Built-In Self-Test

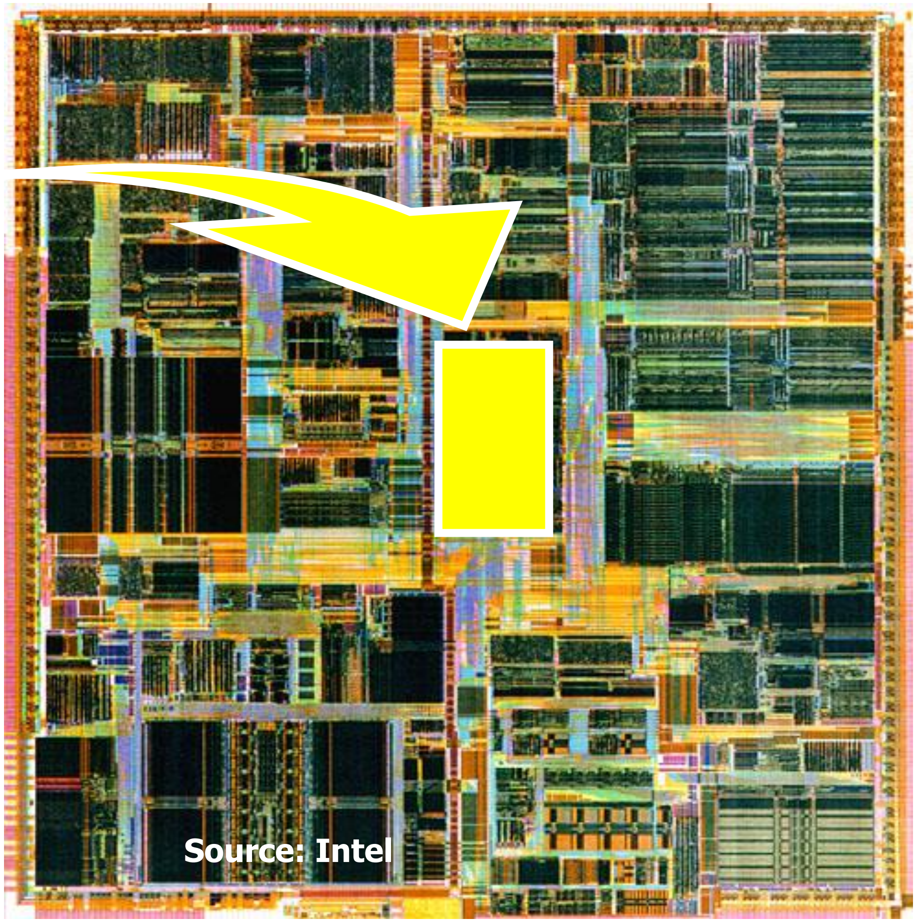
Outline

- **Motivation for BIST**
- **Testing SoC with BIST**
- **Test per Scan and Test per Clock**
- **HW and SW based BIST**
- **Exhaustive and pseudoexhaustive test generation**
- **Pseudorandom test generation with LFSR**
- **Hybrid BIST**
- **Response compaction methods**
- **Signature analyzers**

Testing Challenges: SoC Test

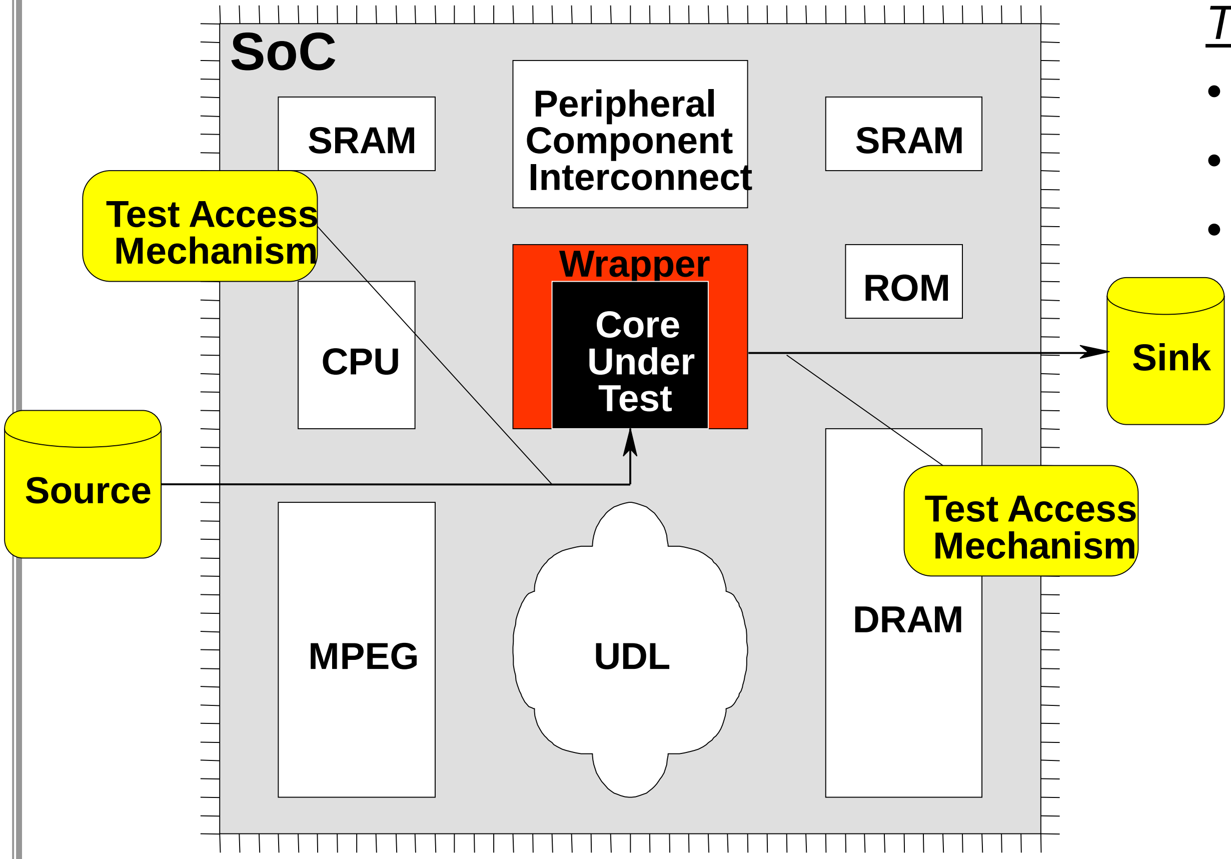
Source: Elcoteq

Cores have to be tested on chip



Source: Intel

Self-Test in Complex Digital Systems



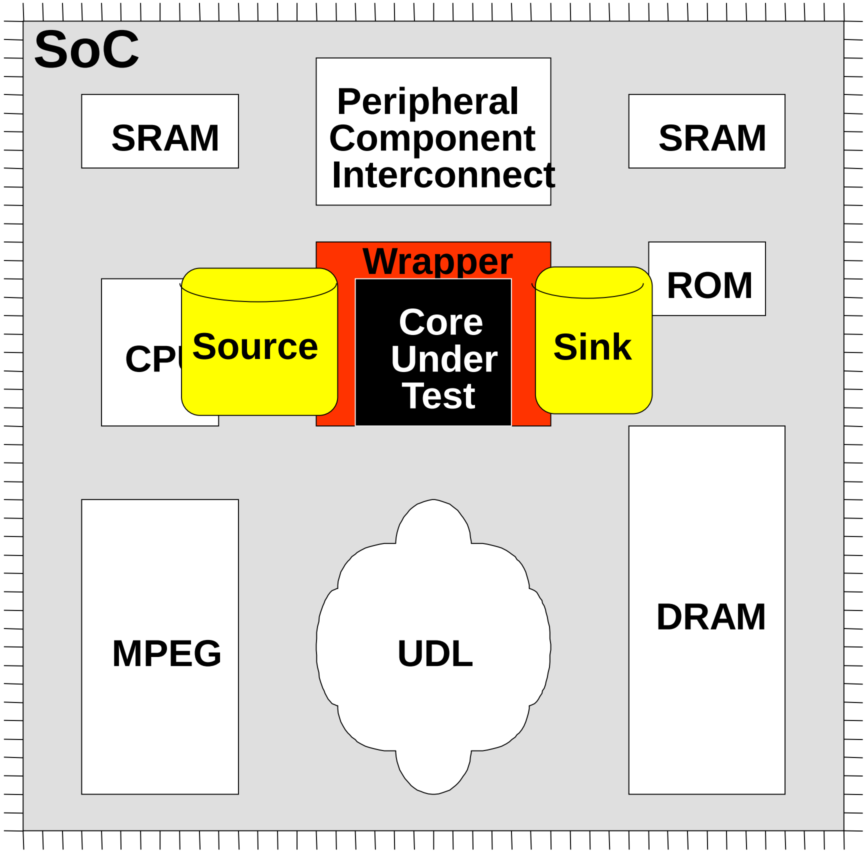
Test architecture components:

- Test pattern source & sink
- Test Access Mechanism
- Core test wrapper

Solutions:

- Off-chip solution
  - need for external ATE
- Combined solution
  - mostly on-chip, ATE needed for control
- On-chip solution
  - BIST

Self-Test in Complex Digital Systems



Test architecture components:

- Test pattern source & sink
- Test Access Mechanism
- Core test wrapper

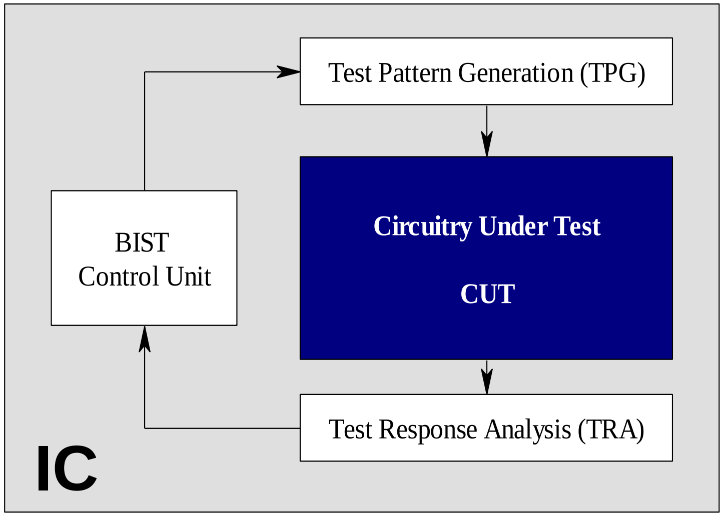
Solutions:

- Off-chip solution
  - need for external ATE
- Combined solution
  - mostly on-chip, ATE needed for control
- On-chip solution
  - BIST

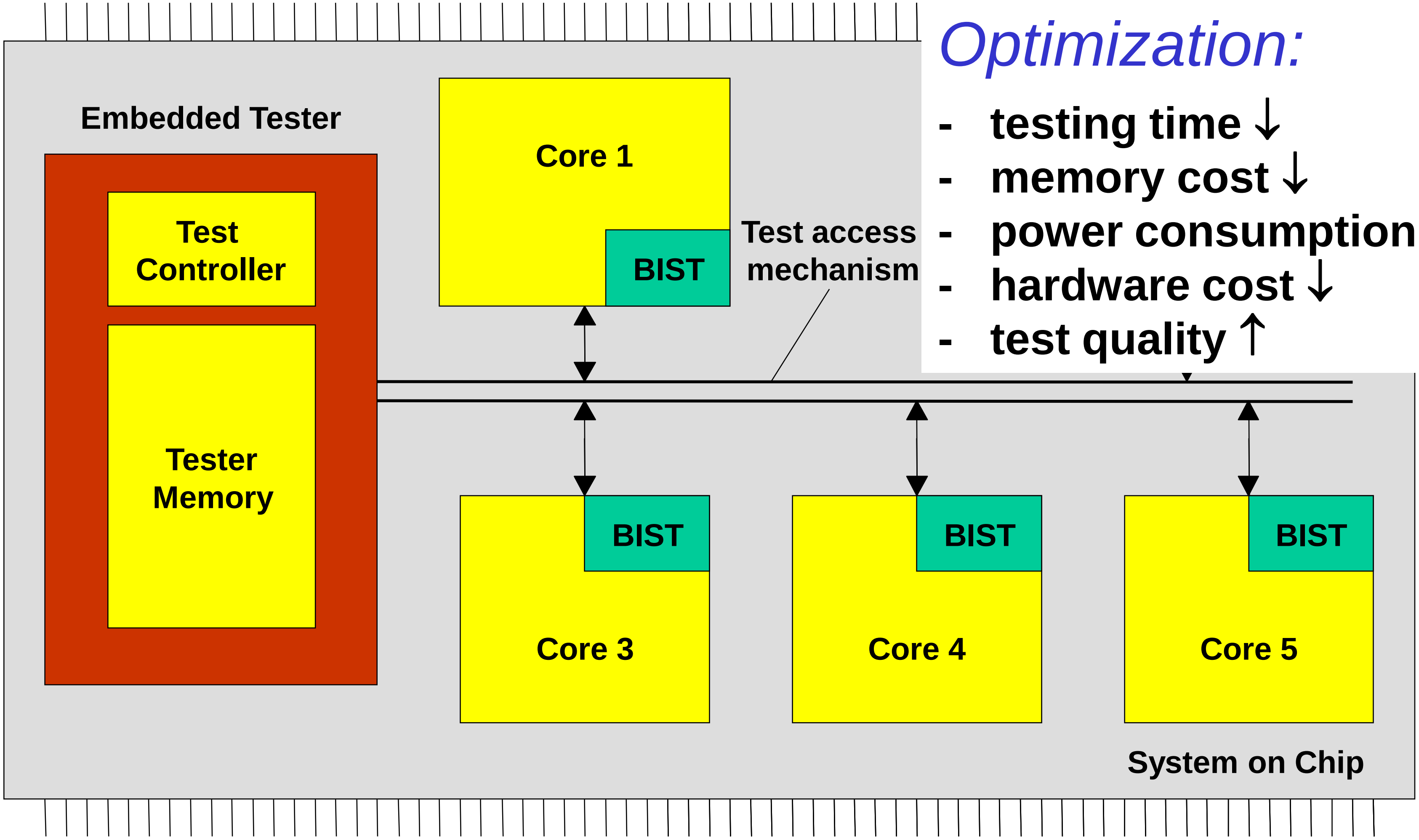
What is BIST

- On circuit
  - Test pattern generation
  - Response verification
- Random pattern generation,

  very long tests
- Response compression



SoC BIST



Optimization:

- testing time ↓
- memory cost ↓
- power consumption ↓
- hardware cost ↓
- test quality ↑

Built-In Self-Test

- **Motivations for BIST:**
  - Need for a cost-efficient testing (general motivation)
  - Doubts about the stuck-at fault model
  - Increasing difficulties with TPG (Test Pattern Generation)
  - Growing volume of test pattern data
  - Cost of ATE (Automatic Test Equipment)
  - Test application time
  - Gap between tester and UUT (Unit Under Test) speeds
- **Drawbacks of BIST:**
  - Additional pins and silicon area needed
  - Decreased reliability due to increased silicon area
  - Performance impact due to additional circuitry
  - Additional design time and cost

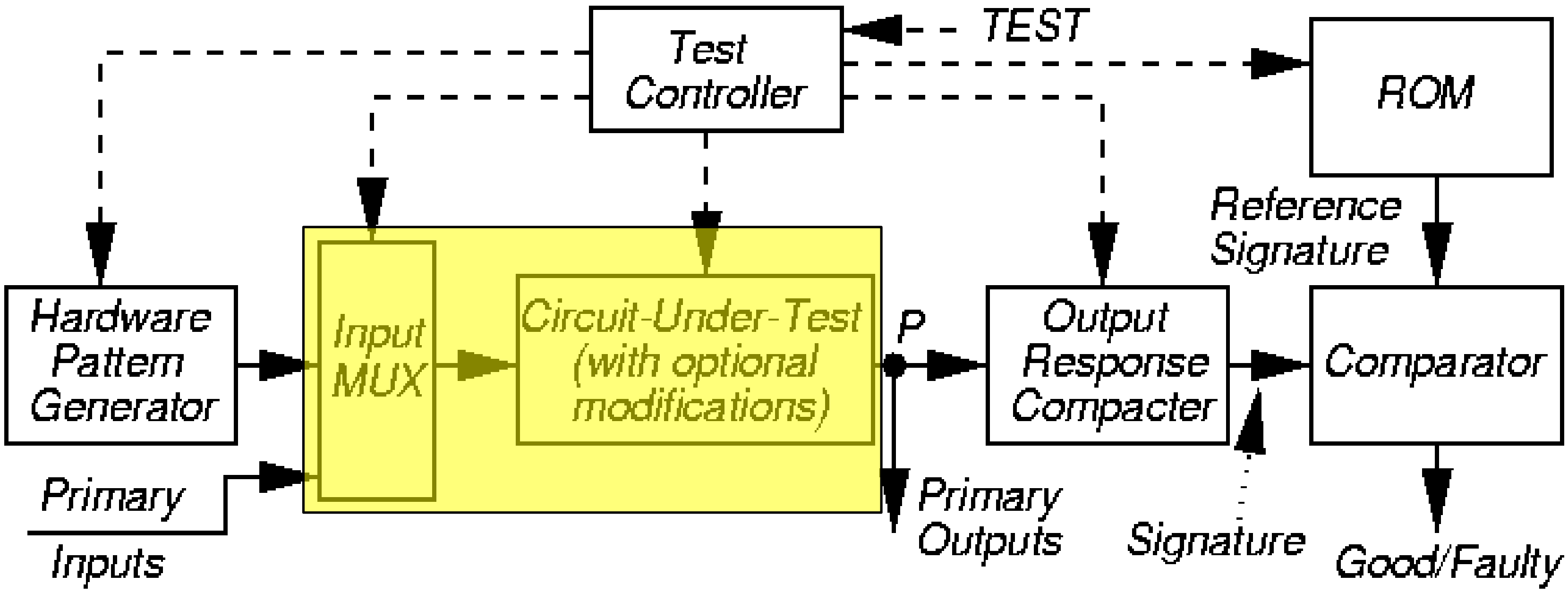
BIST in Maintenance and Repair

- Useful for field test and diagnosis (less expensive than a local automatic test equipment)
- To overcome the disadvantages of software tests for field test and diagnosis (nonBIST):
  - Low hardware fault coverage
  - Low diagnostic resolution
  - Slow to operate
- Hardware BIST benefits:
  - Lower system test effort
  - Improved system maintenance and repair
  - Improved component repair
  - Better diagnosis
  - Possibility to use the functionality of microprocessors

BIST Techniques

- **BIST techniques are classified:**
  - ***on-line* BIST** - includes concurrent and nonconcurrent techniques
  - ***off-line* BIST** - includes functional and structural approaches
- ***On-line* BIST** - testing occurs during normal functional operation
  - ***Concurrent on-line* BIST** - testing occurs simultaneously with normal operation mode, usually **coding techniques or duplication** and comparison are used
  - ***Nonconcurrent on-line* BIST** - testing is carried out while a system is in an **idle** state, often by executing *diagnostic software* or *firmware routines*
- ***Off-line* BIST** - system is not in its normal working mode, usually on-chip test generators and output response analyzers or microdiagnostic routines
  - ***Functional off-line* BIST** is based on a functional description of the Component Under Test (CUT) and uses functional high-level fault models
  - ***Structural off-line* BIST** is based on the structure of the CUT and uses structural fault models (e.g. SAF)

Detailed BIST Architecture



Source: VLSI Test: Bushnell-Agrawal

BIST: Test Generation Methods

Universal test sets

1. **Exhaustive test (trivial test)**
2. **Pseudo-exhaustive test**

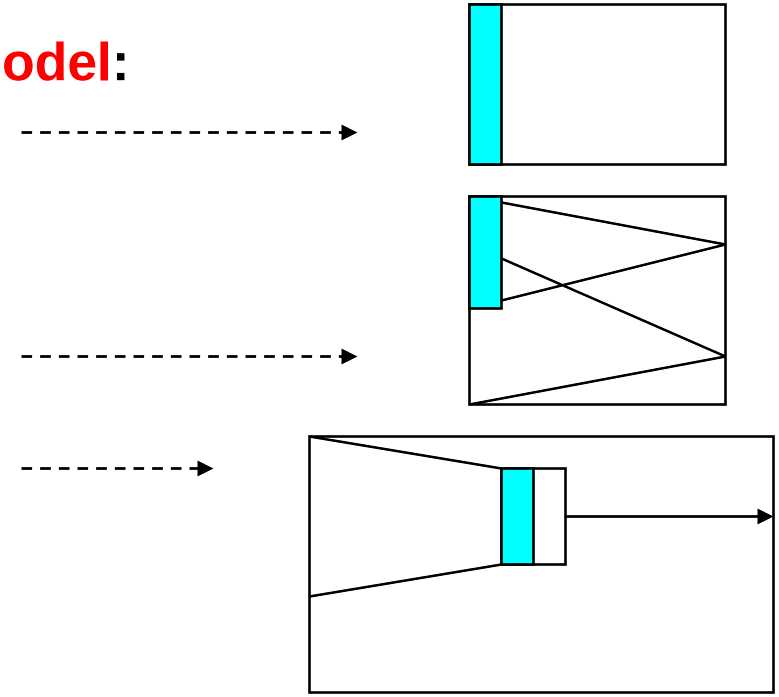
Properties of exhaustive tests

1. **Advantages** (concerning the stuck at fault model):
   - **test pattern generation is not needed**
   - **fault simulation is not needed**
   - **no need for a fault model**
   - **redundancy problem is eliminated**
   - **single and multiple stuck-at fault coverage is 100%**
   - **easily generated on-line by hardware**
2. **Shortcomings:**
   - **long test length (2^n patterns are needed, n - is the number of inputs)**
   - **CMOS stuck-open fault problem**

Exhaustive and Pseudo-Exhaustive Testing

Exhaustive combinational fault model:

- **exhaustive test patterns**
- **pseudoexhaustive test patterns**
  - **exhaustive output line oriented test patterns**
  - **exhaustive module oriented test patterns**



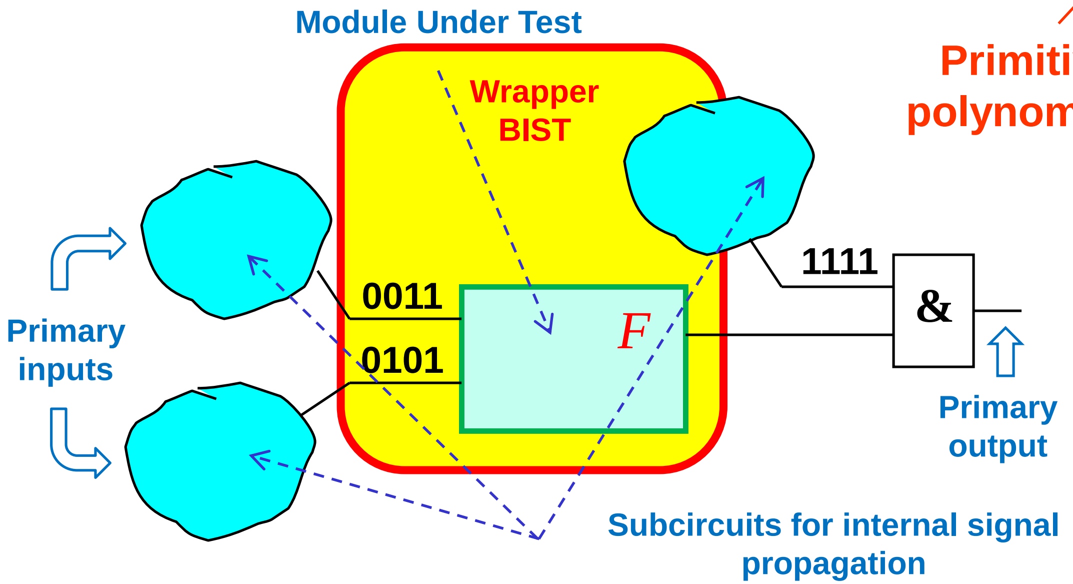
BIST: Pseudoexhaustive Testing

Pseudo-exhaustive test sets:

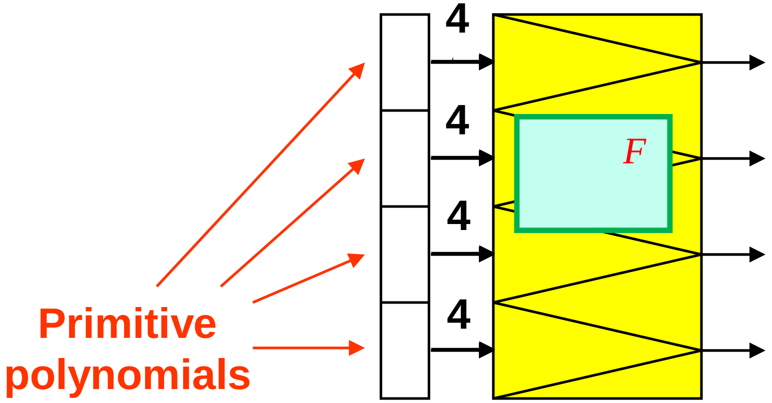
Output function verification

- maximal parallel testability
- partial parallel testability

Module function verification



Output function verification



$2^{16} = 65536$ >> 4x16 = 64 > 16

Exhaustive test	Pseudo-exhaustive sequential	Pseudo-exhaustive parallel

Testing ripple-carry adder

Output function verification (maximum parallelity)

Exhaustive test generation for n-bit adder:

Good news:
Bit number n - arbitrary
Test length - always 8 (!)

Bad news:
The method is correct only for ripple-carry adder

	c_0	a_0	b_0	c_1	a_1	b_1	c_2	a_2	b_2	c_3	…
1	0	0	0	0	0	0	0	0	0	0	
2	0	0	1	0	0	1	0	0	1	0	
3	0	1	0	0	1	0	0	1	0	0	
4	0	1	1	1	0	0	0	1	1	1	
5	1	0	0	0	1	1	1	0	0	0	
6	1	0	1	1	0	1	1	0	1	1	
7	1	1	0	1	1	0	1	1	0	1	
8	1	1	1	1	1	1	1	1	1	1	

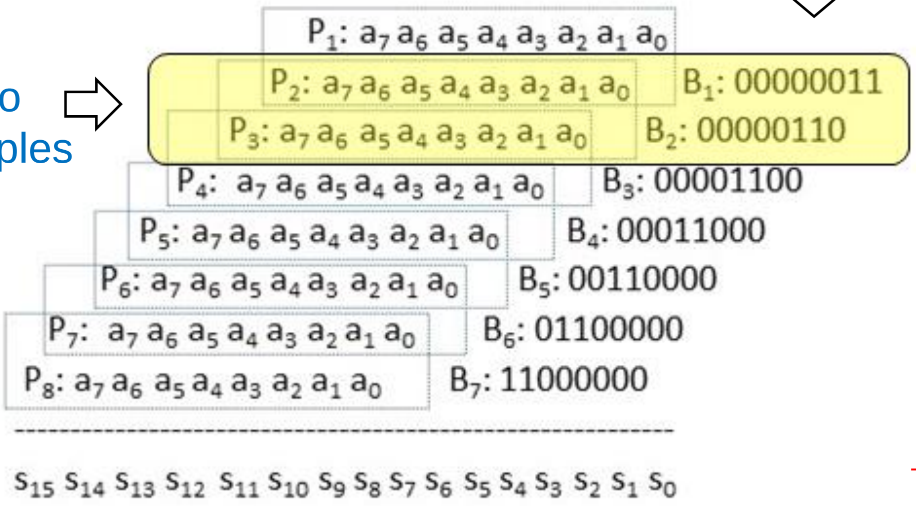
0-bit testing **1-bit testing** **2-bit testing** **3-bit testing** **… etc**

Pseudo-Exhaustive Test for Multiplier

Selectable multiplicands

Multipliers

Two examples



P_1: $a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$

P_2: $a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$ B_1: 00000011

P_3: $a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$ B_2: 00000110

P_4: $a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$ B_3: 00001100

P_5: $a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$ B_4: 00011000

P_6: $a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$ B_5: 00110000

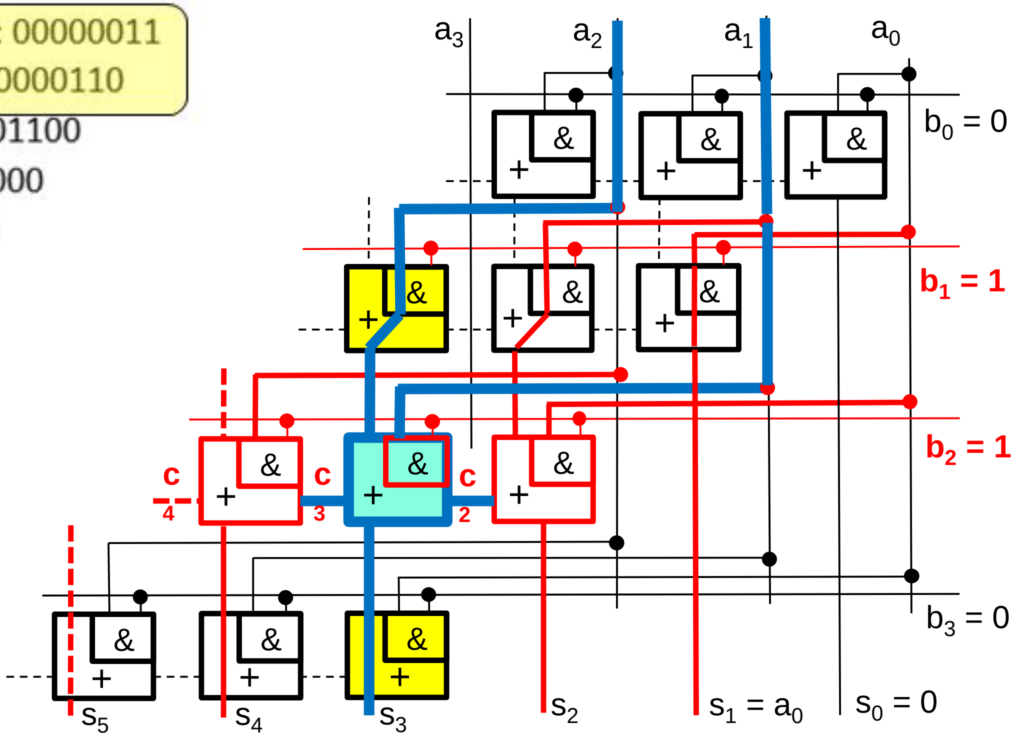
P_7: $a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$ B_6: 01100000

P_8: $a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$ B_7: 11000000

$s_{15}\ s_{14}\ s_{13}\ s_{12}\ s_{11}\ s_{10}\ s_9\ s_8\ s_7\ s_6\ s_5\ s_4\ s_3\ s_2\ s_1\ s_0$

Multiplication with traditional “*paper and pencil*” method

Multiplier array



Pseudo-Exhaustive Test for Multiplier

Replication of columns with pseudo-exhaustive patterns for

Adder ⇨

No	…	4-bit	3-bit	2-bit	1-bit	0-bit
		a_4 b_4 c_4	a_3 b_3 c_3	a_2 b_2 c_2	a_1 b_1 c_1	a_0 b_0
1	…	0 0 0	0 0 0	0 0 0	0 0 0	0 0
2	…	0 1 0	0 1 0	0 1 0	0 1 0	0 1
3	…	1 0 0	1 0 0	1 0 0	1 0 0	1 0
4	…	1 1 0	0 0 1	1 1 0	0 0 1	1 1
5	…	0 0 1	1 1 0	0 0 1	1 1 0	0 0
6	…	0 1 1	0 1 1	0 1 1	0 1 1	1 1
7	…	1 0 1	1 0 1	1 0 1	1 0 1	1 1
8	…	1 1 1	1 1 1	1 1 1	1 1 1	1 1

carry multiplier array

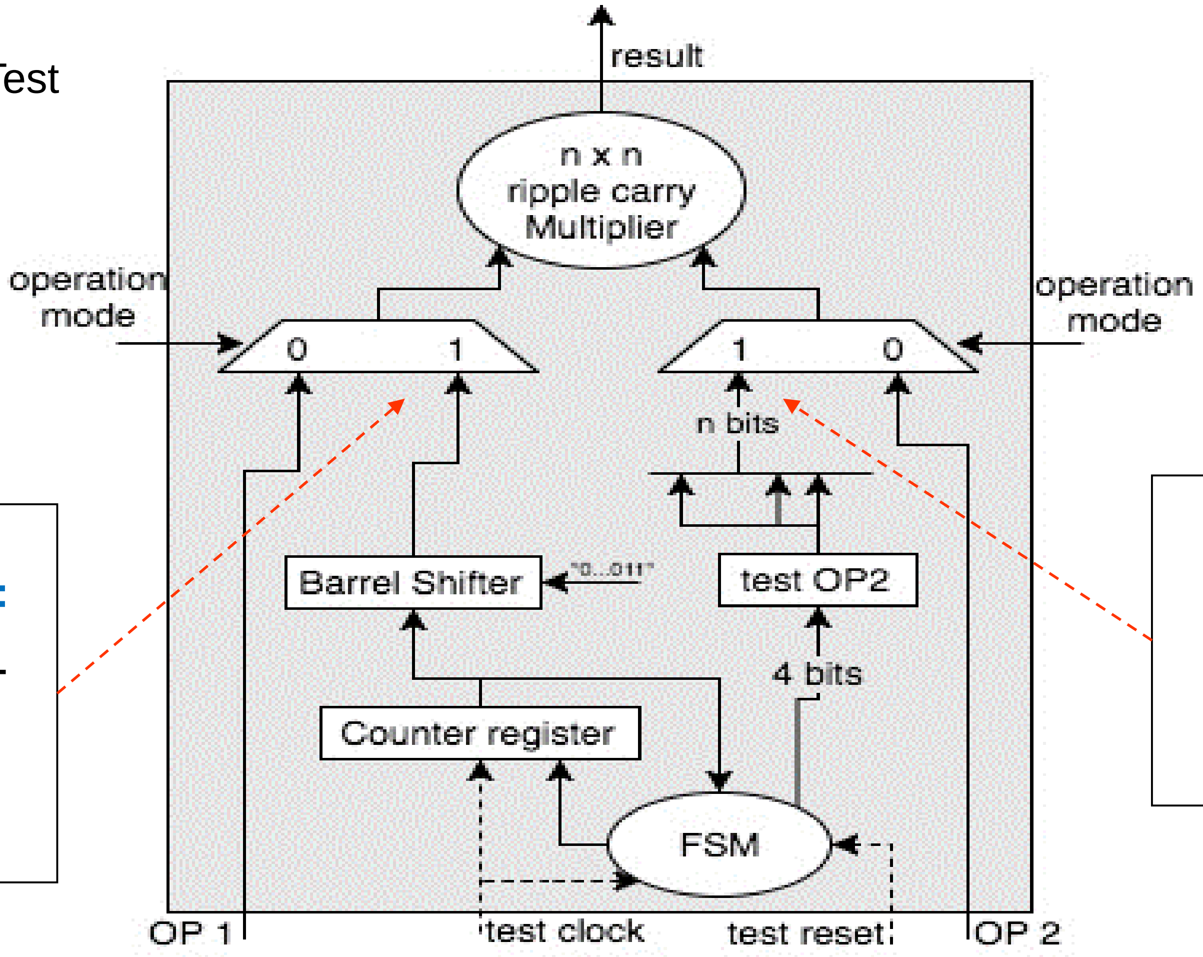
This table is replicated and all replications are repeated for all shifted **b = (…11…)**

Multiplier ⇩

N	6-bit	5-bit	4-bit	3-bit	2-bit	1-bit	0-bit
	$c_6a_7a_6$	$c_5a_6a_5$	$c_4a_5a_4$	$c_3a_4a_3$	$c_2a_3a_2$	$c_1a_2a_1$	a_1a_0
1	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0
2	0 1 0	0 0 1	0 1 0	0 0 1	0 1 0	0 0 1	1 0
3	0 0 1	0 1 0	0 0 1	0 1 0	0 0 1	0 1 0	0 1
4	1 0 1	0 1 1	0 1 0	1 0 0	1 0 1	0 1 1	1 0
5	1 1 0	1 0 1	1 1 0	1 0 1	1 1 0	1 0 1	1 1
6	1 0 1	1 1 1	1 1 1	1 1 0	1 0 1	1 1 1	1 1
7	0 1 1	0 1 0	1 0 0	1 0 1	0 1 1	0 1 0	0 0
8	1 0 0	1 0 1	0 1 1	0 1 0	1 0 0	1 0 1	1 1
9	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1
10	0 1 0	1 0 0	1 0 1	0 1 1	0 1 0	0 1 0	1 0
11	1 1 1	1 1 0	1 0 1	1 1 1	1 1 1	1 1 1	1 1

Exhaustively Self-Testing Multiplier

BIST
Built-in Self-Test



Multiplier operands:

Shifted 11

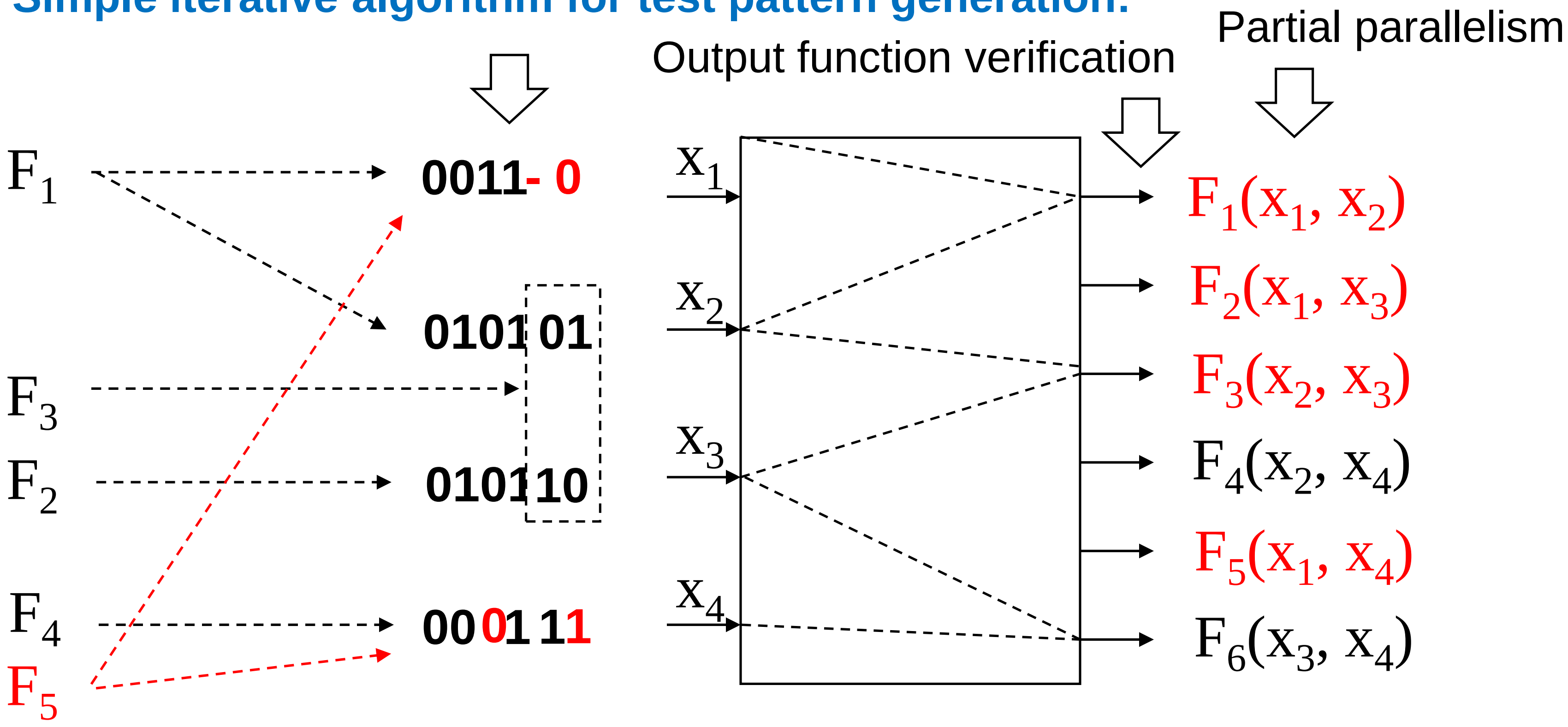
00000011
00000110
- - - -
11000000

Multiplicand operands: generated with FSM and replicated

Test length:
(n-1) × 11

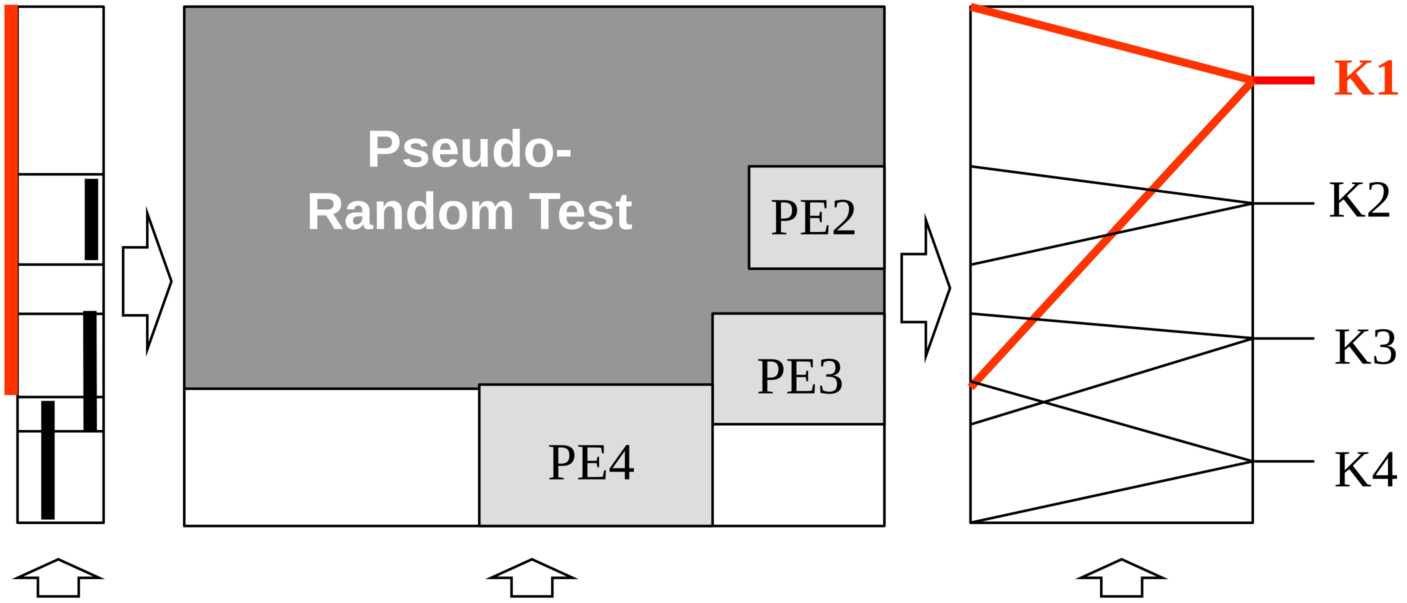
Pseudoexhaustive Test Optimization

Simple iterative algorithm for test pattern generation:



Exhaustive testing - 16
Pseudo-exhaustive, full parallel – 4 (not possible)
Pseudo-exhaustive, partially parallel - 6

Combined Pseudo-Exhaustive-Random Testing



Segmented LFSR

Combined PE-PR Test

Circuit with 4 cones

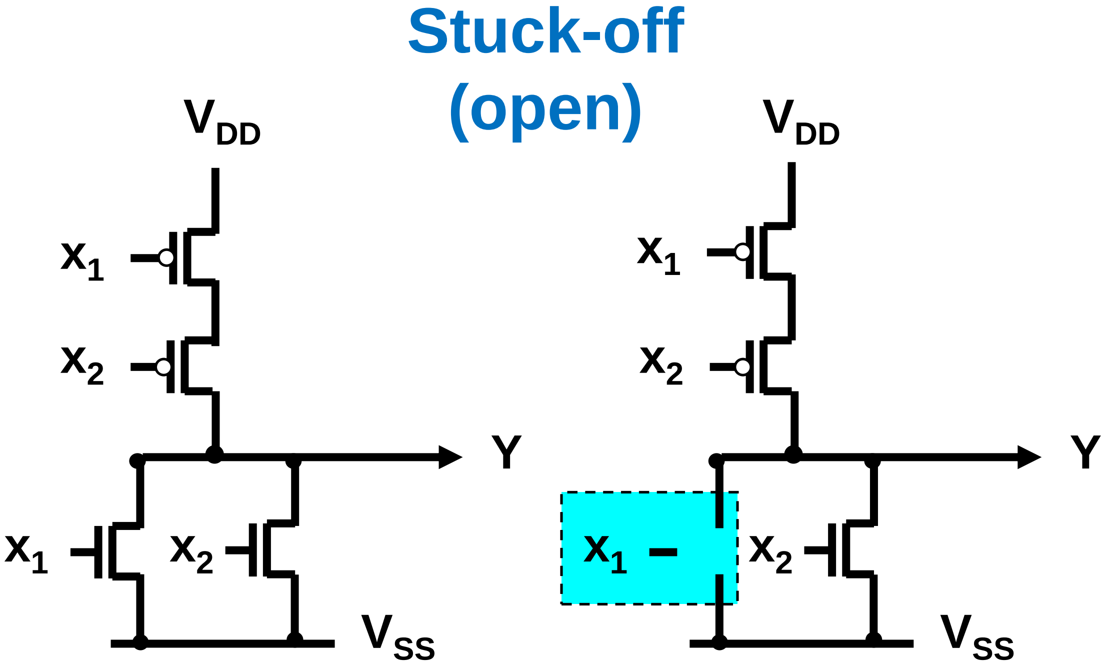
A set of Partial Pseudo-Exhaustive tests can be combined with

(1) **Pseudorandom BIST or**
(2) **Stored Deterministic test set**

Problems with Exhaustive Testing

Problem: Sequential fault class - Transistor Level Stuck-off Faults

NOR gate test:



x_1	x_2	y	y^d
0	0	1	1
0	1	0	0
1	0	0	Y’
1	1	0	0

Test sequence is needed: 00,10

No conducting path from V_{DD} to V_{SS} for “10”

Problems with Exhaustive Testing

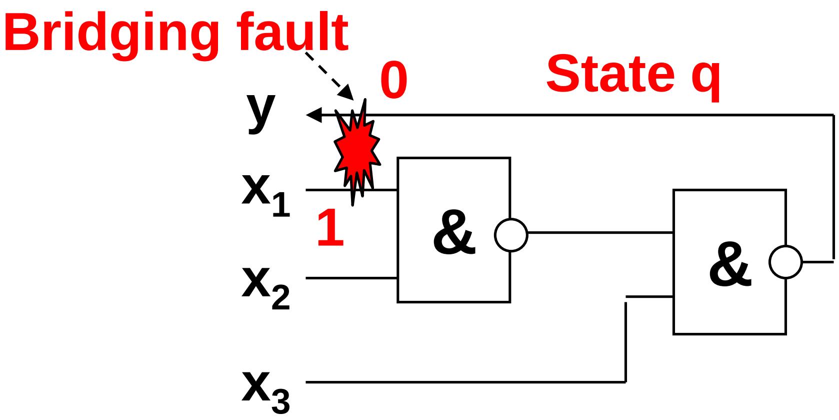
Problem: Sequential fault class - Bridging Fault Sequentiality

A short will change the circuit
into sequential one,
and you will need because of that

$2^4 = 16$ input patterns

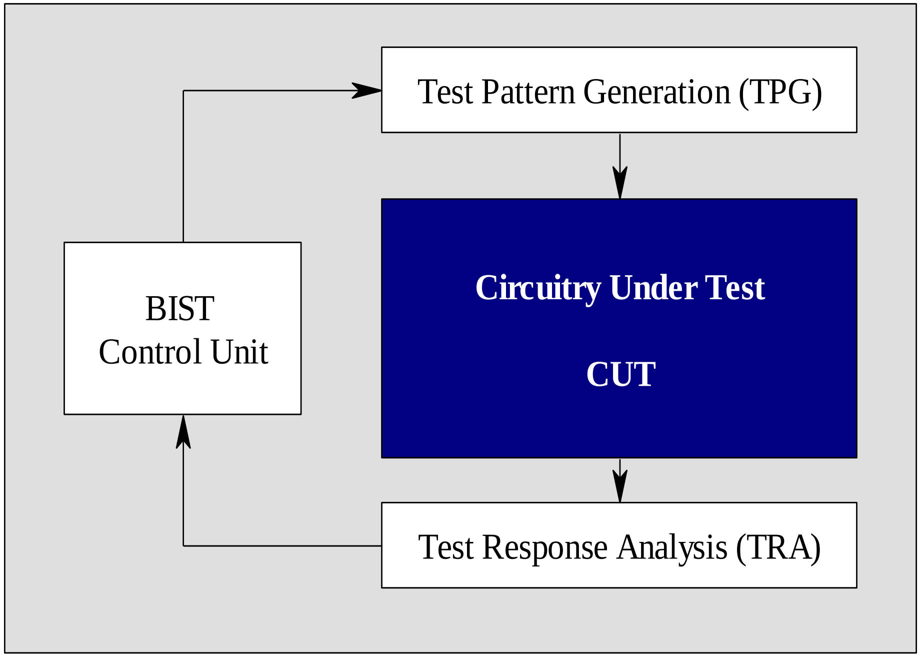
Instead of $2^3 = 8$

$Y = F(x_1, x_2, x_3)$



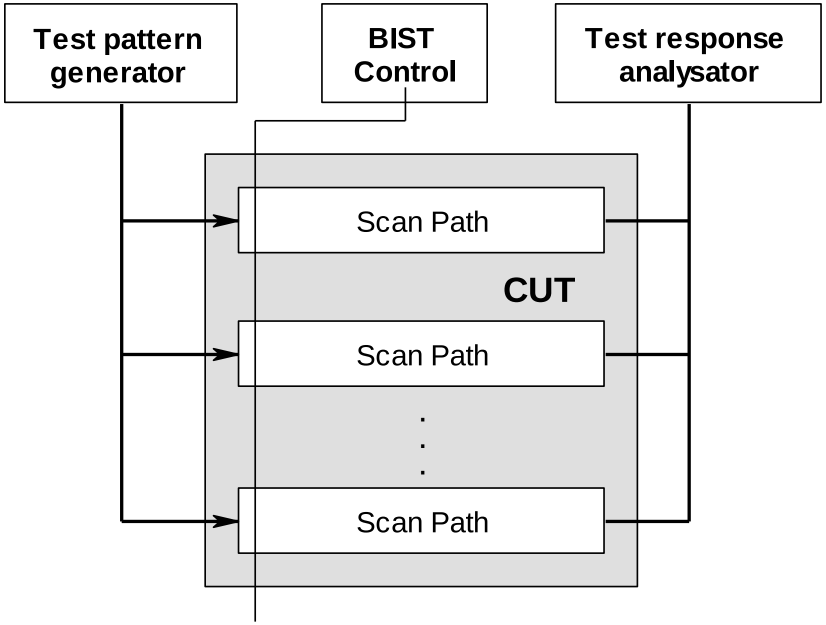
$Y = F(x_1, x_2, x_3,q)$

General Architecture of BIST



- BIST components:
  - Test pattern generator (TPG)
  - Test response analyzer (TRA)
- TPG & TRA are usually implemented as linear feedback shift registers (LFSR)
- Two widespread schemes:
  - test-per-scan
  - test-per-clock

Built-In Self-Test



- **Assumes existing scan architecture**
- **Drawback:**
  - **Long test application time**

Test per Scan:

Initial test set:

T1: 1100

T2: 1010

T3: 0101

T4: 1001

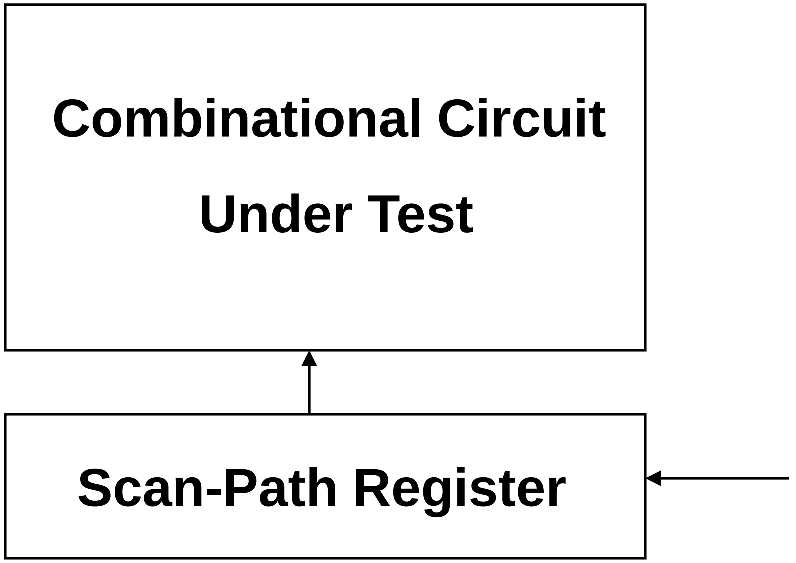
Test application:

1100 **T** 1010 **T** 0101**T** 1001 **T**

Number of clocks = (4 x 4) + 4 = 20

Built-In Self-Test

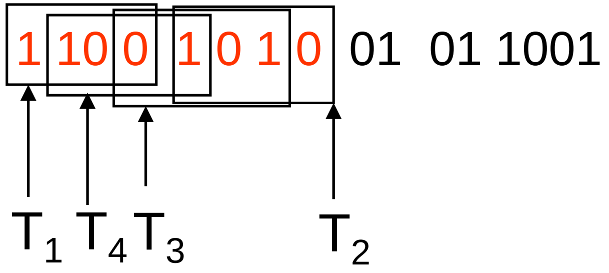
Test per Clock:



- **Initial test set:**
- T1: 1100
- T2: 1010
- T3: 0101
- T4: 1001

Assume, this is the full test sequence needed

- **Test application:**
- 1 10 0 1 0 1 0 01 01 1001



- Number of clocks = 8 < 20

Pattern Generation

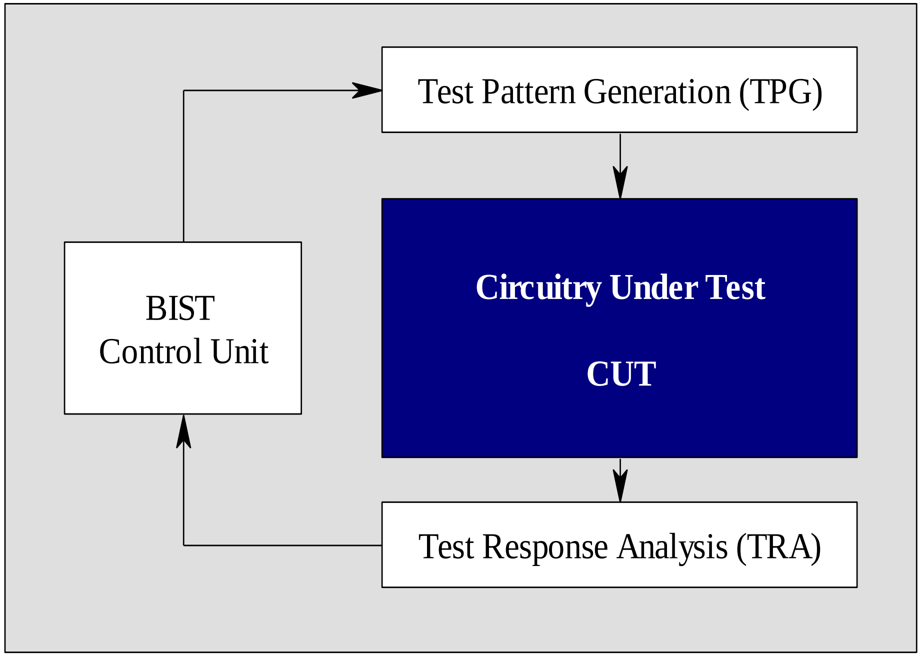
- **Store in ROM – too expensive**
- ***Exhaustive*** **– too long**
- ***Pseudo-exhaustive*** **– preferred**
- ***Pseudo-random*** **(LFSR) – preferred**
- **Binary counters – use more hardware than LFSR**
- **Modified counters**
- **Test pattern *augmentation* (Hybrid BIST)**
  - **LFSR combined with a few patterns in ROM**
  - **LFSR with bit flipping**
  - **LFSR with bit fixing**

LFSR Based Testing: Some Definitions

- ***Exhaustive testing* –** Apply all possible 2^n patterns to a circuit with n inputs
- ***Pseudo-exhaustive testing* –** Break circuit into small blocks (overlapping if needed) and test each exhaustively
- ***Pseudo-random testing* –** Algorithmic pattern generator that produces a subset of all possible tests with most of the properties of randomly-generated patterns
- **LFSR – *Linear feedback shift register,*** hardware that generates pseudo-random pattern sequence
- **BILBO – *Built-in logic block observer*,** extra hardware added to flip-flops so they can be **reconfigured** as an LFSR pattern generator or response compacter, a scan chain, or as flip-flops

Pattern Generation

Pseudorandom test generation by LFSR:

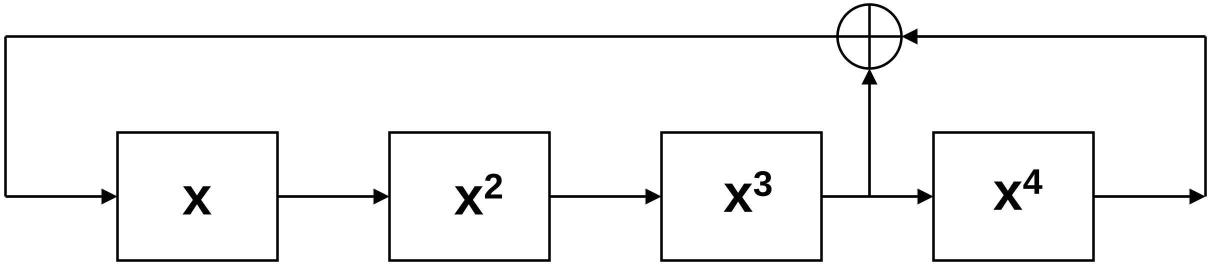


- **Using special LFSR registers**
  - Test pattern generator
  - Signature analyzer
- **Several proposals:**
  - BILBO
  - CSTP
- **Main characteristics of LFSR:**
  - polynomial
  - initial state
  - test length

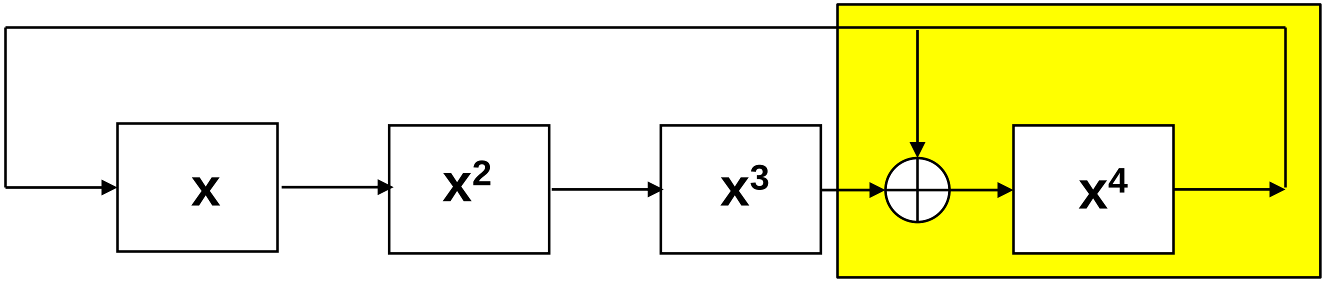
Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:

Standard LFSR



Modular LFSR

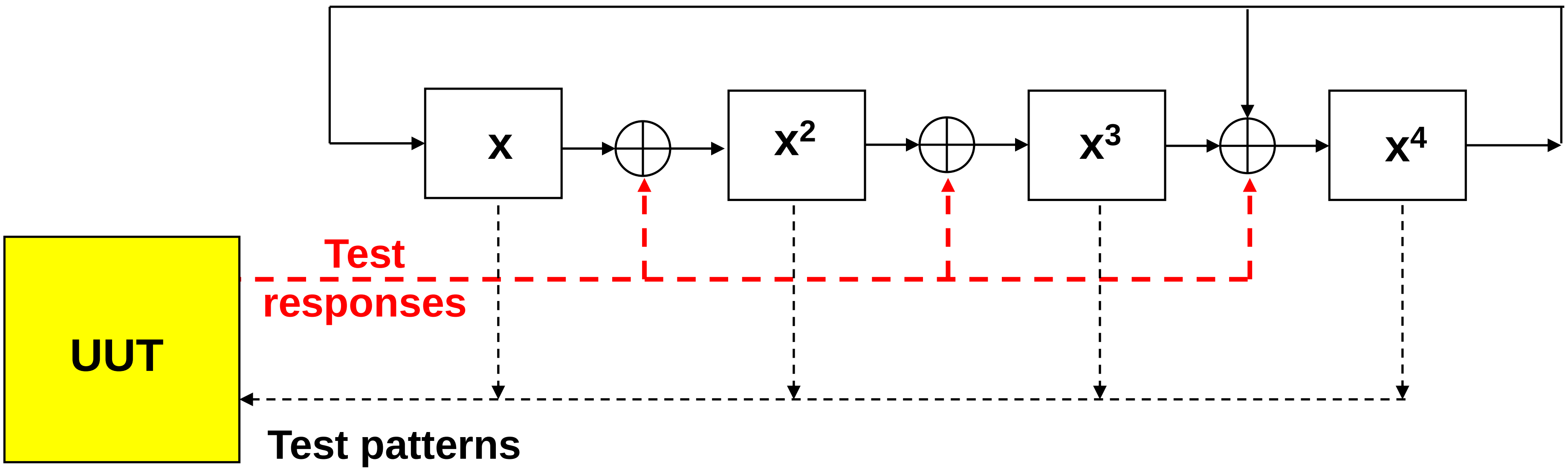


Polynomial: $P(x) = x^4 + x^3 + 1$

Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:

Why modular LFSR is useful for BIST?

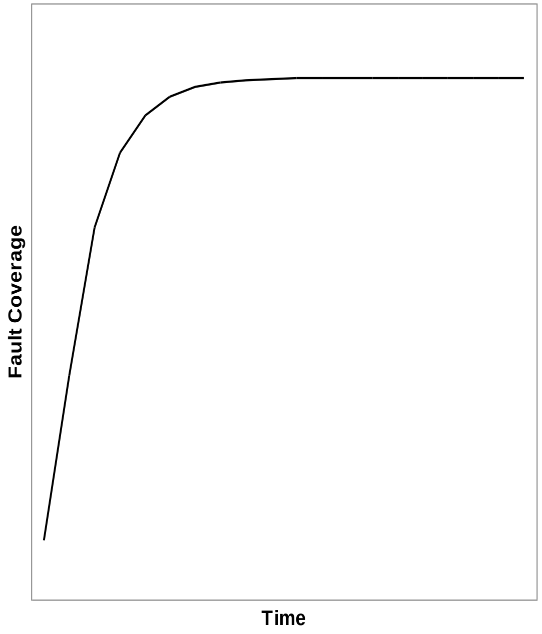


Polynomial: $P(x) = x^4 + x^3 + 1$

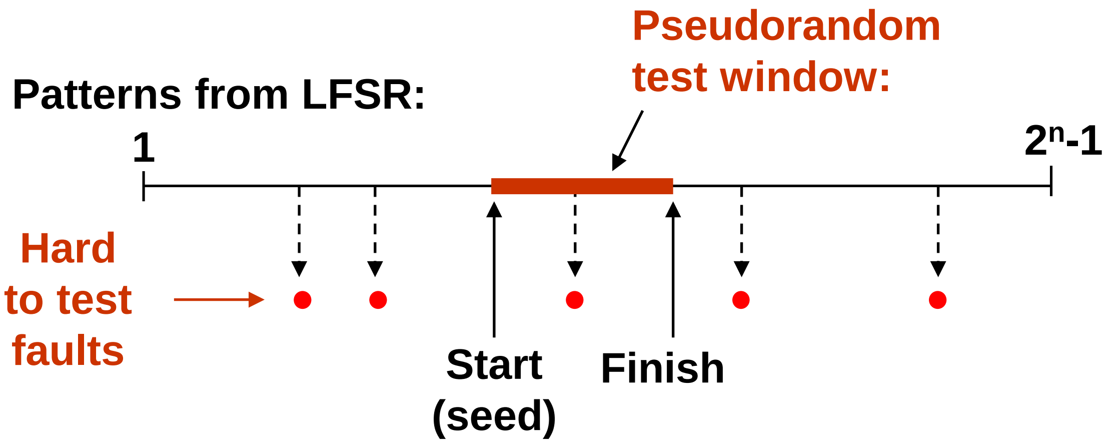
Problems with BIST: Hard to Test Faults

The main motivations of using random patterns are:

- low generation cost
- high initial efeciency



Problem: Low fault coverage



Pseudorandom test window:

Patterns from LFSR:

1

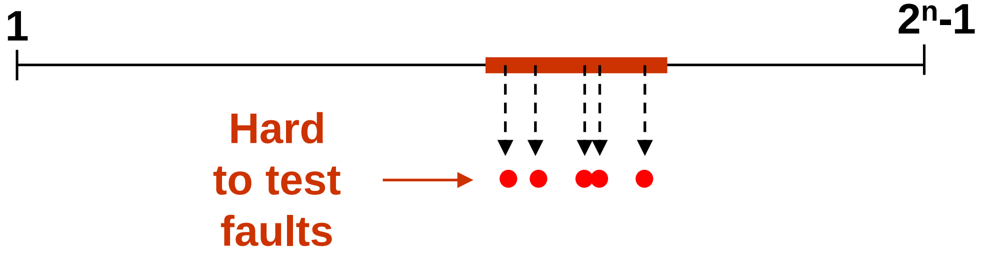
2^n-1

Hard to test faults

Start (seed)

Finish

Dream solution: Find LFSR such that:



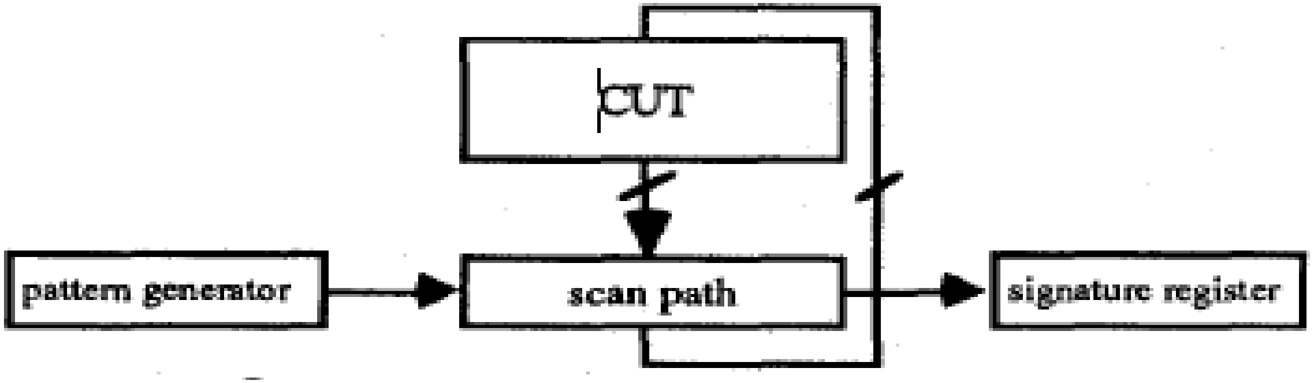
1

2^n-1

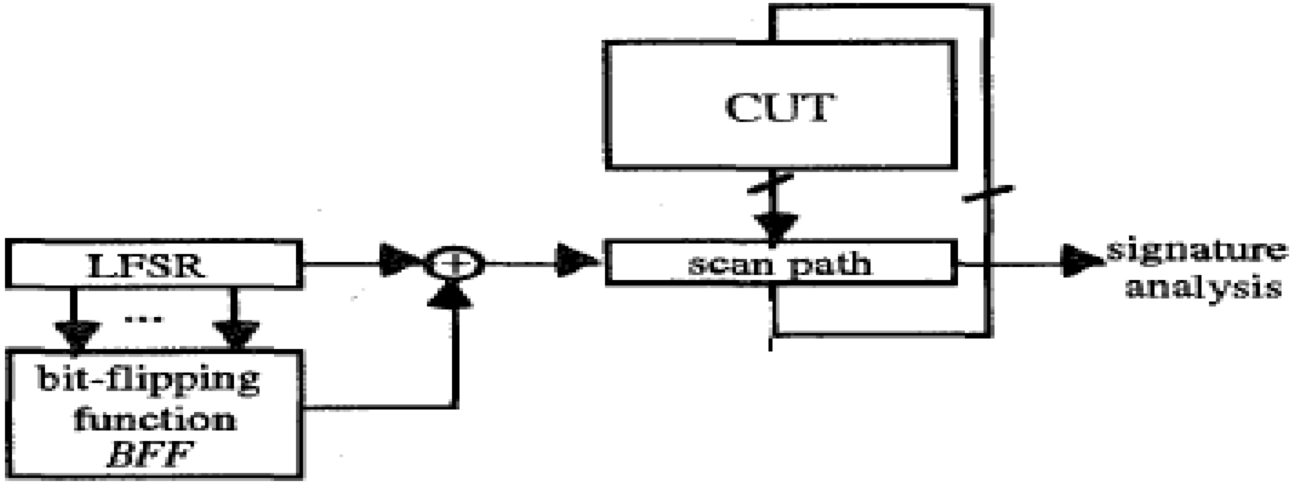
Hard to test faults

Pseudorandom Test Generation

Scan-based BIST

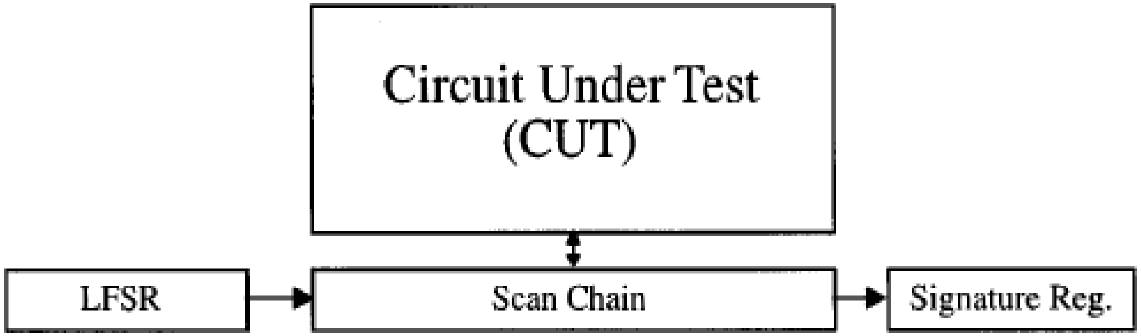


Bit-flipping BIST



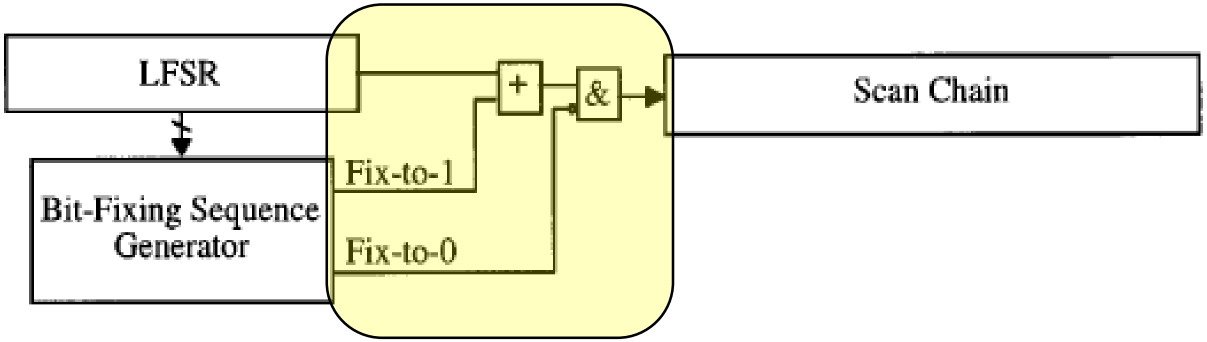
H.-J. Wunderlich, G. Kiefer. Bit flipping BIST. Proc. ICCAD, Nov. 1996, pp.337-343.

Pseudorandom Test Generation



Block diagram for a “test-per-scan” BIST scheme.

Bit-fixing BIST



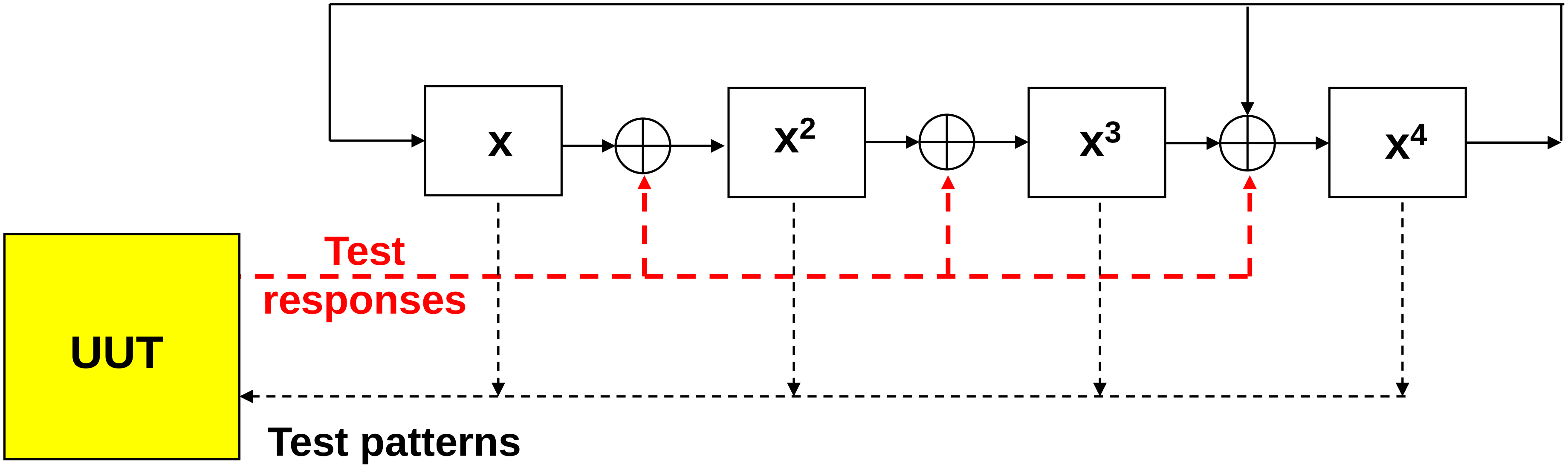
Logic for altering the pseudorandom bit sequence.

N.A. Touba, E.J. McCluskey. Bit-fixing in pseudorandom sequences for scan BIST. IEEE Trans. on CAD of IC and Systems, Vol.20, No.4, Apr.2001.

Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:

Why modular LFSR is useful for BIST?



Polynomial: $P(x) = x^4 + x^3 + 1$

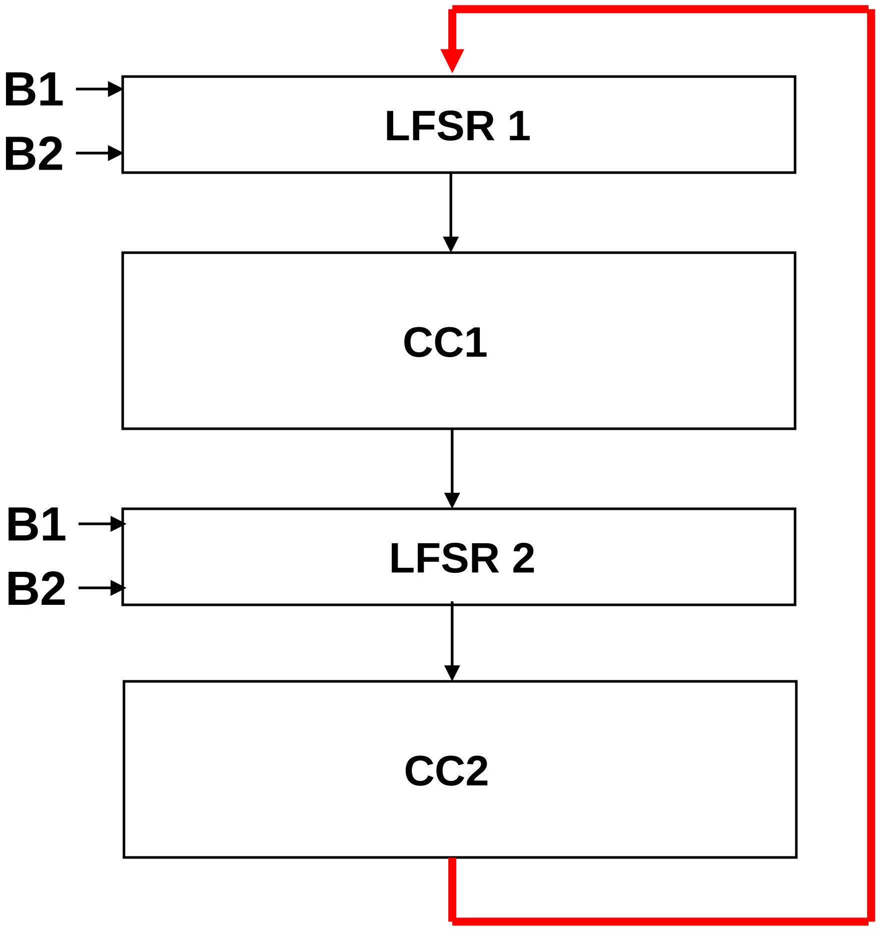
BILBO BIST Architecture

Working modes:

B1	B2	
0	0	Normal mode
0	1	Reset
1	0	Test mode
1	1	Scan mode

Testing modes:

CC1:	LFSR 1	-	TPG
	LFSR 2	-	SA
CC2:	LFSR 2	-	TPG
	LFSR 1	-	SA



BILBO BIST Architecture

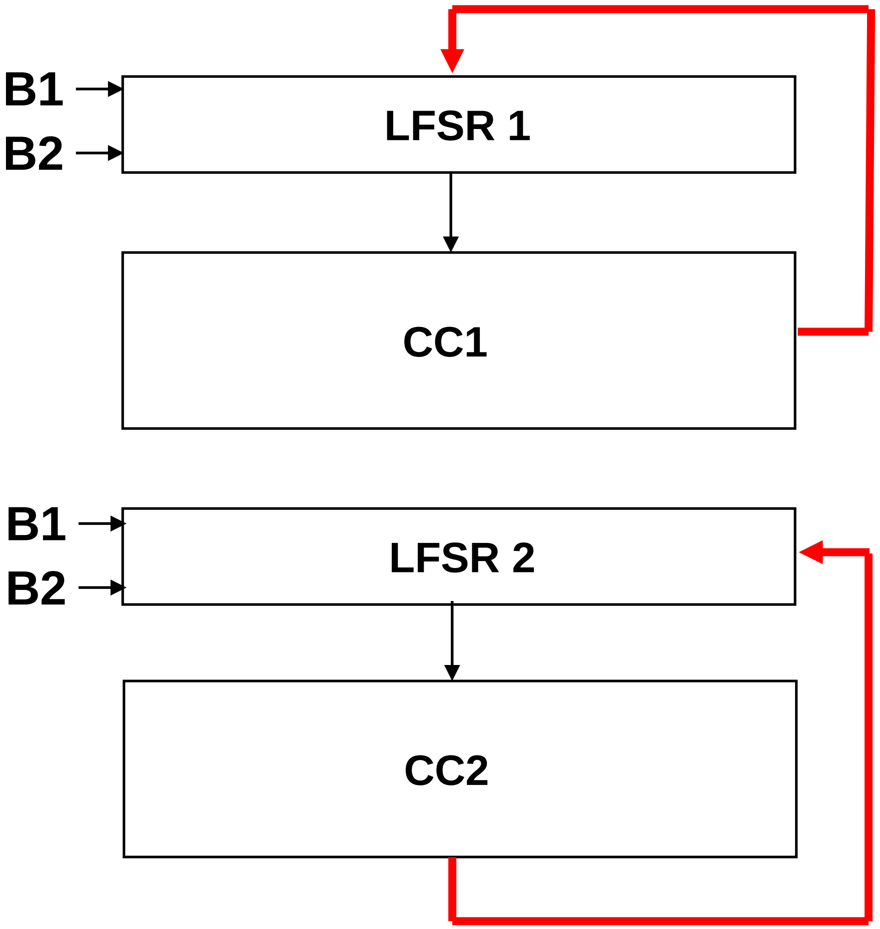
Working modes:

B1	B2	
0	0	Normal mode
0	1	Reset
1	0	Test mode
1	1	Scan mode

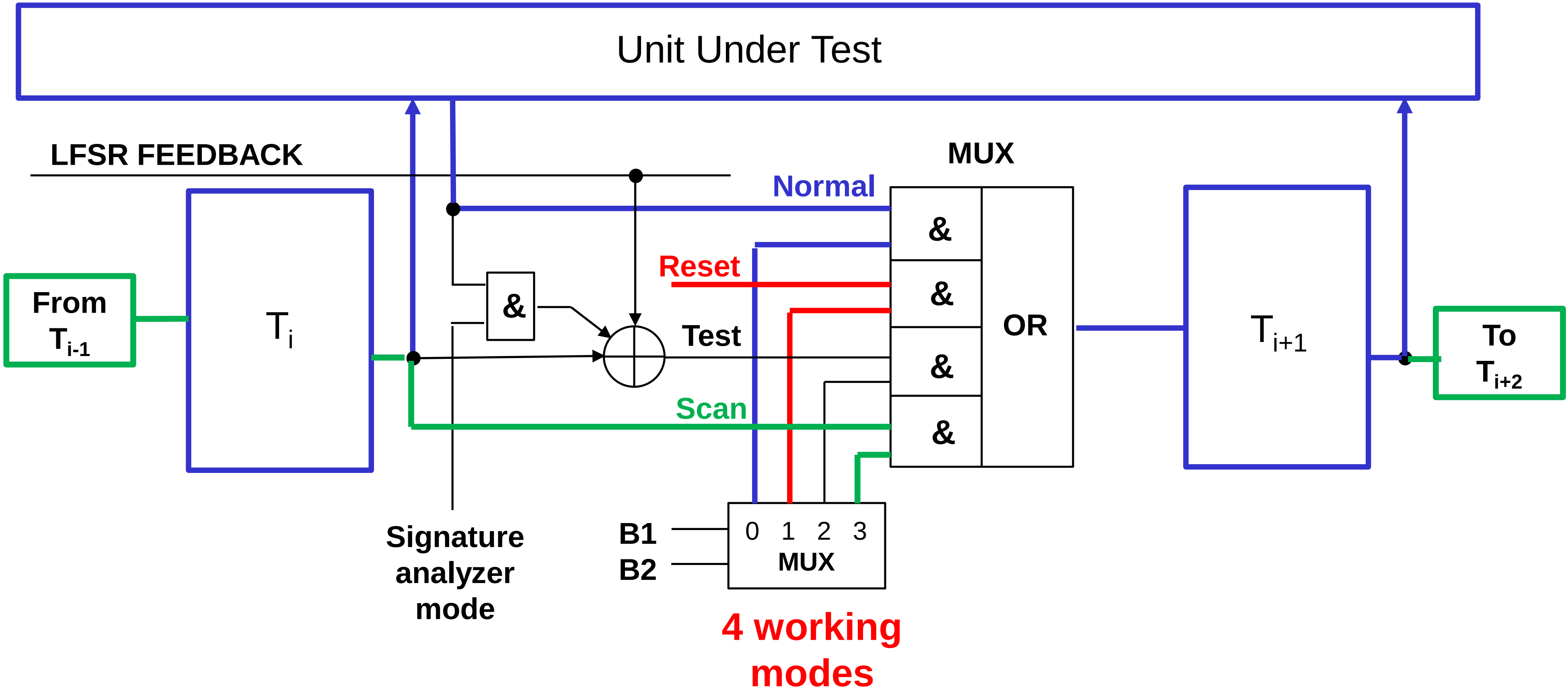
Testing modes:

CC1, CC2 Tested in parallel:

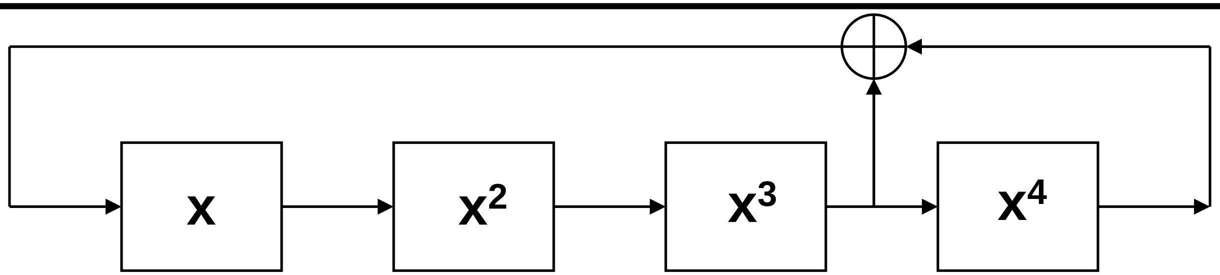
LFSR 1
LFSR 2
} TPG + SA



Reconfiguration of the LFSR



Pseudorandom Test Generation - LFSR



Two approaches to LFSR simulation:

Polynomial: $P(x) = x^4 + x^3 + 1$

Matrix calculation:

$$\begin{pmatrix} X_4(t+1)\\ X_3(t+1)\\ X_2(t+1)\\ X_1(t+1)\end{pmatrix} = \begin{pmatrix} \boxed{\begin{matrix}0&1&0&0\\0&0&1&0\\0&0&0&1\end{matrix}} \\ \boxed{\begin{matrix}1&h_3&h_2&h_1\end{matrix}}\end{pmatrix} \begin{pmatrix} X_4(t)\\ X_3(t)\\ X_2(t)\\ X_1(t)\end{pmatrix} = \begin{pmatrix} X_3\\ X_2\\ X_1\\ X_4 \oplus X_3\end{pmatrix}$$

Shift

1 0 0

Feedback

t	x	x²	x³	x⁴	t	x	x²	x³	x⁴
1	0	0	0	1	9	0	1	0	1
2	1	0	0	0	10	1	0	1	0
3	0	1	0	0	11	1	1	0	1
4	0	0	1	0	12	1	1	1	0
5	1	0	0	1	13	1	1	1	1
6	1	1	0	0	14	0	1	1	1
7	0	1	1	0	15	0	0	1	1
8	1	0	1	1	16	0	0	0	1

Theory of LFSR: Primitive Polynomials

Properties of Polynomials:

- ***Irreducible polynomial*** – cannot be factored, is divisible only by itself
- Any polynomial with all even exponents can be factored and hence is ***reducible***
- Irreducible polynomial of degree ***n*** is characterized by:
  - An odd number of terms including 1 term $x^3 + x^2 + 1$
  - Divisibility into $x^k + 1$, where $\boldsymbol{k} = 2^n - 1$ $\boxed{x^7 + 1}$
- An irreducible polynomial of degree ***n*** is ***primitive*** if it divides the polynomial $\mathbf{x^k + 1}$ for $\mathbf{k = 2^n - 1}$, but not for any smaller positive integer k

Theory of LFSR: Examples

Polynomials of degree n=3 (examples):

$$\boxed{k = 2^n - 1 = 2^3 - 1 = 7}$$

Primitive polynomials:

$$x^3 + x^2 + 1$$

$$x^3 + x + 1$$

The polynomials will divide evenly the polynomial ***$x^7 + 1$***

but not any one of ***$k<7$***, hence, they are primitive

They are also **reciprocal**: coefficients are 1011 and 1101

Reducible polynomials (non-primitive):

Primitive polynomial

$$x^3 + 1 = (x+1)\boxed{(x^2 + x + 1)}$$

$$x^3 + x^2 + x + 1 = (x+1)(x^2 + 1)$$

Theory of LFSR: Examples

Is $x^4 + x^2 + 1$ **a primitive polynomial?**

Irreducible polynomial of degree *n* is characterized by:

- An odd number of terms including 1 term?

Yes, it includes 3 terms

- Divisibility into 1 + x^k, where $k = 2^n – 1$

No, there is remainder

$x^4 + x^2 + 1$ **is non-primitive?**

Divisibility check:

$$\begin{array}{r|l} & x^{11}+x^9+x^5+x^3 \\ \hline \boxed{x^4+x^2+1} & \boxed{x^{15}+1} \\ & \underline{x^{15}+x^{13}+x^{11}} \\ & x^{13}+x^{11}\quad+1 \\ & \underline{x^{13}+x^{11}+x^9} \\ & x^9+1 \\ & \underline{x^9+x^7+x^5} \\ & x^7+x^5+1 \\ & \underline{x^7+x^5+x^3} \\ & \boxed{x^3+1} \end{array}$$

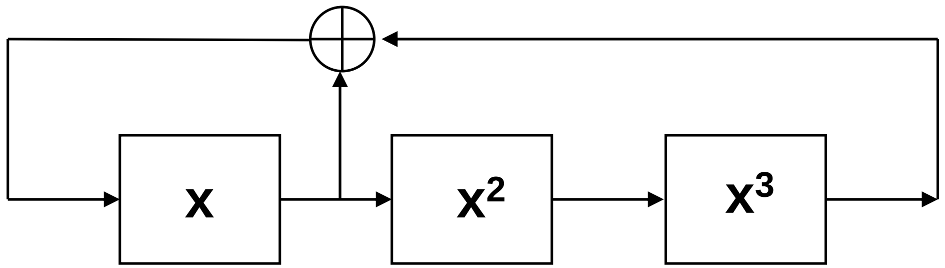
Theory of LFSR: Examples

Simulation of the behaviour of LFSR by polynomial:

Primitive polynomials

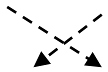
$$x^3 + x + 1$$

100
110
111
011
101
010
001
100



Theory of LFSR: Examples

Comparison of test sequences generated:

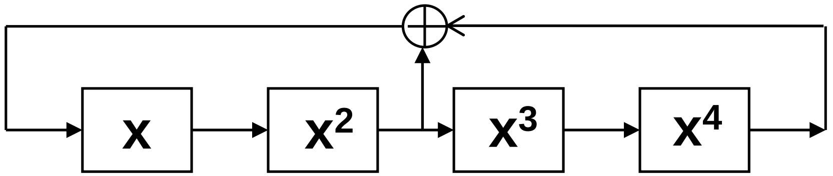


Primitive polynomials		Non-primitive polynomials	
x^3+x+1	x^3+x^2+1	x^3+1	x^3+x^2+x+1
100	100	100	100
110	010	010	110
111	101	001	011
011	110	100	001
101	111	010	100
010	011	001	110
001	001	100	011
100	100	010	001

Theory of LFSR: Examples

Non-primitive polynomial

$x^4 + x^2 + 1$



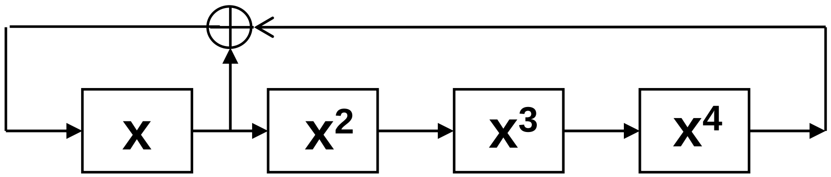
0001
1000
0100
1010
0101
0010
0001

1001
1100
1110
1111
0111
0011
1001

0110
1011
1101
0110

Primitive polynomial

$x^4 + x + 1$



0001
1000
1100
1110
1111
0111

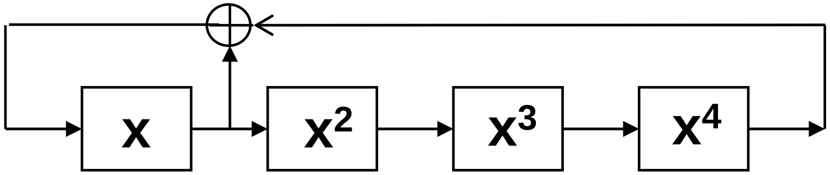
1011
0101
1010
1101
0110
0011

1001
0100
0010
0001

Theory of LFSR: Examples

Primitive polynomial

$x^4 + x + 1$



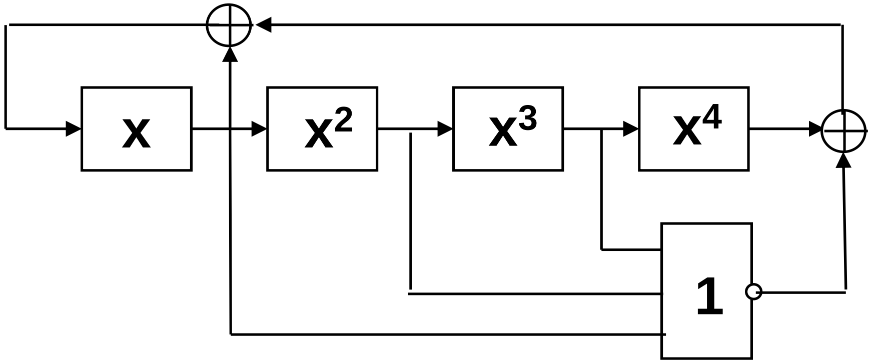
0001
1000
1100
1110
1111
0111

1011
0101
1010
1101
0110
0011

1001
0100
0010
0001

The code 0000 is missing

Zero generation:



0000
1000
1100
1110
1111
0111

1011
0101
1010
1101
0110
0011

1001
0100
0010
0001
0000

Pseudorandom Testing with LFSR

Primitive polynomial

$x^4 + x + 1$

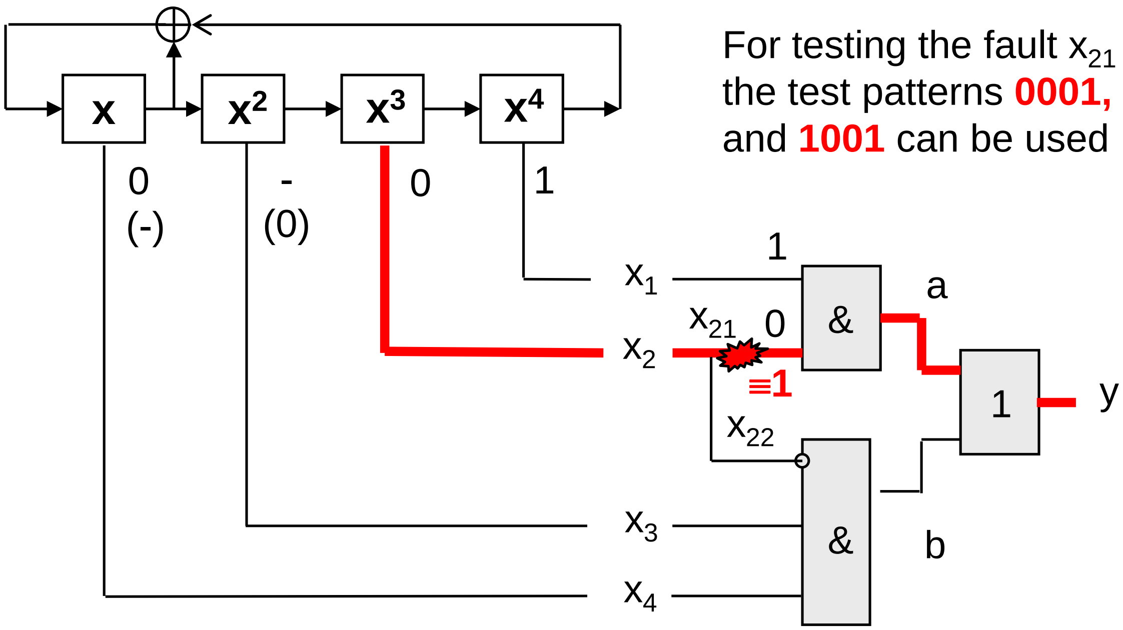
Red patterns are test patterns

0001
1000
1100
1110
1111
0111
1011
0101
1010
1101
0110
0011
1001
0100
0010
0001

No match in the blue sequence

LFSR

Circuit Under test

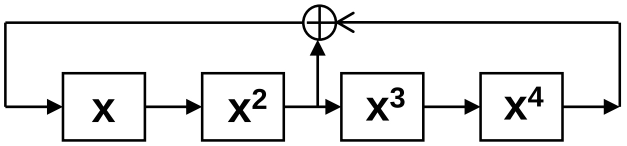


For testing the fault $x_{21} \equiv 1$ the test patterns **0001, 0101** and **1001** can be used

Pseudorandom Testing with LFSR

Non-primitive polynomial

$x^4 + x^2 + 1$

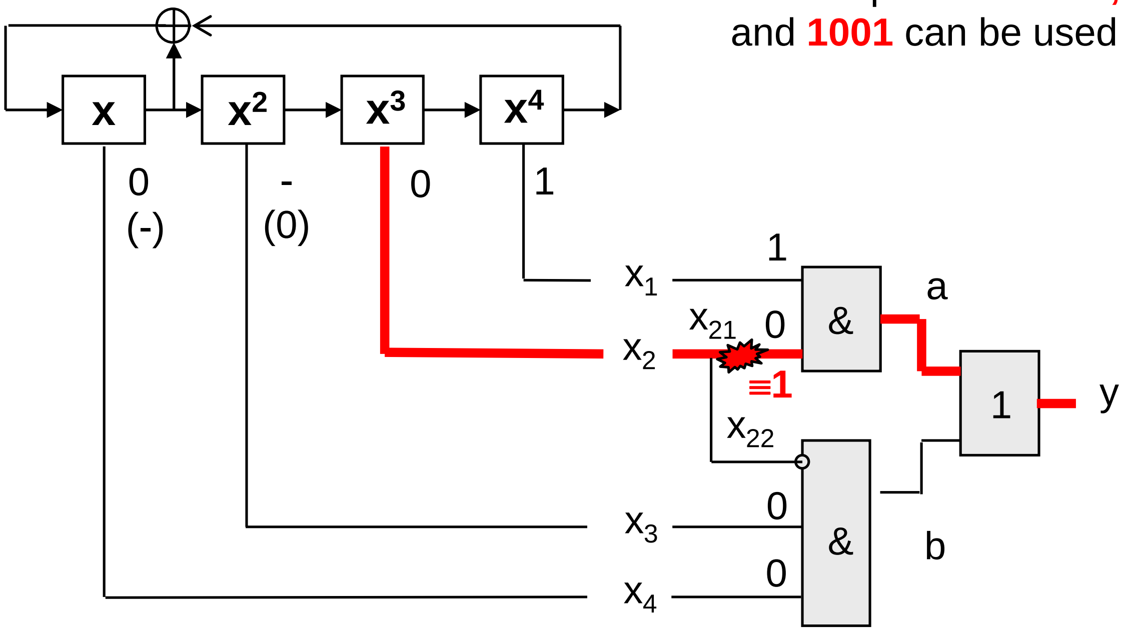


0001
1000
0100
1010
0101
0010
0001

1001
1100
1110
1111
0111
0011
1001

0110
1011
1101
0110

For testing the fault $x_{21} \equiv 1$ the test patterns **0001, 0101** and **1001** can be used



Be careful: **no proper patterns can be generated using the seed 0110**

Blue patterns are not testing the fault

Theory of LFSR: Primitive Polynomials

Number of primitive polynomials of degree *N*

N	No
1	1
2	1
4	2
8	16
16	2048
32	67108864

Table of primitive polynomials up to degree 31

N	Primitive Polynomials
1,2,3,4,6,7,15,22	$1 + X + X^n$
5,11, 21, 29	$1 + X^2 + X^n$
10,17,20,25,28,31	$1 + X^3 + X^n$
9	$1 + X^4 + X^n$
23	$1 + X^5 + X^n$
18	$1 + X^7 + X^n$
8	$1 + X^2 + X^3 + X^4 + X^n$
12	$1 + X + X^3 + X^4 + X^n$
13	$1 + X + X^4 + X^6 + X^n$
14, 16	$1 + X + X^3 + X^4 + X^n$

Theory of LFSR: Primitive Polynomials

Examples of PP (exponents of terms):

$x^3 + x + 1$

n	other				n	other			
1	0				9	4	0		
2	1	0			10	3	0		
3	1	0			11	2	0		
4	1	0			12	7	4	3	0
5	2	0			13	4	3	1	0
6	1	0			14	12	11	1	0
7	1	0			15	1	0		
8	6	5	1	0	16	5	3	2	0

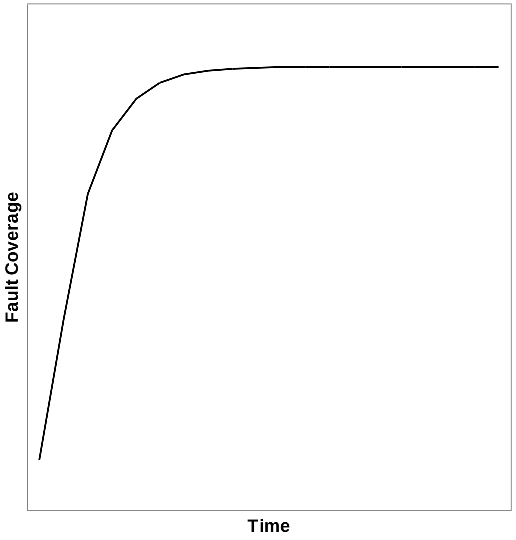
$x^{13} + x^4 + x^3 + x + 1$

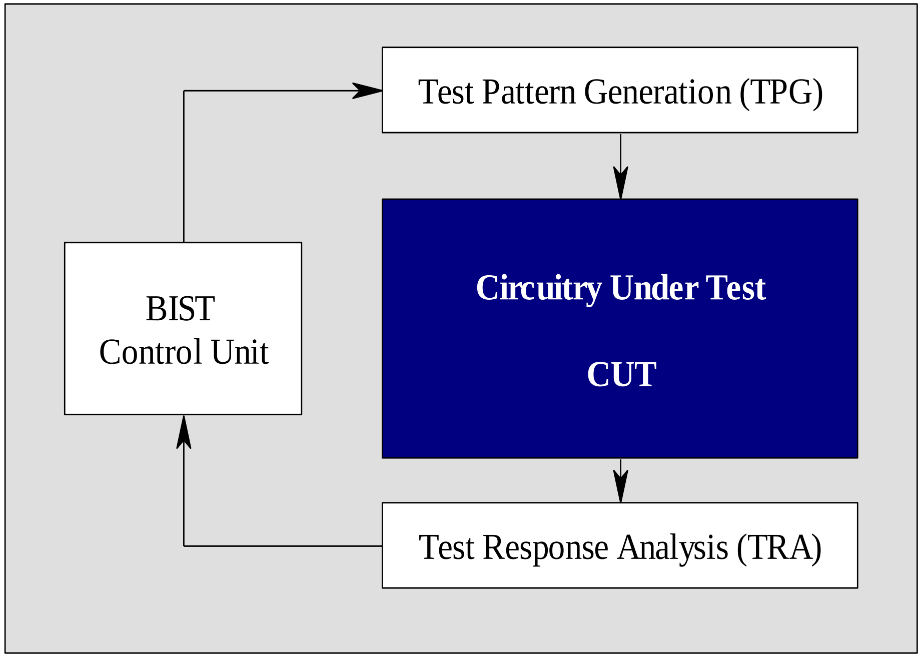
BIST: Fault Coverage

Pseudorandom Test generation by LFSR:

Motivation for LFSR:

- low generation cost
- high initial efeciency





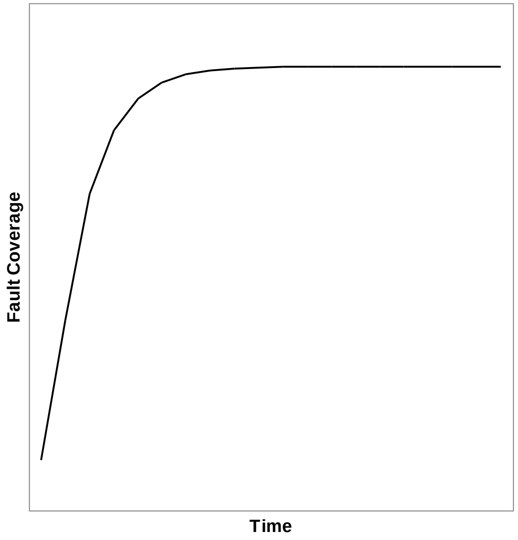
Drawback: 100% fault coverage is difficult to achieve

BIST: Fault Coverage

Pseudorandom Test generation by LFSR:

Motivation for LFSR:

- low generation cost
- high initial efeciency



Reasons of the high initial efficiency:

A circuit may implement 2^{2^n} functions

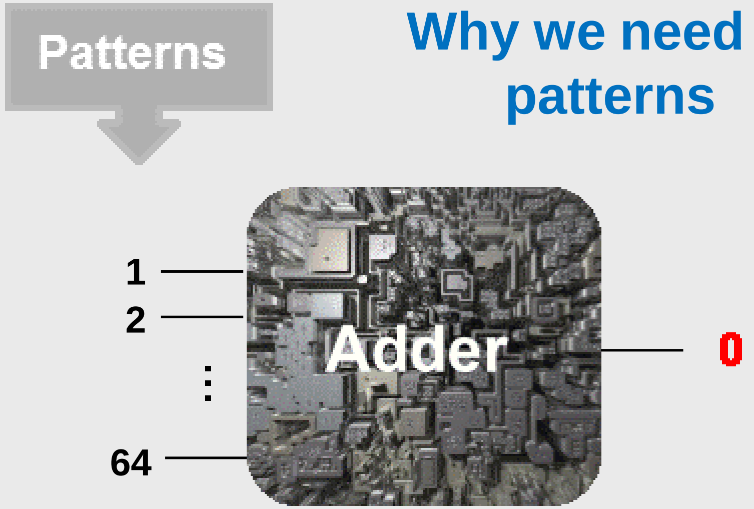
A test vector partitions the functions into 2 equal sized equivalence classes (correct circuit in one of them)

The second vector partitions into 4 classes etc.

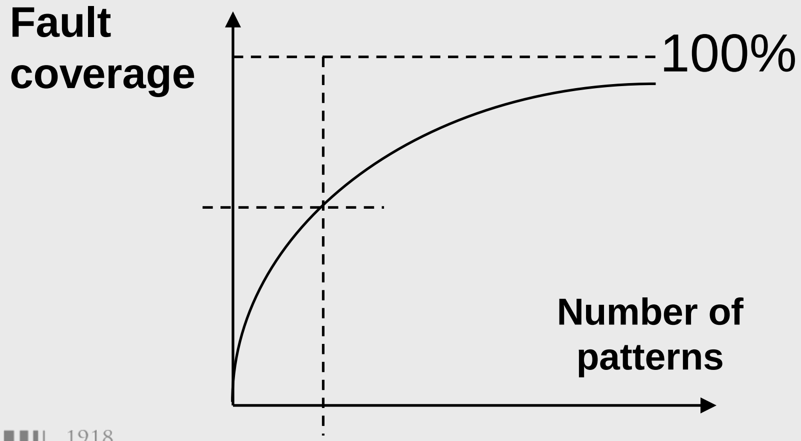
After m patterns the fraction of functions distinguished from the correct function is

$$\frac{1}{2^{2^n}-1}\sum_{i=1}^{m} 2^{2^n-i}, \quad 1 \le m \le 2^n$$

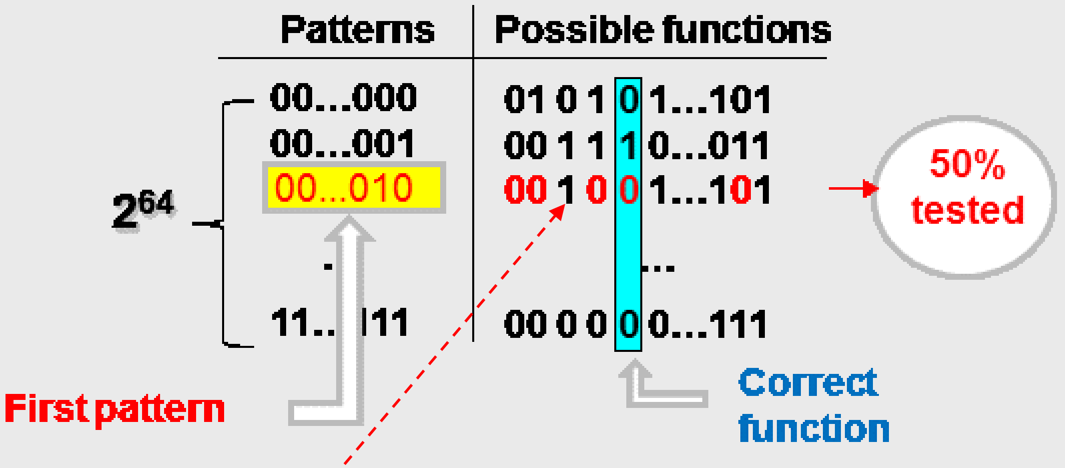
Fault Coverage: Functional View



Why we need 2^{64} patterns



Truth table for adder:



Patterns	Possible functions
00...000	01 0 1 0 1...101
00...001	00 1 1 1 0...011
00...010	00 1 0 0 1...101
.	...
11...111	00 0 0 0 0...111

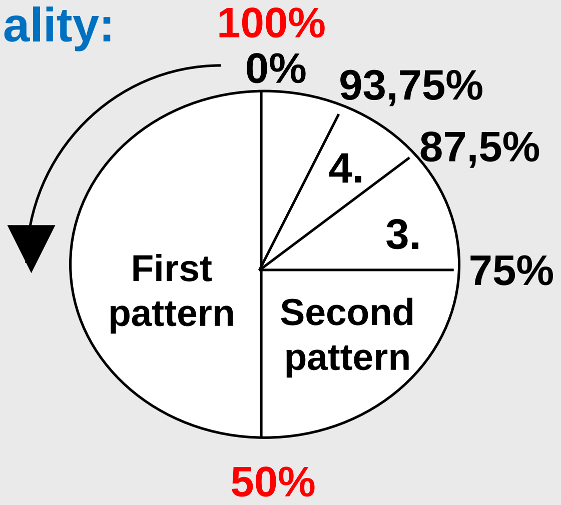
2^{64}

First pattern

Correct function

50% tested

Test quality:



All columns in truth table can be removed where for yellow pattern the result is 1

BIST: Fault Coverage

Explanation of the formula of fault coverage:

1) General case:

$$\frac{1}{2^{2^n}-1}\sum_{i=1}^{m}2^{2^n-i}, \quad 1\le m\le 2^n$$

# tested functions

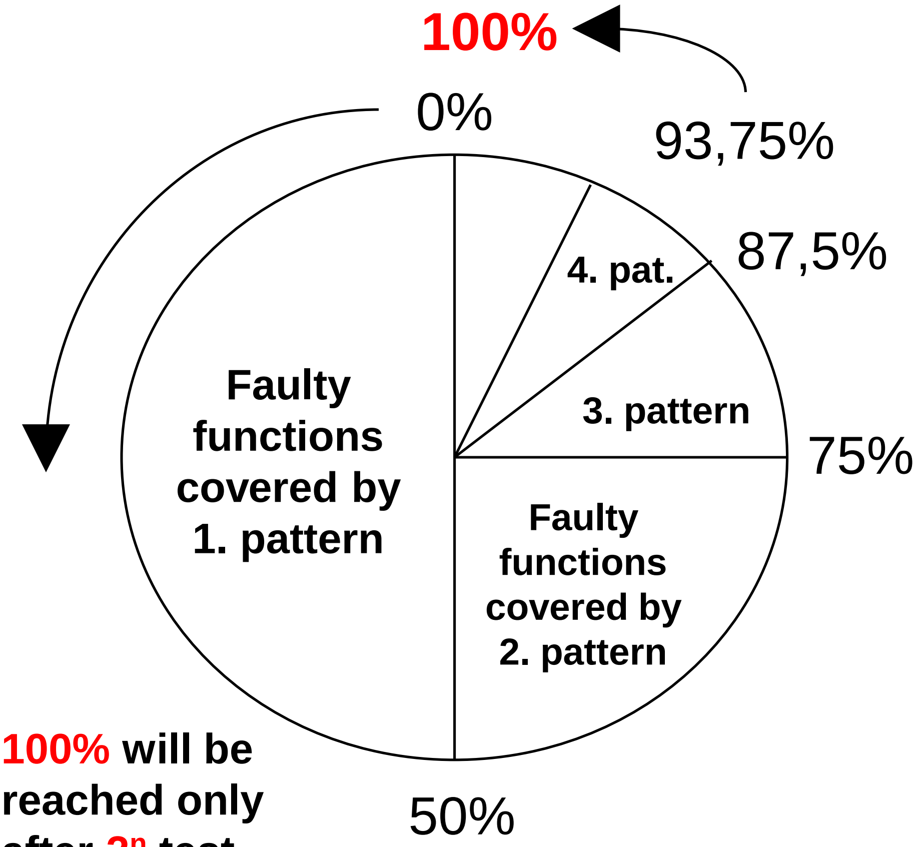
# all functions

2) Example:

n = 2, **m = 1**, i = 1:

$$\frac{1}{2^{2^2}-1}\sum_{i=1}^{1}2^{2^2-1}=\frac{2^3}{15}=\frac{8}{15}$$

n – number of inputs,
m – number of test patterns,
i – share of each pattern

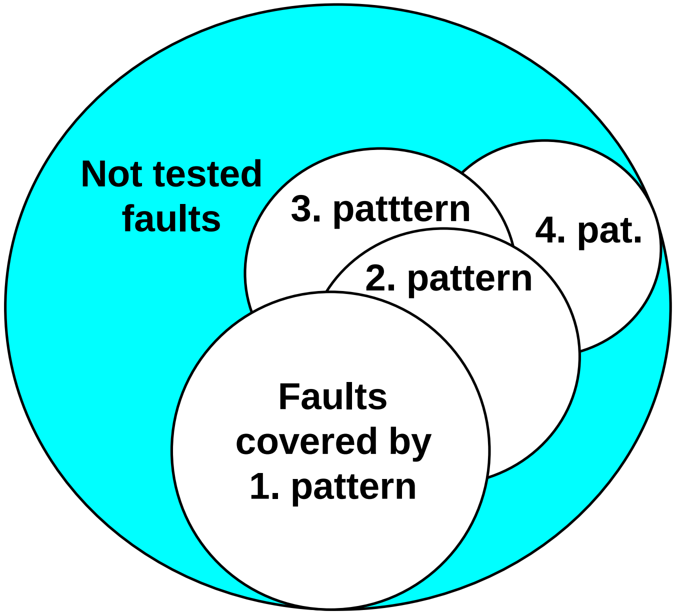

100%
0%
93,75%
87,5%
4. pat.
3. pattern
75%
Faulty functions covered by 1. pattern
Faulty functions covered by 2. pattern
50%

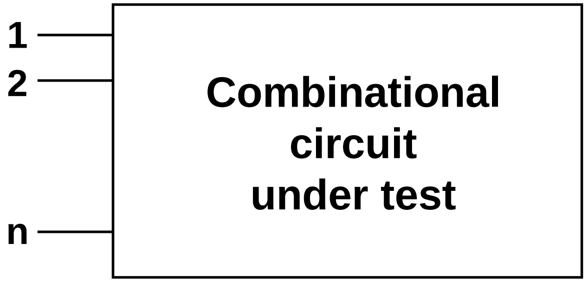
100% will be reached only after 2^n test patterns

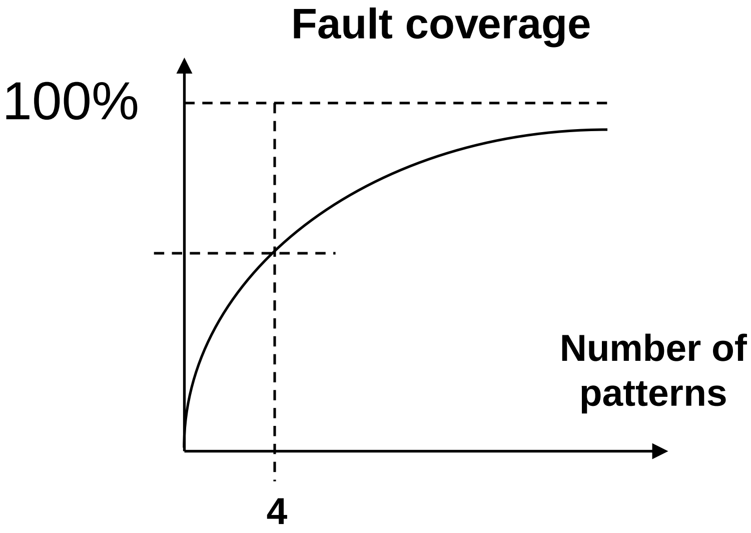
BIST: Structural Approach to Test

Deterministic test approach:

Testing of structural faults:

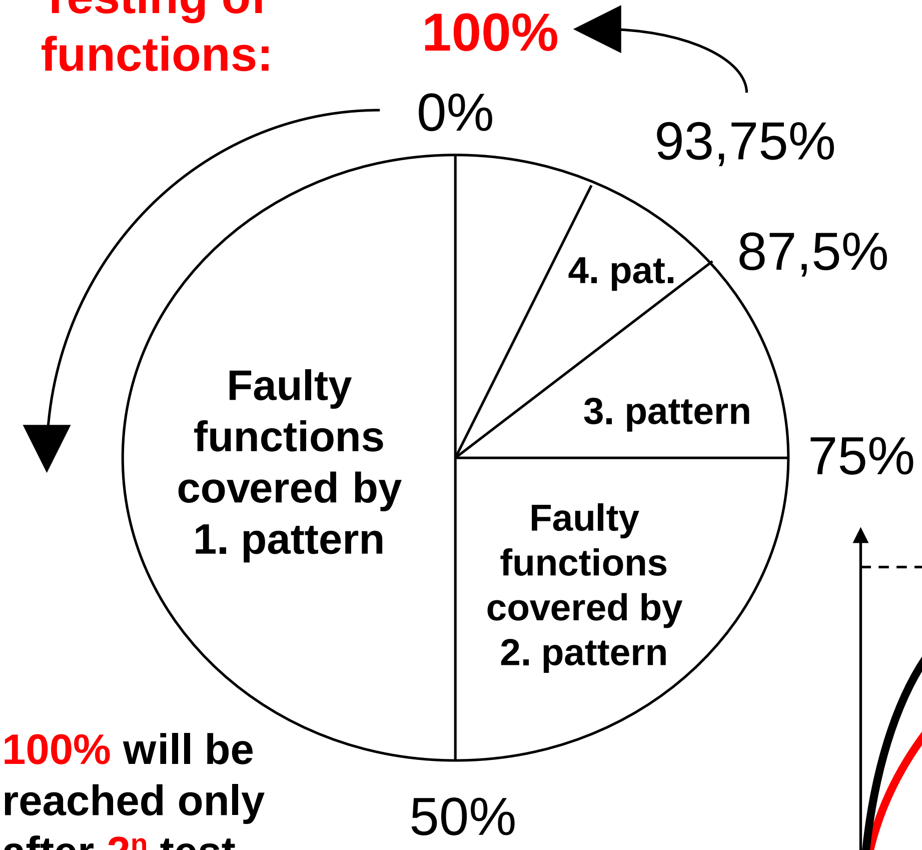






BIST: Two Approaches to Test

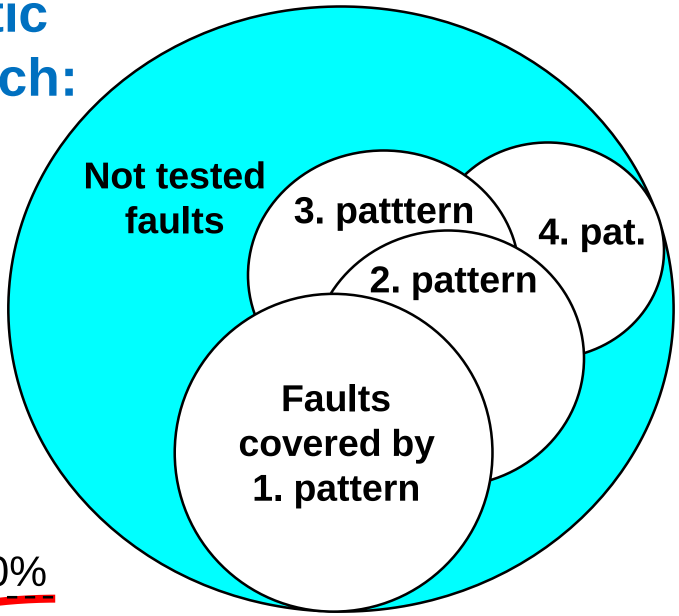
Testing of functions:

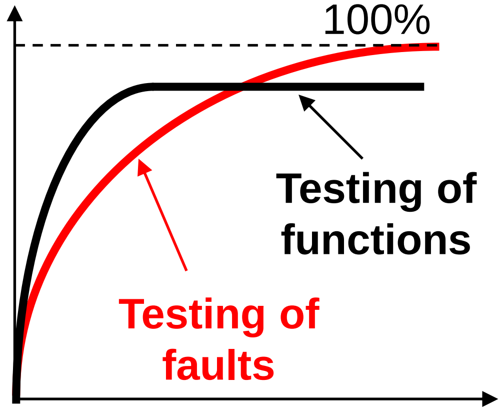

100%
0%
93,75%
87,5%
4. pat.
3. pattern
75%
Faulty functions covered by 1. pattern
Faulty functions covered by 2. pattern
50%

100% will be reached only after 2^n test patterns

Deterministic test approach:

Testing of faults:



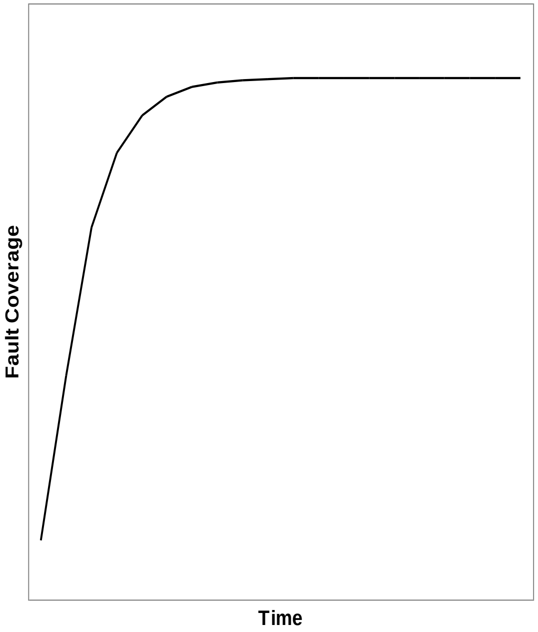


100% will be reached when all faults from the fault list are covered

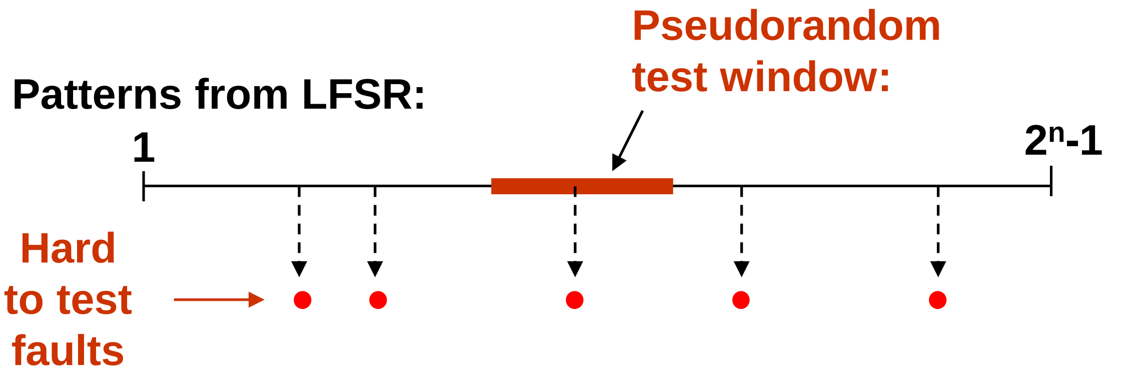
Problems with BIST: Hard to Test Faults

The main motivations of using random patterns are:

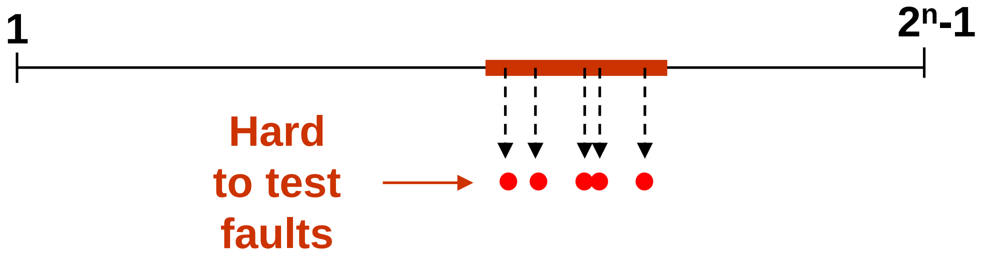
- low generation cost
- high initial efeciency



Problem: Low fault coverage

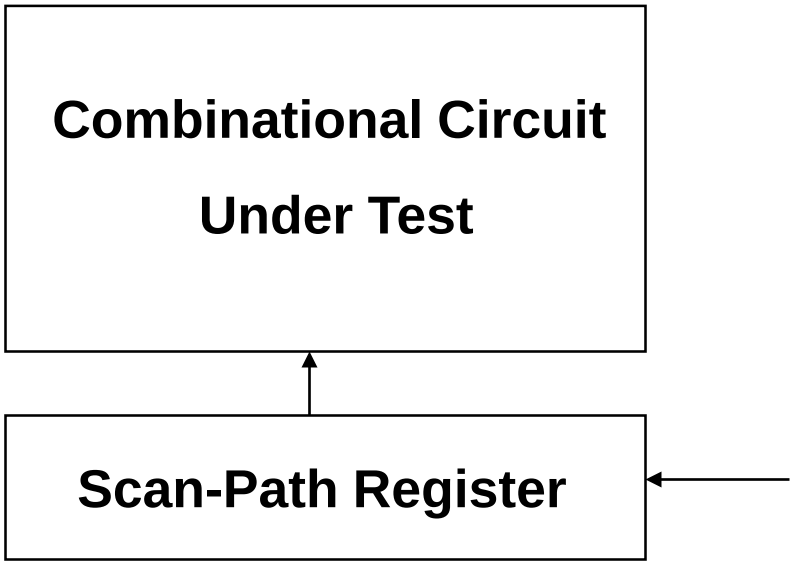


Dream solution: Find LFSR such that:



Deterministic Scan-Path Test

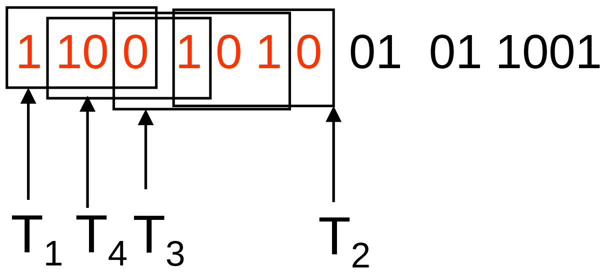
Test per Clock:



- **Initial test set:**
- T1: 1100
- T2: 1010
- T3: 0101
- T4: 1001

How to generate the shortest sequence by LFSR

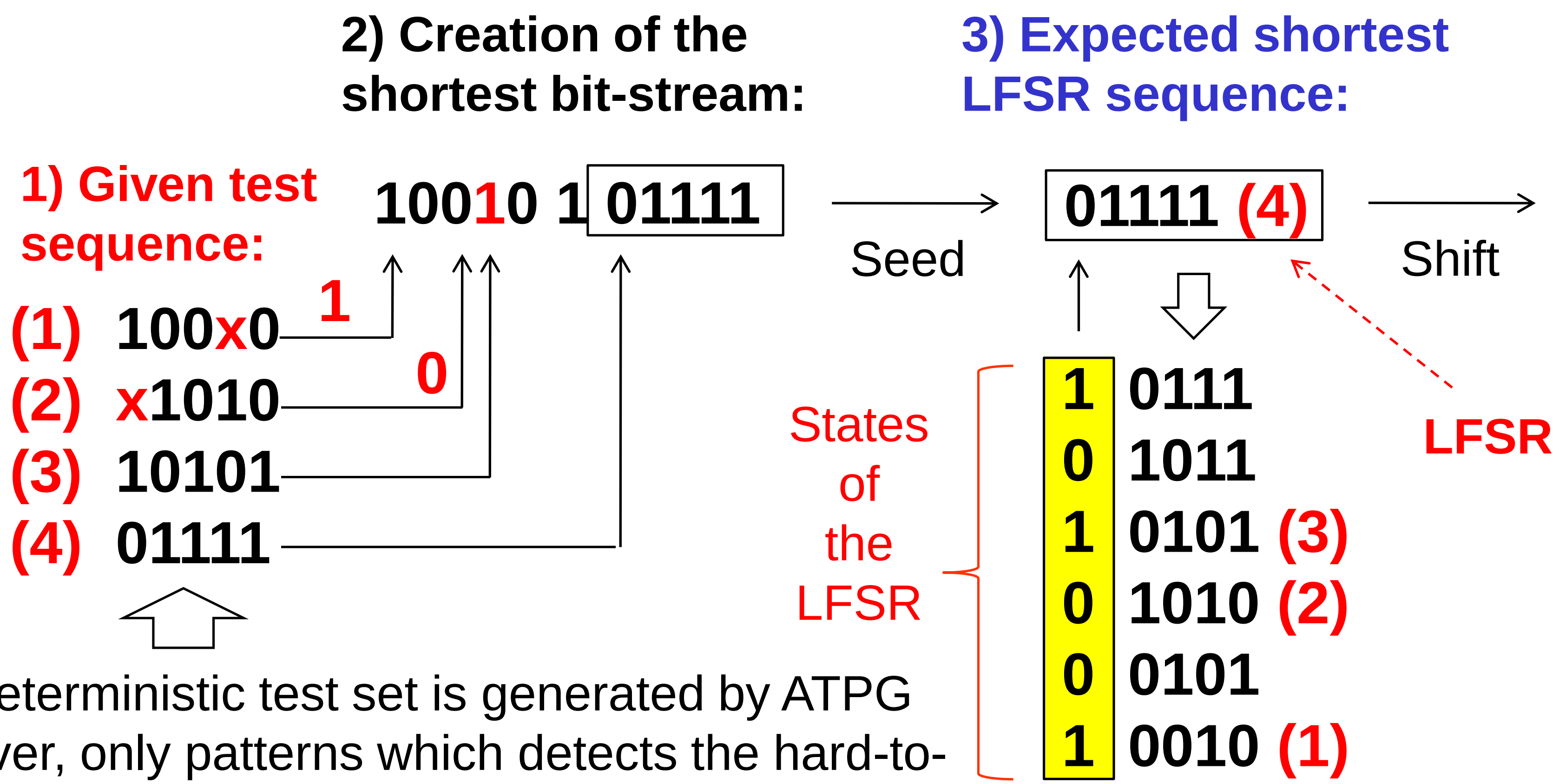
- **Test application:**



- 1 10 0 1 0 1 0 01 01 1001
- Number of clocks = 8 < 20

Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence



1) Given test sequence:

(1) 100x0

(2) x1010

(3) 10101

(4) 01111

2) Creation of the shortest bit-stream:

10010 1 01111

1

0

3) Expected shortest LFSR sequence:

Seed

01111 (4)

Shift

States of the LFSR

1 0111

0 1011

1 0101 (3)

0 1010 (2)

0 0101

1 0010 (1)

LFSR

This deterministic test set is generated by ATPG
However, only patterns which detects the hard-to-test faults can be chosen

Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence

System of linear equations: $a_k x_1 \oplus b_k x_2 \oplus c_k x_3 \oplus d_k x_4 \oplus e_k x_5 = f_k$

Expected shortest LFSR sequence:

01111 (4)

1 0111
0 1011
1 0101 (3)
0 1010 (2)
0 0101
1 0010 (1)

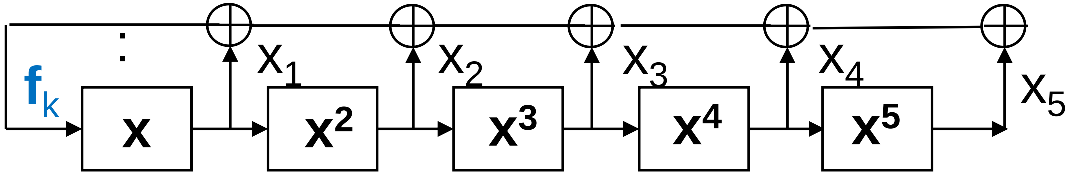
Currrent state of the LFSR

k=1,2…6
j=1,2,,,5

$$\begin{array}{c} a_kb_kc_kd_ke_k \\ \begin{vmatrix} 01111\\ 10111\\ 01011\\ 10101\\ 01010\\ 00101 \end{vmatrix} \end{array} k\downarrow \; \times \; \begin{array}{c} x_j \\ \begin{vmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5 \end{vmatrix} \end{array} j\downarrow \; = \; \begin{array}{c} f_k \\ \begin{vmatrix} 1\\ 0\\ 1\\ 0\\ 0\\ 1 \end{vmatrix} \end{array} k\downarrow$$

Next input signal into LFSR

We are looking for the values of x_i



Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence

System of linear equations:

$$\begin{array}{c} 1\\ 2\\ 3\\ 4\\ 5\\ 6 \end{array} \begin{vmatrix} 01111\\ 10111\\ 01011\\ 10101\\ 01010\\ 00101 \end{vmatrix} \times \begin{vmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5 \end{vmatrix} = \begin{vmatrix} 1\\ 0\\ 1\\ 0\\ 0\\ 1 \end{vmatrix}$$

$$a_k x_1 \oplus b_k x_2 \oplus c_k x_3 \oplus d_k x_4 \oplus e_k x_5 = f_k,\ k = 1,2,..,6$$

Solving the equation by Gaussian elimination **with swapping of rows**

Rows:

1,2,4,6
4,6
1,3
2,4
1,3,6

Results:

$$\begin{vmatrix} 01000\\ 10000\\ 00100\\ 00010\\ 00001\\ 00001 \end{vmatrix} \times \begin{vmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5 \end{vmatrix} = \begin{vmatrix} 0\\ 1\\ 0\\ 0\\ 1\\ 1 \end{vmatrix} \begin{array}{c} f_k\\ \\ f_2\\ f_3\\ \\ \\ \\ \end{array}$$

Examples:

(4) 10101 0
(6) 00101 1
(4 ⊕ 6) **10000 1** ⇨ **k=2**

(1) 01111 1
(3) 01011 1
(1 ⊕ 3) **00100 0** ⇨ **k=3**

4) Solution:

x_1	x_2	x_3	x_4	x_5
1	0	0	0	1

Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence

Solving the equation by Gaussian elimination with swapping of rows

$a_k x_1 \oplus b_k x_2 \oplus c_k x_3 \oplus d_k x_4 \oplus e_k x_5 = f_k$, $k = 1,2,..,6$

$$\begin{vmatrix} 01000 \\ 10000 \\ 00100 \\ 00010 \\ 00001 \\ 00001 \end{vmatrix} \times \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{vmatrix} = \begin{array}{c} f_k \\ \begin{vmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{vmatrix} \end{array}$$

f_k

$\wedge \; \begin{array}{c} 01000 \\ x_1x_2x_3x_4x_5 \end{array} = 0$ ⇨ $x_2 = 0$

$\wedge \; \begin{array}{c} 10000 \\ x_1x_2x_3x_4x_5 \end{array} = 1$ ⇨ $x_1 = 1$

$\wedge \; \begin{array}{c} 00100 \\ x_1x_2x_3x_4x_5 \end{array} = 0$ ⇨ $x_3 = 0$

$\wedge \; \begin{array}{c} 00010 \\ x_1x_2x_3x_4x_5 \end{array} = 0$ ⇨ $x_4 = 0$

$\wedge \; \begin{array}{c} 00001 \\ x_1x_2x_3x_4x_5 \end{array} = 1$ ⇨ $x_5 = 1$

4) Solution:

x_1	x_2	x_3	x_4	x_5
1	0	0	0	1

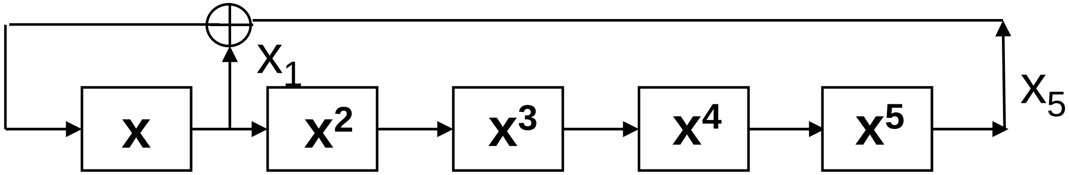
Deterministic Synthesis of LFSR

Embedding deterministic test patterns into LFSR sequence:

4) Solution: $x_1\ x_2\ x_3\ x_4\ x_5$

$1\ \ 0\ \ 0\ \ 0\ \ 1$

5) Polynomial: $x^5 + x + 1$ Seed: **01111**



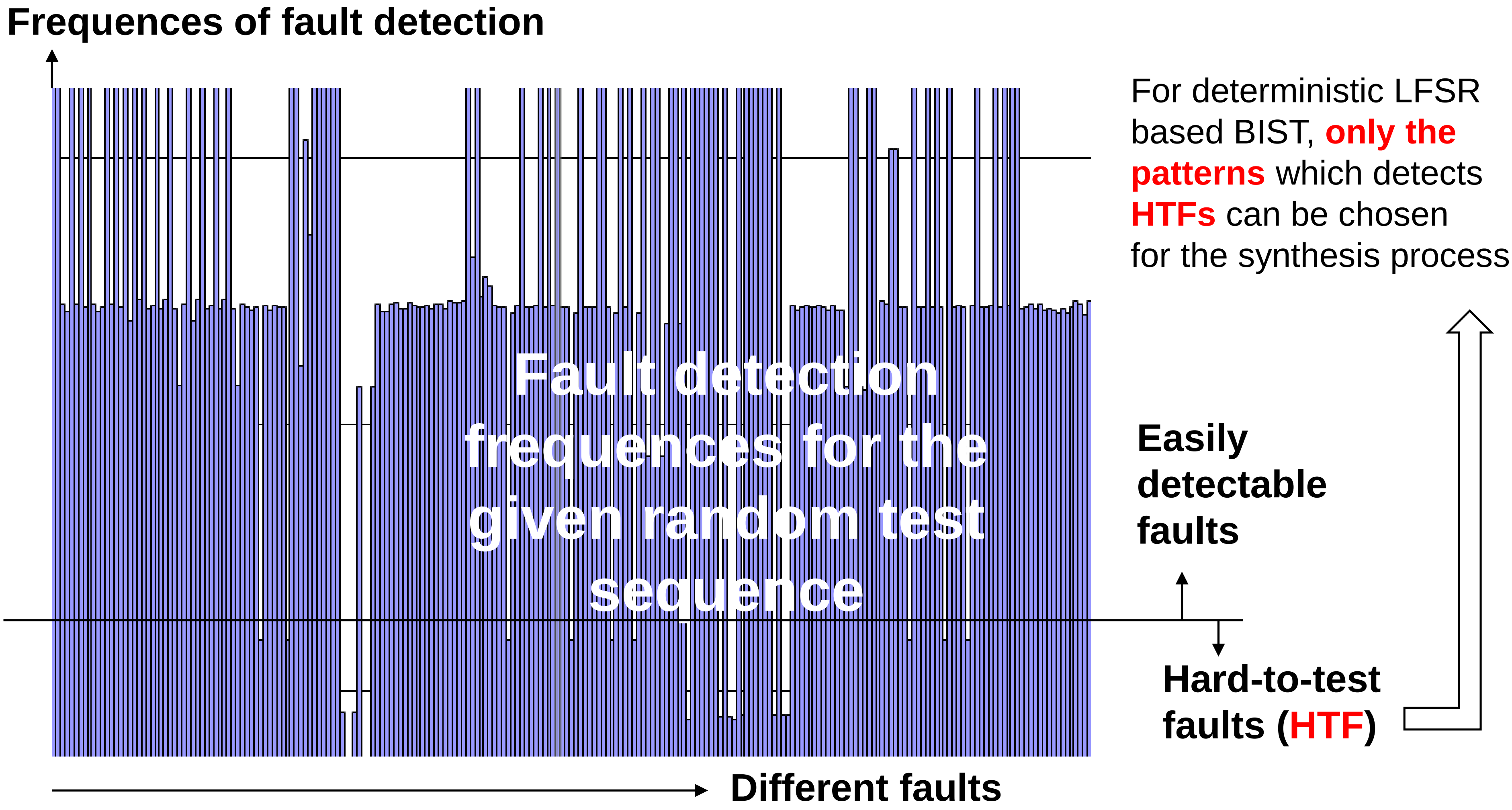
LFSR sequence:

(1) 01111 (4)
(2) 10111
(3) 01011
(4) 10101 (3)
(5) 01010 (2)
(6) 00101
(7) 10010 (1)

Given deterministic test sequence:

(1) 100x0
(2) x1010
(3) 10101
(4) 01111

Which Test Patterns to Select for as HTF?

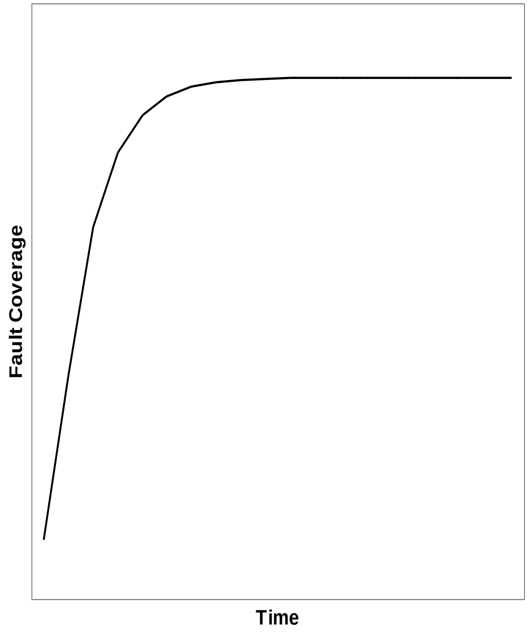


For deterministic LFSR based BIST, **only the patterns** which detects **HTFs** can be chosen for the synthesis process

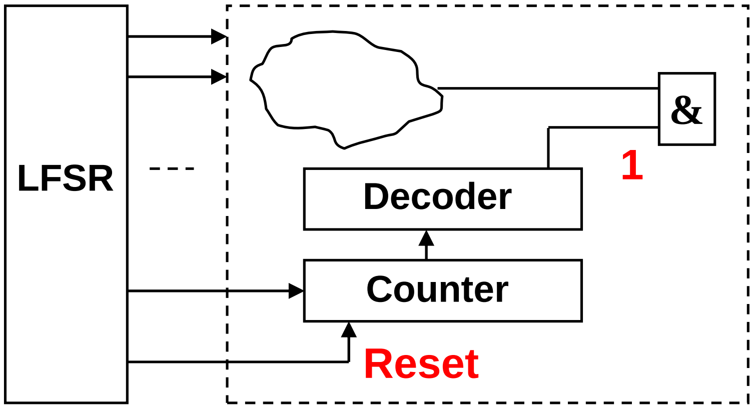
Other Problems with Pseudorandom Test

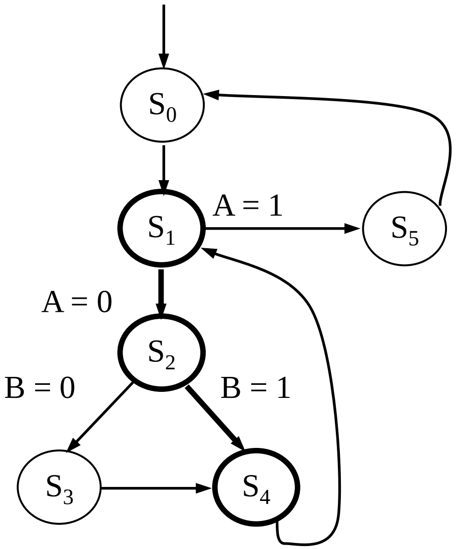
The main motivations of using random patterns are:

- **low generation cost**
- **high initial efeciency**



Problem: **low fault coverage**



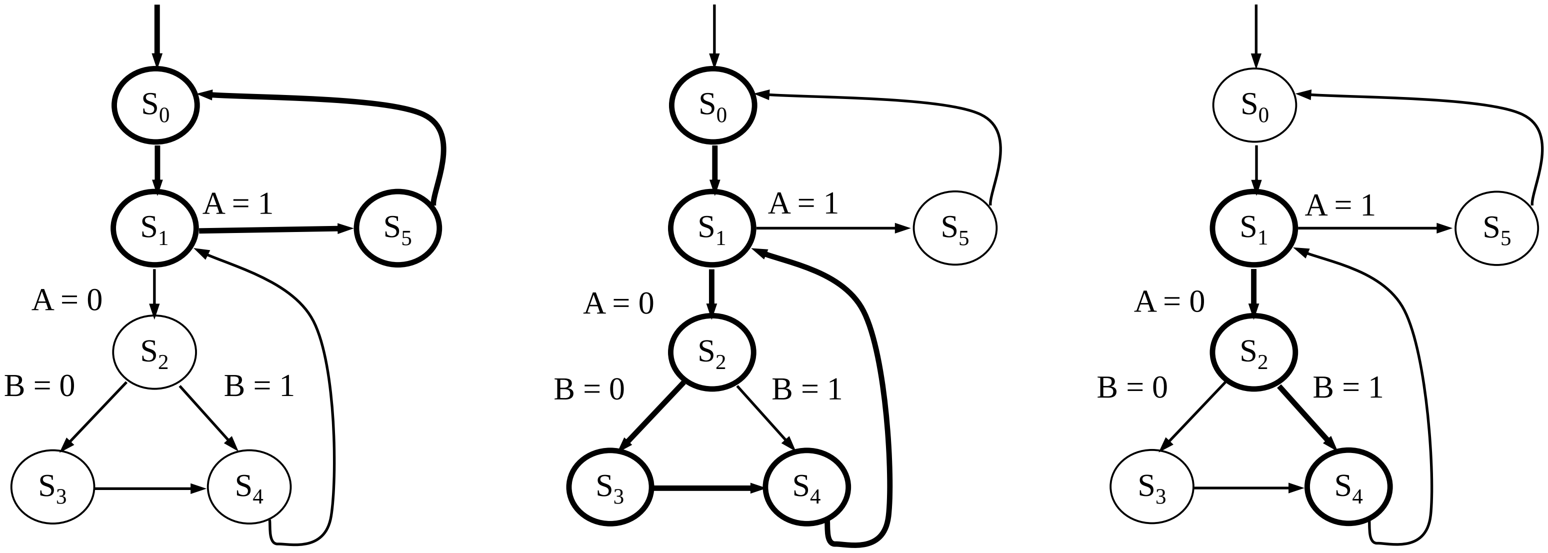


If Reset = 1 signal has probability 0,5 then counter will not work and
1 for AND gate may never be produced

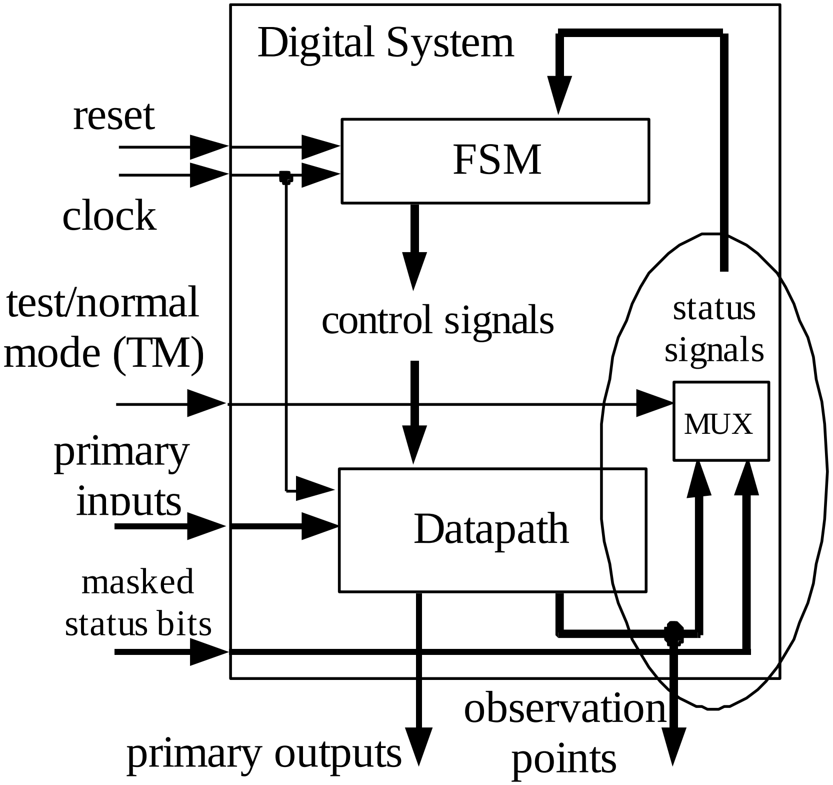
Sequential BIST

A DFT technique of BIST for sequential circuits is proposed

The approach proposed is based on all-branches coverage metrics which is known to be more powerful than all-statement coverage

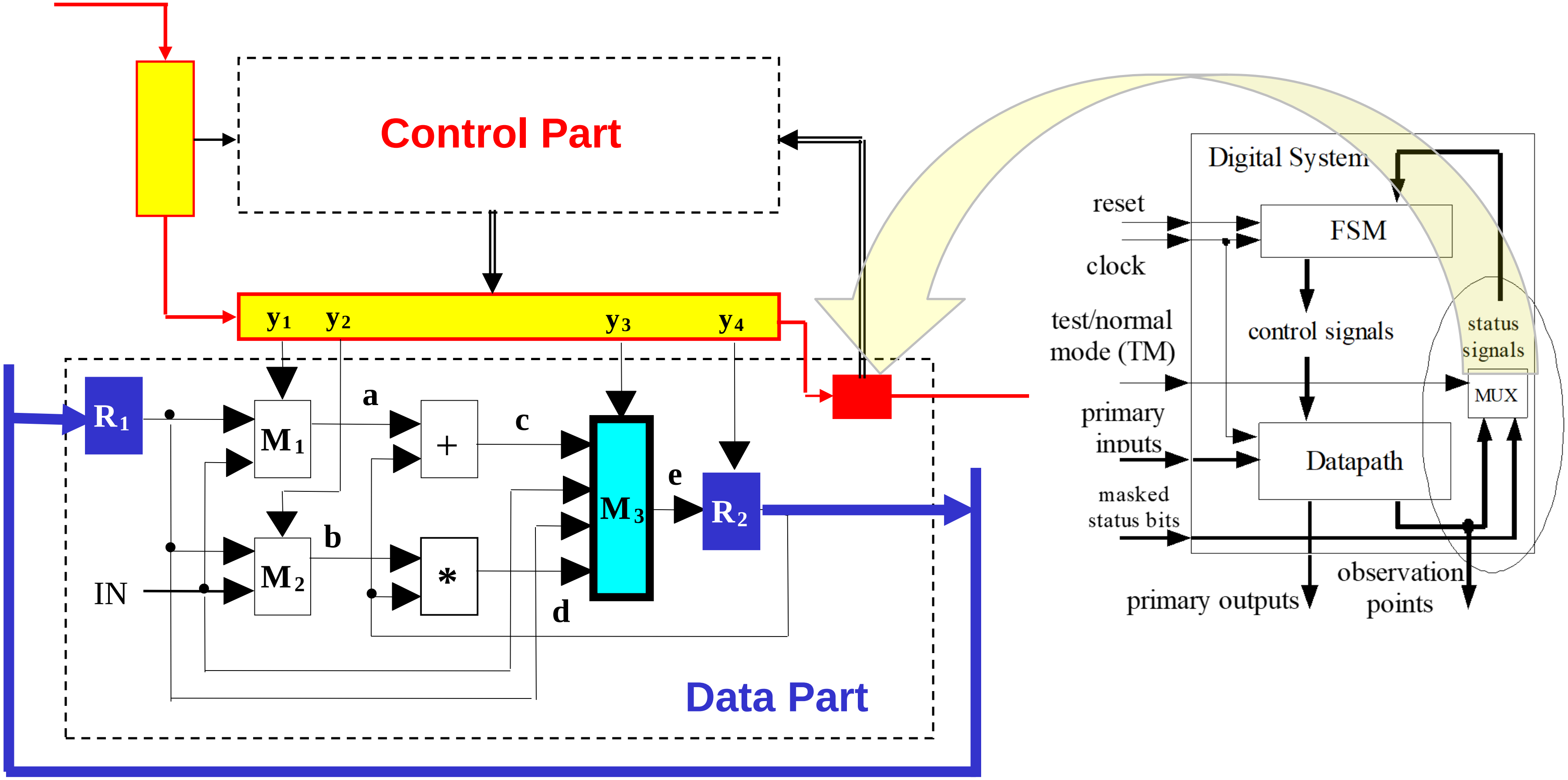


Sequential BIST



- Status signals entering the control part are made controllable
- In the test mode we can force the UUT to traverse all the branches in the FSM state transition graph
- The proposed idea of architecture requires small device area overhead since a simple controller can be implemented to manipulate the control signals

Example for Sequential BIST



BIST: Different Techniques

Pseudorandom Test generation by LFSR:

Full identification is achieved only after 2^n input combinations have been tried out (exhaustive test)

$$\frac{1}{2^{2^n}-1}\sum_{i=1}^{m} 2^{2^n-1},$$

$$1 \le m \le 2^n$$

A better fault model (stuck-at-0/1) may limit the number of partitions necessary

Pseudorandom testing of sequential circuits:

The following rules suggested:

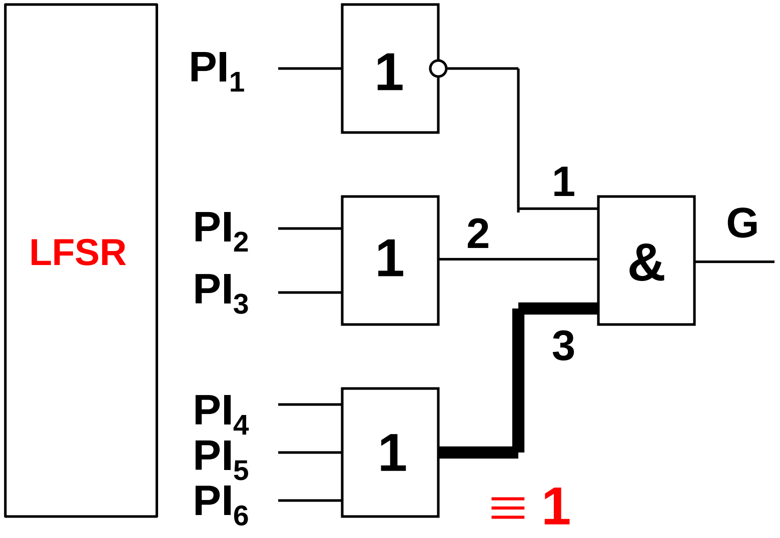
- **clock-signals should not be random**
- **control signals such as reset, should be activated with low probability**
- **data signals are chosen randomly**

Microprocessor testing

- **A test generator picks randomly an instruction and generates random data patterns**
- **By repeating this sequence a specified number of times it will produce a test program which will test the microprocessor by randomly exercising its logic**

BIST: Weighted pseudorandom test

Calculation of signal probabilities:



For PI_1 **:** **P = 0.15**

For PI_2 and PI_3 **:** **P = 0.6**

For PI_4 - PI_6 **:** **P = 0.4**

Probability of detecting the fault ≡1 at the input 3 of the gate G:

1) equal probabilities (p = 0.5):

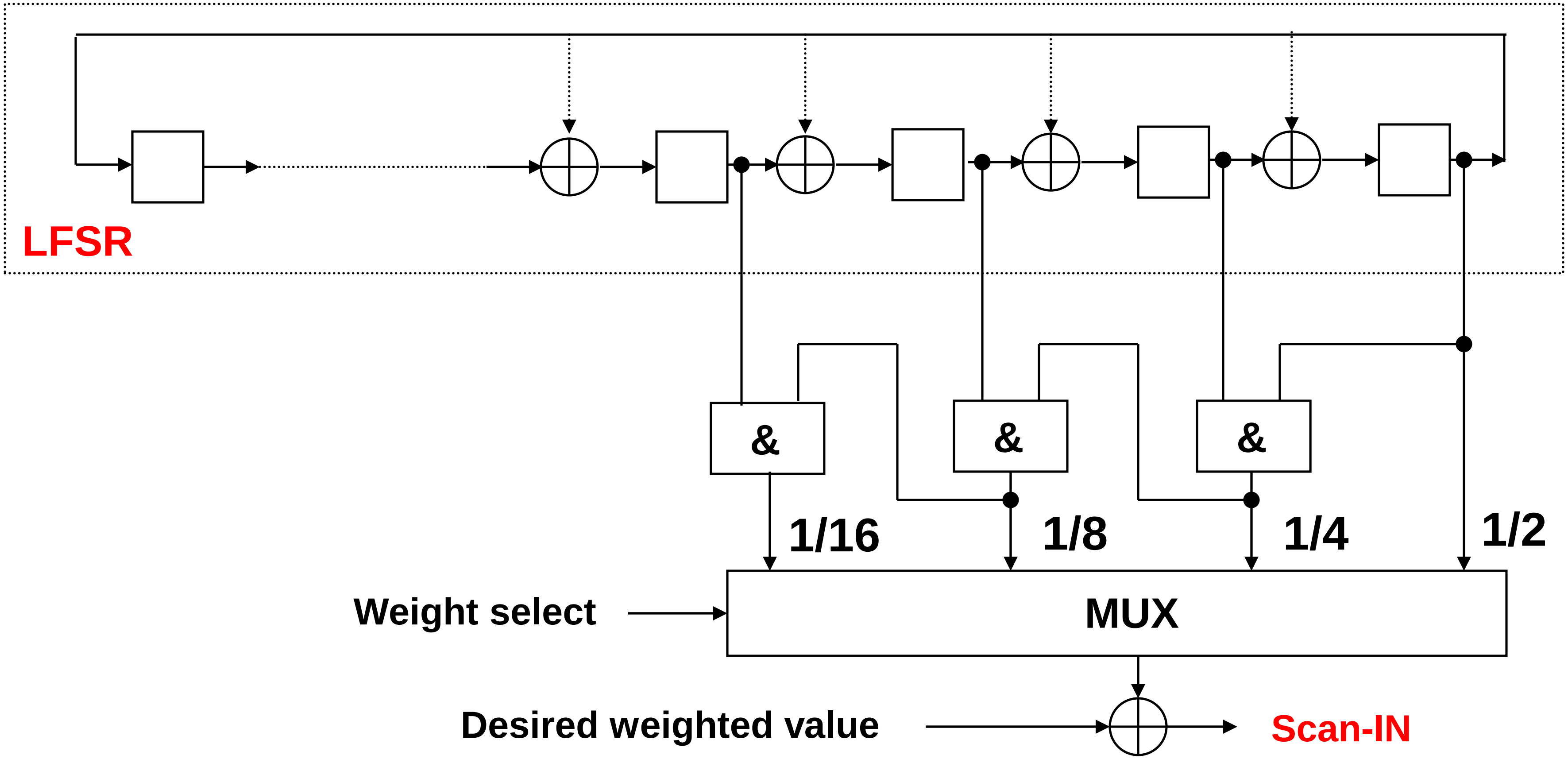
$$P = 0.5 * (0.25 + 0.25 + 0.25) * 0.5^3 =$$
$$= 0.5 * 0.75 * 0.125 =$$
$$= 0.046$$

2) weighted probabilities:

$$P = 0.85 *$$
$$* (0.6 * 0.4 + 0.4 * 0.6 + 0.6^2) *$$
$$* 0.6^3 =$$
$$= 0.85 * 0.84 * 0.22 =$$
$$= 0.16$$

BIST: Weighted pseudorandom test

Hardware implementation of weight generator

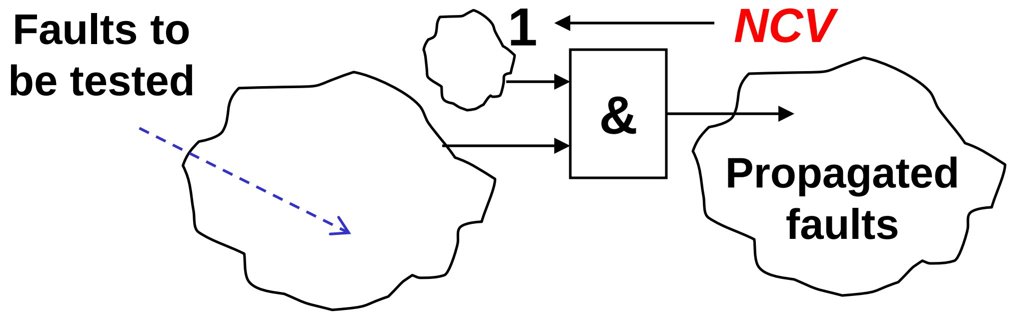


BIST: Weighted pseudorandom test

Problem: random-pattern-resistant faults

Solution: weighted pseudorandom testing

The probabilities of pseudorandom signals are weighted, the weights are determined by circuit analysis

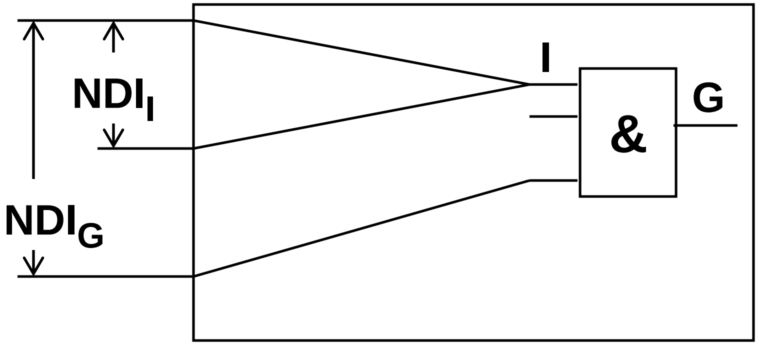


NCV – non-controlling value

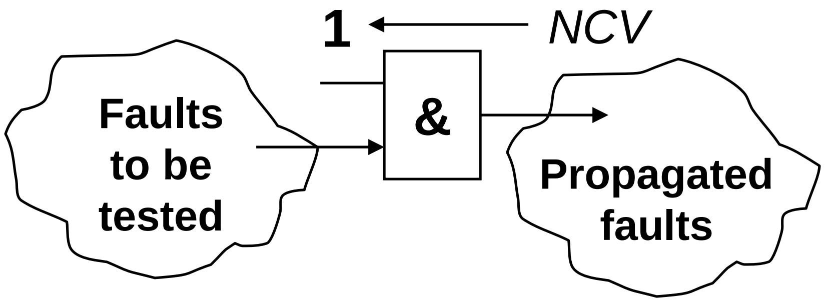
The more faults that must be tested through a gate input, the more the other inputs should be weighted to NCV

NDI - number of primary inputs for each gate determined by the back-trace cone

NDI - relative measure of the number of faults to be detected through the gate



BIST: Weighted pseudorandom test

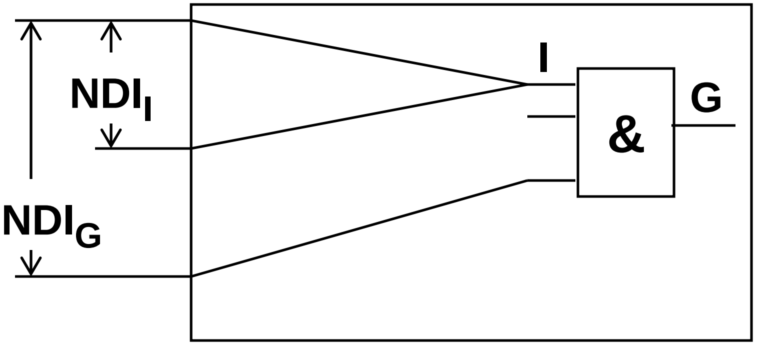


$R_I = NDI_G / NDI_I$

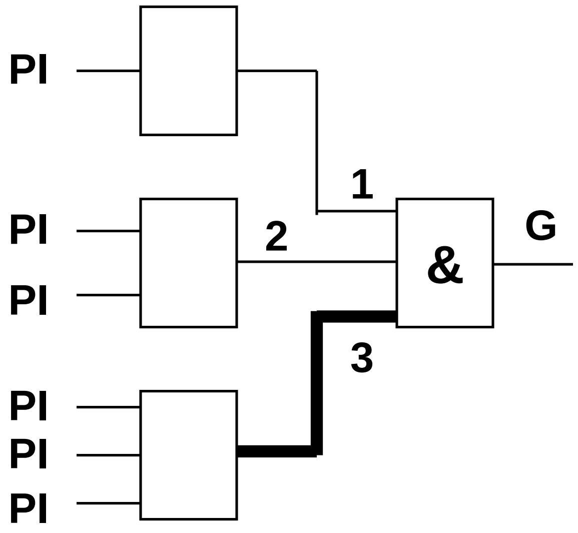
R_I - the desired ratio of the NCV (1) to the controlling value (0) for each gate input

NCV - noncontrolling value

The more faults that must be tested through a gate input, the more the other inputs should be weighted to NCV



BIST: Weighted pseudorandom test



Example:

$R_1 = NDI_G / NDI_I = 6/1 = 6$

$R_2 = NDI_G / NDI_I = 6/2 = 3$

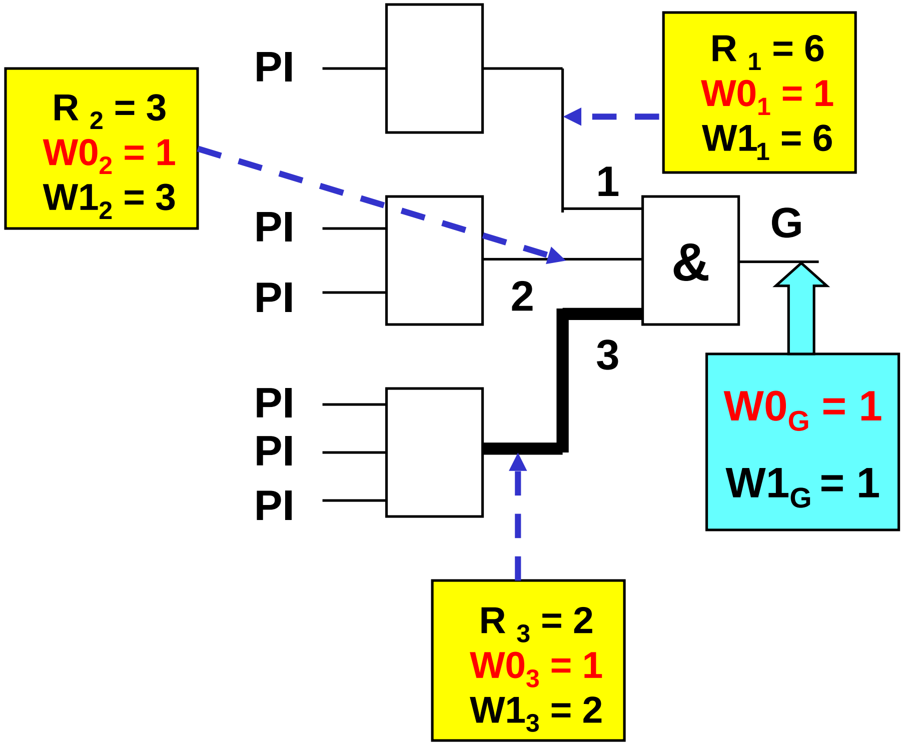
$R_3 = NDI_G / NDI_I = 6/3 = 2$

More faults must be detected through the third input than through others

This results in the other inputs being weighted more heavily towards NCV

BIST: Weighted pseudorandom test

Calculation of signal weights:



W0, W1 - weights of the signals

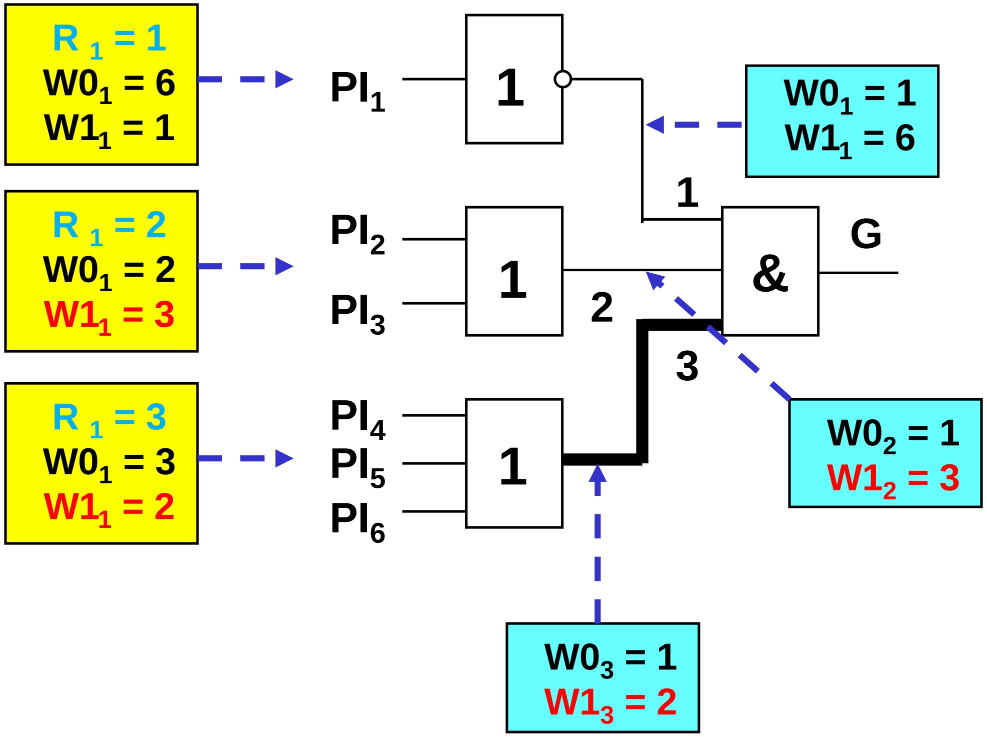
are calculated by backtracking

Calculation of W0, W1 for inputs

Function	$W0_{IN}$	$W1_{IN}$
AND	$W0_G$	$R_I * W1_G$
NAND	$W1_G$	$R_I * W0_G$
OR	$R_I * W0_G$	$W1_G$
NOR	$R_I * W1_G$	$W0_G$

BIST: Weighted pseudorandom test

Calculation of signal weights:



$R_1 = 1$
$W0_1 = 6$
$W1_1 = 1$

$R_1 = 2$
$W0_1 = 2$
$W1_1 = 3$

$R_1 = 3$
$W0_1 = 3$
$W1_1 = 2$

$W0_1 = 1$
$W1_1 = 6$

$W0_2 = 1$
$W1_2 = 3$

$W0_3 = 1$
$W1_3 = 2$

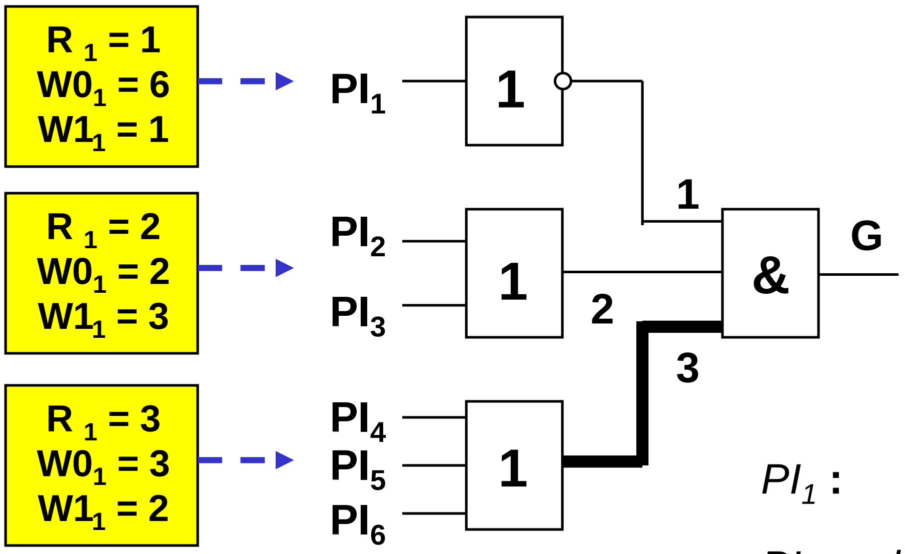
Backtracing from all the outputs to all the inputs of the given cone

Weights are calculated for all gates and inputs

Function	$W0_I$	$W1_I$
OR	$R_I * W0_G$	$W1_G$
NOR	$R_I * W1_G$	$W0_G$

BIST: Weighted pseudorandom test

Calculation of signal probabilities:



$R_1 = 1$
$W0_1 = 6$
$W1_1 = 1$

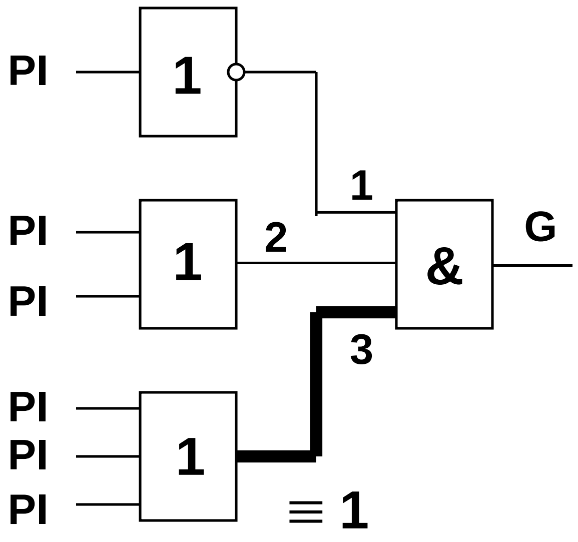
$R_1 = 2$
$W0_1 = 2$
$W1_1 = 3$

$R_1 = 3$
$W0_1 = 3$
$W1_1 = 2$

PI_1 :	**W0 = 6 W1 = 1**	**P1 = 1/7 = 0.15**
PI_2 and PI_3 :	**W0 = 2 W1 = 3**	**P1 = 3/5 = 0.6**
PI_4 - PI_6 :	**W0 = 3 W1 = 2**	**P1 = 2/5 = 0.4**

BIST: Weighted pseudorandom test

Calculation of signal probabilities:



For PI_1 **:** **P1 = 0.15**

For PI_2 and PI_3 **:** **P1 = 0.6**

For PI_4 - PI_6 **:** **P1 = 0.4**

Probability of detecting the fault ≡1 at the input 3 of the gate G:

1) equal probabilities (p = 0.5):

$$P = 0.5 * (0.25 + 0.25 + 0.25) * 0.5^3 =$$
$$= 0.5 * 0.75 * 0.125 =$$
$$= 0.046$$

2) weighted probabilities:

$$P = 0.85 *$$
$$* (0.6 * 0.4 + 0.4 * 0.6 + 0.6^2) *$$
$$* 0.6^3 =$$
$$= 0.85 * 0.84 * 0.22 =$$
$$= 0.16$$

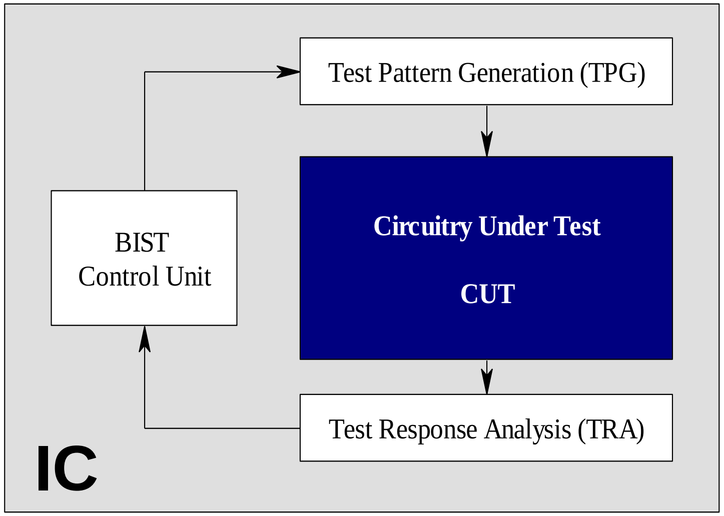
The Main BIST Problems

- On circuit
  - Test pattern generation
  - Response verification
- Random pattern generation,

  Very long tests

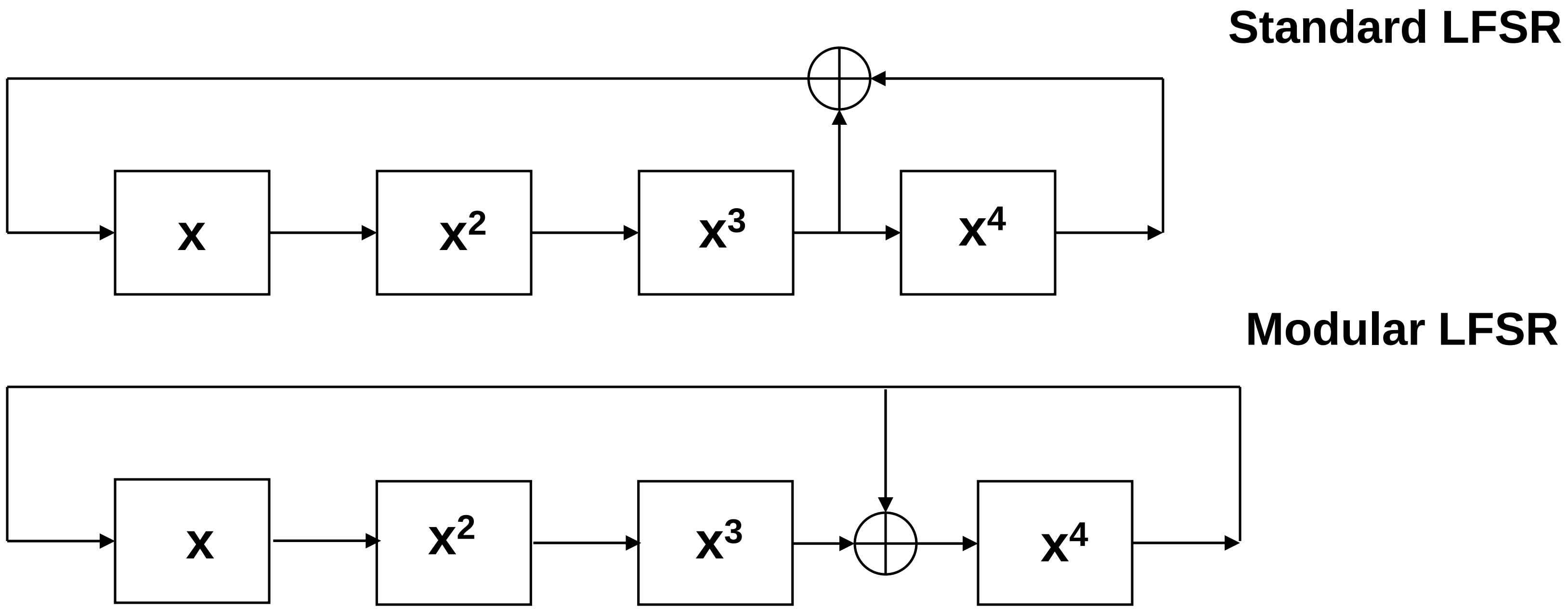
  Hard-to-test faults
- Response compression

  Aliasing of results



Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:



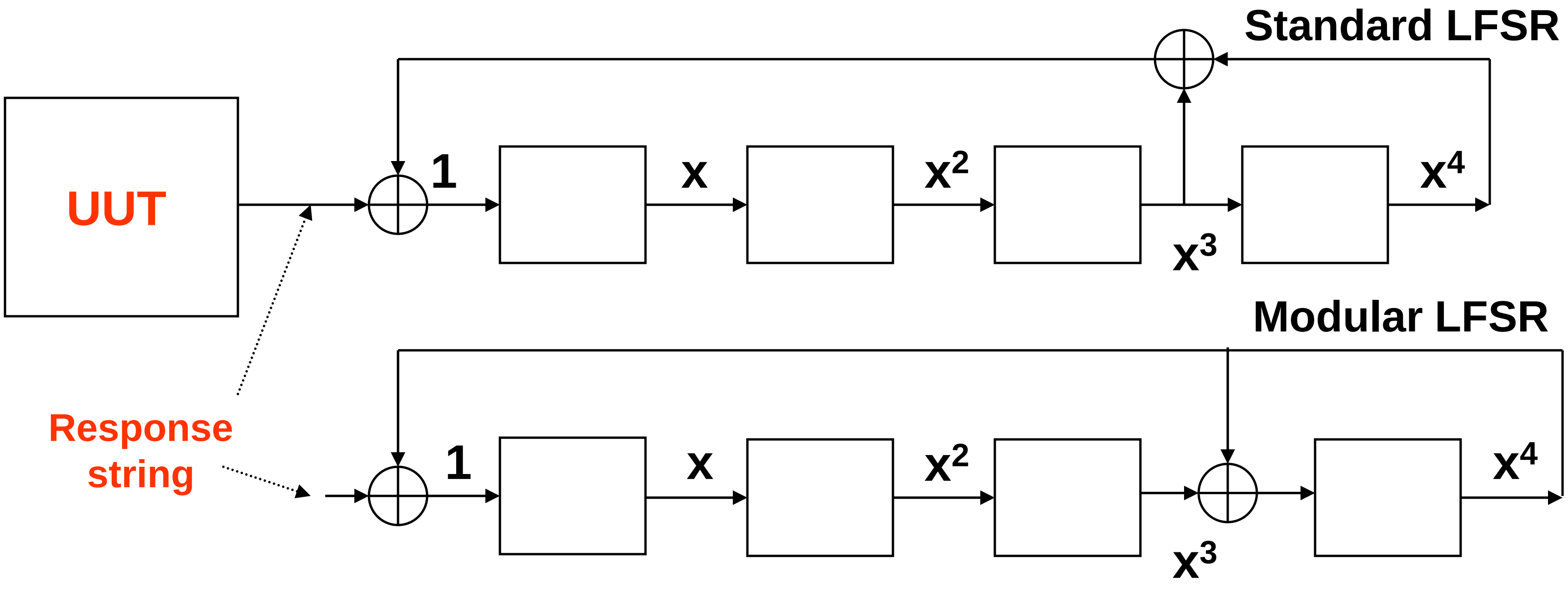
Standard LFSR

Modular LFSR

Polynomial: $P(x) = x^4 + x^3 + 1$

BIST: Signature Analysis

Signature analyzer:



Response in compacted by LFSR

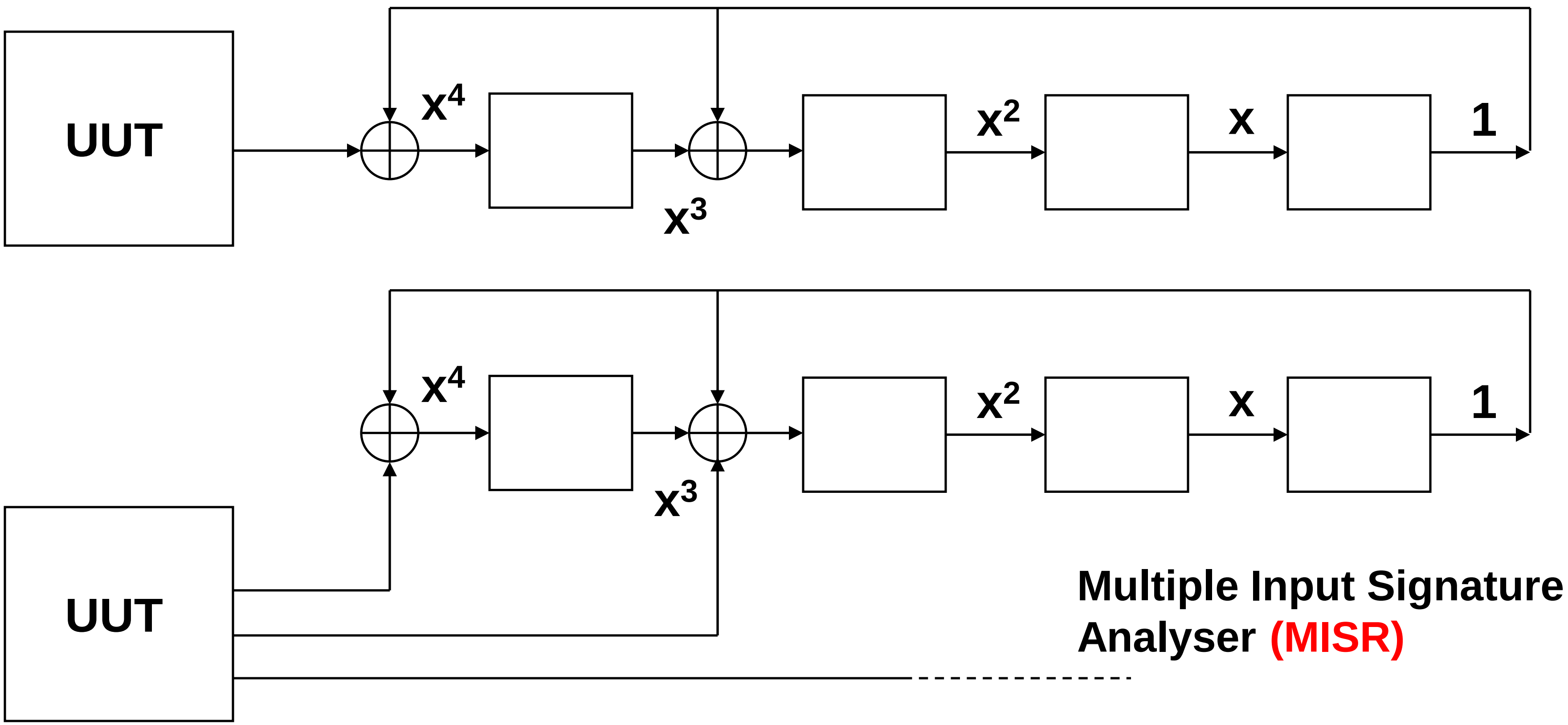
The content of LFSR after test is called ***signature***

Polynomial: $P(x) = x^4 + x^3 + 1$

BIST: Signature Analysis

Parallel Signature Analyzer:

Single Input Signature Analyser

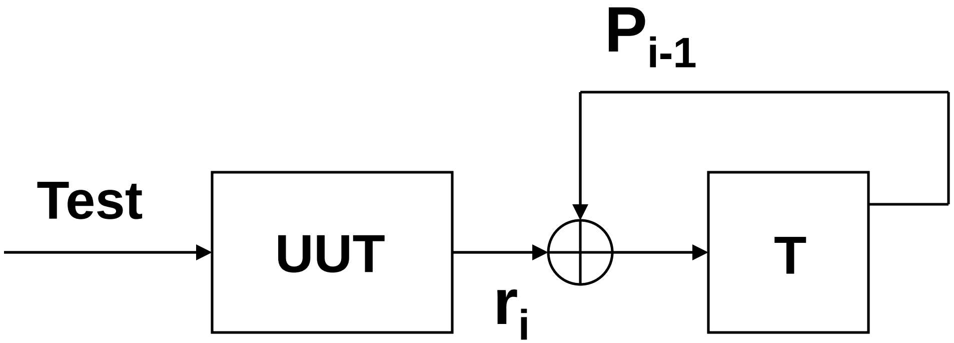


Multiple Input Signature Analyser **(MISR)**

Special Cases of Response Compression

1. **Parity checking**

$$P(R) = (\sum_{i=1}^{m} r_i) \bmod 2$$

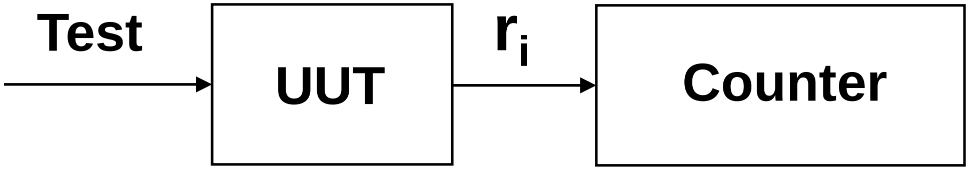


2. **One counting**

$$P(R) = \sum_{i=1}^{m} r_i$$

3. **Zero counting**

$$P(R) = \sum_{i=1}^{m} \overline{r_i}$$

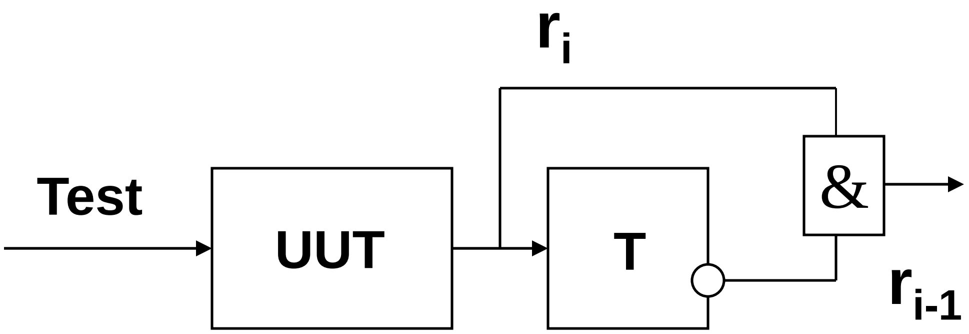


Special Cases of Response Compression

4. Transition counting

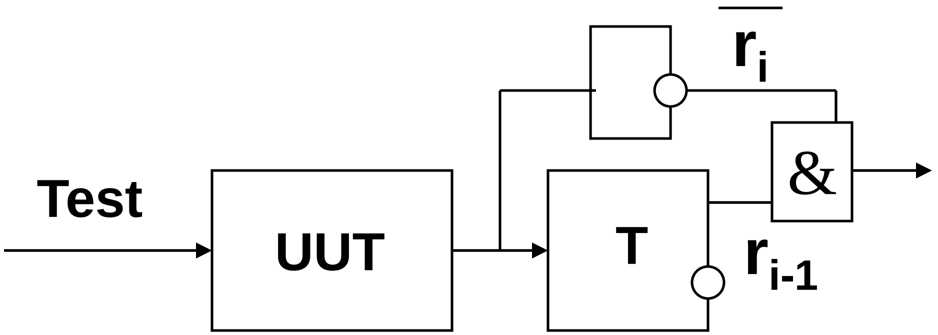
a) Transition 0→1

$$P(R) = \sum_{i=2}^{m} (\overline{r_{i-1}} r_i)$$



b) Transition 1→0

$$P(R) = \sum_{i=2}^{m} (r_{i-1} \overline{r_i})$$



5. Signature analysis

Theory of LFSR

The principles of CRC (Cyclic Redundancy Coding) are used in LFSR based test response compaction

Coding theory treats binary strings as polynomials:

$R = r_{m-1}\, r_{m-2} \ldots r_1\, r_0$ - m-bit binary sequence (binary string)

$R(x) = r_{m-1}\, x^{m-1} + r_{m-2}\, x^{m-2} + \ldots + r_1\, x + r_0$ - polynomial in x

Example:

11001 → $R(x) = x^4 + x^3 + 1$

Only the coefficients are of interest, not the actual value of x

However, for x = 2, R(x) is the decimal value of the bit string

Theory of LFSR

Arithmetic of coefficients:

- linear algebra over the field of 0 and 1: all integers mapped into either 0 or 1
- mapping: representation of any integer n by remainder r resulting from the division of n by 2:

$$n = 2m + r, \quad r \in \{ 0,1 \} \qquad \text{or} \qquad r = n \text{ (modulo 2)}$$

Linear - refers to the arithmetic unit (modulo-2 adder), used in CRC generator (linear, since each bit has equal weight upon the output)

Examples (addition, multiplication):

$$\begin{array}{rccccccccc}
 & x^4 & + & x^3 & & & + & x & + & 1 \\
+ & x^4 & & & + & x^2 & + & x & & \\
\hline
 & & & x^3 & + & x^2 & & & + & 1
\end{array}$$

$$\begin{array}{rccccccccccc}
 & & & x^4 & + & x^3 & & & + & x & + & 1 \\
 & & & & & & * & x & + & 1 & & \\
\hline
x^5 & + & x^4 & & & & + & x^2 & + & x & & \\
 & & x^4 & + & x^3 & & & & + & x & + & 1 \\
\hline
x^5 & & & + & x^3 & + & x^2 & & & & + & 1
\end{array}$$

Theory of LFSR

Characteristic Polynomials:

$$G(x) = c_0 + c_1x + c_2x^2 + \ldots + c_mx^m + \ldots = \sum_{m=0}^{\infty} c_mx^m$$

$$\begin{array}{rrl}
 & & x^2+x+1 \quad \dashleftarrow \textbf{Quotient}\\
\textbf{Divider} \dashrightarrow & x^2+1 & \big|\ \overline{x^4+x^3\qquad\qquad\quad +1} \quad \dashleftarrow \textbf{Dividend}\\
 & & x^4\qquad\quad +x^2\\
 & & \qquad\overline{x^3+x^2\qquad +1}\\
 & & \qquad x^3\qquad +x\\
 & & \qquad\qquad\overline{x^2+x+1}\\
 & & \qquad\qquad x^2\qquad +1\\
 & & \qquad\qquad\overline{\qquad\quad x\ \ } \quad \dashleftarrow \textbf{Remainder}
\end{array}$$

Division of polynomials

BIST: Signature Analysis

Division of one polynomial P(x) by another G(x) produces a quotient polynomial Q(x), and if the division is not exact, a remainder polynomial R(x)

$$\frac{P(x)}{G(x)} = Q(x) + \frac{R(x)}{G(x)}$$

Example:

$$\frac{P(x)}{G(x)} = \frac{x^7 + x^3 + x}{x^5 + x^3 + x + 1} = x^3 + x^2 + 1 + \frac{x^2 + 1}{x^5 + x^3 + x + 1}$$

Remainder R(x) is used as a check word in data transmission

The transmitted code consists of the message P(x) followed by the check word R(x)

Upon receipt, the reverse process occurs: the message P(x) is divided by known G(x), and a mismatch between R(x) and the remainder from the division indicates an error

BIST: Signature Analysis

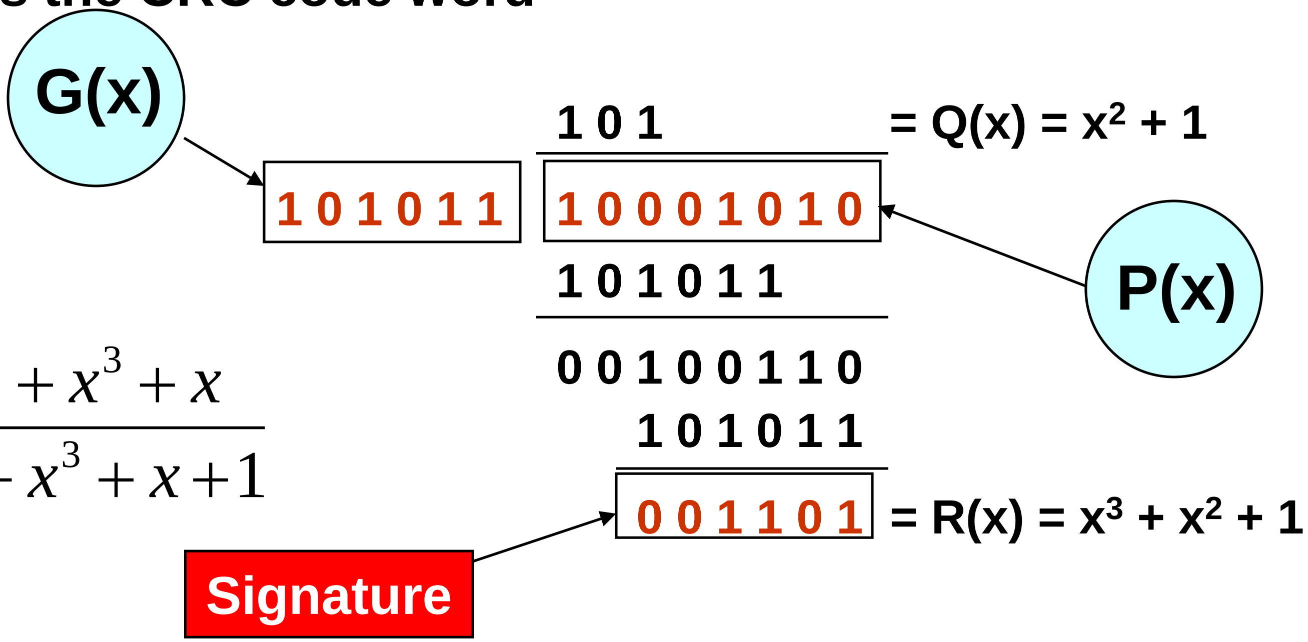
In signature testing we mean the use of CRC encoding as the **data compressor G(x)** and the use of the **remainder R(x)** as the **signature** of the **test response** string P(x) from the UUT

$$\frac{P(x)}{G(x)} = Q(x) + \frac{R(x)}{G(x)}$$

Signature is the CRC code word

Example:

$$\frac{P(x)}{G(x)} = \frac{x^7 + x^3 + x}{x^5 + x^3 + x + 1}$$



G(x)

1 0 1 0 1 1

1 0 1 = Q(x) = $x^2 + 1$

1 0 0 0 1 0 1 0

P(x)

1 0 1 0 1 1

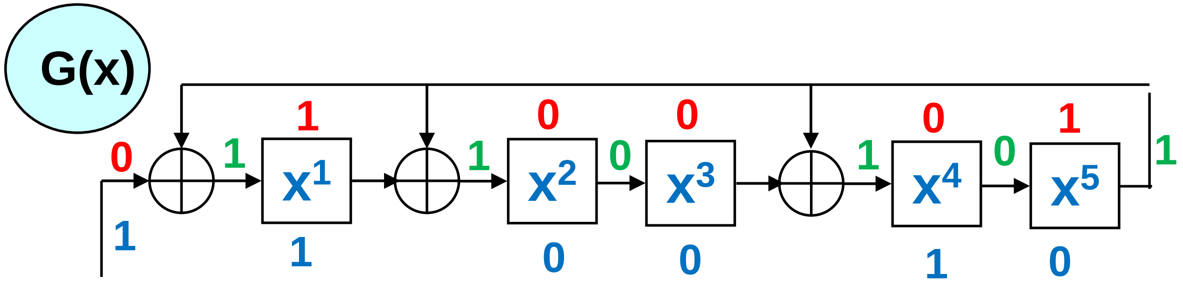
0 0 1 0 0 1 1 0

1 0 1 0 1 1

0 0 1 1 0 1 = R(x) = $x^3 + x^2 + 1$

Signature

BIST: Hardware for Signature Analysis



IN: 01 010001 → Shifted into LFSR

P(x)

Dvision process can be mechanized using LFSR

Divisor polynomial G(x) is defined by the feedback connections

Compressor

G(x) → 1 0 1 0 1 1

x_5 x_4 x_3 x_2 x_1

1 0 1

1 0 0 0 1 0 1 0 ← Response P(x)

1 0 1 0 1 1

0 0 1 0 0 1 1 0 ← x^5

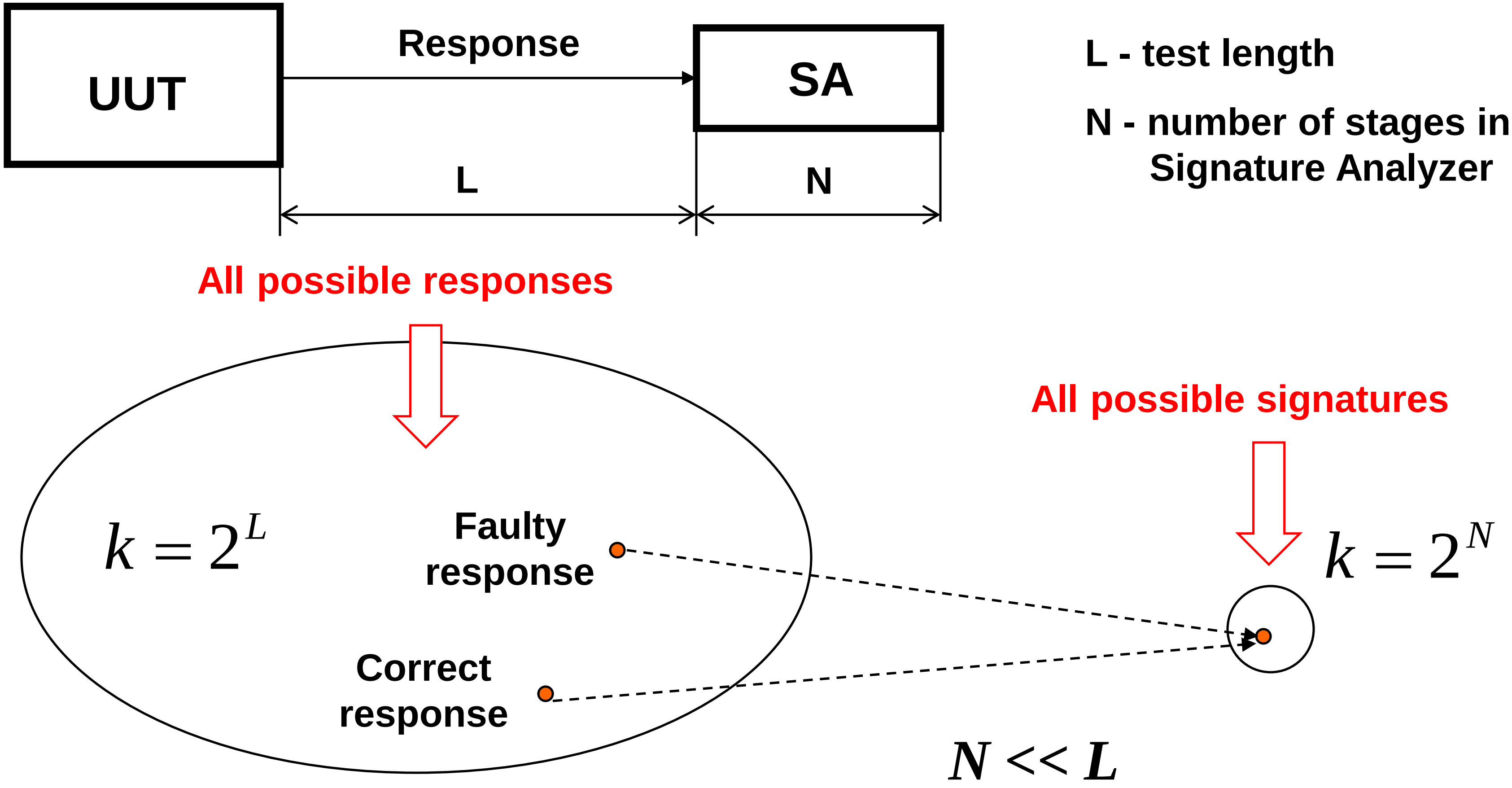
1 0 1 0 1 1

0 0 1 1 0 1 = R(x) = $x^3 + x^2 + 1$ ← Signature

$$\frac{P(x)}{G(x)} = \frac{x^7 + x^3 + x}{x^5 + x^3 + x + 1}$$

BIST: Signature Analysis

Aliasing:



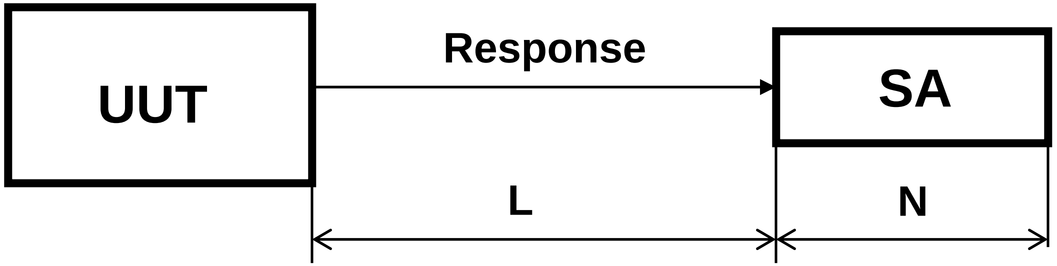
L - test length

N - number of stages in
Signature Analyzer

$N << L$

BIST: Signature Analysis

Aliasing:



L - test length

N - number of stages in Signature Analyzer

$k = 2^L$ - **number of different possible responses**

No aliasing is possible for those strings with $L - N$ **leading zeros since they are represented by polynomials of degree** $N - 1$ **that are not divisible by characteristic polynomial of LFSR**

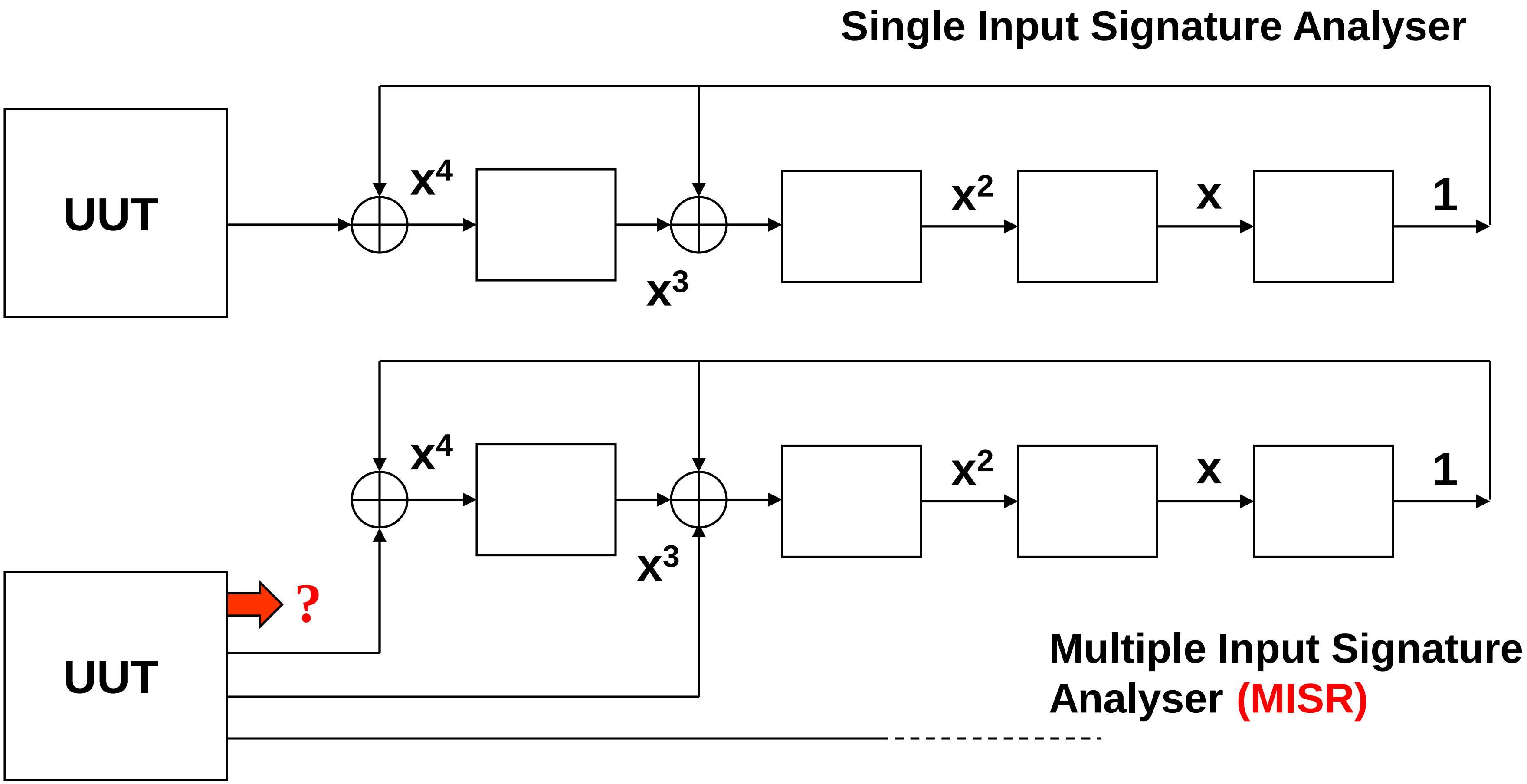
$$\underbrace{000000000000000 \ldots 00000\,\underbrace{XXXXX}_{N}}_{L}$$

$2^{L-N} - 1$ ---- **Aliasing is possible**

Probability of aliasing: $$P = \frac{2^{L-N} - 1}{2^L - 1} \xrightarrow{L >> 1} \boxed{P = \frac{1}{2^N}}$$

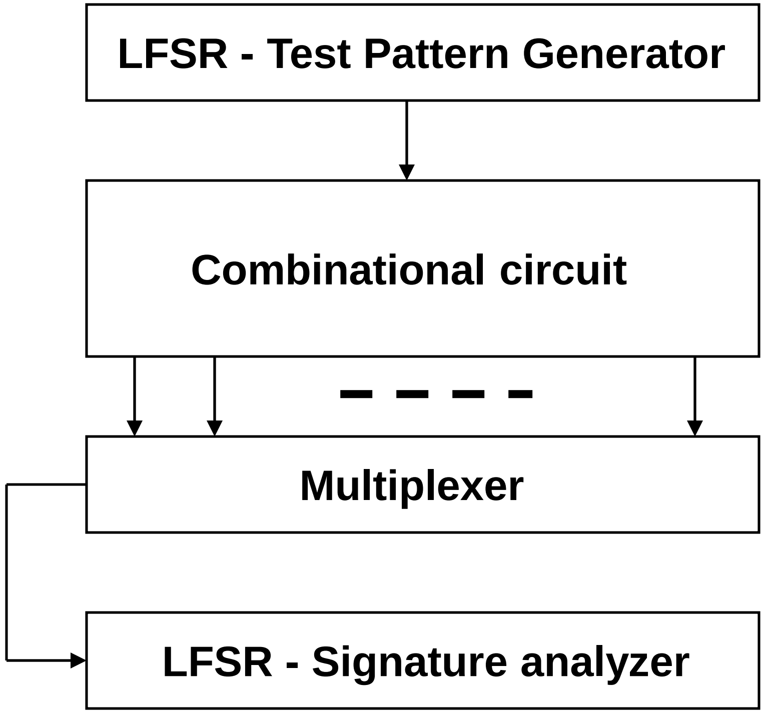
BIST: Signature Analysis

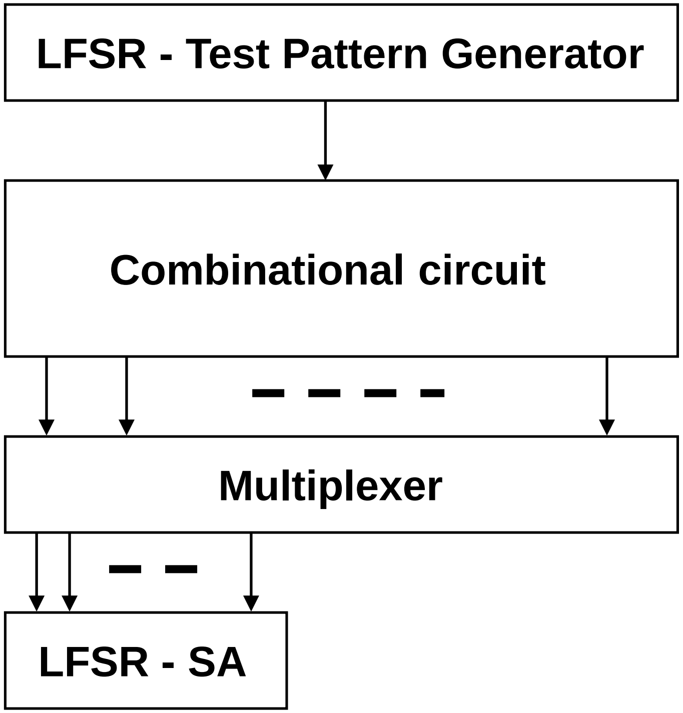
Parallel Signature Analyzer:


Single Input Signature Analyser
UUT
x^4
x^3
x^2
x
1
UUT
?
x^4
x^3
x^2
x
1
Multiple Input Signature Analyser (MISR)

BIST: Signature Analysis

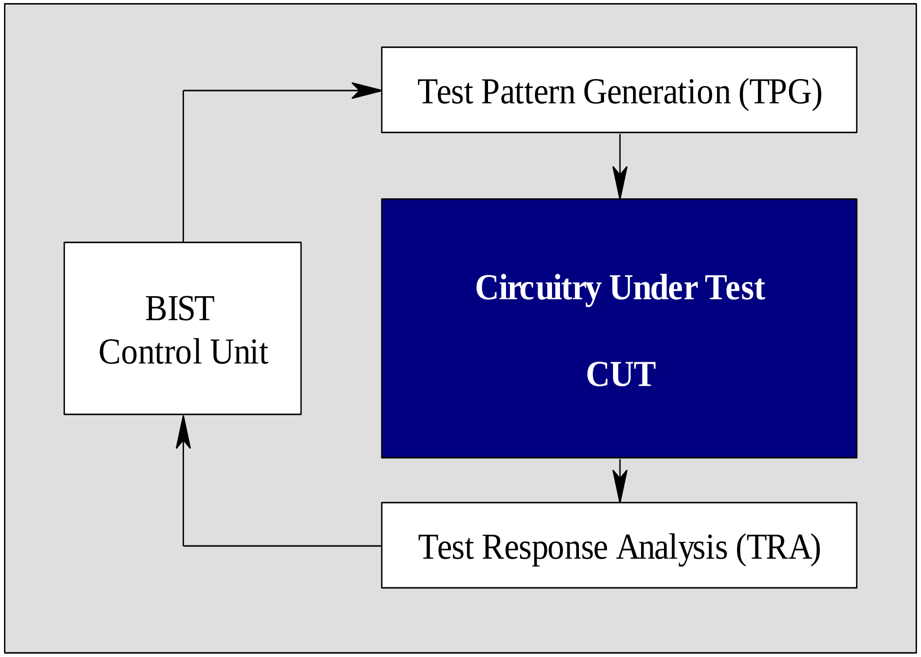
Signature calculating for multiple outputs:





BIST Architectures

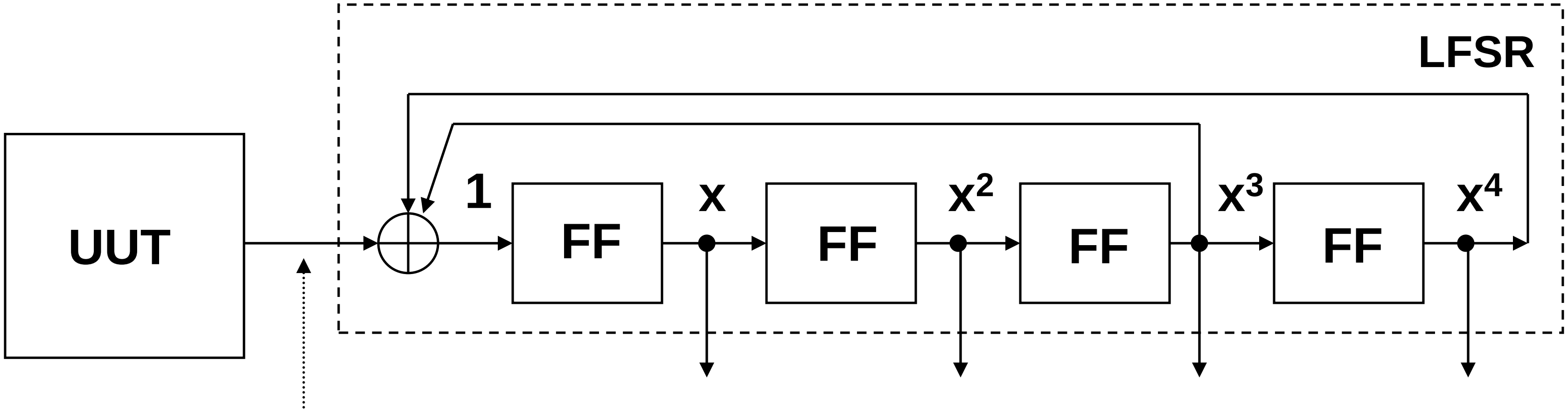
General Architecture of BIST



- BIST components:
  - Test pattern generator (TPG)
  - Test response analyzer (TRA)
  - BIST controller
- A part of a system (***hardcore***) must be operational to execute a self-test
- At minimum the hardcore usually includes ***power***, ***ground***, and ***clock*** circuitry
- Hardcore should be tested by
  - external test equipment or
  - it should be designed self-testable by using various forms of redundancy

BIST: Joining TPG and SA

Two functionalities of LFSR:



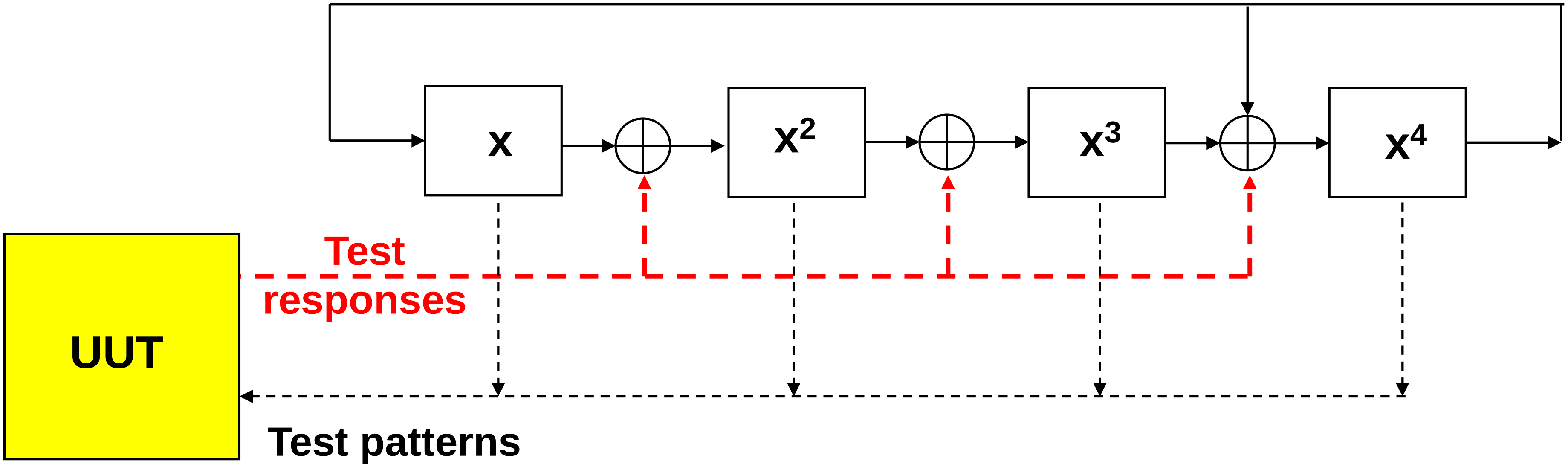
Response string for
Signature Analysis

Test Pattern (when generating tests)
Signature (when analyzing test responses)

Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:

Why modular LFSR is useful for BIST?



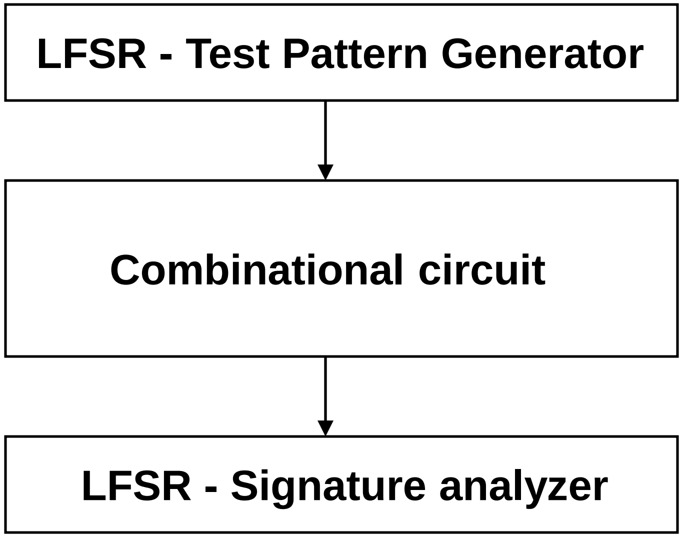
Polynomial: $P(x) = x^4 + x^3 + 1$

Instead of BILBO we have now CSTP architecture

BIST Architectures

Test per Clock:

Disjoint TPG and SA:
BILBO



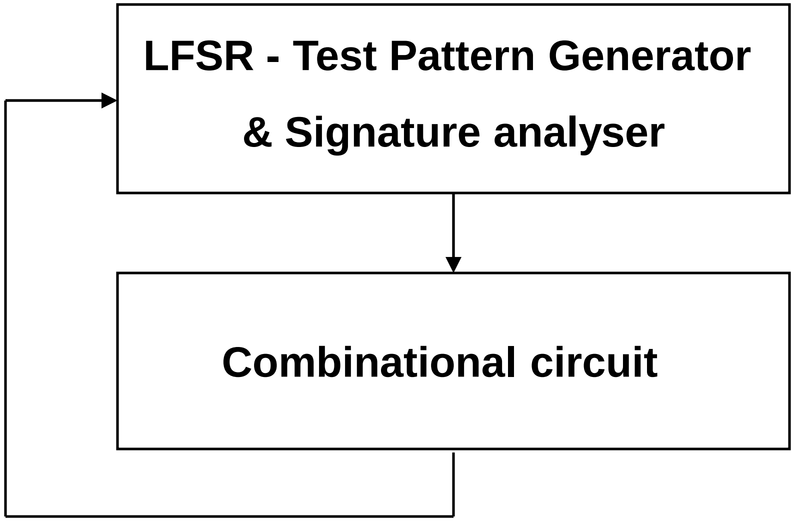
LFSR - Test Pattern Generator

Combinational circuit

LFSR - Signature analyzer

Joint TPG and SA:

CSTP - Circular Self-Test Path:

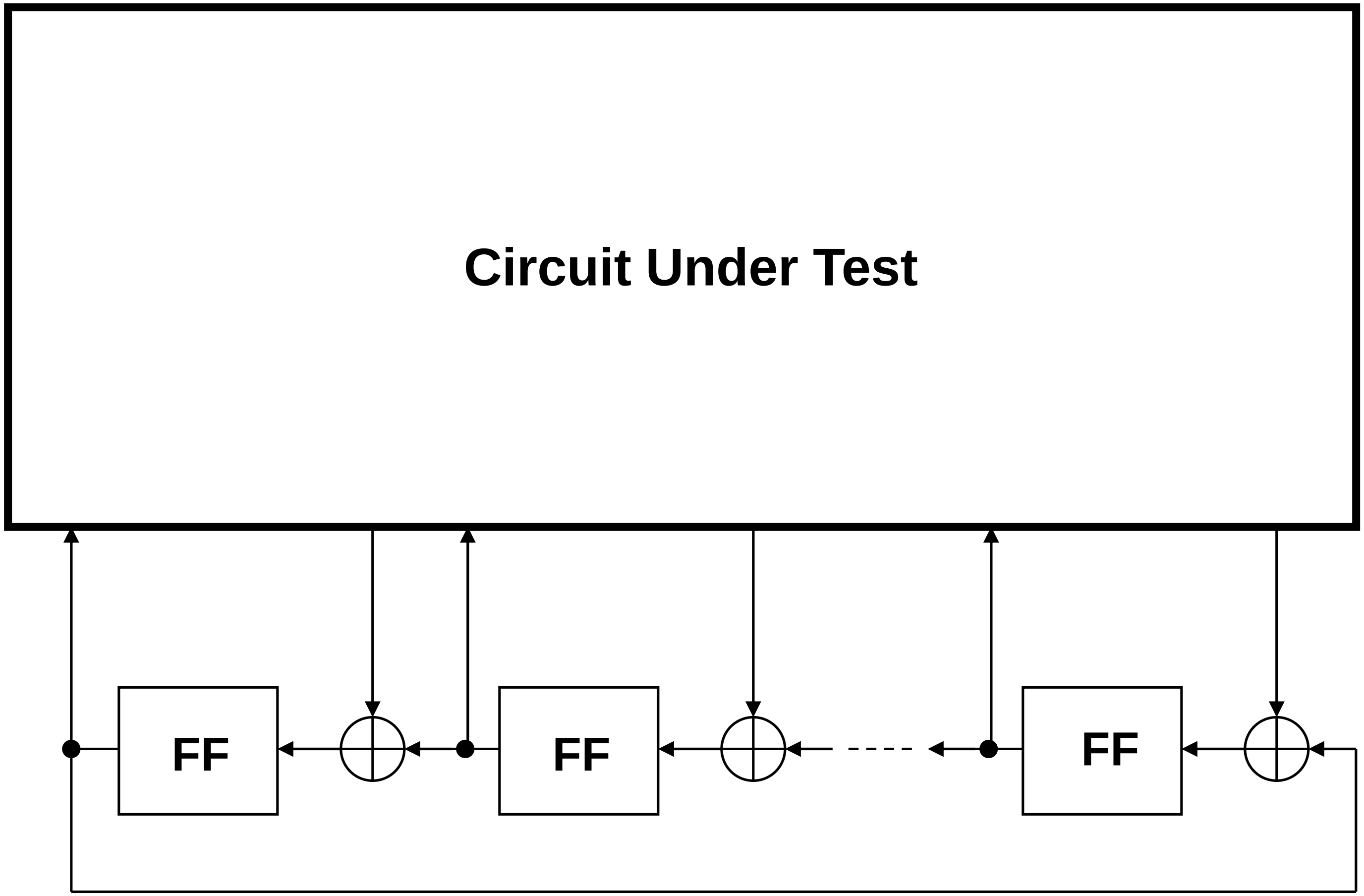


LFSR - Test Pattern Generator

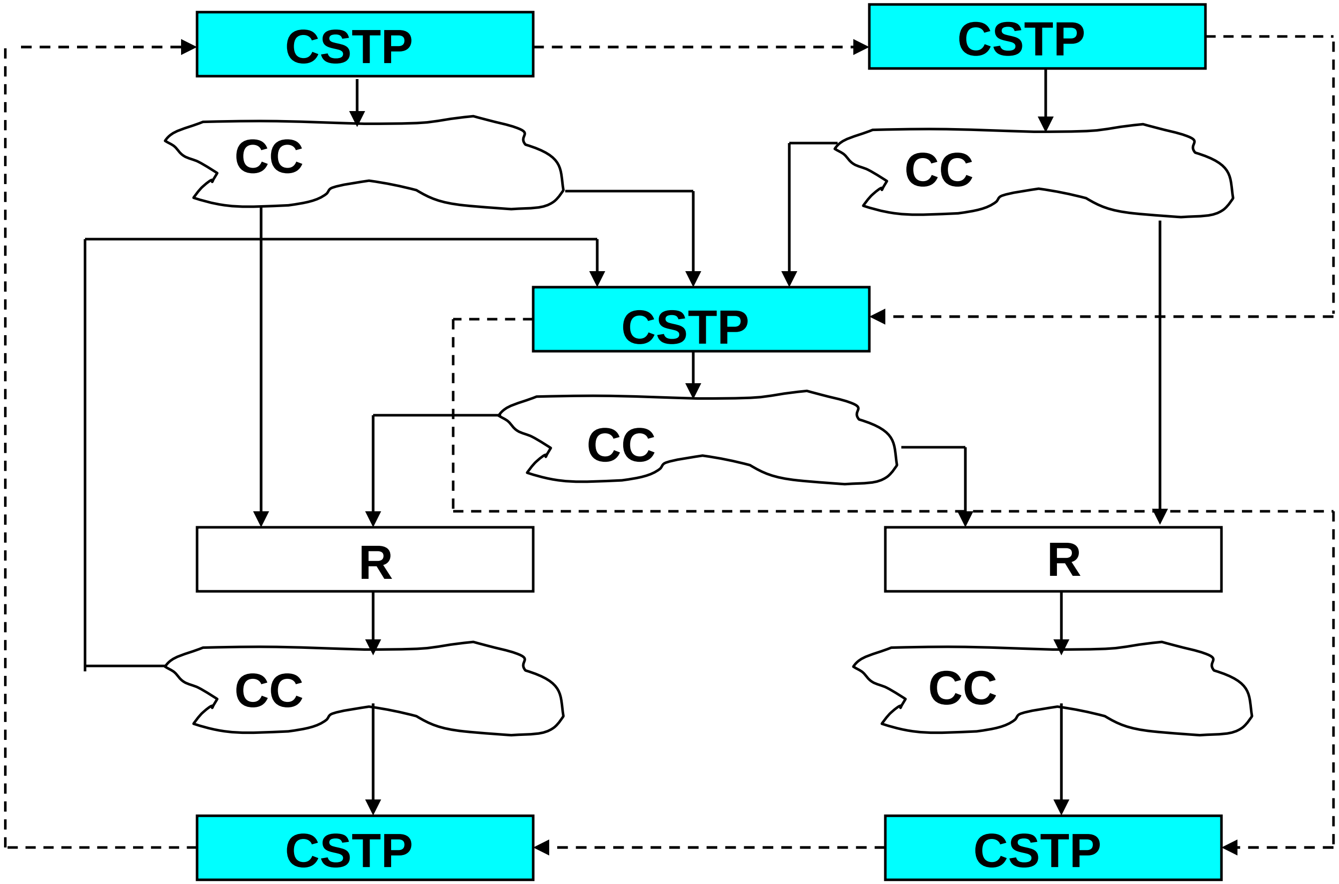
& Signature analyser

Combinational circuit

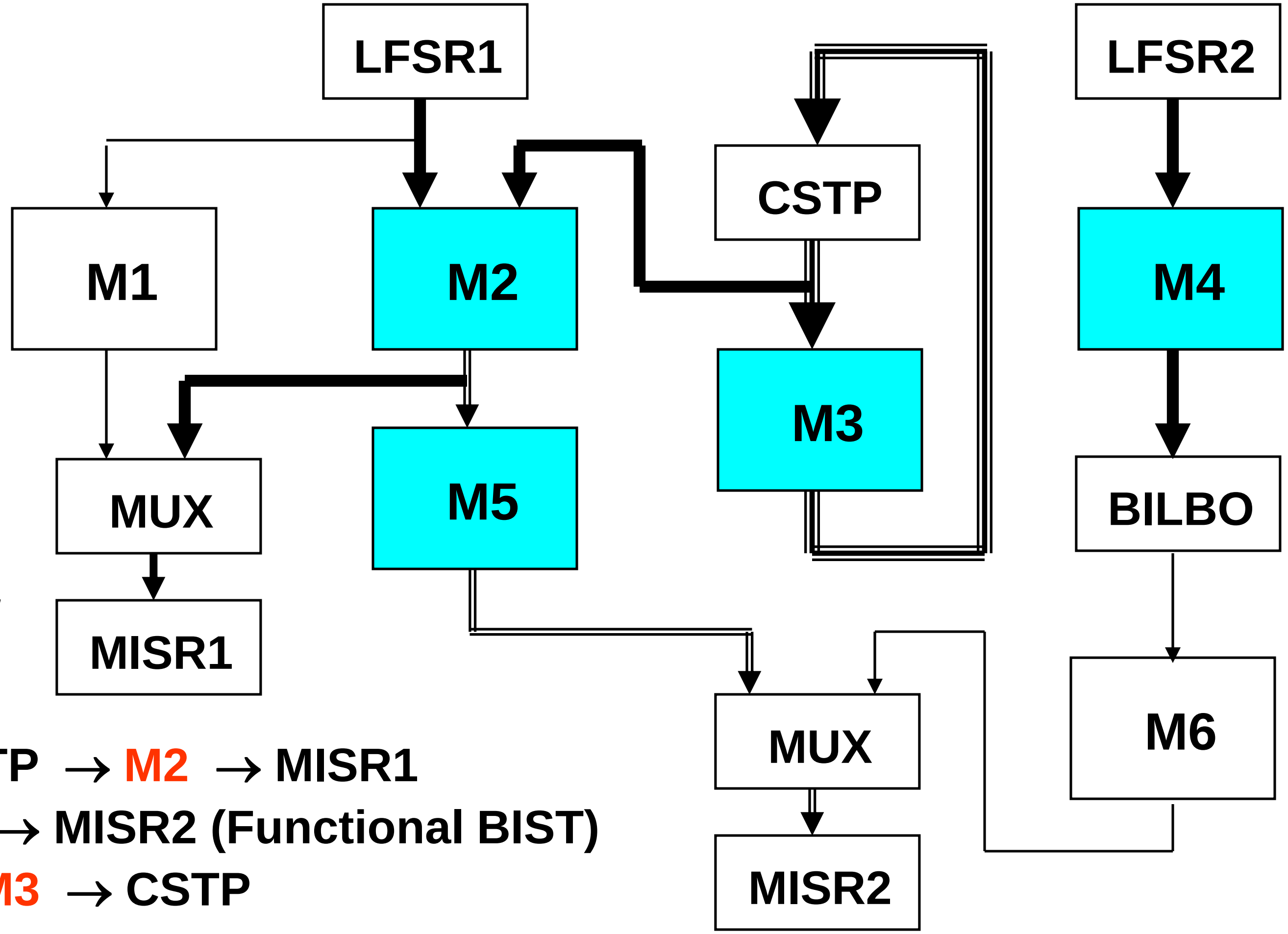
BIST: Circular Self-Test Architecture



BIST: Circular Self-Test Path



BIST Embedding Example



Concurrent testing:

LFSR, CSTP → M2 → MISR1

M2 → M5 → MISR2 (Functional BIST)

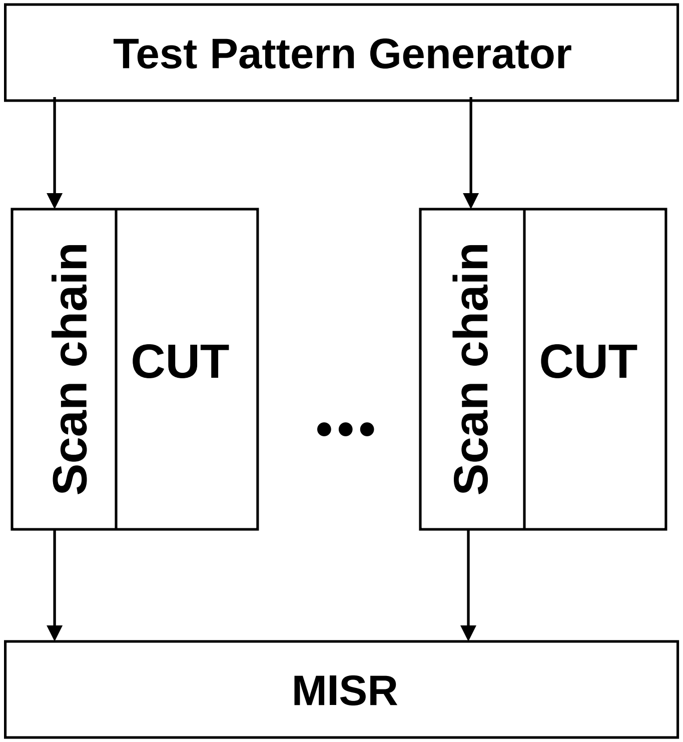
CSTP → M3 → CSTP

LFSR2 → M4 → BILBO

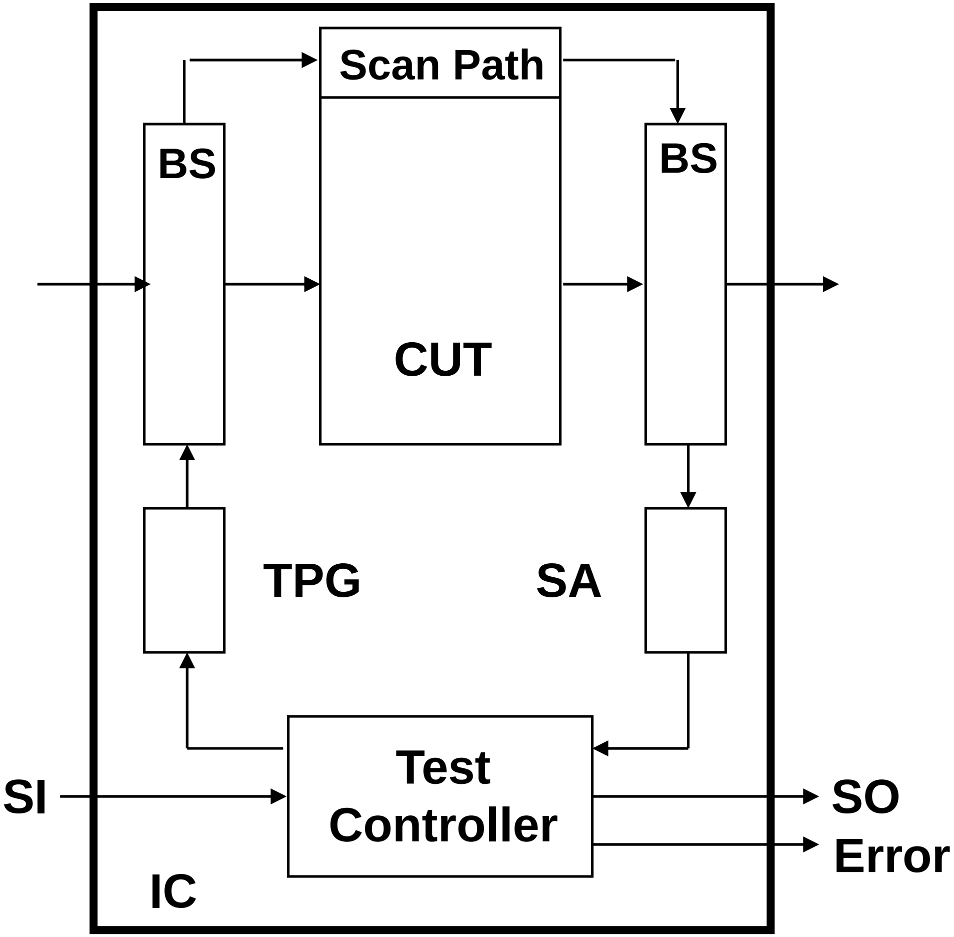
BIST Architectures

STUMPS:

Self-Testing Unit Using MISR and Parallel Shift Register Sequence Generator



LOCST: LSSD On-Chip Self-Test



Scan-Based BIST Architecture

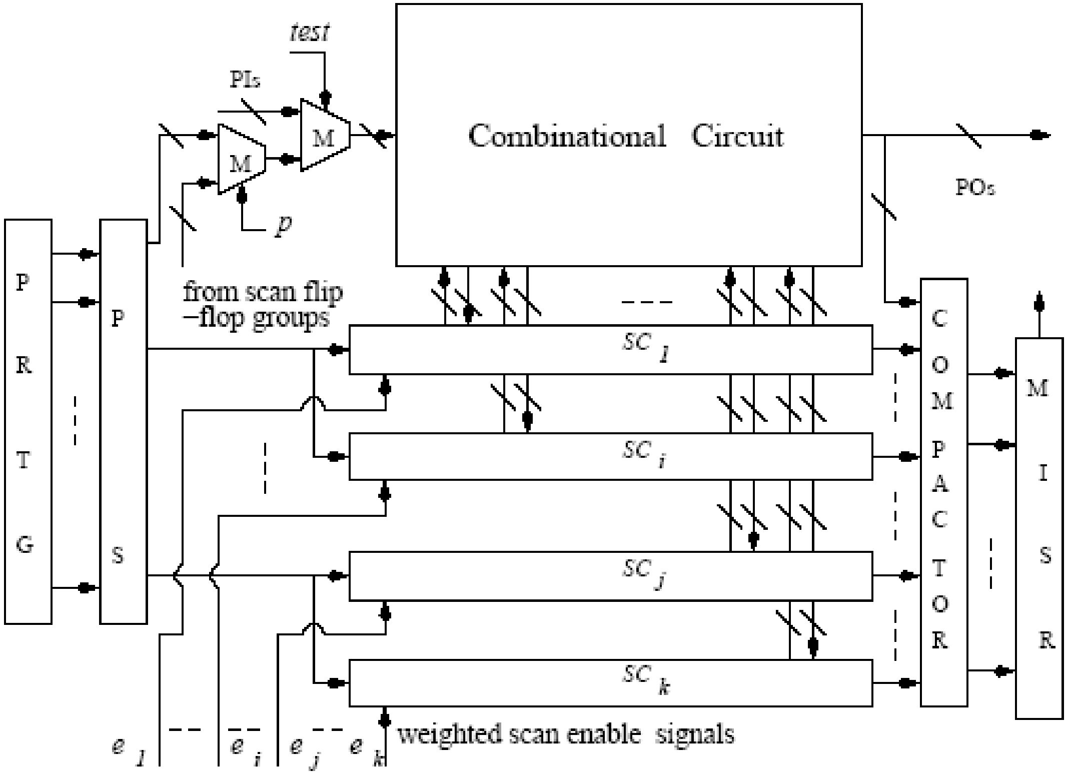


Figure 1: Scan-based BIST for n-detection with weighted scan-enable signals and scan forest.

PS – Phase shifter

Scan-Forest

Scan-Trees

Scan-Segments (SC)

Weighted scan-enables for SS

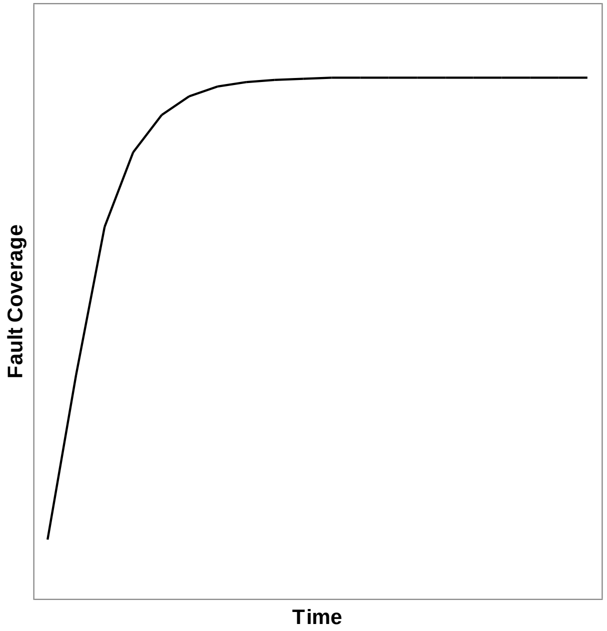
Compactor - EXORs

Copyright: D.Xiang 2003

Problems with BIST

The main motivations of using random patterns are:

- low generation cost
- high initial efeciency

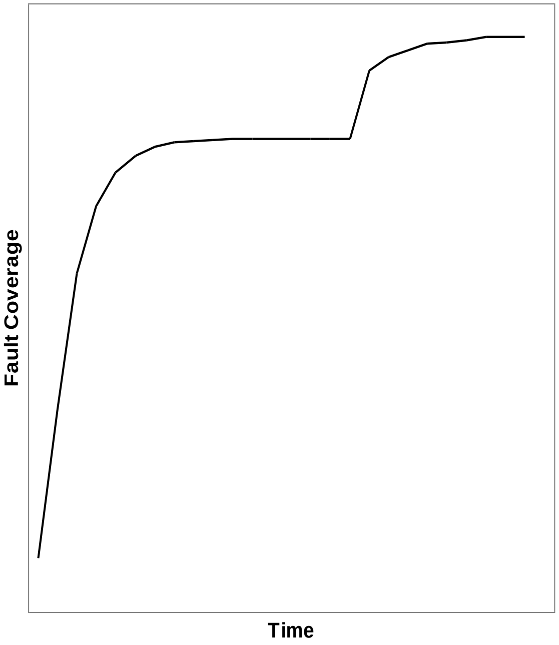


Problems:

- Very long test application time
- Low fault coverage
- Area overhead
- Additional delay

Possible solutions

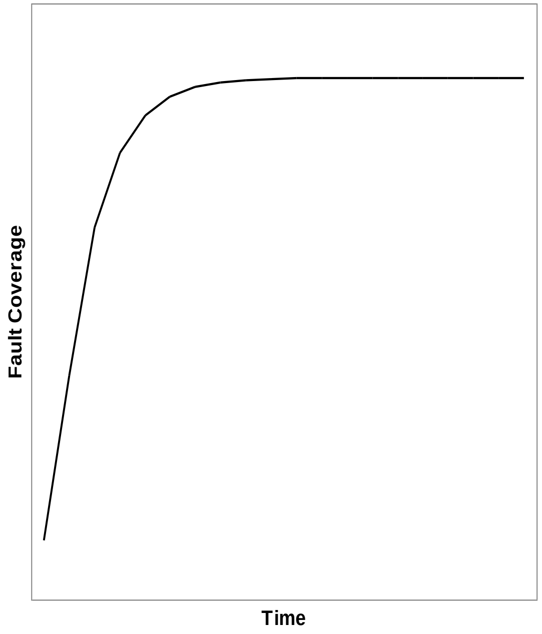
- Weighted pseudorandom test
- Combining pseudorandom test with deterministic data
  - Multiple seed
  - Bit flipping
- Hybrid BIST



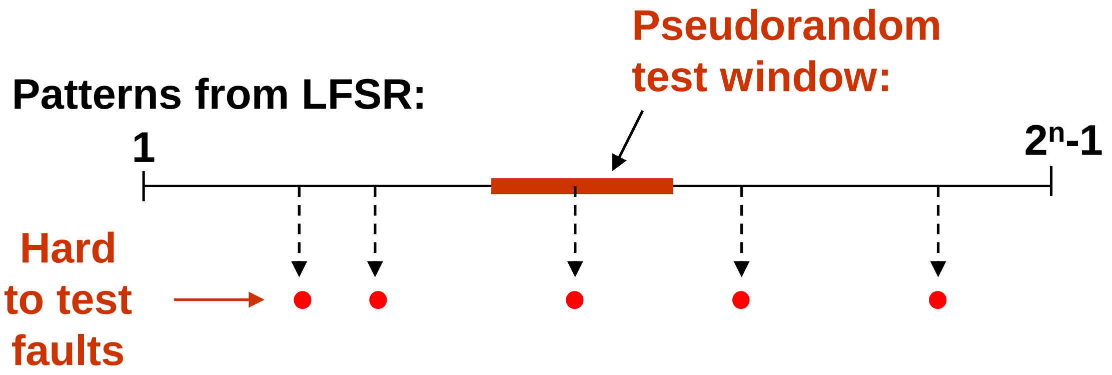
Problems with BIST: Hard to Test Faults

The main motivations of using random patterns are:

- low generation cost
- high initial efeciency



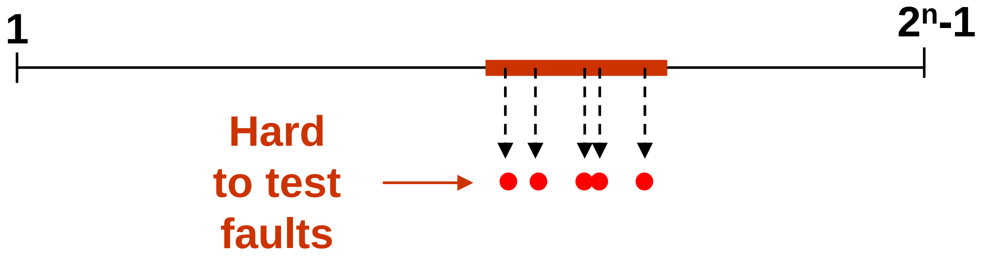
Problem: Low fault coverage


Patterns from LFSR:

Pseudorandom test window:

Hard to test faults

Dream solution: Find LFSR such that:



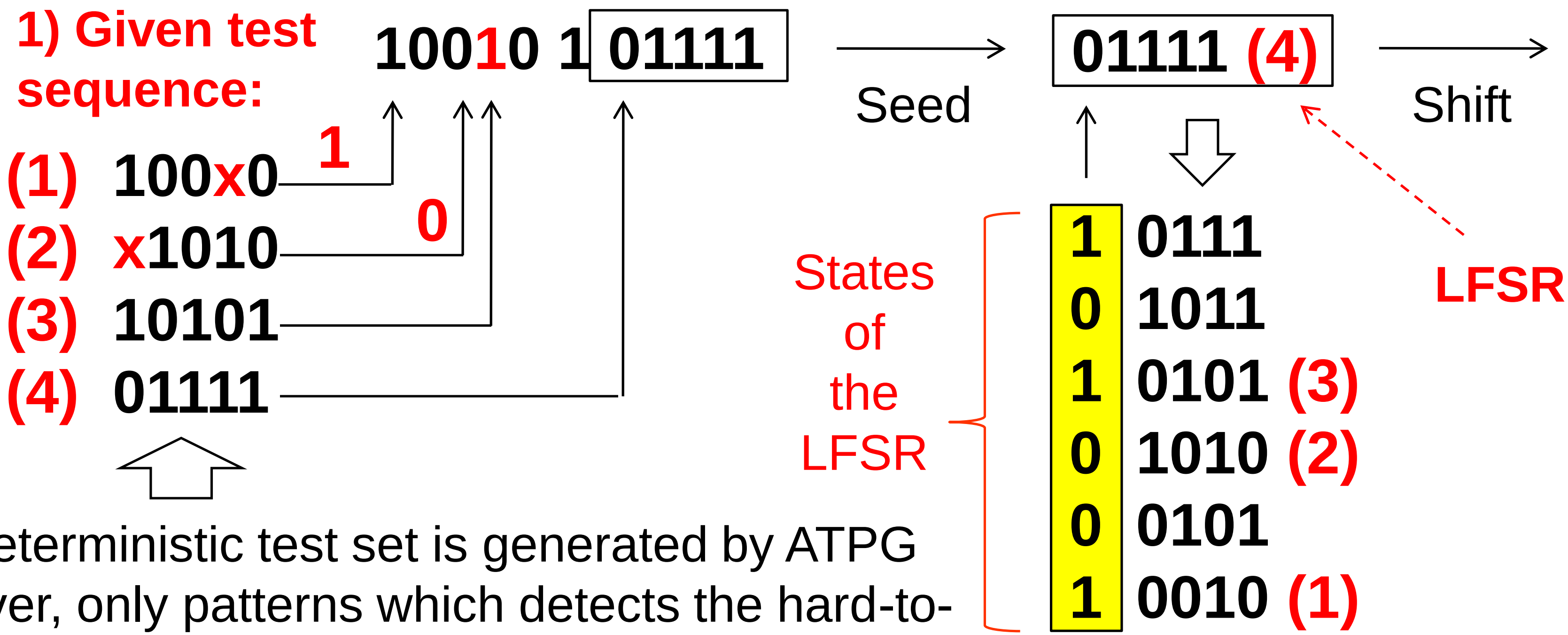
Hard to test faults

Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence

2) Creation of the shortest bit-stream:

3) Expected shortest LFSR sequence:



1) Given test sequence:

(1) 100x0

(2) x1010

(3) 10101

(4) 01111

1

0

10010 1 01111

Seed

01111 (4)

Shift

LFSR

States of the LFSR

1 0111

0 1011

1 0101 (3)

0 1010 (2)

0 0101

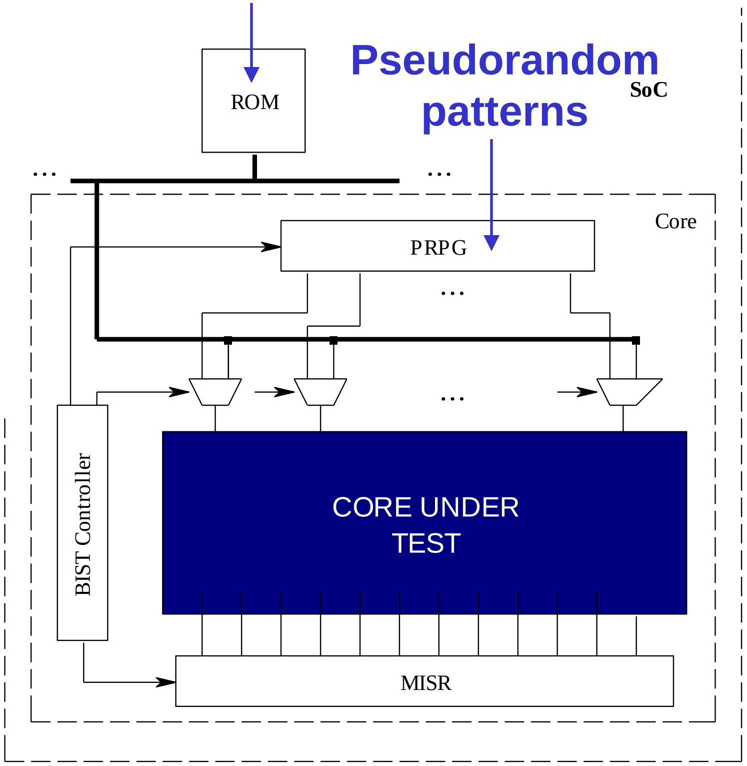
1 0010 (1)

This deterministic test set is generated by ATPG
However, only patterns which detects the hard-to-test faults can be chosen

Hybrid Built-In Self-Test

Deterministic patterns

Pseudorandom patterns

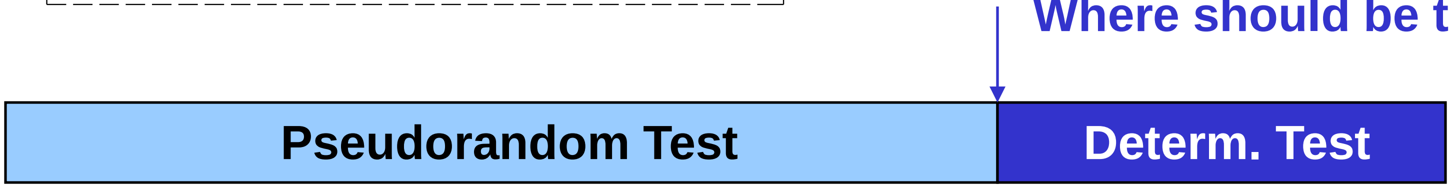


Hybrid test set contains **pseudorandom** and **deterministic** vectors

Pseudorandom test is improved by a stored test set which is specially generated to target the **random resistant faults**

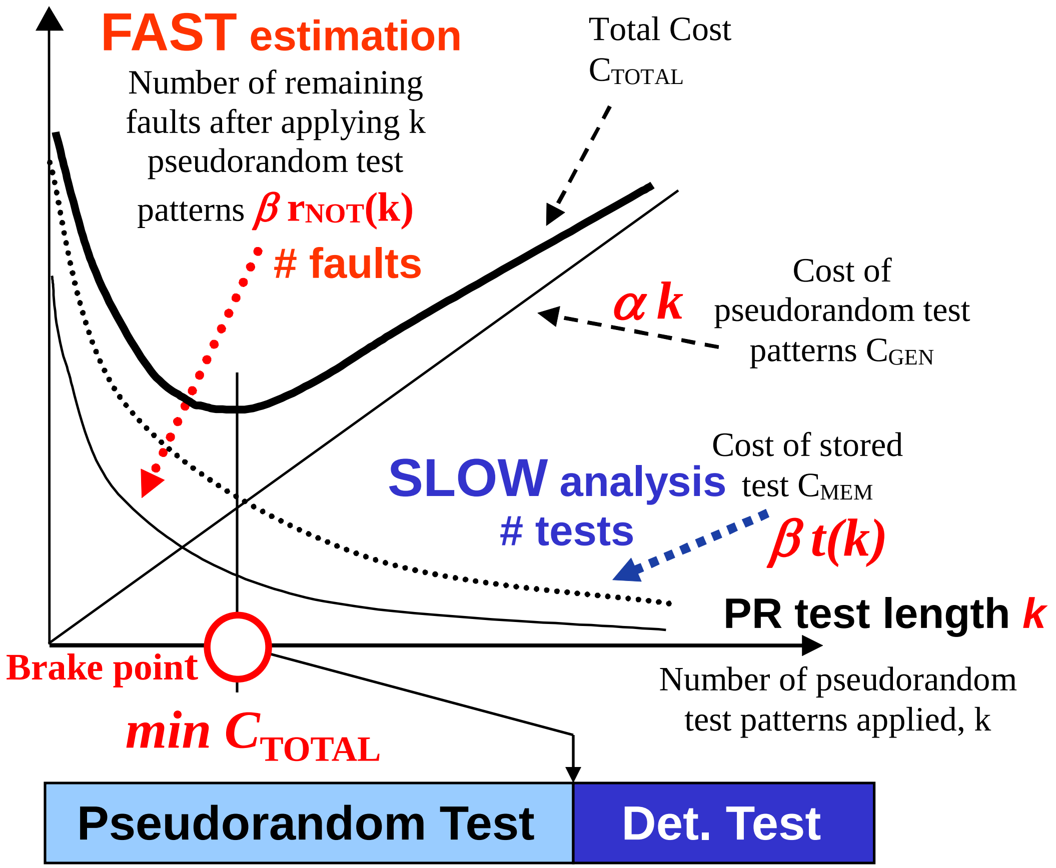
Optimization problem:

Where should be this **breakpoint?**



Optimization of Hybrid BIST

Cost of BIST: $C_{TOTAL} = \alpha k + \beta t(k)$



FAST estimation

Number of remaining faults after applying k pseudorandom test patterns $\beta r_{NOT}(k)$

# faults

Total Cost C_{TOTAL}

αk

Cost of pseudorandom test patterns C_{GEN}

SLOW analysis

# tests

Cost of stored test C_{MEM}

$\beta t(k)$

PR test length *k*

Number of pseudorandom test patterns applied, k

Brake point

min C_{TOTAL}

Pseudorandom Test | **Det. Test**

PR test length

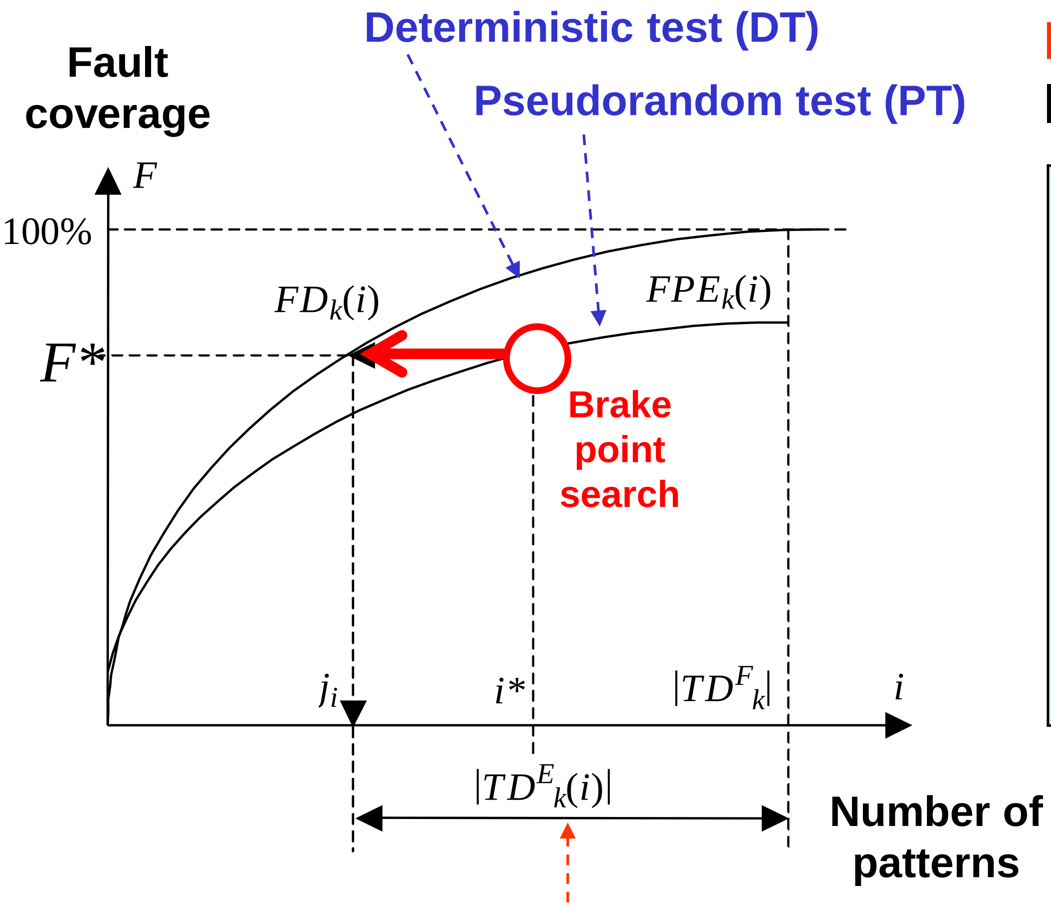
# faults not detected (fast analysis)

# tests needed (slow analysis)

k	$r_{DET}(k)$	$r_{NOT}(k)$	*FC(k)*	*t(k)*
1	155	839	15.6%	104
2	76	763	23.2%	104
3	65	698	29.8%	100
4	90	608	38.8%	101
5	44	564	43.3%	99
10	104	421	57.6%	95
20	44	311	68.7%	87
50	51	218	78.1%	74
100	16	145	85.4%	52
200	18	114	88.5%	41
411	31	70	93.0%	26
954	18	28	97.2%	12
1560	8	16	98.4%	7
2153	11	5	99.5%	3
3449	2	3	99.7%	2
4519	2	1	99.9%	1
4520	1	0	100.0%	0

How to convert #faults to #tests

Deterministic Test Length Estimation



Deterministic test length estimation

Fast estimation for the length of deterministic test:

For each PT length i^* we determine

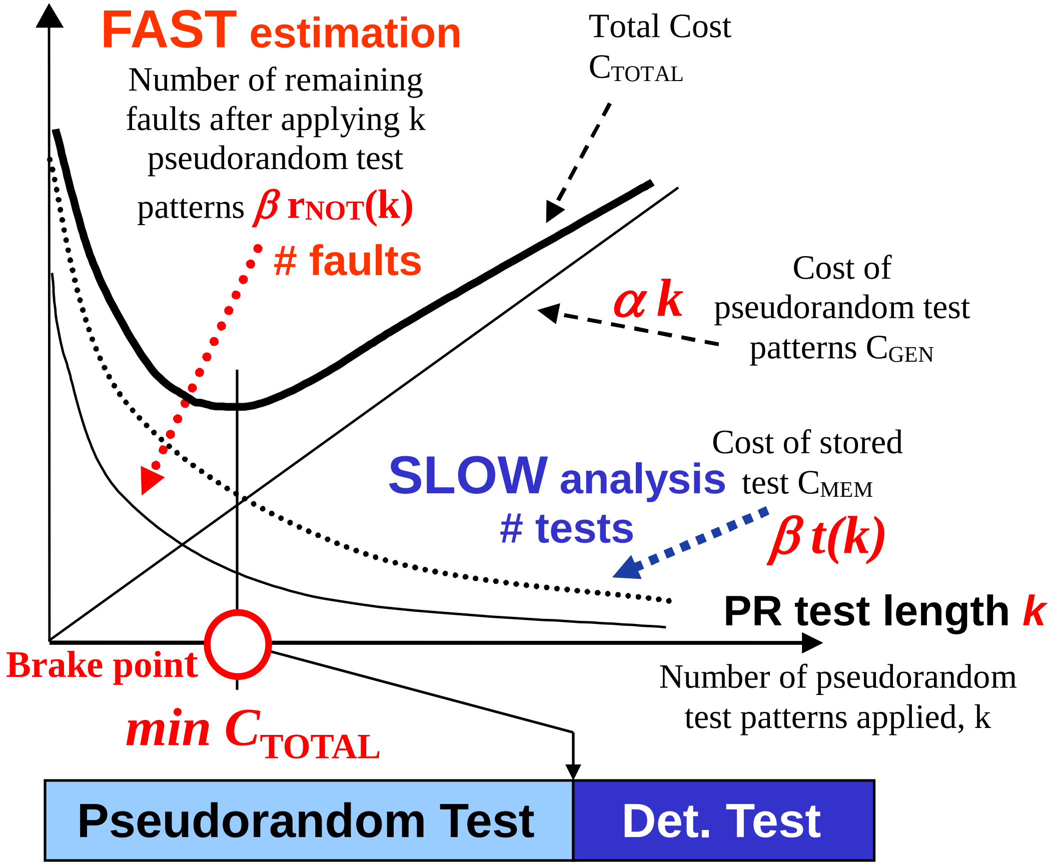
- PT fault coverage F^*, and
- the imaginable part of DT $FD_k(i)$ to be needed for the same fault coverage

Then the remaining part of DT $TD^E_k(i)$ will be the estimation of the DT length

Second idea for estimation: estimating number of patterns

Deterministic Test Length Estimation

Cost of BIST: $C_{TOTAL} = \alpha k + \beta t(k)$



PR test length

# faults not detected (fast analysis)

# tests needed (slow analysis)

k	$r_{DET}(k)$	$r_{NOT}(k)$	*FC(k)*	*t(k)*
1	155	839	15.6%	104
2	76	763	23.2%	104
3	65	698	29.8%	100
4	90	608	38.8%	101
5	44	564	43.3%	99
10	104	421	57.6%	95
20	44	311	68.7%	87
50	51	218	78.1%	74
100	16	145	85.4%	52
200	18	114	88.5%	41
411	31	70	93.0%	26
954	18	28	97.2%	12
1560	8	16	98.4%	7
2153	11	5	99.5%	3
3449	2	3	99.7%	2
4519	2	1	99.9%	1
4520	1	0	100.0%	0

How to convert #faults to #tests

Calculation of the Deterministic Test Cost

Two possibilities to find the length of deterministic data for each possible breakpoint in the pseudorandom test sequence:

ATPG based approach

For each breakpoint of P-sequence, ATPG is used

Fault table based approach

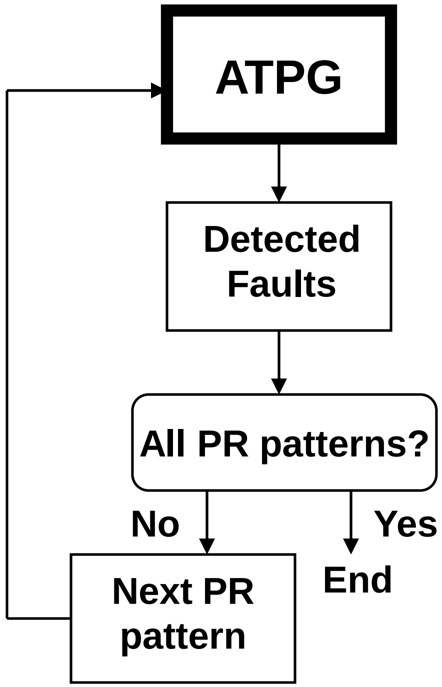
A deterministic test set with fault table is calculated

For each breakpoint of P-sequence, the fault table is updated for not yet detected faults

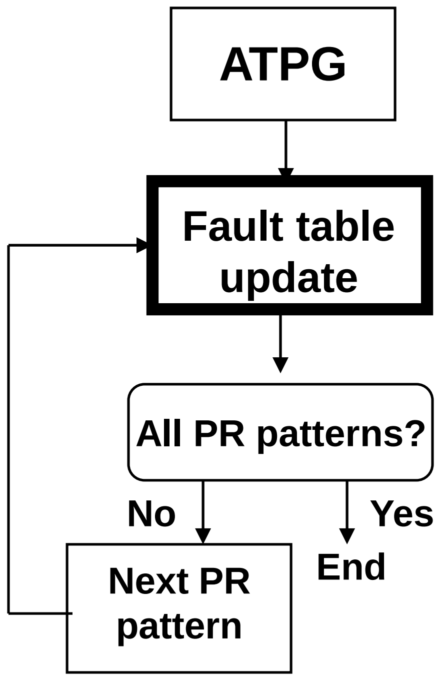
FAST estimation

Only fault coverage is calculated

ATPG based:



Fault table based:

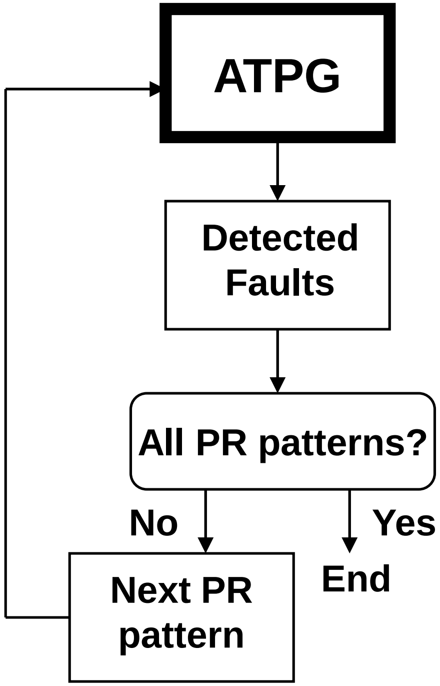


Calculation of the Deterministic Test Cost

ATPG based approach

For each breakpoint of P-sequence, ATPG is used

ATPG based:



ATPG

Detected Faults

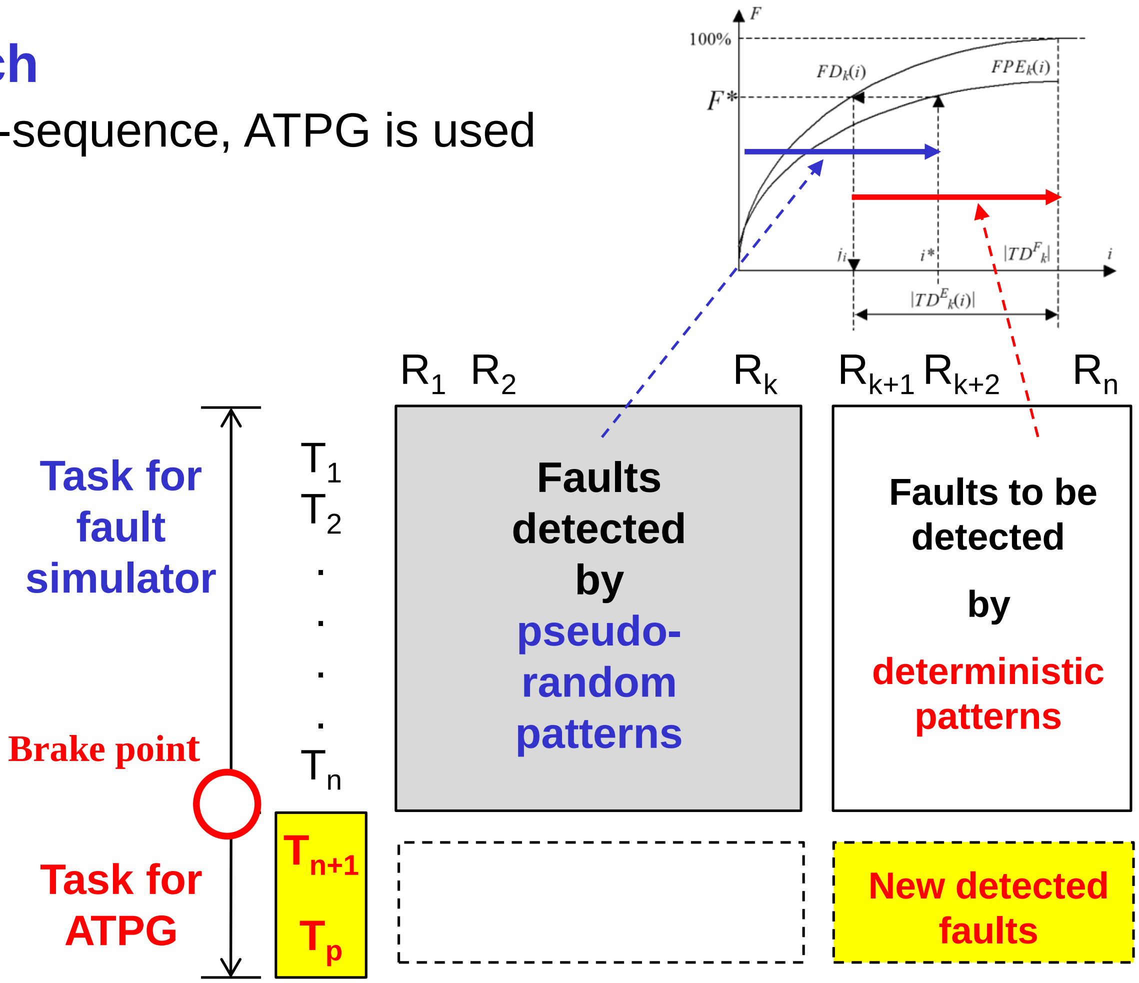
All PR patterns?

No

Yes

Next PR pattern

End



Task for fault simulator

Brake point

Task for ATPG

T_1

T_2

. . . .

T_n

T_{n+1}

T_p

R_1 R_2 R_k R_{k+1} R_{k+2} R_n

Faults detected by pseudo-random patterns

Faults to be detected by deterministic patterns

New detected faults

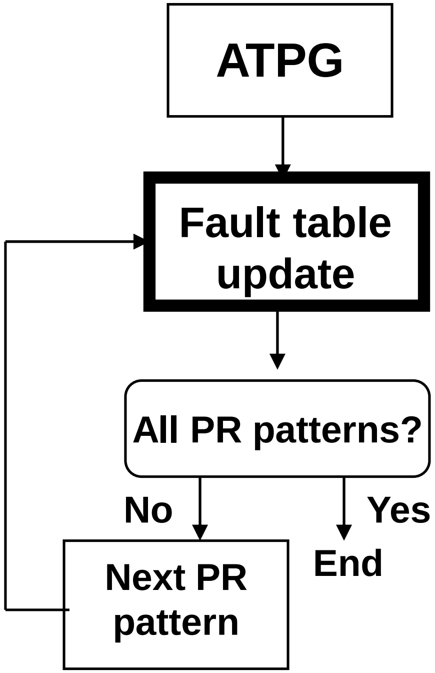
Calculation of the Deterministic Test Cost

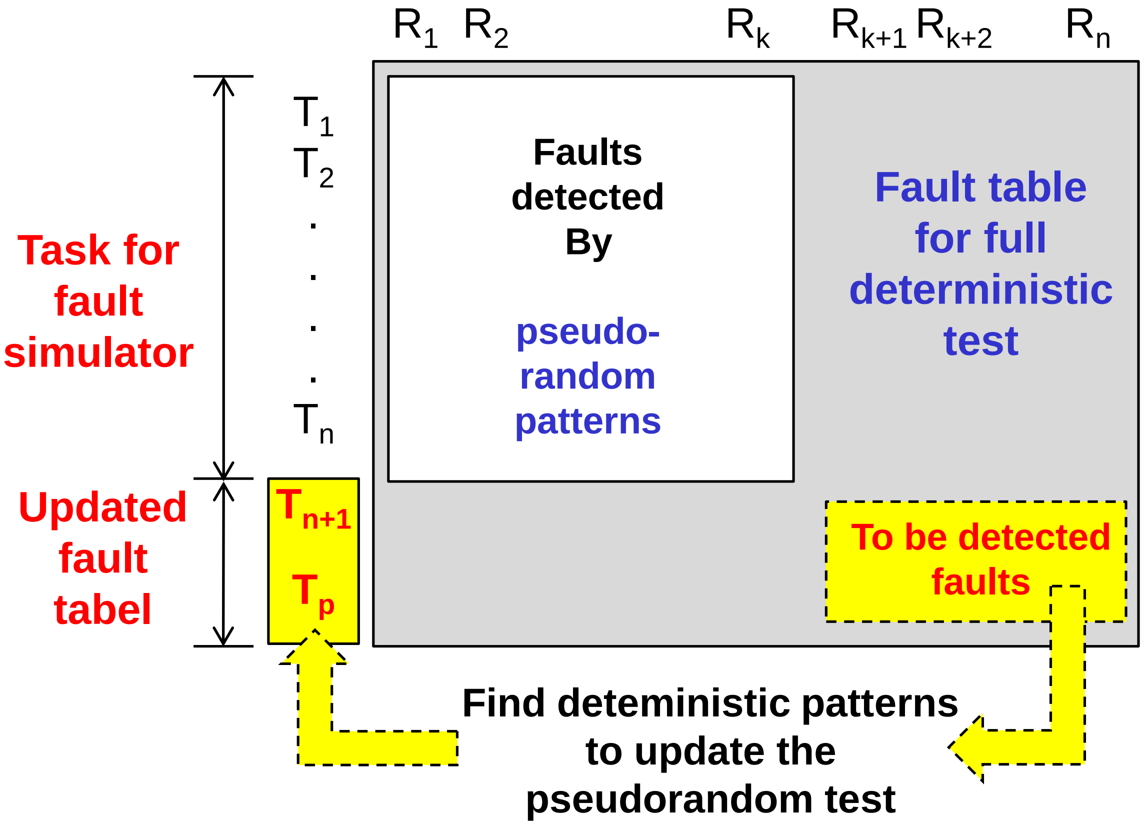
Fault table based approach

A deterministic test set with fault table is calculated

For each breakpoint of P-sequence, the fault table is updated and remaining det. patterns are determined

Fault table based:




R_1 R_2 R_k R_{k+1} R_{k+2} R_n
T_1
T_2
.
.
.
.
T_n
Task for fault simulator
Faults detected By
pseudo-random patterns
Fault table for full deterministic test
Updated fault tabel
T_{n+1}
T_p
To be detected faults
Find deteministic patterns to update the pseudorandom test

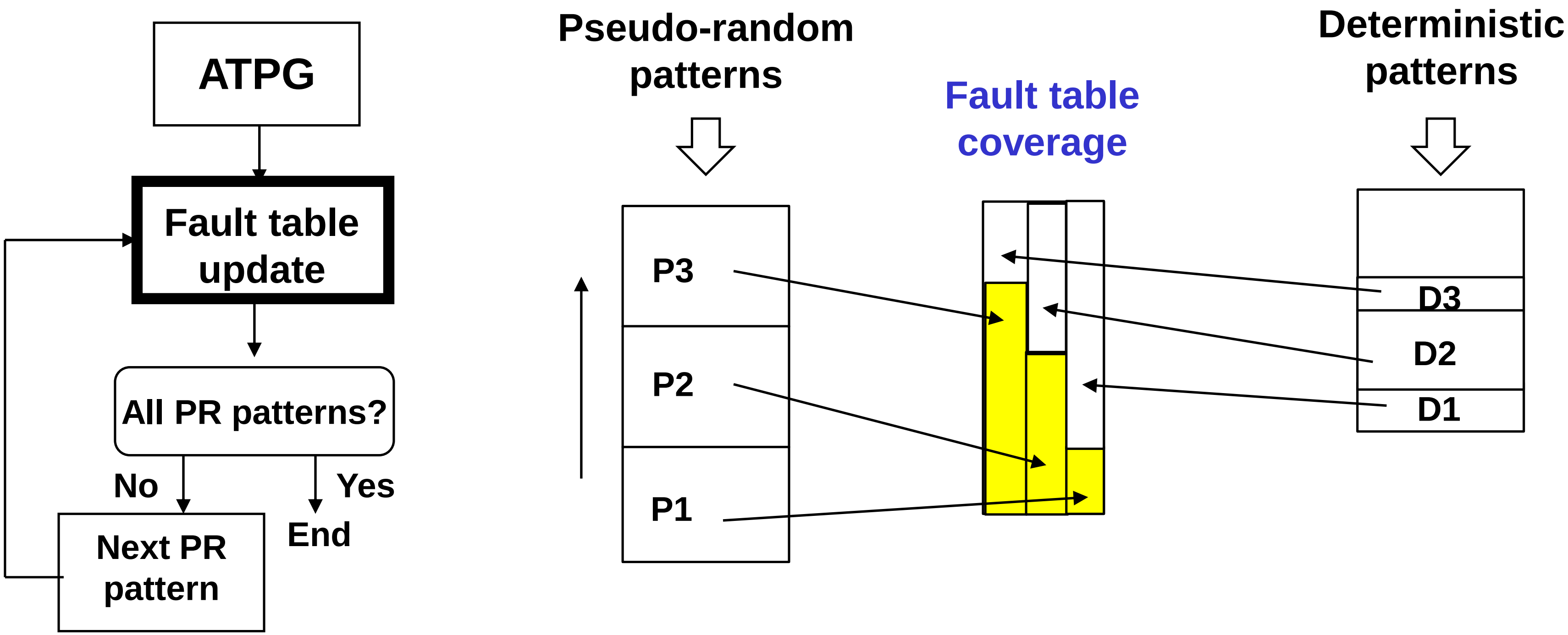
Calculation of the Deterministic Test Cost

Fault table based approach

A deterministic test set with fault table is calculated

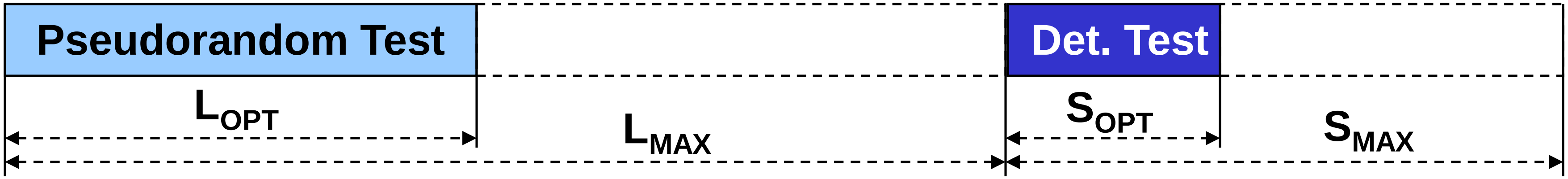
For each breakpoint of P-sequence, the fault table is updated and remaining det. patterns are determined

Fault table based:



Experimental Data: HybBIST Optimization

Finding optimal brakepoint in the pseudorandom sequence:



Optimized hybrid test process:

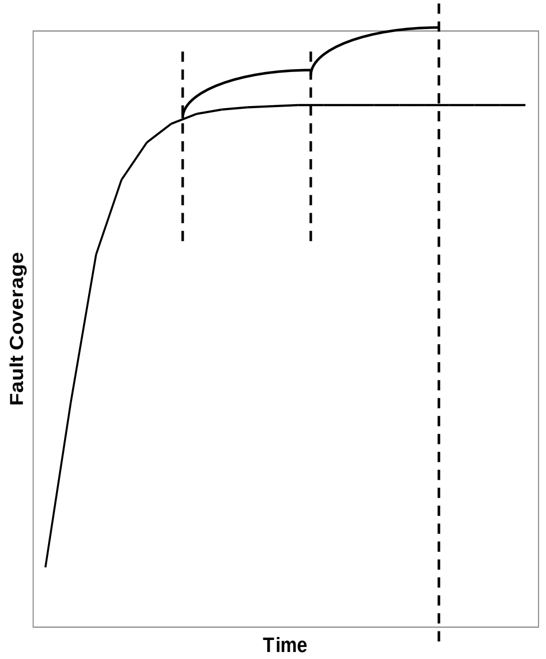
Pseudorandom Test
Det. Test

Circuit	L_{MAX}	L_{OPT}	S_{MAX}	S_{OPT}	B_k	C_{TOTAL}
C432	**780**	**91**	**80**	**21**	**4**	**175**
C499	**2036**	**78**	**132**	**60**	**6**	**438**
C880	**5589**	**121**	**77**	**48**	**8**	**505**
C1355	**1522**	**121**	**126**	**52**	**6**	**433**
C1908	**5803**	**105**	**143**	**123**	**5**	**720**
C2670	**6581**	**444**	**155**	**77**	**30**	**2754**
C3540	**8734**	**297**	**211**	**110**	**7**	**1067**
C5315	**2318**	**711**	**171**	**12**	**23**	**987**
C6288	**210**	**20**	**45**	**20**	**4**	**100**
C7552	**18704**	**583**	**267**	**61**	**51**	**3694**

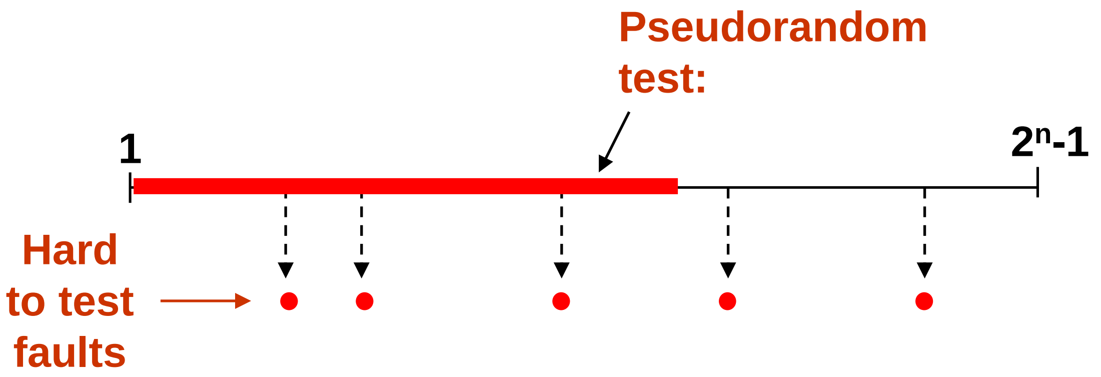
Hybrid BIST with Reseeding

The motivation of using random patterns is:

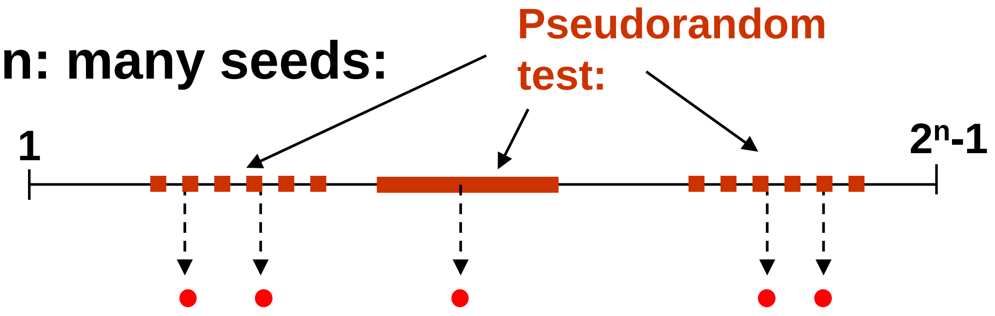
- low generation cost
- high initial efeciency



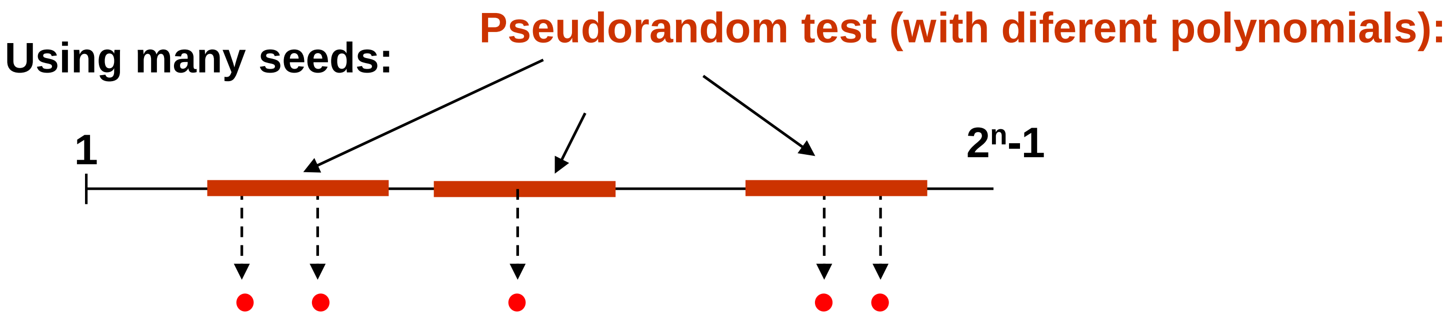
Problem: low fault coverage → long PR test



Solution: many seeds:

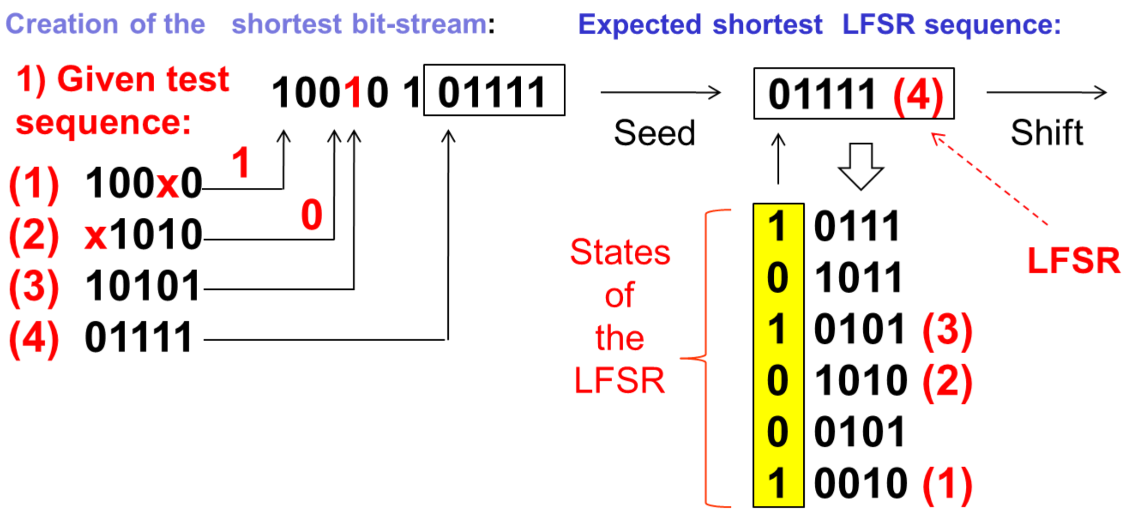


Hybrid BIST with Reseeding



Using many seeds:

Pseudorandom test (with diferent polynomials):



Creation of the shortest bit-stream:

Expected shortest LFSR sequence:

1) Given test sequence:

(1) 100x0
(2) x1010
(3) 10101
(4) 01111

10010 1 01111

01111 (4)

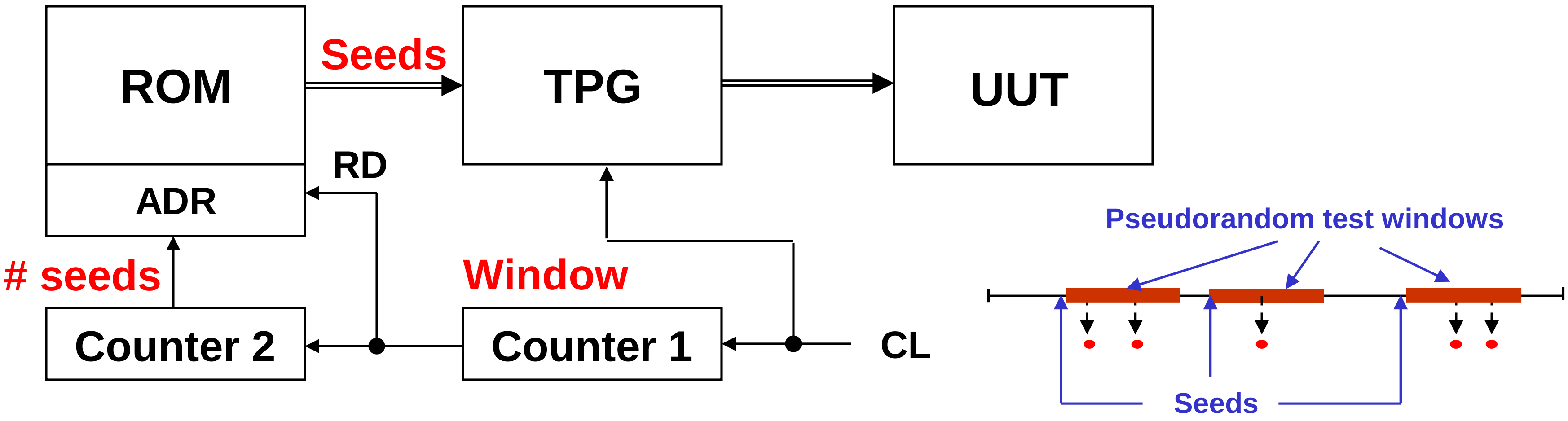
States of the LFSR

1 0111
0 1011
1 0101 (3)
0 1010 (2)
0 0101
1 0010 (1)

Problems:

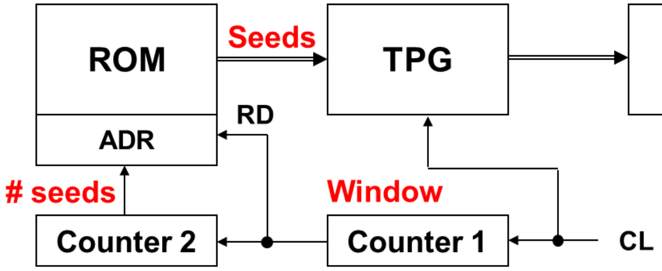
Which polynomials and seeds should be used for the blocks?

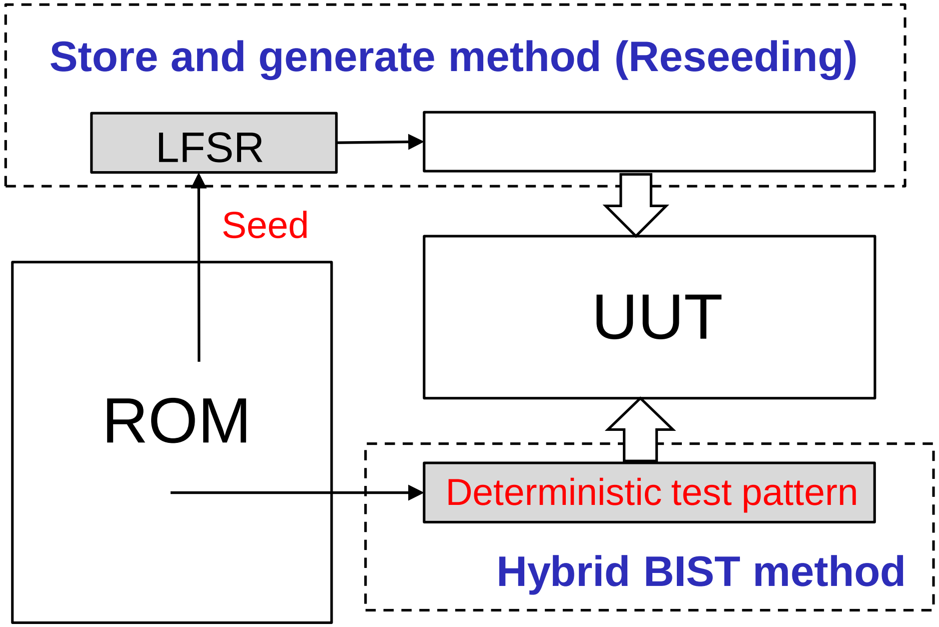
Store-and-Generate Test Architecture

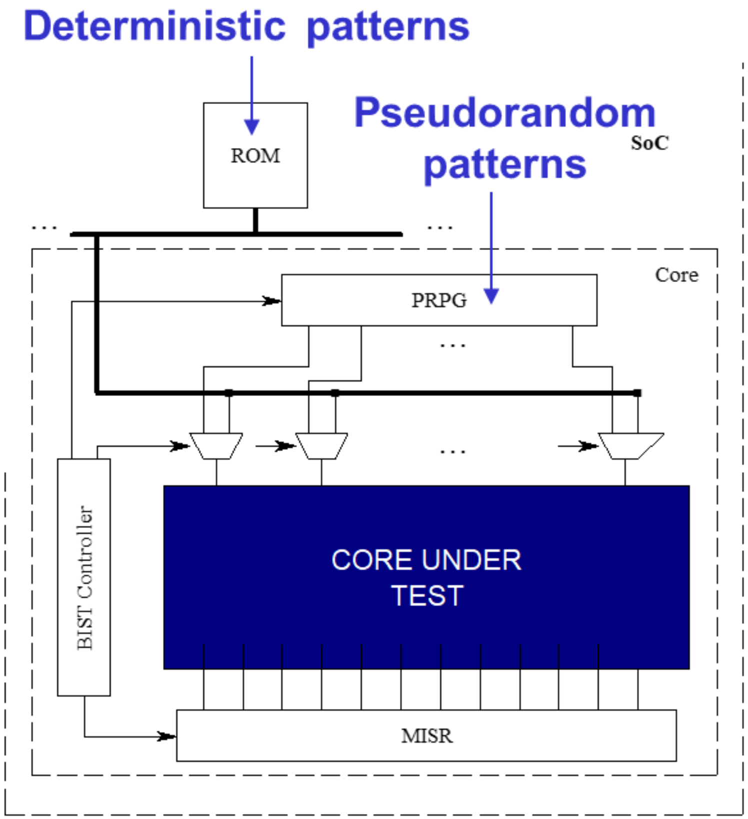


- **ROM** contains deterministic data for BIST control to target **hard-to-test-faults**
- Each pattern P_k in ROM serves as an initial state of the LFSR for test pattern generation (TPG) - **seeds**
- Counter 1 counts the number of pseudorandom patterns generated starting from P_k - **width of the windows**
- After finishing the cycle for Counter 2 is incremented for reading the next pattern P_{k+1} – for **starting the new window**

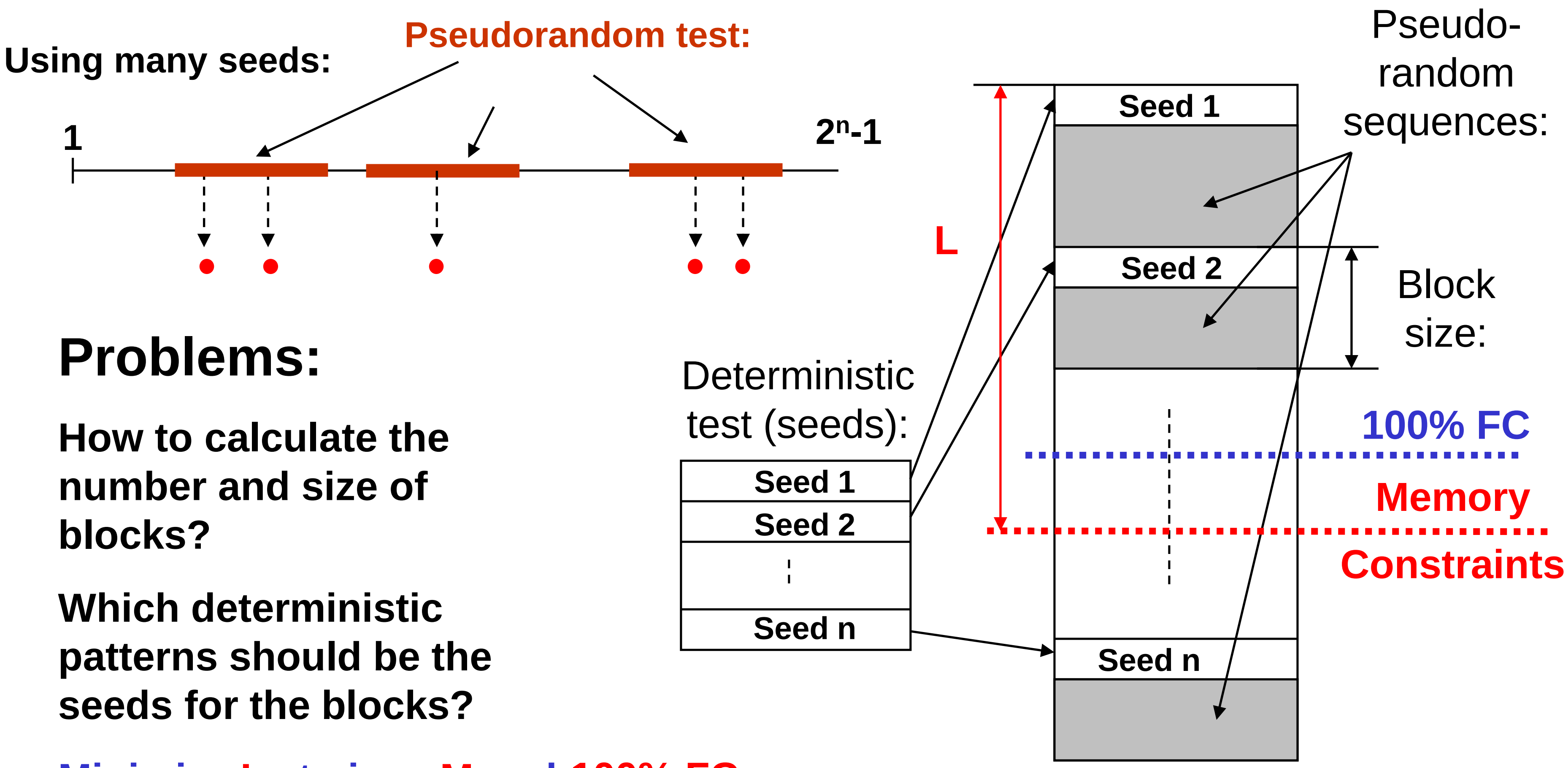
Store-and-Generate vs. Hybrid BIST




Store and generate method (Reseeding)
LFSR
Seed
UUT
ROM
Deterministic test pattern
Hybrid BIST method



HBIST Optimization Problem



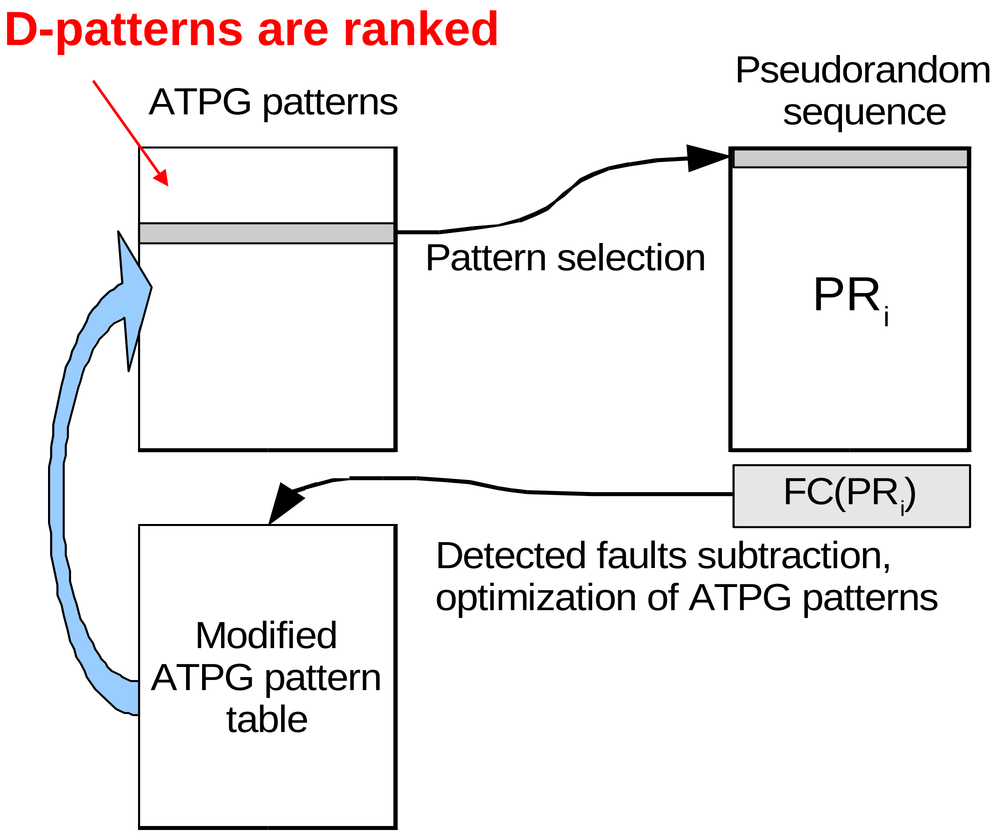
Problems:

How to calculate the number and size of blocks?

Which deterministic patterns should be the seeds for the blocks?

Minimize L at given M and 100% FC

Hybrid BIST Optimization Algorithm 1



Algorithm is based on D-patterns ranking

Deterministic test patterns with 100% quality are generated by ATPG

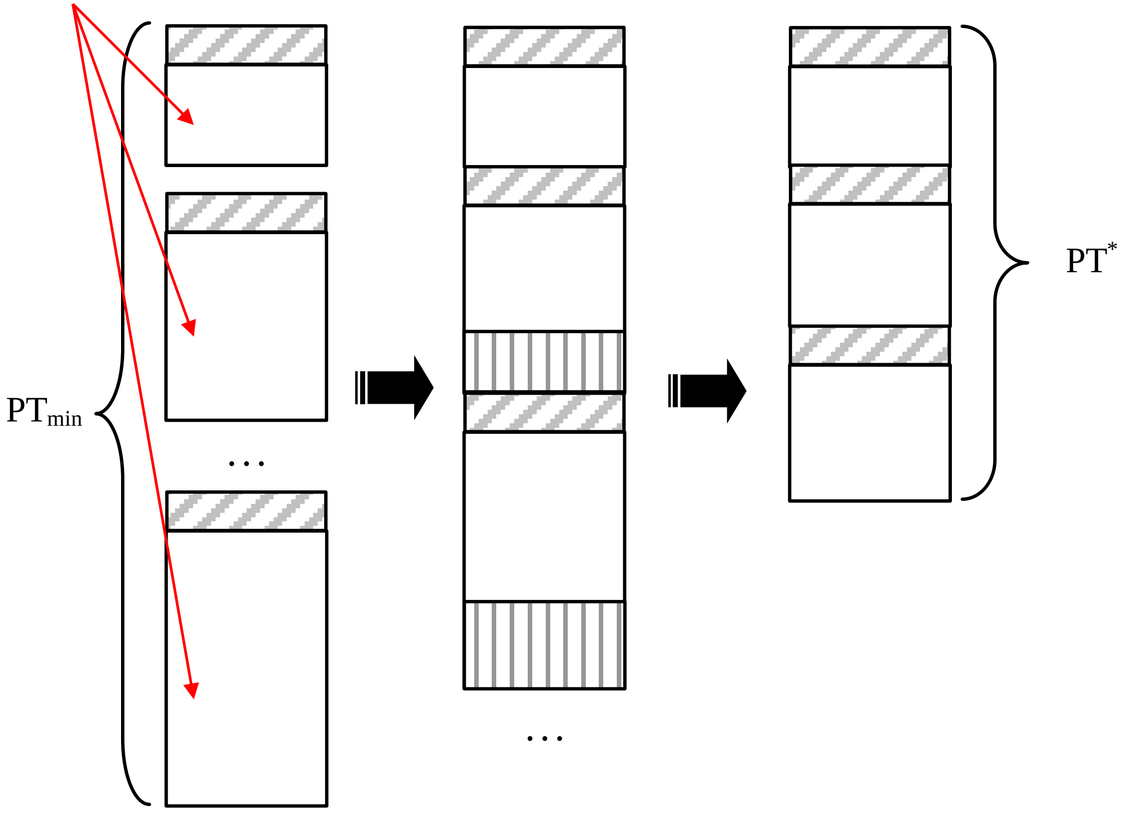
The best pattern is selected as a seed

A pseudorandom block is produced and the fault table of ATPG patterns is updated

The procedure ends when 100% fault coverage is achieved

Hybrid BIST Optimization Algorithm 2

P-blocks are ranked



▨ Deterministic test vector (seed) DT_i
□ Pseudorandom test sequence PR_i
▥ Pseudorandom sequence removed with the block length optimization

Algorithm is based on P-blocks ranking

Deterministic test patterns with 100% quality are generated by ATPG

All P-blocks are generated for all D-patterns and ranked

The best P-block is selected included into sequence and updated

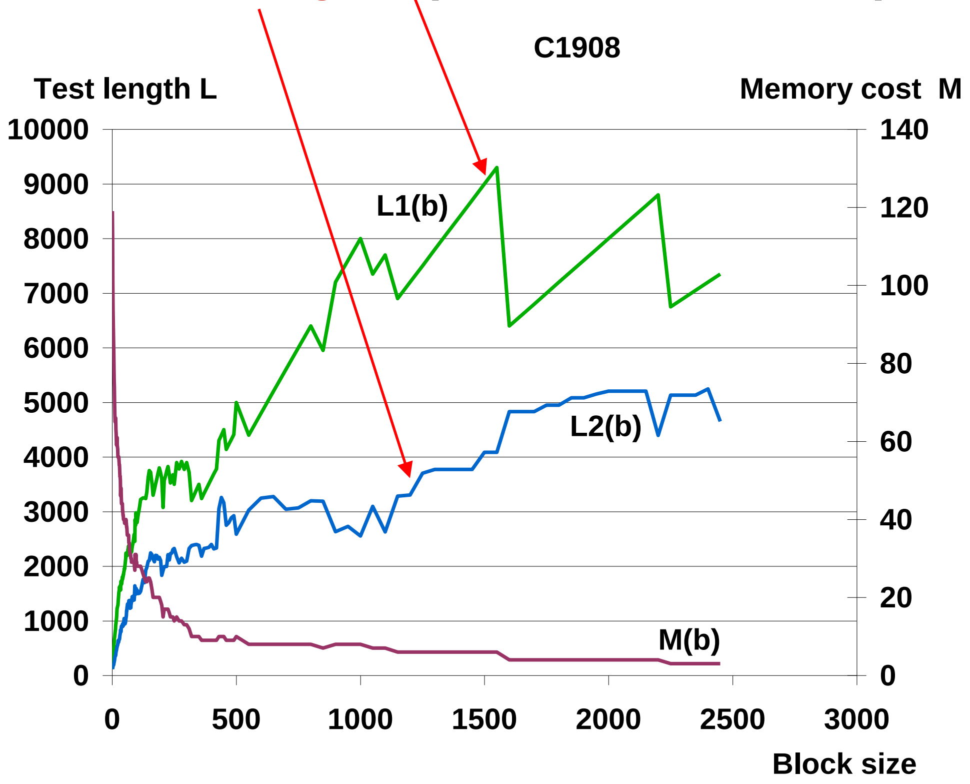
The procedure ends when 100% fault coverage is achieved

Cost Curves for Hybrid BIST with Reseeding

Two possibilities for reseeding:

Constant block length (less HW overhead)

Dynamic block length (more HW overhead)



Functional Self-Test

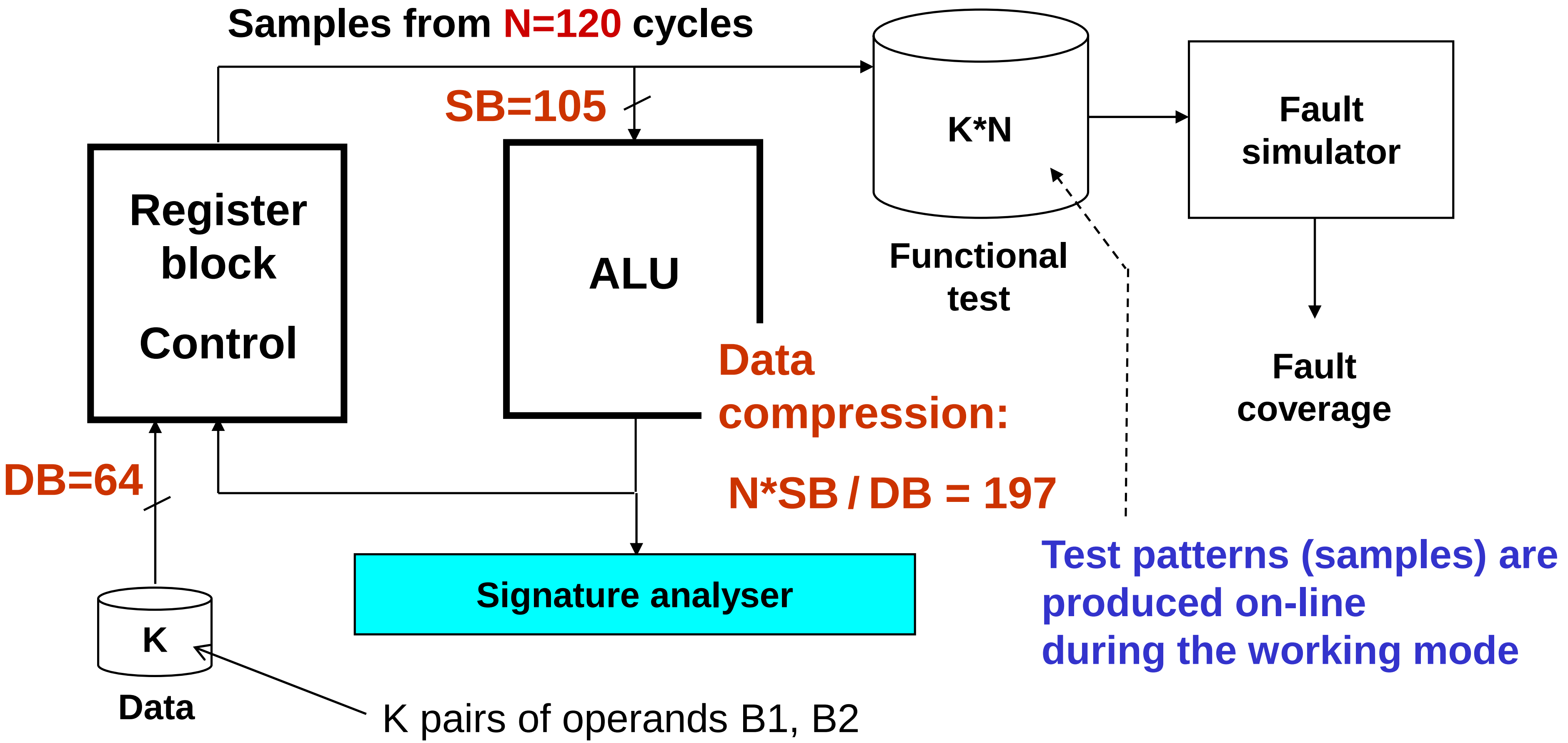
- Traditional BIST solutions use special hardware for pattern generation on chip, this may introduce area overhead and performance degradation
- New methods have been proposed which exploit specific functional units like arithmetic blocks or processor cores for on-chip test generation
- It has been shown that adders can be used as test generators for pseudorandom and deterministic patterns
- Today, there is no general method how to use arbitrary functional units for built-in test generation

Hybrid Functional BIST

- To improve the quality of FBIST we introduce the method of Hybrid FBIST
- The idea of Hybrid FBIST consists in using the mixture of
  - functional patterns produced by the microprogram (no additional HW is needed), and
  - additional stored deterministic test patterns to improve the total fault coverage (HW overhead: MUX-es, Memory)
- Tradeoffs should be found between
  - the testing time and
  - the HW/SW overhead cost

Example: Functional BIST for Divider

Functional BIST quality analysis for



Data compression:

N*SB / DB = 197

Test patterns (samples) are produced on-line during the working mode

K pairs of operands B1, B2

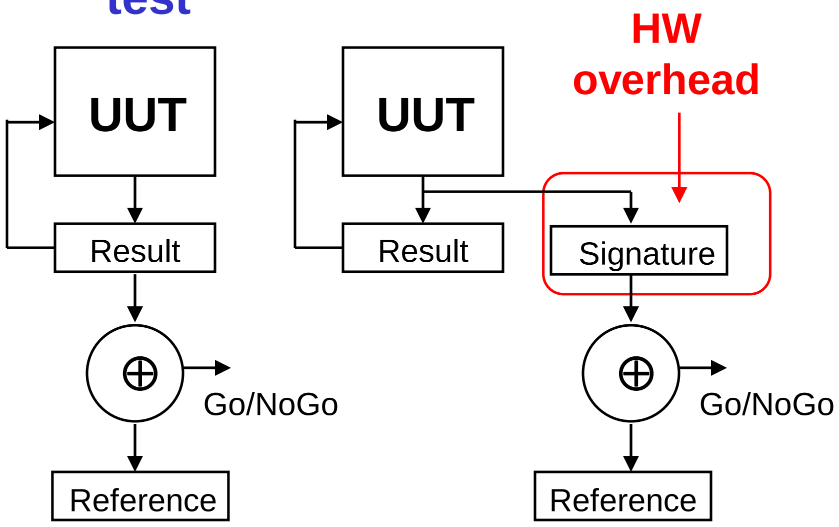
Example: Functional BIST Quality for Divider

Fault coverage of FBIST compared to Functional test

Data	Functional testing			Functional BIST		
	B1	B2	Total	B1	B2	Total
4/2	13.21	15.09	14.15	35.14	40.57	29.72
7/2	21.23	16.98	19.10	38.44	47.64	29.25
6/3	19.34	31.6	25.47	41.04	39.62	42.45
8/2	25.47	10.38	17.92	32.07	40.57	25.00
9/4	8.96	5.66	7.31	36.56	47.64	25.47
9/3	**32.55**	26.89	29.72	43.63	46.07	40.57
12/6	13.44	8.02	18.87	36.08	39.62	32.55
14/2	18.16	25.00	11.32	37.50	49.06	25.94
15/3	29.48	**31.13**	**27.83**	**47.88**	**50.00**	**45.75**
2/4	7.8	7.55	8.02	29.01	20.75	33.02
Aver.	18.96	17.83	17.97	37.74	42.15	32.97
Gain	1.0	1.0	1.0	2.0	2.4	1.8

Traditional Functional test

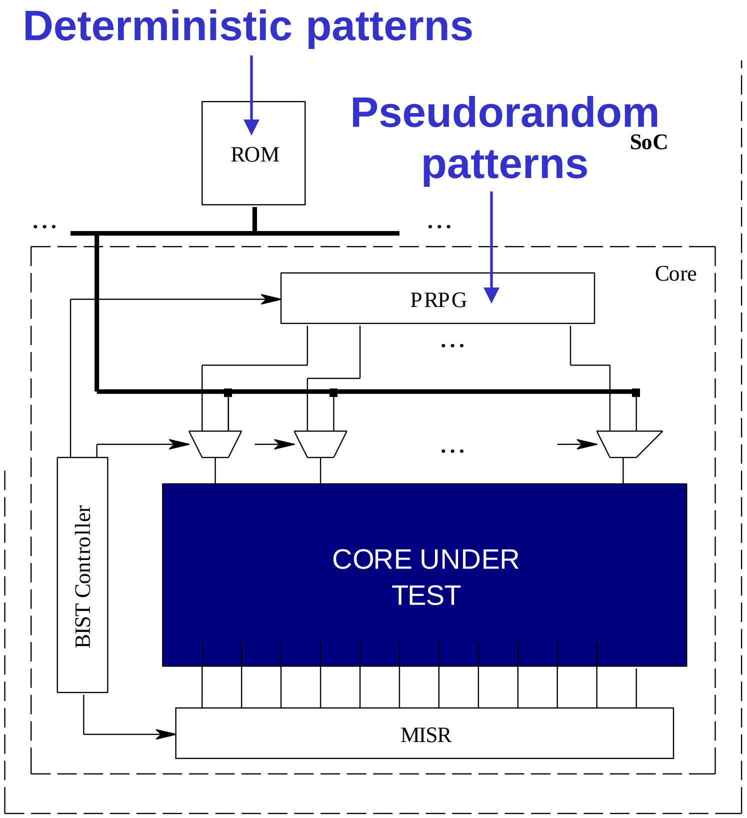
FBIST



FBIST: collection and analysis of samples during the working mode

Fault coverage is better, however, still very low (ranging from 42% to 70%)

Hybrid Built-In Self-Test



Deterministic patterns

Pseudorandom patterns

Hybrid test set contains pseudorandom and deterministic vectors

Pseudorandom test is improved by a stored test set which is specially generated to target the random resistant faults

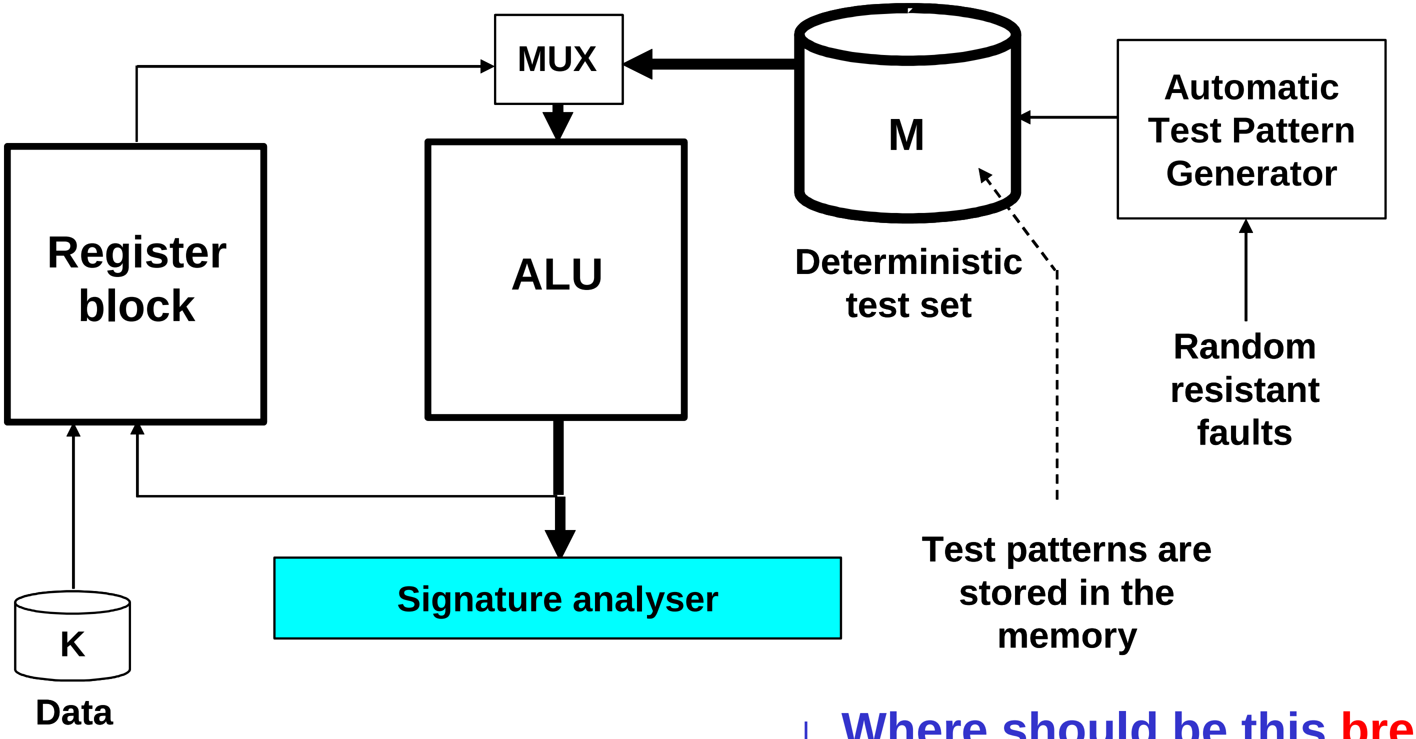
Optimization problem:

Where should be this breakpoint?

Pseudorandom Test | **Determ. Test**

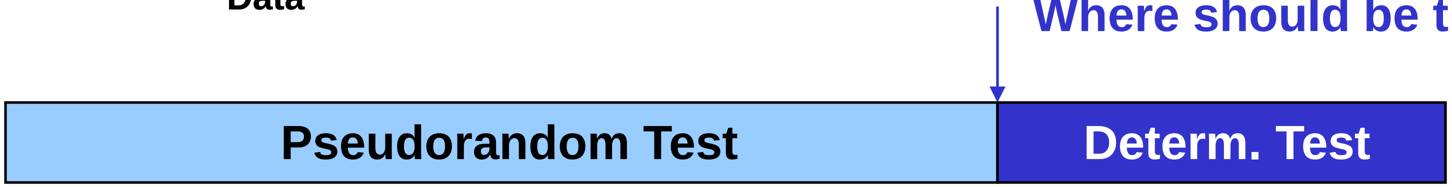
Hybrid Functional BIST for Divider

Hybrid Functional BIST implementation



Test patterns are stored in the memory

Where should be this breakpoint?



Cost Functions for Hybrid Functional BIST

Total cost:

$$\boxed{C_{Total} = C_{FB\_Total} + C_{D\_Total}}$$

The cost of functional test part:

$$C_{FB\_Total} = C_{FB\_Const} + \alpha C_{FB\_T} + \beta C_{FB\_M}$$

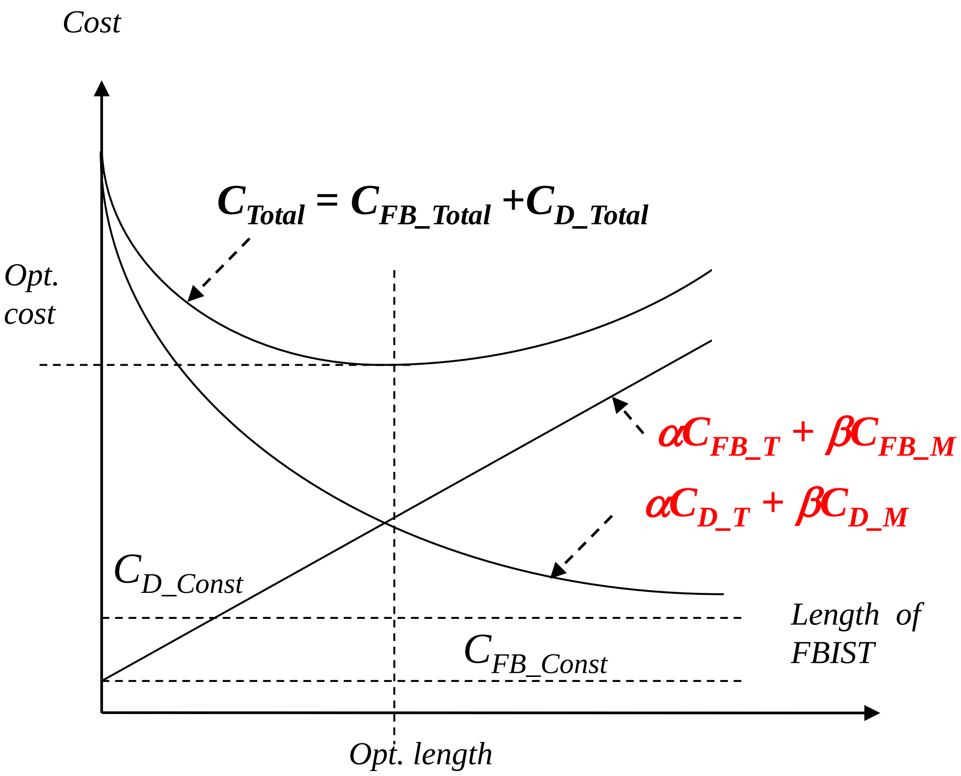
The cost of deterministic test part:

$$C_{D\_Total} = C_{D\_Const} + \alpha C_{D\_T} + \beta C_{D\_M}$$

C_{FB_Const}, C_{D_Const} - HW/SW overhead

C_{FB_T}, C_{D_T} - testing time cost

α, β - weights of time and memory expenses



Problem: minimize C_{Total}

Hybrid Functional BIST Quality

Hyb FBIST with multiple seeds (data operands)

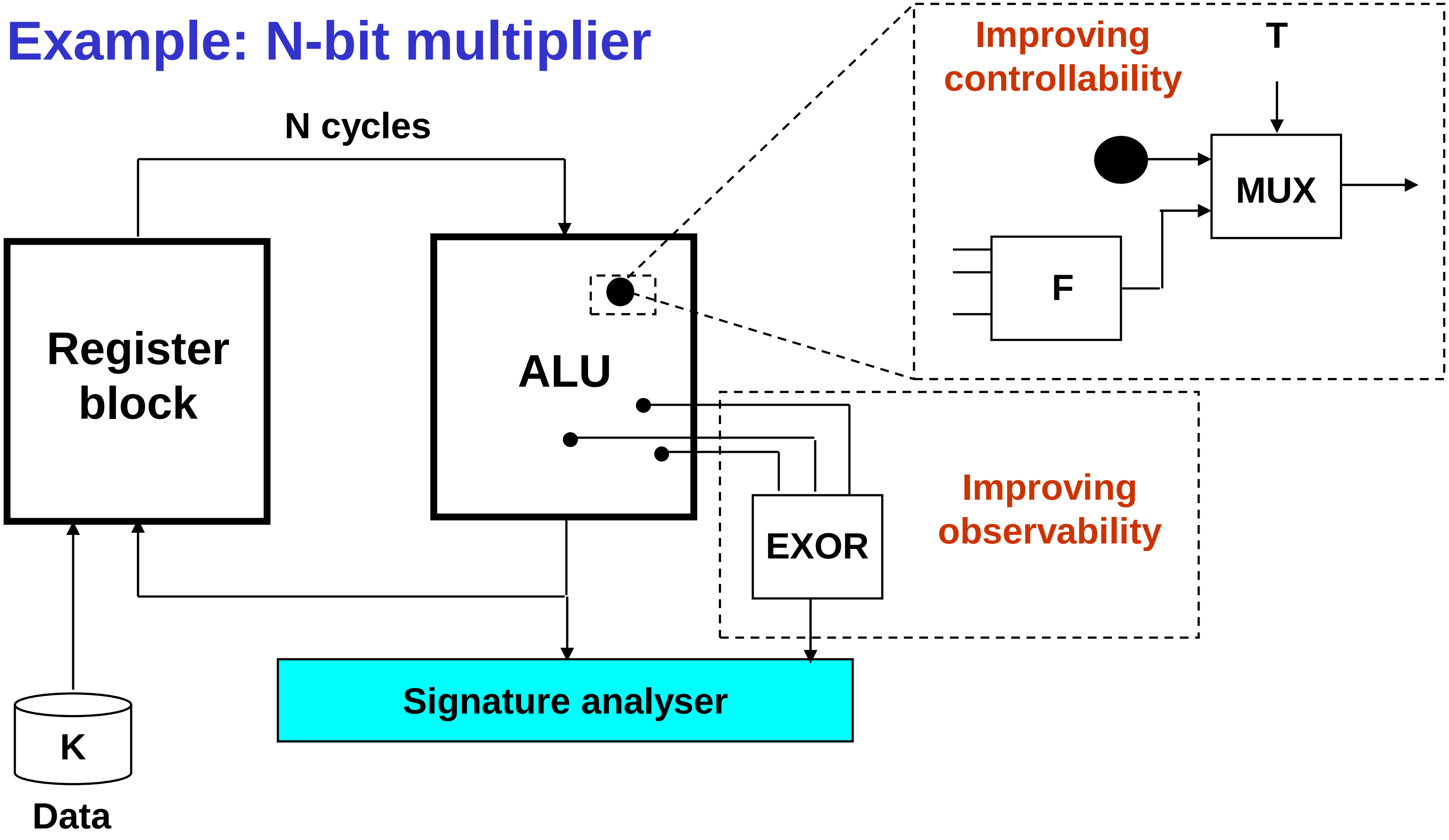
Functional test part					Determ. test part		Total cost
k	N_j	*N*	*FC* %	Total cost	*D*	Total cost	
0	0	0	100	0	58	6148	6148
1	108	108	66,8	140	24	2544	2684
2	105	213	76,7	**277**	**18**	1908	**2185**
3	113	326	83,3	518	17	1802	2320
4	108	434	85,5	690	16	1696	2386
5	110	544	88,4	864	15	1590	2454

k – number of operands used in the FBIST

The fault coverage increases if ***k*** increases

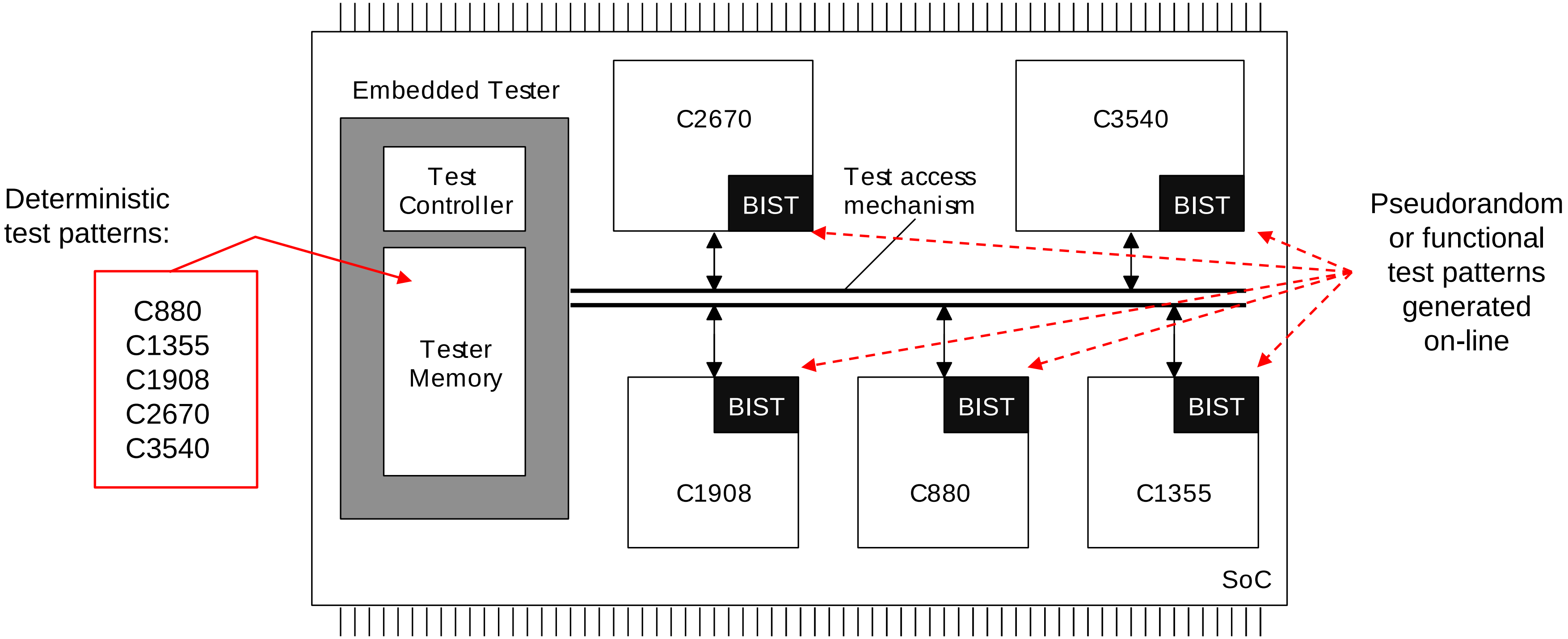
Functional Self-Test with DFT

Example: N-bit multiplier

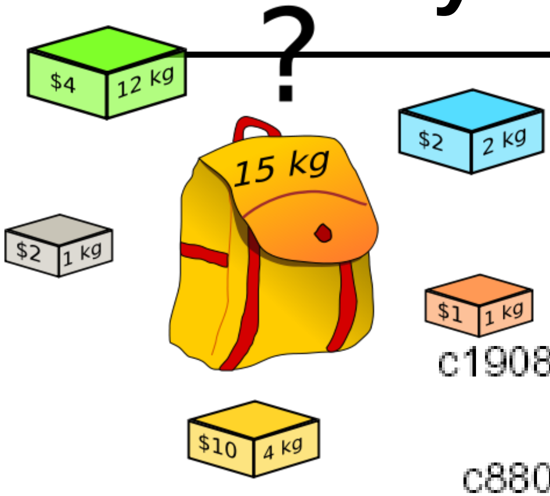


Hybrid BIST for Multiple Cores

Embedded tester for testing multiple cores



Hybrid BIST for Multiple Cores



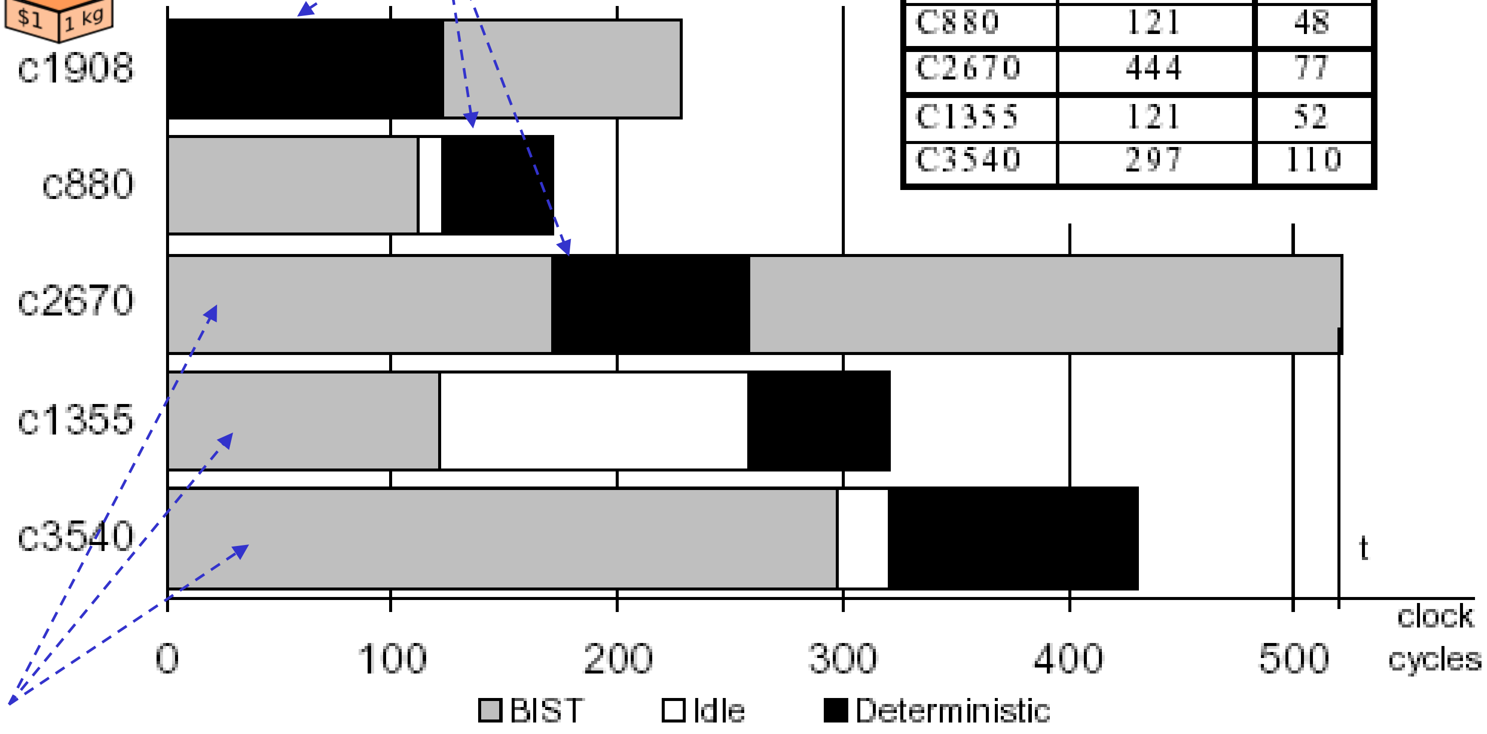
How to pack knapsack?

How to compress the test sequence?

Deterministic test (DT)

The optimal test set for each core

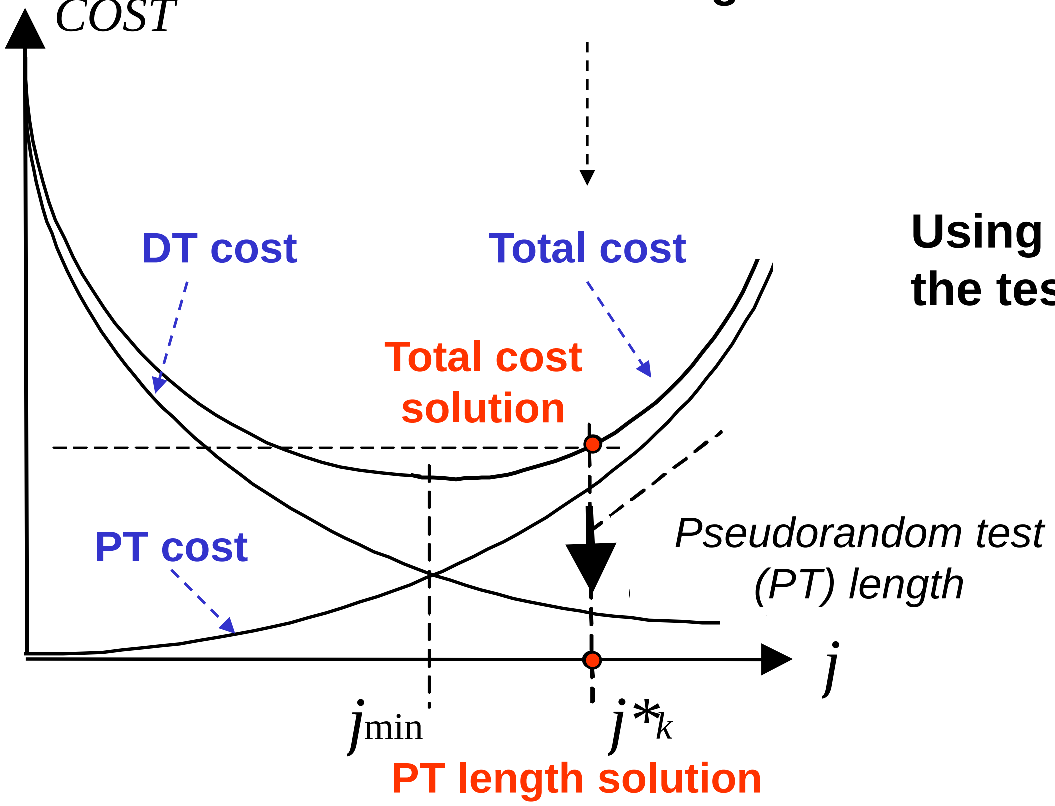
Core	Random	Det.
C1908	105	123
C880	121	48
C2670	444	77
C1355	121	52
C3540	297	110

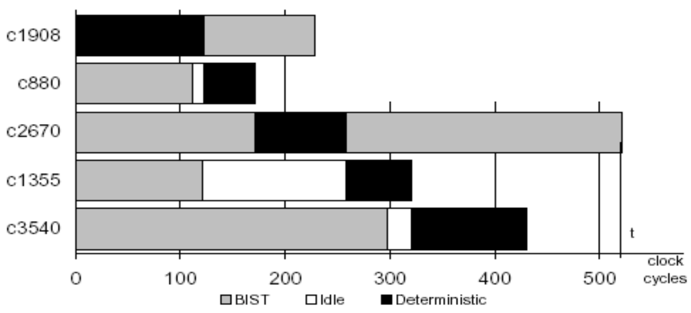


Pseudorandom test (PT)

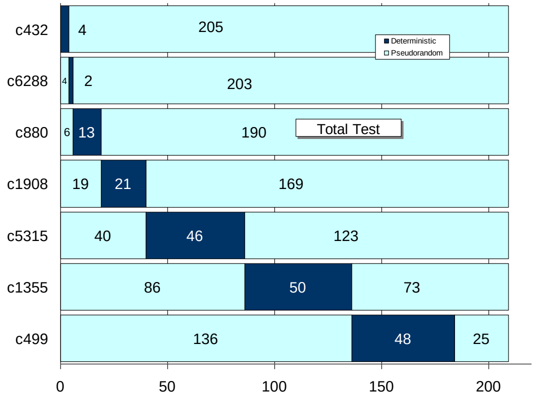
Total Test Cost Estimation

Using total cost solution we find the PT length:



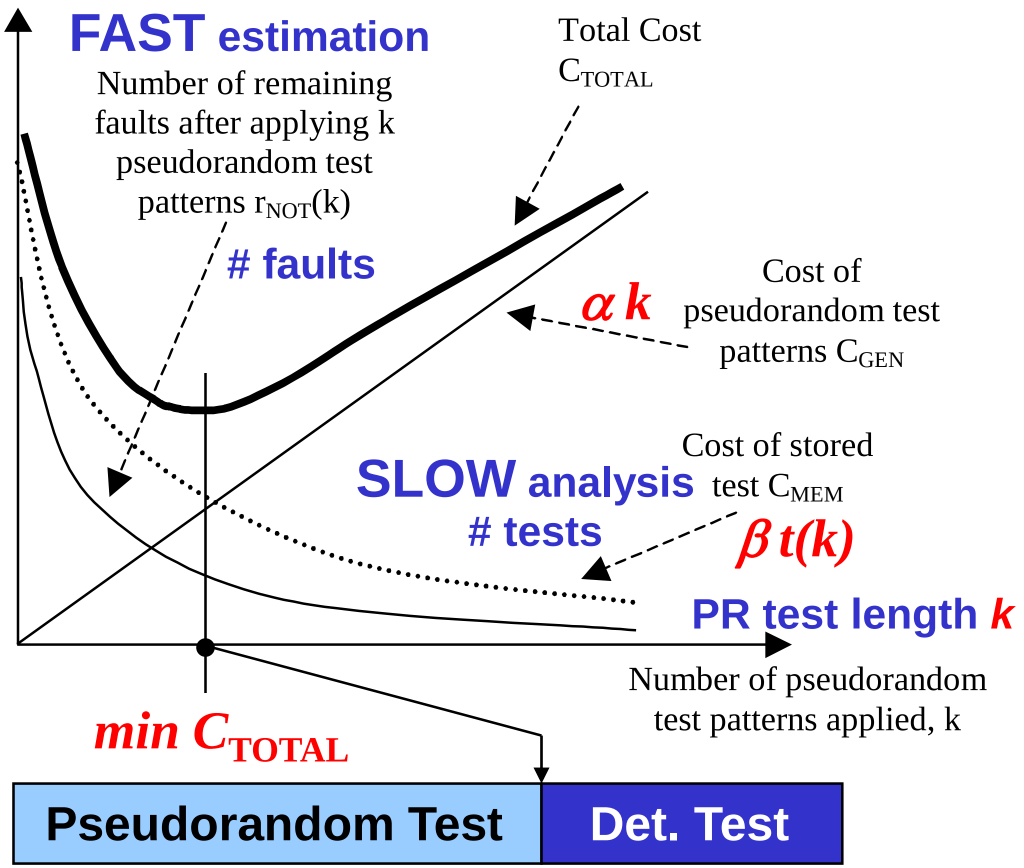


Using PT length, we calculate the test processes for all cores:



Multi-Core Hybrid BIST Optimization

Cost of BIST: $C_{TOTAL} = \alpha k + \beta t(k)$



FAST estimation

Number of remaining faults after applying k pseudorandom test patterns $r_{NOT}(k)$

# faults

Total Cost C_{TOTAL}

αk

Cost of pseudorandom test patterns C_{GEN}

SLOW analysis

# tests

Cost of stored test C_{MEM}

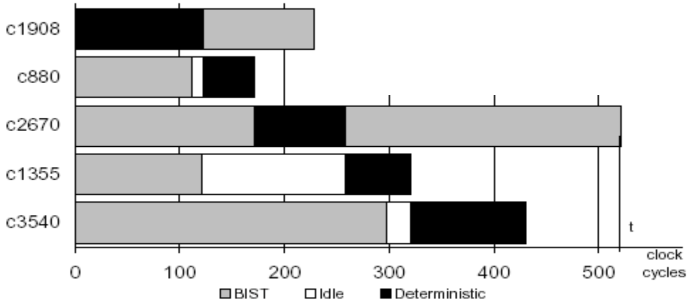
$\beta t(k)$

PR test length *k*

Number of pseudorandom test patterns applied, k

min C_{TOTAL}

Pseudorandom Test | **Det. Test**

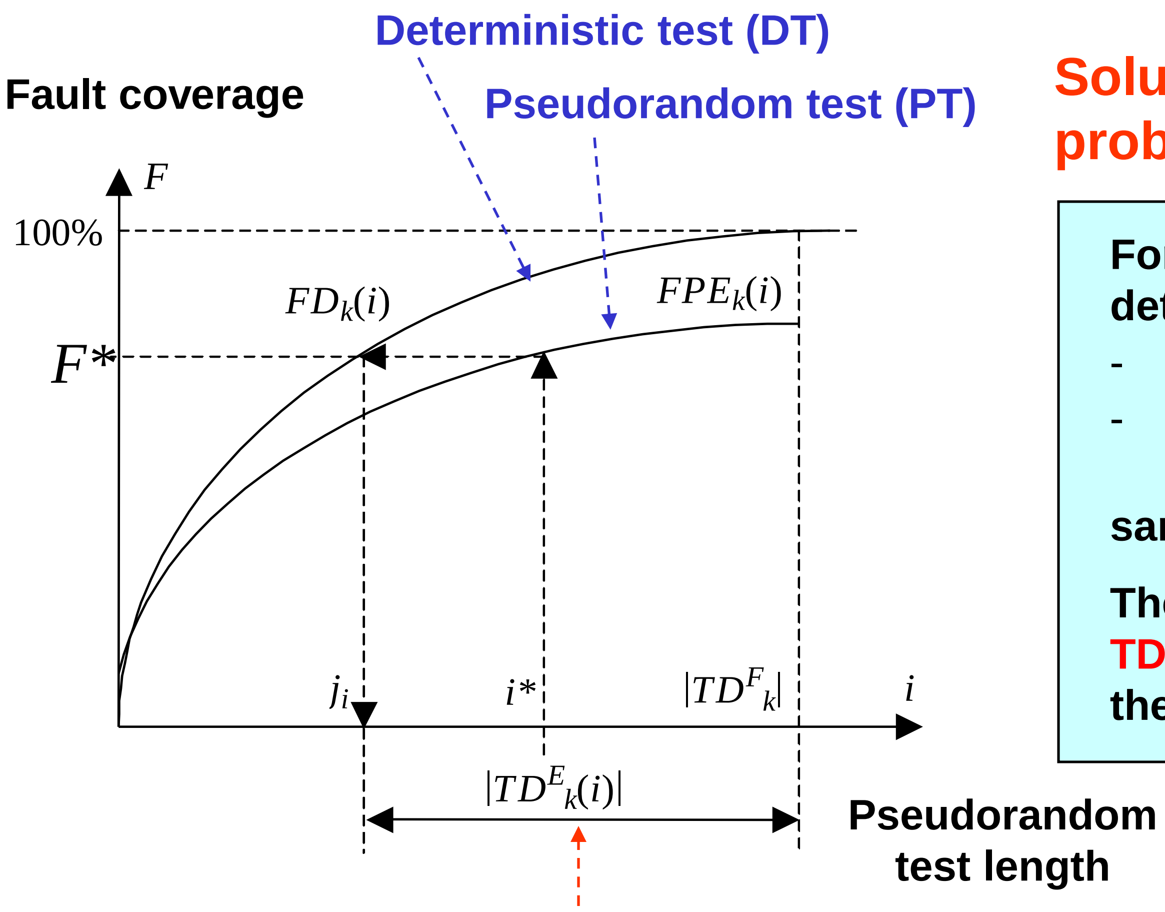


Two problems:

1) **Calculation of DT $\beta t(k)$ cost is difficult**
2) **We have to optimize *n* (!) processes**

How to avoid the calculation of the very expensive full DT $\beta t(k)$ cost curve?

Deterministic Test Length Estimation



Deterministic test length estimation for a single core

Solution of the first problem:

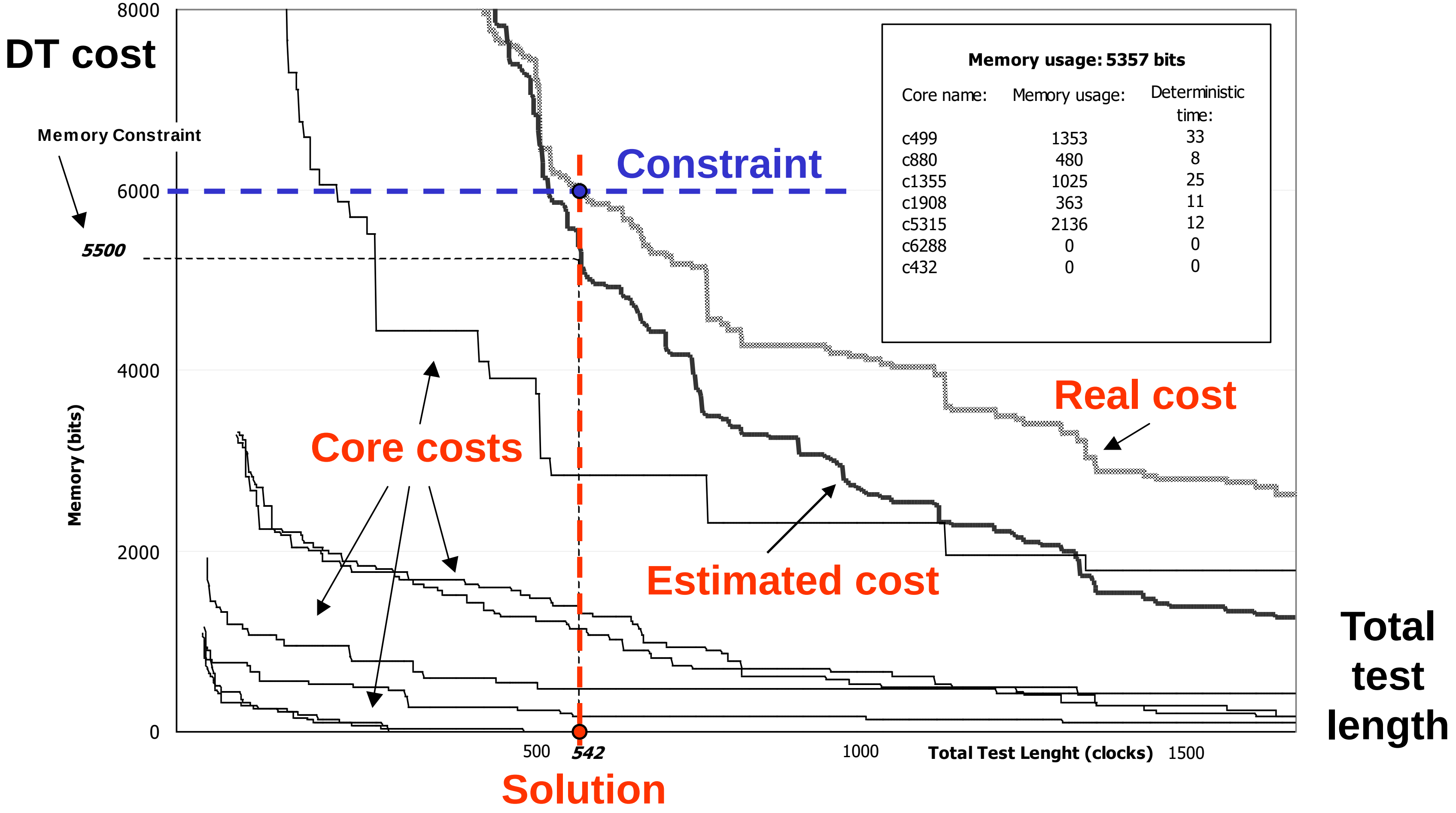
For each PT length **i*** we determine

- PT fault coverage **F***, and
- the imaginable part of DT $FD_k(i)$ to be used for the same fault coverage

Then the remaining part of DT $TD^E_k(i)$ will be the estimation of the DT length

Deterministic Test Cost Estimation

Total cost calculation of core costs:



Memory usage: 5357 bits

Core name:	Memory usage:	Deterministic time:
c499	1353	33
c880	480	8
c1355	1025	25
c1908	363	11
c5315	2136	12
c6288	0	0
c432	0	0

DT cost

Constraint

Core costs

Real cost

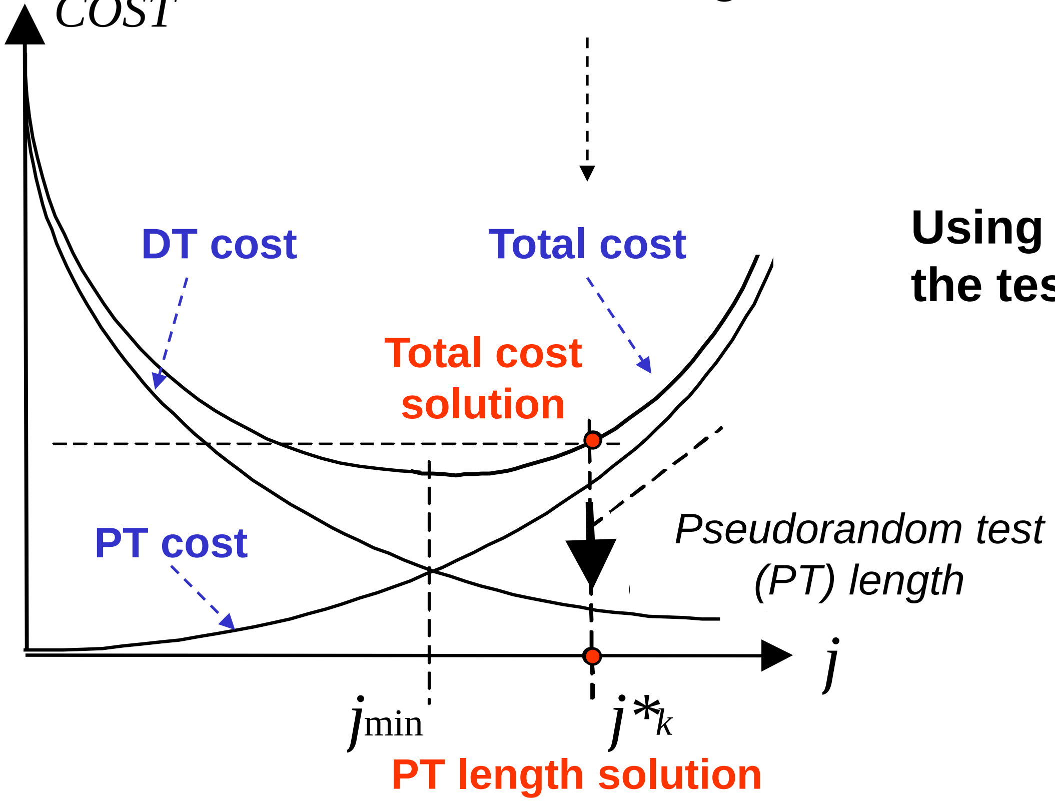
Estimated cost

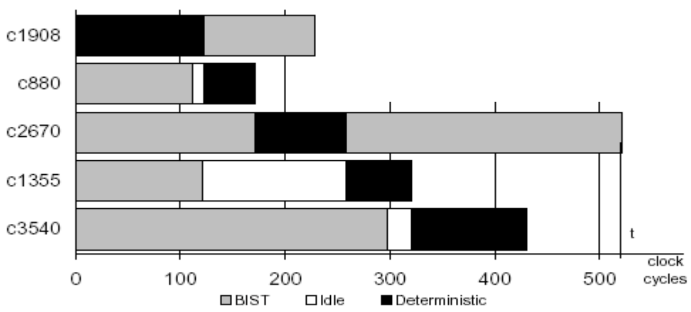
Solution

Total test length

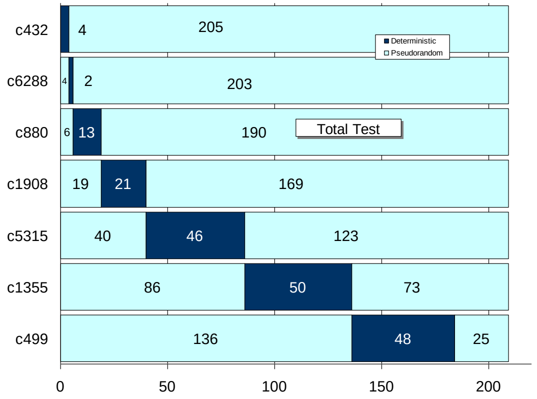
Total Test Cost Estimation

Using total cost solution we find the PT length:



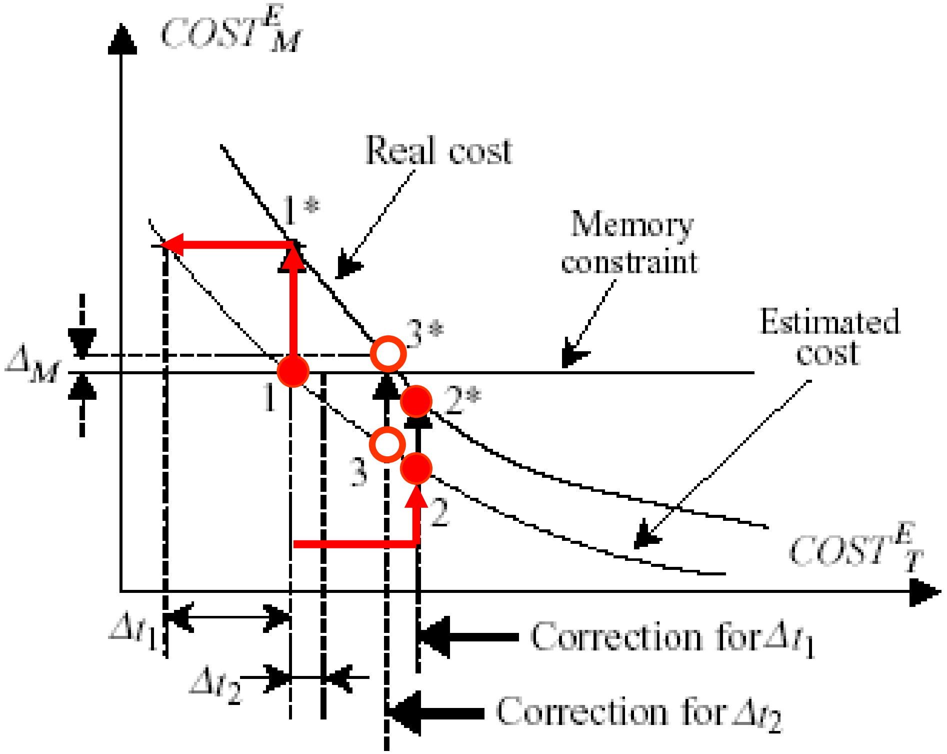


Using PT length, we calculate the test processes for all cores:



Multi-Core Hybrid BIST Optimization

Iterative optimization process:



1 - First estimation

1* - Real cost calculation

2 - Correction of the estimation

2* - Real cost calculation

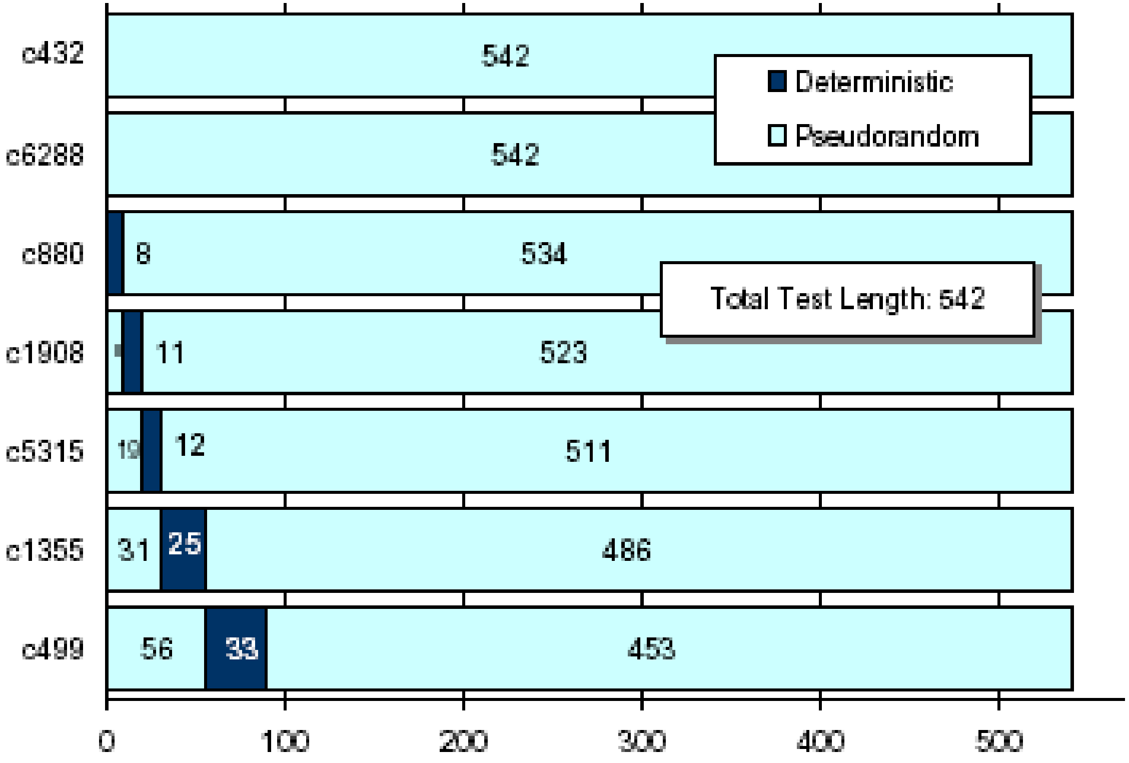
3 - Correction of the estimation

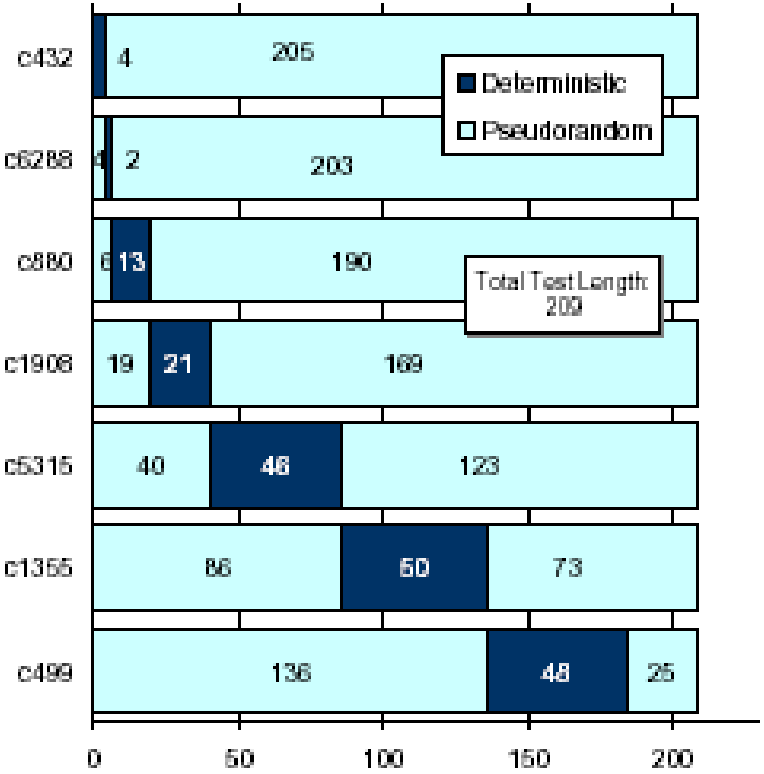
3* - Final real cost

G.Jervan, P.Eles, Z.Peng, R.Ubar, M.Jenihhin. Test Time Minimization for Hybrid BIST of Core-Based Systems. *Asian Test Symposium 2003,* Xi’an, China, November 17-19, 2003,

Optimized Multi-Core Hybrid BIST

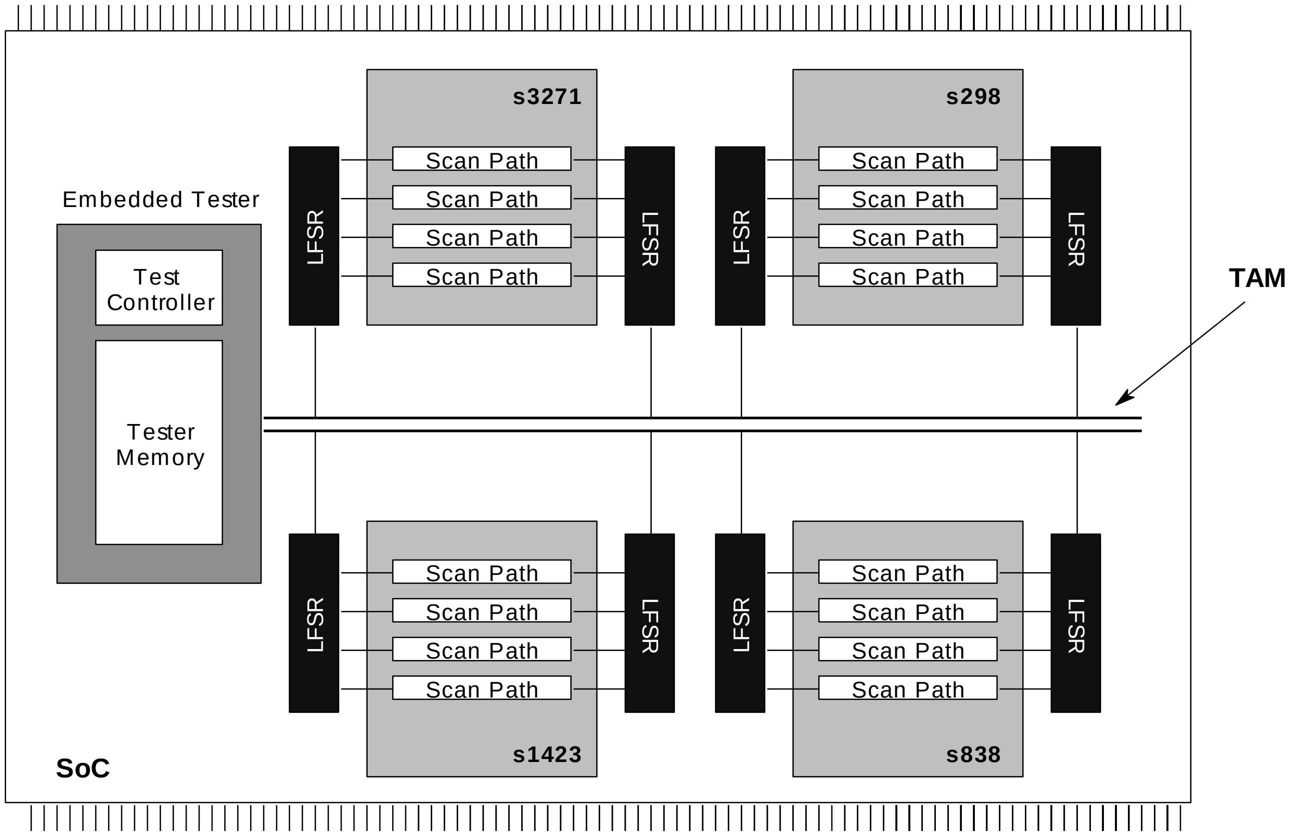
Pseudorandom test is carried out in parallel, deterministic test - sequentially





Test-per-Scan Hybrid BIST

Every core's BIST logic is capable to produce a set of independent pseudorandom test
The pseudorandom test sets for all the cores can be carried out simultaneously

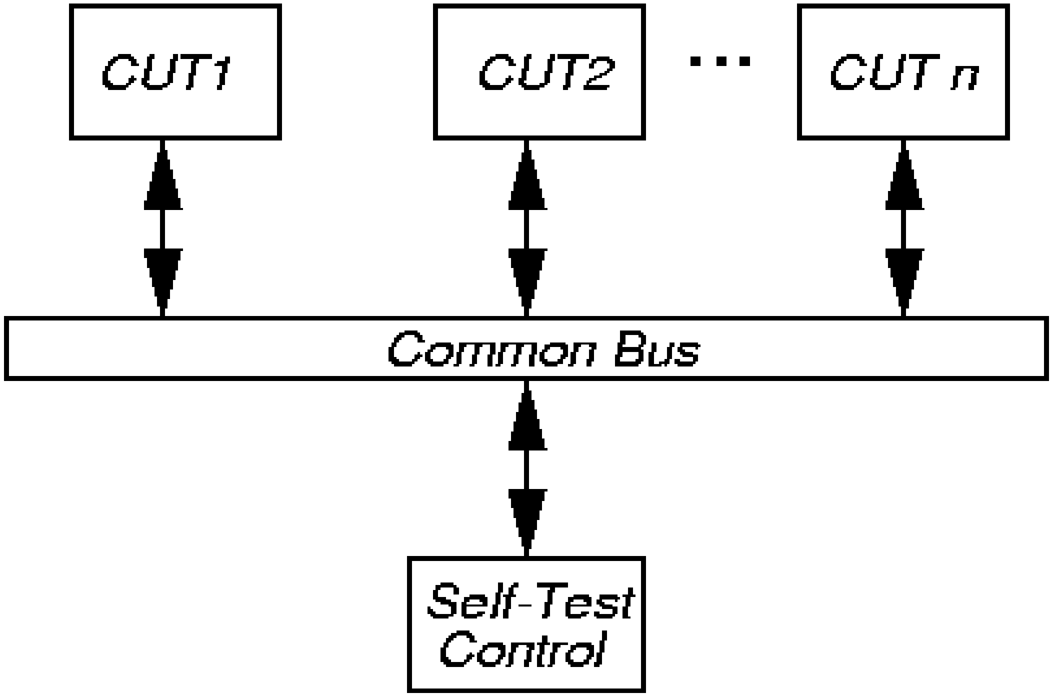


Deterministic tests can only be carried out for one core at a time

Only one test access bus at the system level

is needed.

Bus-Based BIST Architecture

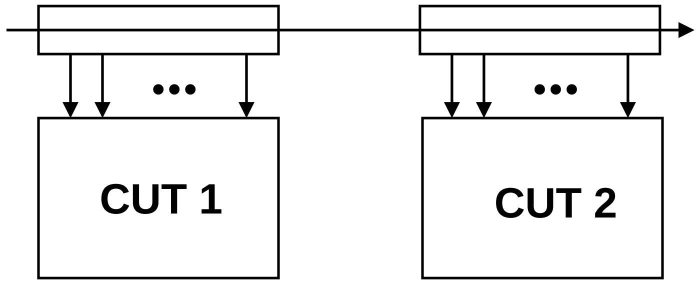


- *Self-test control* broadcasts patterns to each CUT over bus – parallel pattern generation
- Awaits bus transactions showing CUT’s responses to the patterns: serialized compaction

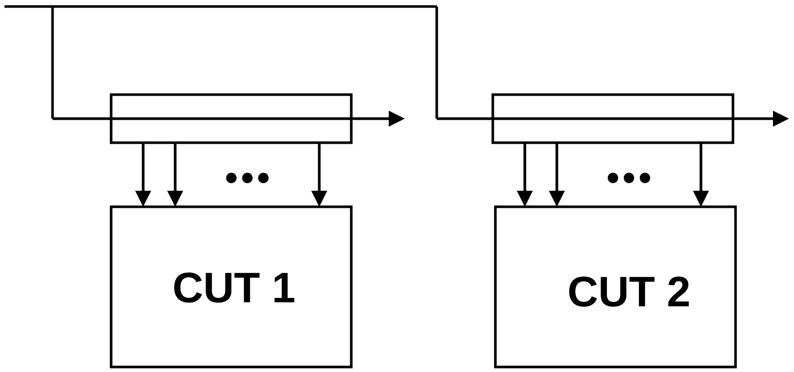
Source: VLSI Test: Bushnell-Agrawal

Broadcasting Test Patterns in BIST

Concept of test pattern sharing via novel scan structure – to reduce the test application time:



Traditional single scan design

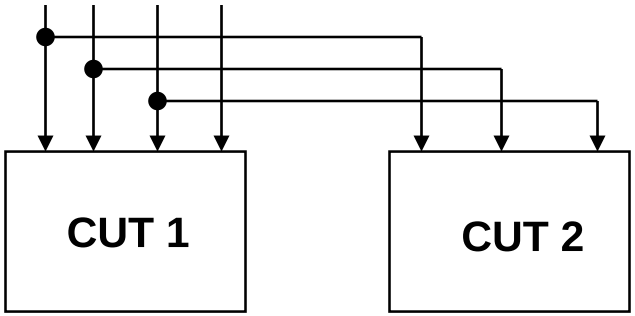


Broadcast test architecture

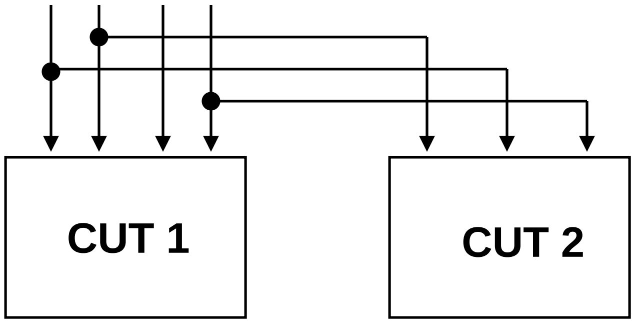
While one module is tested by its test patterns, the same test patterns can be applied simultaneously to other modules in the manner of pseudorandom testing

Broadcasting Test Patterns in BIST

Examples of connection possibilities in Broadcasting BIST:



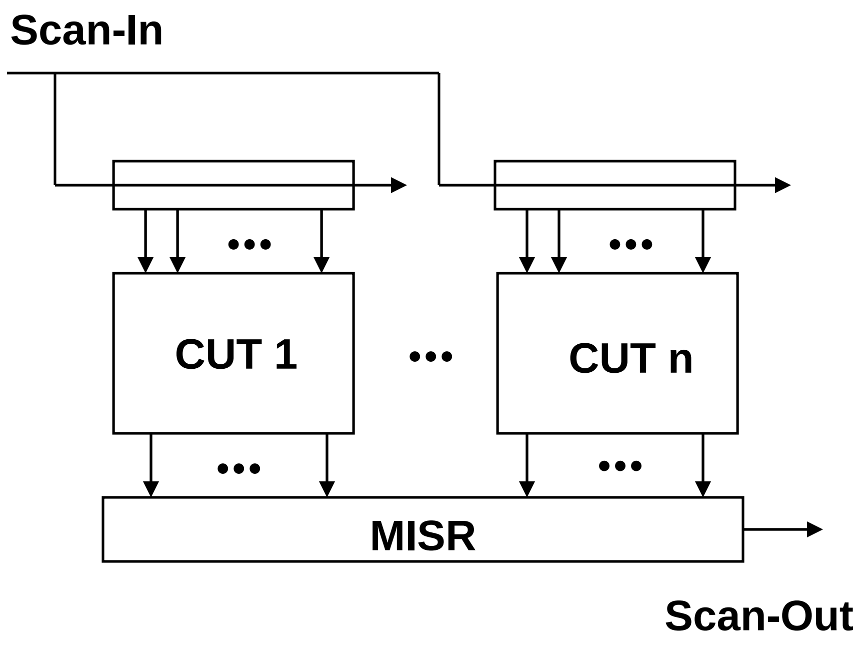
***j*-to-*j* connections**



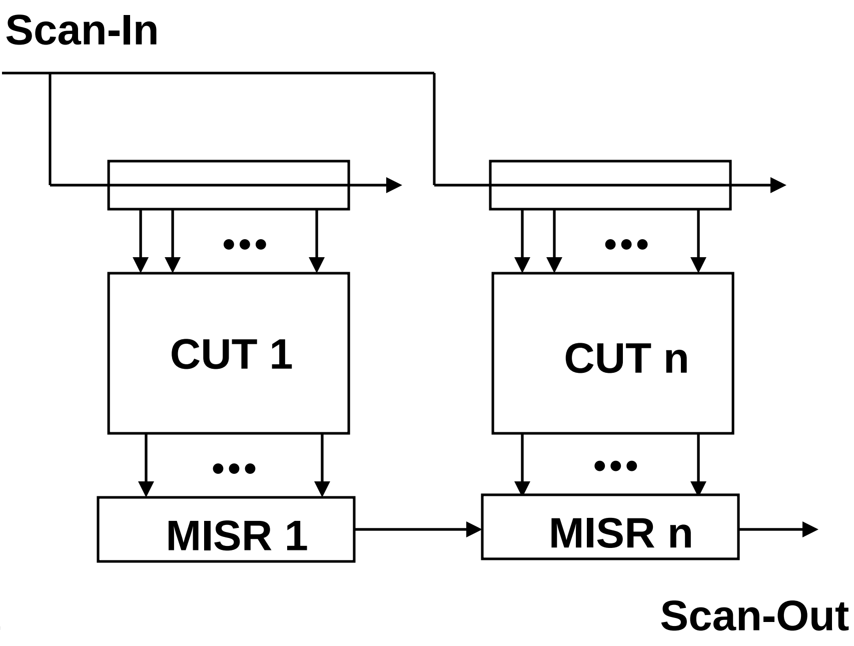
Random connections

Broadcasting Test Patterns in BIST

Scan configurations in Broadcasting BIST:



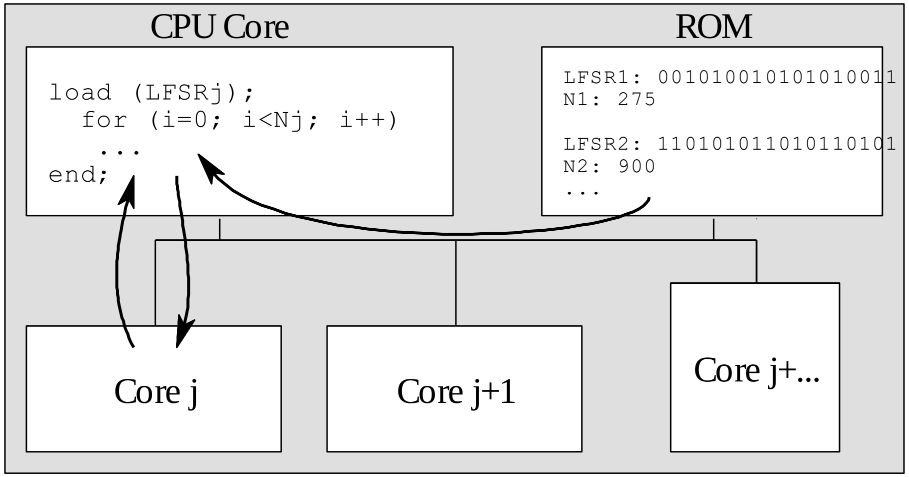
Common MISR



Individual and multiple MISRs

Software BIST

Software based test generation:



```
load (LFSRj);
  for (i=0; i<Nj; i++)
    ...
end;
```

```
LFSR1: 001010010101010011
N1: 275

LFSR2: 110101011010110101
N2: 900
...
```

To reduce the hardware overhead cost in the BIST applications the hardware LFSR can be replaced by software

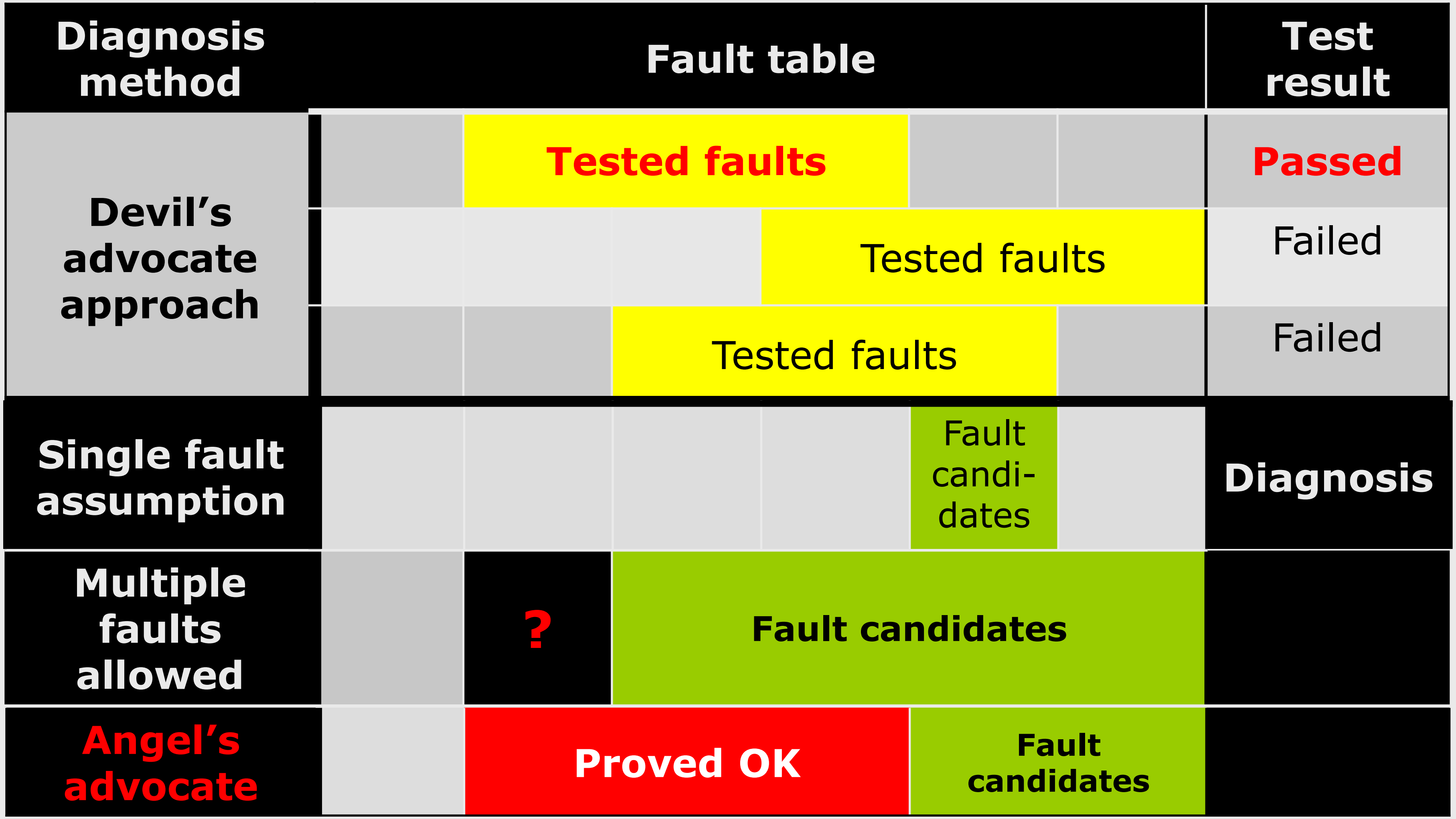
Software BIST is especially attractive to test SoCs, because of the availability of computing resources directly in the system (a typical SoC usually contains at least one processor core)

The TPG software is the same for all cores and is stored as a single copy
All characteristics of the LFSR are specific to each core and stored in the ROM
They will be loaded upon request.
For each additional core, only the BIST characteristics for this core have to be stored

Embedded Built-in Self-Diagnosis (BISD)

- Introduction to Fault Diagnosis
  - Combinational diagnosis (effect-cause approach)
  - Sequential (adaptive) diagnosis (cause-effect approach)
- General conception of embedded BISD
- Diagnostic resolution
  - Intersection based on test subsequences
  - Intersection based on using signature analyzers
- Fault model free diagnosis
- Fault evidence based diagnosis

Why Fault Masking is Important Issue?



Diagnosis method

Fault table

Test result

Devil's advocate approach

Tested faults

Passed

Tested faults

Failed

Tested faults

Failed

Single fault assumption

Fault candi-dates

Diagnosis

Multiple faults allowed

?

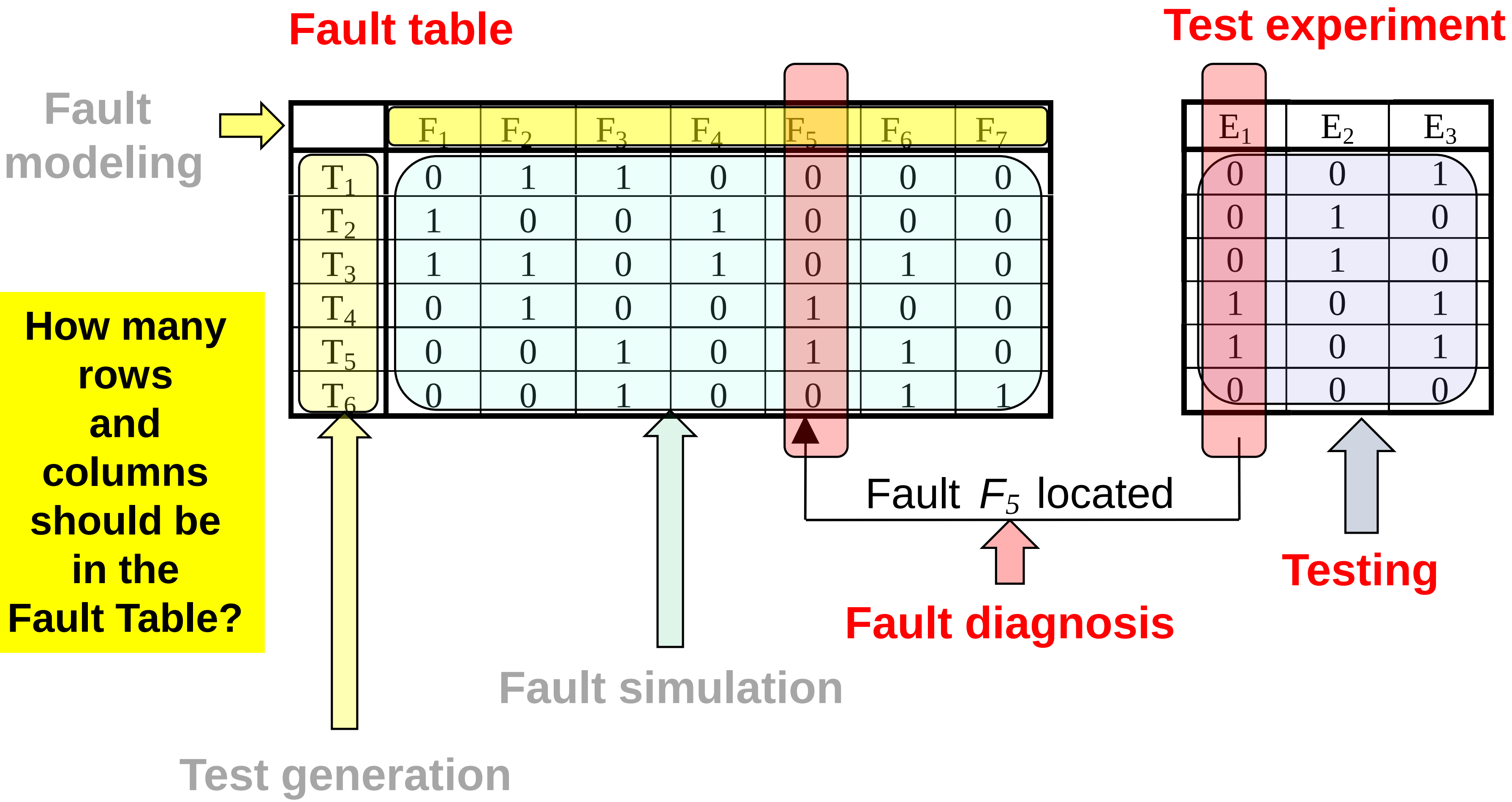
Fault candidates

Angel's advocate

Proved OK

Fault candidates

Fault Diagnosis



Fault table

Test experiment

Fault modeling

	F_1	F_2	F_3	F_4	F_5	F_6	F_7
T_1	0	1	1	0	0	0	0
T_2	1	0	0	1	0	0	0
T_3	1	1	0	1	0	1	0
T_4	0	1	0	0	1	0	0
T_5	0	0	1	0	1	1	0
T_6	0	0	1	0	0	1	1

E_1	E_2	E_3
0	0	1
0	1	0
0	1	0
1	0	1
1	0	1
0	0	0

Fault F_5 located

Fault diagnosis

Testing

Fault simulation

Test generation

How many rows and columns should be in the Fault Table?

Sequential Fault Diagnosis

Sequential fault diagnosis by Edge-Pin Testing (cause-effect)

	F_1	F_2	F_3	F_4	F_5	F_6	F_7
T_1	0	1	1	0	0	0	0
T_2	1	0	0	1	0	0	0
T_3	1	1	0	1	0	1	0
T_4	0	1	0	0	1	0	0
T_5	0	0	1	0	1	1	0
T_6	0	0	1	0	0	1	1

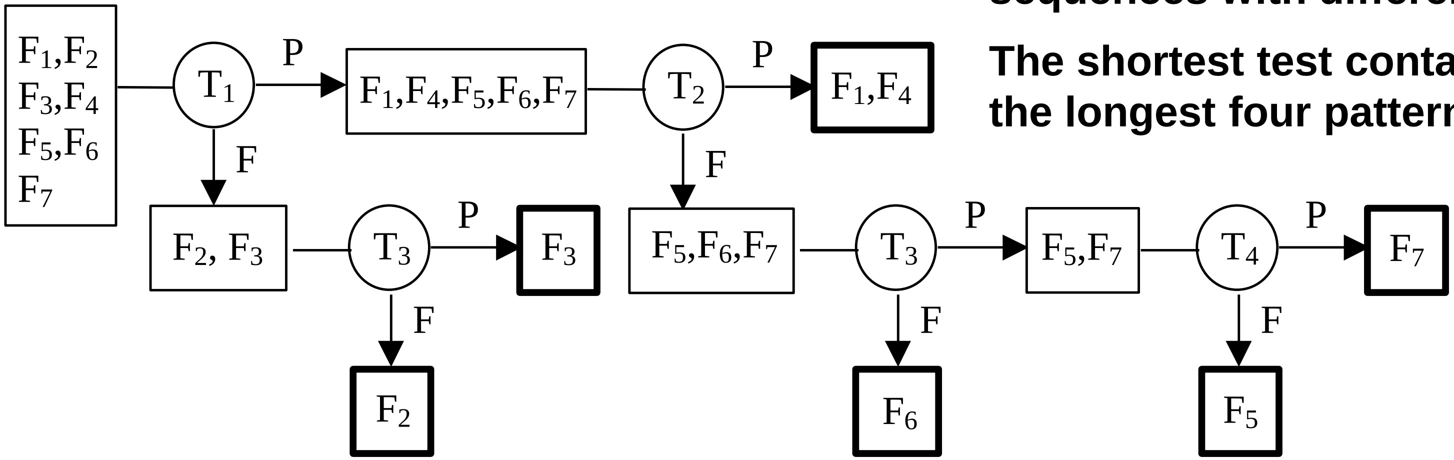
Diagnostic tree

Two faults *F_1,F_4* remain indistinguishable

Not all test patterns used in the fault table are needed

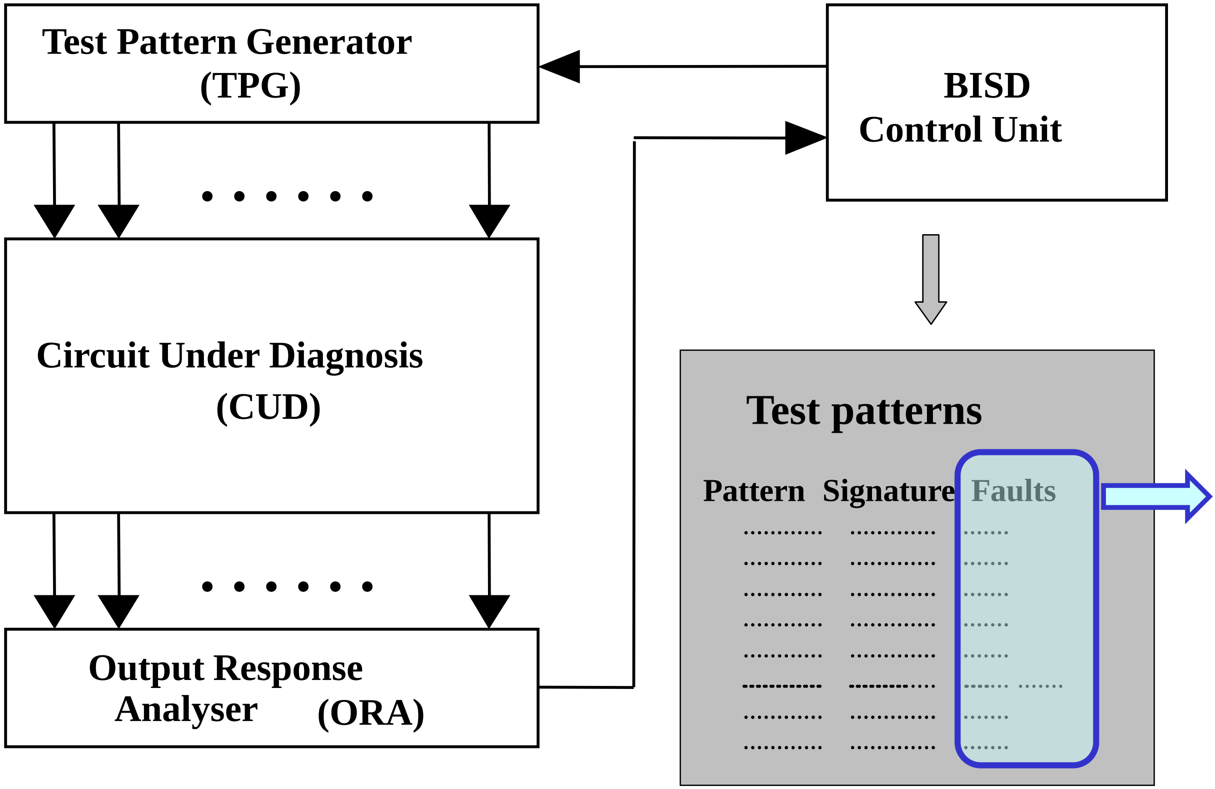
Different faults need for identifying test sequences with different lengths

The shortest test contains two patterns, the longest four patterns

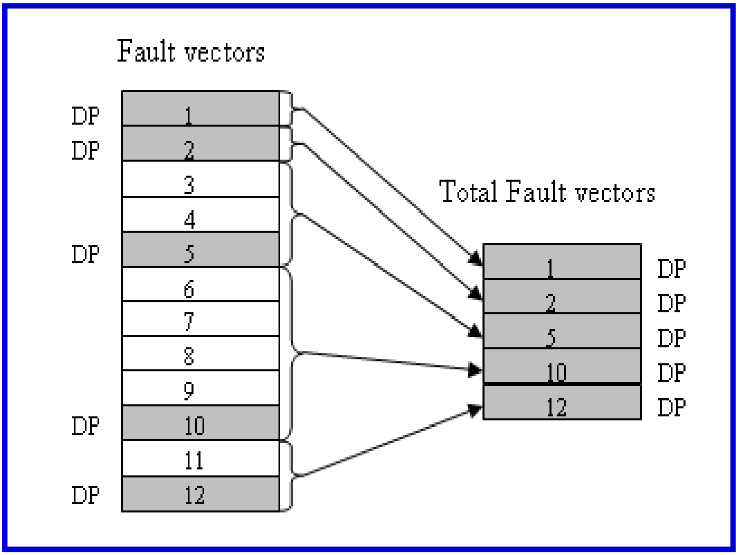


Embedded BIST Based Fault Diagnosis

BISD scheme:



Pseudorandom test sequence:

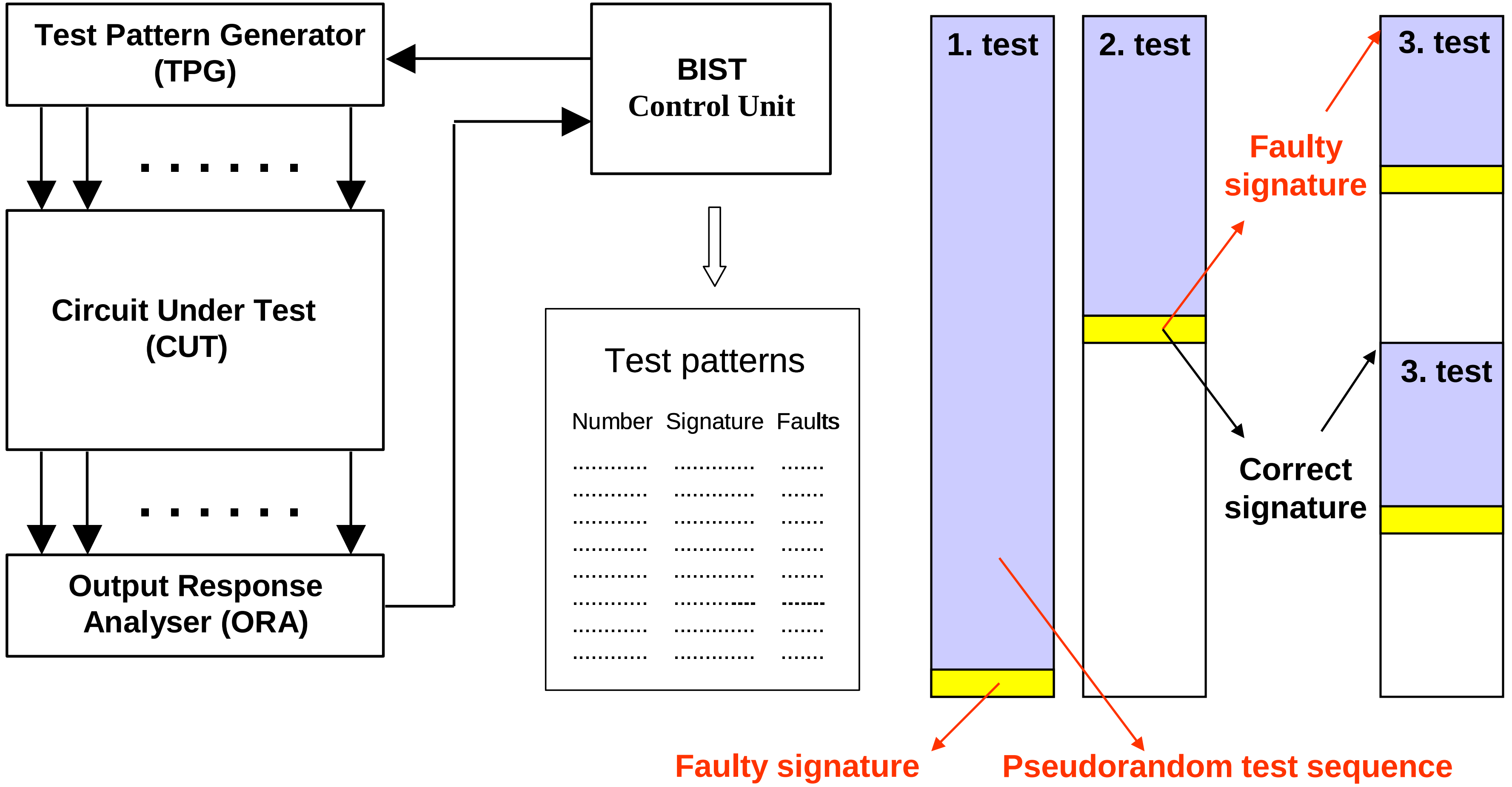


Diagnostic Points (DPs) – patterns that detect new faults

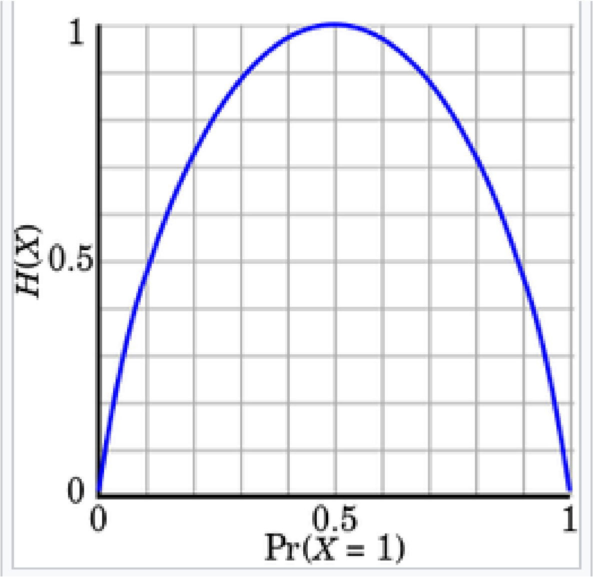
Further minimization of DPs – as a tradeoff with diagnostic resolution

Built-In Fault Diagnosis

Diagnosis procedure:



Introduction to Information Theory



Entropy H_X of a discrete random variable X is a measure of the amount of ***uncertainty*** associated with the value of X

$$H = -\sum_i p_i \log_2 (p_i)$$

where p_i is the probability of occurrence of the i-th possible value of the source symbol; (the entropy is given in the units of "bits" (per symbol) because it uses log of base 2)

$$H_X = - p \log_2 p - (1-p) \log_2 (1-p)$$

$$\mathbf{I = - p \log_2 p - (1-p) \log_2 (1-p)}$$

p – probability of detecting a fault

Built-In Fault Diagnosis

Measuring of information we get from the test:

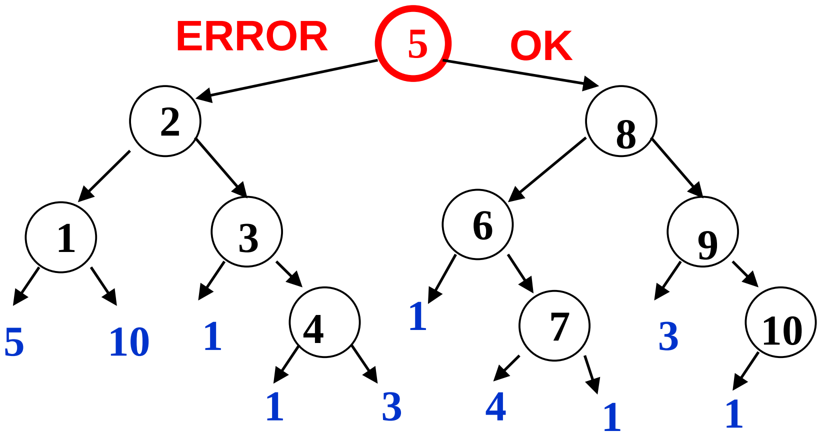
$$I = -\,p\,log_2\,p - (1\text{-}p)\,log_2\,(1\text{-}p)$$

p – probability of detecting a fault

Pseudorandom test fault simulation (detected faults)

№	All faults	New faults	Coverage
1	5	5	16.67%
2	15	10	50.00%
3	16	1	53.33%
4	17	1	56.67%
5	20	3	66.67%
6	21	1	70.00%
7	25	4	83.33%
8	26	1	86.67%
9	29	3	96.67%
10	30	1	100.00%

Binary search with bisectioning of test patterns



Average number of test sessions: 3,3

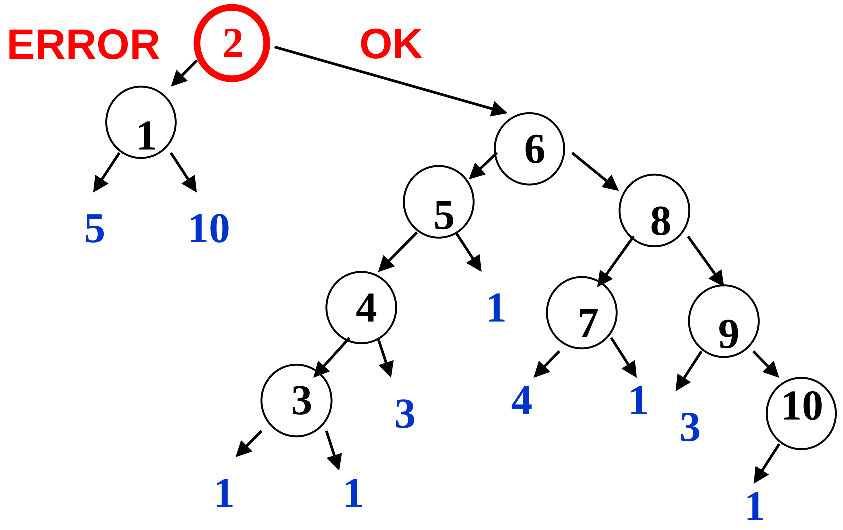
Average number of clocks: 8,67

Built-In Fault Diagnosis

Pseudorandom test fault simulation (detected faults)

№	All faults	New faults	Coverage
1	5	5	16.67%
2	15	10	50.00%
3	16	1	53.33%
4	17	1	56.67%
5	20	3	66.67%
6	21	1	70.00%
7	25	4	83.33%
8	26	1	86.67%
9	29	3	96.67%
10	30	1	100.00%

Binary search with bisectioning of faults



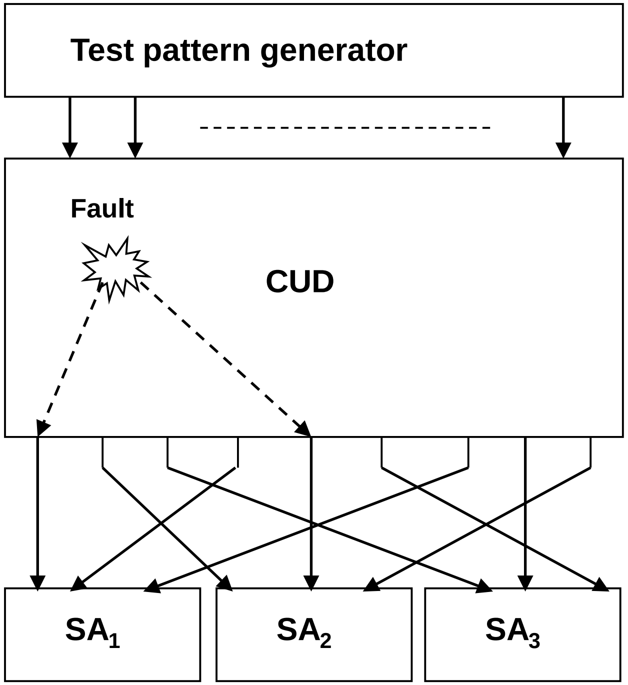
Average number of test sessions: 3,06

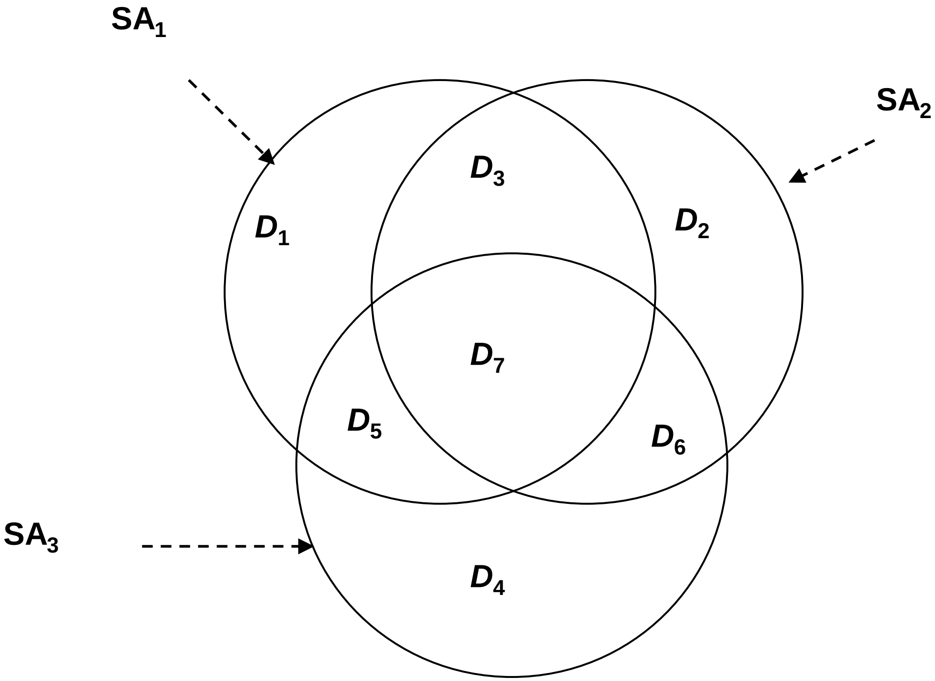
Average number of clocks: 6,43

Built-In Fault Diagnosis

Diagnosis with multiple signatures

(based on reasoning of spacial information):



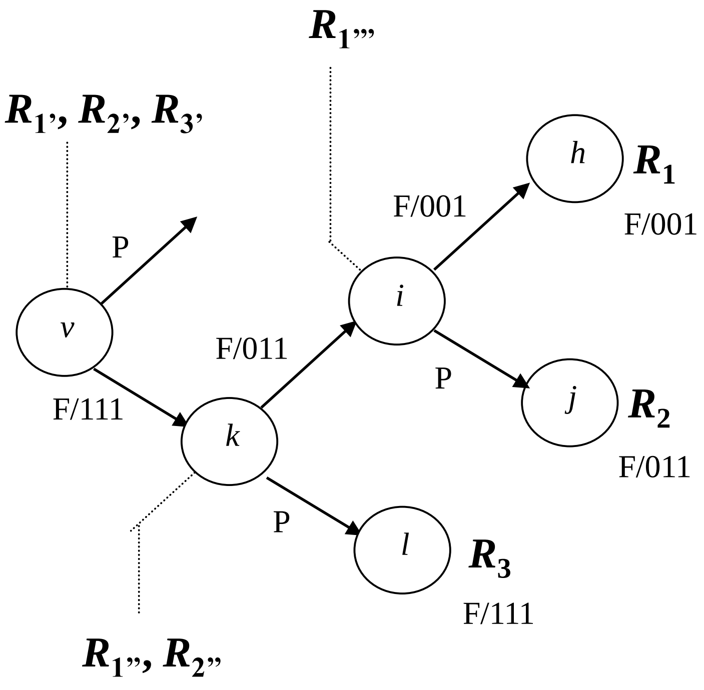


Built-In Fault Diagnosis

Diagnosis with multiple signatures:

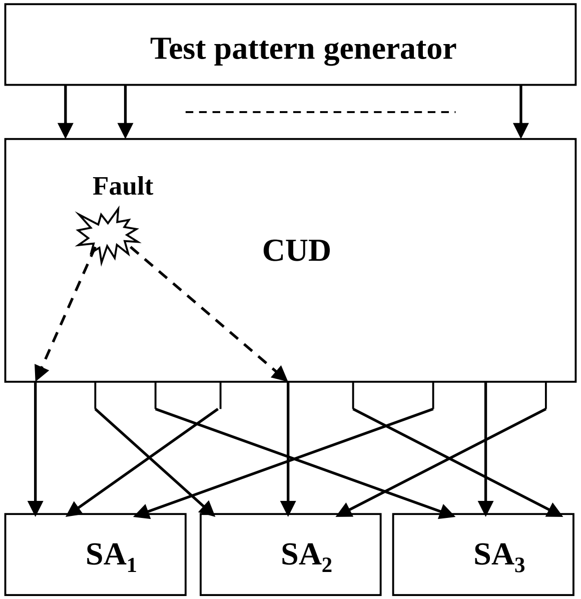
No	Codeword			Diagnosis
h	0	0	1	R_1
i	0	0	1	R_1'''
j	0	1	1	R_2
k	0	1	1	R_1'', R_2''
l	1	1	1	R_3
v	1	1	1	R_1', R_2', R_3'

Diagnostic tree



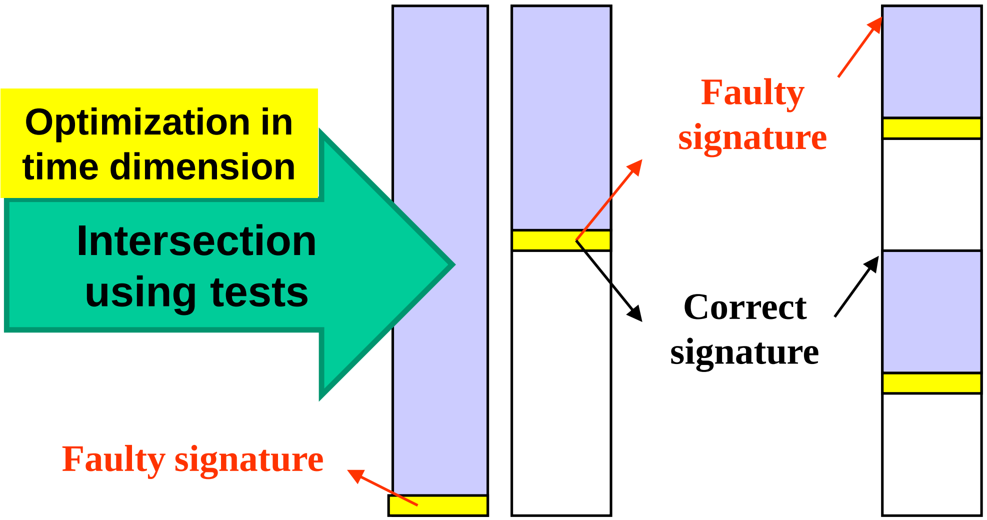
Built-In Fault Diagnosis

BIST with multiple signature analyzers



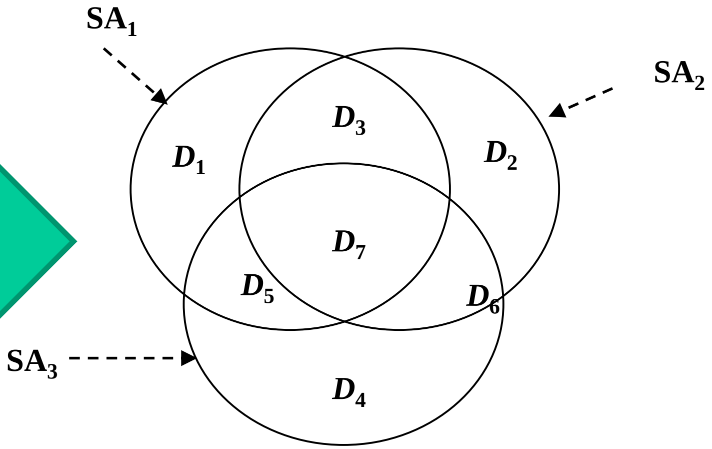
Optimization in time dimension

Intersection using tests



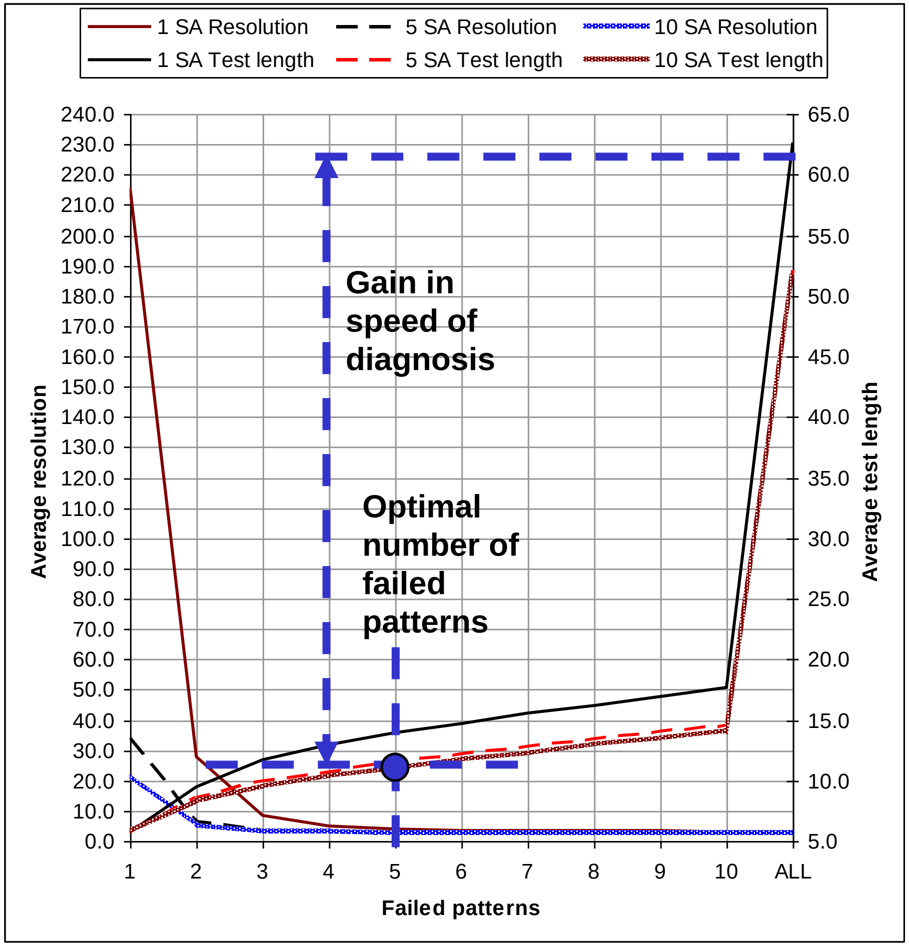
Optimization in space dimension

Intersection using SA-s



Optimization of the interface between CUD and SA-s

Built-In Fault Diagnosis



Diagnosis with multiple signatures:

Measured:

- **average resolution**
- **average test length**

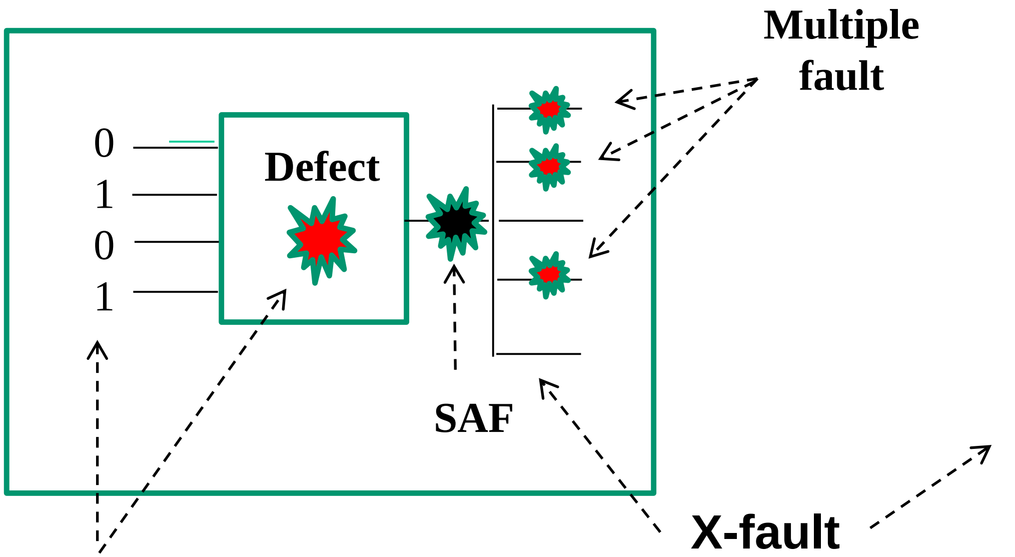
Compared: 1SA, 5SA, 10SA

Gain in test length: 6 times

R.Ubar, S.Kostin, J.Raik. Embedded Fault Diagnosis in Digital Systems with BIST. J. of Microprocessors and Microsystems, Volume 32, August 2008, pp. 279-287.

Extended Fault Models

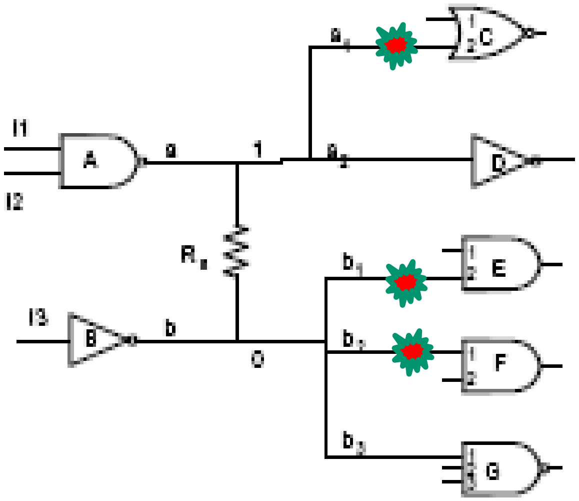
Extensions of the parallel critical path tracing for two large general fault classes for modeling physical defects:



Multiple fault

SAF

Resistive bridge fault

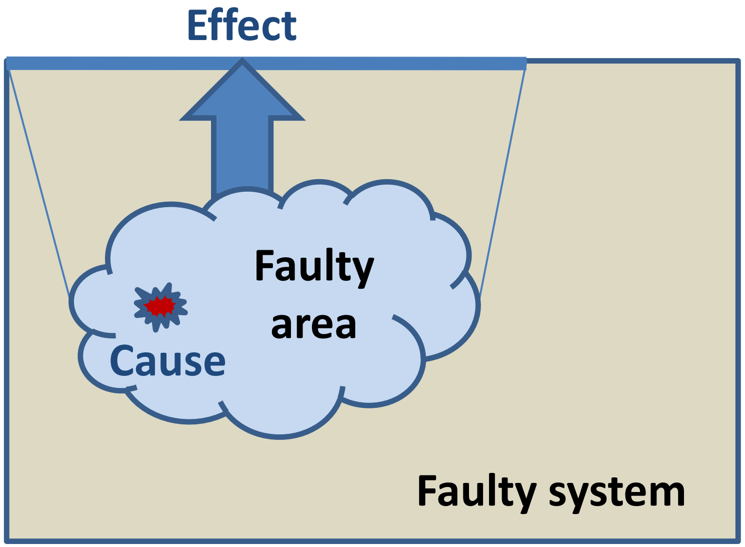


Conditional fault
Pattern fault
Constrained SAF
Single faulty signal

X-fault
Byzantine fault
Bridges
Stuck-opens
Multiple faulty signal

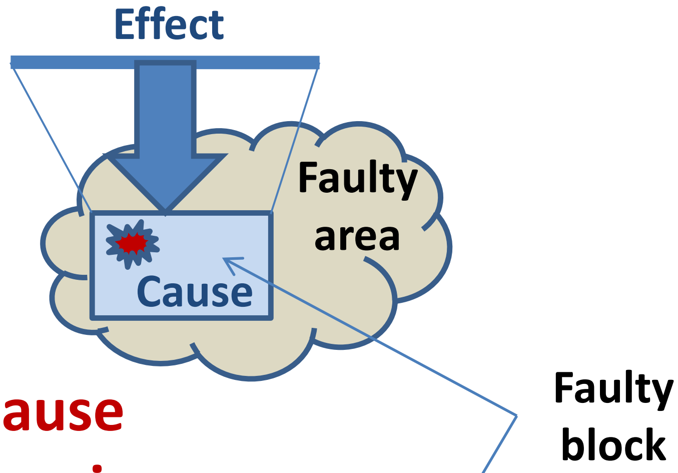
Fault-Model Free Fault Diagnosis

Combined cause-effect and effect-cause diagnosis



1) Cause-Effect Fault Diagnosis

Suspected faulty area is located based on the fault table (dictionary)

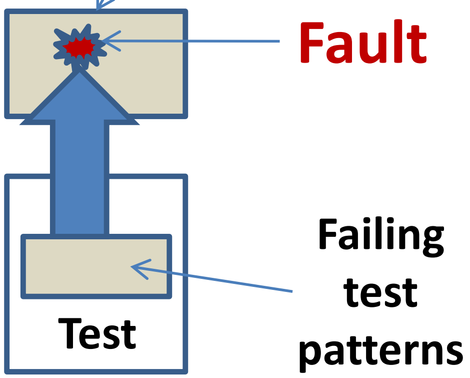


2) Effect-Cause Fault Diagnosis

Faulty block is located in the suspected faulty area

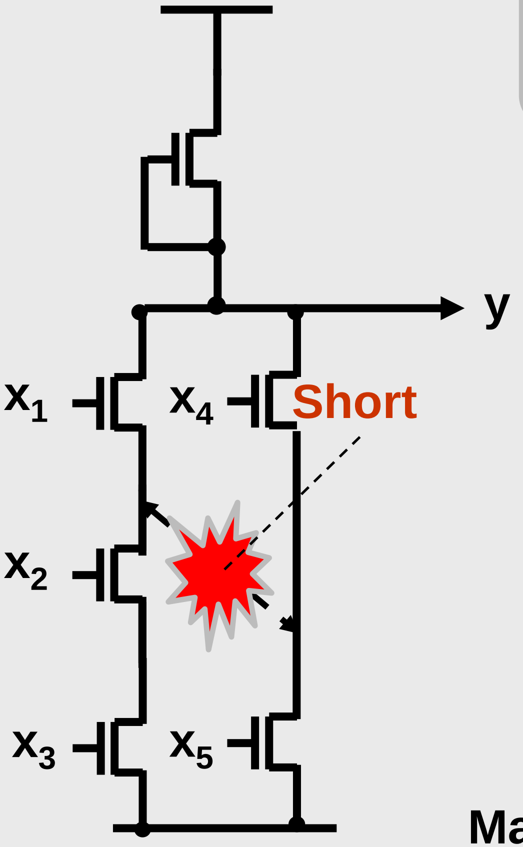
3) Fault Reasoning

Failing test patterns are mapped into the suspected defect or into a set of suspected defects in the faulty block



Practical Use of Boolean Differences

A transistor fault causes a change in a logic function not representable by SAF model



Correct function: $y = x_1 x_2 x_3 \vee x_4 x_5$

Faulty function: $y^d = (x_1 \vee x_4)(x_2 x_3 \vee x_5)$

Defect variable: $d = \begin{cases} 0 \text{ – defect } d \text{ is missing} \\ 1 \text{ – defect } d \text{ is present} \end{cases}$

Generic function with defect:

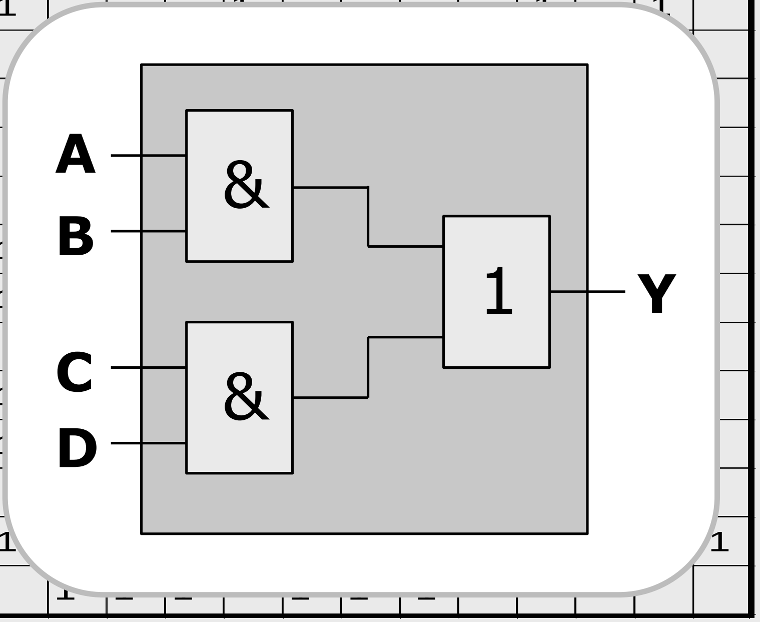
$$y^* = (y \wedge \overline{d}) \vee (y^d \wedge d)$$

Mapping the physical defect onto the logic level by solving the equation:

$$\frac{\partial y^*}{\partial d} = 1$$

Fault Table: Mapping Defects to Faults

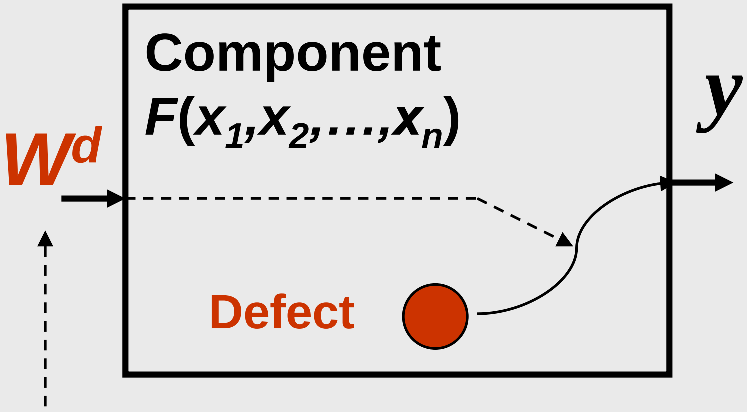
i	Fault d_i	Erroneous function f^{di}	p_i	Input patterns t_j															
				0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	B/C	not((B*C)*(A+D))	0.010307065				1								1	1	1		
2	B/D	not((B*D)*(A+C))	0.000858922				1								1	1		1	
3	B/N9	B*(not(A))	0.043375564	1	1	1					1	1	1	1					
4	B/Q	B*(not(C*D))	0.007515568	1	1	1						1	1	1		1	1	1	
5	B/VDD	not(A+(C*D))	0.001717844									1	1	1					
6	B/VSS	not(C*D)	0.035645265													1	1	1	
7	A/C	not((A*C)*(B+D))	0.098990767				1				1					1	1		
8	A/D	not((A*D)*(B+C))	0.013098561				1				1					1		1	
9	A/N9	A*(not(B))	0.038651492	1	1	1													
10	A/Q	A*(not(C*D))	0.025982392	1	1	1													
11	A/VDD	not(B+(C*D))	0.000214731																
12	C/N9	not(A+B+D)+(C*(not((A*B)+D)))	0.020399399		1														
13	C/Q	C*(not(A*B))	0.033927421	1	1		1												
14	C/VSS	not(A*B)	0.005153532				1												
15	D/N9	not(A+B+C)+(D*(not((A*B)+C)))	0.007730298			1													
16	D/Q	D*(not(A*B))	0.149452437	1		1	1												
17	N9/Q	not((A*B)+(B*C*D)+(A*C*D))	0.143654713				1												
18	N9/VDD	not((C*D)+(A*B*D)+(A*B*C))	0.253382006																
19	Q/VDD	SA1 at Q	0.014386944				1												1
20	Q/VSS	SA0 at Q	0.095555078	1	1	1		1	1	1		1	1	1					



Generalization: Functional Fault Model

Conditional Stuck-at-Fault model
Constrained SAF

$$\boxed{W^d = \frac{\partial y^*}{\partial d} = 1}$$



Logical constraints

Fault model:

(dy,W^d) — SAF, Constraint

(dy,{W_k^d}) — SAF, All constraints for all defects

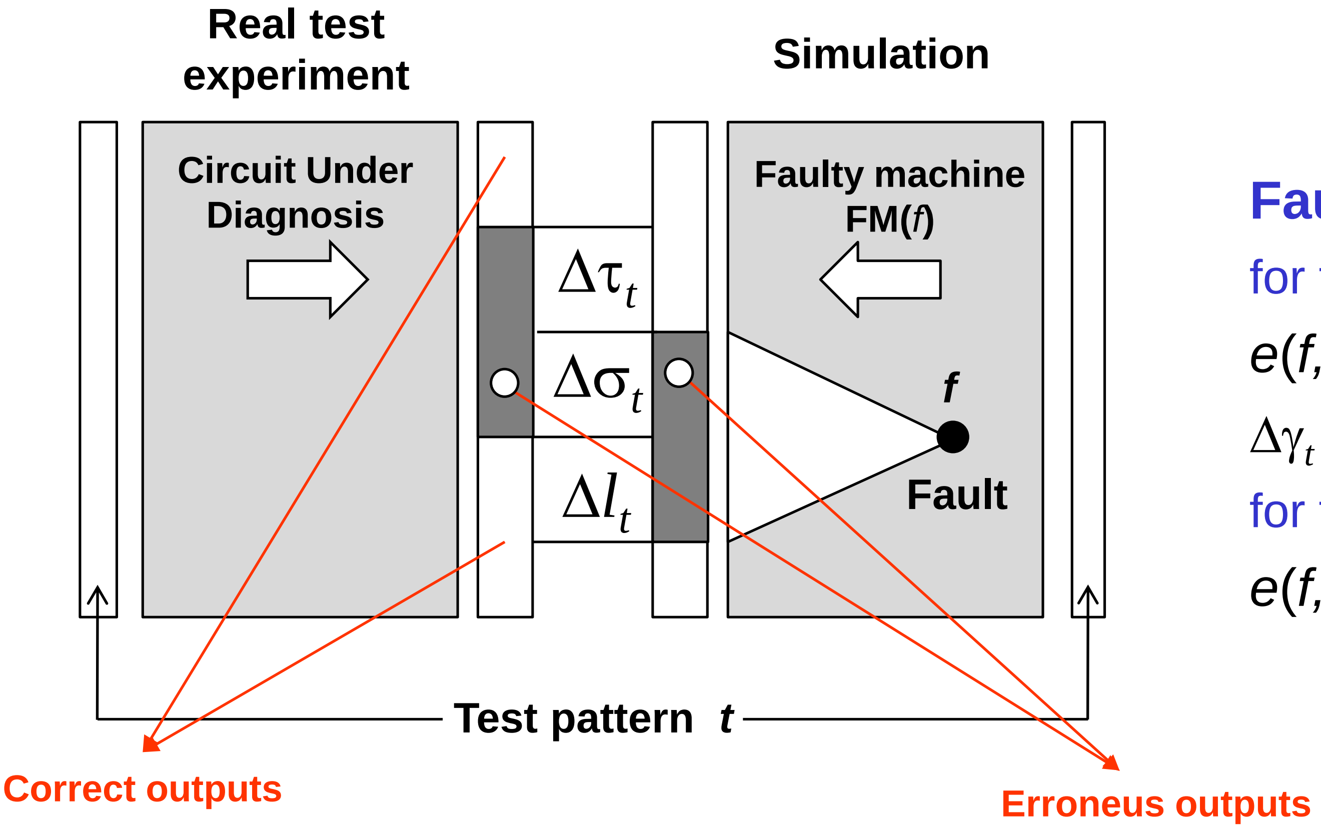
Constraints calculation:

Fault-free **Faulty**

$$y^* = F^*(x_1, x_2, \ldots, x_n, d) = \overline{d}F \vee dF^d$$

***d* = 1**, if the **defect** is present

Diagnosis of Fault Model Free Defects



Real test experiment

Simulation

Circuit Under Diagnosis

Faulty machine FM(*f*)

$\Delta\tau_t$

$\Delta\sigma_t$

Δl_t

f

Fault

Test pattern *t*

Correct outputs

Erroneus outputs

Fault evidence:

for test pattern ***t***

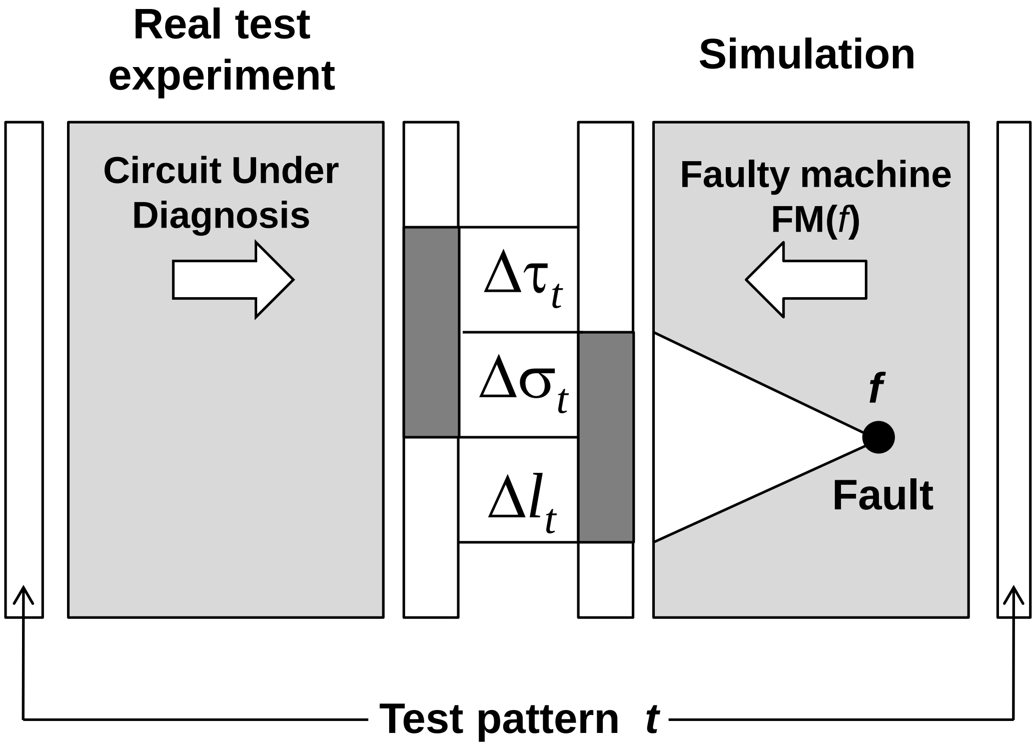
$e(f,t) = (\Delta\tau_t\ , \Delta\sigma_t, \Delta l_t, \Delta\gamma_t)$

$\Delta\gamma_t = \mathbf{min}\ (\Delta\sigma_t, \Delta l_t)$

for full test ***T*** (sum)

$e(f,T) = (\Delta\tau\ , \Delta\sigma, \Delta l, \Delta\gamma)$

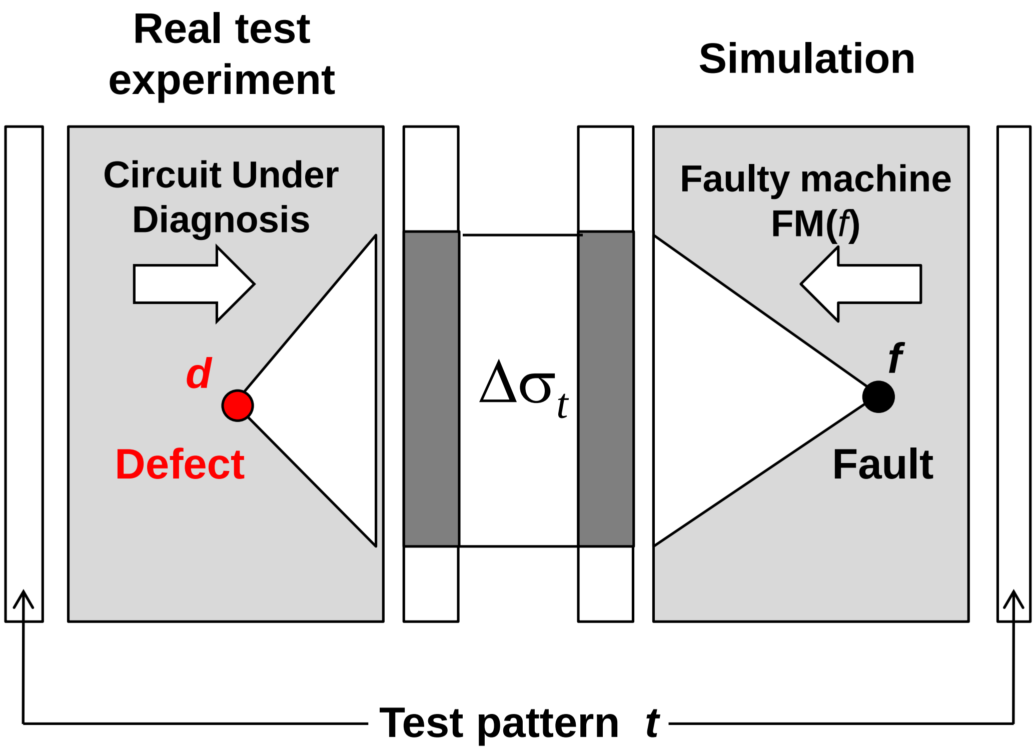
Diagnosis of Fault Model Free Defects



Different classical fault cases

Classic model	l_t	τ_t	γ_t
Single SAF	0	0	0
Multiple SAF	0	>0	0
Single conditional SAF	>0	0	0
Multiple cond. SAF	>0	>0	0
Delay fault	>0	0	>0
General case	>0	>0	>0

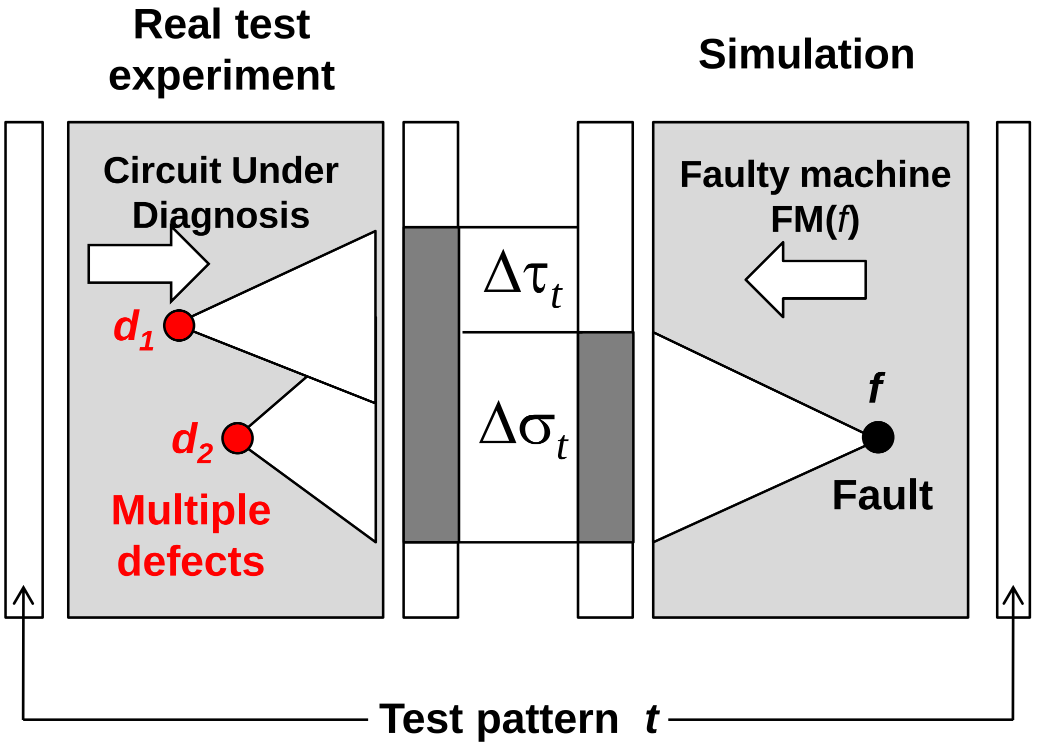
Diagnosis of Fault Model Free Defects



Different classical fault cases

Classic model	I_t	τ_t	γ_t
Single SAF	**0**	**0**	**0**
Multiple SAF	0	>0	0
Single conditional SAF	>0	0	0
Multiple cond. SAF	>0	>0	0
Delay fault	>0	0	>0
General case	>0	>0	>0

Diagnosis of Fault Model Free Defects

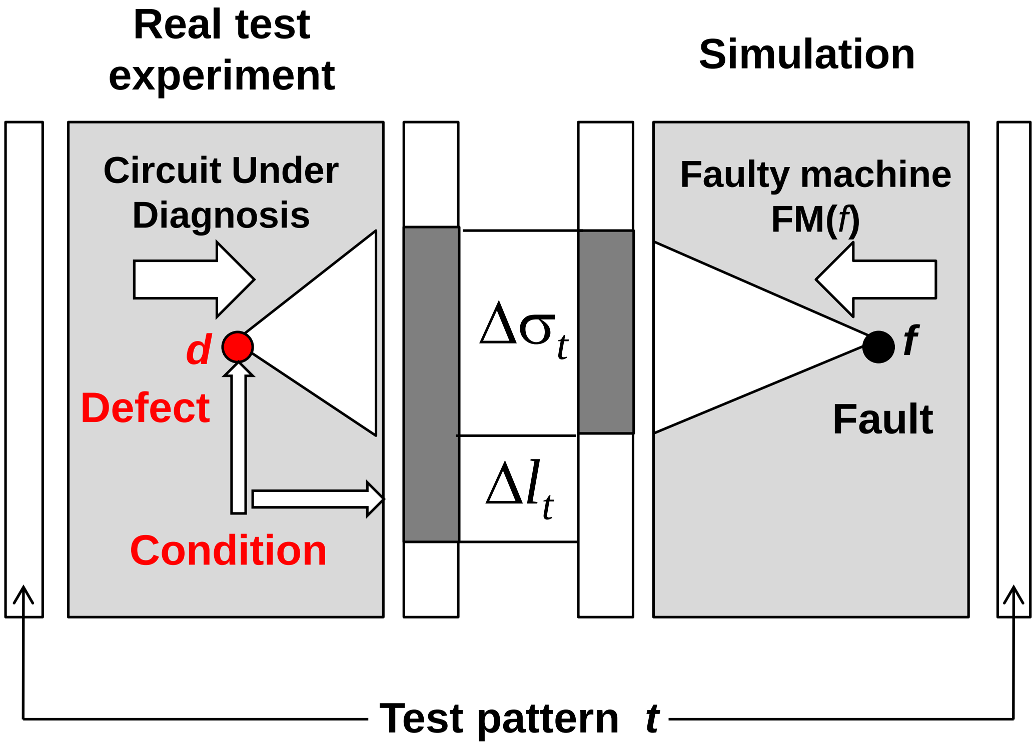


Different classical fault cases

Classic model	I_t	τ_t	γ_t
Single SAF	0	0	0
Multiple SAF (defects)	**0**	**>0**	**0**
Single conditional SAF	>0	0	0
Multiple cond. SAF	>0	>0	0
Delay fault	>0	0	>0
General case	>0	>0	>0

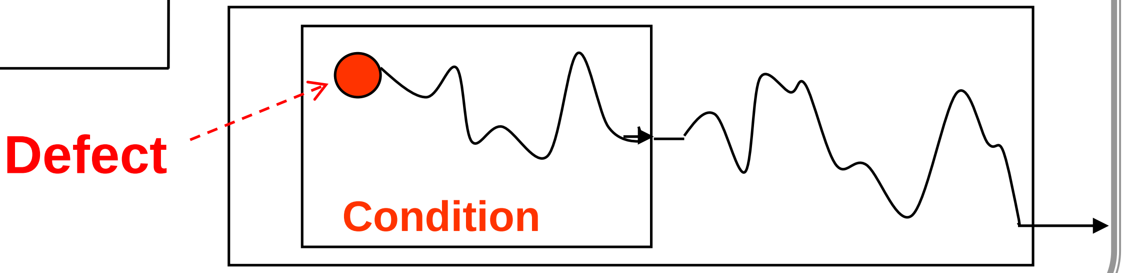
Copyright: H.J.Wunderlich 2007

Diagnosis of Fault Model Free Defects



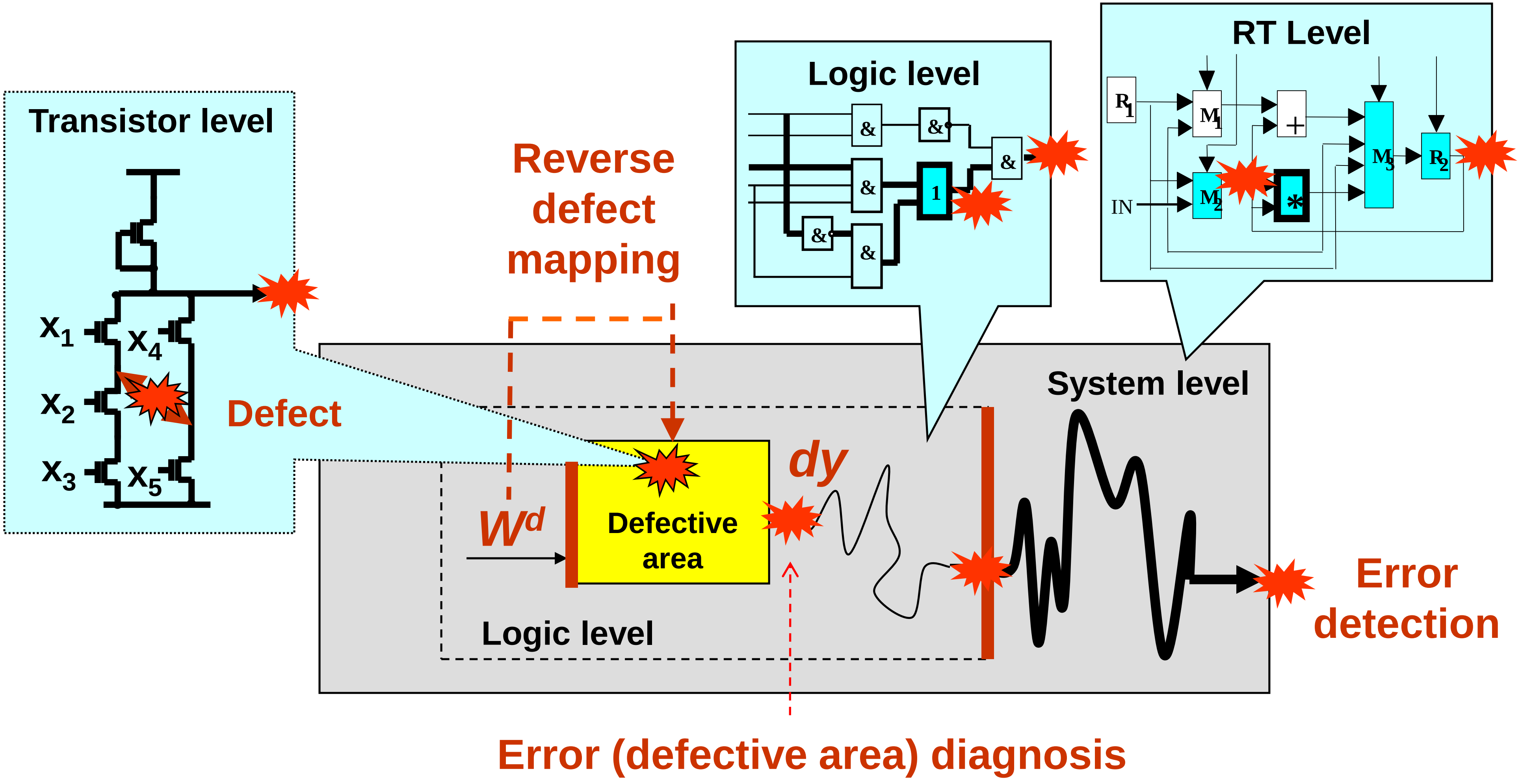
Different classical fault cases

Classic model	I_t	τ_t	γ_t
Single SAF	0	0	0
Multiple SAF	0	>0	0
Single conditional SAF	**>0**	**0**	**0**
Multiple cond. SAF	>0	>0	0
Delay fault	>0	0	>0
General case	>0	>0	>0



Copyright: H.J.Wunderlich 2007

Fault Diagnosis Without Fault Models



Transistor level

x_1 x_4 x_2 x_3 x_5

Defect

Reverse defect mapping

Logic level

RT Level

System level

W^d

Defective area

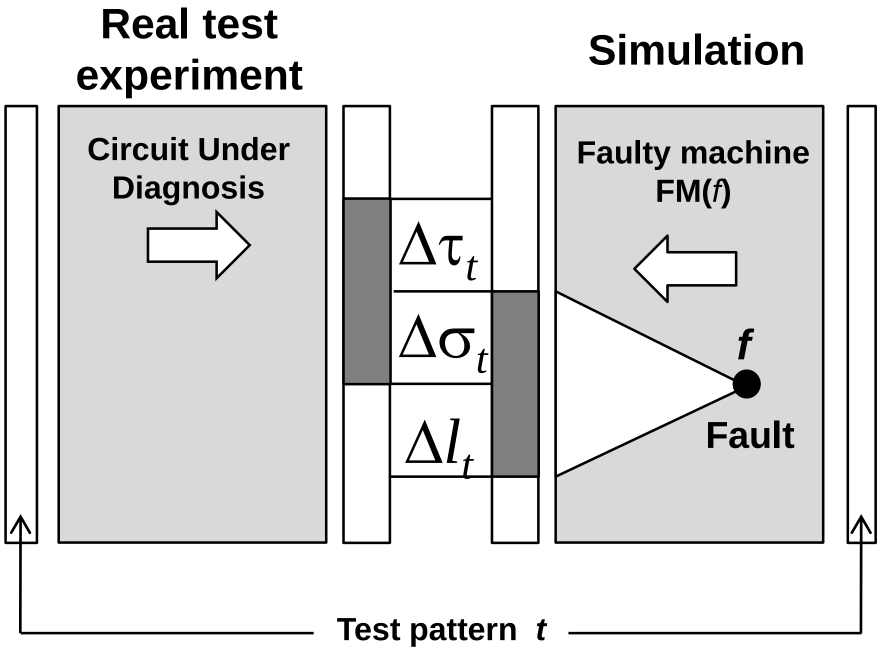
dy

Logic level

Error detection

Error (defective area) diagnosis

Diagnosis of Fault Model Free Defects



Ranking

(on the top the most suspicious faults):

(1) By increasing γ_T

(single SAF on top)

(2) If γ_T are equal then by decreasing σ_T

(3) If γ_T and σ_T are equal then by increasing l_T

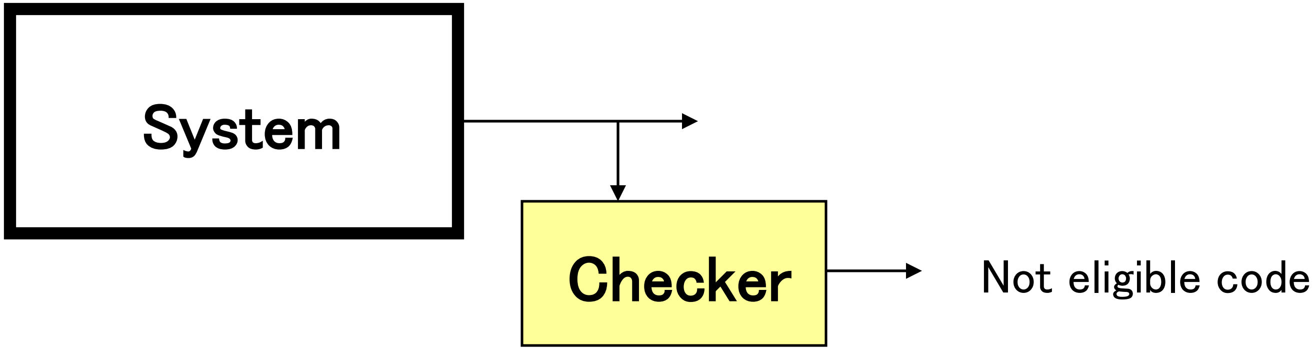
$$\Delta\gamma_t = \min(\Delta\sigma_t, \Delta l_t)$$

Example:

SAF	γ_T	σ_T	l_T
f_1	0	**42**	0
f_2	**30**	**42**	**15**
f_3	**30**	**42**	**25**
f_4	**30**	**42**	**30**
f_5	**30**	**36**	38
f_6	38	23	22
f_7	38	23	23

Copyright: H.J.Wunderlich 2007

Fault Tolerance: Error Detecting Codes



Examples:

Decimal digits:

Eligible: 0,1,2,..., 9

Not eligible: 10,11,..., 15

Parity check:

Parity bit

00	**0**	0	**1**
01	**1**	3	**2**
10	**1**	5	**4**
11	**0**	6	**7**

Eligible

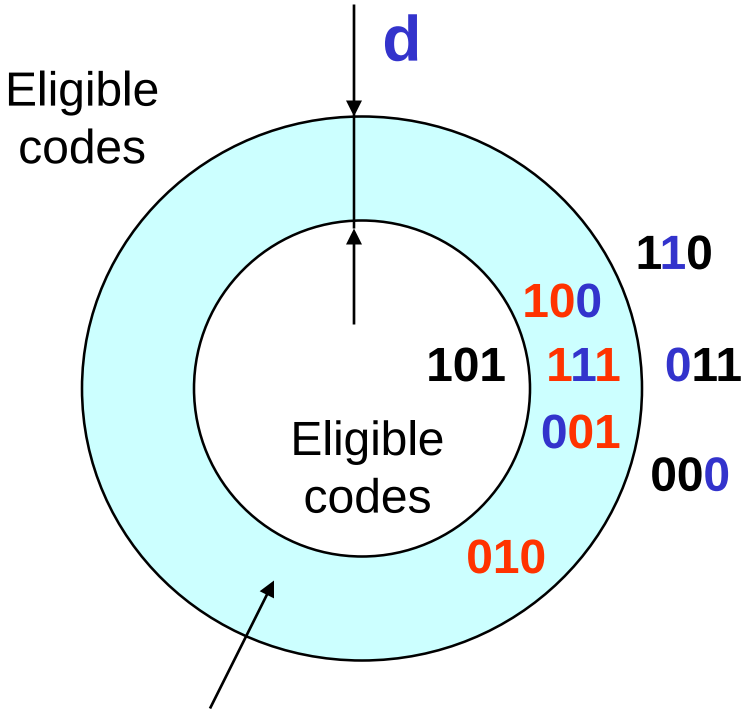
Not

eligible

Error Detecting/Correcting Codes

Hamming distance between codes:

Minimal number of bits how two codes differ from each other



Not eligible codes

Parity check:

d = 2

Parity bit

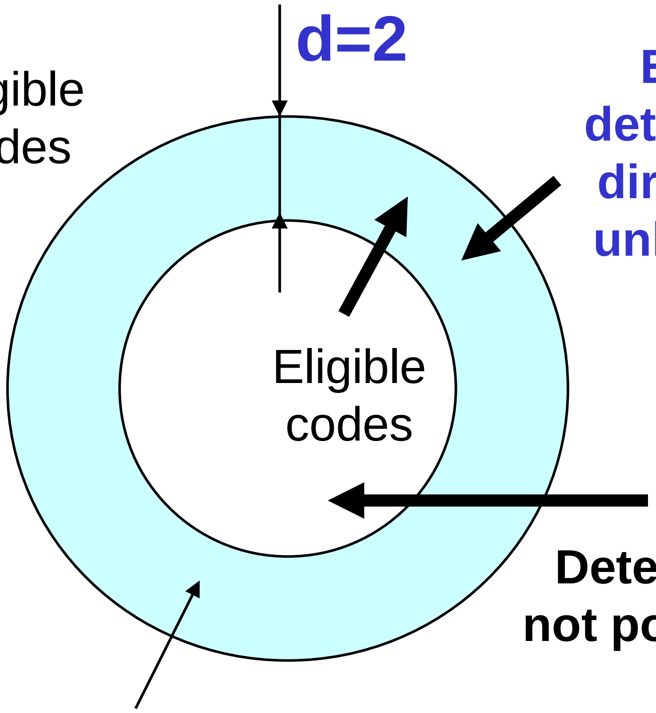
00	0	0	1
01	1	3	2
10	1	5	4
11	0	6	7

Eligible

Not eligible

Error Detecting/Correcting Codes

Error detecting codes:



d=2

Eligible codes

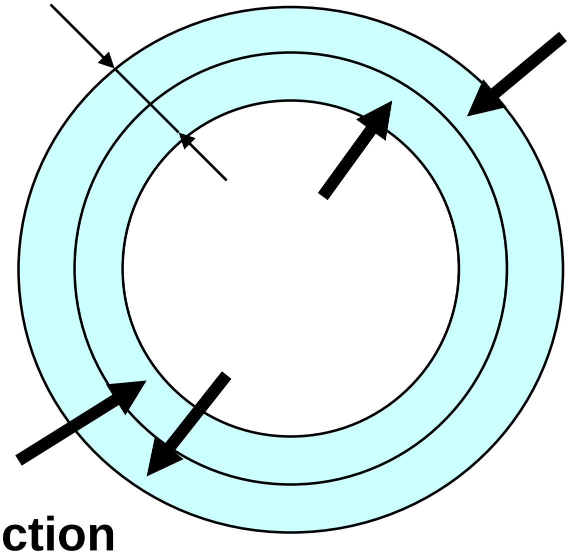
Error detection: direction unknown

Eligible codes

Detection not possible

Not eligible codes

Error correcting codes:



Error correction is possible: direction is known

d=3

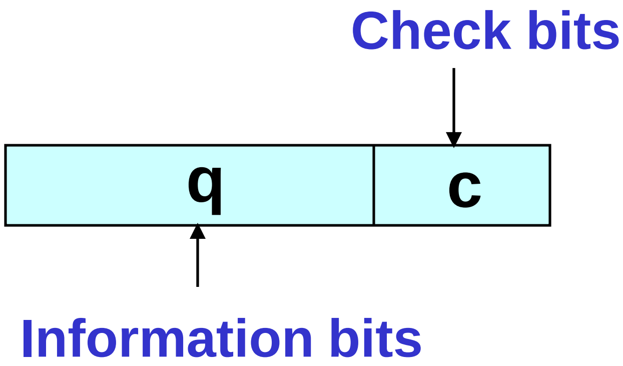
Correction not possible

Fault Tolerance: Error Correcting Codes

$d = 2e + 1$ - $2e$ - error detection

e - error correction

One error correction code: $2^c \geq \underbrace{q + c}_{\text{For addressing of the erroneous bit}} + \underbrace{1}_{\text{Error free}}$



Check bits

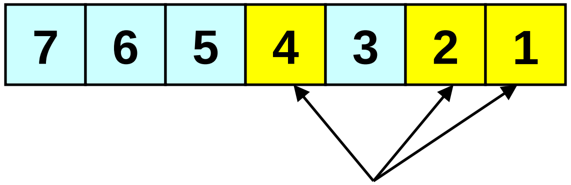
q

c

Information bits

Fault Tolerance: One Error Correcting Code

Location of erroneous bit:



Check bits

b_{2^i}, i = 1,...,c

$P_1 = b_1 \oplus b_3 \oplus b_5 \oplus b_7 = 0$

$P_2 = b_2 \oplus b_3 \oplus b_6 \oplus b_7 = 0$

$P_3 = b_4 \oplus b_5 \oplus b_6 \oplus b_7 = 0$

Check bits have to be independently assigned

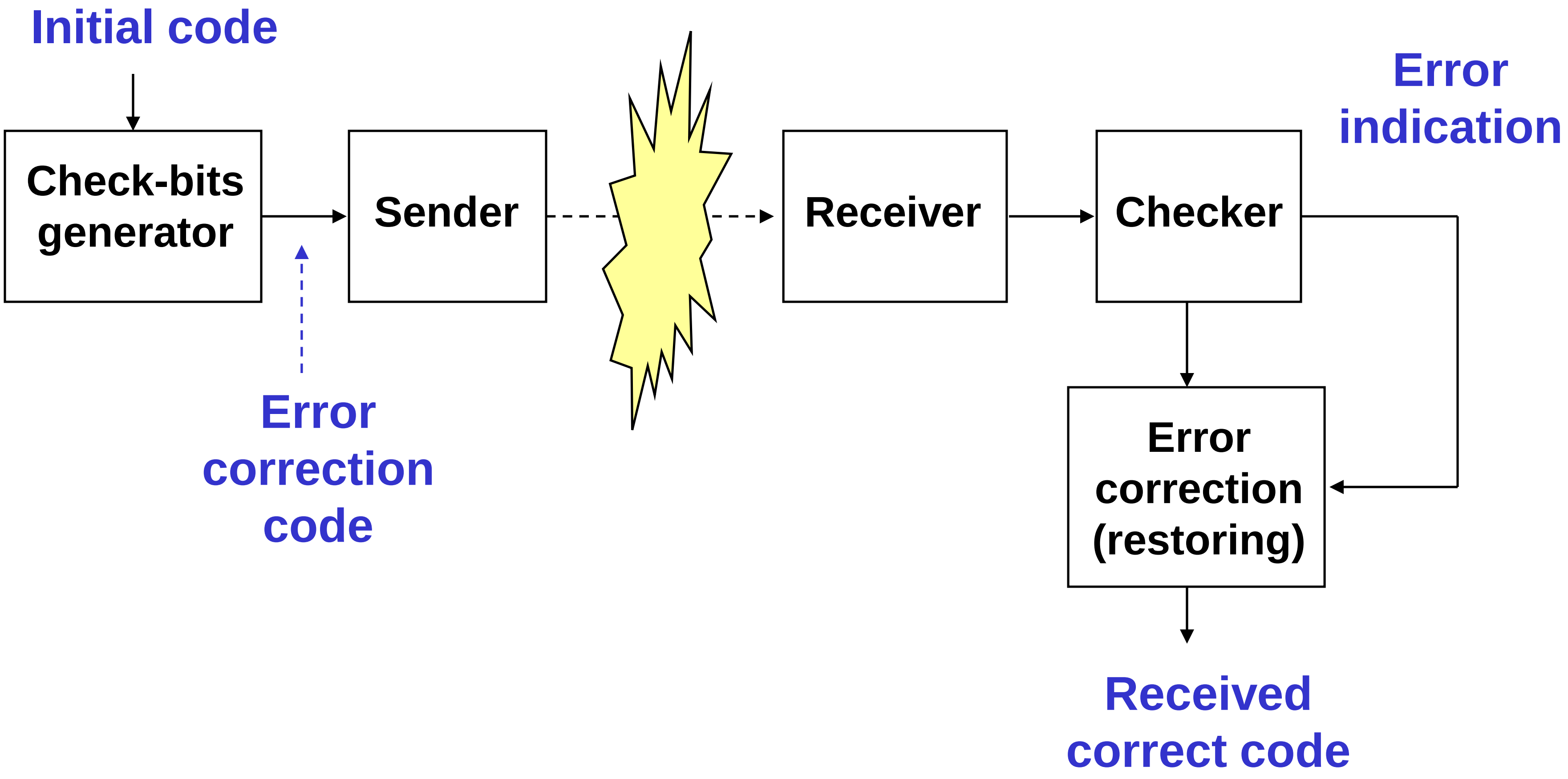
Analogy with fault diagnosis
by using fault table:

	7	6	5	4	3	2	1
Initial code	1	0	1	0	1	0	1
Received code	1	0	1	1	1	0	1

	7	6	5	4	3	2	1	0	Test
P_1	1		1		1		1		0
P_2	1	1			1	1			0
P_3	1	1	1	1					1

Diagnosis

Fault Tolerant Communication System



Error Detection in Arithmetic Operations

Residue codes

N – information code

$C = (N) \bmod m$ - check code

m – residue of the code

$p = \lceil \log_2 m \rceil$ – number of check bits

Example

Information bits: I_2, I_1, I_0

$m = 3, p = 2$

Check bits: c_1, c_0

Information bits					Check bits	
I_2	I_1	I_0	I	c	c_1	c_0
0	0	0	0	0	0	0
0	0	1	1	1	0	1
0	1	0	2	2	1	0
0	1	1	3	0	0	0
1	0	0	4	1	0	1
1	0	1	5	2	1	0
1	1	0	6	0	0	0
1	1	1	7	1	0	1

Error Detection in Arithmetic Operations

Addition:

Information bits				Check bits		
0	0	1	0	1	0	2.2
0	1	0	0	0	1	4.1
0	1	1	0	1	1	6.3

(6)mod3 = 0 (3)mod3 = 0

Multiplication:

Information bits				Check bits		
0	0	1	0	1	0	2.2
0	1	0	0	0	1	4.1
1	0	0	0	1	0	8.2

(8)mod3 = 2 (2)mod3 = 2

Information bits				Check bits		
0	0	1	0	1	0	2.2
0	1	0	0	0	1	4.1
0	1	0	0	1	1	4.3

(4)mod3 = 1 (3)mod3 = 0

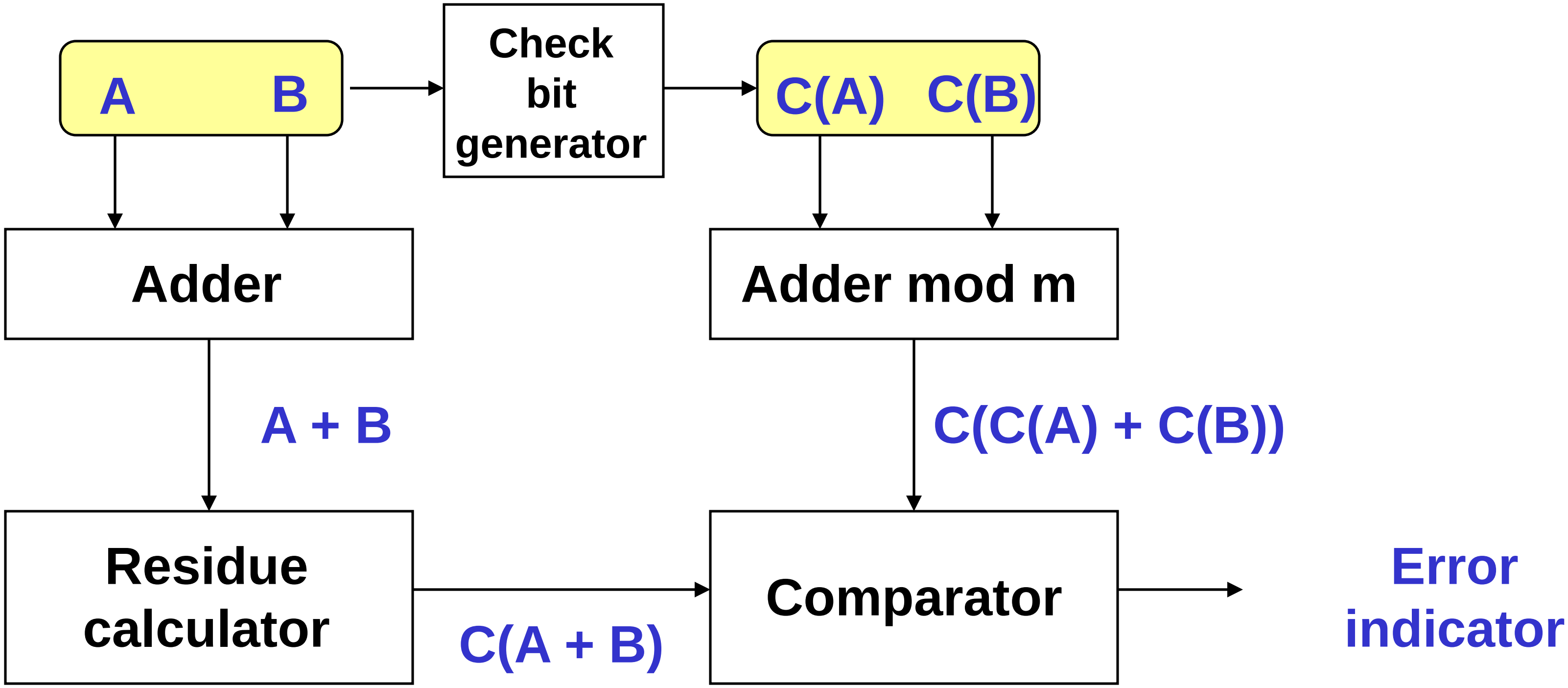
Error!

Information bits				Check bits		
0	0	1	0	1	0	2.2
0	1	0	0	0	1	4.1
1	0	0	1	1	0	9.2

(9)mod3 = 0 (2)mod3 = 2

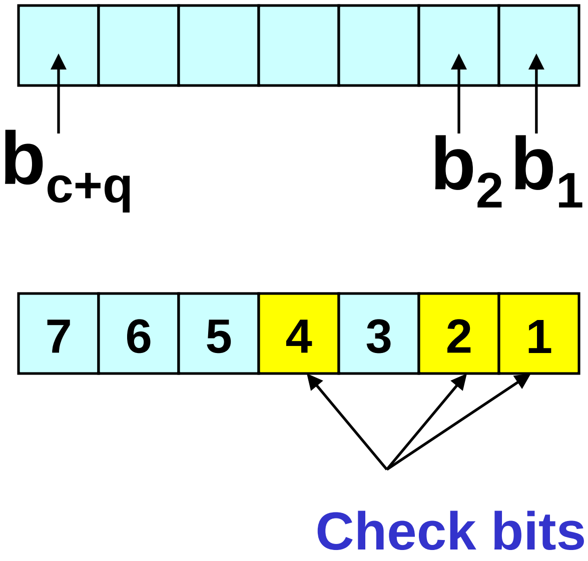
Error!

Error Detection in Arithmetic Operations



Fault Tolerance: One Error Correcting Code

One error correction code: $2^c \geq q + c + 1$



Calculation of check sums:

$$\sum_{k \in P_i} b_k = 0, i = 1, \ldots, c$$

Parity bits for c = 3:

$$P_1 = b_1 \oplus b_3 \oplus b_5 \oplus b_7 = 0$$
$$P_2 = b_2 \oplus b_3 \oplus b_6 \oplus b_7 = 0$$
$$P_3 = b_4 \oplus b_5 \oplus b_6 \oplus b_7 = 0$$

Theory of LFSR

Characteristic Polynomials:

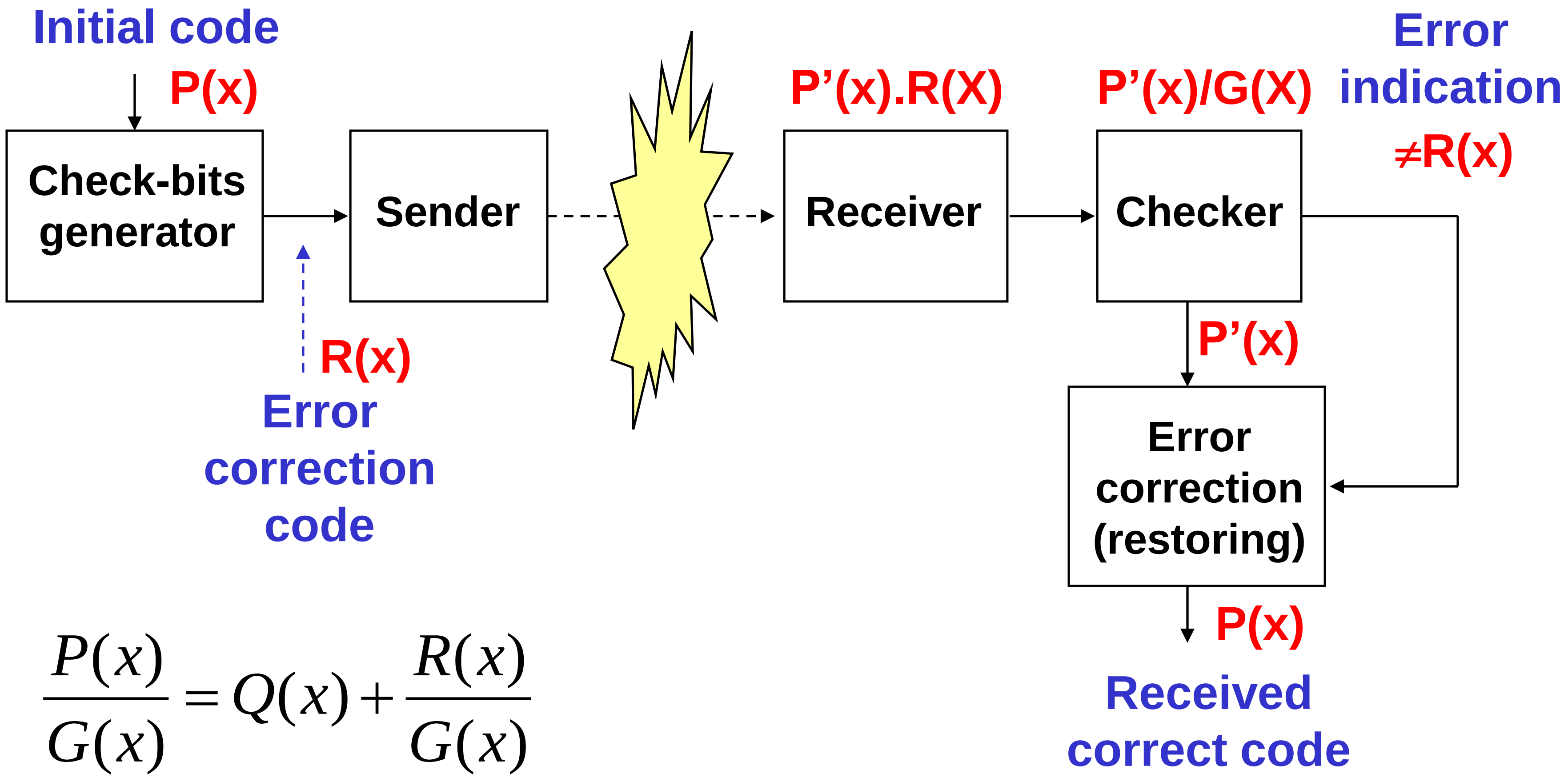
$$G(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_m x^m + \ldots = \sum_{m=0}^{\infty} c_m x^m$$

Multiplication of

polynomials

$$\begin{array}{rrrrr}
 & & x^2 & +x & +1 \\
 & & x^2 & & +1 \\
\cline{3-5}
 & & x^2 & +x & +1 \\
x^4 & +x^3 & +x^2 & & \\
\hline
x^4 & +x^3 & & +x & +1
\end{array}$$

Fault Tolerant Communication System



$$\frac{P(x)}{G(x)} = Q(x) + \frac{R(x)}{G(x)}$$