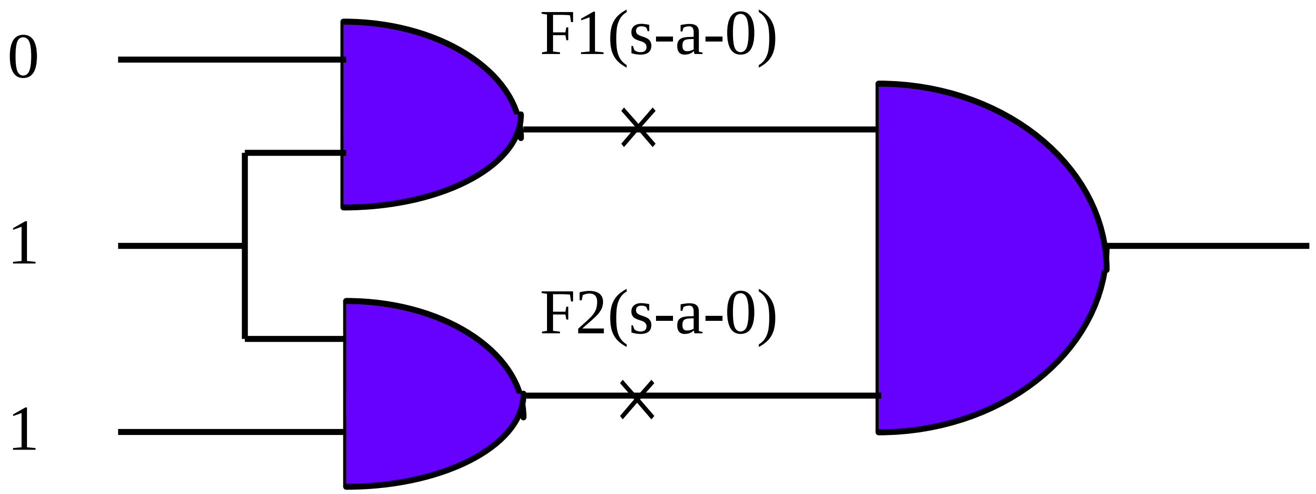
Complexity of Fault Simulation



- Complexity ~ $F \cdot P \cdot G$ ~ $O(G^3)$
- The complexity is higher than logic simulation by a factor of *F*, while usually is much lower than ATPG
- The complexity can be greatly reduced using
  - Fault dropping and other advanced techniques

Characteristics of Fault Simulation

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



Fault Simulation Techniques

- Serial Fault Simulation
  - trivial single-fault single-pattern
- Parallel Fault Simulation
- Deductive Fault Simulation
- Concurrent Fault Simulation

Parallel Fault Simulation

- Simulate multiple circuits at a time:
  - 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
- Extra Cost:
  - An event, a value-change of a single fault or fault-free circuit leads to the computation of the entire word
  - The fault-free logic simulation is repeated for each pass

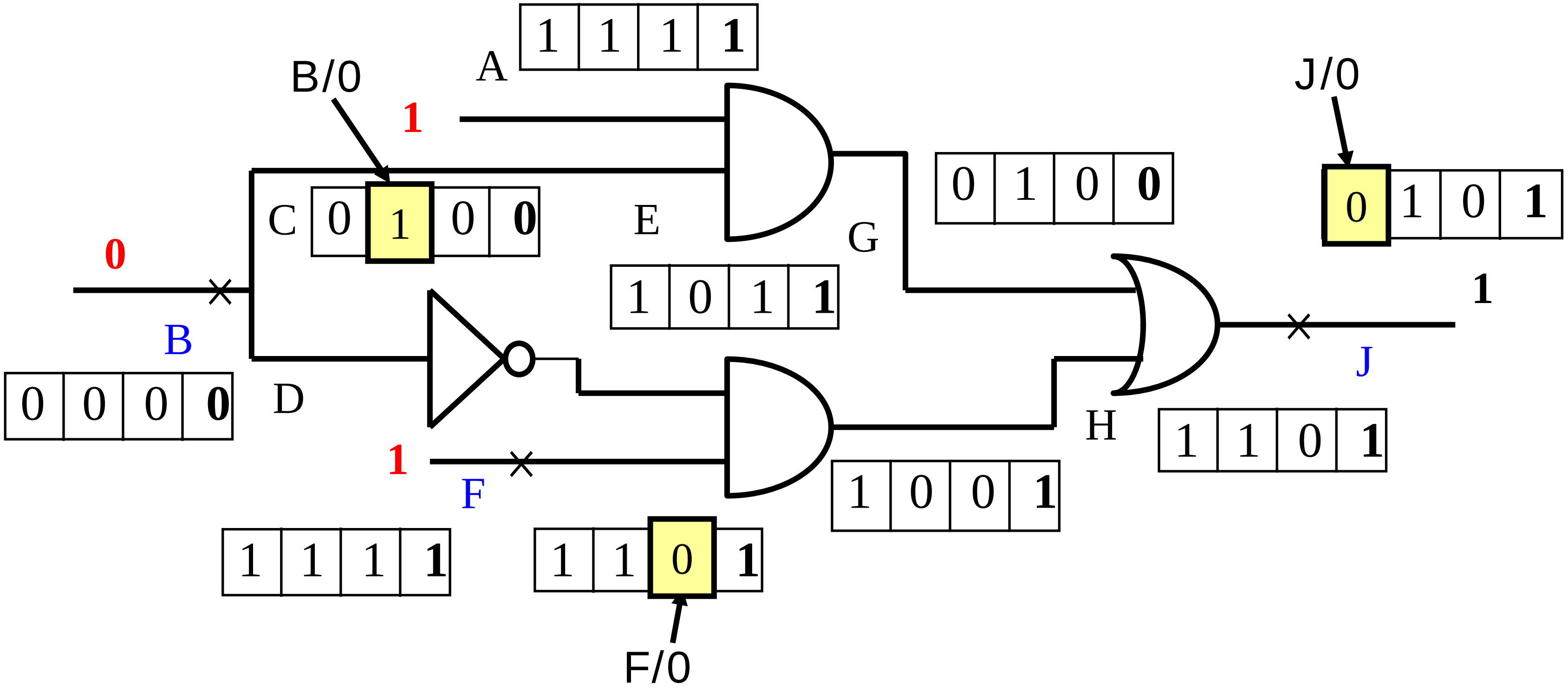
Example: Parallel Fault Simulation

- Consider three faults:

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

fault-free

J/0	B/1	F/0	FF

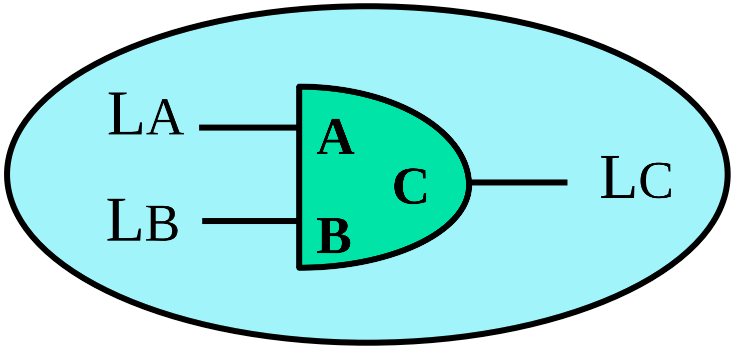


Deductive Fault Simulation

- Simulate all faulty circuits in one pass
  - For each pattern, sweep the circuit from PI's to PO's.
  - During the process, a list of faults is associated with each line
  - The list contains faults that would produce a fault effect on this line
  - The union fault list at every PO contains the detected faults by the simulated input vector
- Major operation: fault list propagation
  - Related to the gate types and values
  - The size of the list may grow dynamically, leading to a 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,

$L_C = L_A + L_B + \{C/0\}$

Controlling cases: **Case 2:** A=1, B=0, C=0 at fault-free,

$L_C = \overline{L_A} * L_B + \{C/1\}$

Case 3: A=0, B=0, C=0 at fault-free,

$L_C = L_A * L_B + \{C/1\}$

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

Fault List Propagation Rule

- Notations:
  - Let *I* be the set of inputs of a gate *Z* with controlling value *c* and inversion *i*

    (i.e., i is 0 for AND, OR and 1 for NAND, NOR)
  - Let *C* be the set of inputs with value *c*

Non-controlling case:

union

$$\text{if } C=\phi \text{ then } L_z = \{\bigcup_{j\in I} L_j\} \cup \{Z \ s-a-(c\oplus i)\}$$

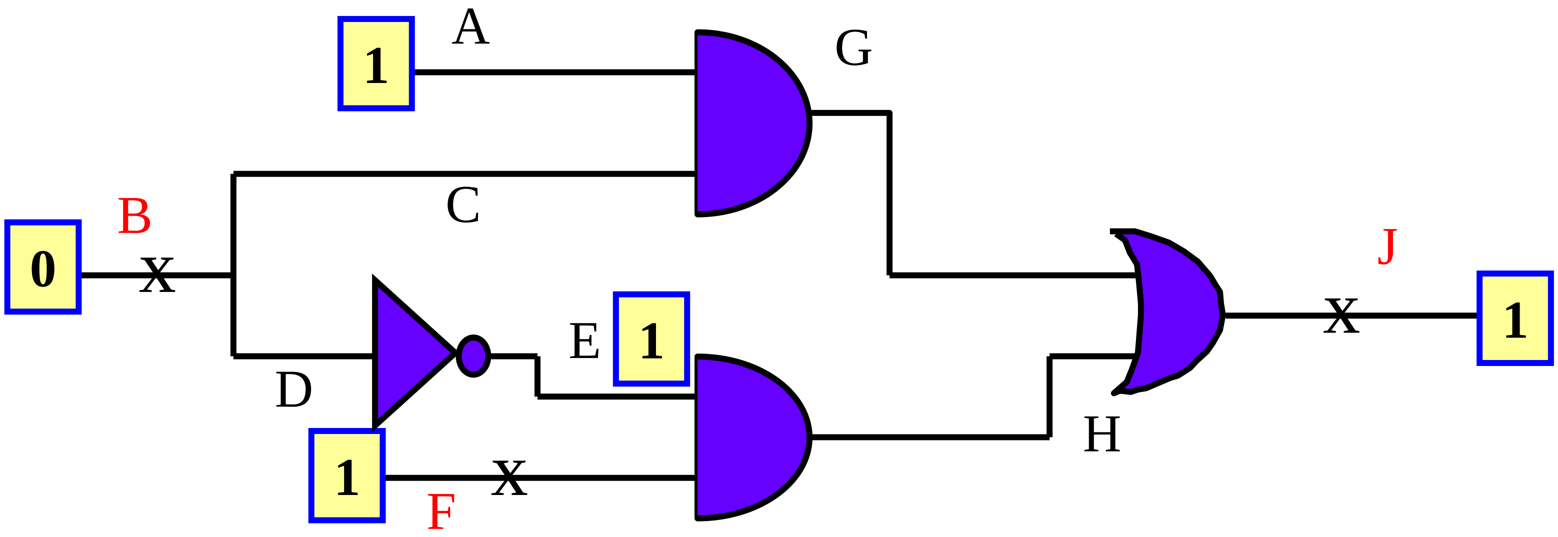
Controlling cases:

intersection

$$\text{else } L_z = \{\bigcap_{j\in C} L_j\} - \{\bigcup_{j\in I-C} L_j\} \mathrm{Y} \{Z \ s-a-(\bar{c}\oplus i)\}$$

Example: Deductive Simulation (1)

- Consider 3 faults: B/1, F/0, and J/0

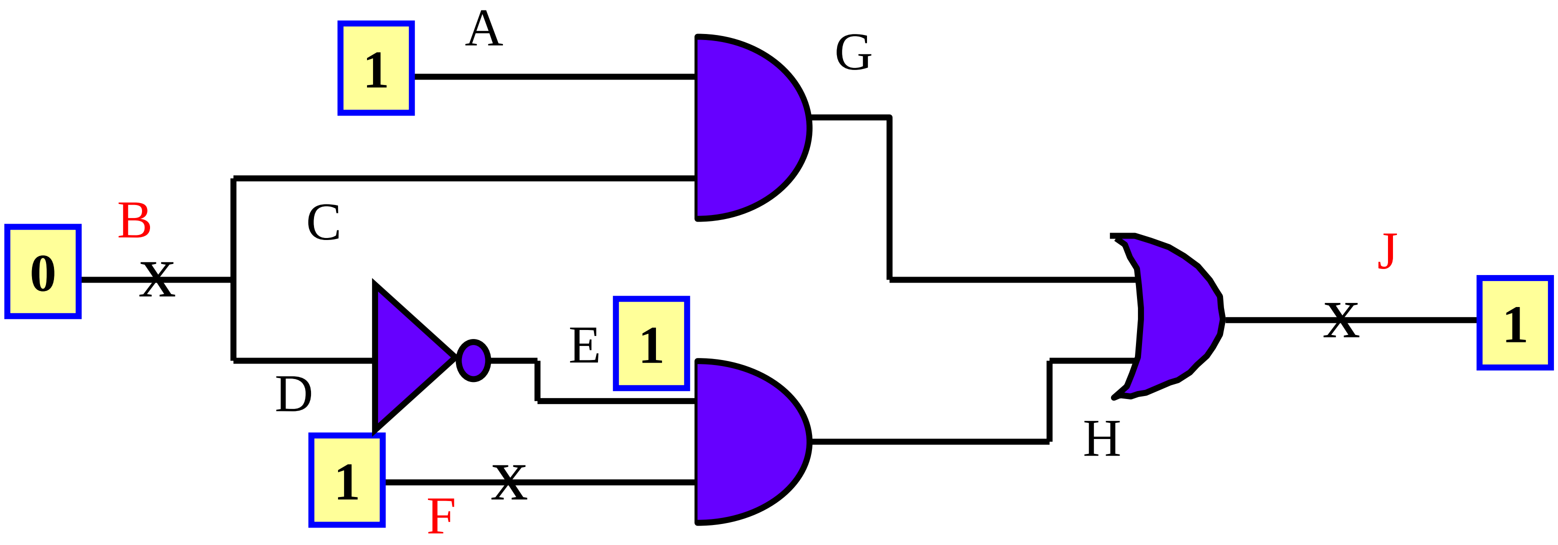


Fault List at PI's:

L$_B$ = {B/1}, L$_F$ = {F/0}, L$_A$ = ϕ, L$_C$=L$_D$ = {B/1}

Example: Deductive Simulation (2)

- Consider 3 faults: B/1, F/0, and J/0



Fault Lists at G and E:

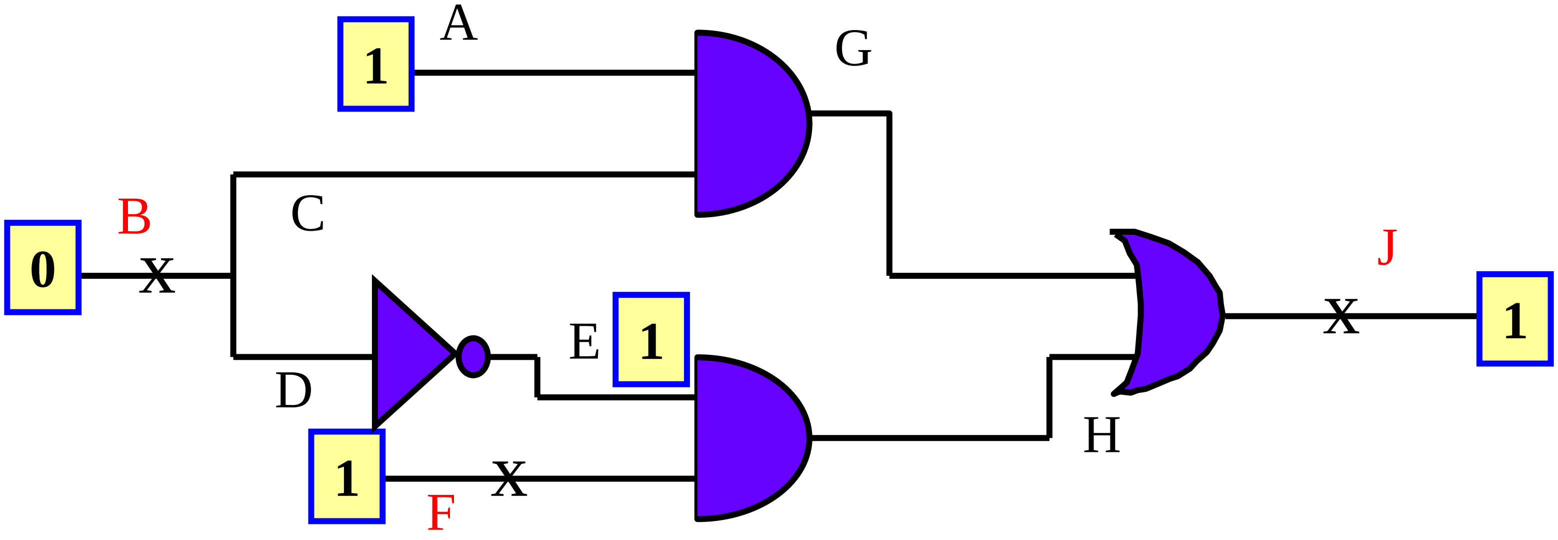
L_B = {B/1}, L_F = {F/0}, $L_A = \phi$, L_C=L_D = {B/1},

$L_G = (\overline{L_A} * L_C)$ = {B/1}

$L_E = (L_D)$ = {B/1}

Example: Deductive Simulation (3)

- Consider 3 faults: B/1, F/0, and J/0



Computed Fault List at H:

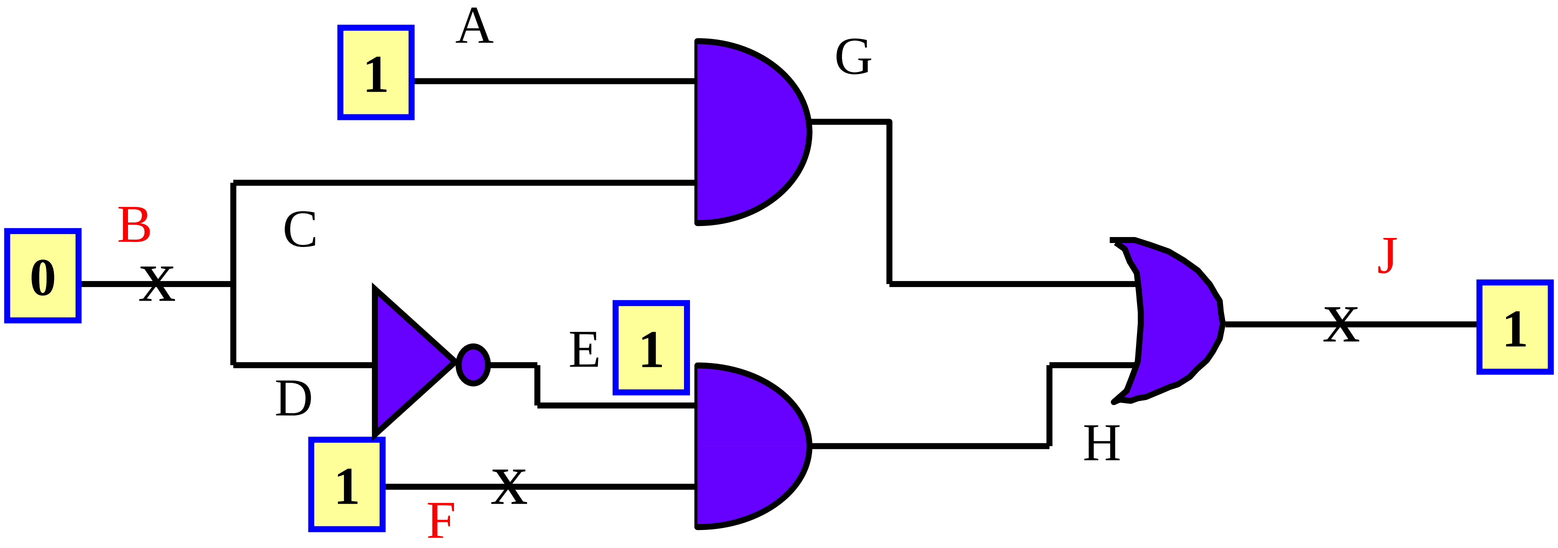
$L_B = \{B/1\}$, $L_F = \{F/0\}$, $L_C = L_D = \{B/1\}$,

$L_G = \{B/1\}$, $L_E = \{B/1\}$

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

Example: Deductive Simulation (4)

- Consider 3 faults: B/1, F/0, and J/0



Final Fault List at the output J:

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

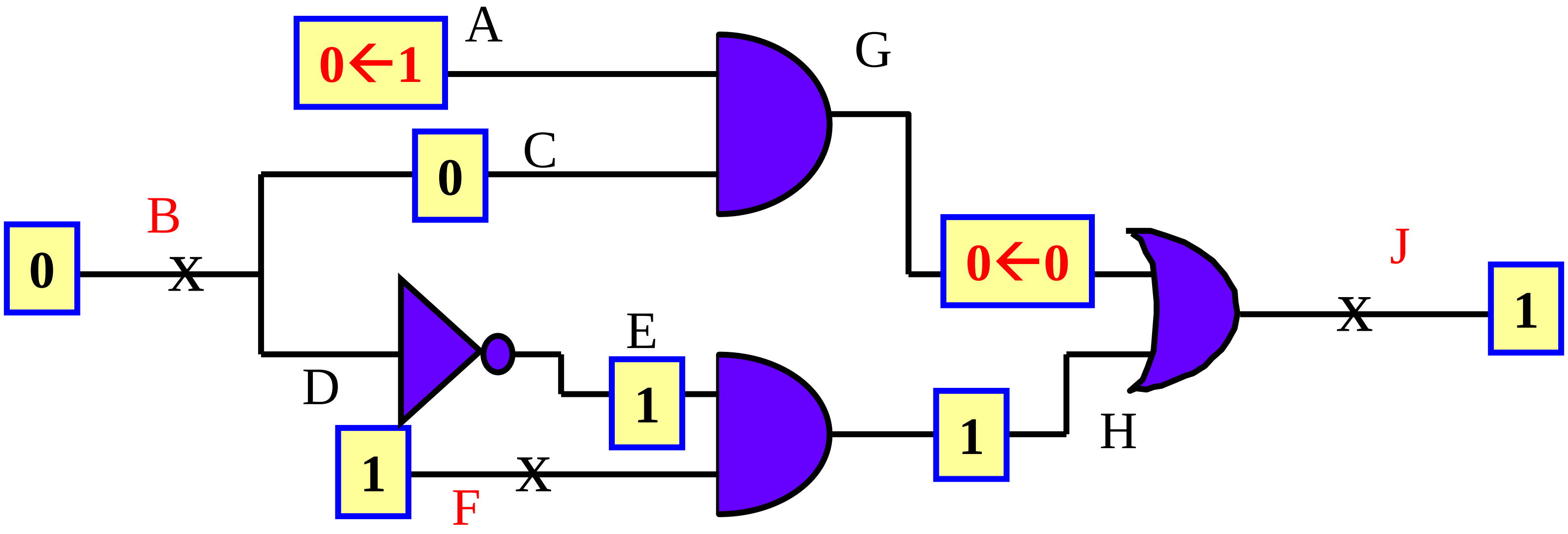
$L_G = \{B/1\}$, $L_E = \{B/1\}$

$L_H = \{B/1, F/0\}$,

$L_J = \{F/0, J/0\}$

Example: Deductive Simulation

- When A changes from 1 to 0



Event-driven operation:

L_B = {B/1}, L_F = {F/0}, L_A = ϕ

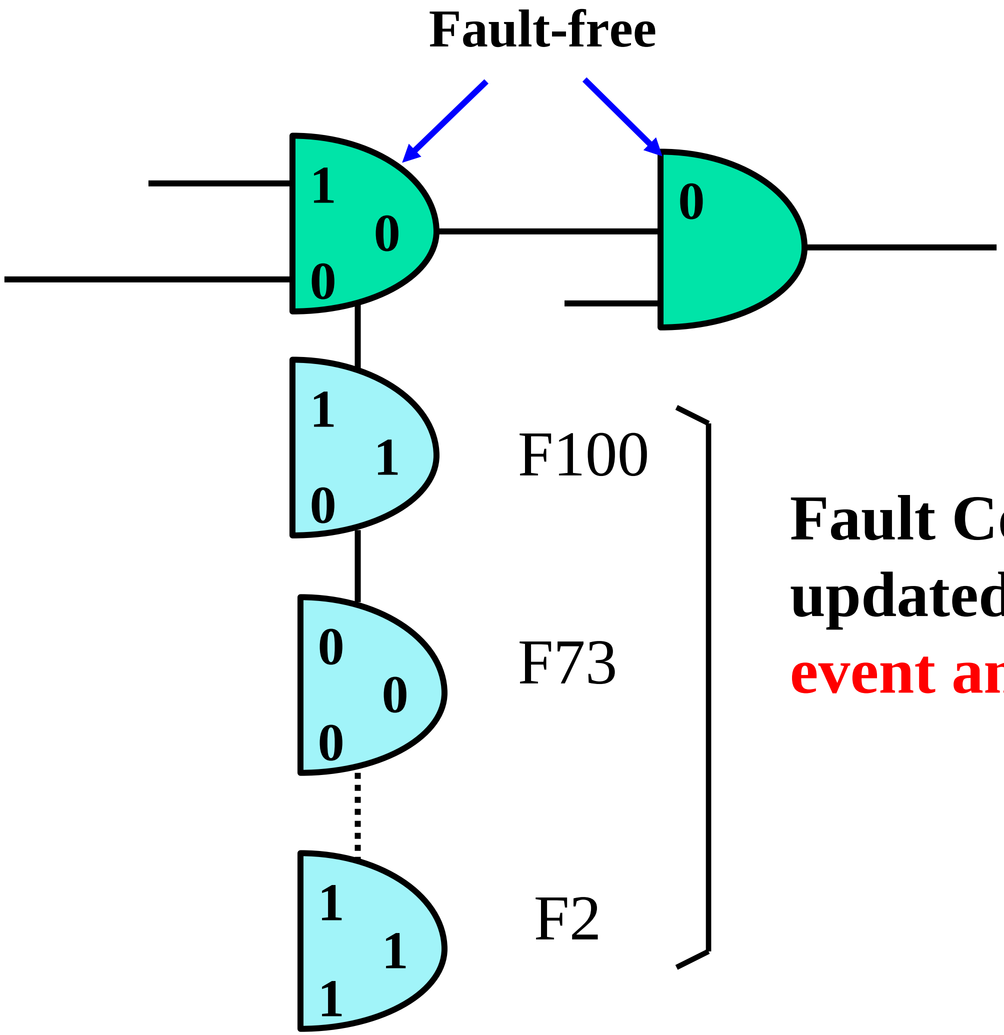
L_C=L_D = {B/1}, L_G = ϕ,

L_E = {B/1}, L_H = {B/1,F/0}, L_J = {B/1,F/0,J/0}

Concurrent Fault Simulation

- Simulate all faulty circuits in one pass:
  - Each gate retains a list of fault copies and each of them stores the status of a fault exhibiting difference from fault-free values
- Simulation mechanism
  - is similar to the conceptual fault simulation except that only the dynamical difference w.r.t. fault-free circuit is retained.
- Theoretically,
  - all faults in a circuit can be processed in one pass
- Practically,
  - memory explosion problem may restrict the number of faults that can be processed in each pass

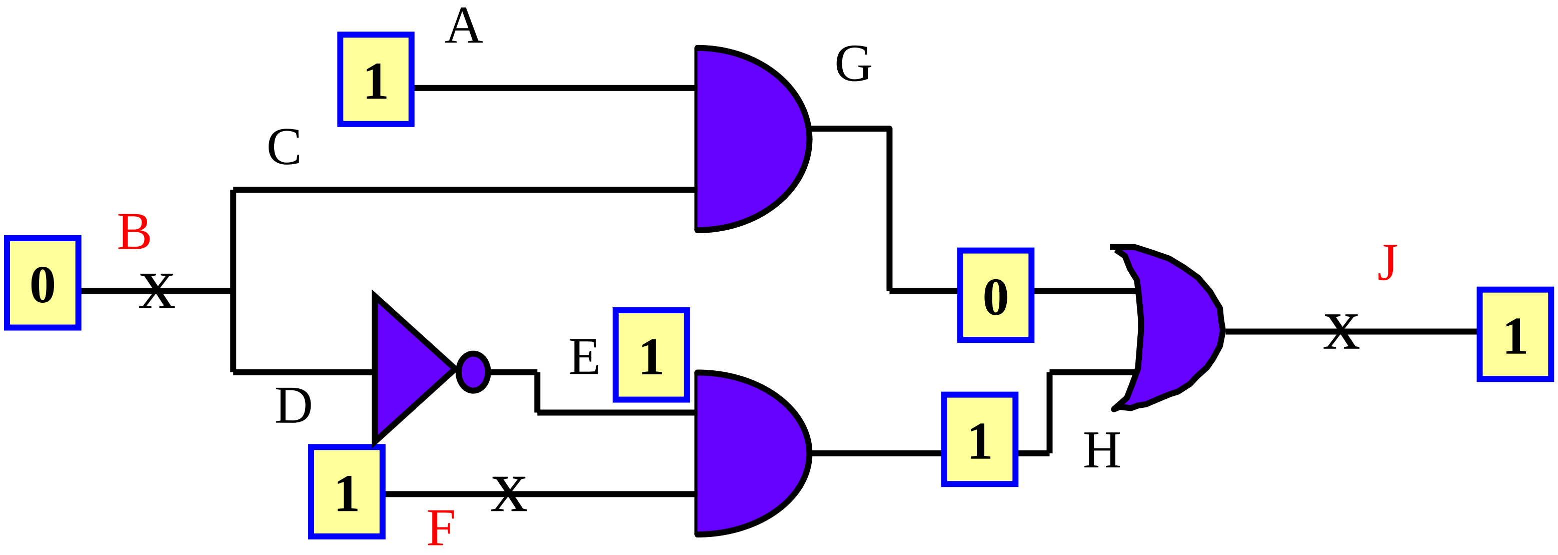
Concurrent Fault Simulation



Fault Copies:
updated by both logic event and fault events

Example: Concurrent Simulation (1)

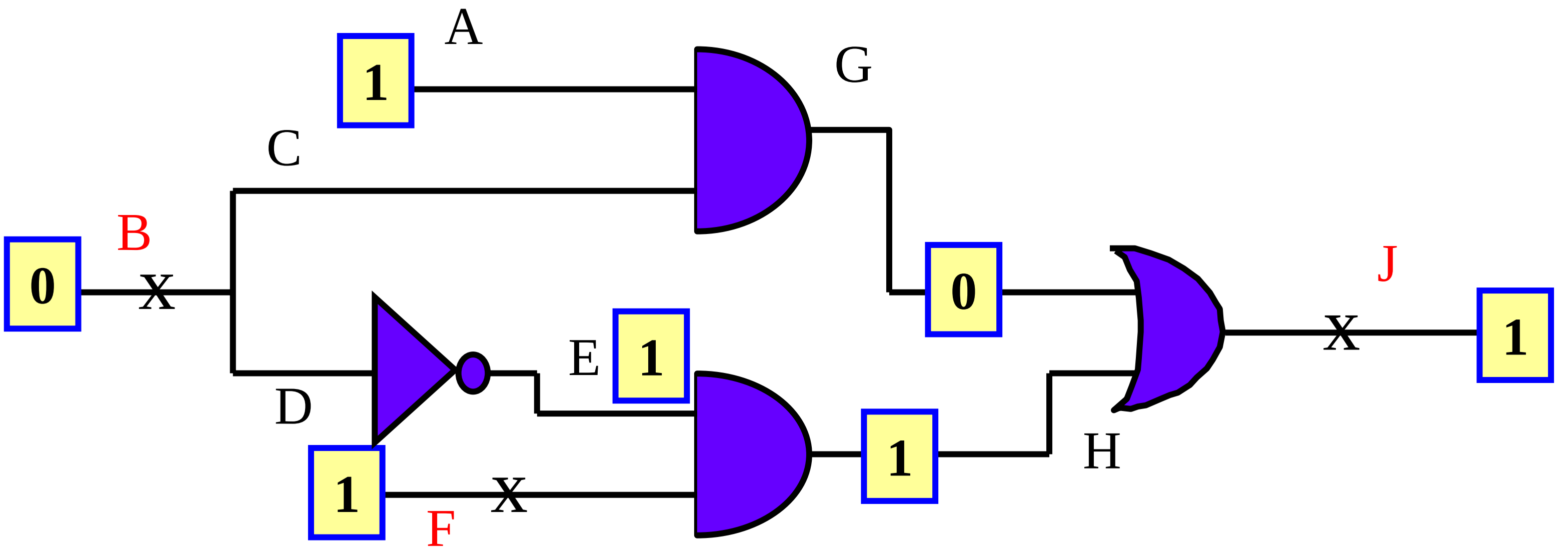
- Consider 3 faults: B/1, F/0, and J/0



L

Example: Concurrent Simulation (2)

- Consider 3 faults: B/1, F/0, and J/0

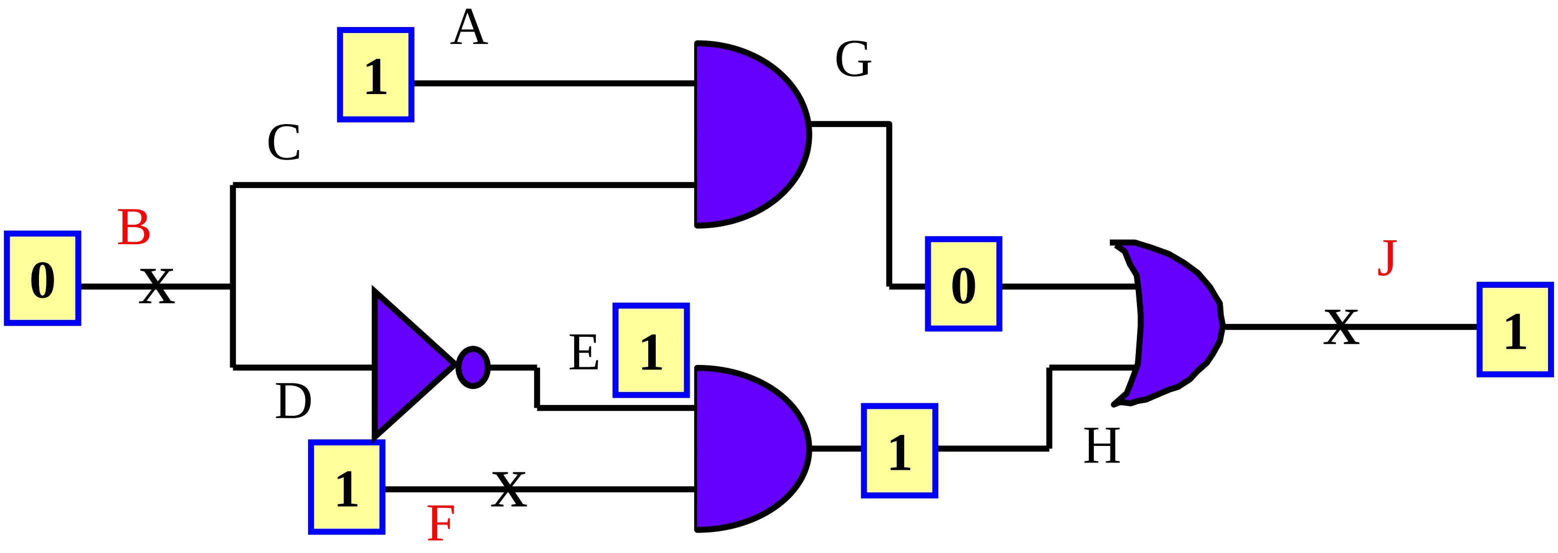


LG = {10_0, B/1:11_1}   LE = {0_1, B/1:1_0}

LH = {11_1, B/1:01_0, F/0:10_0}

Example: Concurrent Simulation (3)

- Consider 3 faults: B/1, F/0, and J/0



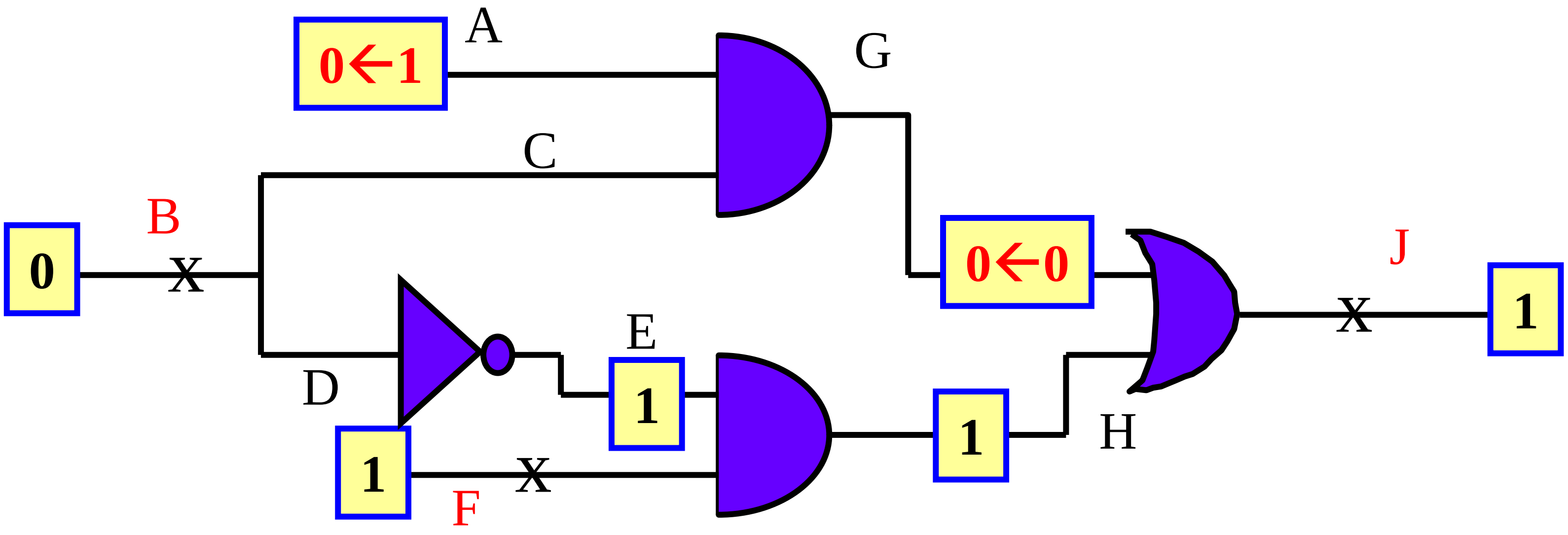
LG = {10_0, B/1:11_1}   LE = {0_1, B/1:1_0}

LH = {11_1, B/1:01_0, F/0:10_0}

LJ = {01_1, B/1:10_1, F/0:00_0, J/0:01_0}

Example: Concurrent Simulation (4)

- When A changes from 1 to 0

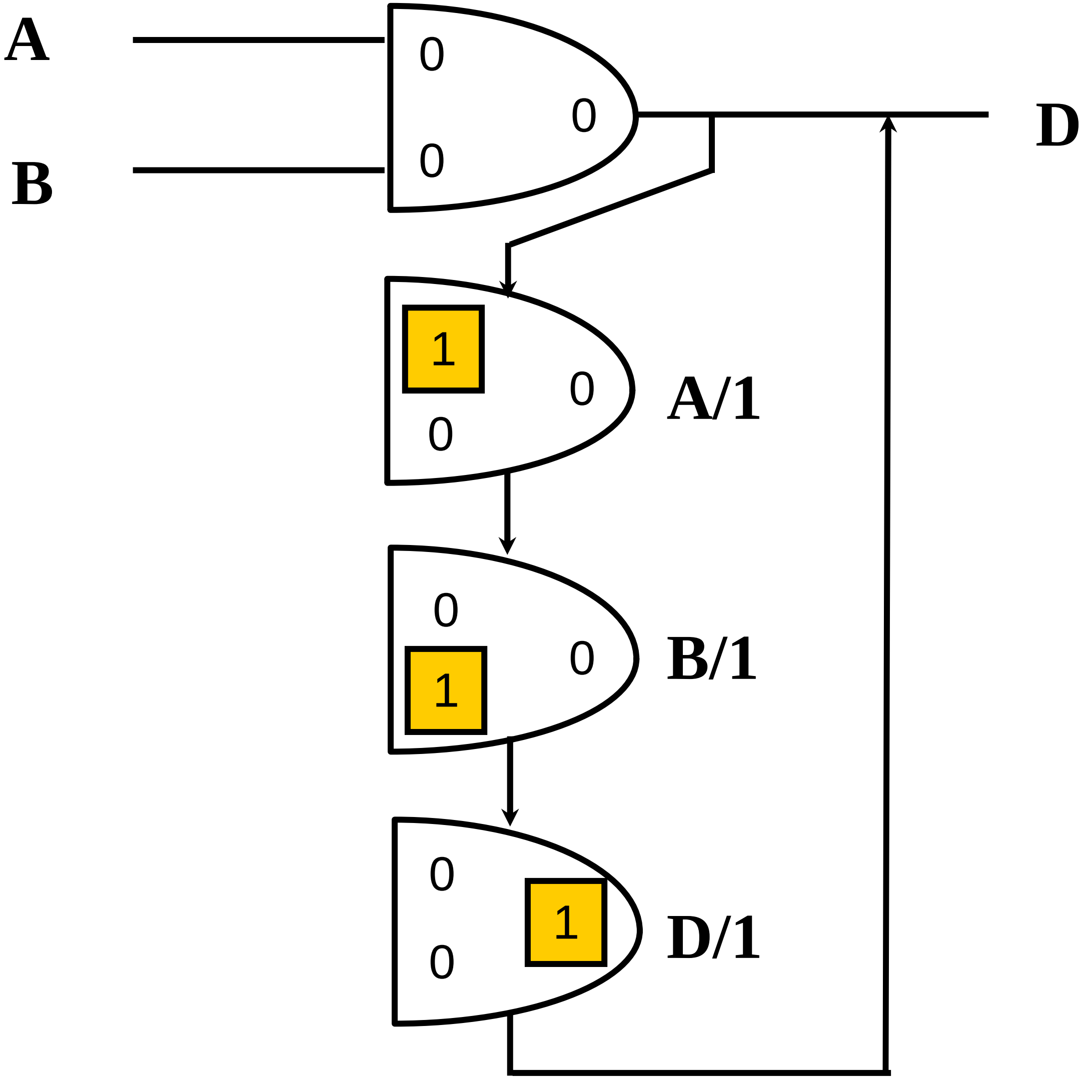


LG = {00_0, B/1:01_0} LE = {0_1, B/1:1_0}

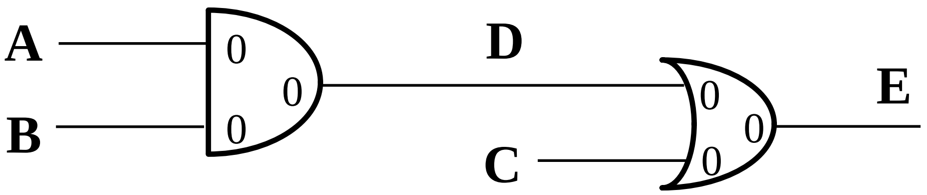
LH = {11_1, B/1:01_0, F/0:10_0}

LJ = {01_1, B/1:00_0, F/0:00_0, J/0:01_0}

Fault List of AND Gate



Fault List Propagation



These 2 faults are not propagated after evaluation →

A/1: 10_0

B/1: 01_0

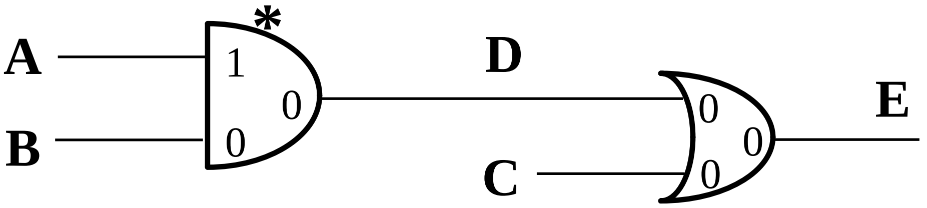
D/1: 00_1

propagated

C/1: 01_1

D/1: 10_1

E/1: 00_1



*A/0: 00_0

*B/1: 11_1

*D/1: 10_1

propagated

*B/1: 10_1

C/1: 01_1

D/1: 10_1

E/1: 00_1

Parallel-Pattern Single-Fault Simulation

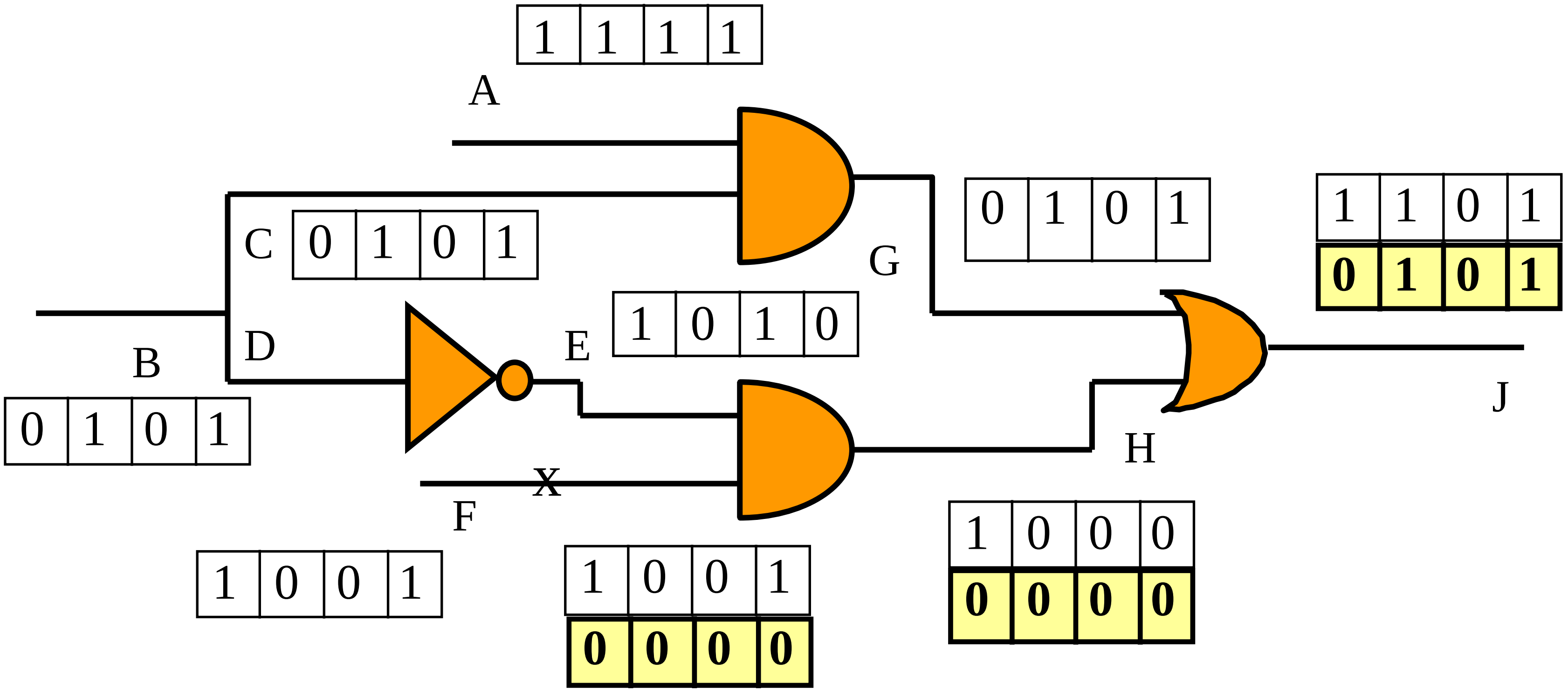
- Basic Idea:
  - Event Driven + Parallel Simulation
- Parallel-Pattern Simulation
  - Many patterns are simulated in parallel for both fault-free circuit and faulty circuits
  - The number of patterns is a multiple of the computer word-length
- Event-Driven
  - reduction of logic event simulation time
- Simple and Extremely Efficient
  - basis of most modern combinational fault simulators

Example: Parallel Pattern Simulation

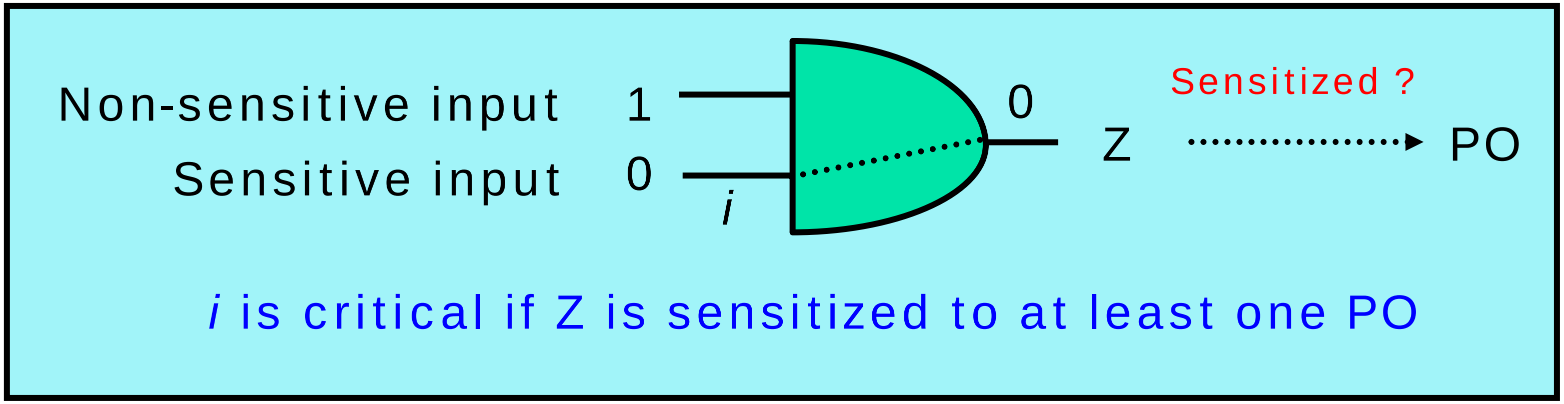
- Consider one fault F/0 and four patterns: P3,P2,P1,P0

Bit-Space:

P3	P2	P1	P0



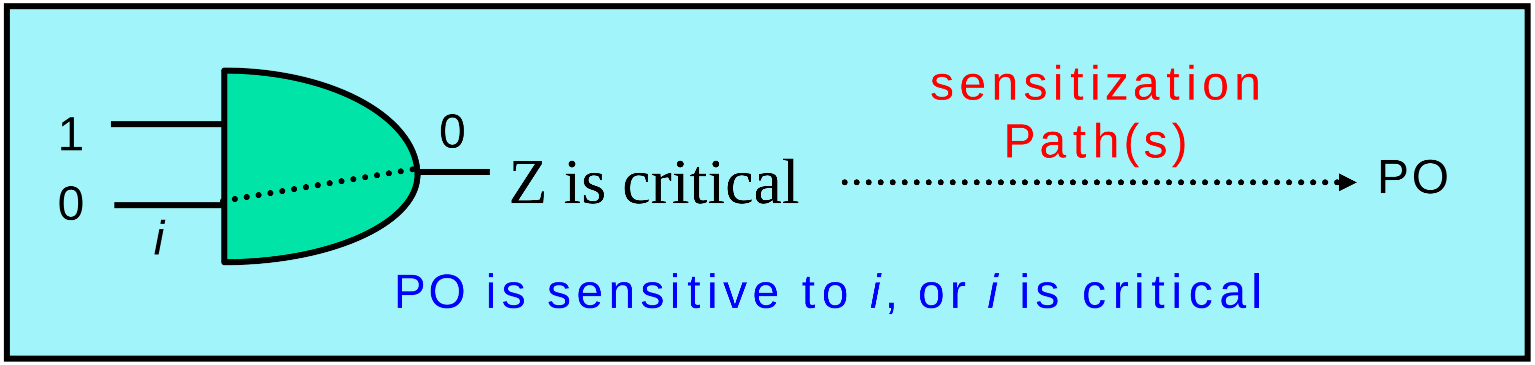
Sensitive Input and Critical Path



i is critical if Z is sensitized to at least one PO

- Sensitive Input of a gate:
  - A gate input *i* is *sensitive* if complementing the value of *i* changes the value of the gate output
- Critical line
  - Assume that the fault-free value of *w* is *v* in response to *t*
  - A line *w* is critical w.r.t. a pattern *t* iff *t* detects the fault *w* stuck-at *v*
- Critical paths
  - Paths consisting of critical lines only

Basics of Critical Path Tracing



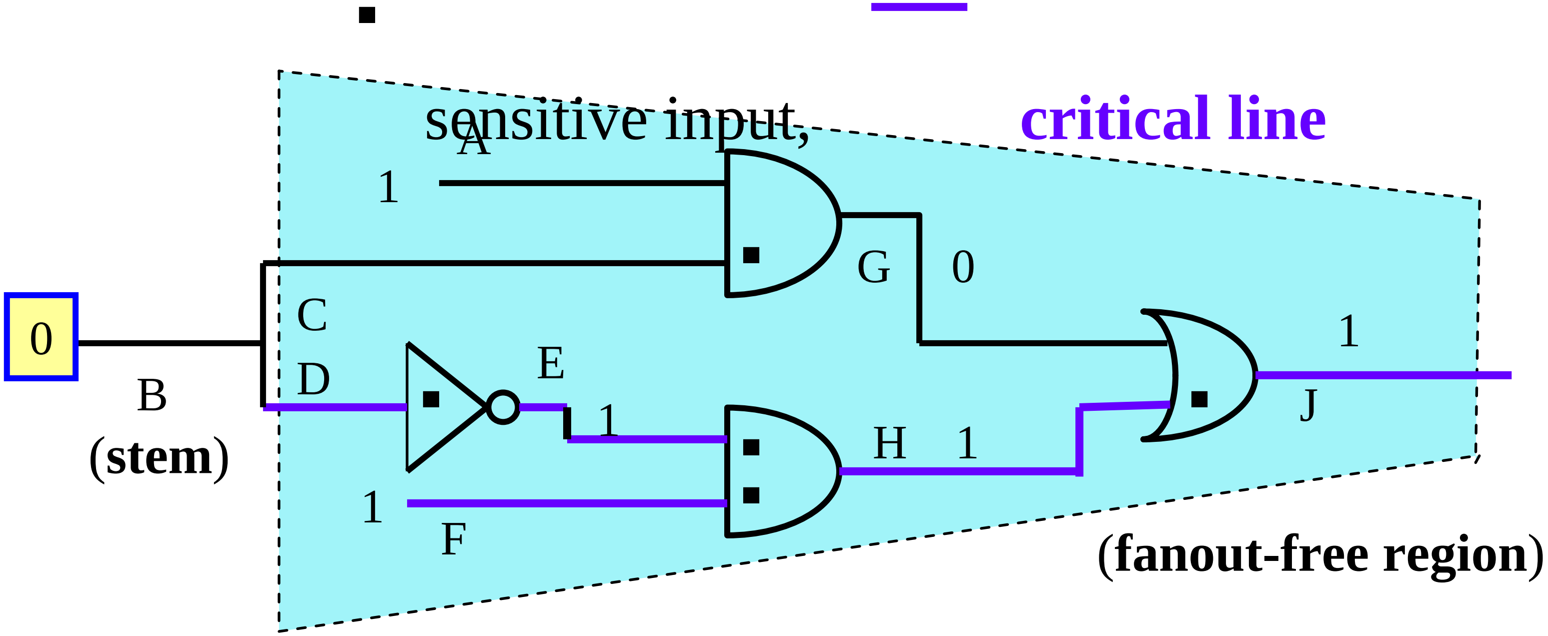
- A gate input *i* is critical w.r.t. a pattern *t* if
  - (1) the gate output is critical and
  - (2) *i* is a sensitive input to *t*
  - Use recursion to prove that *i* is also critical
- In a fanout-free circuit
  - the criticality of a line can be determined by backward traversing the sensitive gate inputs from PO's, in linear time

Analysis of Critical Path Tracing

- Three-step Procedure:
  - Step 1: Fault-free simulation
  - Step 2: Mark the sensitive inputs of each gate
  - Step 3: Identification of the critical lines by backward critical path tracing)
- Complexity is O(G)
  - Where G is the gate count
  - for fanout-free circuits --- very rare in practice
- Application
  - Applied to fanout-free regions, while stem faults are still simulated by parallel-pattern fault simulator.

Example of Critical Path Tracing

■ sensitive input, — critical line

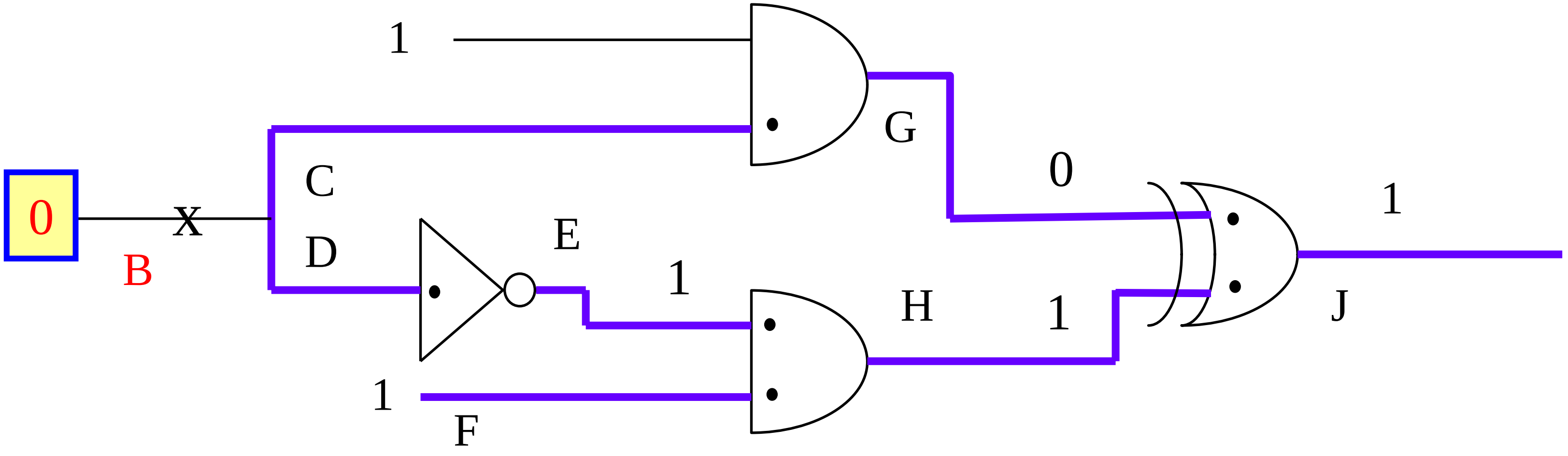


Detected faults in the fanout-free region:
{J/0, H/0, F/0, E/0, D/1}

Question: is B stuck-at-1 detected ?

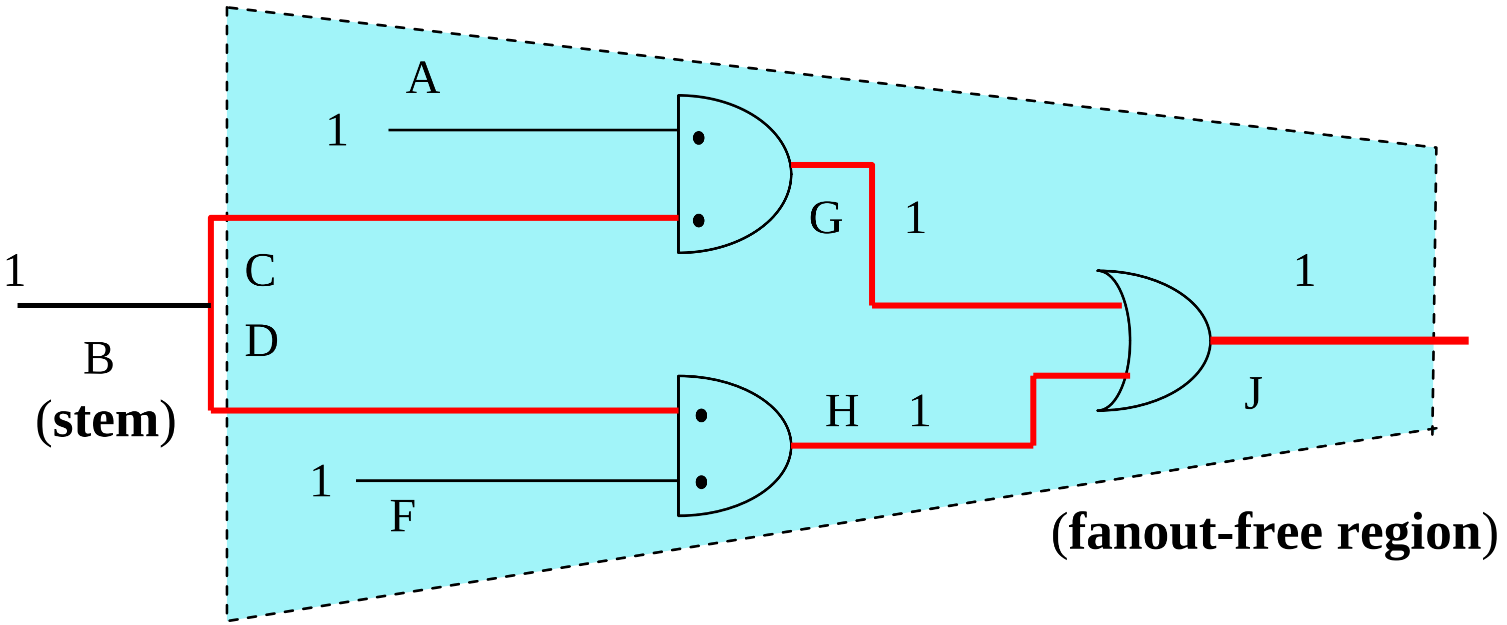
Anomaly of Critical Path Tracing

- **Stem criticality is hard to infer from branches. E.g. is B/1 detectable by the given pattern?**



- **It turns out that B/1 is not detectable even though both C and D are critical, because their effects cancel out each other at gate J, (i.e., fault masking problem)**
- **There is also a so-called multiple path sensitization problem.**

Multiple Path Sensitization



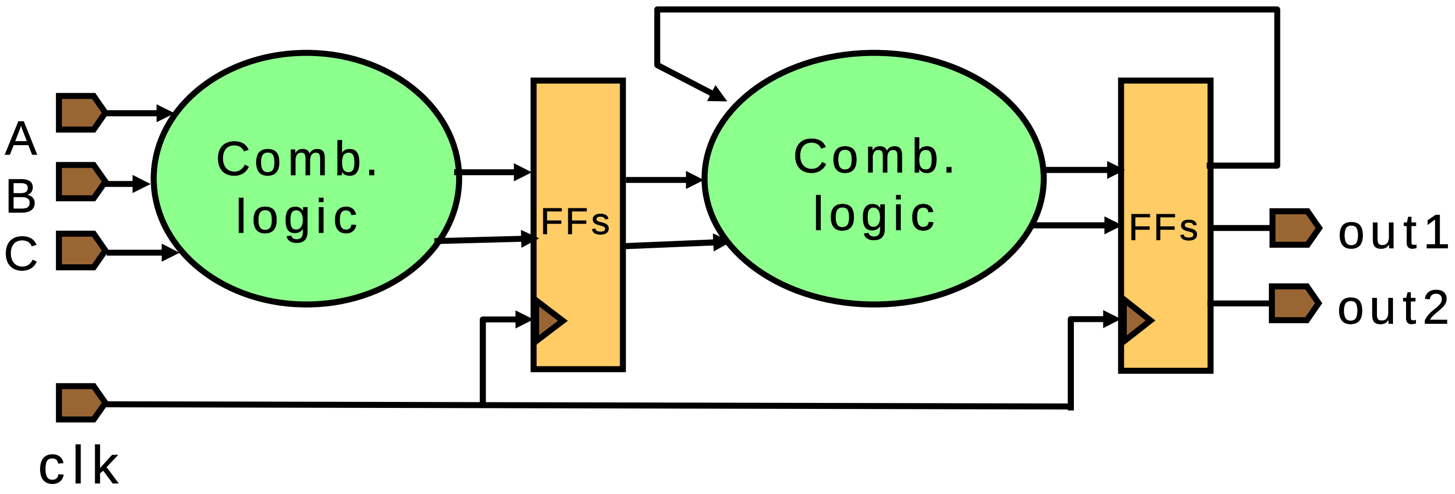
Both C and D are not critical, yet B is critical and B/0 can be detected at J by multiple path sensitization.

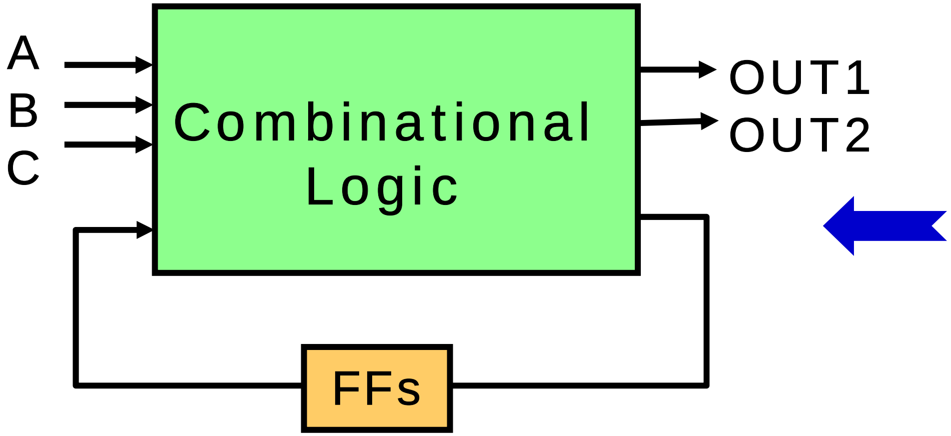
Sequential Fault Simulation

- Modern Techniques:
  - Fault simulation without restoration
  - Parallel fault simulation
  - Adoption of advanced combinational techniques
  - Management of hypertrophic faults
  - Other techniques:
    - parallel-sequence and parallel-pattern

Sequential Design Model

Sequential Circuits

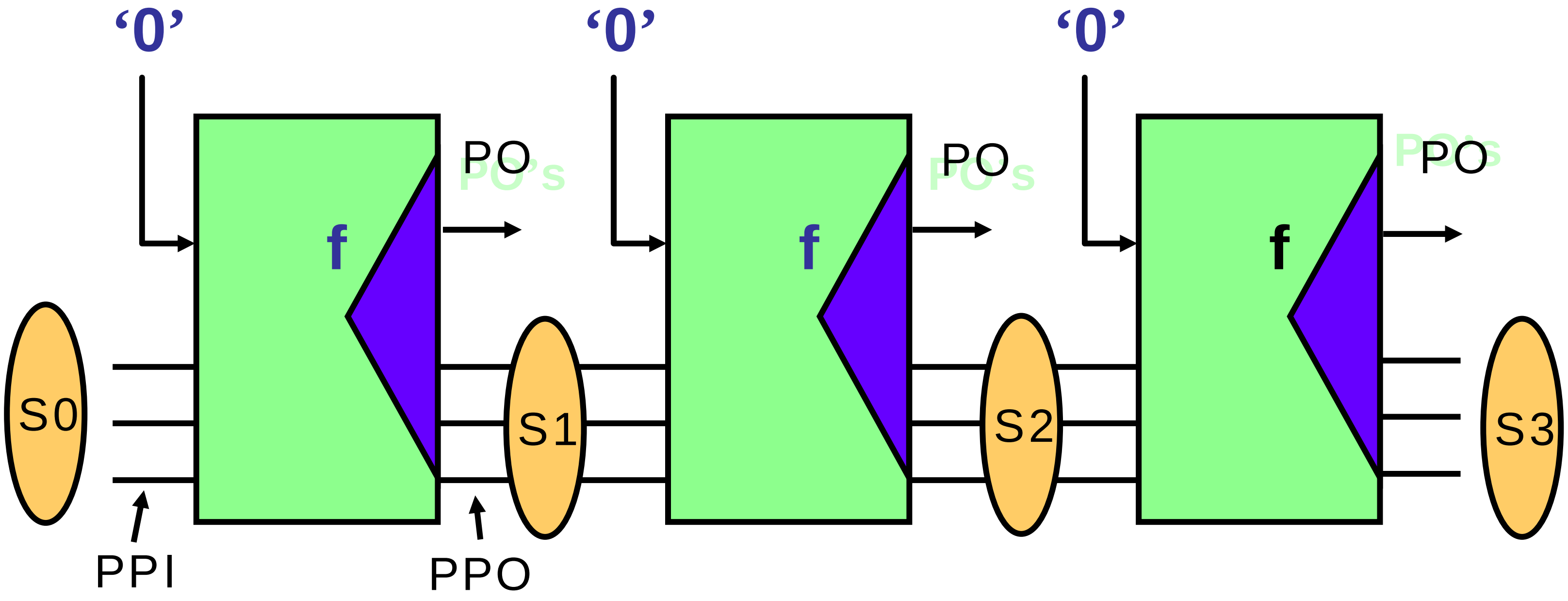




Hoffman Model

Time-Frame-Expansion Model

Ex: Input Sequence (‘0’, ‘0’, ‘0’)
State Sequence (S0 → S1 → S2 → S3)

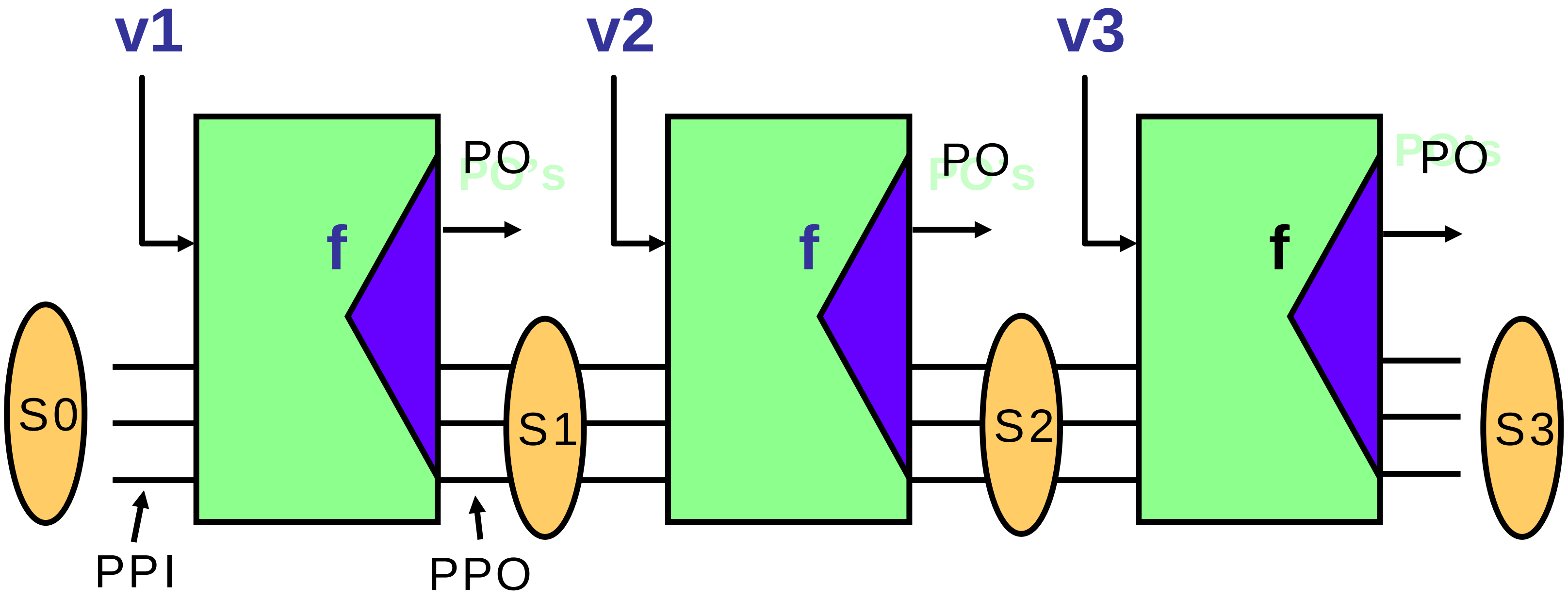


Notations: PPI: pseudo primary inputs (I.e., outputs of flip-flops)
PPO: pseudo primary outputs (I.e., inputs of flip-flops)

A single fault becomes multiple faults in the time-frame-expansion model

Parallel Pattern Simulation for Sequential Circuits?

Ex: Input Sequence (v1, v2, v3, …)
State Sequence (S0 → S1 → S2 → S3 → …)



If (next-state function) depends on only k previous input vectors
→Then parallel pattern simulation would take (k+1) passes to converge !

Outline

- Introduction
- Fault Modeling
- Fault Simulation
- Test Generation
- Design For Testability

Outline of ATPG (Automatic Test Pattern Generation)

- Test Generation (TG) Methods
  - Based on Truth Table
  - Based on Boolean Equation
  - Based on Structural Analysis
  - D-algorithm [Roth 1967]
  - 9-Valued D-algorithm [Cha 1978]
  - PODEM [Goel 1981]
  - FAN [Fujiwara 1983]

General ATPG Flow

- ATPG (Automatic Test Pattern Generation)
  - Generate a set of vectors for a set of target faults
- Basic flow

```
Initialize the vector set to NULL
Repeat
    Generate a new test vector
    Evaluate fault coverage for the test vector
    If the test vector is acceptable, then add it to the vector set
Until required fault coverage is obtained
```

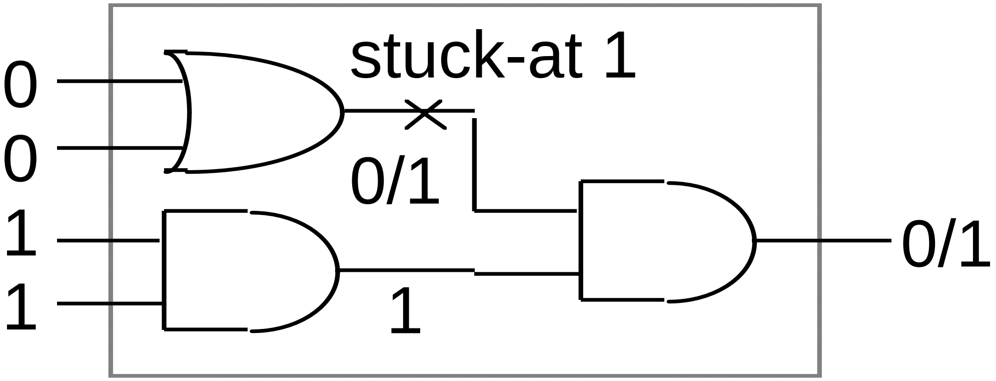
- To accelerate the ATPG
  - Random patterns are often generated first to detect easy-to-detect faults, then a deterministic TG is performed to generate tests for the remaining faults

Combinational ATPG

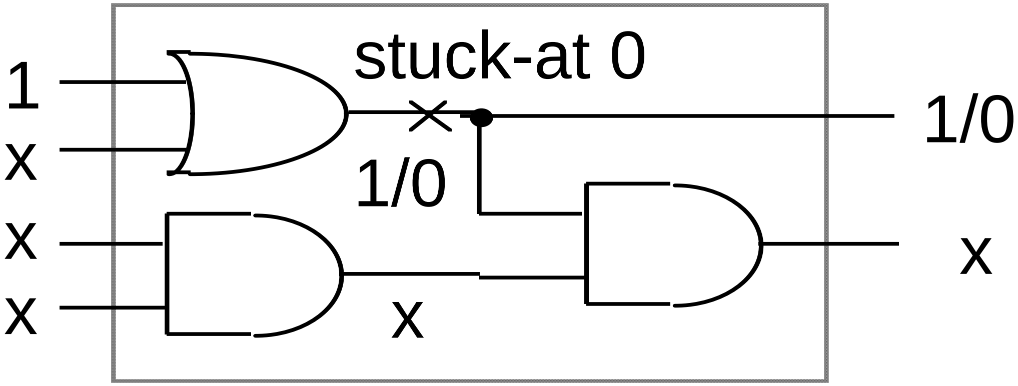
- Test Generation (TG) Methods
  - Based on Truth Table
  - Based on Boolean Equation
  - Based on Structural Analysis
- Milestone Structural ATPG Algorithms
  - D-algorithm [Roth 1967]
  - 9-Valued D-algorithm [Cha 1978]
  - PODEM [Goel 1981]
  - FAN [Fujiwara 1983]

A Test Pattern

A Fully Specified Test Pattern
(every PI is either 0 or 1)



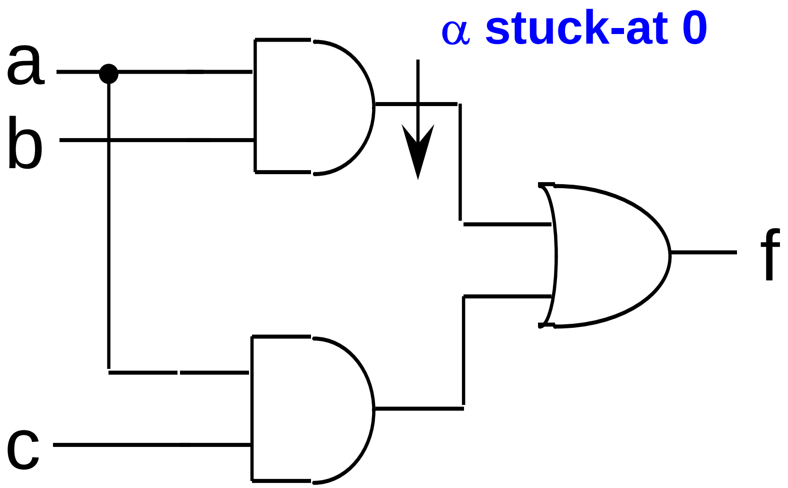
A Partially Specified Test Pattern
(certain PI's could be undefined)



Test Generation Methods

(From Truth Table)

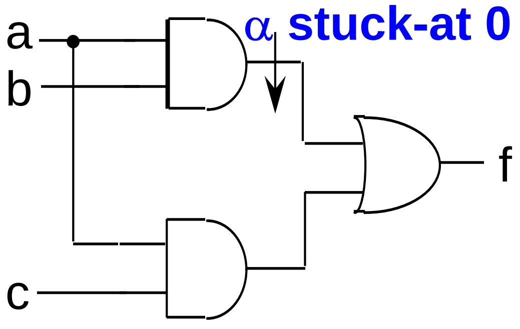
Ex: How to generate tests for the stuck-at 0 fault (fault α)?



abc	f	fα
000	0	0
001	0	0
010	0	0
011	0	0
100	0	0
101	1	1
110	1	0
111	1	1

Test Generation Methods

(Using Boolean Equation)



$f = ab+ac$, $f_\alpha = ac$

T_α = the set of all tests for fault α

$= \text{ON\_set}(f \oplus f_\alpha)$

$= \text{ON\_set}(f) * \text{OFF\_set}(f_\alpha) + \text{OFF\_set}(f) * \text{ON\_set}(f_\alpha)$

$= \{(a,b,c) \mid (ab+ac)(ac)' + (ab+ac)'(ac) = 1\}$ ← **Boolean equation**

$= \{(a,b,c) \mid abc'=1\}$

$= \{ (110) \}$.

High complexity !!
Since it needs to compute the faulty function for each fault.

* **ON_set(f): All input combinations to which f evaluates to 1.**
OFF_set(f): All input combinations to which f evaluates to 0.
Note: a function is characterized by its ON_SET

Boolean Difference

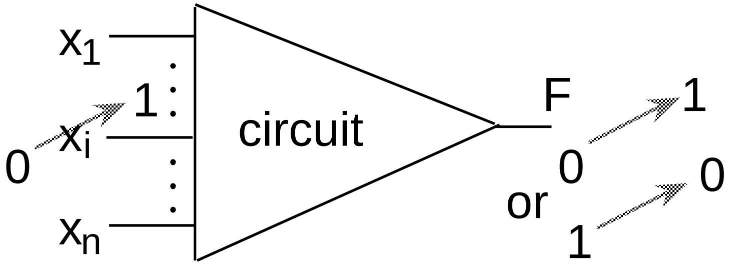
- Physical Meaning of Boolean Difference
  - For a logic function F(X)=F(x1, ..., xi, ..., xn), find all the input combinations that make a value-change at xi also cause a value-change at F.
- Logic Operation of Boolean Difference
  - The Boolean difference of F(X) w.r.t. input xi is

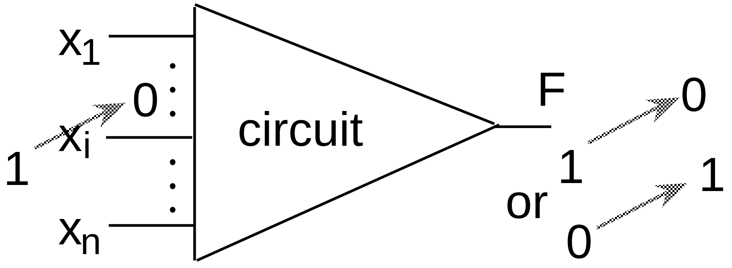
$$dF(x)/dx_i = F_i(0) \oplus F_i(1) = F_i(0) \cdot F_i(1)' + F_i(0)' \cdot F_i(1)$$

Where

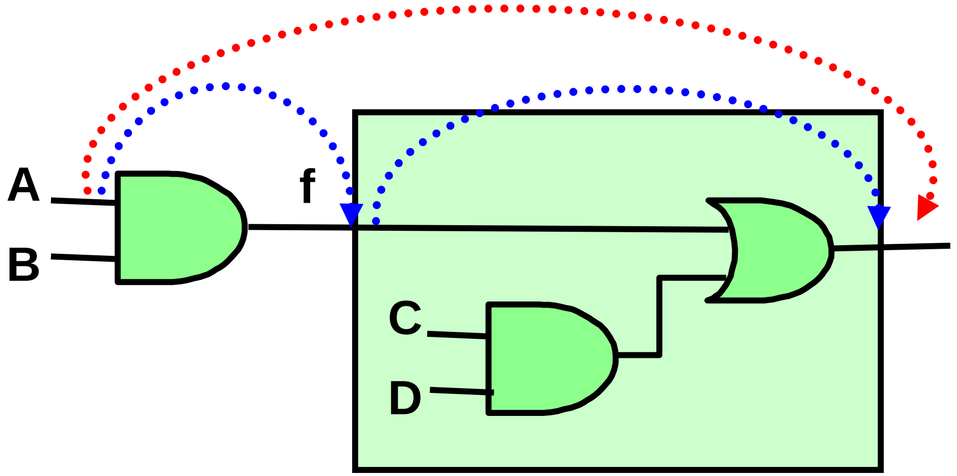
$$F_i(0) = F(x_1, \ldots, 0, \ldots, x_n)$$
$$F_i(1) = F(x_1, \ldots, 1, \ldots, x_n)$$

- Illustrations of Boolean Difference





Chain Rule



G(f(A, B), {C, D})

{A,B} and {C,D} have no variables in common

f = AB

G = f + CD

➡

dG/df = (C’ + D’)

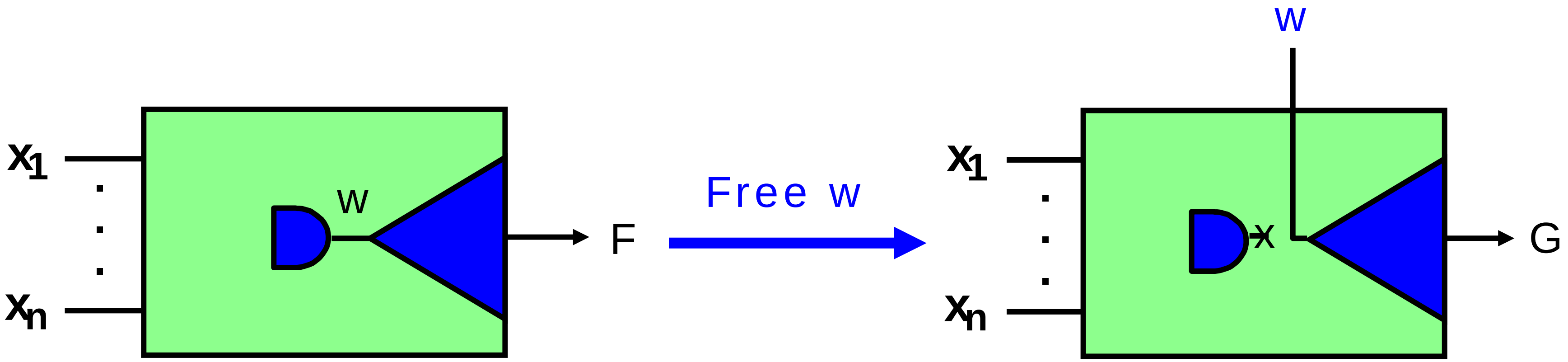
df/dA = B

➡ dG/dA = (dG/df) · (df/dA) = (C’+D’) · B

An Input vector v sensitizes a fault effect from **A to G**
Iff v sensitizes the effect from **A to f** and from **f to G**

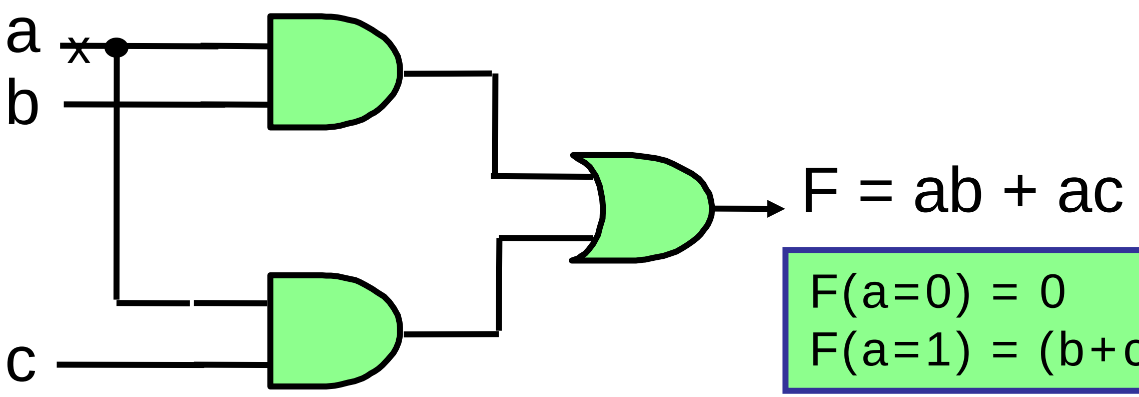
Boolean Difference (con't)

- Boolean Difference
  - With respect to an internal signal, w, Boolean difference represents the set of input combinations that sensitize a fault effect from w to the primary output F
- Calculation
  - Step 1: convert the function F into a new one G that takes the signal w as an extra primary input
  - Step 2: $dF(x_1, \ldots, x_n)/dw = dG\ (x_1, \ldots, x_n, w)/dw$



Test Gen. By Boolean Difference

Case 1: Faults are present at PIs.



$F(a=0) = 0$
$F(a=1) = (b+c)$

Fault Sensitization Requirement:
$dF/da = F(a=0) \oplus F(a=1) = 0 \oplus (b+c) = (b+c)$

Test-set for *a* s-a-1 = {(a,b,c) | a'• (b+c)=1} = {(01x), (0x1)}.
Test-set for *a* s-a-0 = {(a,b,c) | $\underline{a}$ • $\underline{(b+c)}$=1} = {(11x), (1x1)}.

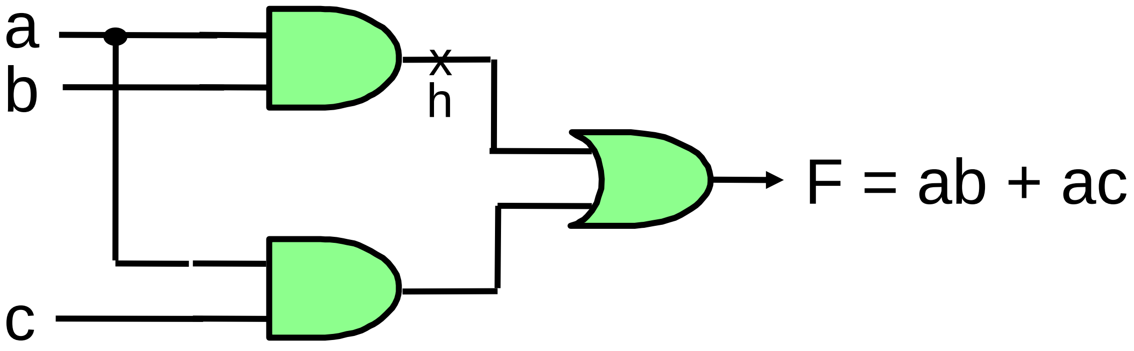
Fault activation requirement

Fault sensitization requirement

No need to compute
The faulty function !!

Test Generation By Boolean Difference (con’t)

Case 2: Faults are present at internal lines.



G(i.e., F with h floating) = h + ac
$dG/dh = G(h=0) \oplus G(h=1) = (ac \oplus 1) = (a'+c')$

Test-set for *h* s-a-1 is

$$\{ (a,b,c) |\ h' \bullet (a'+c')=1 \} = \{ (a,b,c) |\ (a'+b') \bullet (a'+c')=1 \} = \{ (0xx), (x00) \}.$$

Test-set for *h* s-a-0 is

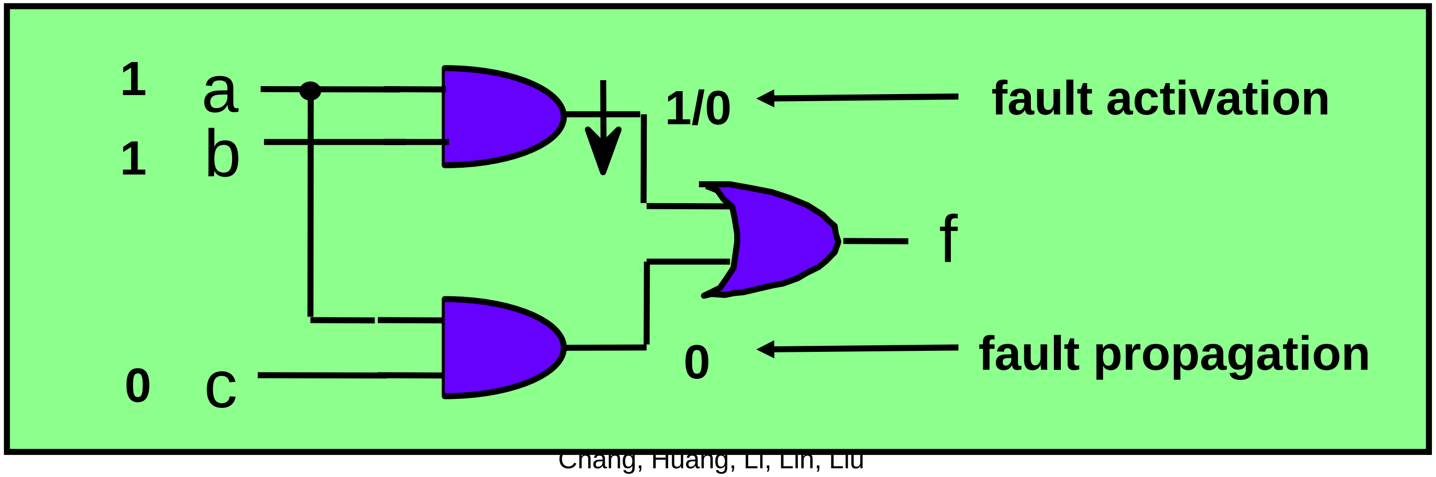
$$\{(a,b,c) |\ \underbrace{h}_{\text{For fault activation}} \bullet \underbrace{(a'+c')}_{\text{For fault sensitization}}=1\} = \{(110)\}.$$

Outline of ATPG
(Automatic Test Pattern Generation)

- Test Generation (TG) Methods
  - Based on Truth Table
  - Based on Boolean Equation
  - Based on Structural Analysis
  - D-algorithm [Roth 1967]
  - 9-Valued D-algorithm [Cha 1978]
  - PODEM [Goel 1981]
  - FAN [Fujiwara 1983]

Test Generation Method (From Circuit Structure)

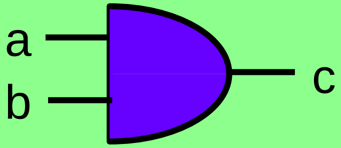
- Two basic goals
  - (1) Fault activation (FA)
  - (2) Fault propagation (FP)
  - Both of which requires Line Justification (LJ), I.e., finding input combinations that force certain signals to their desired values
- Notations:
  - 1/0 is denoted as D, meaning that good-value is 1 while faulty value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



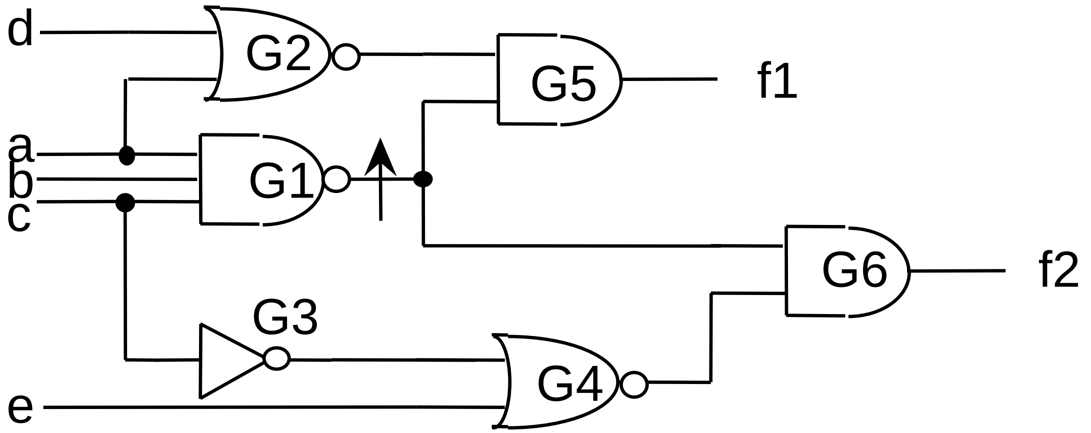
Common Concepts for Structural TG

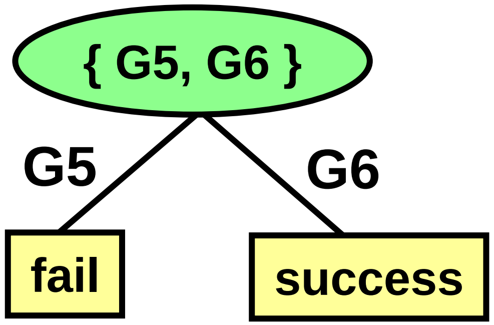
- Fault activation
  - Setting the faulty signal to either 0 or 1 is a Line Justification problem
- Fault propagation
  - (1) select a path to a PO → decisions
  - (2) Once the path is selected → a set of line justification (LJ) problems are to be solved
- Line Justification
  - Involves decisions or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



Ex: Decision on Fault Propagation





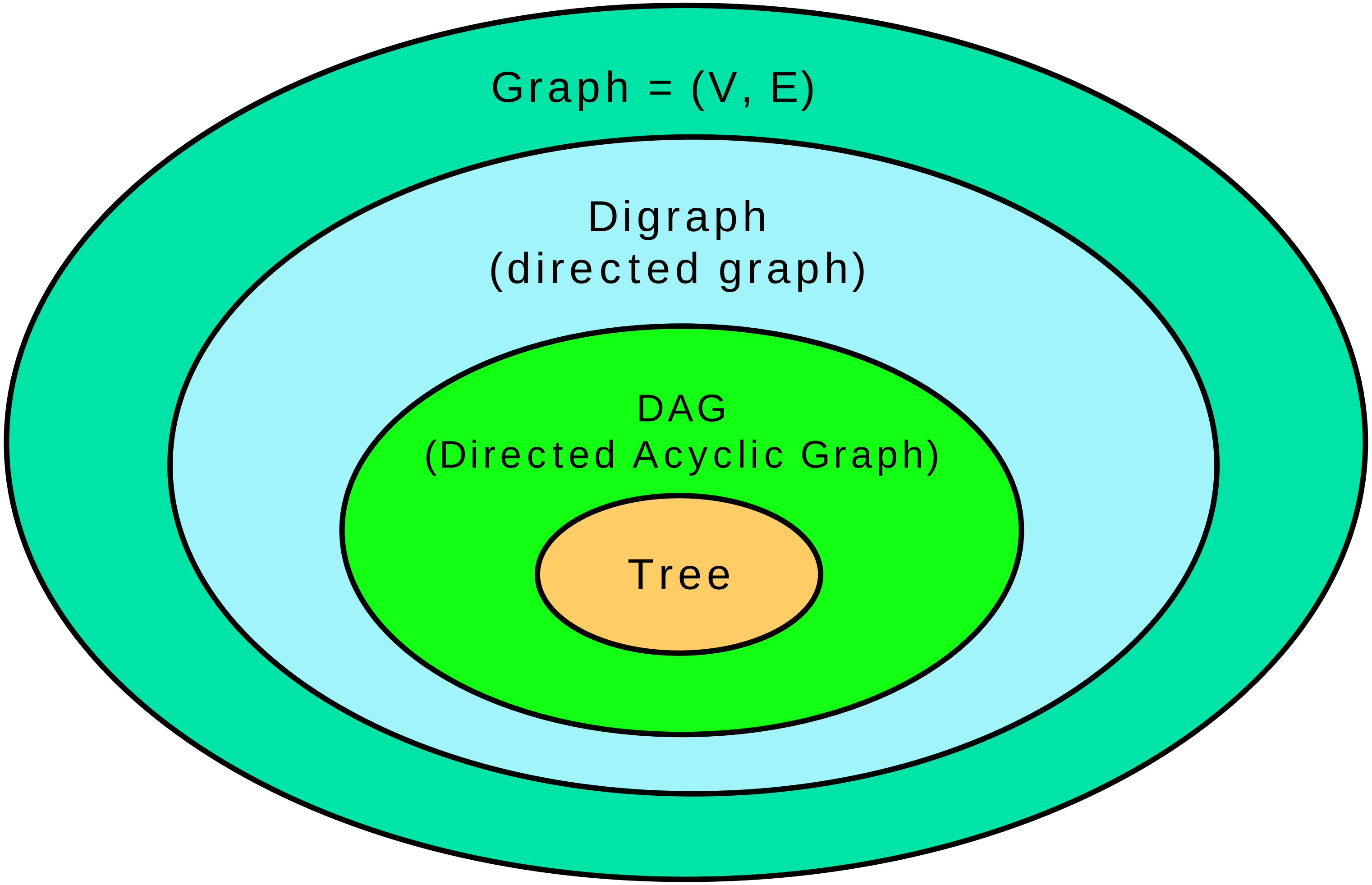
decision tree

- 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)

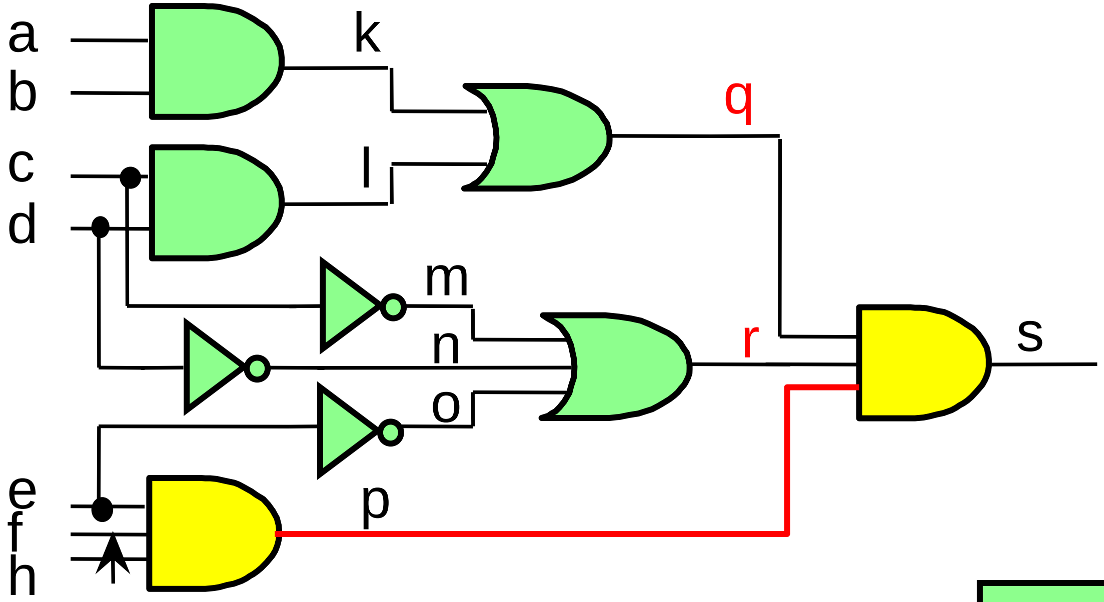
D-frontiers: are the gates whose output value is x, while one or more Inputs are D or D'. For example, initially, the D-frontier is { G5, G6 }.

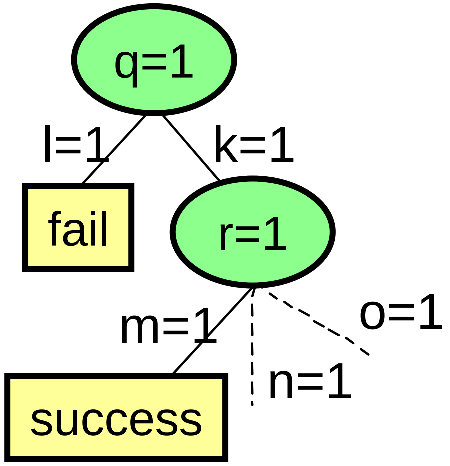
Various Graphs

A Combinational Circuit: is usually modeled as a DAG, but not tree



Ex: Decisions On Line Justification





The 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 point

- 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)

Branch-and-Bound Search

- Test Generation
  - Is a branch-and-bound search
  - Every decision point is a branching point
  - If a set of decisions lead to a conflict (or bound), a backtrack is taken to explore other decisions
  - A test is found when
    - (1) fault effect is propagated to a PO
    - (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.

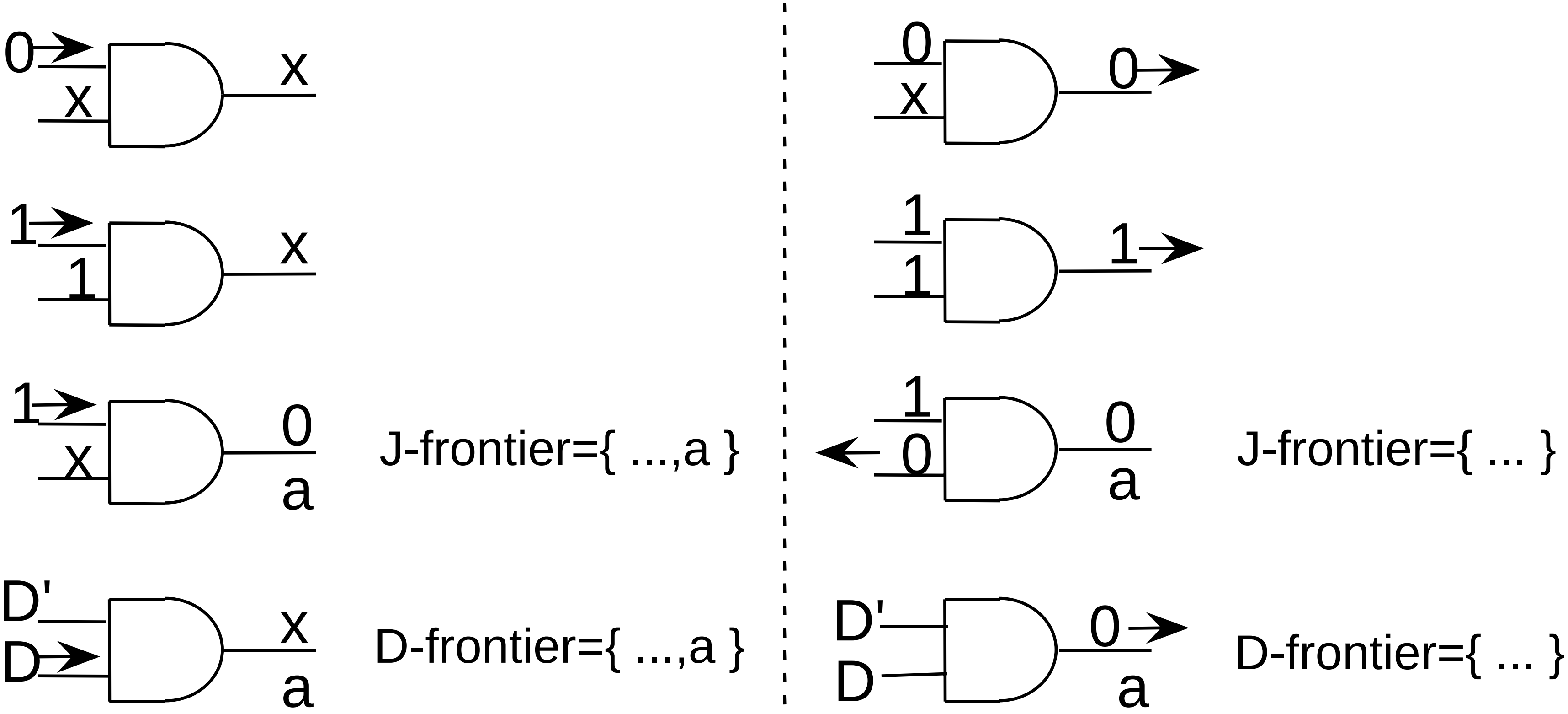
Implications

- Implications
  - Computation of 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

Local Implications (Forward)

Before

After



J-frontier={ ...,a }

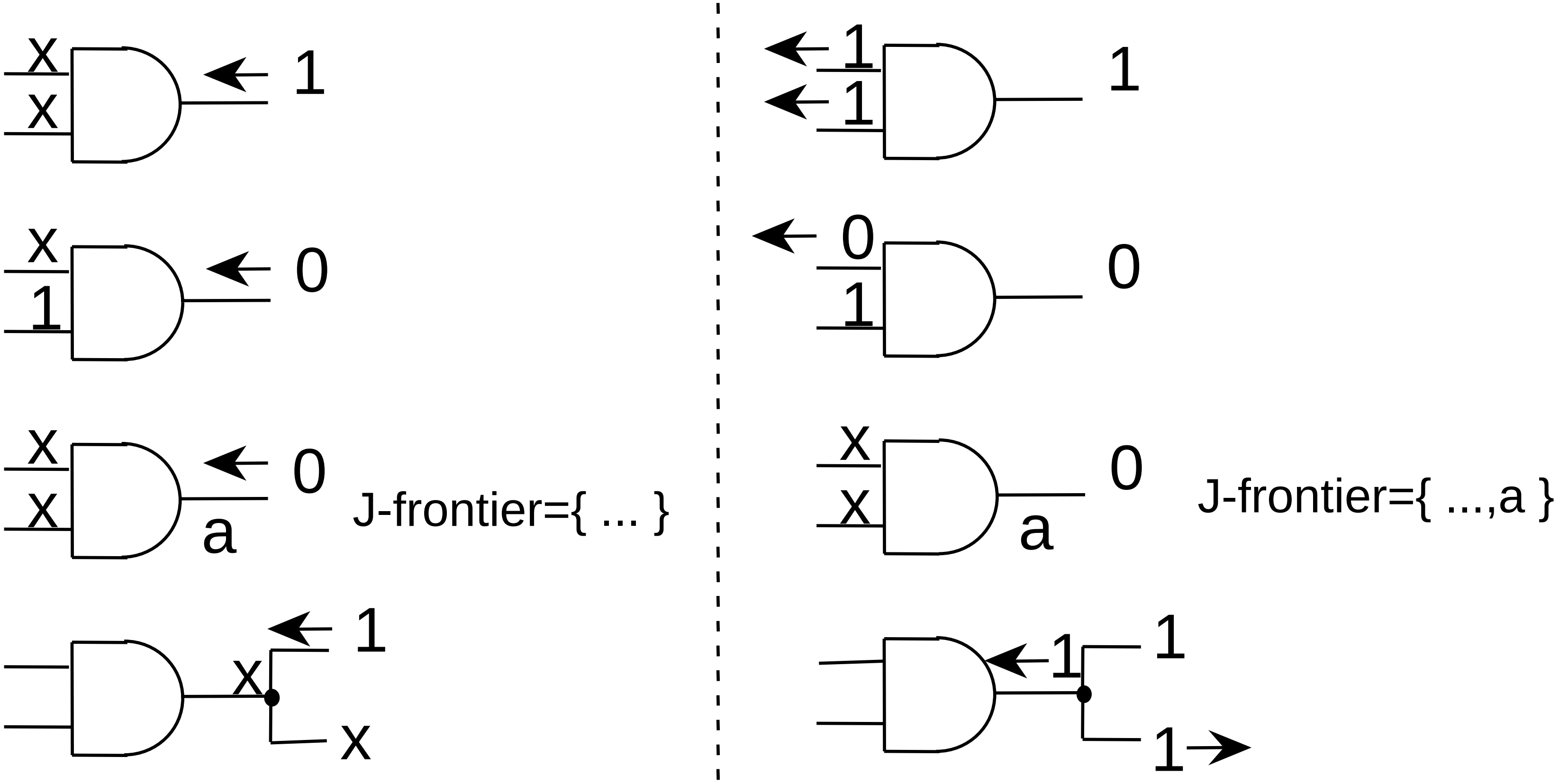
J-frontier={ ... }

D-frontier={ ...,a }

D-frontier={ ... }

Local Implications (Backward)

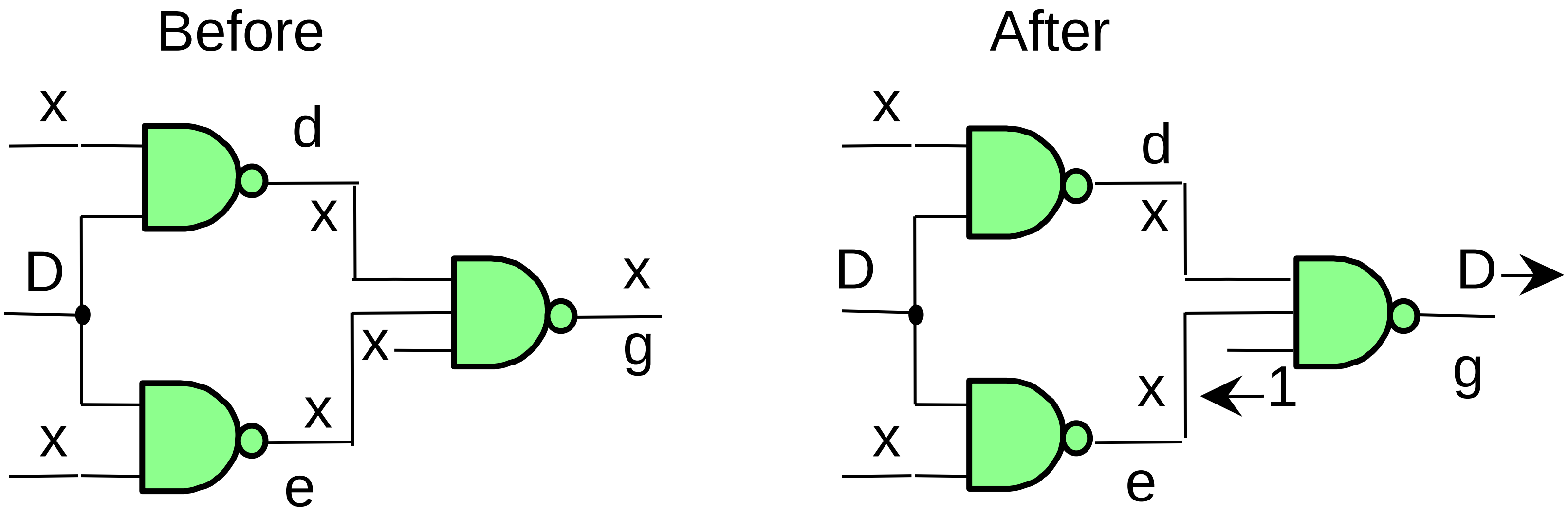
Before After



J-frontier={ ... }

J-frontier={ ...,a }

Global Implications



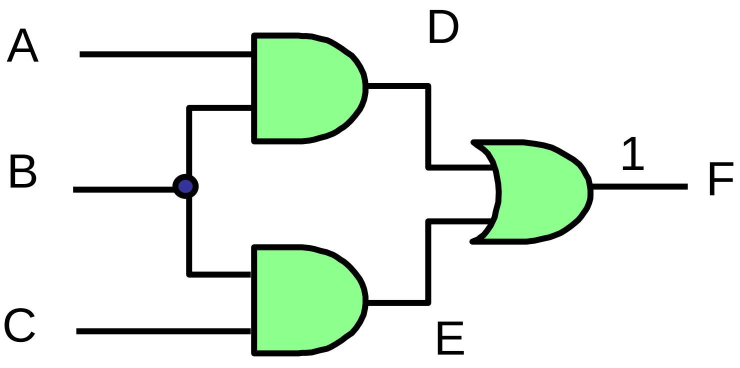
- Unique D-Drive Implication
  - Suppose D-frontier (or D-drive) is {d, e}, → g is a dominator for both d and e, hence a unique D-drive is at g

g is called a dominator of d:
because every path from d to an PO passes through g

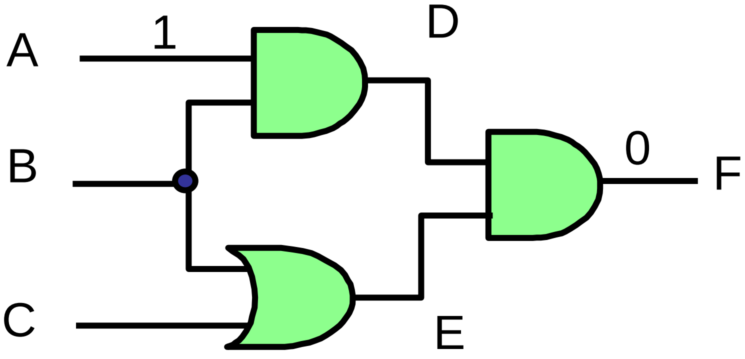
Learning for Global Implication

- Static Learning
  - Global implication derived by contraposition law
  - Learn static (I.e., input independent) signal implications
- Dynamic Learning
  - Contraposition law + other signal values
  - Is input pattern dependent

A→B => ~B → ~A



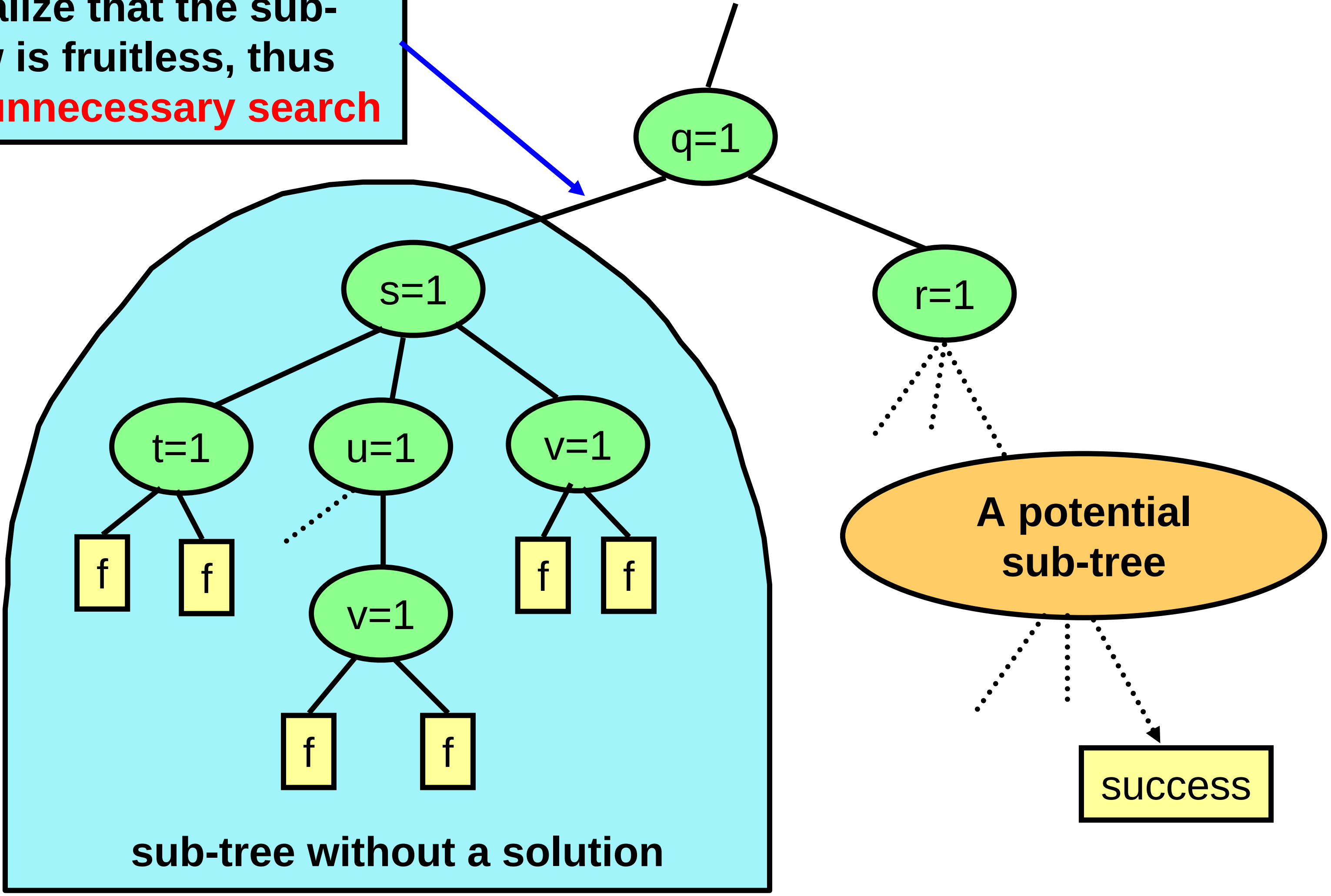
F=1 implies B=1
Because B=0 → F=0
(Static Learning)



F=0 implies B=0 When A=1
Because {B=1, A=1} → F=1
(Dynamic Learning)

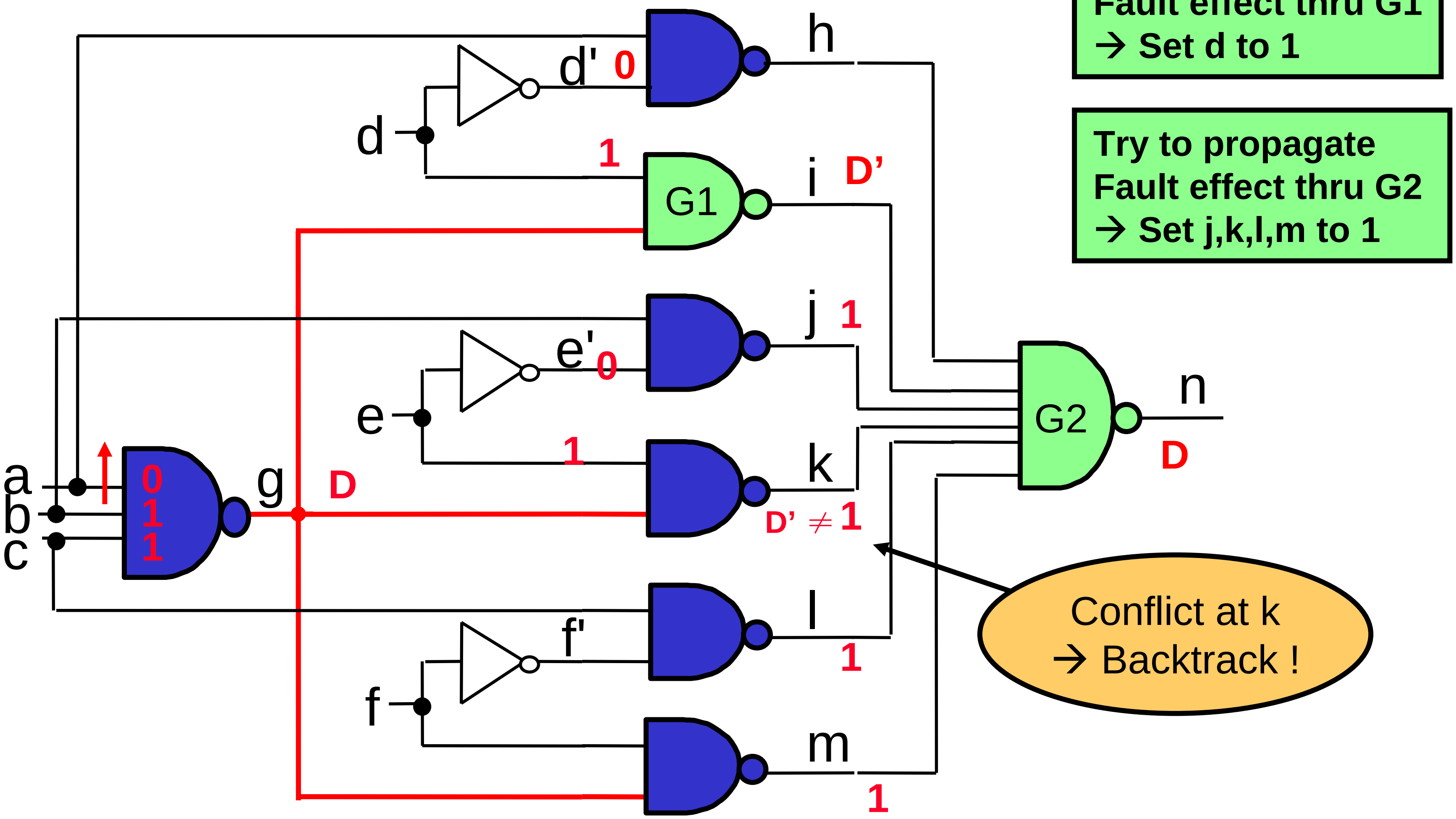
Early Detection of Inconsistency

Aggressive implication may help to realize that the sub-tree below is fruitless, thus avoiding unnecessary search



Ex: D-Algorithm (1/3)

- Five logic values
  - { 0, 1, x, D, D’ }



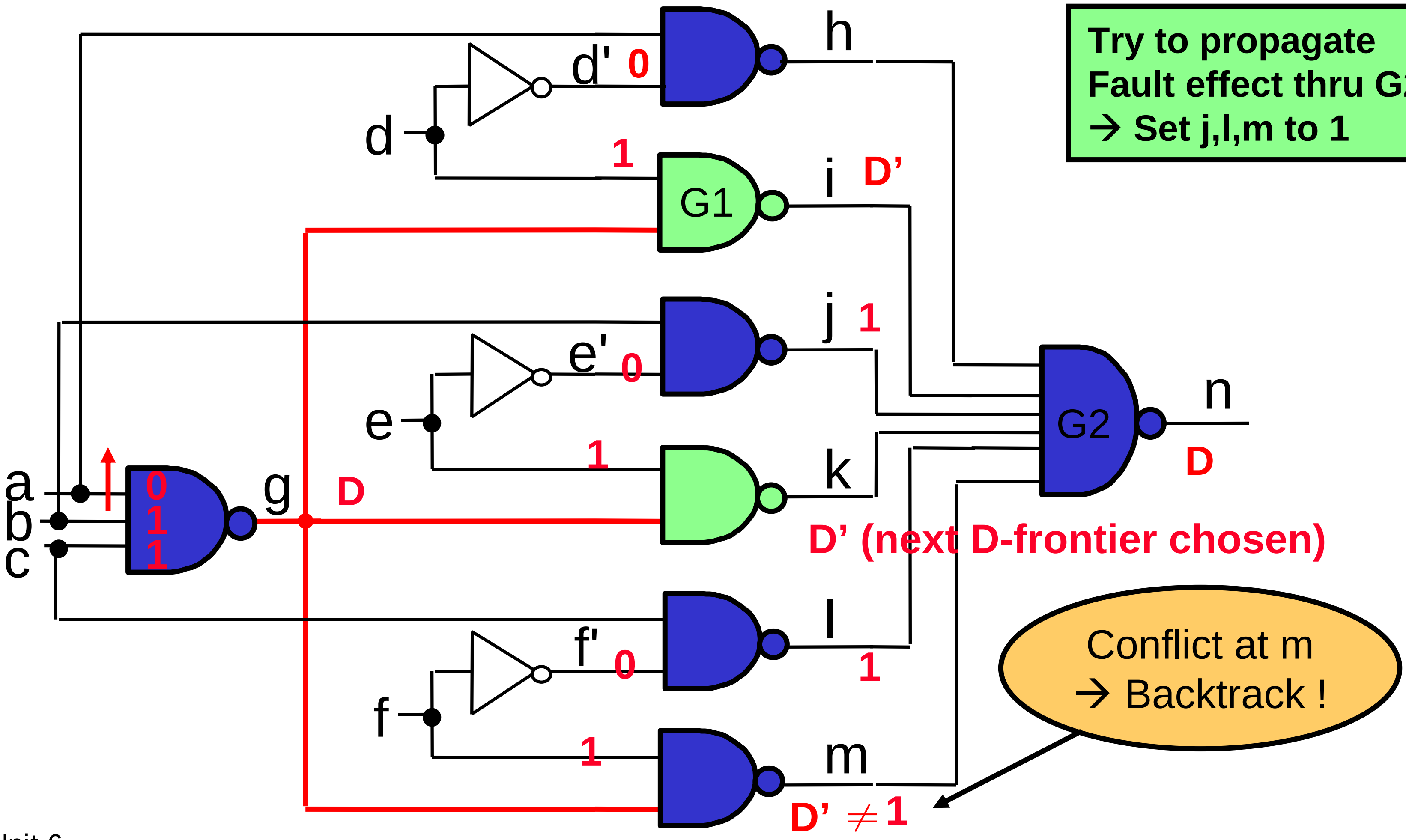
Try to propagate Fault effect thru G1 → Set d to 1

Try to propagate Fault effect thru G2 → Set j,k,l,m to 1

Conflict at k → Backtrack !

Ex: D-Algorithm (2/3)

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



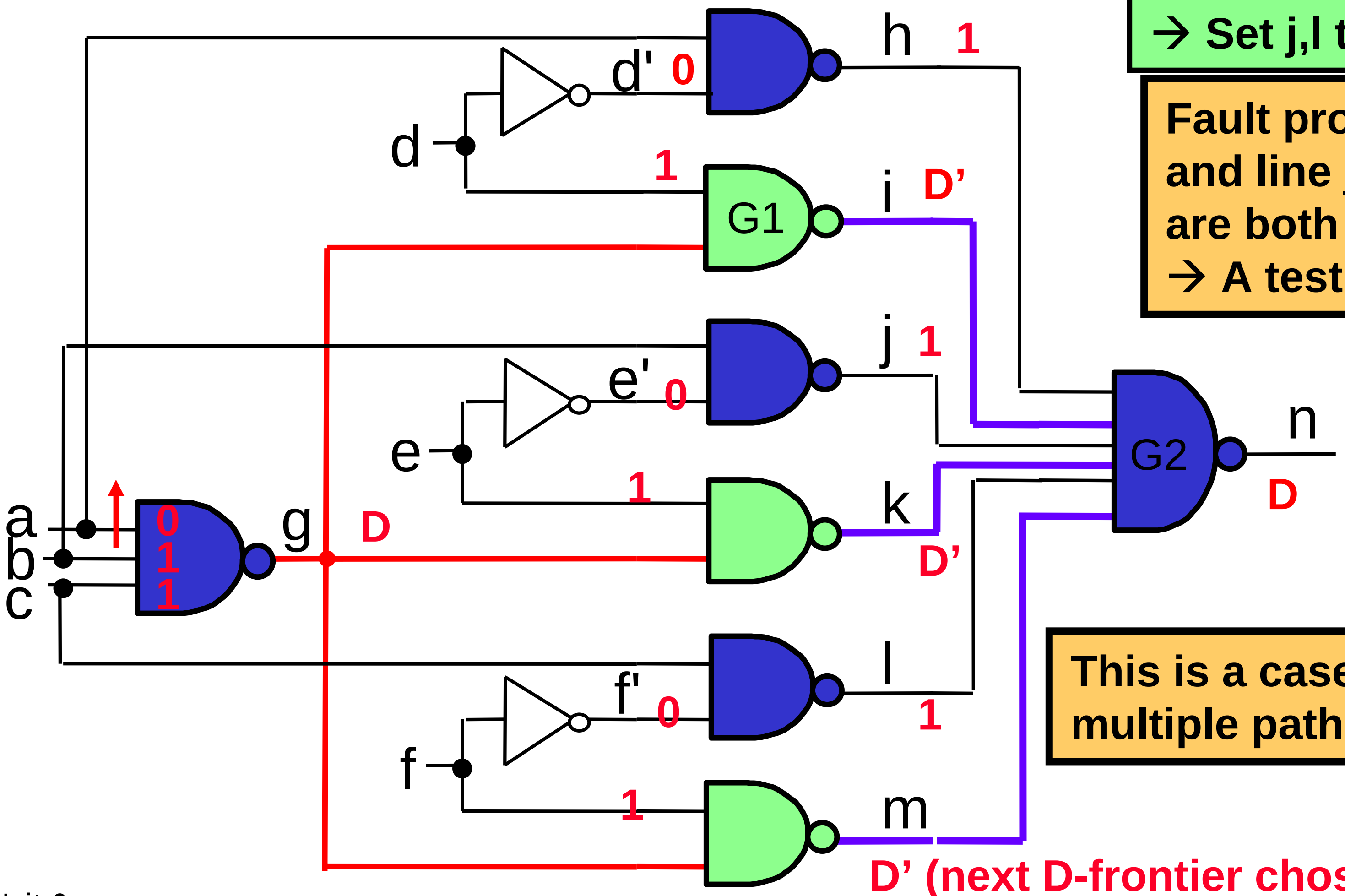
Try to propagate Fault effect thru G2 → Set j,l,m to 1

D' (next D-frontier chosen)

Conflict at m → Backtrack !

Ex: D-Algorithm (3/3)

- Five logic values
  - { 0, 1, x, D, D’ }



D’ (next D-frontier chosen)

Try to propagate Fault effect thru G2 → Set j,l to 1

Fault propagation and line justification are both complete → A test is found !

This is a case of multiple path sensitization !

D-Algorithm: Value Computation

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

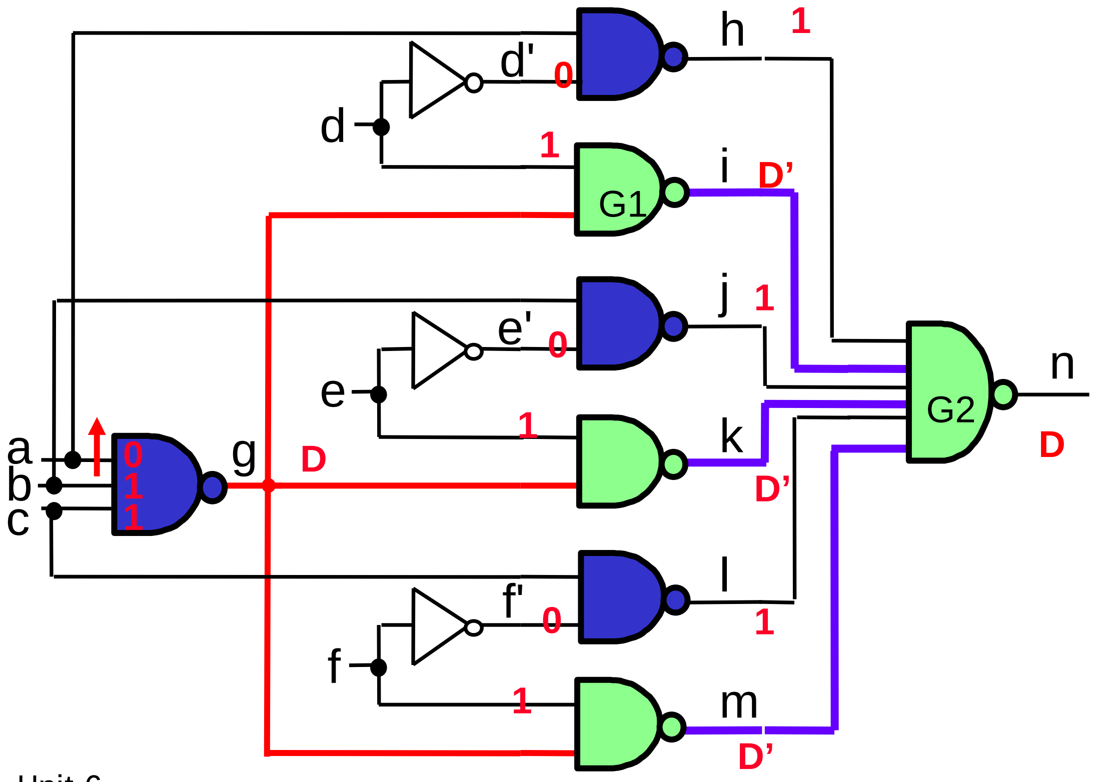
e=1	k=D’
	e’=0
	j=1
		Propagate via k

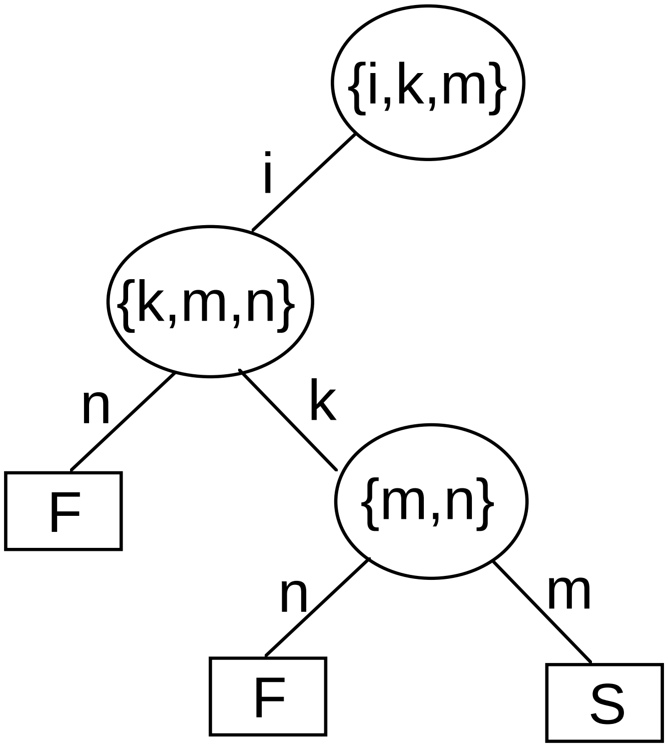
l=1
m=1
	n=D
	f’=0
	f=1
	m=D’
		Propagate via n
		Contradiction

f=1	m=D’
	f’=0
	l=1
	n=D
		Propagate via m

Decision Tree on D-Frontier

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

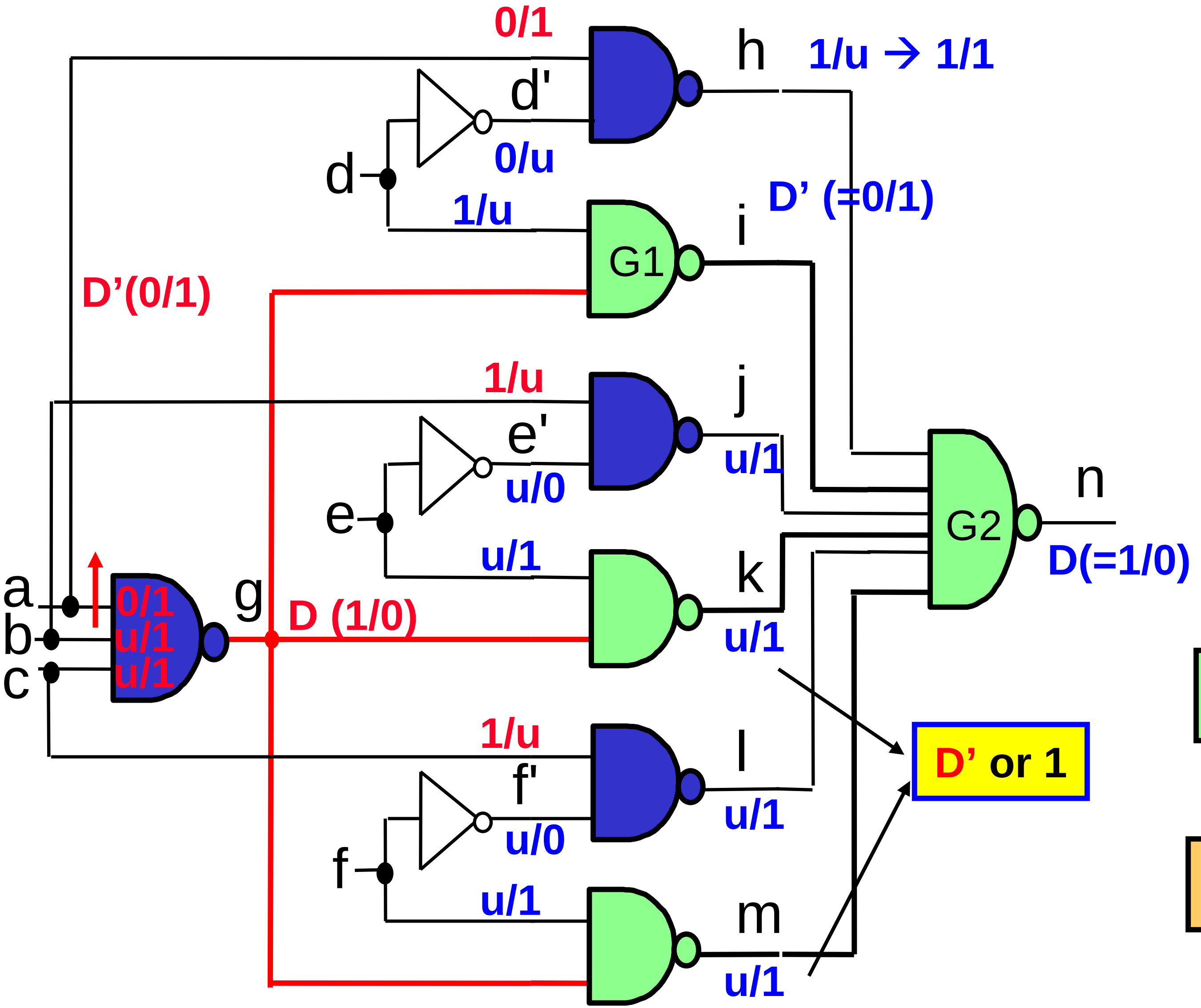




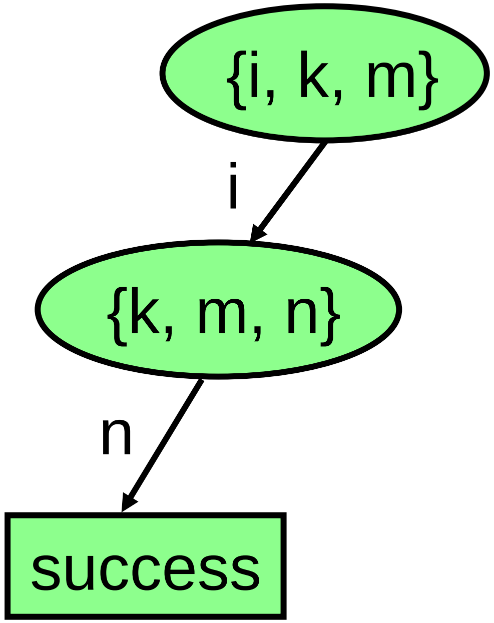
9-Value D-Algorithm

- Logic values (fault-free / faulty)
  - **{0/0, 0/1, 0/u, 1/0, 1/1, 1/u, u/0, u/1, u/u},**
  - **where 0/u={0,D'}, 1/u={D,1}, u/0={0,D}, u/1={D',1}, u/u={0,1,D,D'}.**
- Advantage:
  - **Automatically considers multiple-path sensitization, thus reducing the amount of search in D-algorithm**
  - **The speed-up is NOT very significant in practice because most faults are detected through single-path sensitization**

Example: 9-Value D-Algorithm



Decision Tree



No-backtrack !

Final Step of 9-Value D-Algorithm

- To derive the test vector
  - A = (0/1) → 0 (take the fault-free one)
  - B = (1/u) → 1
  - C = (1/u) → 1
  - D = (u/1) → 1
  - E = (u/1) → 1
  - F = (u/1) → 1
- The final vector
  - (A,B,C,D,E,F) = (0, 1, 1, 1, 1, 1)

Outline of ATPG (Automatic Test Pattern Generation)

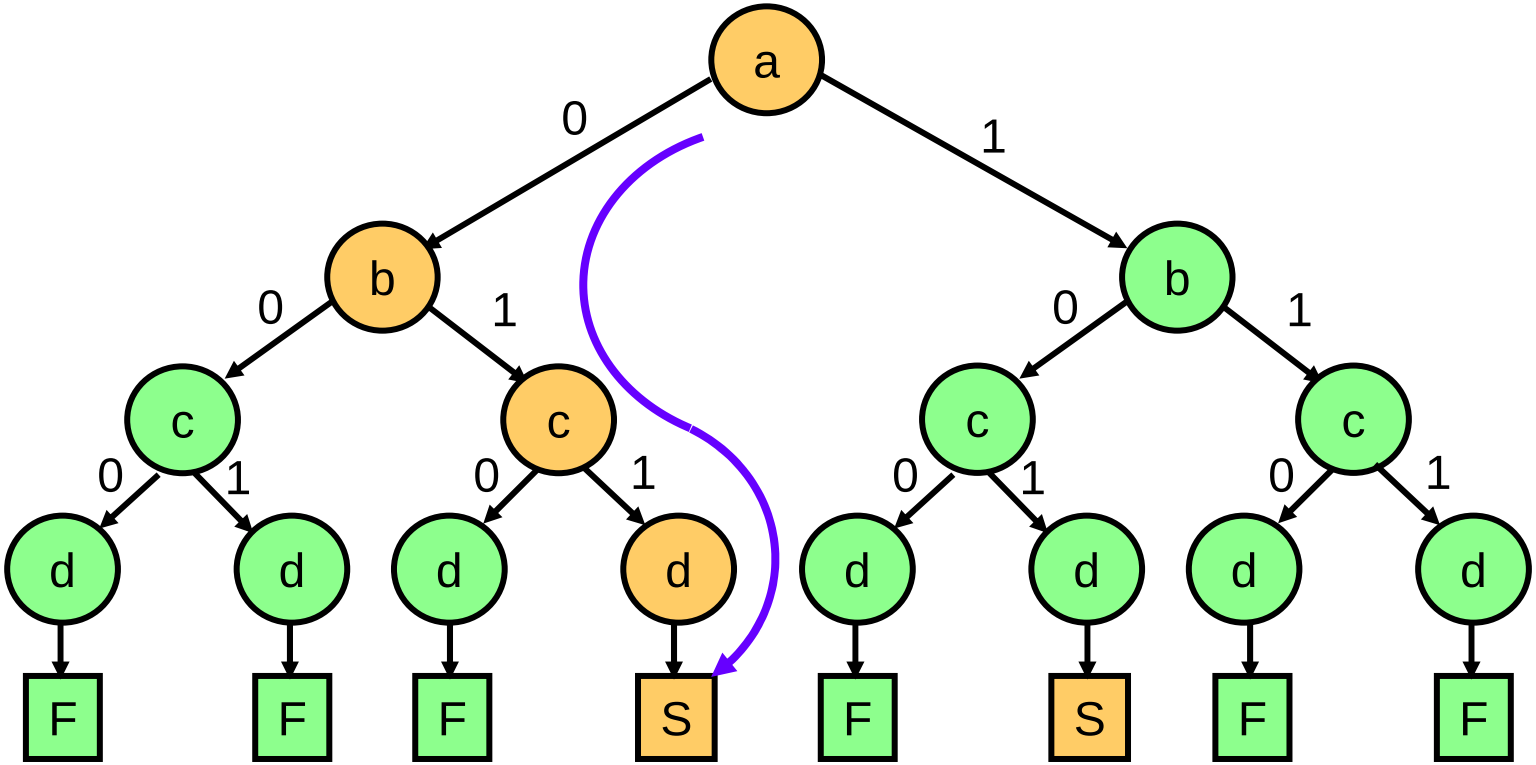
- Test Generation (TG) Methods
  - Based on Truth Table
  - Based on Boolean Equation
  - Based on Structural Analysis
  - D-algorithm [Roth 1967]
  - 9-Valued D-algorithm [Cha 1978]
  - PODEM [Goel 1981]
  - FAN [Fujiwara 1983]

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

Search Space of PODEM

- Complete Search Space
  - A binary tree with 2^n leaf nodes, where n is the number of PI's
- Fast Test Generation
  - Need to find a path leading to a SUCCESS terminal quickly



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 PI's
- 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 PI's
  - The result is a PI signal-value pair (x, v_x)
  - No signal value is actually assigned during backtrace !

往輸入端追蹤

Objective Routine

- Objective Routine Involves
  - The selection of a D-frontier, G
  - The 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);
}
```

Backtrace Routine

- 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$\oplus$inv;  /* new objective value */
    }
    /* G is a PI */    return (G, v);
}
```

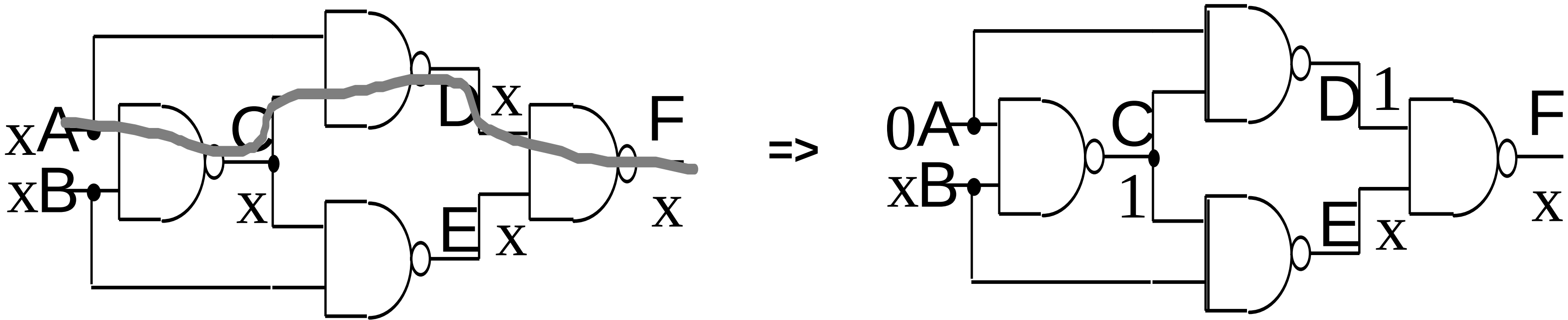
Example: Backtrace

Objective to achieved: (F, 1)

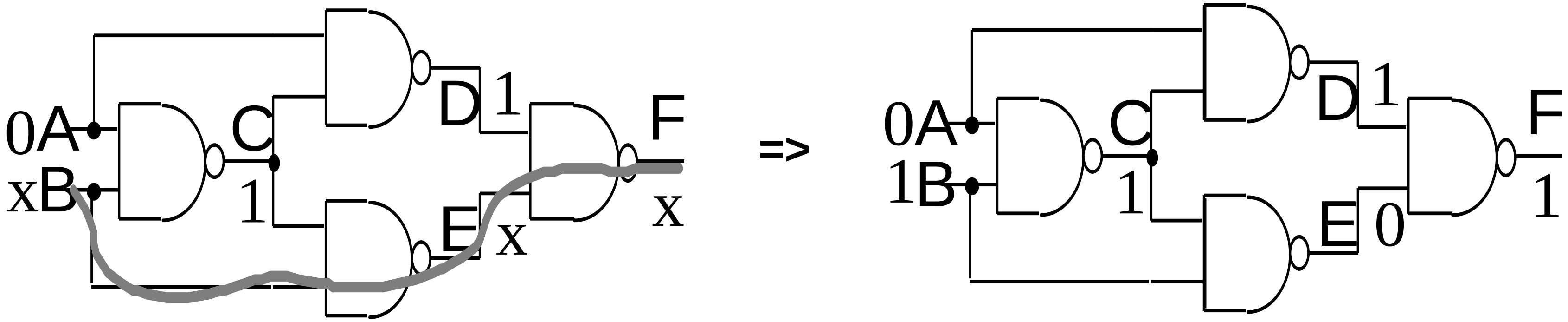
PI assignments:

(1) A = 0 → fail

(2) B = 1 → succeed



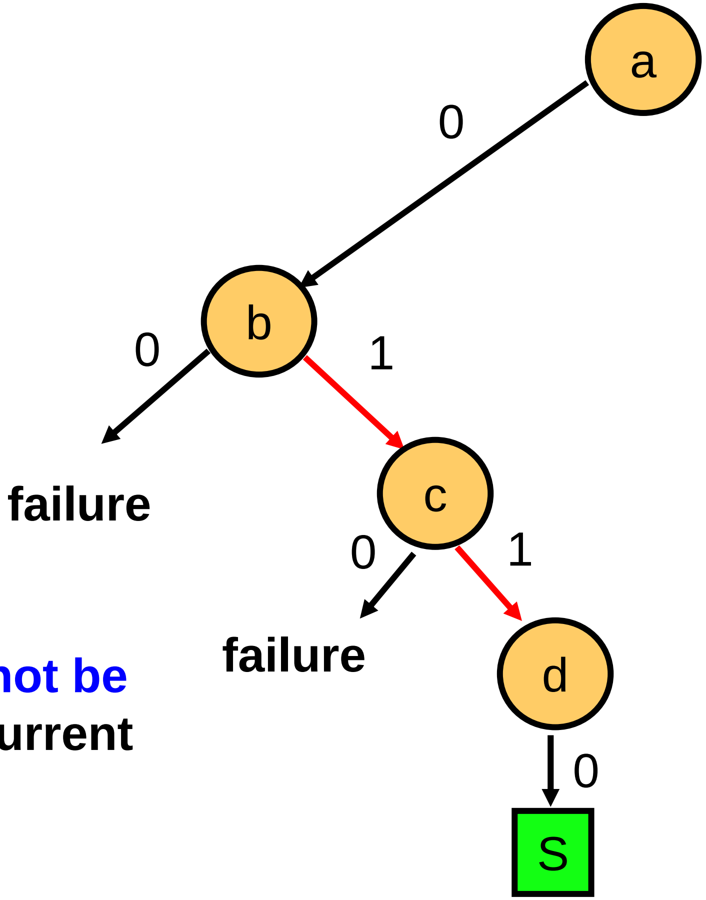
The first time of backtracing



The second time of backtracing

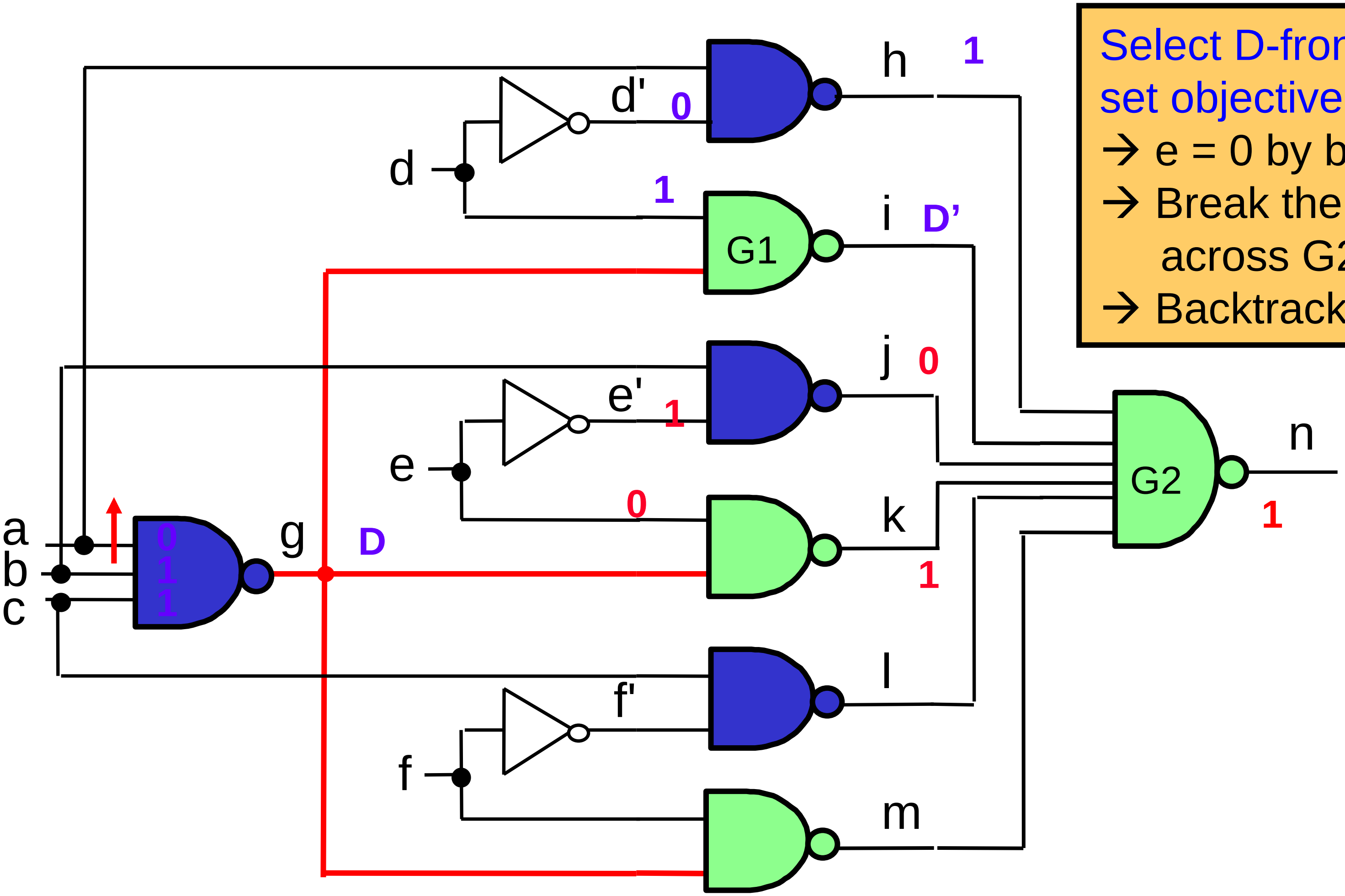
PI Assignment in PODEM

Assume that: PI's: { 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

Example: PODEM (1/3)



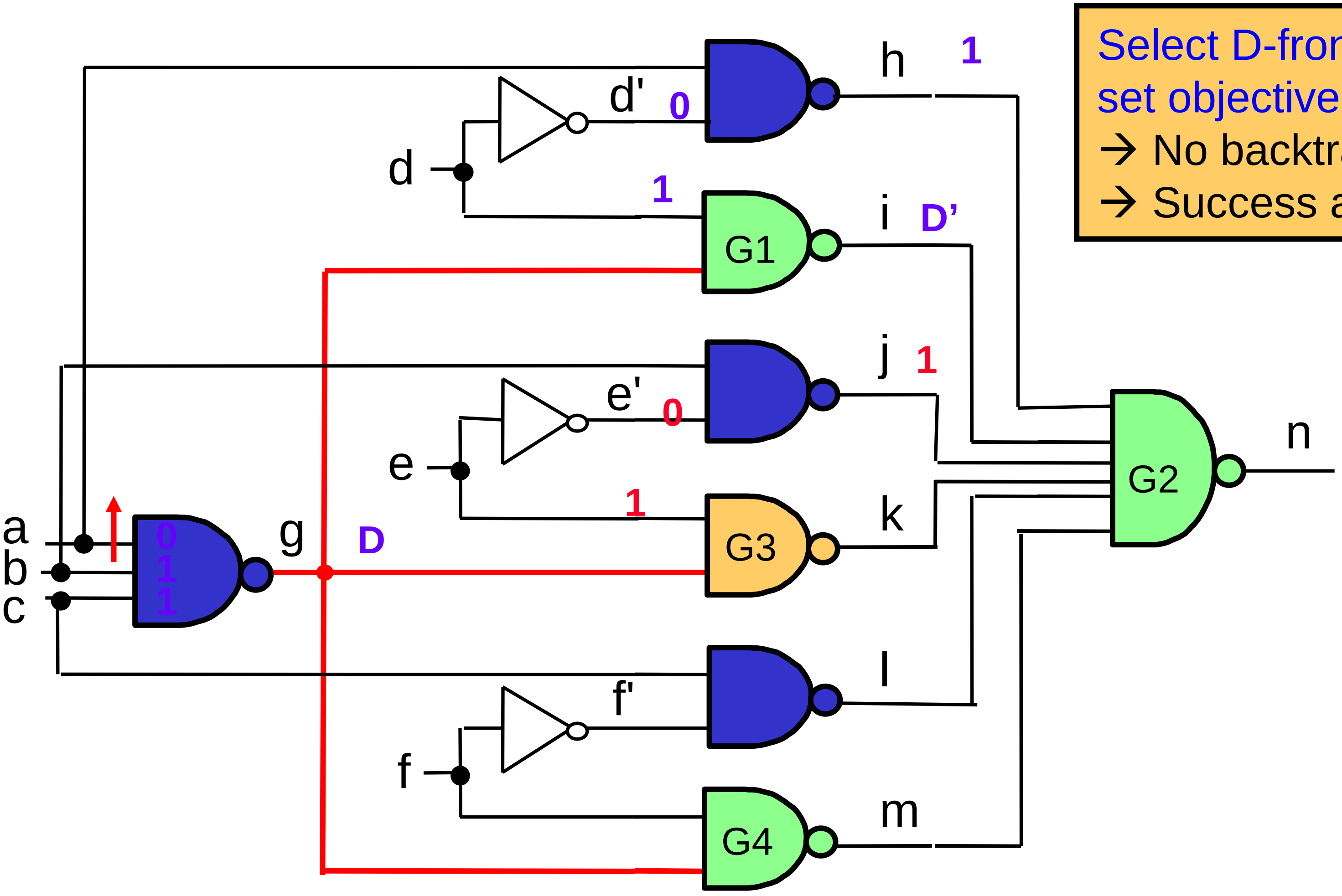
Select D-frontier G2 and set objective to (k,1)

→ e = 0 by backtrace

→ Break the sensitization across G2

→ Backtrack !

Example: PODEM (2/3)

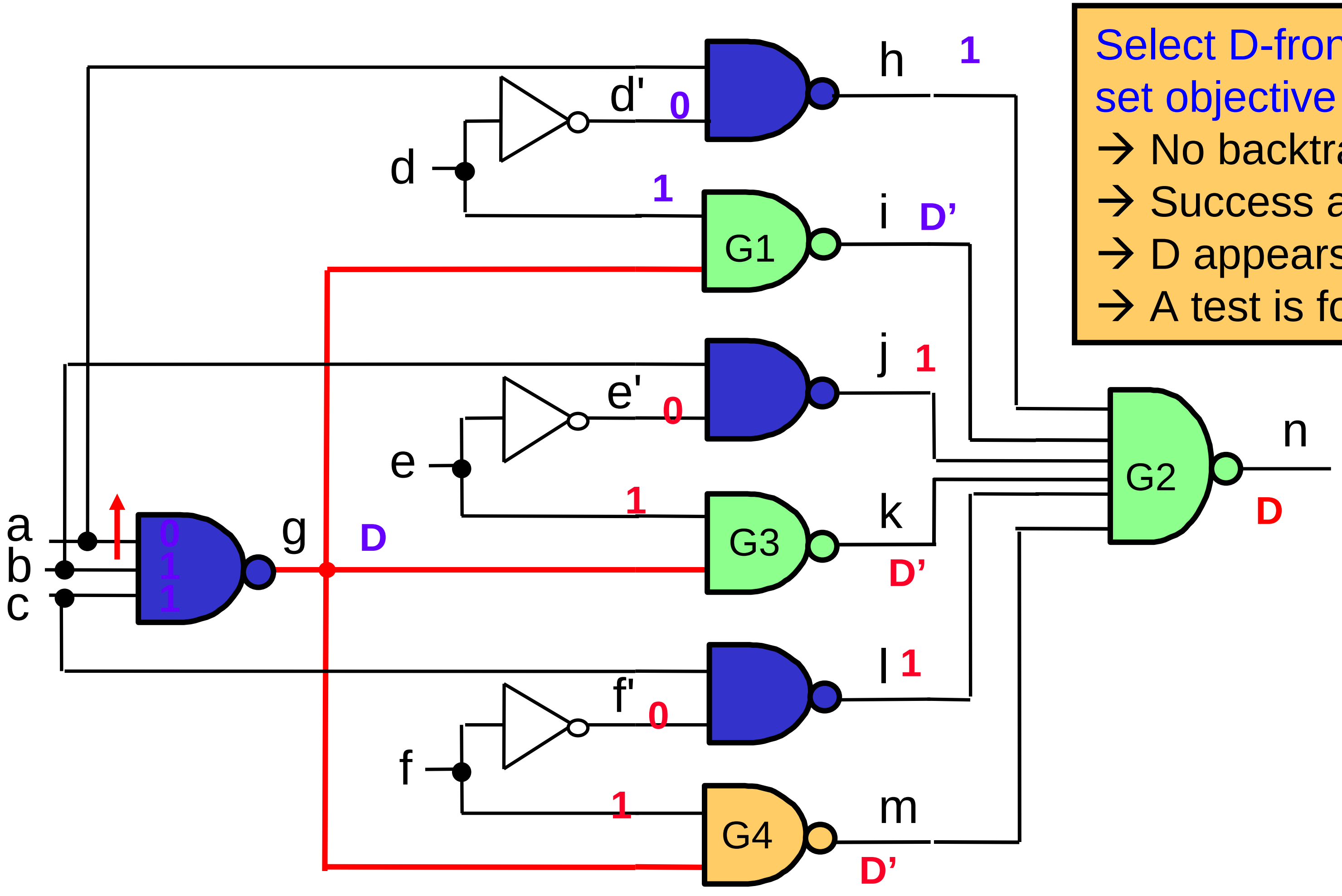


Select D-frontier G3 and set objective to (e,1)

→ No backtrace is needed

→ Success at G3

Example: PODEM (3/3)



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

PODEM: Value Computation

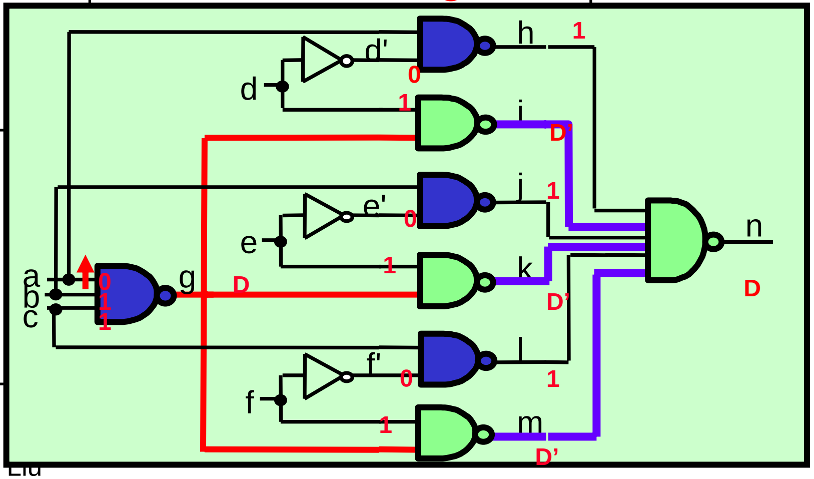
Objective	PI assignment	Implications	D-frontier	Comments
a=0	a=0	h=1	g	
b=1	b=1		g	
c=1	c=1	g=D	i,k,m	
d=1	d=1	d'=0 i=D'	k,m,n	

k=1	e=0	e'=1
		j=0
		k=1
		n=1
			m	no solutions ! → backtrack

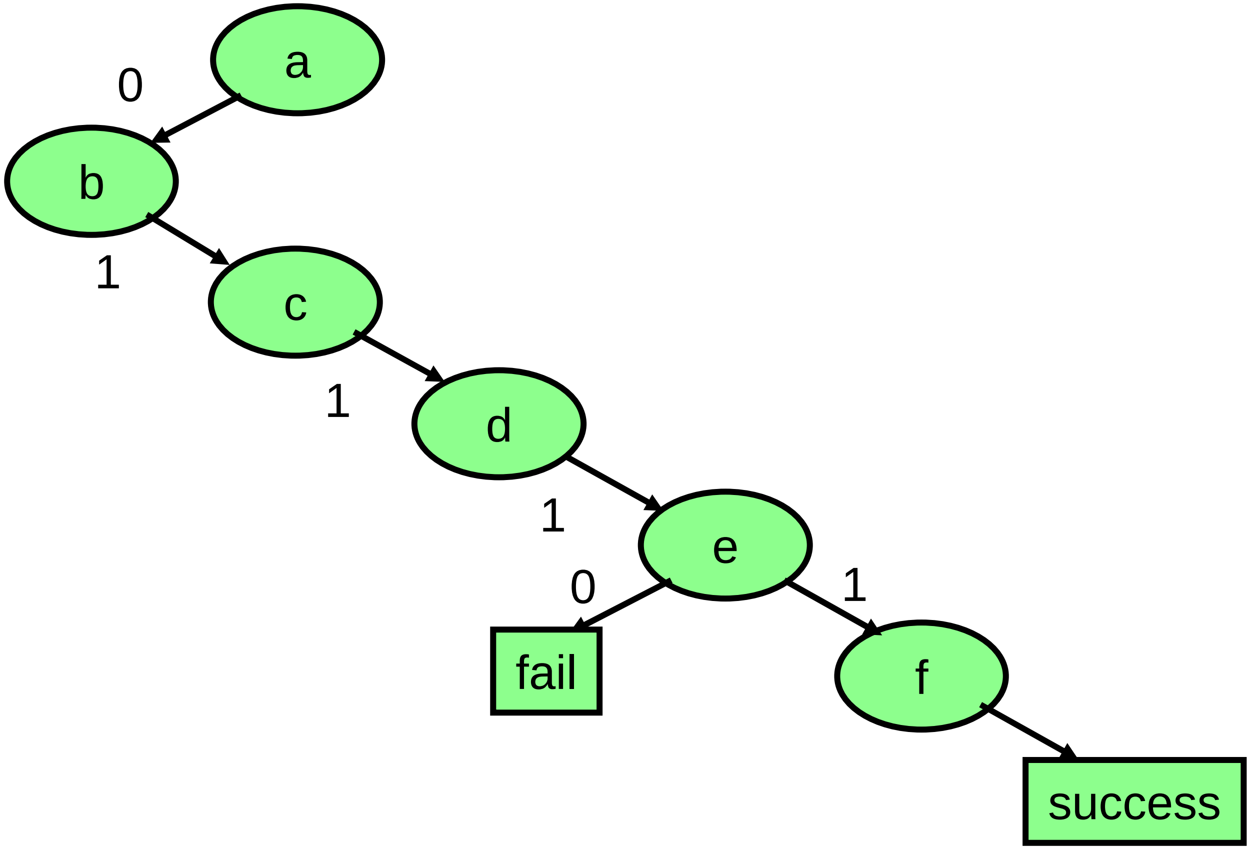
	e=1	e'=0
		j=1
		k=D'
			m,n	reverse PI assignment

l=1	f=1	f'=0
		l=1
		m=D'
		n=D		

Assignments need to be reversed during backtracking



Decision Tree in PODEM



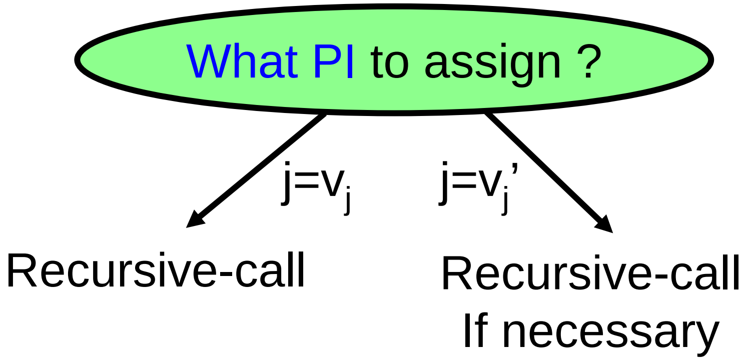
- **Decision node: the PI selected through backtrace for value assignment**
- **Branch: the value assignment to the selected PI**

Terminating 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: Recursive 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
```

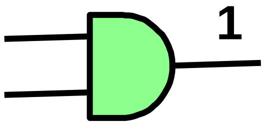


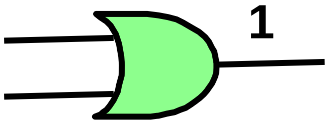
Overview of PODEM

- PODEM
  - examines all possible input patterns implicitly but exhaustively (branch-and-bound) for finding a test
  - It is complete like D-algorithm (I.e., will find one if a test 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
  - Experimental results show that PODEM is generally faster than the D-algorithm

The Selection Strategy in PODEM

- In Objective() and Backtrace()
  - Selections are done arbitrarily in original PODEM
  - The algorithm will be more efficient if certain guidance used in the selections of objective node and backtrace path
- Selection Principle
  - Principle 1: Among several unsolved problems
    - → Attack the hardest one
    - Ex: to justify a ‘1’ at an AND-gate output
  - Principle 2: Among several solutions for solving a problem
    - → Try the easiest one
    - Ex: to justify a ‘1’ at OR-gate output

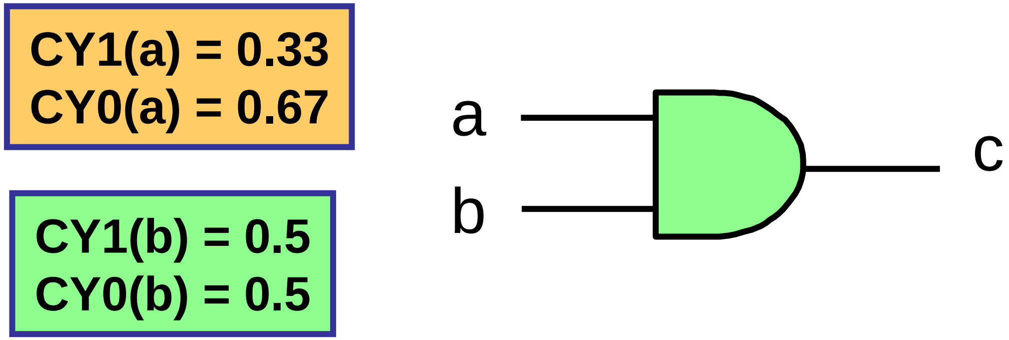




Controllability As Guidance

- Controllability of a signal w
  - CY1(w): the probability that line w has value 1.
  - CY0(w): the probability that line w has value 0.
  - Example:
    - f = ab
    - Assume CY1(a)=CY0(a)=CY1(b)=CY0(b)=0.5
    - → CY1(f)=CY1(a)xCY1(b)=0.25,
    - → CY0(f)=CY0(a)+CY0(b)-CY0(a)xCY0(b)=0.75

- Example of Smart Backtracing
  - Objective (c, 1) → choose path c→a for backtracing
  - Objective (c, 0) → choose path c→a for backtracing



CY1(a) = 0.33
CY0(a) = 0.67

CY1(b) = 0.5
CY0(b) = 0.5

Testability Analysis

- Applications
  - To give an early warning about the testing problems that lie ahead
  - To provide guidance in ATPG
- Complexity
  - Should be simpler than ATPG and fault simulation, I.e., need to be linear or almost linear in terms of circuit size
- Topology analysis
  - Only the structure of the circuit is analyzed
  - No test vectors are involved
  - Only approximate, reconvergent fanouts cause inaccuracy

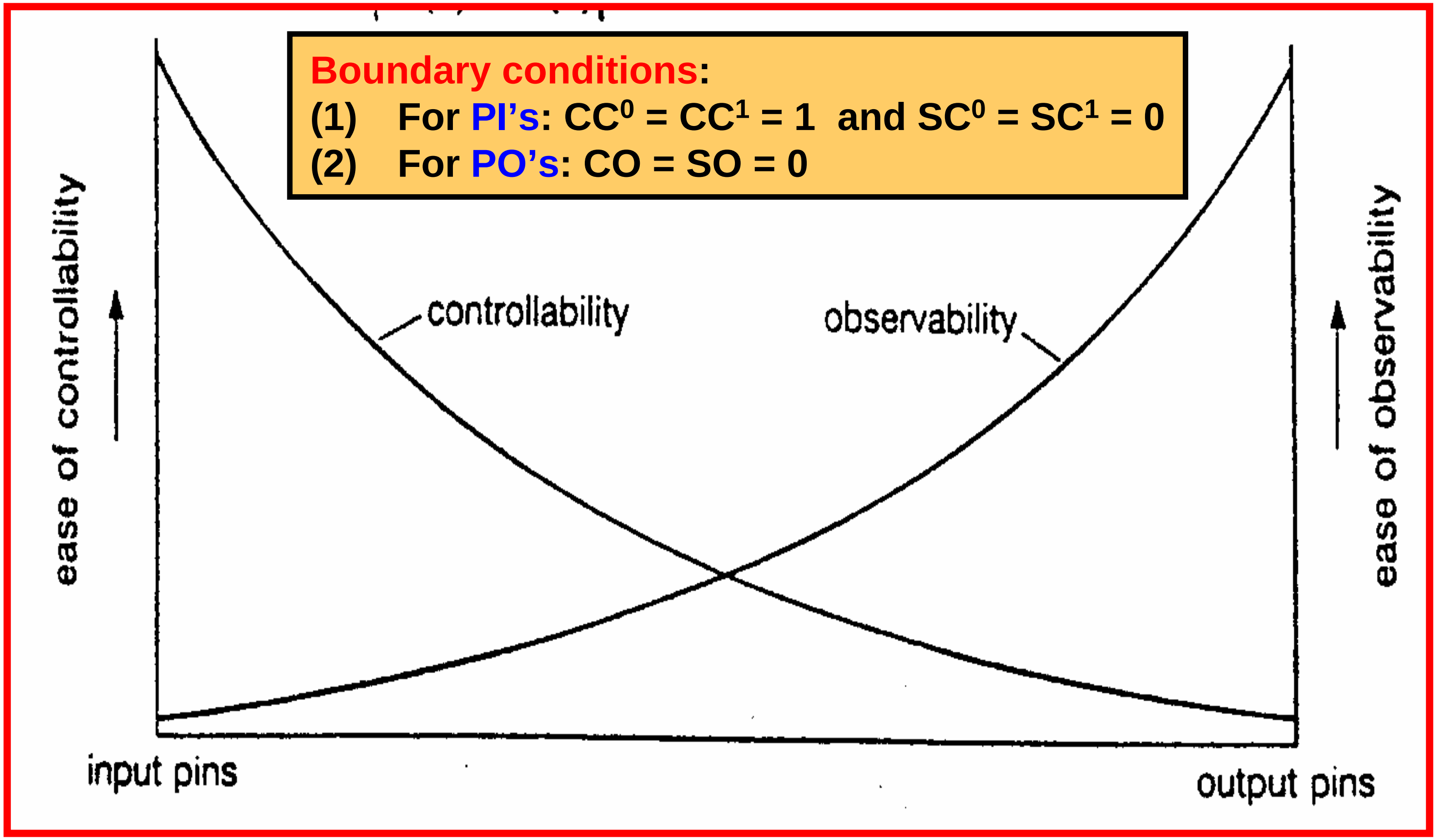
SCOAP

(Sandia Controllability/Observability Analysis Program)

- Computes six numbers for each node N
  - $CC^0(N)$ and $CC^1(N)$
    - Combinational 0 and 1 controllability of a node N
  - $SC^0(N)$ and $SC^1(N)$
    - Sequential 0 and 1 controllability of a node N
  - CO(N)
    - Combinational observability
  - SO(N)
    - Sequential observability

General Characteristic of Controllability and Observability

Controllability calculation: sweeping the circuit from PI to PO
Observability calculation: sweeping the circuit from PO to PI



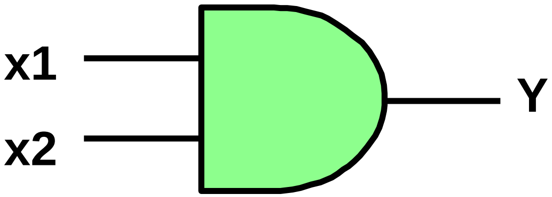
Boundary conditions:

(1) For PI's: $CC^0 = CC^1 = 1$ and $SC^0 = SC^1 = 0$

(2) For PO's: CO = SO = 0

Controllability Measures

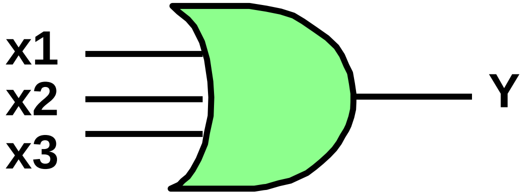
- $CC^0(N)$ and $CC^1(N)$
  - The number of combinational nodes that must be assigned values to justify a 0 or 1 at node N
- $SC^0(N)$ and $SC^1(N)$
  - The number of sequential nodes that must be assigned values to justify a 0 or 1 at node N



$$CC^0(Y) = \min [CC^0(x1) , CC^0(x2)] + 1$$
$$CC^1(Y) = CC^1(x1) + CC^1(x2) + 1$$
$$SC^0(Y) = \min [SC^0(x1) , SC^0(x2)]$$
$$SC^1(Y) = SC^1(x1) + SC^1(x2)$$

Controllability Measure (con't)

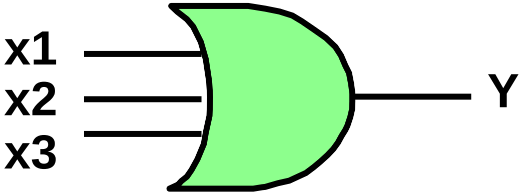
- $CC^0(N)$ and $CC^1(N)$
  - The number of combinational nodes that must be assigned values to justify a 0 or 1 at node N
- $SC^0(N)$ and $SC^1(N)$
  - The number of sequential nodes that must be assigned values to justify a 0 or 1 at node N



$$CC^0(Y) = CC^0(x1) + CC^0(x2) + CC^0(x3) + 1$$
$$CC^1(Y) = \min [\, CC^1(x1), CC^1(x2), CC^1(x3) \,] + 1$$
$$SC^0(Y) = SC^0(x1) + SC^0(x2) + SC^0(x3)$$
$$SC^1(Y) = \min [\, SC^1(x1) , SC^1(x2) , SC^1(x3) \,]$$

Observability Measure

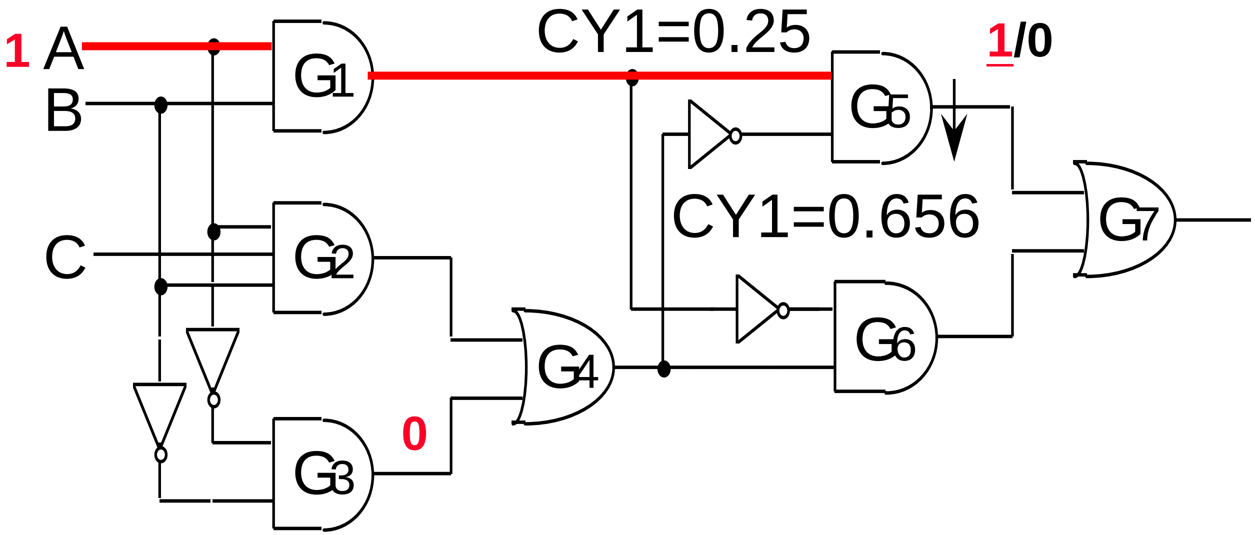
- CO(N) and SO(N)
  - The observability of a node N is a function of the output observability and of the cost of holding all other inputs at non-controlling values



$$CO(x1) = CO(Y) + CC^0(x2) + CC^0(x3) + 1$$
$$SO(x1) = SO(Y) + SC^0(x2) + SC^0(x3)$$

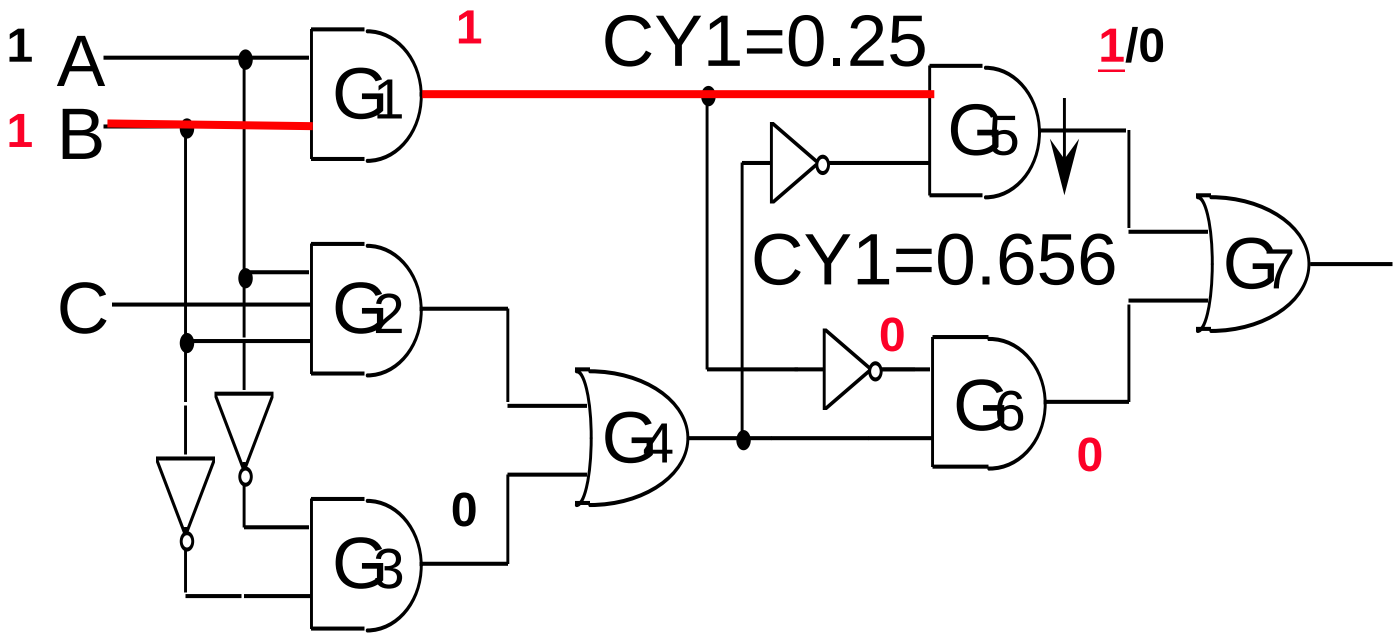
PODEM: Example 2 (1/3)

Initial objective=(G5,1).
G5 is an AND gate → Choose the hardest-1
→ Current objective=(G1,1).
G1 is an AND gate → Choose the hardest-1
→ Arbitrarily, Current objective=(A,1). A is a PI → Implication → G3=0.



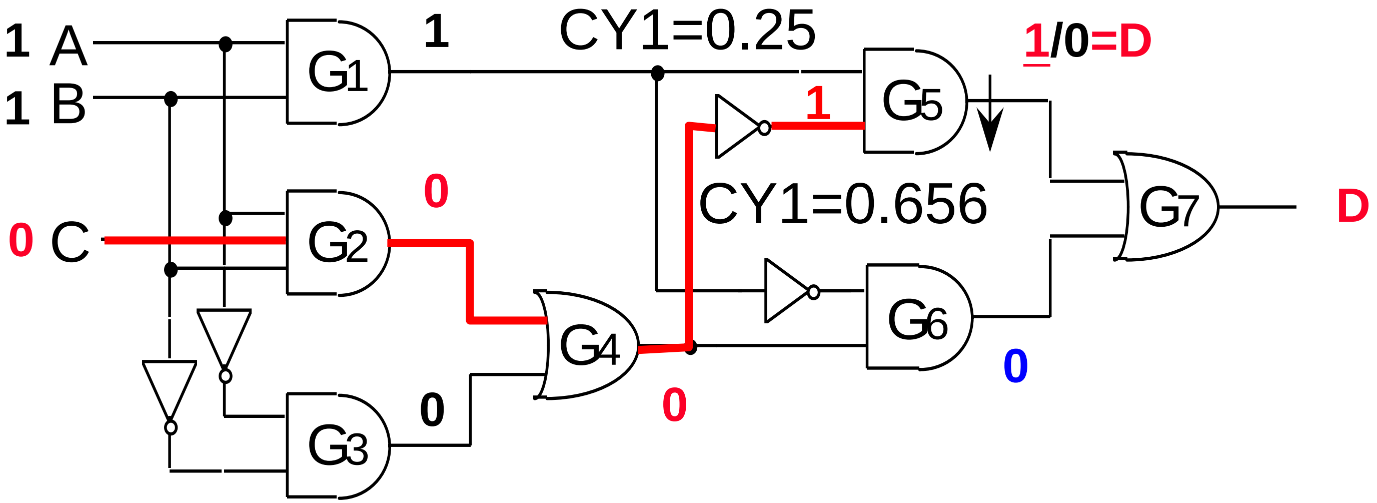
PODEM: Example 2 (2/3)

The initial objective satisfied? No! → Current objective=(G5,1).
G5 is an AND gate → Choose the hardest-1 → Current objective=(G1,1).
G1 is an AND gate → Choose the hardest-1
→ Arbitrarily, Current objective=(B,1). B is a PI → Implication → G1=1, G6=0.



PODEM: Example 2 (3/3)

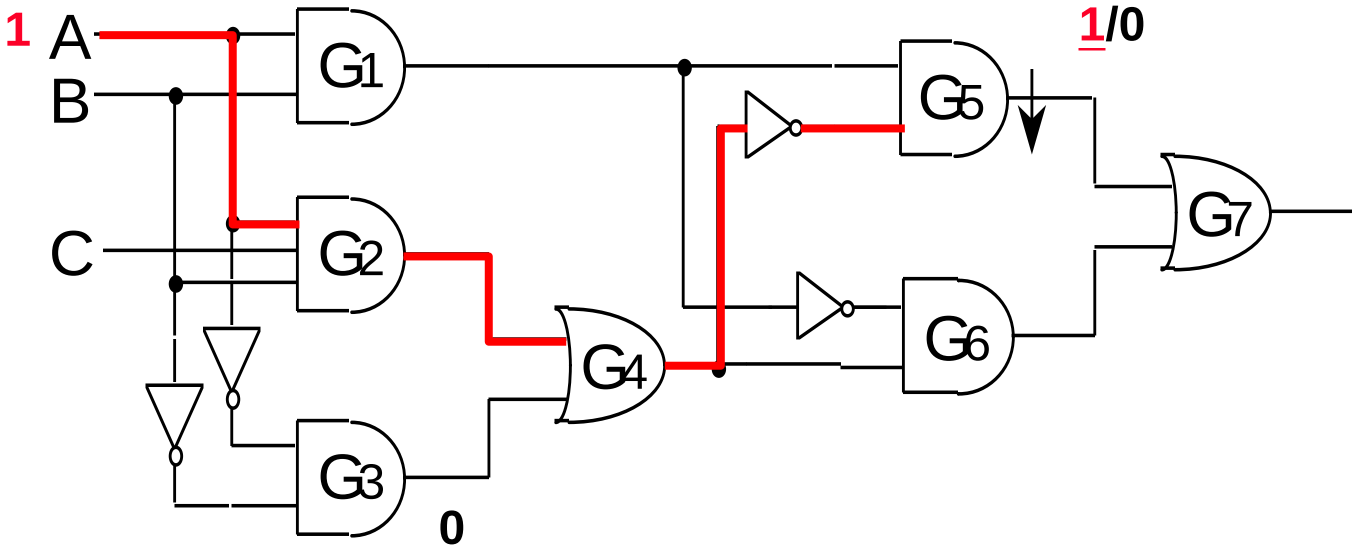
The initial objective satisfied? No! → Current objective=(G5,1).
The value of G1 is known → Current objective=(G4,0).
The value of G3 is known → Current objective=(G2,0).
A, B is known → Current objective=(C,0).
C is a PI → Implication → G2=0, G4=0, G5=D, G7=D.



No backtracking !!

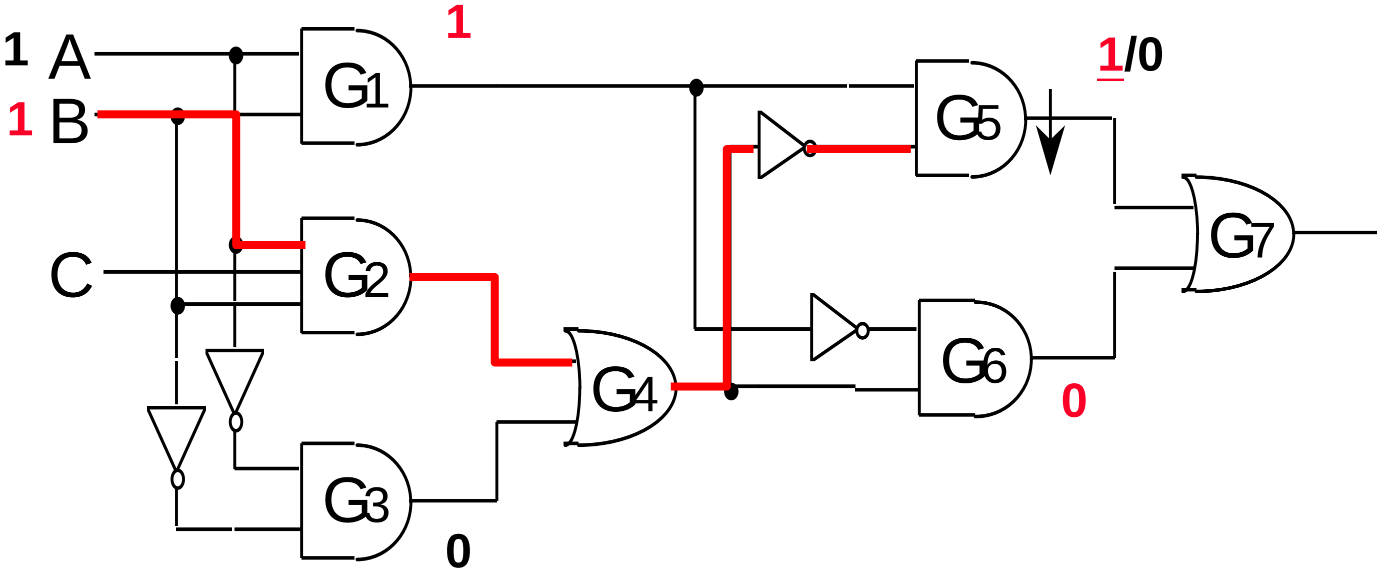
If The Backtracing Is Not Guided (1/3)

Initial objective=(G5,1).
Choose path G5-G4-G2-A → A=0.
Implication for A=0 → G1=0, G5=0 → Backtracking to A=1.
Implication for A=1 → G3=0.



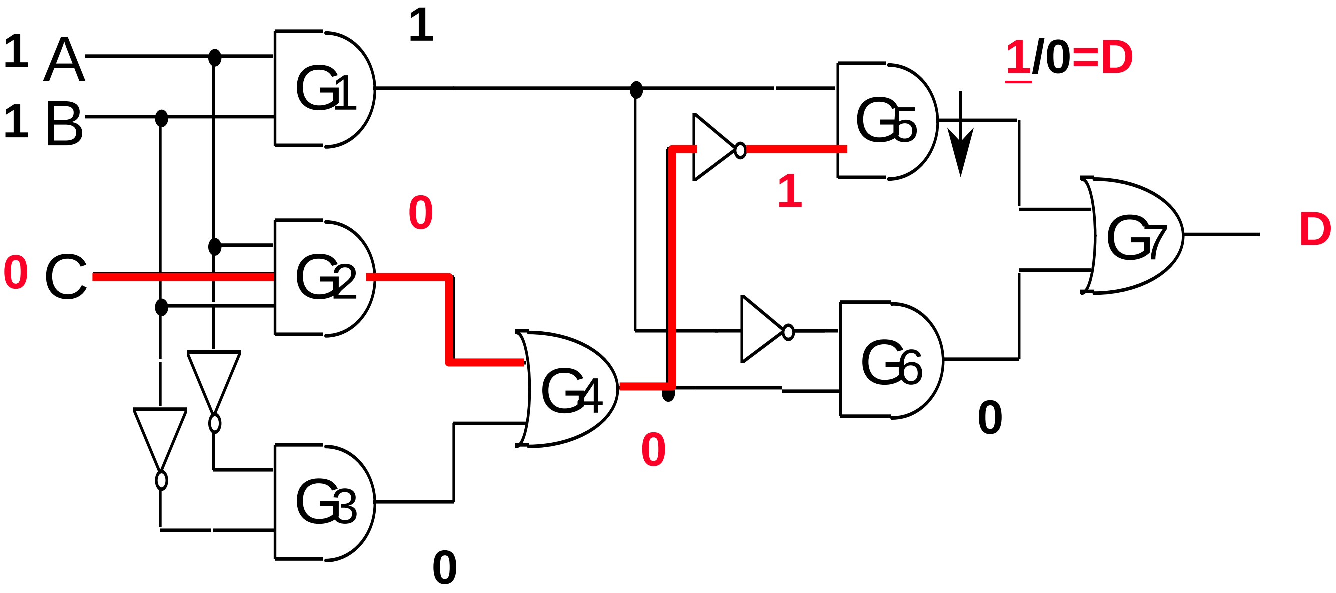
If The Backtracing Is Not Guided (2/3)

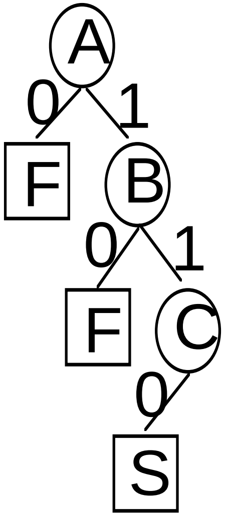
The initial objective satisfied? No! → Current objective=(G5,1).
Choose path G5-G4-G2-B → B=0.
Implication for B=0 → G1=0, G5=0 → Backtracking to B=1.
Implication for B=1 → G1=1, G6=0.



If The Backtracing Is Not Guided (3/3)

The initial objective satisfied? No! → Current objective=(G5,1).
Choose path G5-G4-G2-C → C=0.
Implication for C=0 → G2=0, G4=0, G5=D, G7=D.





Two times of backtracking !!

Outline of ATPG
(Automatic Test Pattern Generation)

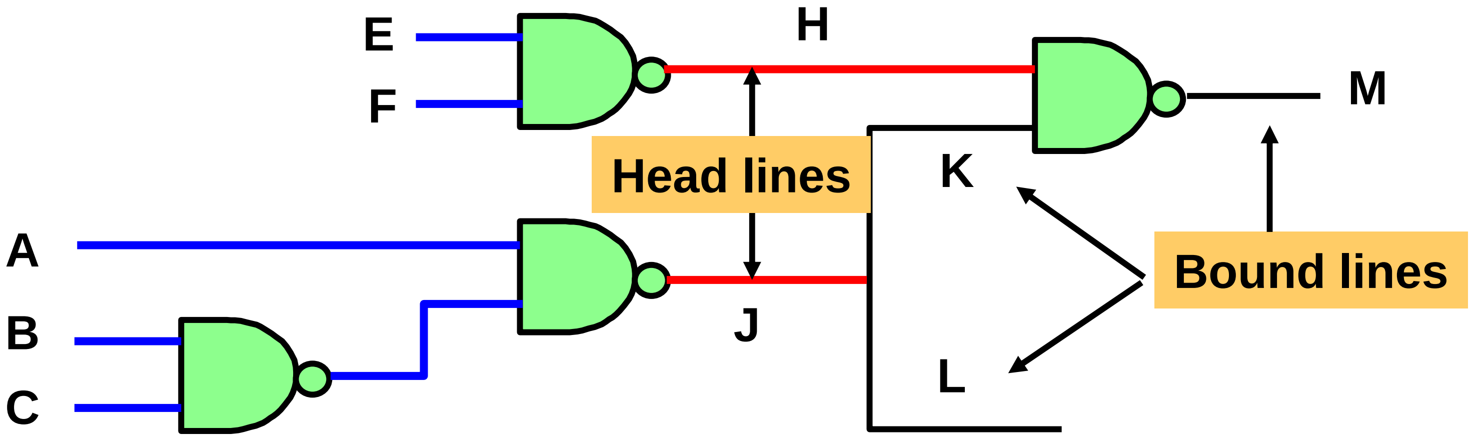
- Test Generation (TG) Methods
  - Based on Truth Table
  - Based on Boolean Equation
  - Based on Structural Analysis
  - D-algorithm [Roth 1967]
  - 9-Valued D-algorithm [Cha 1978]
  - PODEM [Goel 1981]
  - FAN [Fujiwara 1983]

FAN (Fanout Oriented) Algorithm

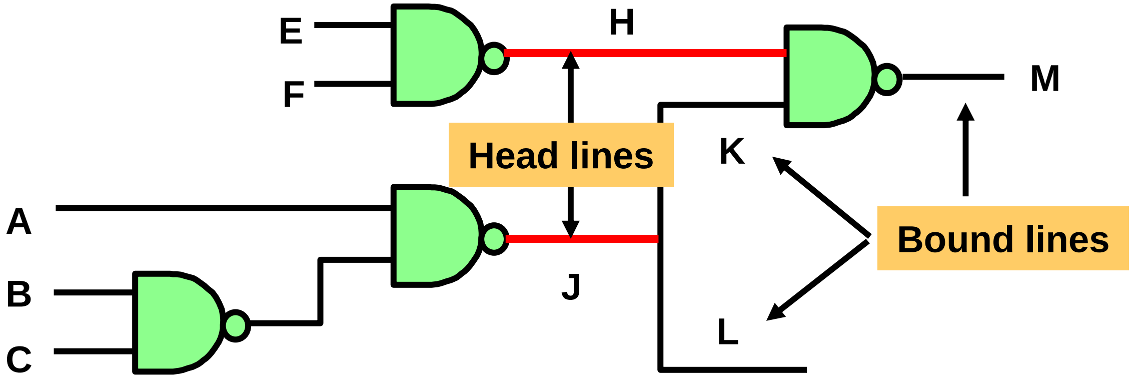
- FAN
  - Introduces two major extensions to PODEM's backtracing algorithm
- 1st extension
  - Rather than stopping at PI's, backtracing in FAN may stop at an internal lines
- 2nd extension
  - FAN uses multiple backtrace procedure, which attempts to satisfy a set of objectives simultaneously

Headlines and Bound Lines

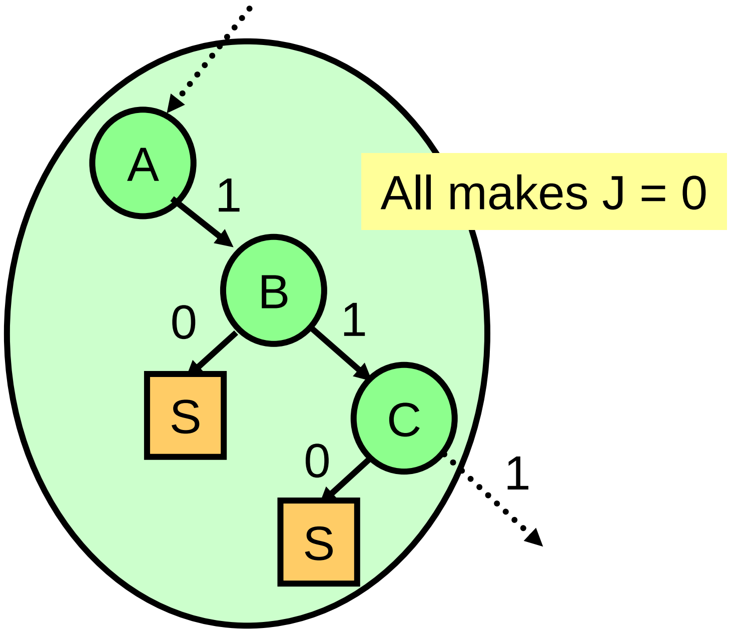
- Bound line
  - A line reachable from at least one stem
- Free line
  - A line that is NOT bound line
- Head line
  - A free line that directly feeds a bound line



Decision Tree (PODEM v.s. FAN)



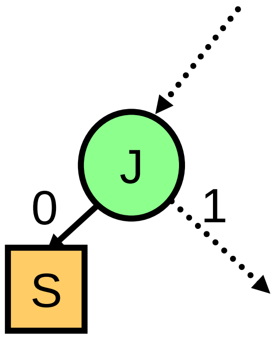
Assume that:
Objective is (J, 0)



All makes J = 0

PODEM

J is a head line
→ Backtrace stops at J
→ Avoid unnecessary search



FAN

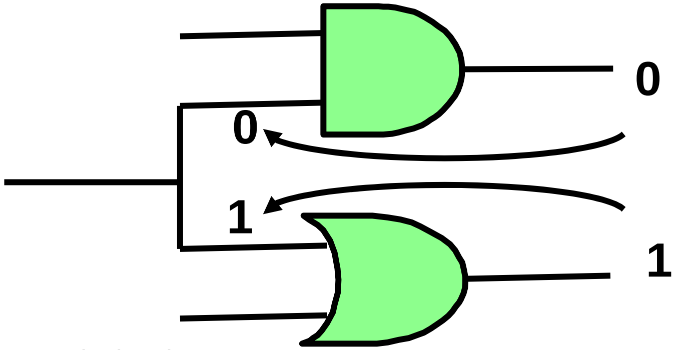
Why Stops at Head Lines ?

- Head lines are mutually independent
  - Hence, for each given value combination at head lines, there always exists an input combination to realize it.
- FAN has two-steps
  - Step 1: PODEM using headlines as pseudo-PI's
  - Step 2: Generate real input pattern to realize the value combination at head lines.

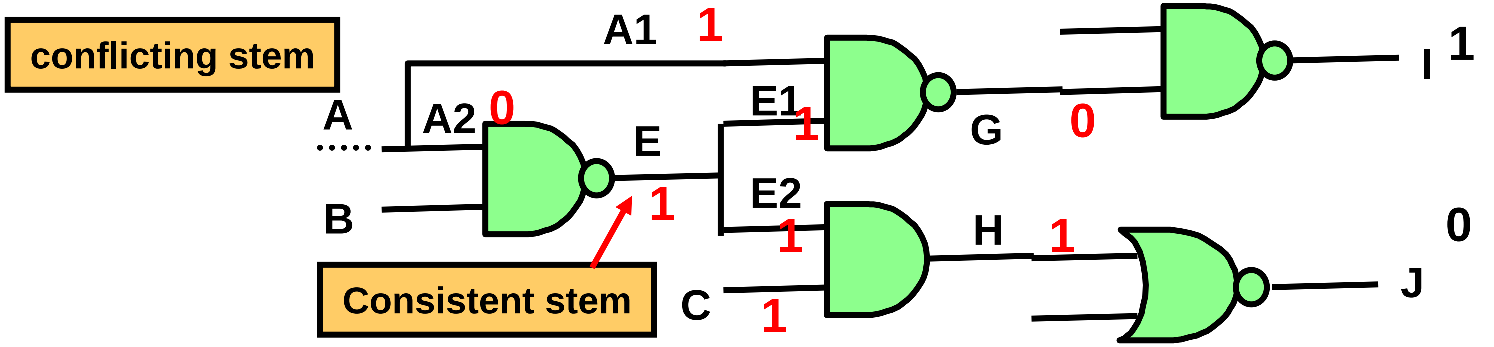
Why Multiple Backtrace ?

- Drawback of Single Backtrace
  - A PI assignment satisfying one objective →may preclude achieving another one, and this leads to backtracking
- Multiple Backtrace
  - Starts from a set of objectives (Current_objectives)
  - Maps these multiple objectives into a head-line assignment $k=v_k$ that is likely to
    - Contribute to the achievement of a subset of the objectives
    - Or show that some subset of the original objectives cannot be simultaneously achieved

Multiple objectives May have conflicting Requirements at a stem



Example: Multiple Backtrace



conflicting stem

Consistent stem

Current_objectives	Processed entry	Stem_objectives	Head_objectives
(I,1), (J,0)	(I,1)		
(J,0), (G,0)	(J,0)		
(G,0), (H,1)	(G,0)		
(H,1), (A1,1), (E1,1)	(H,1)		
(A1,1), (E1,1), (E2,1), (C,1)	(A1,1)	A	
(E1,1), (E2,1), (C,1)	(E1,1)	A,E	
(E2,1), (C,1)	(E2,1)	A,E	
(C,1)	(C,1)	A,E	C
Empty → restart from (E,1)		A	C
(E,1)	(E,1)	A	C
(A2,0)	(A2,0)	A	C
empty		A	C

References For ATPG

[1] Sellers et al., "Analyzing errors with the Boolean difference", IEEE Trans. Computers, pp. 676-683, 1968.

[2] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method", IBM Journal of Research and Development, pp. 278-291, July, 1966.

[2'] J. P. Roth et al., "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits", IEEE Trans. Electronic Computers, pp. 567-579, Oct. 1967.

[3] C. W. Cha et al, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits", IEEE TC, pp. 193-200, March, 1978.

[4] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", IEEE Trans. Computers, pp. 215-222, March, 1981.

[5] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms", IEEE TC, pp. 1137-1144, Dec. 1983.

[6] M. H. Schulz et al., "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", IEEE Trans. on CAD, pp. 126-137, 1988.

[6'] M. H. Schulz and E. Auth, "Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification", IEEE Trans CAD, pp. 811-816, 1989.

Outline

- Introduction
- Fault Modeling
- Fault Simulation
- Test 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 is inevitable
  - Like death and tax

Design For Testability

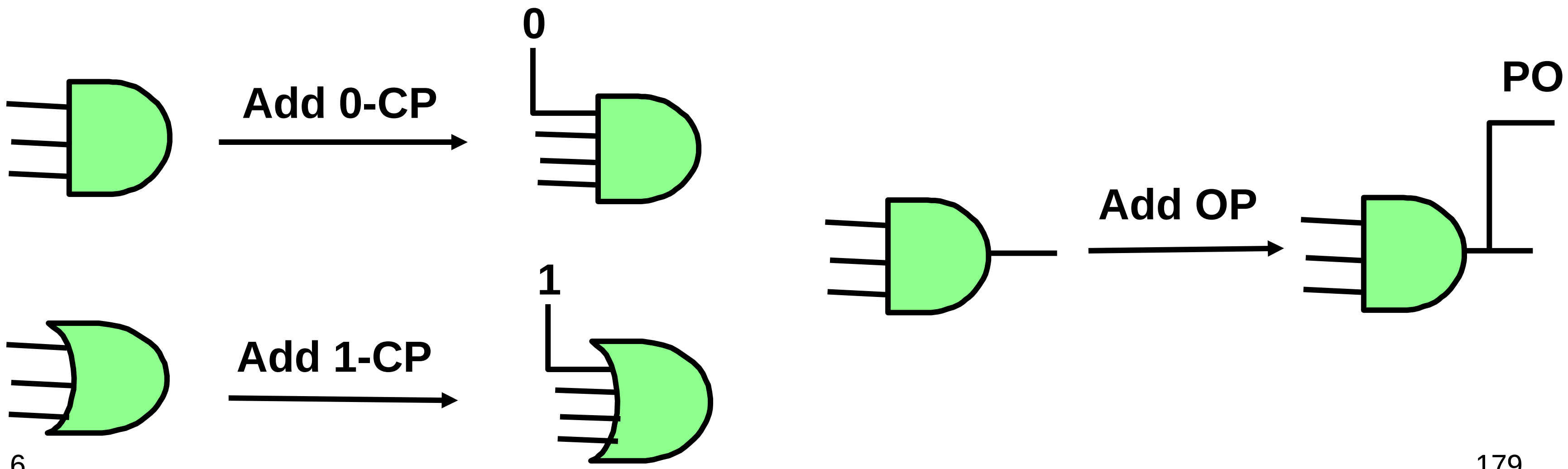
- Definition
  - Design For Testability (DFT) refers to those design techniques that make test generation and testing cost-effective
- DFT Methods
  - Ad-hoc methods
  - Scan, full and partial
  - Built-In Self-Test (BIST)
  - Boundary scan
- Cost of DFT
  - Pin count, area, performance, design-time, test-time

Important Factors

- Controllability
  - Measure the ease of controlling a line
- Observability
  - Measure the ease of observing a line at PO
- Predictability
  - Measure the ease of predicting output values
- DFT deals with ways of improving
  - Controllability
  - Observability
  - Predictability

Test Point Insertion

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

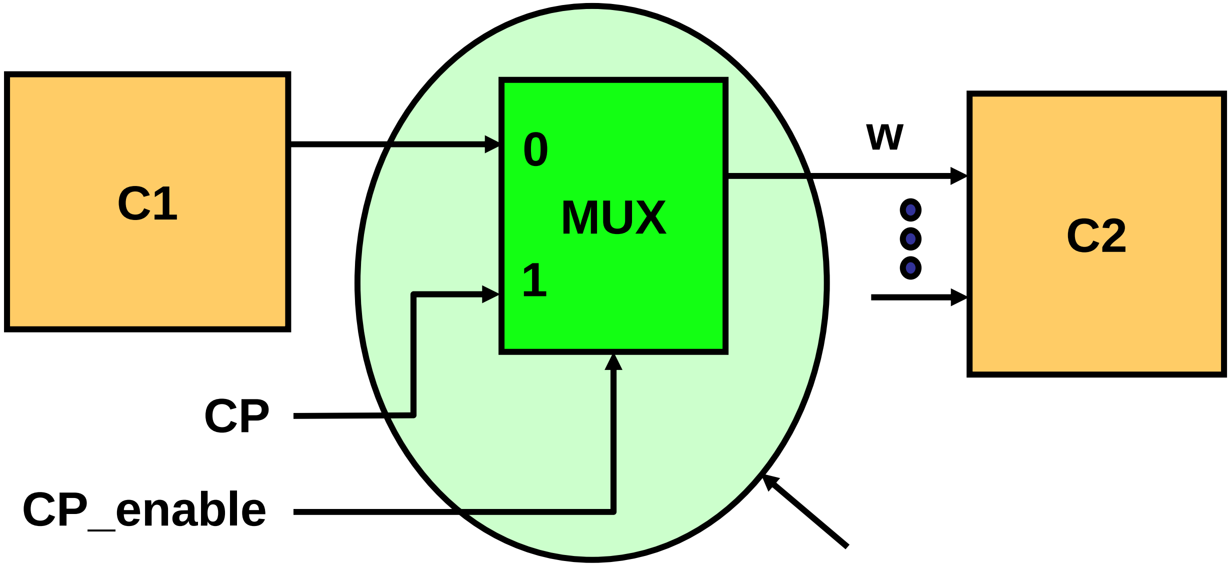


0/1 Injection Circuitry

- Normal operation

  When CP_enable = 0

- Inject 0
  - Set CP_enable = 1 and CP = 0
- Inject 1
  - Set CP_enable = 1 and CP = 1



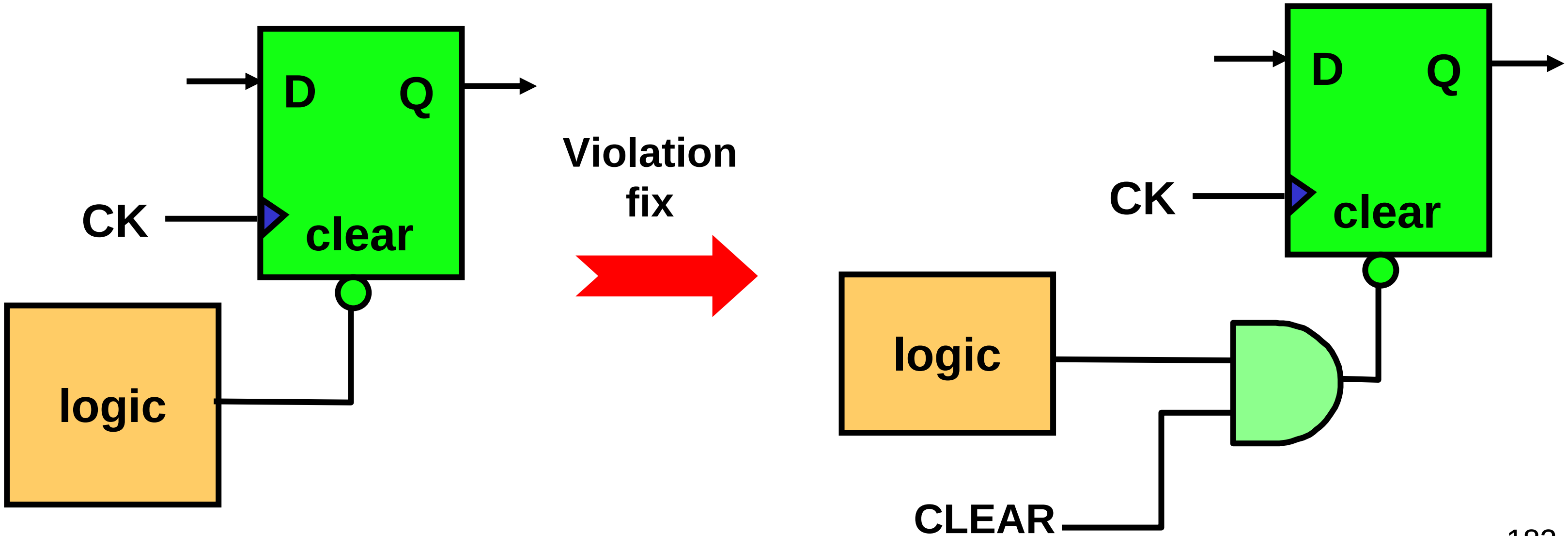
Inserted circuit for controlling line w

Control Point Selection

- Impact
  - The 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

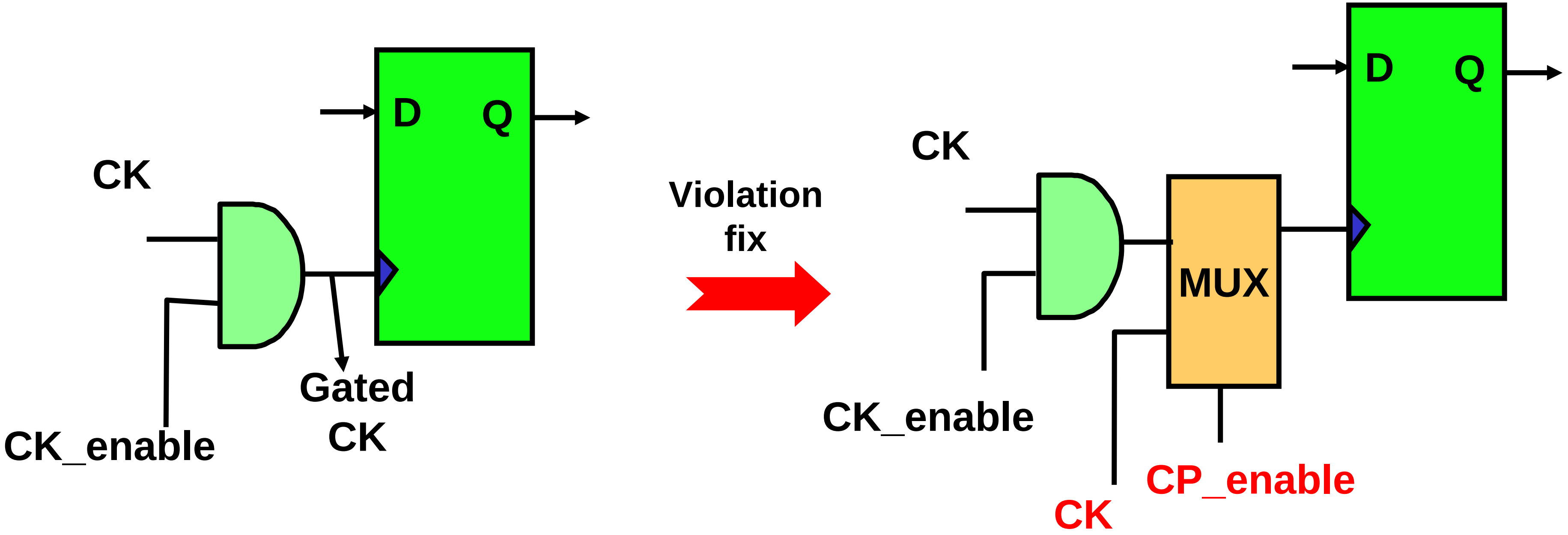
Example: Use CP to Fix DFT Rule Violation

- DFT rule violations
  - The set/clear signal of a flip-flop is generated by other logic, instead of directly controlled by an input pin
  - Gated clock signals
- Violation Fix
  - Add a control point to the set/clear signal or clock signals



Example: Fixing Gated Clock

- Gated Clocks
  - Advantage: power dissipation of a logic design can thus reduced
  - Drawback: the design's testability is also reduced
- Testability Fix

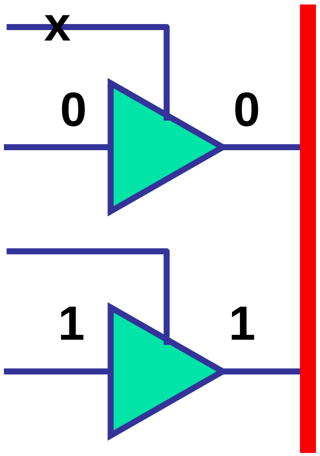


Example: Fixing Tri-State Bus Contention

- Bus Contention
  - **A stuck-at-fault at the tri-state enable line may cause bus contention – multiple active drivers connect to the bus simultaneously**
- Fix
  - **Add CP's to turn off tri-state devices during testing**

Enable line stuck-at-1

Enable line active



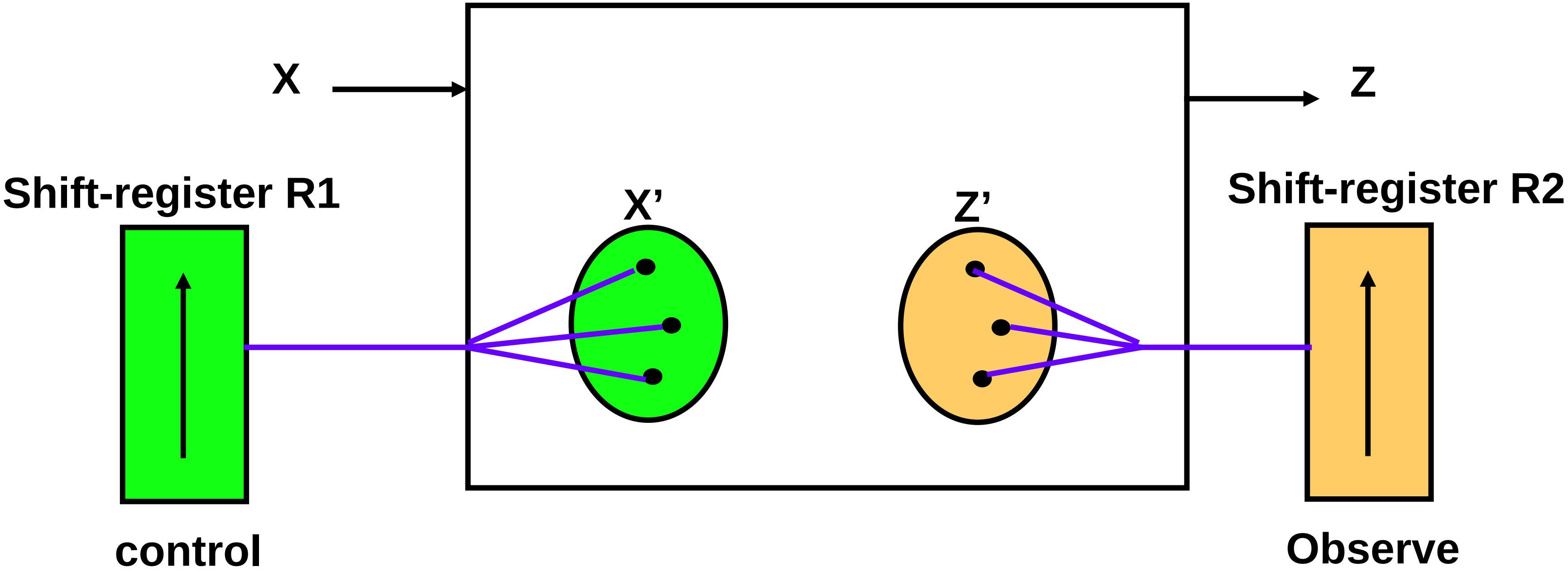
Unpredicted voltage on bus may cause fault to go unnoticed

Observation Point Selection

- Impact
  - The observability of the transitive fanins of the added point is improved
- Common choice
  - Stem lines having high fanout
  - 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 of CP & OP

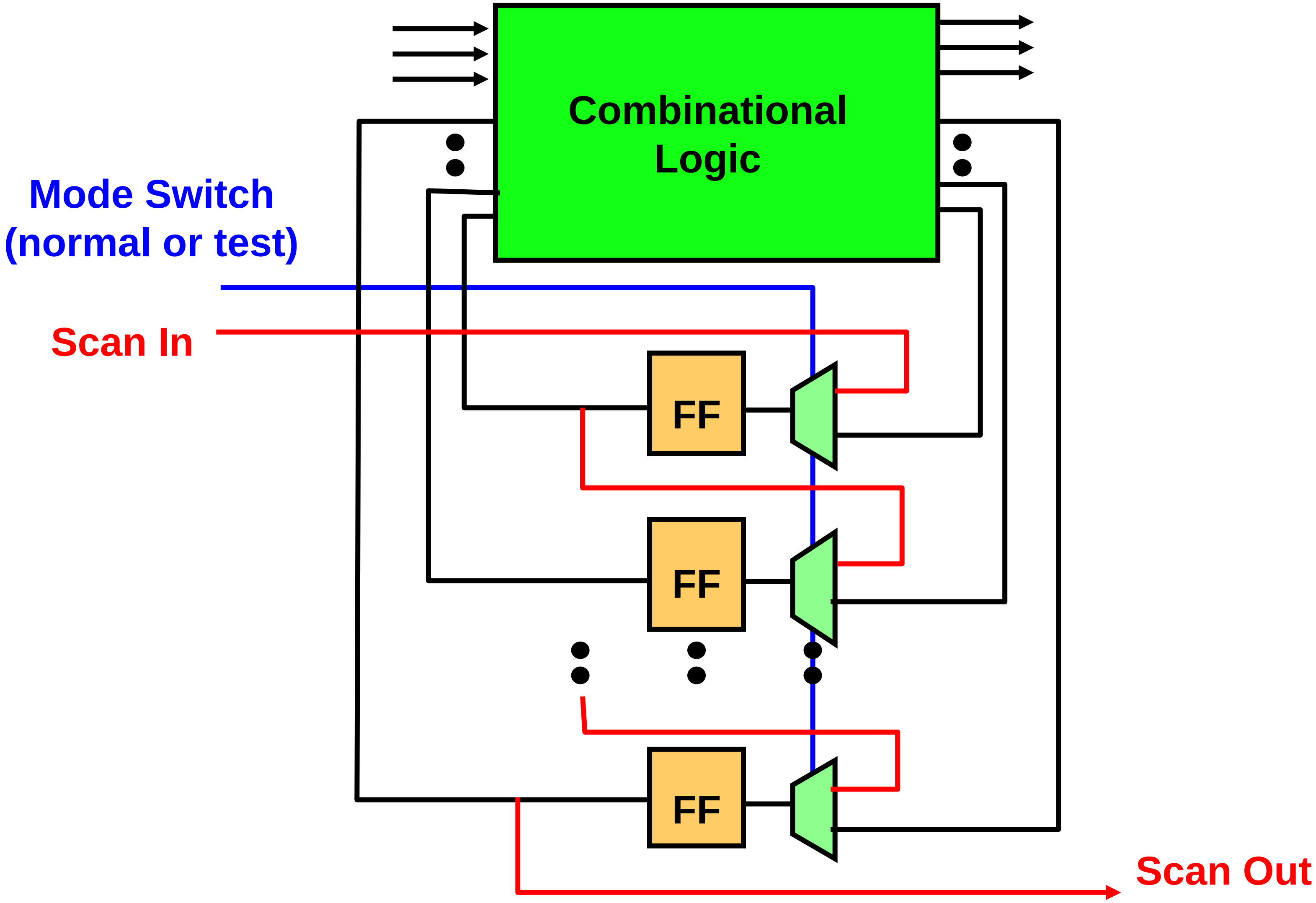
- Large number of I/O pins
  - Add MUX's to reduce the number of I/O pins
  - Serially shift CP values by shift-registers
- Larger test time



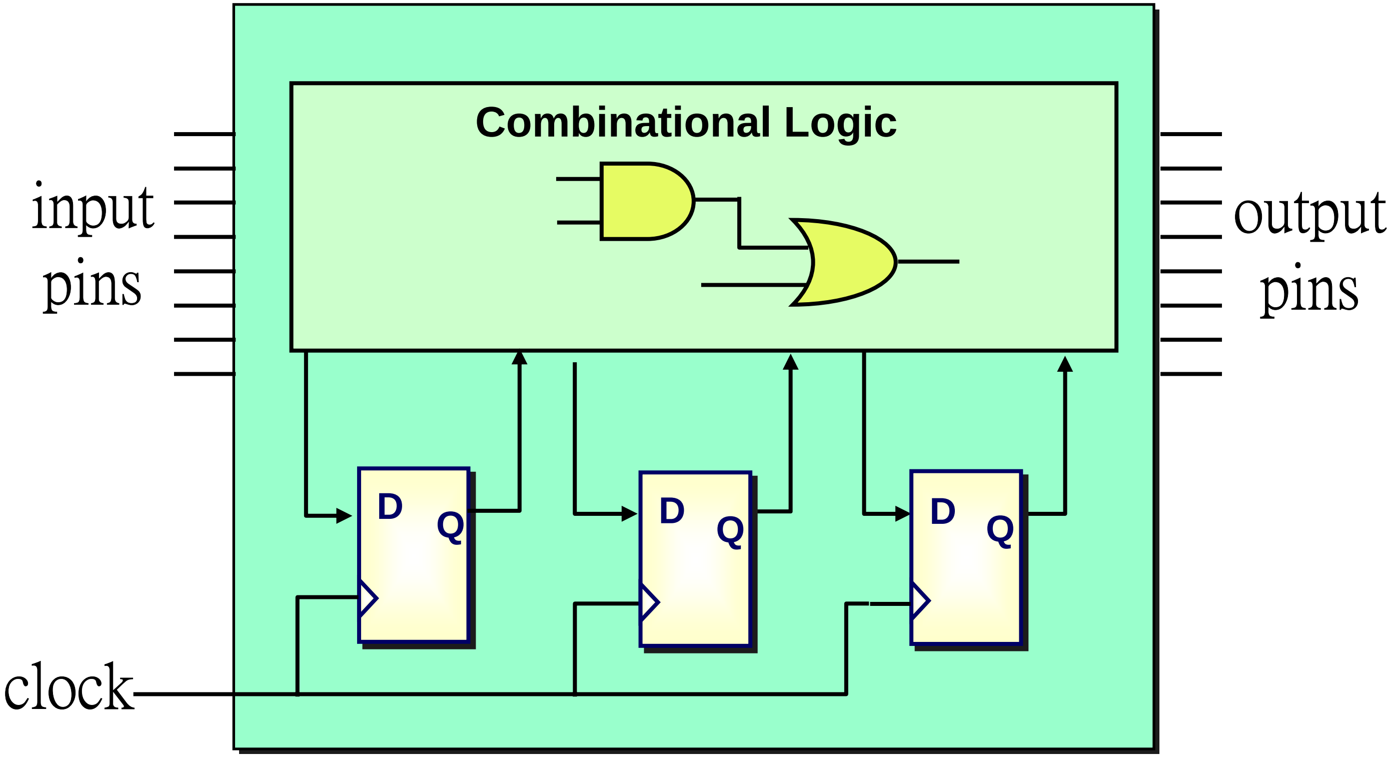
What Is Scan ?

- 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

The Scan Concept

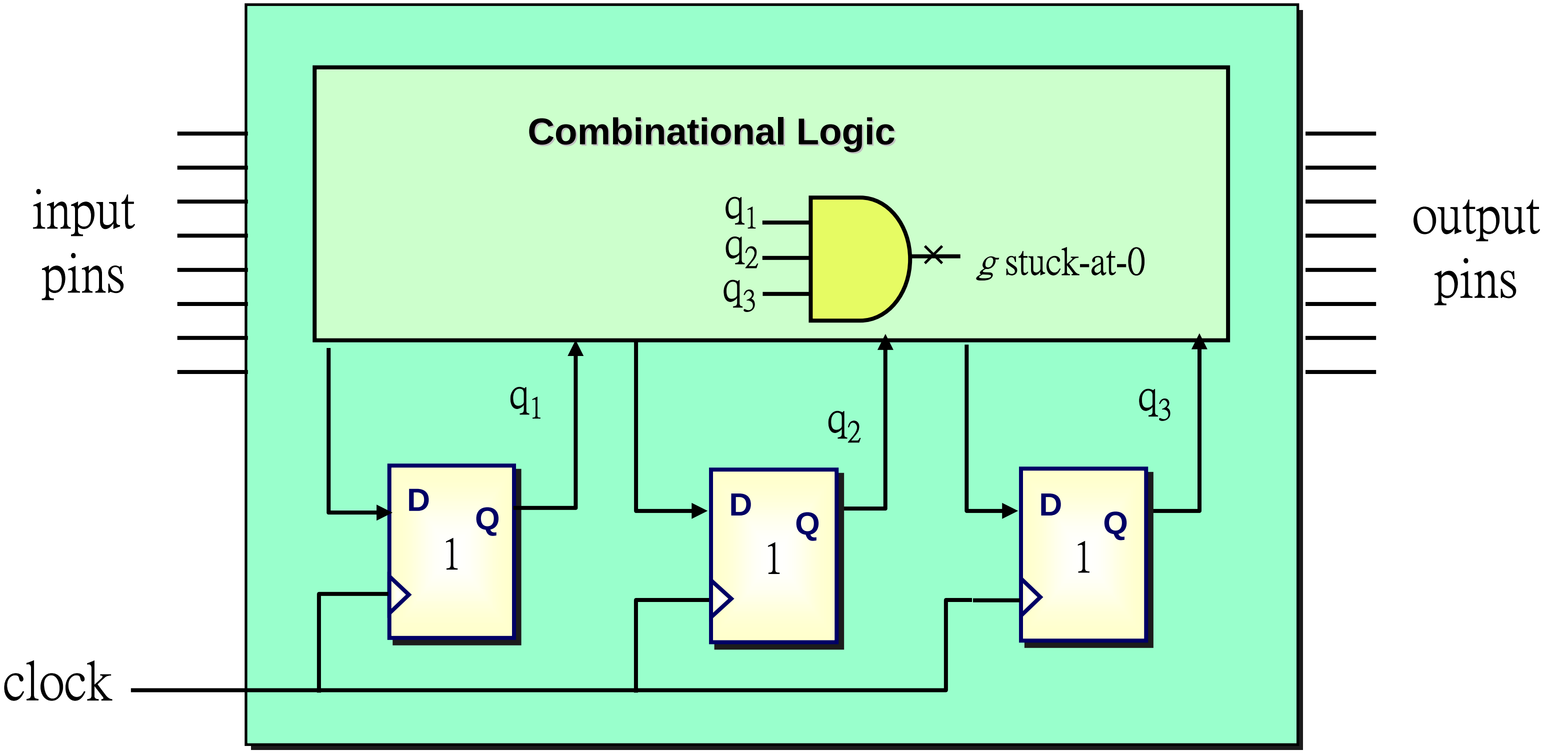


A Logic Design Before Scan Insertion



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

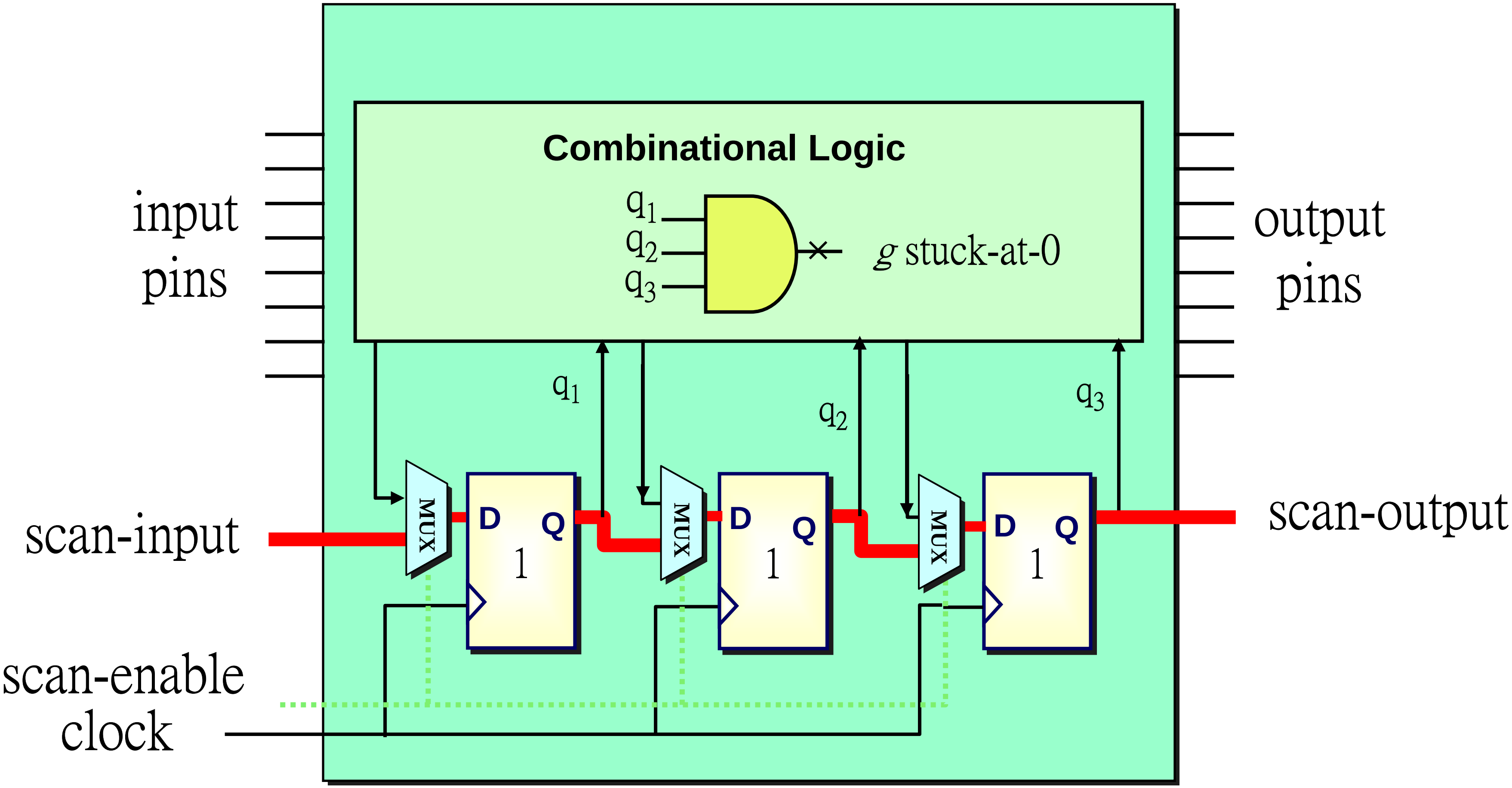
Example: A 3-stage Counter



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 !)

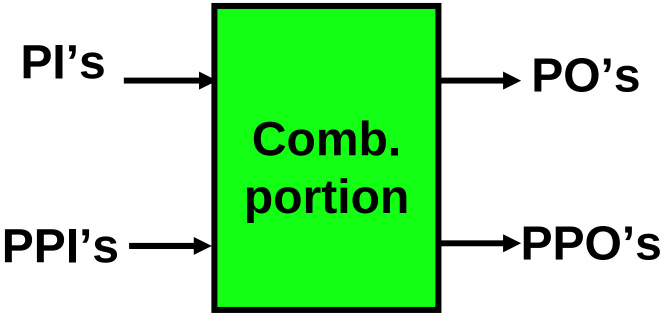
A Logic Design After Scan Insertion



Scan Chain provides an easy access to flip-flops
→ Pattern Generation is much easier !!

Procedure Of Applying Test Patterns

- Notation
  - Test vectors T = < t_i^I, t_i^F > i= 1, 2, …
  - Output Response R = < r_i^O, r_i^F > i= 1, 2, …
- Test Application
  - (1) i = 1;
  - (2) Scan-in t_1^F /* scan-in the first state vector for PPI's */
  - (3) Apply t_i^I /* apply current input vector at PI's */
  - (4) Observe r_i^O /* observe current output response at PO's */
  - (5) Parallelly load register /* load-in the next vector at PPO's */
    - (I.e., set Mode to 'Normal')
  - (6) Scan-out r_i^F while scanning-in t_{i+1}^F /* overlap scan-in and scan-out */
  - (7) i = i+1; Goto step (3)

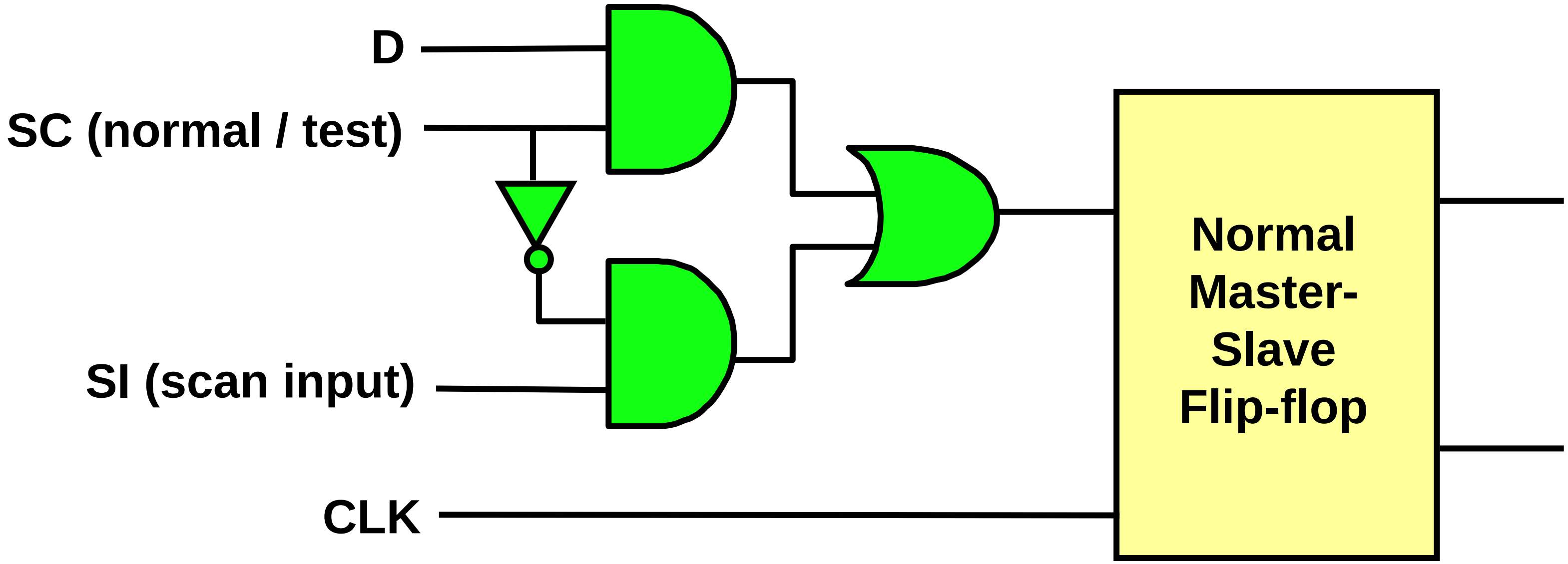


Testing Scan Chain ?

- Common practice
  - Scan chain is often first tested before testing core logic by pumping-in and pumping-out random vectors
- Procedure
  - (1) i = 0;
  - (2) Scan-in 1st random vector to flip-flops
  - (3) Scan-out (i)th random vector while scanning-in (i+1)th vector for flip-flops.
    - The (i)th scan-out vector should be identical to (i)th vector scanned in earlier, otherwise scan-chain is mal-functioning
  - (4) If necessary i = i+1, goto step (3)

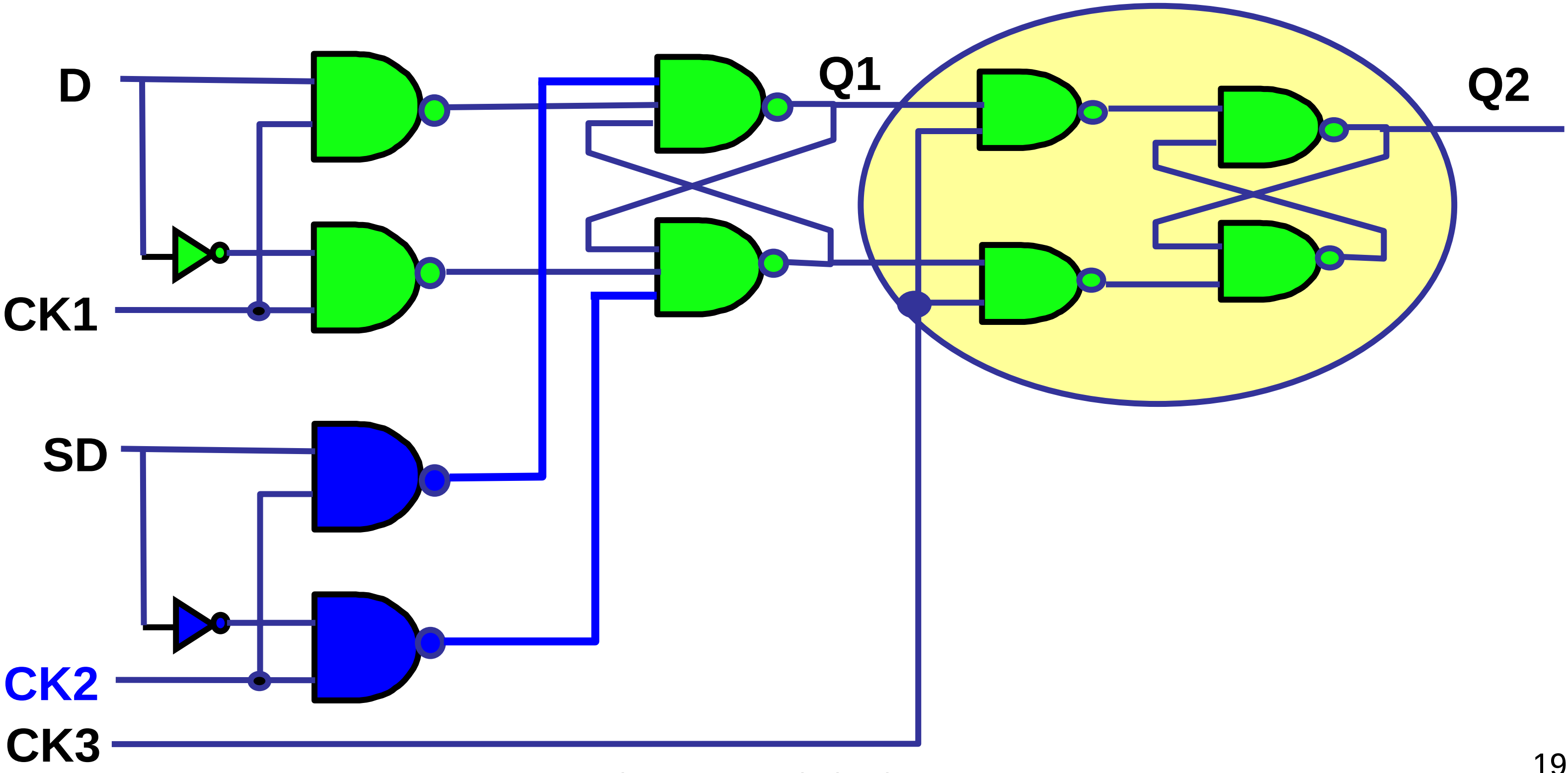
MUXed Scan Flip-Flop

- Only D-type master-slave flip-flops are used
- One extra primary input pin available for test
- All flip-flop clocks controlled from primary inputs
  - No gated clock allowed
- Clocks must not feed data inputs of flip-flops
- Most popularly supported in standard cell libraries

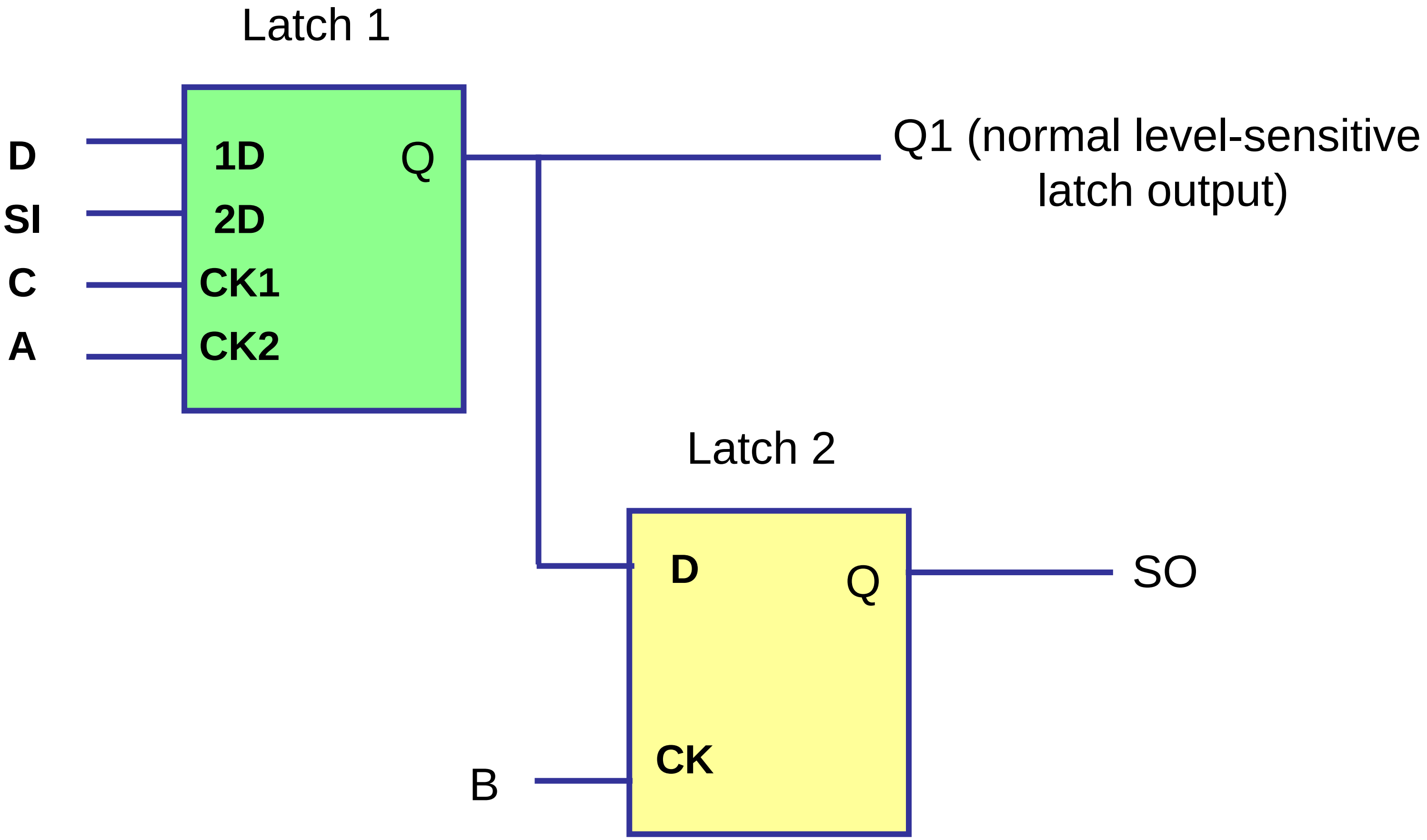


LSSD Scan flip-flop (1977 IBM)

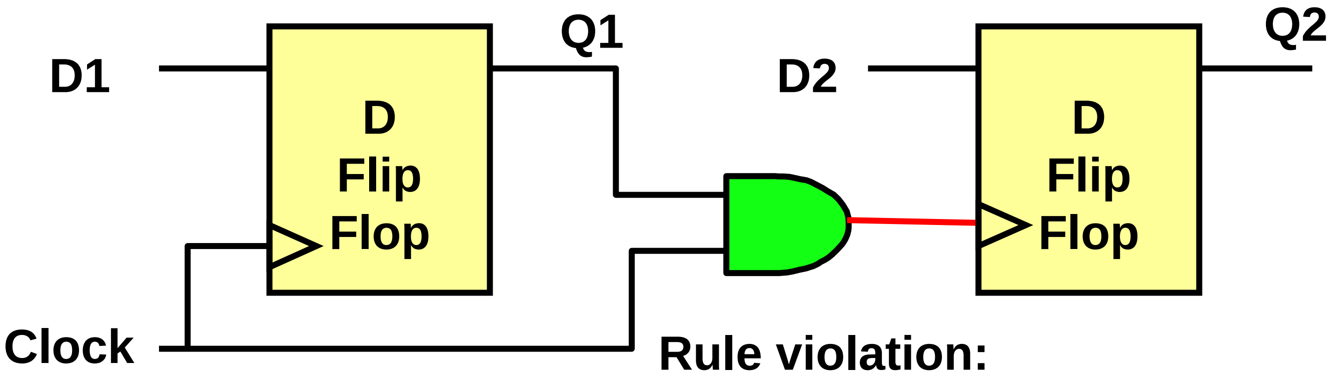
- LSSD: Level Sensitive Scan Design
  - Less performance degradation than MUXed scan FF
- Clocking
  - Normal operation: non-overlapping CK1=1 → CK3=1
  - Scan operation: non-overlapping CK2=1 → CK3=1



Symbol of LSSD Scan FF

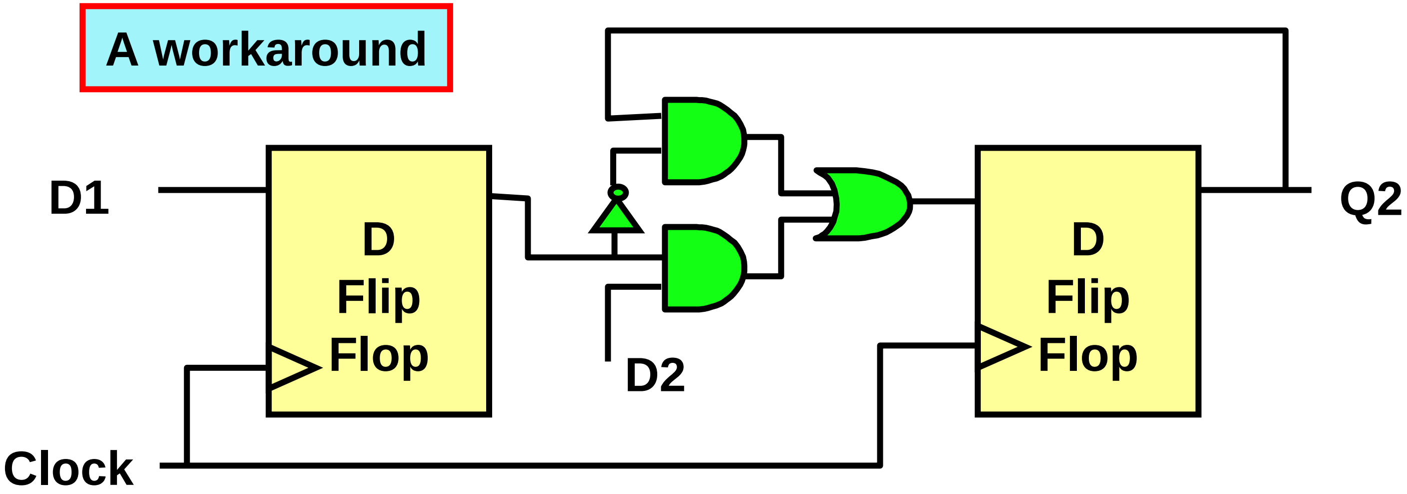


Scan Rule Violation Example



Rule violation:
Flip-flops cannot form a shift-register

A workaround



All FFs are triggered by the same clock edge
Set or reset signals are not controlled by any internal signals

Some Problems With Full Scan

Major Commercial Test Tool Companies
Synopsys
Mentor-Graphics
SynTest (華騰科技)
LogicVision

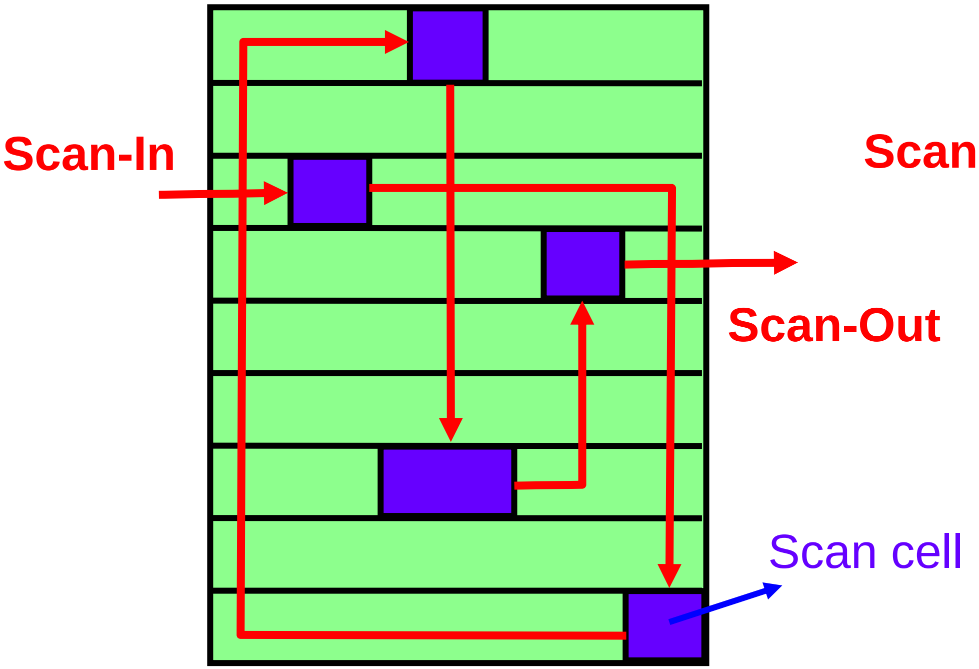
- Problems
  - Area overhead
  - Possible performance degradation
  - High test application time
  - Power dissipation
- Features of Commercial Tools
  - Scan-rule violation check (e.g., DFT rule check)
  - Scan insertion (convert a FF to its scan version)
  - ATPG (both combinational and sequential)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

Performance Overhead

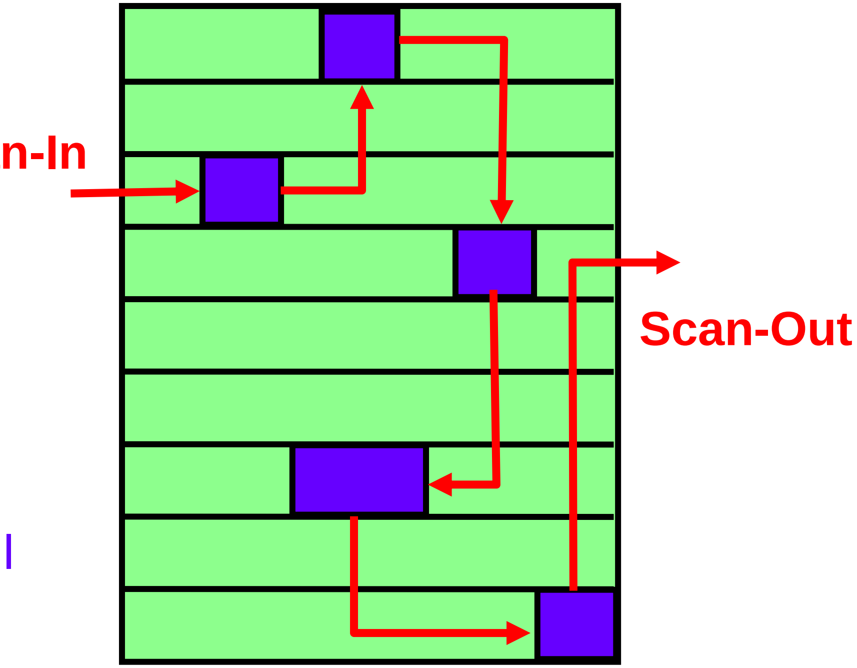
- The increase of delay along the normal data paths include:
  - Extra gate delay due to the multiplexer
  - Extra delay due to the capacitive loading of the scan-wiring at each flip-flop’s output
- Timing-driven partial scan
  - Try to avoid scan flip-flops that belong to the timing critical paths
  - The flip-flop selection algorithm for partial scan can take this into consideration to reduce the timing impact of scan to the design

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

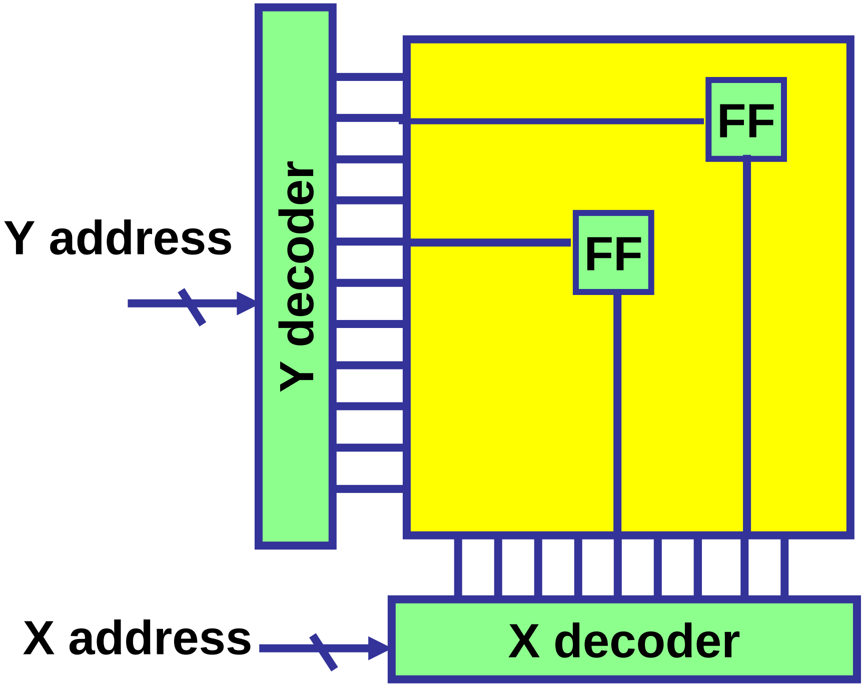
Overhead of Scan Design

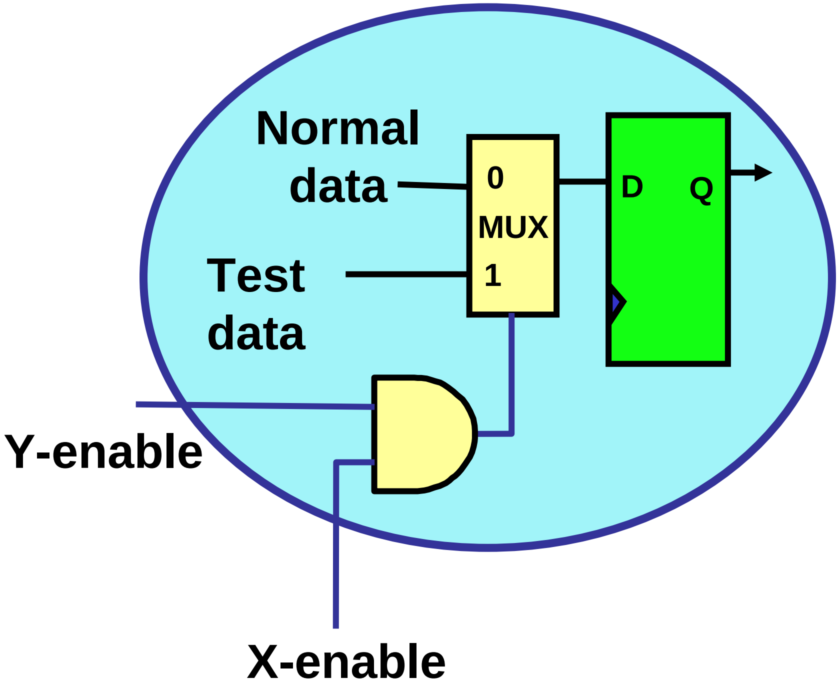
- No. of CMOS gates = 2000
- Fraction of flip-flops = 0.478
- Fraction of normal routing = 0.471

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

Random Access Scan

- Comparison with Scan-Chain
  - More flexible – any FF can be accessed in constant time
  - Test time could be reduced
  - More hardware and routing overhead

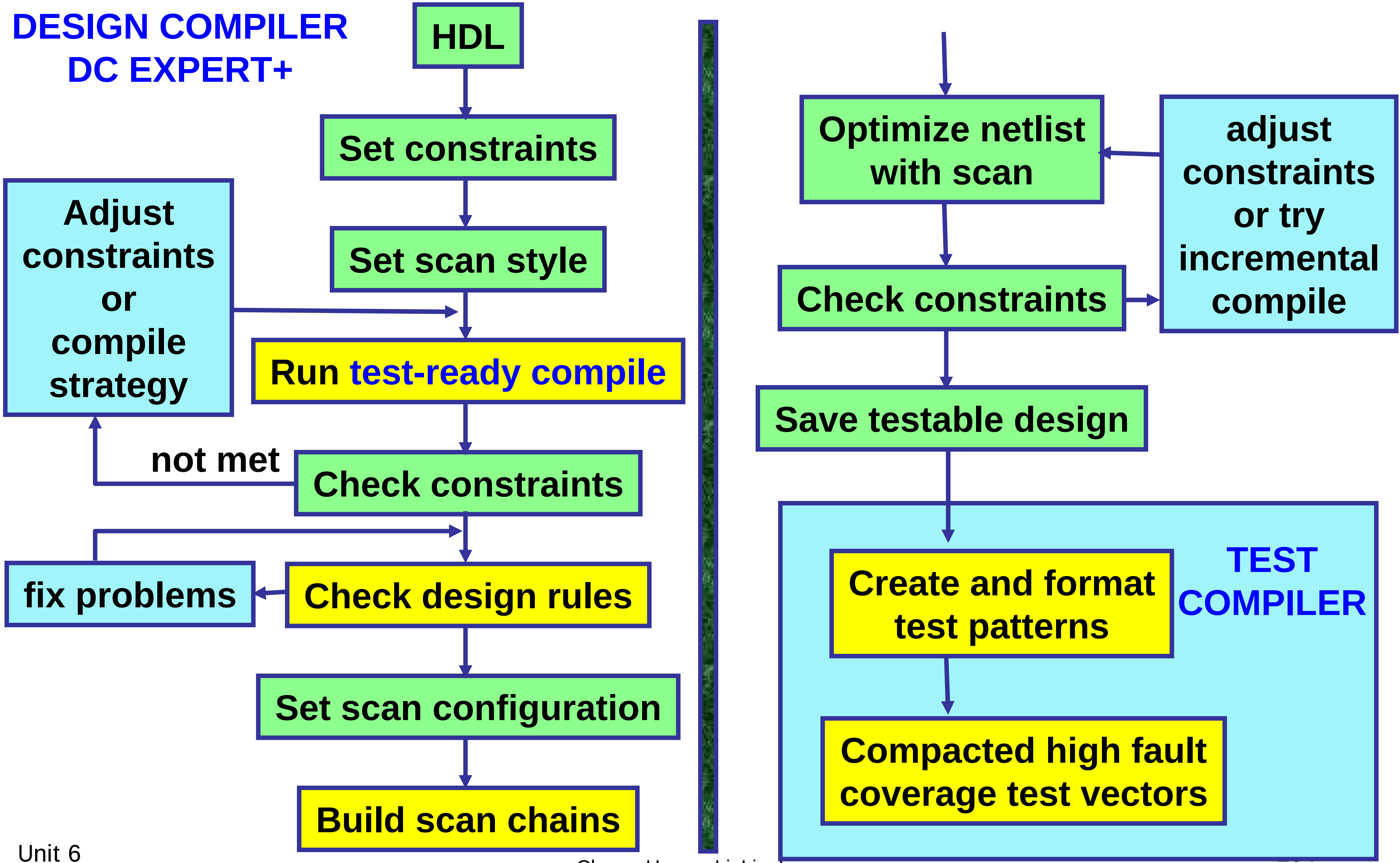




Parameters

- link_library
  - The ASIC vendor library where you design is initially represented
- target_library
  - Usually the same as your link_library, unless you are translating a design between technologies
- test_default_scan_style
  - multiplexed_flip_flop, clocked_scan, aux_clock_lssd
  - Note that: the cell library incorporated should support these scan cells

Typical Flat Design Flow



DESIGN COMPILER
DC EXPERT+

HDL

Set constraints

Set scan style

Adjust constraints or compile strategy

Run test-ready compile

not met

Check constraints

fix problems

Check design rules

Set scan configuration

Build scan chains

Optimize netlist with scan

adjust constraints or try incremental compile

Check constraints

Save testable design

TEST COMPILER

Create and format test patterns

Compacted high fault coverage test vectors

Synthesizing The Design

- Read in HDL code

```
dc_shell > read –format verilog design_name.v
```

- Set target_library

```
dc_shell > target_library = asic_vendor.db
```

- Constraint the design

```
dc_shell > max_area 1000
dc_shell > create_clock clock_port –period 20 -waveform {10,15}
```

- Declare the test methodology

```
dc_shell > set_scan_configuration –methodology partial_scan
```

- Select scan style

```
dc_shell > test_default_scan_style = multiplexed_flip_flop
```

- Compilation

```
dc_shell > compile -scan
```

Synthesizing The Design (con’t)

- Check constraints

```
dc_shell > report_constraint –all_violators
```

- Save design

```
dc_shell > write –format db –out design_test_ready.db
```

- Check design rules

```
dc_shell > check_test
```

Building Scan Chains

- Declare Scan Enable Signal

```
dc_shell > set_scan_signal test_scan_enable –port SE
dc_shell > set_drive 2 SE
```

- Set Min. Fault Coverage If Partial Scan

```
dc_shell > set_min_fault_coverage 95
```

- No Scan Replacement

```
dc_shell > set_scan_configuration –replace false
```

- Build the Scan Chains

```
dc_shell > insert_scan
```

- Check the Design Rules Again

```
dc_shell > check_test
```

Generating Test Patterns

- Generate Test Report

  dc_shell > report_test –scan_path

- Write out Scan Design

  dc_shell > write –format db –hierarchy –out design.db

- Generate The Test

  dc_shell > create_test_patterns –compact_effort high

- Write Out Test Patterns

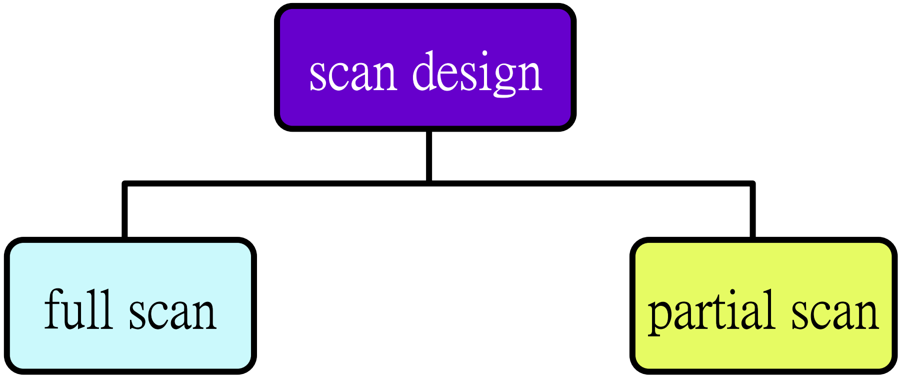
  dc_shell > write_test –out test_vectors –format WGL

- WGL (Waveform Generation Language)
  - Is a format supported by Summit Design Software

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

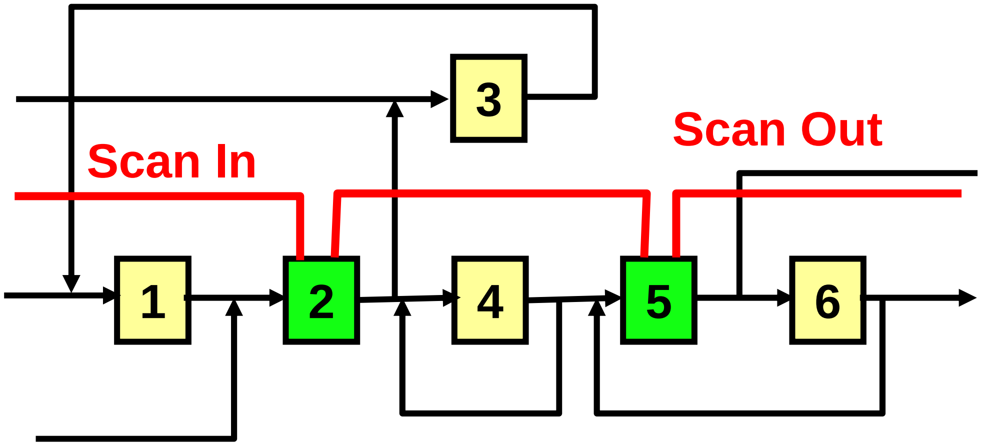


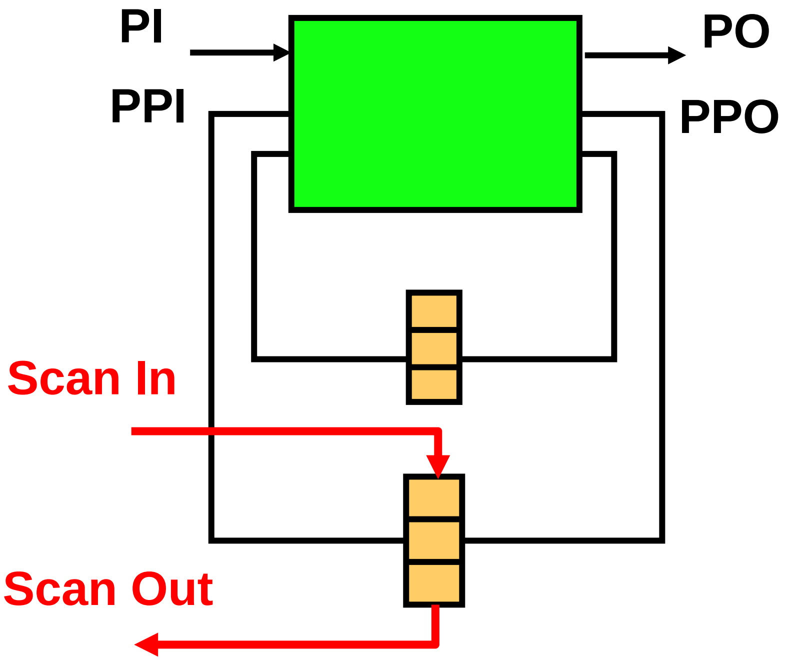
every flip-flop is a scan-FF

NOT every flip-flop is a scan-FF

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

Partial Scan Design





Scan Flip-Flops: {2, 5}
Non-Scan FFs: {1, 3, 4, 6}

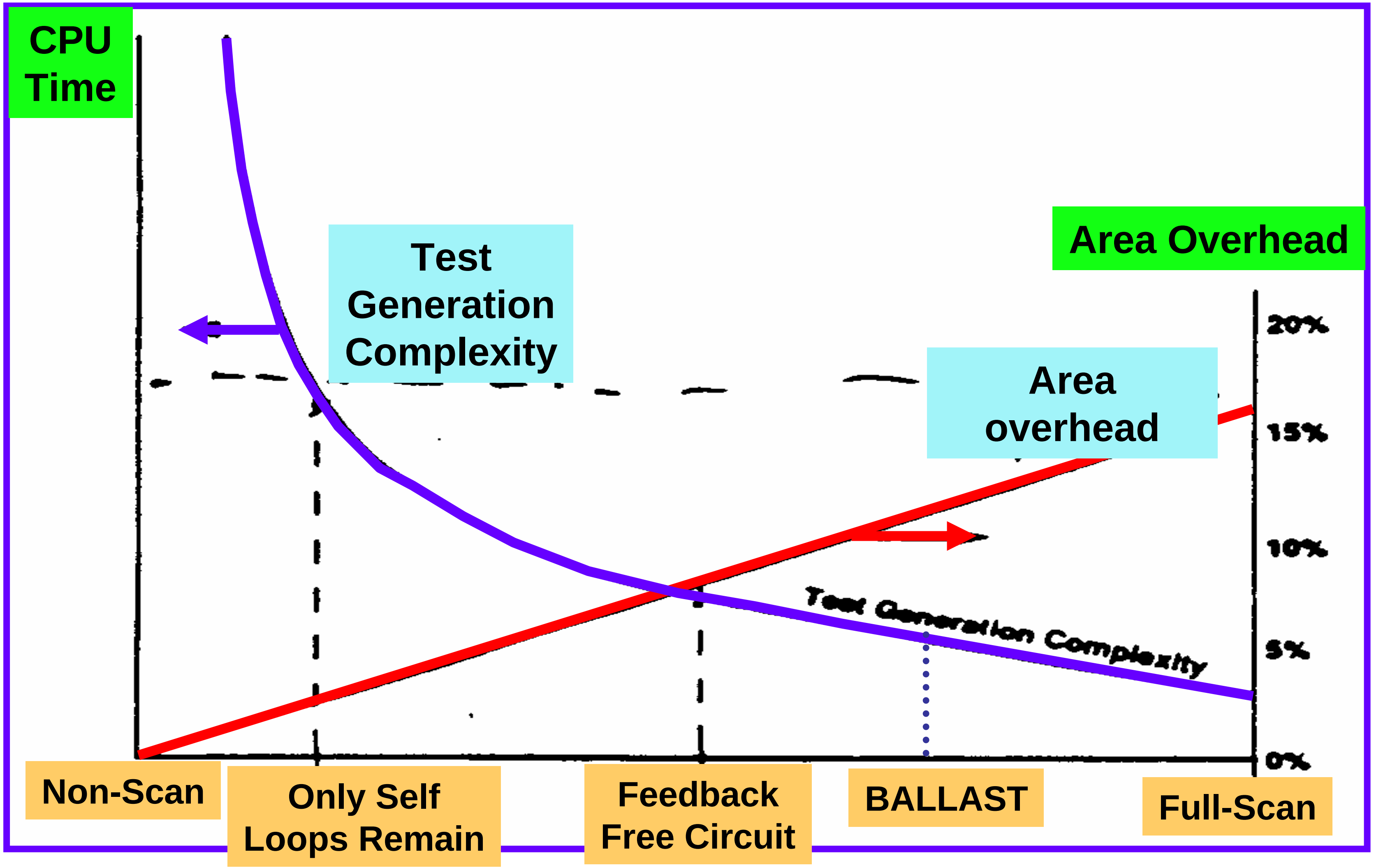
Cycle Breaking Algorithm

- Algorithm 1
  - Select nodes in a digraph G=(V,E) to cut all cycles of length greater than one
- Step 1
  - Find all the non-trivial SCCs $G_i=(V_i, E_i)$, where $1 \leqq i \leqq r$
  - If no non-trivial SCC stop
- Step 2
  - (1) Delete a node v using one of the heuristics H1, H2, H3 (to be discussed later)
  - (2) Delete node v from V_i and delete all incoming and outgoing edges of v
  - (3) Let the resulting graph be $G_i'=(V_i', E_i')$
    - Execute Algorithm 1 recursively with G_i' as input graph

Reducing the Length of Consecutive Self-Loop Paths

- Definition
  - A graph has a consecutive self-loop path of length K if there exists a directed path of exactly K vertices, each having a self-loop
- Observation
  - Circuits with consecutive self-loop paths, such as counters, need longer test generation time
- Solution
  - Reduce the length of consecutive self-loop paths by scanning some of the flip-flops with self-loops

Trade-Off of Area Overhead v.s. Test Generation 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
  - Automatic Test Pattern Generation
- Other Applications in CAD
  - ATPG is a way of Boolean Reasoning, applicable to may logic-domain CAD problems