Overview

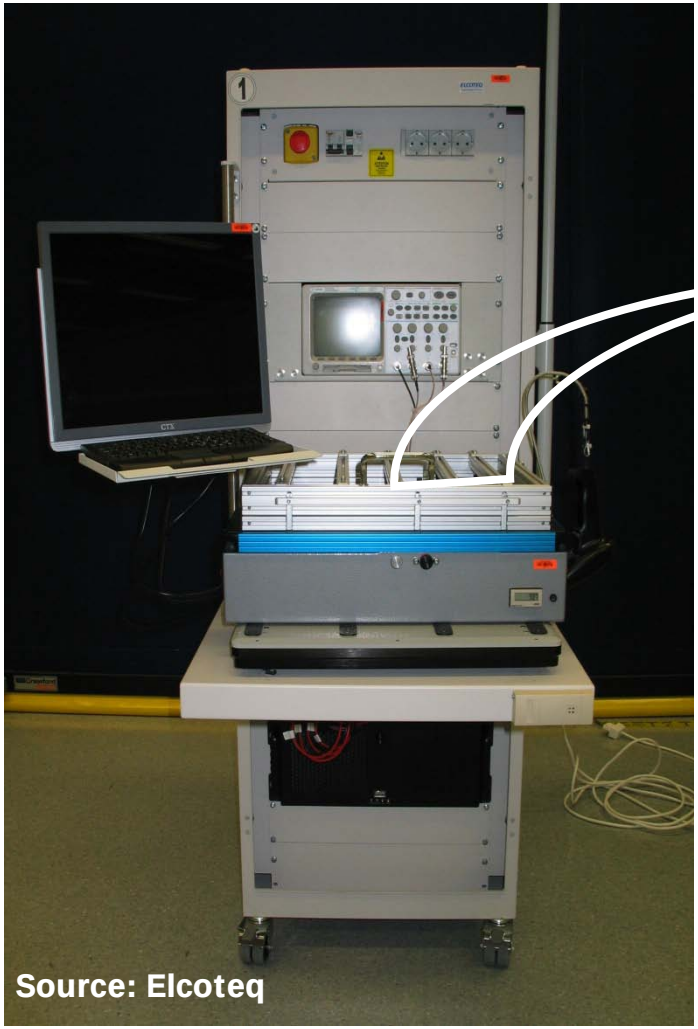
1. Introduction
2. Testability measuring
3. Design for testability
4. **Built in Self-Test**

Built-In Self-Test

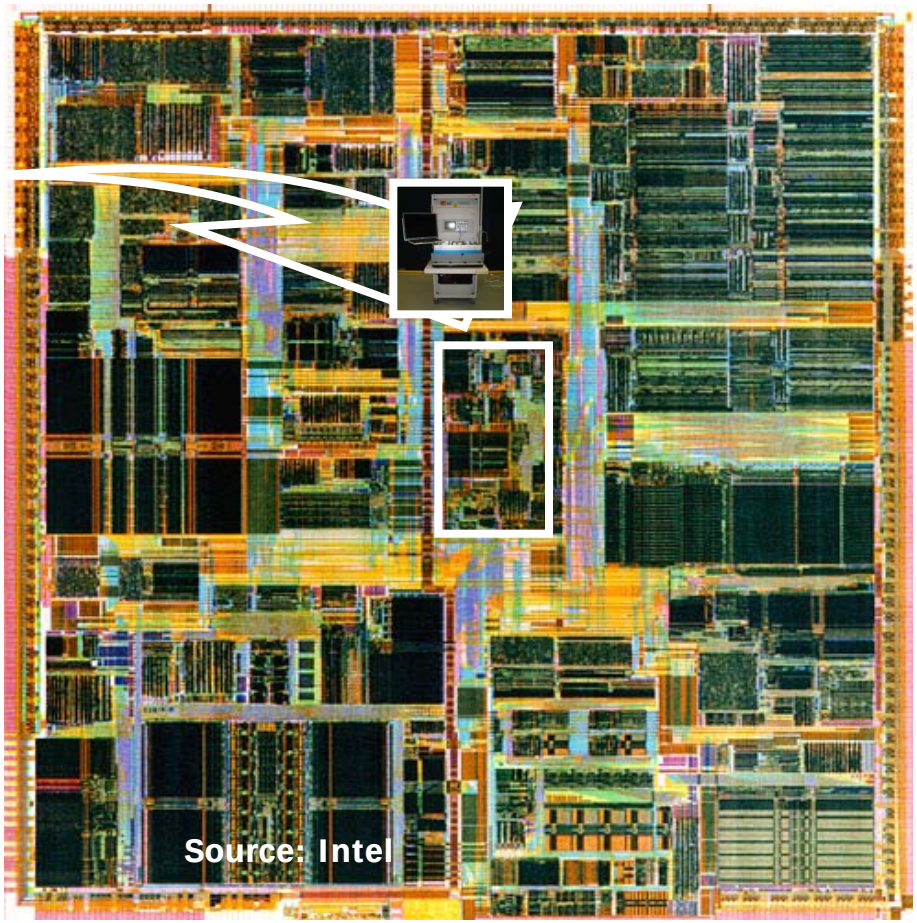
Outline

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

Testing Challenges: SoC Test

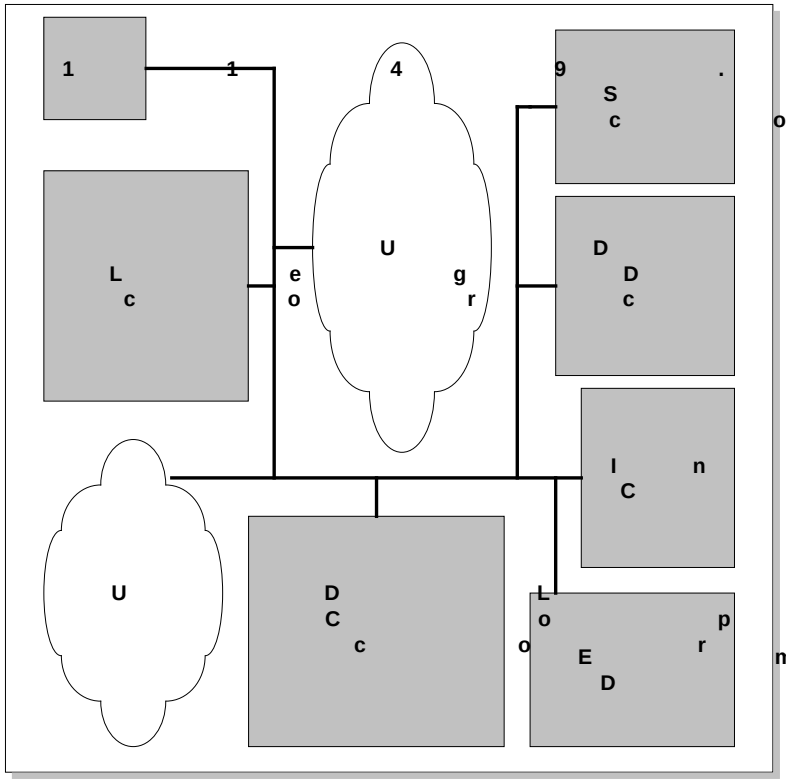


Cores have to be tested on chip



Built-In Self-Test

System-on-Chip

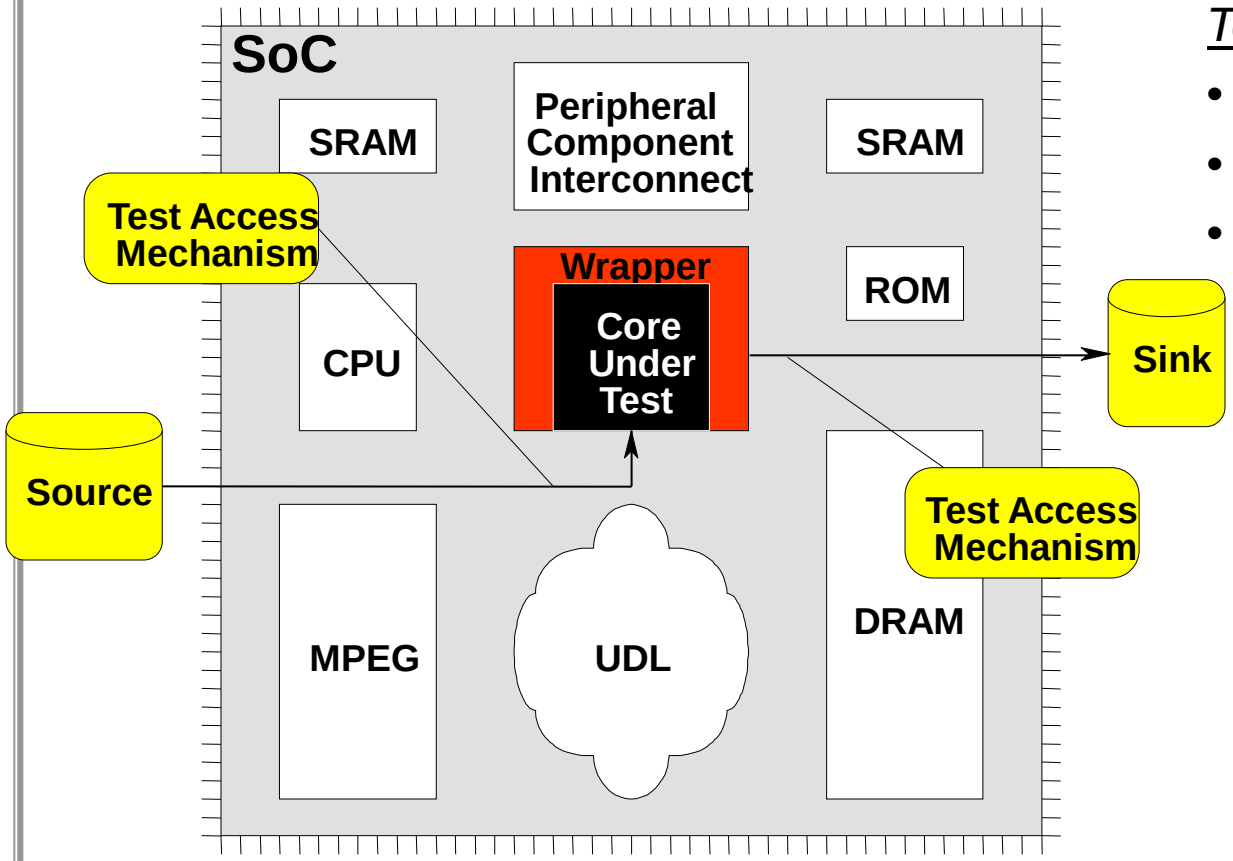


- Advances in microelectronics technology have introduced a new paradigm in IC design: System-on-Chip (SoC)
- Many systems are nowadays designed by embedding predesigned and preverified complex functional blocks (cores) into one single die
- Such a design style allows designers to reuse previous designs and will lead to shorter time-to-market and reduced cost

SoC structure breakdown:

- 10% UDL
- 75% memory
- 50% in-house cores
- 60-70% soft cores

Self-Test in Complex Digital Systems



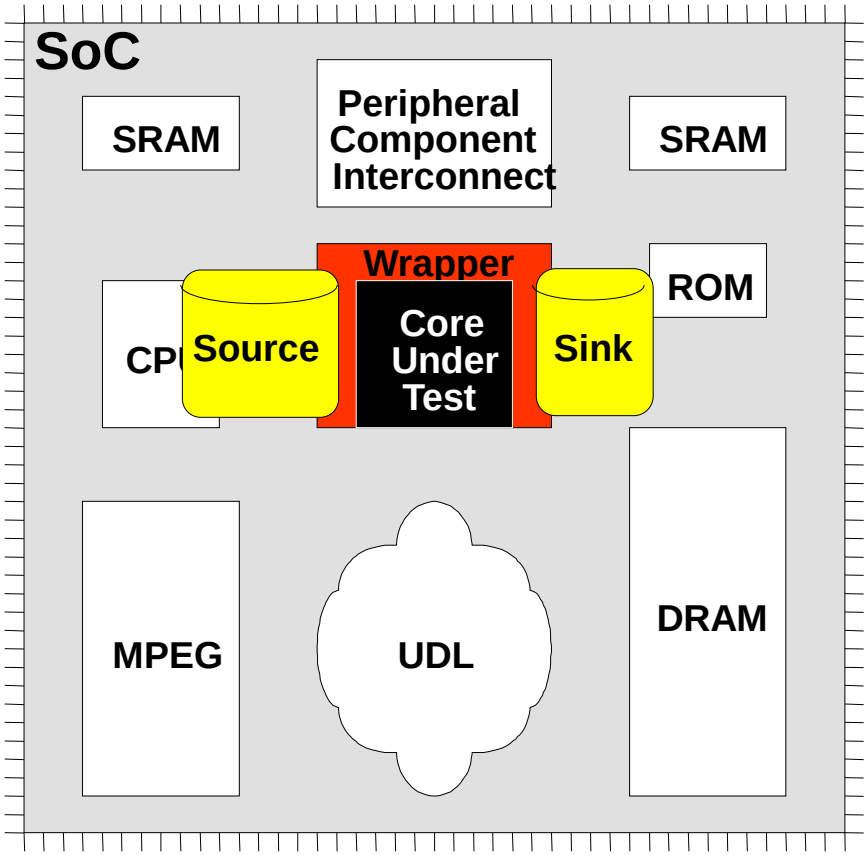
Test architecture components:

- **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

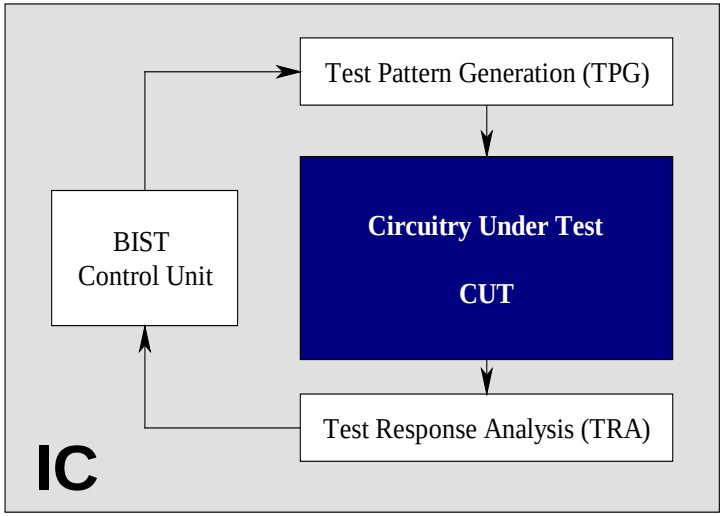
Solutions:

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

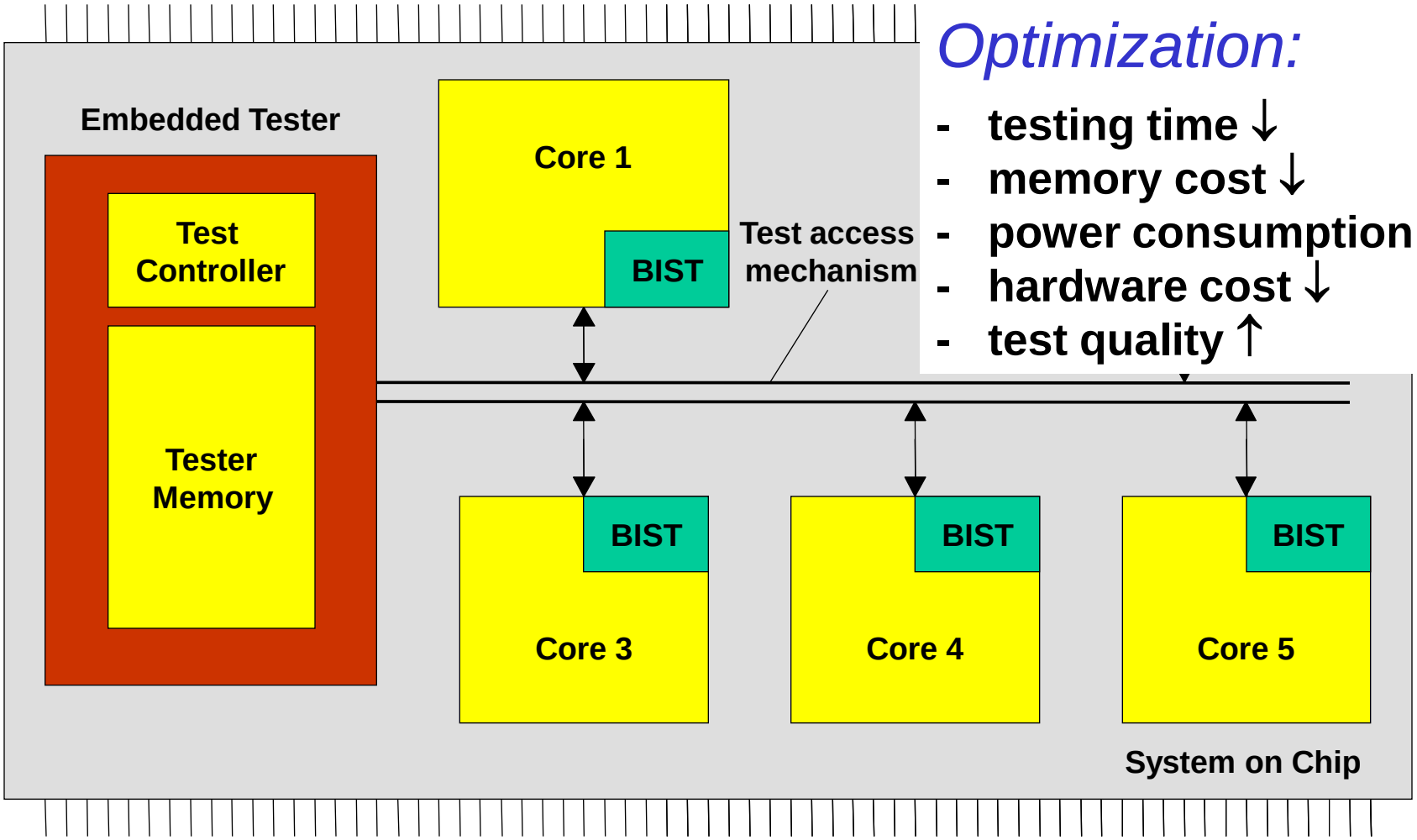
What is BIST

- **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**

Costly Test Problems Alleviated by BIST

- Increasing chip logic-to-pin ratio – harder observability
- Increasingly dense devices and faster clocks
- Increasing test generation and application times
- Increasing size of test vectors stored in ATE
- Expensive ATE needed for 1 GHz clocking chips
- Hard testability insertion – designers unfamiliar with gate-level logic, since they design at behavioral level
- *In-circuit testing* no longer technically feasible
- Shortage of test engineers
- Circuit testing cannot be easily partitioned

BIST in Maintenance and Repair

- **Useful for field test and diagnosis (less expensive than a local automatic test equipment)**
- **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**

Benefits and Costs of BIST with DFT

Level	Design and test	Fabri-cation	Manuf. Test	Maintenance test	Diagnosis and repair	Service interruption
Chips	+ / -	+	-			
Boards	+ / -	+	-		-	
System	+ / -	+	-	-	-	-

+ Cost increase
- Cost saving
+/- Cost increase may balance cost reduction

Economics – BIST Costs

- Chip area overhead for:
  - Test controller
  - Hardware pattern generator
  - Hardware response compacter
  - Testing of BIST hardware
- Pin overhead -- At least 1 pin needed to activate BIST operation
- Performance overhead – extra path delays due to BIST
- *Yield loss* – due to increased chip area or more chips In system because of BIST
- Reliability reduction – due to increased area
- Increased BIST hardware complexity – happens when BIST hardware is made testable

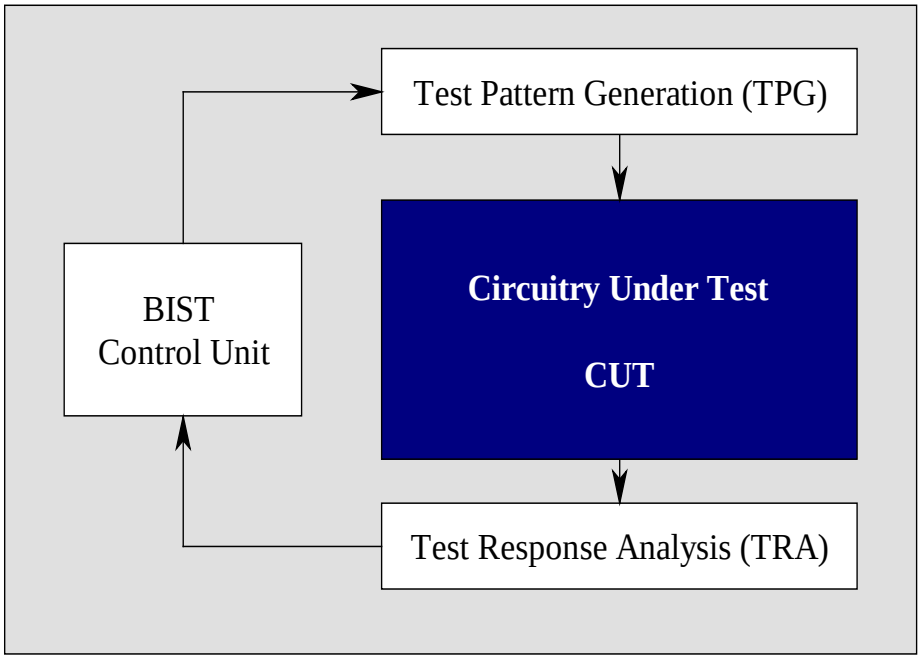
BIST Benefits

- **Faults tested:**
  - **Single stuck-at faults**
  - **Delay faults**
  - **Single stuck-at faults in BIST hardware**
- **BIST benefits**
  - **Reduced testing and maintenance cost**
  - **Lower test generation cost**
  - **Reduced storage / maintenance of test patterns**
  - **Simpler and less expensive ATE**
  - **Can test many units in parallel**
  - **Shorter test application times**
  - **Can test at functional system speed**

BIST Techniques

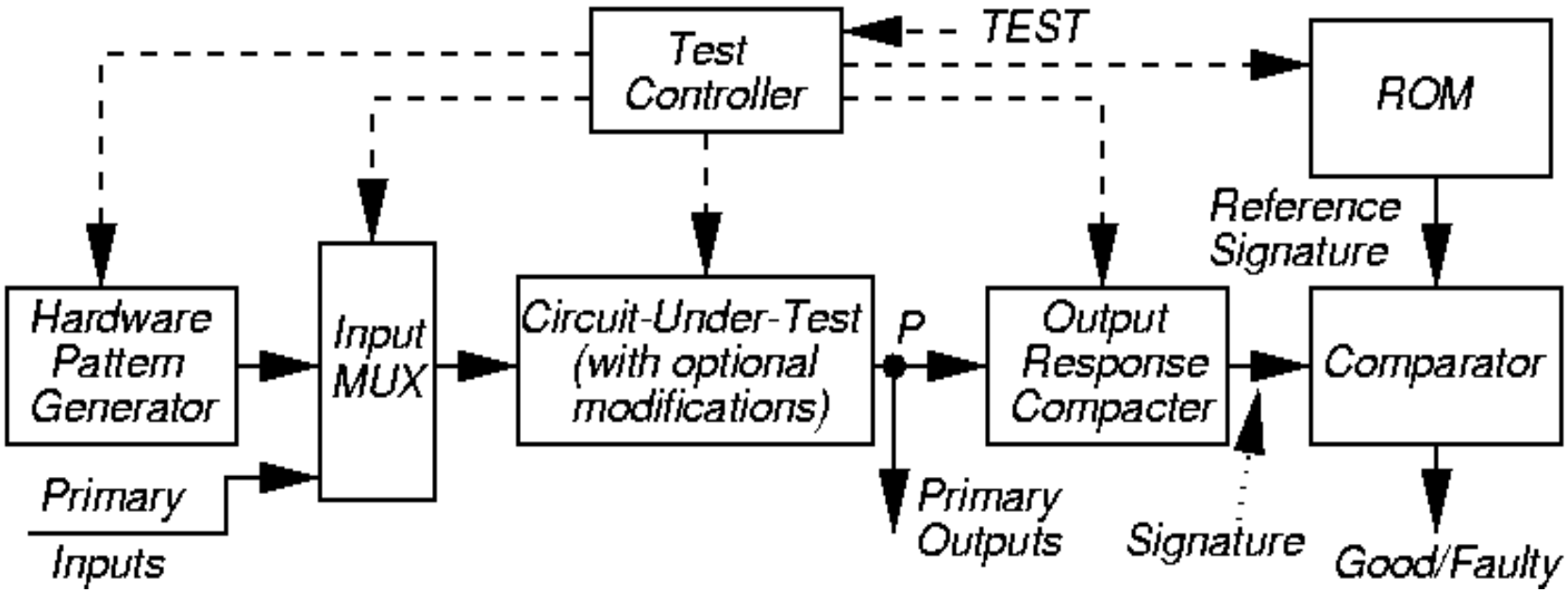
- **BIST techniques are classified:**
  - ***on-line* BIST** - includes concurrent and nonconcurrent techniques
  - ***off-line* BIST** - includes functional and structural approaches
- ***On-line* BIST** - testing occurs during normal functional operation
  - ***Concurrent on-line* BIST** - testing 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)

General Architecture of BIST

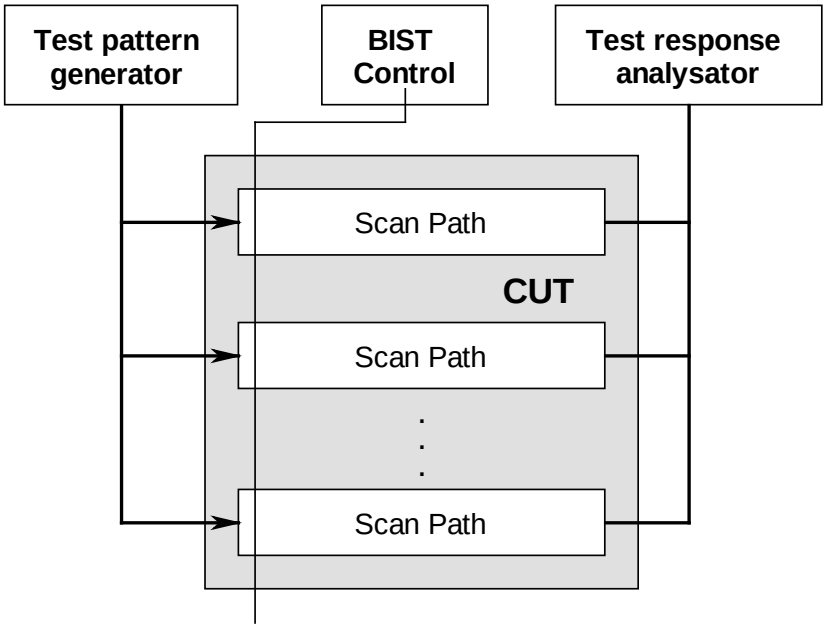


- **BIST components:**
  - **Test pattern generator (TPG)**
  - **Test response analyzer (TRA)**
- **TPG & TRA are usually implemented as linear feedback shift registers (LFSR)**
- **Two widespread schemes:**
  - **test-per-scan**
  - **test-per-clock**

Detailed BIST Architecture



Built-In Self-Test



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

Test per Scan:

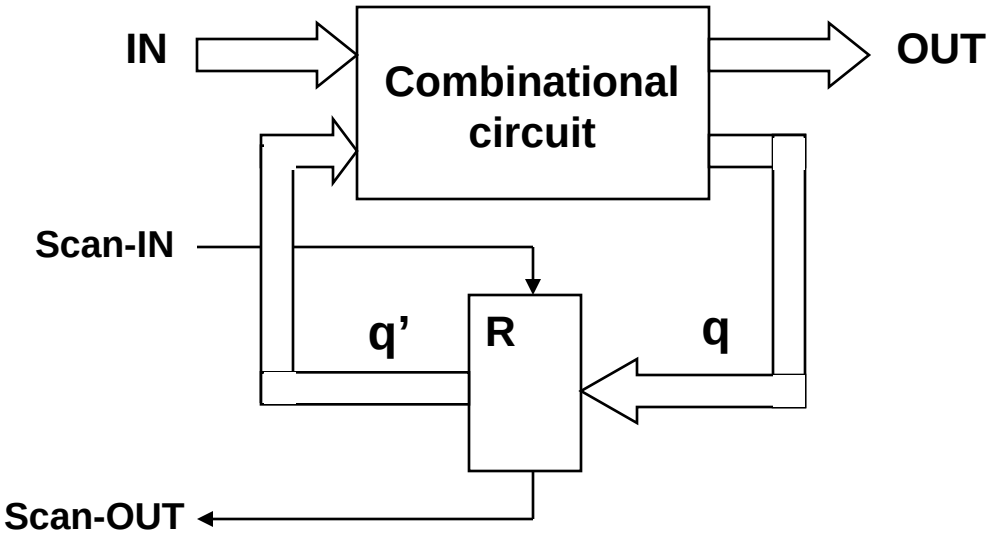
Initial test set:

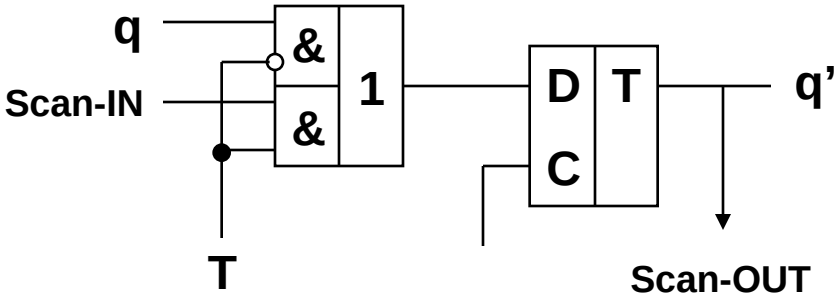
T1: 1100
T2: 1010
T3: 0101
T4: 1001

Test application:

1100 T 1010 T 0101T 1001 T
Number of clocks = 4 x 4 + 4 = 20

Scan-Path Design





The complexity of testing is a function of the number of feedback loops and their length

The longer a feedback loop, the more clock cycles are needed to initialize and sensitize patterns

Scan-register is a aregister with both shift and parallel-load capability

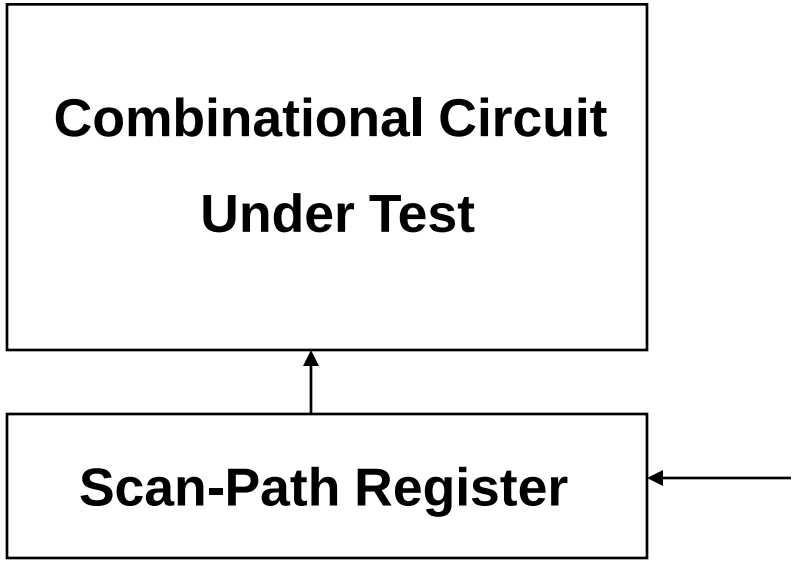
T = 0 - normal working mode T = 1 - scan mode

Normal mode : **flip-flops are connected to the combinational circuit**

Test mode: **flip-flops are disconnected from the combinational circuit and connected to each other to form a shift register**

Built-In Self-Test

Test per Clock:



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

- ***Test application:***
- 1 10 0 1 0 1 0 01 01 1001
- T_1 T_4 T_3 T_2
- Number of clocks = 10

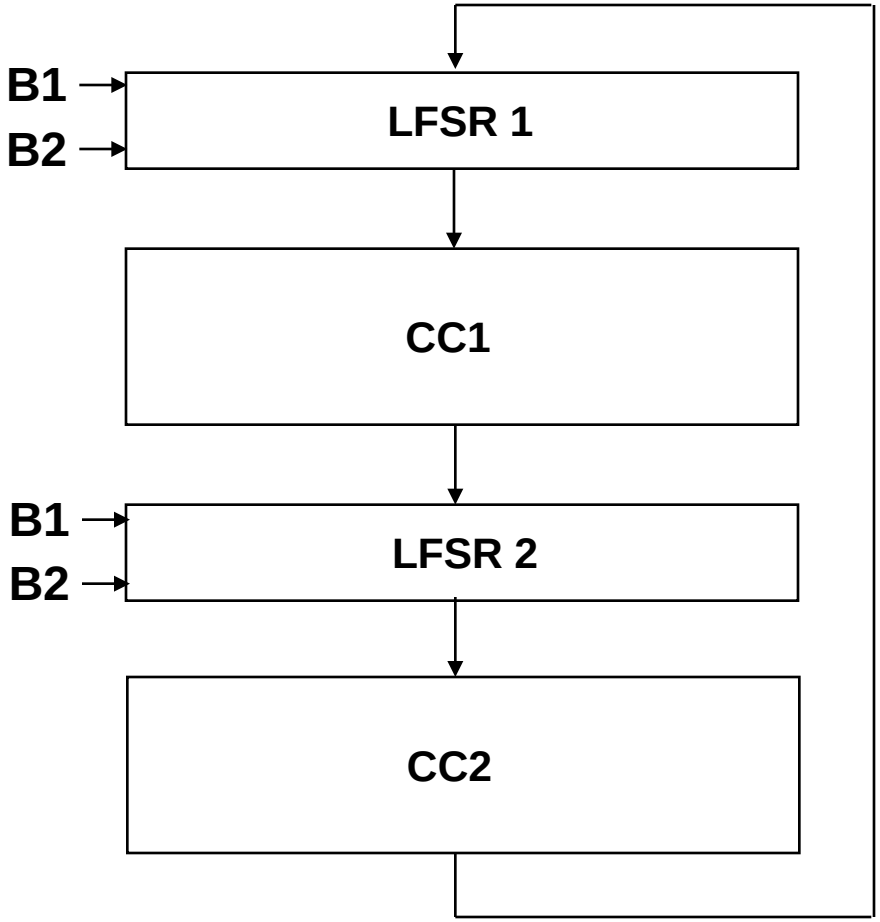
BILBO BIST Architecture

Working modes:

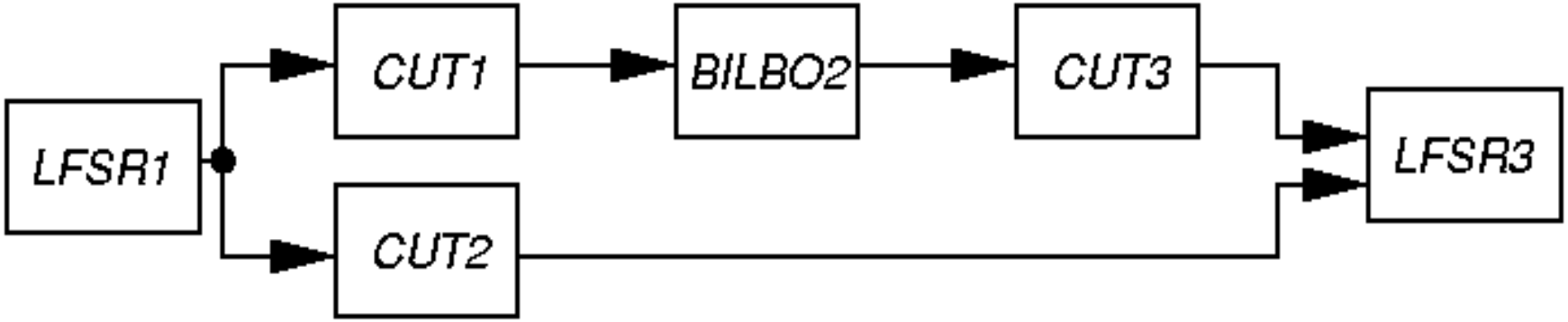
B1	B2	
0	0	Reset
0	1	Flip-flop (normal)
1	0	Scan mode
1	1	Test mode

Testing modes:

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



BILBO BIST Architecture: Example



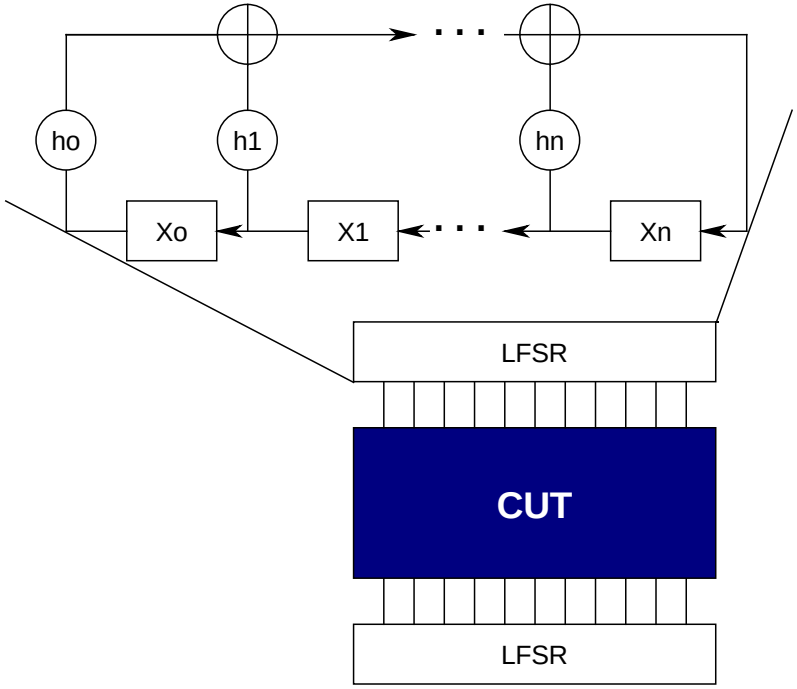
- **Testing epoch I:**
  - **LFSR1 generates tests for CUT1 and CUT2**
  - **BILBO2 (LFSR3) compacts CUT1 (CUT2)**
- **Testing epoch II:**
  - **BILBO2 generates test patterns for CUT3**
  - **LFSR3 compacts CUT3 response**

Pattern Generation

- **Store in ROM – too expensive**
- ***Exhaustive***
- ***Pseudo-exhaustive***
- ***Pseudo-random* (LFSR) – Preferred method**
- **Binary counters – use more hardware than LFSR**
- **Modified counters**
- **Test pattern *augmentation***
  - **LFSR combined with a few patterns in ROM**
  - ***Hardware diffracter* – generates pattern cluster in neighborhood of pattern stored in ROM**

Pattern Generation

Pseudorandom Test generation by LFSR:



- **Using special LFSR registers**
- **Several proposals:**
  - **BILBO**
  - **CSTP**
- **Main characteristics of LFSR:**
  - **polynomial**
  - **initial state**
  - **test length**

Some Definitions

- **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
- ***Exhaustive testing –*** Apply all possible 2^n patterns to a circuit with n inputs
- ***Pseudo-exhaustive testing –*** Break circuit into small, overlapping blocks 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

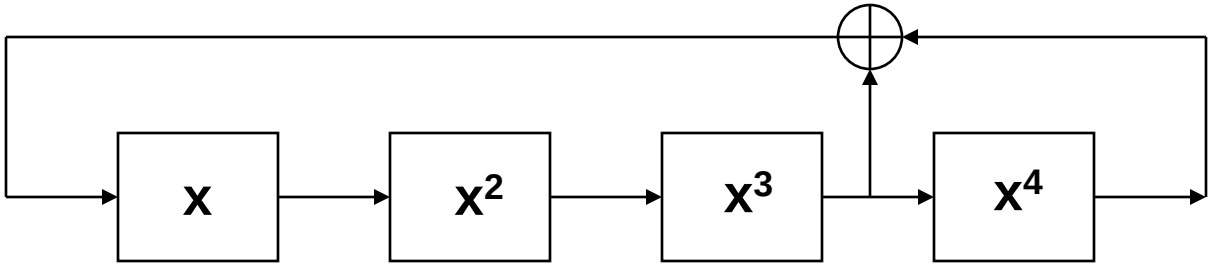
More Definitions

- ***Irreducible polynomial*** – Boolean polynomial that cannot be factored
- ***Primitive polynomial*** – Boolean polynomial *p(x)* that can be used to compute increasing powers *n* of x^n *modulo p(x)* to obtain all possible non-zero polynomials of degree less than *p(x)*
- ***Signature*** – Any statistical circuit property distinguishing between bad and good circuits
- **TPG** – Hardware *test pattern generator*
- **PRPG** – Hardware Pseudo-Random Pattern Generator
- **MISR** – Multiple Input Response Analyzer

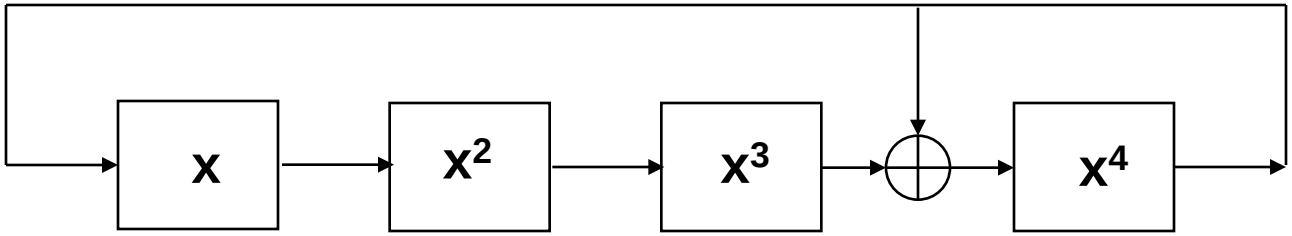
Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:

Standard LFSR

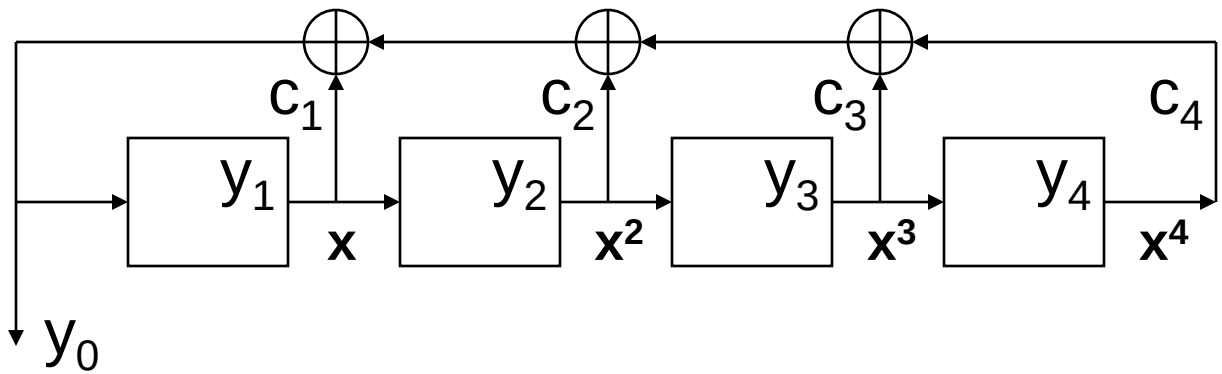


Modular LFSR



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

Theory of LFSR



$$y_j(t) = y_{j-1}(t-1) \quad \text{for } j \neq 0$$

$$y_j(t) = y_0(t-j)$$

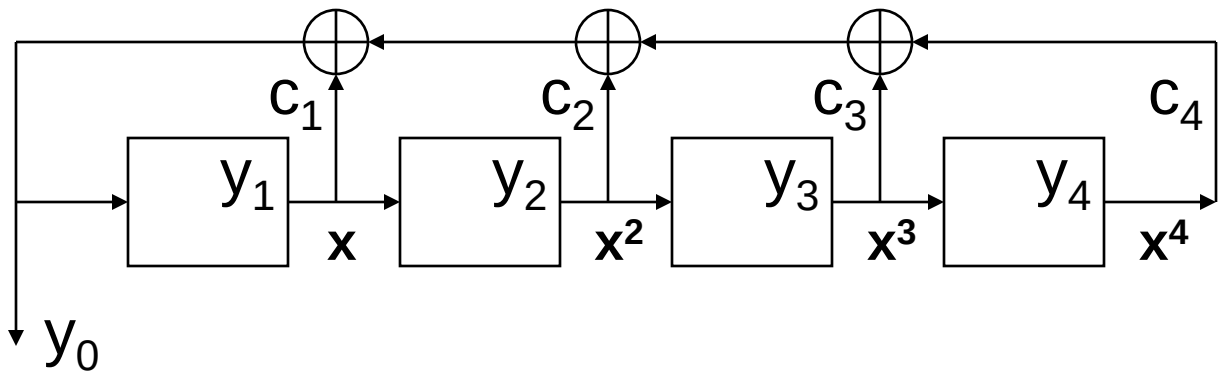
$$y_j(t) = y_0(t)x^j$$

$$y_0(t) = \sum_{j=1}^{j=n} c_j y_j(t)$$

$$y_0(t) = \sum_{j=1}^{j=n} c_j y_0(t) x^j$$

where ***j*** represents the time translation units

Theory of LFSR



$$y_0(t) = \sum_{j=1}^{j=n} c_j y_0(t) x^j$$

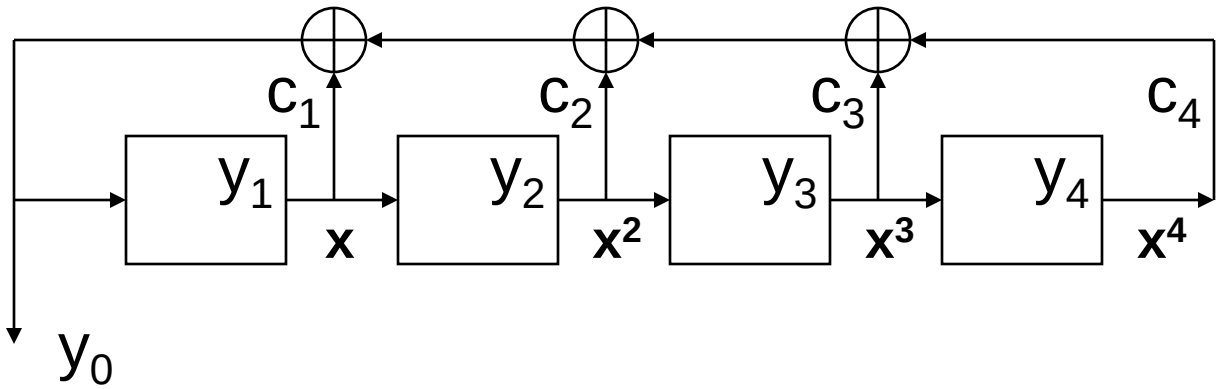
$$y_0(t) = y_0(t) \sum_{j=1}^{j=n} c_j x^j$$

$$y_0(t)(\sum_{j=1}^{j=n} c_j x^j + 1) = 0$$

Polynomial:

$$y_0(t) P_n(x) = 0$$

Theory of LFSR



$$y_0(t)P_n(x) = 0$$

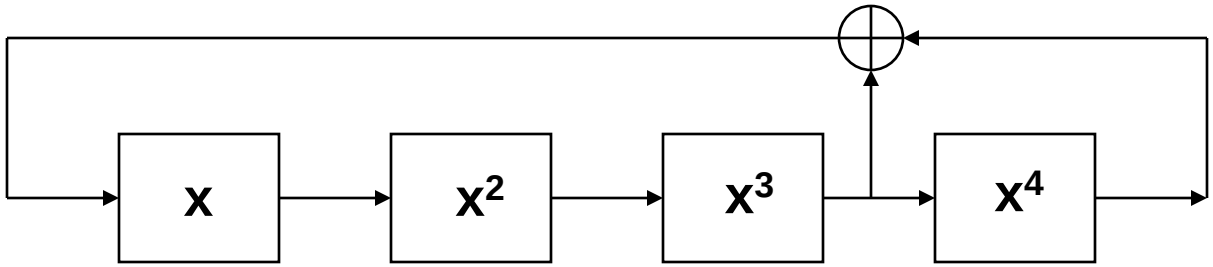
Characteristic polynomial:

For $y_0(t) \neq 0$ $\quad P_n(x) = 0$

where $\quad P_n(x) = 1 + \sum_{j=1}^{j=n} c_j x^j$

Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:

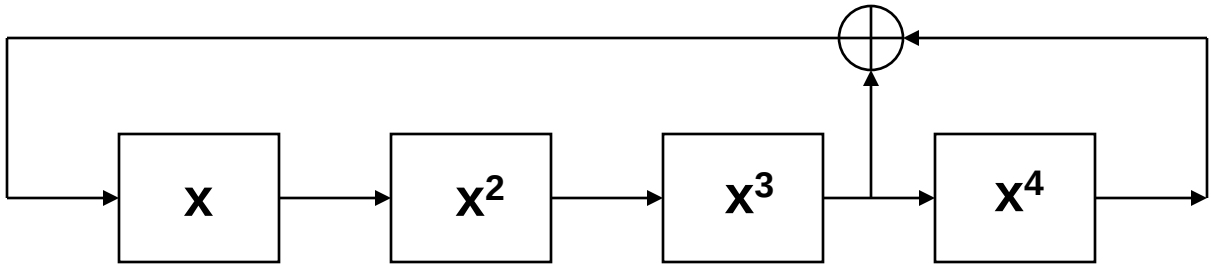


Polynomial: $P(x) = x^4 + x^3 + 1$

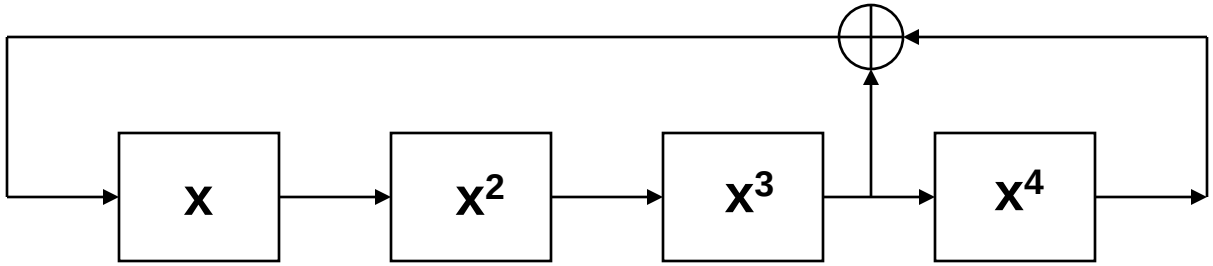
Matrix Equation for Standard LFSR

$$\begin{bmatrix} X_n(t+1) \\ X_{n-1}(t+1) \\ \vdots \\ X_3(t+1) \\ X_2(t+1) \\ X_1(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \\ 1 & h_{n-1} & h_{n-2} & \dots & h_2 & h_1 \end{bmatrix} \begin{bmatrix} X_n(t) \\ X_{n-1}(t) \\ \vdots \\ X_3(t) \\ X_2(t) \\ X_1(t) \end{bmatrix}$$

$X(t+1) = T_s X(t)$ (T_s is *companion matrix*)



Pseudorandom Test Generation



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

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

1 0 0

t	x	x^2	x^3	x^4	t	x	x^2	x^3	x^4
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
- Irreducible polynomial of degree ***n*** is characterized by:
  - An odd number of terms including 1 term
  - Divisibility into $1 + x^k$, where $\boldsymbol{k} = 2^n - 1$
- Any polynomial with all even exponents can be factored and hence is ***reducible***
- An irreducible polynomial is ***primitive*** if it divides the polynomial $1+x^k$ for $k = 2^n - 1$, but not for any smaller positive integer k

Theory of LFSR: Examples

Polynomials of degree n=3 (examples):

$k = 2^n - 1 = 2^3 - 1 = 7$

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):

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

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

The polynomials don't divide evenly the polynomial $x^7 + 1$

Theory of LFSR: Examples

Comparison of test sequences generated:

Primitive polynomials

x^3+x+1	x^3+x^2+1
100	100
110	010
111	101
011	110
101	111
010	011
001	001
100	100

Non-primitive polynomials

x^3+1	x^3+x^2+x+1
100	100
010	110
001	011
100	001
010	100
001	110
100	011
010	001

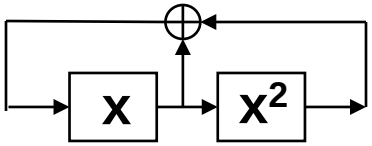
Theory of LFSR: Examples

Reducible polynomial (non-primitive):

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

x^3+1 $\quad x+1$ $\quad x^2+x+1$

100	1
010	0
001	1
100	



01
10
11
01

Primitive polynomial

Multiplication of two primitive polynomials:

$$\begin{array}{l} x^2+x+1 \\ \underline{x^2+x+1} \\ x^2+x\ \ +1 \\ x^3+x^2+x \\ \underline{x^4+x^3+x^2} \\ x^4+x^2+1 \end{array}$$

Is

$$x^4+x^2+1$$

a primitive polynomial?

Theory of LFSR: Examples

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

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 $x^3 + 1$

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

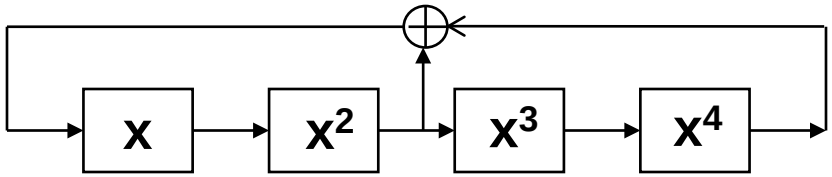
Divisibility check:

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

Theory of LFSR: Examples

Non-primitive polynomial

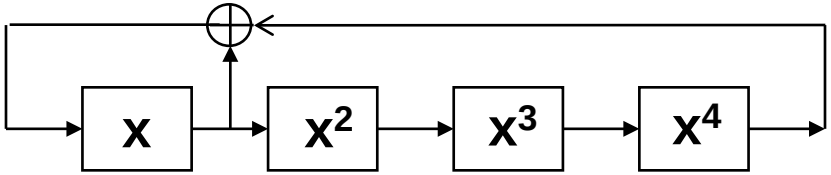
$x^4 + x^2 + 1$



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

Primitive polynomial

$x^4 + x + 1$

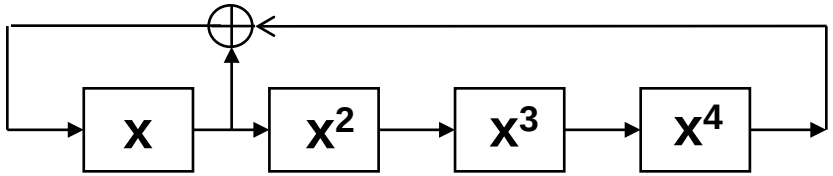


0001	1011	1001
1000	0101	0100
1100	1010	0010
1110	1101	0001
1111	0110	
0111	0011	

Theory of LFSR: Examples

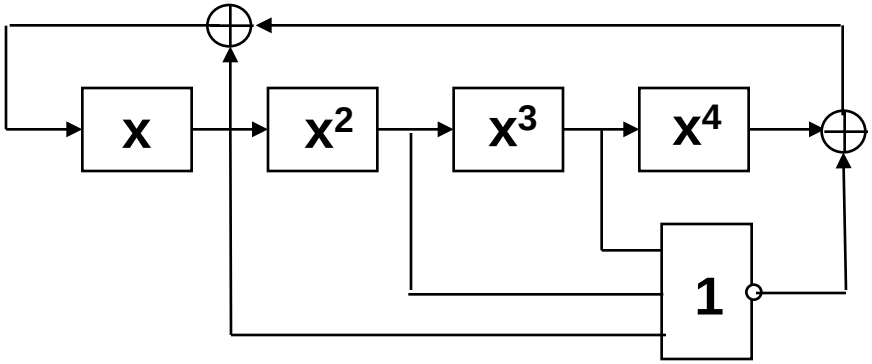
Primitive polynomial

$x^4 + x + 1$



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

Zero generation:



0000	1011	1001
1000	0101	0100
1100	1010	0010
1110	1101	0001
1111	0110	0000
0111	0011	

Theory of LFSR: Reciprocal Polynomials

The reciprocal polynomial of *P(*X) is defined by:

$(X) = X^N P_N (1/X) = X^N \{1 + C_j X^{-J}\}$

$(X) = X^N + C_j X^{N-J}$ *for* $1 \leq i \leq N$

Thus every coefficient C_i in $P(X)$ is replaced by $C_{N-I.}$

Example:

The reciprocal of polynomial $P_3(X) = 1 + X + X^3$

is $P'_3(X) = 1 + X^2 + X^3$

The reciprocal of a primitive polynomial is also primitive

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

Number of PP of degree *n*

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

Examples of PP (exponents of terms):

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

Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence

Creation of the shortest bit-stream:

Given test sequence:

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

10010 1 01111

1
0

Seed

Expected shortest LFSR sequence:

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

Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence

Expected shortest LFSR sequence:

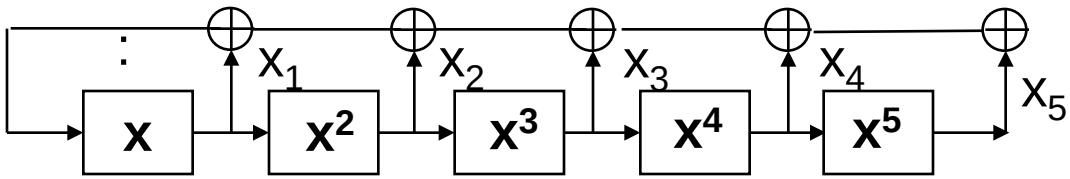
01111 (4)

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

System of linear equations:

$$\begin{vmatrix} 01111 \\ 10111 \\ 01011 \\ 10101 \\ 01010 \\ 00101 \end{vmatrix} \times \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{vmatrix} = \begin{vmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{vmatrix}$$

We are looking for values of x_i



Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence

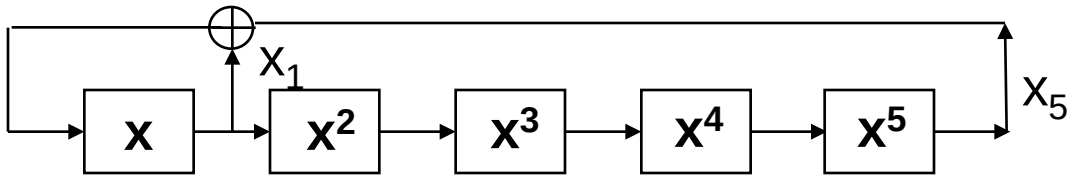
System of linear equations:

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

Solving the equation by Gaussian elimination:

$$\begin{array}{c} 1,2,4,6 \\ 4,6 \\ 1,3 \\ 2,4 \\ 1,2,3,4,6 \end{array} \begin{vmatrix} 01000 \\ 10000 \\ 00100 \\ 00010 \\ 00001 \\ 00001 \end{vmatrix} \times \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{vmatrix} = \begin{vmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{vmatrix}$$

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



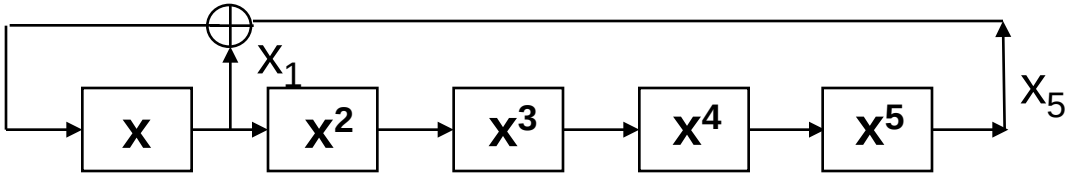
Solution:

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:

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



LFSR sequence:

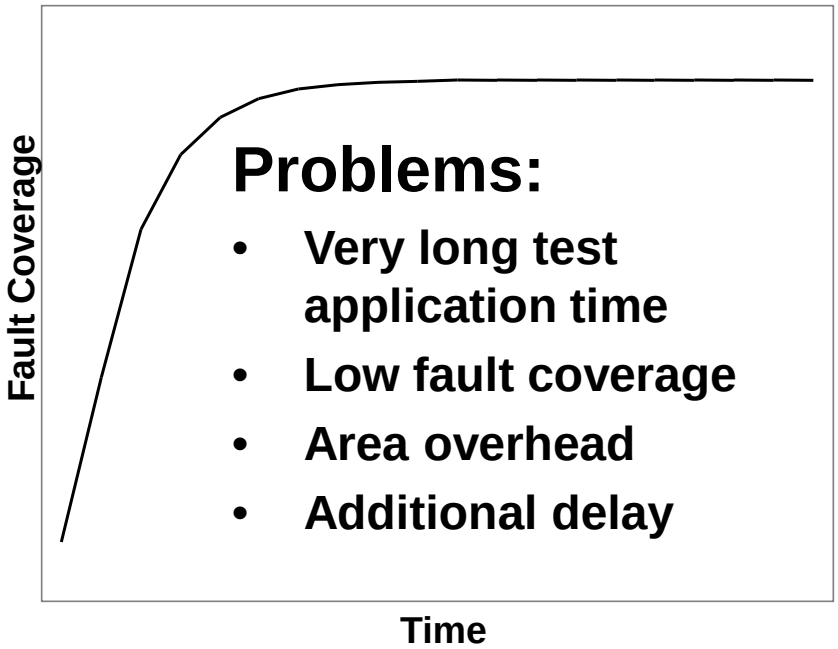
(1) 01111 (4)
(2) 10111
(3) 01011
(4) 10101 (3)
(5) 01010 (2)
(6) 00101
(7) 10010 (1)

Given deterministic test sequence:

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

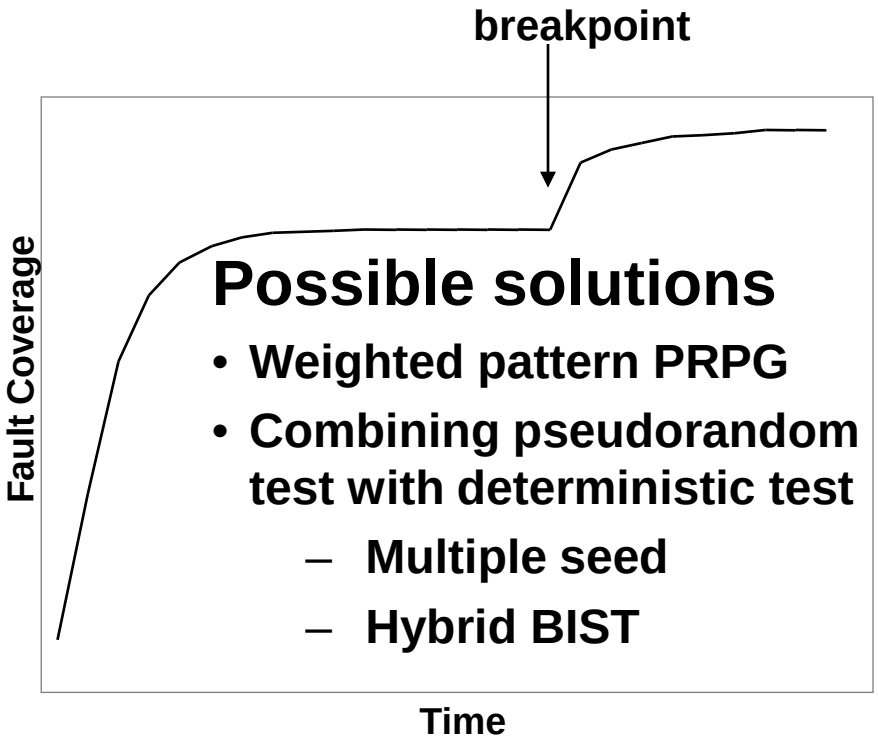
BIST: Test Generation

Pseudorandom Test generation by LFSR:



Problems:

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

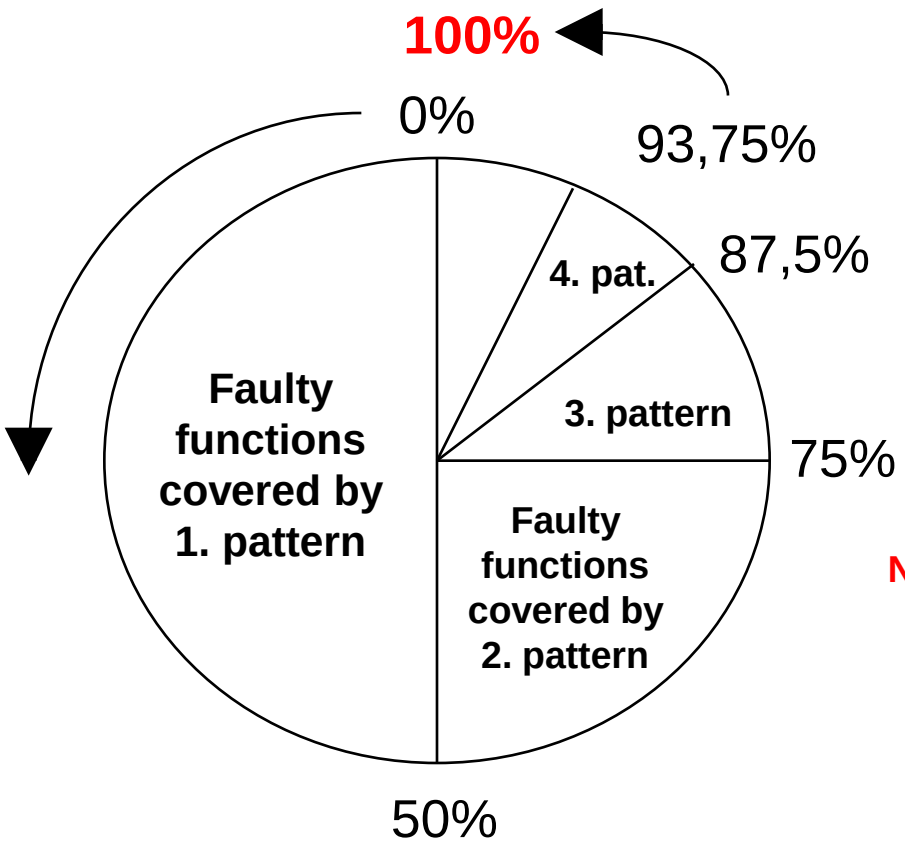


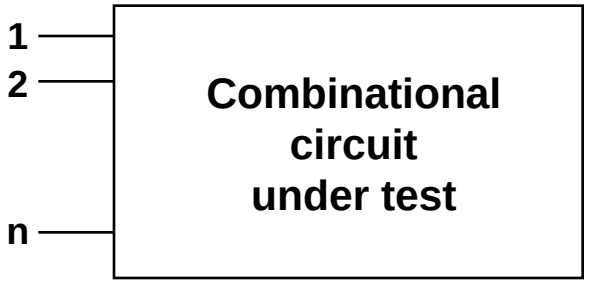
Possible solutions

- Weighted pattern PRPG
- Combining pseudorandom test with deterministic test
  - Multiple seed
  - Hybrid BIST

BIST: Fault Coverage

Fault coverage is rapidly growing:





Truth table:

	Patterns	Functions			
1	00…000	01	01	01…101	→ 2^{2^n-1} tested 50%!
	00…001	00	11	00…011	
	00…010	00	00	11…111	
	…			…	
2^n	11…111	00	00	00…111	

Number of patterns

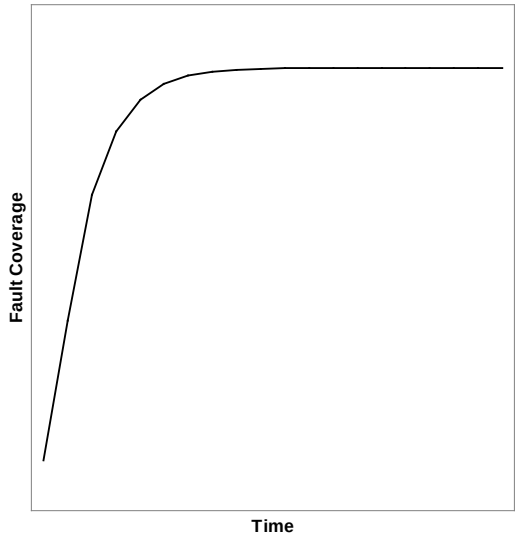
Number of functions 1 → 2^{2^n}

BIST: Fault Coverage

Pseudorandom Test generation by LFSR:

Motivation of using 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

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

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:

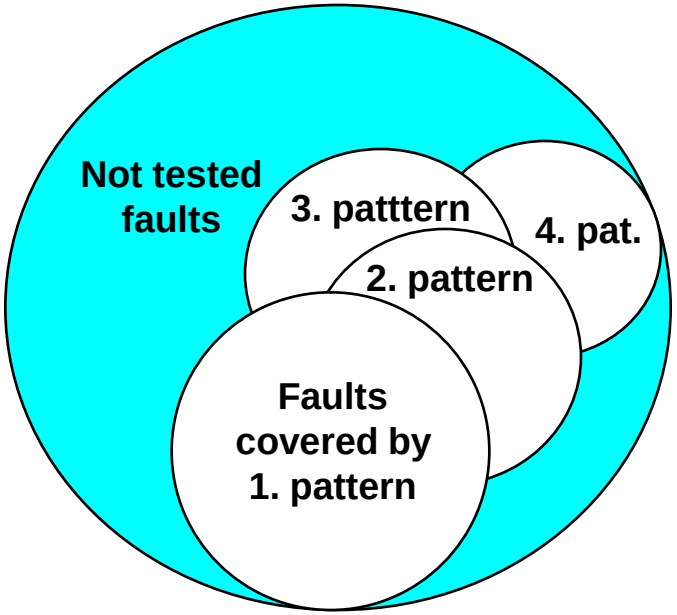
- **clock-signals should not be random**
- **control signals such as reset, should be activated with low probability**
- **data signals may be chosen randomly**

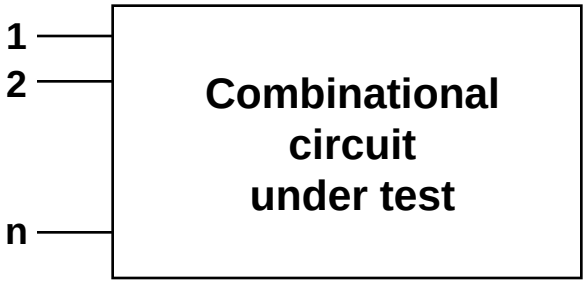
Microprocessor testing

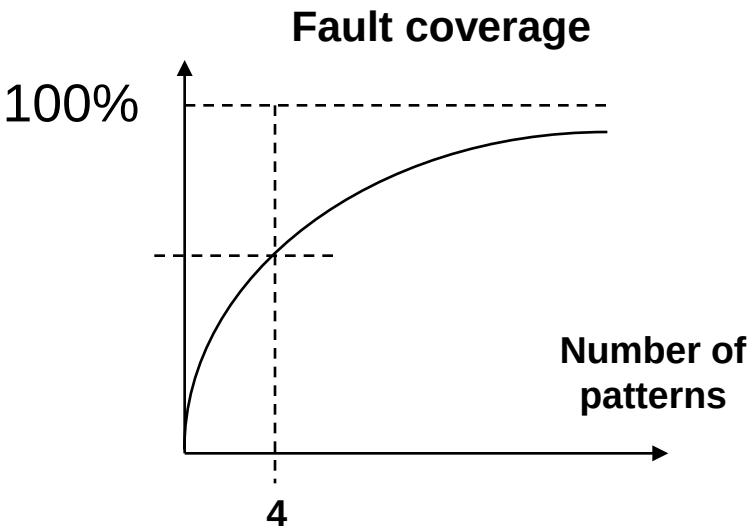
- **A test generator picks randomly an instruction and generates random data patterns**
- **By repeating this sequence a specified number of times it will produce a test program which will test the microprocessor by randomly excercising its logic**

BIST: Structural Approach to Test

Testing of structural faults:

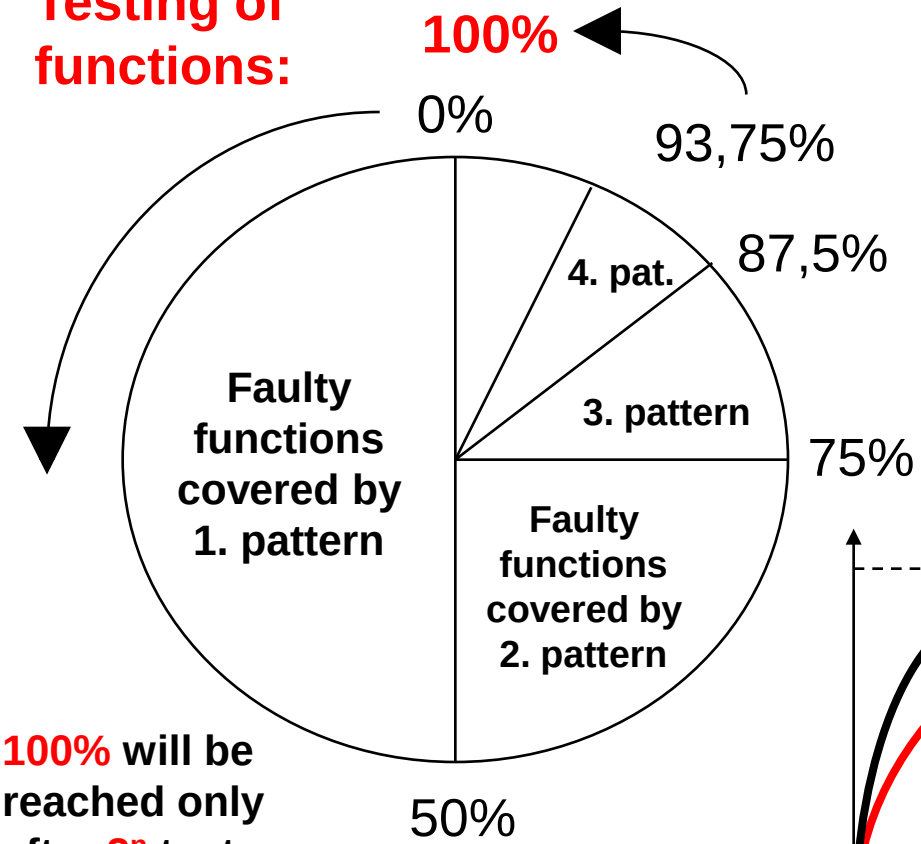






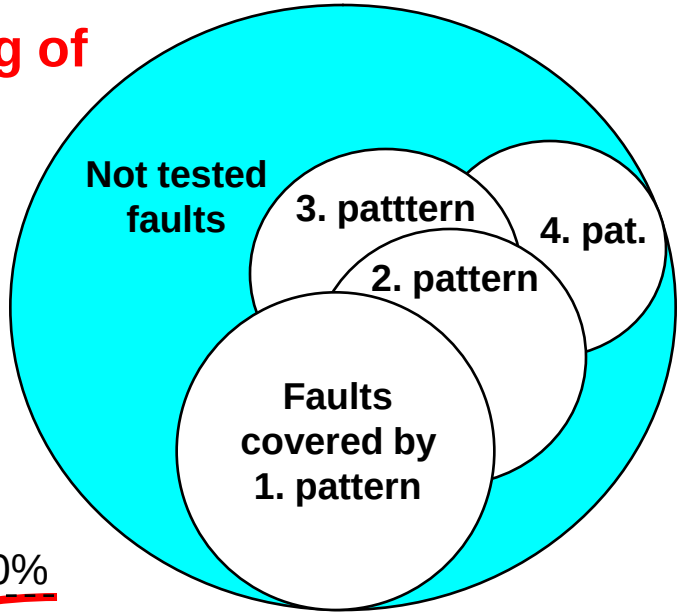
BIST: Two Approaches to Test

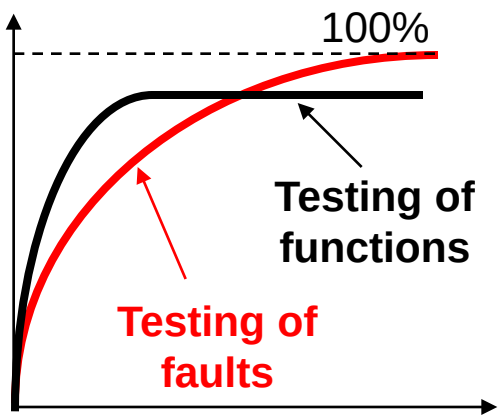
Testing of functions:



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

Testing of faults:





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

BIST: Other test generation methods

Universal test sets

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

Properties of exhaustive tests

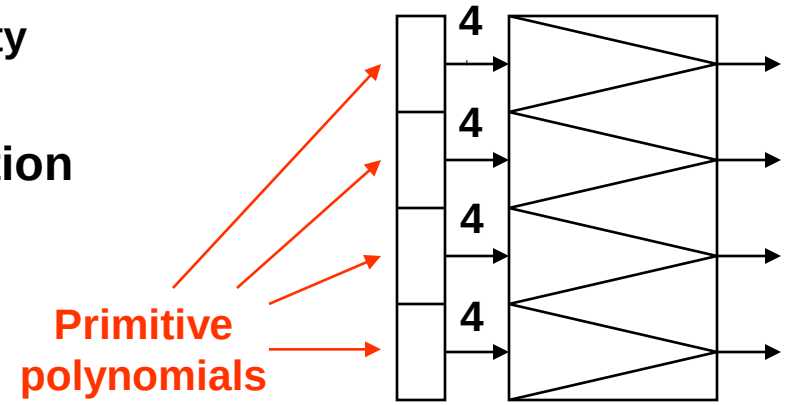
1. **Advantages** (concerning the stuck at fault model):
   - **test pattern generation is not needed**
   - **fault simulation is not needed**
   - **no need for a fault model**
   - **redundancy problem is eliminated**
   - **single and multiple stuck-at fault coverage is 100%**
   - **easily generated 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**

BIST: Other test generation methods

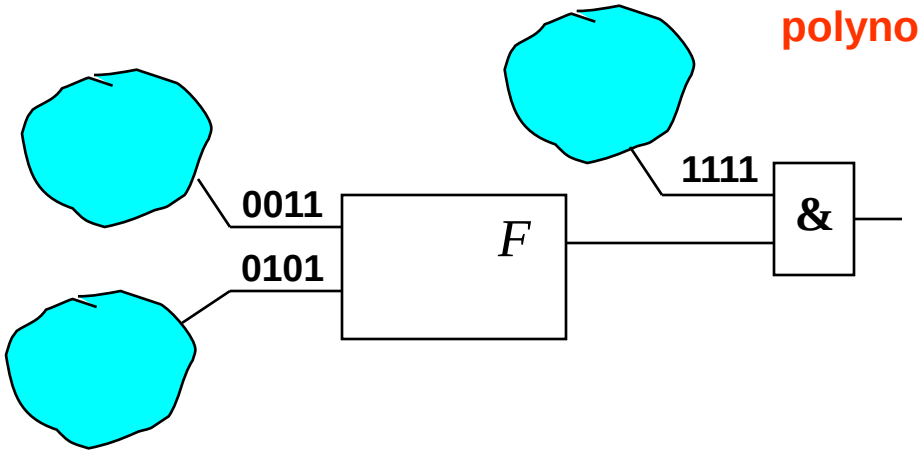
Pseudo-exhaustive test sets:

- **Output function verification**
  - **maximal parallel testability**
  - **partial parallel testability**
- **Segment function verification**

Output function verification



Segment function verification



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

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

Testing carry-lookahead adder

General expressions:

$$G_i = a_i b_i \qquad P_i = a_i \overline{b_i} \vee \overline{a_i} b_i \qquad C_n = G_n \vee P_n C_{n-1}$$

$$C_n = G_n \vee P_n (G_{n-1} \vee P_{n-1} C_{n-2}) = G_n \vee P_n G_{n-1} \vee P_n P_{n-1} C_{n-2}$$

n-bit carry-lookahead adder:

$$C_1 = G_1 \vee P_1 C_0 = a_1 b_1 \vee a_1 \overline{b_1} C_0 \vee \overline{a_1} b_1 C_0 = f(a_1, b_1, C_0)$$

$$C_3 = G_3 \vee P_3 G_2 \vee P_3 P_2 G_1 \vee \boxed{P_3 P_2 P_1 C_0}$$

$$P_3 P_2 P_1 C_0 = (a_3 \overline{b_3} \vee \overline{a_3} b_3)(a_2 \overline{b_2} \vee \overline{a_2} b_2)(a_1 \overline{b_1} \vee \overline{a_1} b_1) C_0$$

Testing carry-lookahead adder

$$\boxed{P_3P_2P_1C_0} = (a_3\overline{b_3} \vee \overline{a_3}b_3)(a_2\overline{b_2} \vee \overline{a_2}b_2)(a_1\overline{b_1} \vee \overline{a_1}b_1)C_0$$

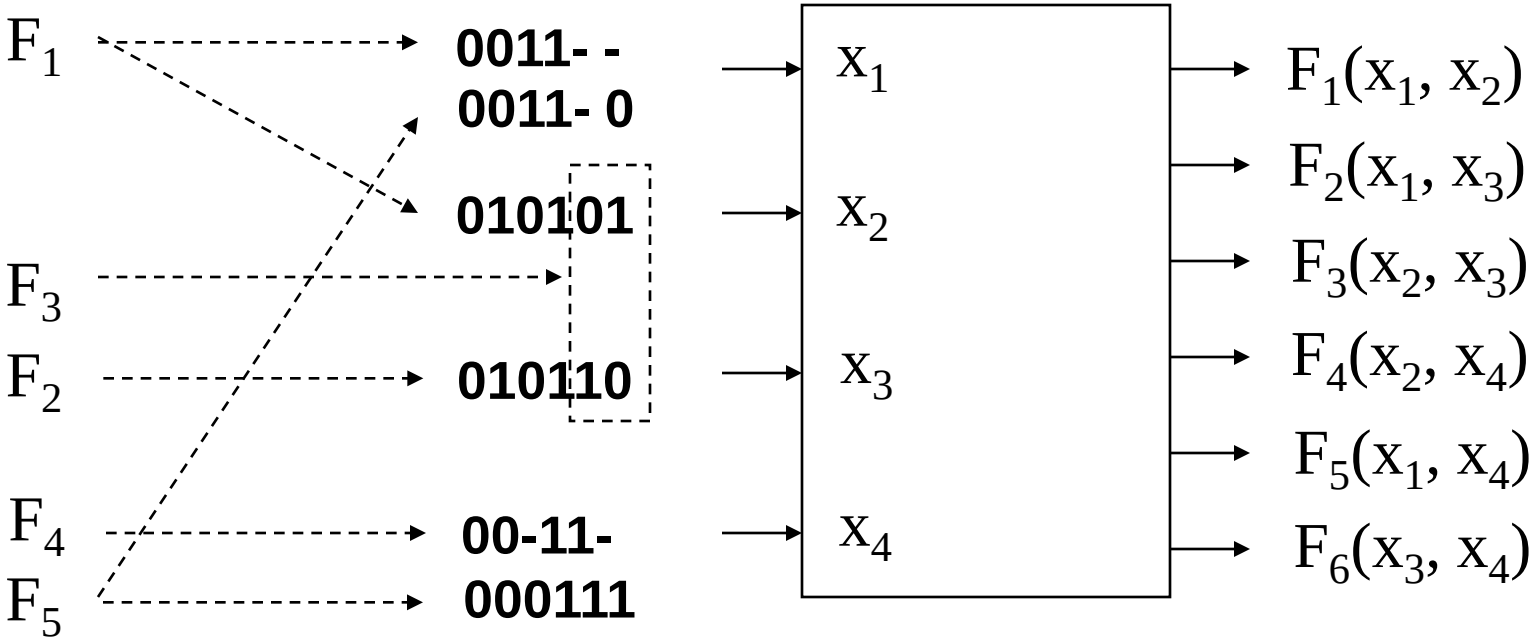
																R
Testing ≡ 0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	0	0	1	1	0	0	1	1	0	0	1	1	1	1
Testing ≡ 1	1	0	0	1	1	1			1	1			1	0
	0	1	1	0	1	1			1	1			1	0
	1	1			0	1	1	0	1	1			1	0
	1	1			1	0	0	1	1	1			1	0
	1	1			1	1			0	1	1	0	1	0
	1	1			1	1			1	0	0	1	1	0
	1	1			1	1			1	1			0	0

For 3-bit carry lookahead adder for testing only this part of the circuit at least 9 test patterns are needed

Increase in the speed implies worse testability

BIST: Other test generation methods

Output function verification (partial parallelity)

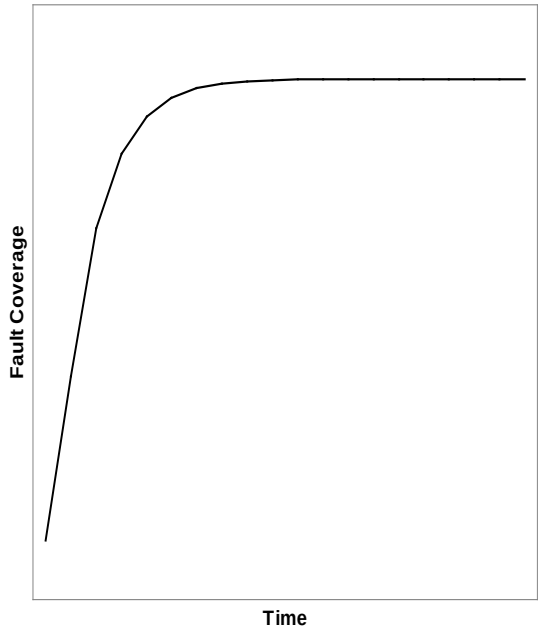


Exhaustive testing - 16
Pseudo-exhaustive, full parallel - 4
Pseudo-exhaustive, partially parallel - 6

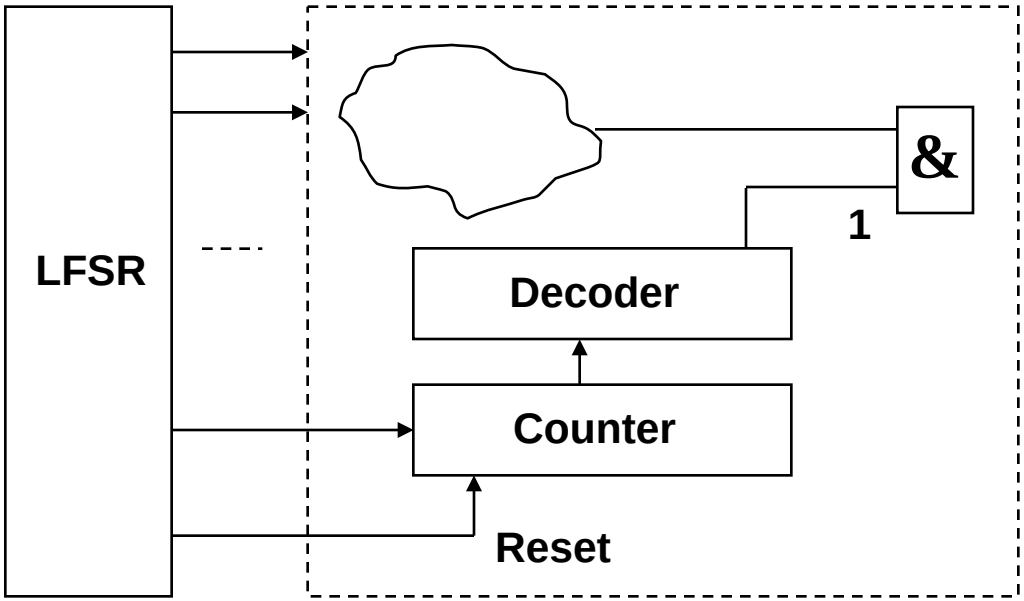
Problems with Pseudorandom Test

The main motivations of using random patterns are:

- low generation cost
- high initial efeciency



Problem: **low fault coverage**

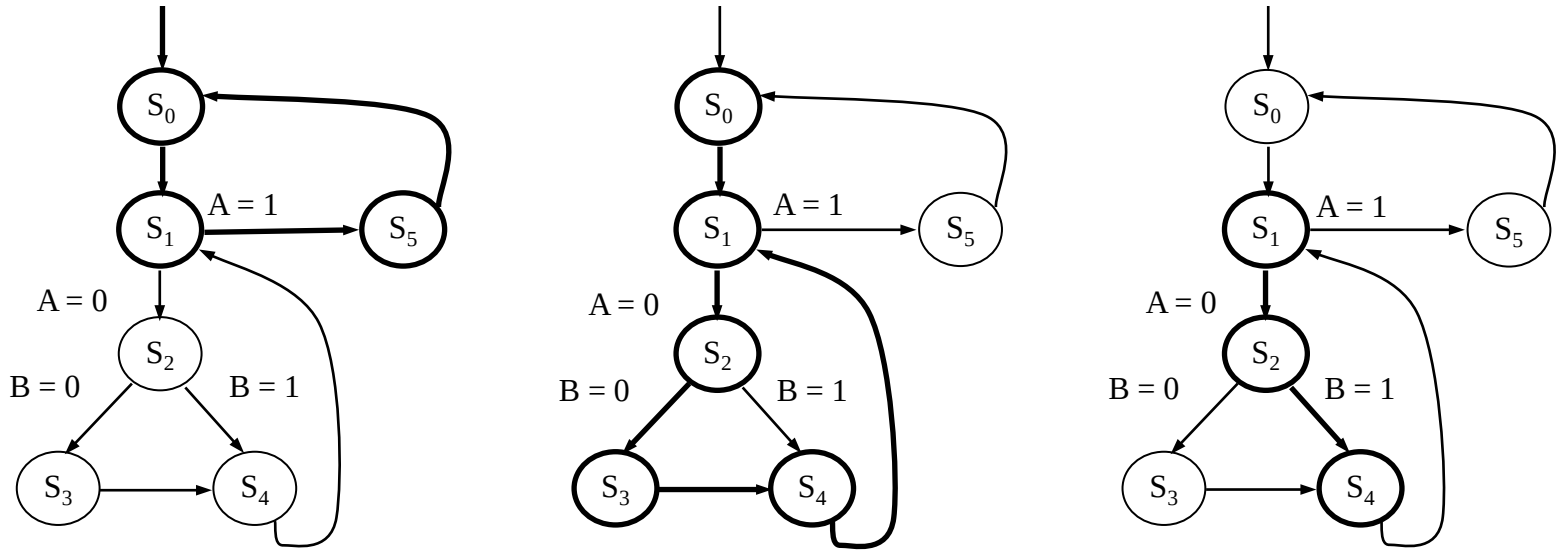


If Reset = 1 signal has probability 0,5 then counter will not work and
1 for AND gate may never be produced

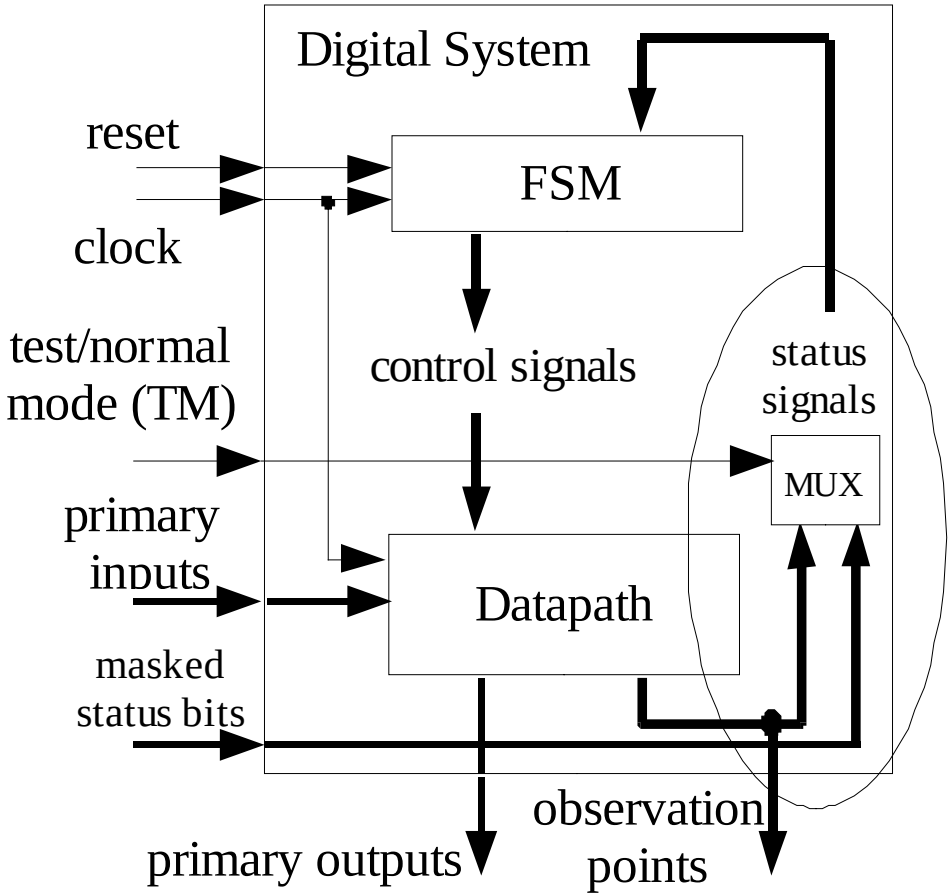
Sequential BIST

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



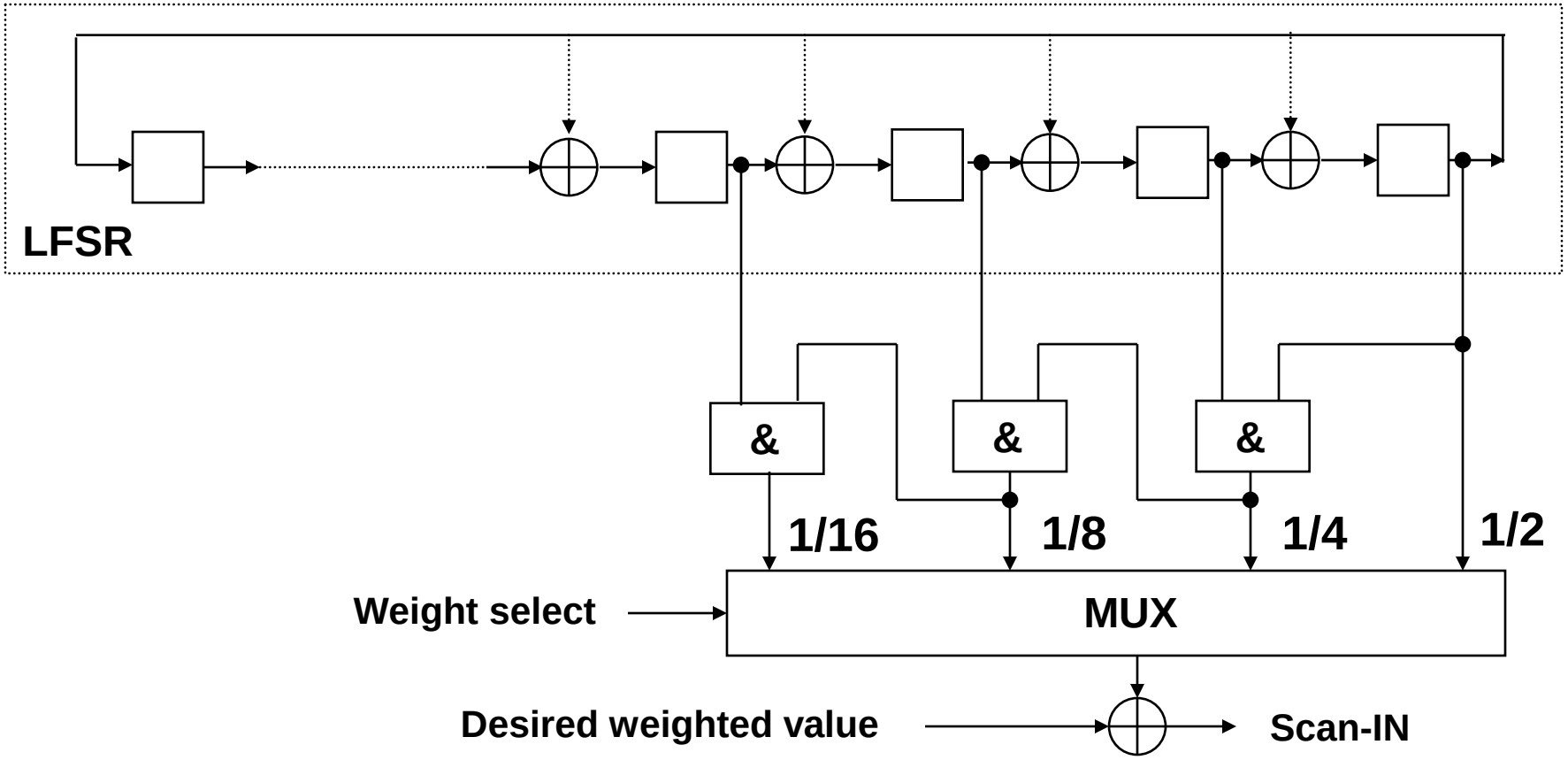
Sequential BIST



- **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**

BIST: Weighted pseudorandom test

Hardware implementation of weight generator

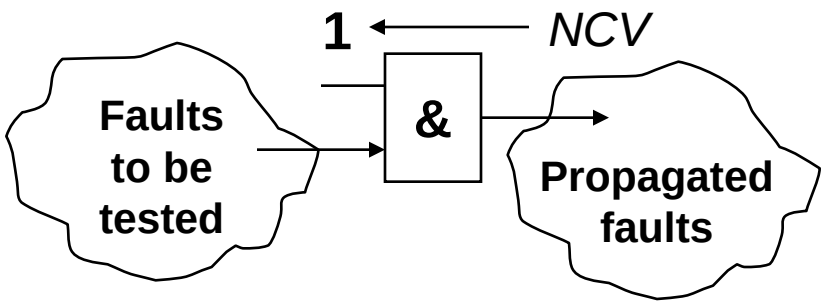


BIST: Weighted pseudorandom test

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

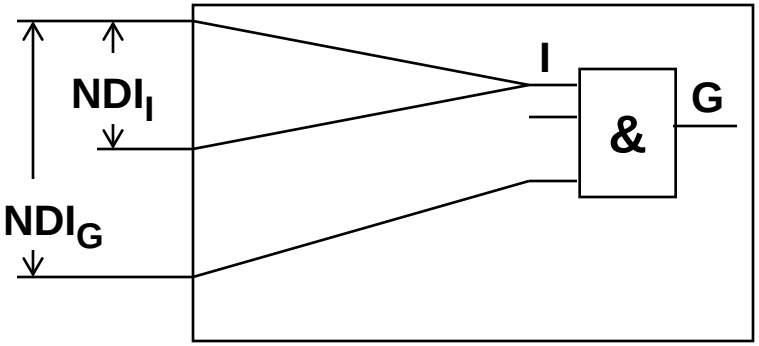
The more faults that must be tested through a gate input, the more the other inputs should be weighted to NCV

NDI - number of circuit inputs for each gate to be the number of PIs or SRLs in the backtrace cone

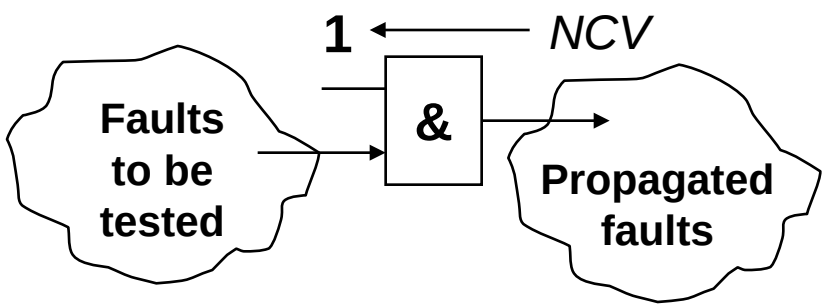
PI - primary inputs

SRL - scan register latch

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



BIST: Weighted pseudorandom test

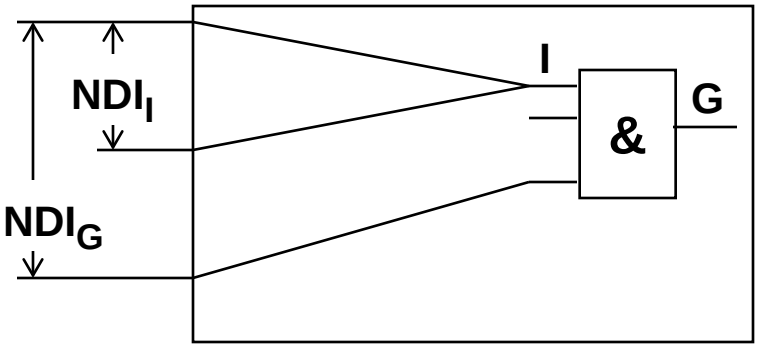


$R_I = NDI_G / NDI_I$

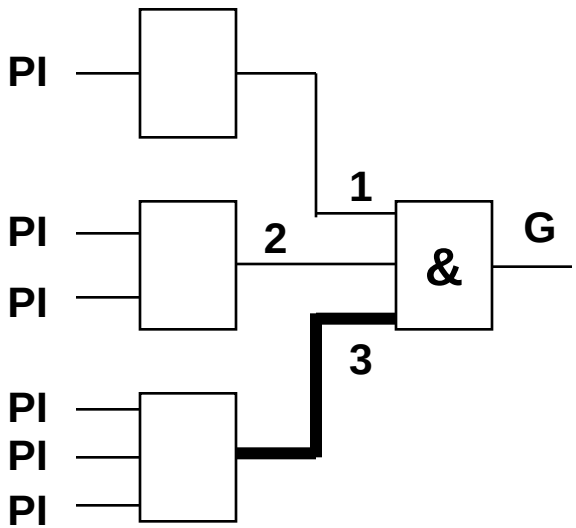
R_I - the desired ratio of the NCV to the controlling value for each gate input

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:

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

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

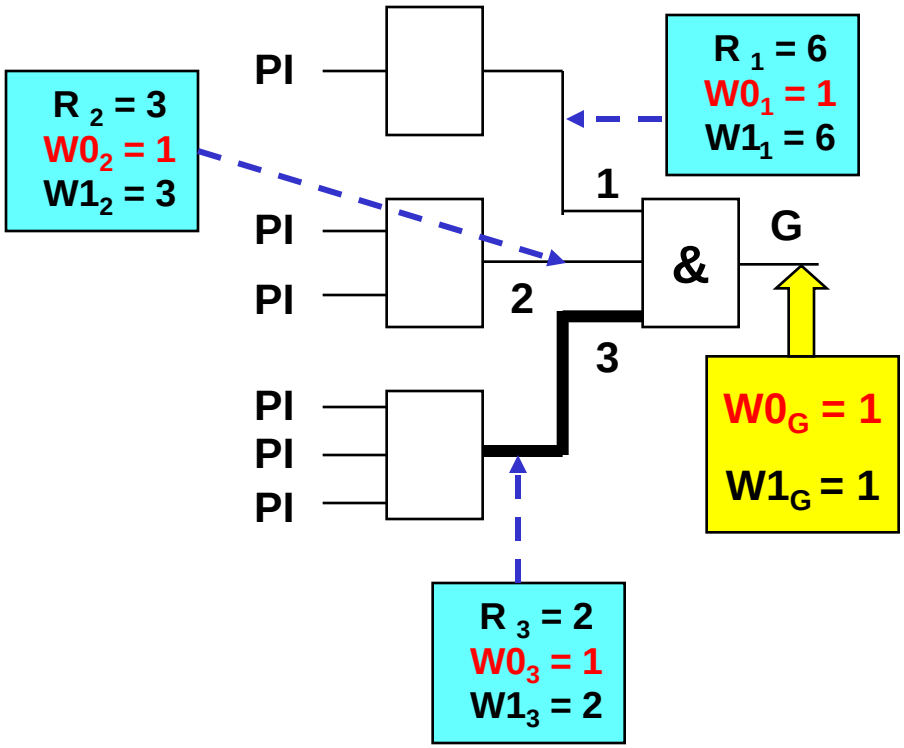
$R_3 = NDI_G / NDI_I = 6/3 = 2$

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

WV - the value to which the input is biased

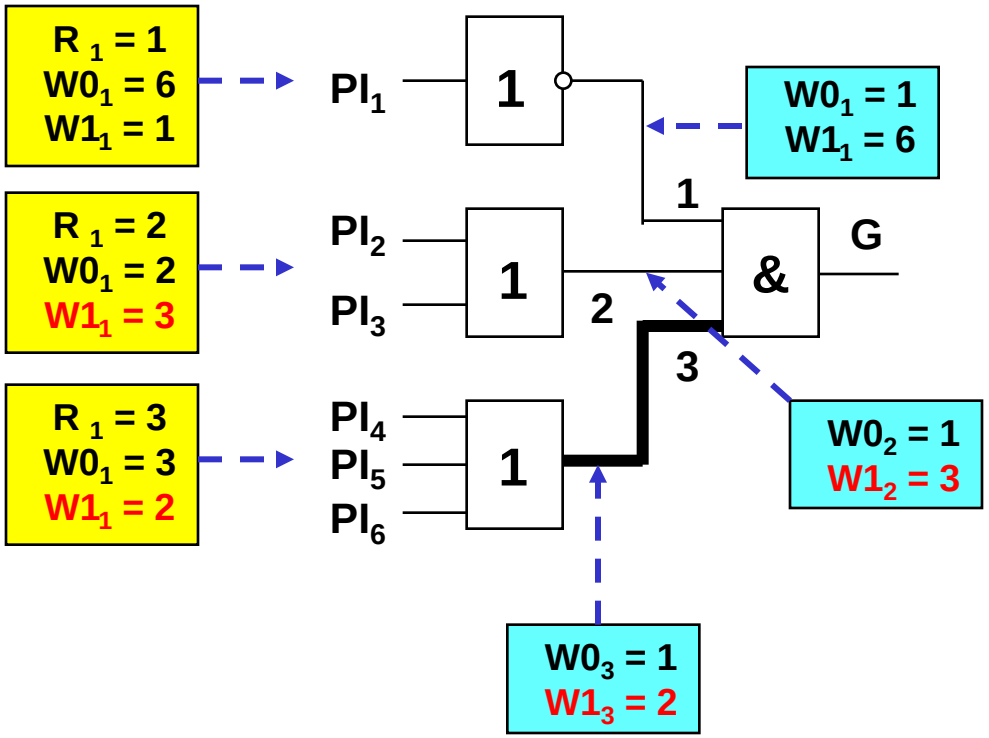
WV = 0, if W0 > W1 else WV = 1

Calculation of W0, W1

Function	WO_I	$W1_I$
AND	WO_G	$R_I * W1_G$
NAND	$W1_G$	$R_I * WO_G$
OR	$R_I * WO_G$	$W1_G$
NOR	$R_I * W1_G$	WO_G

BIST: Weighted pseudorandom test

Calculation of signal weights:



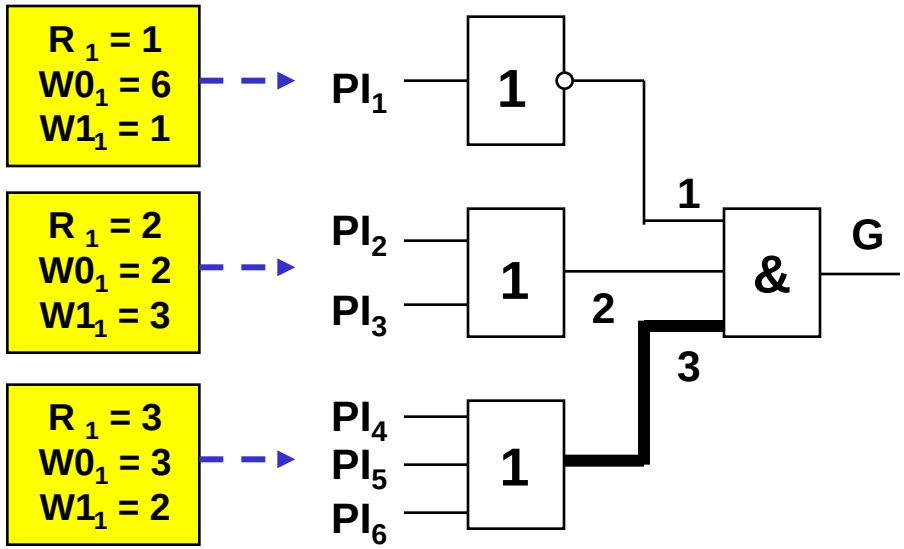
Backtracing from all the outputs to all the inputs of the given cone

Weights are calculated for all gates and PIs

Function	WO_I	$W1_I$
OR	$R_I * WO_G$	$W1_G$
NOR	$R_I * W1_G$	WO_G

BIST: Weighted pseudorandom test

Calculation of signal probabilities:



WF - weighting factor indicating the amount of biasing toward weighted value

WF = max {W0,W1} / min {W1,W0}

Probability:

P = WF / (WF + 1)

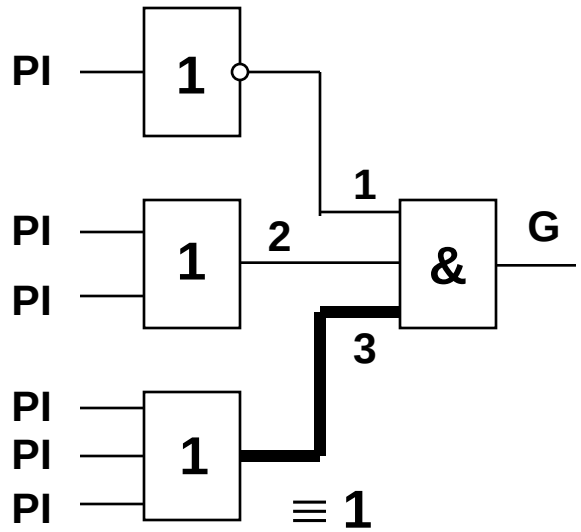
For PI_1 **: W0 = 6 W1 = 1 P1 = 1/7 = 0.15**

For PI_2 and PI_3 **: W0 = 2 W1 = 3 P1 = 3/5 = 0.6**

For 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):

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

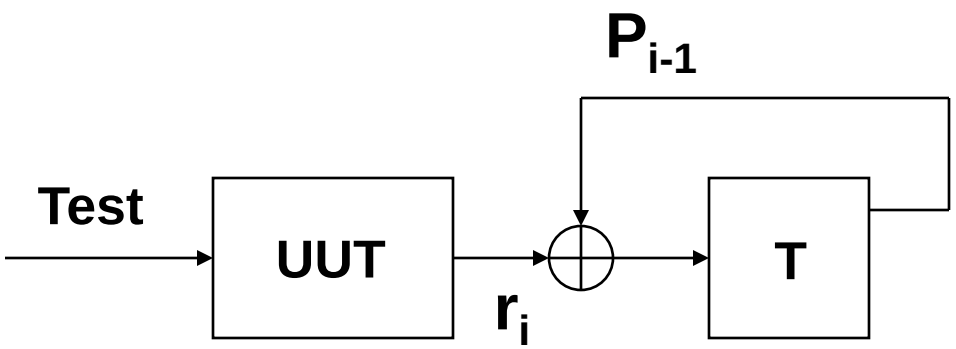
2) weighted probabilities:

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

BIST: Response Compression

1. Parity checking

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

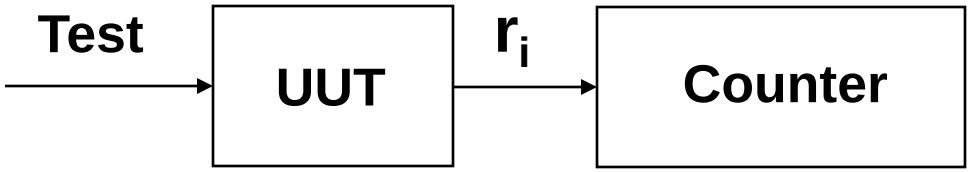


2. One counting

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

3. Zero counting

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

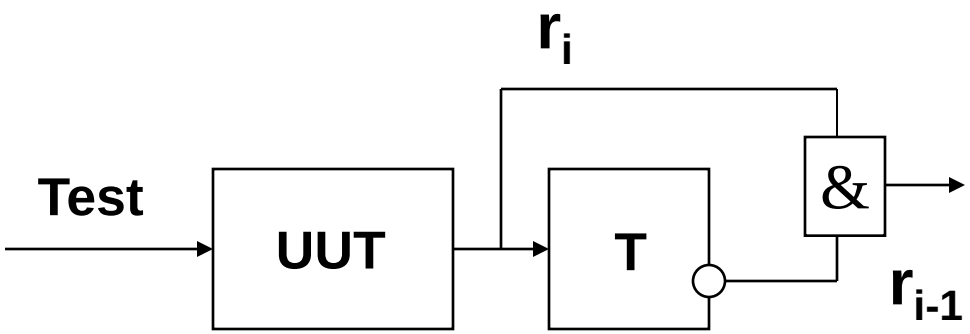


BIST: Response Compression

4. Transition counting

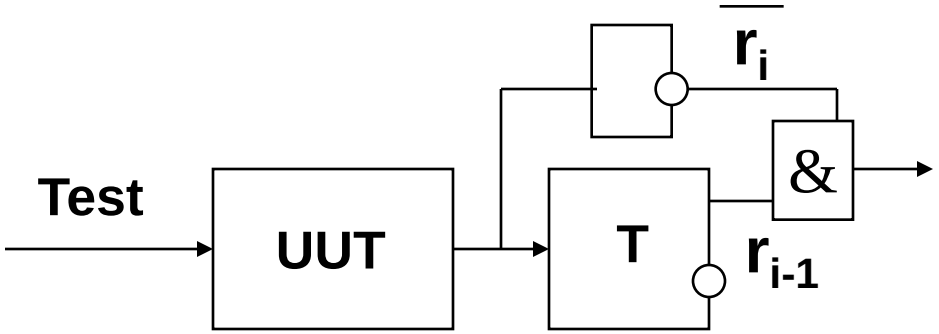
a) Transition 0→1

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



b) Transition 1→0

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

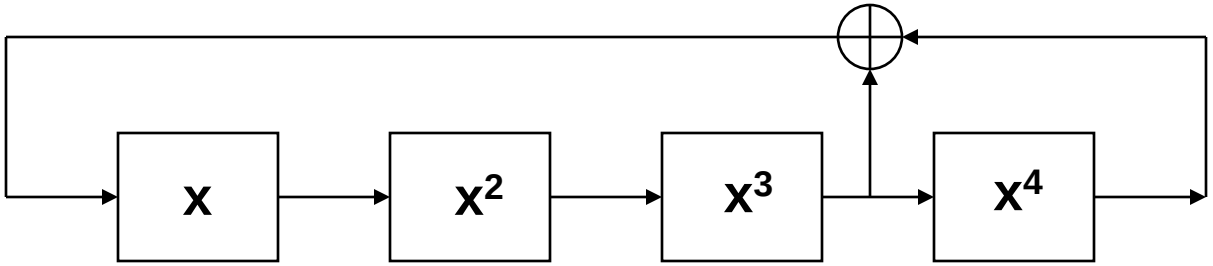


5. Signature analysis

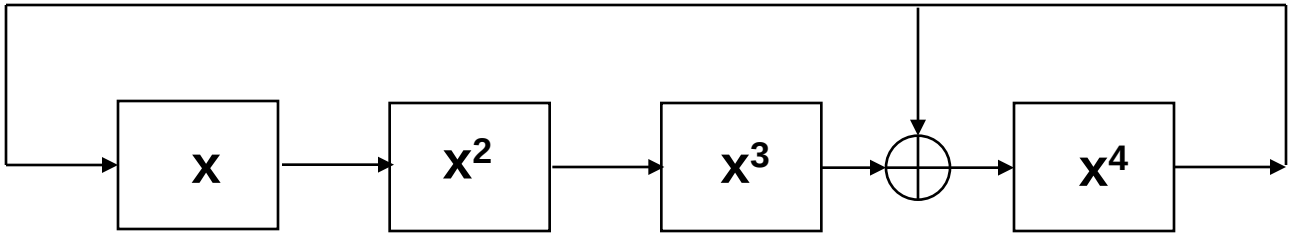
Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:

Standard LFSR



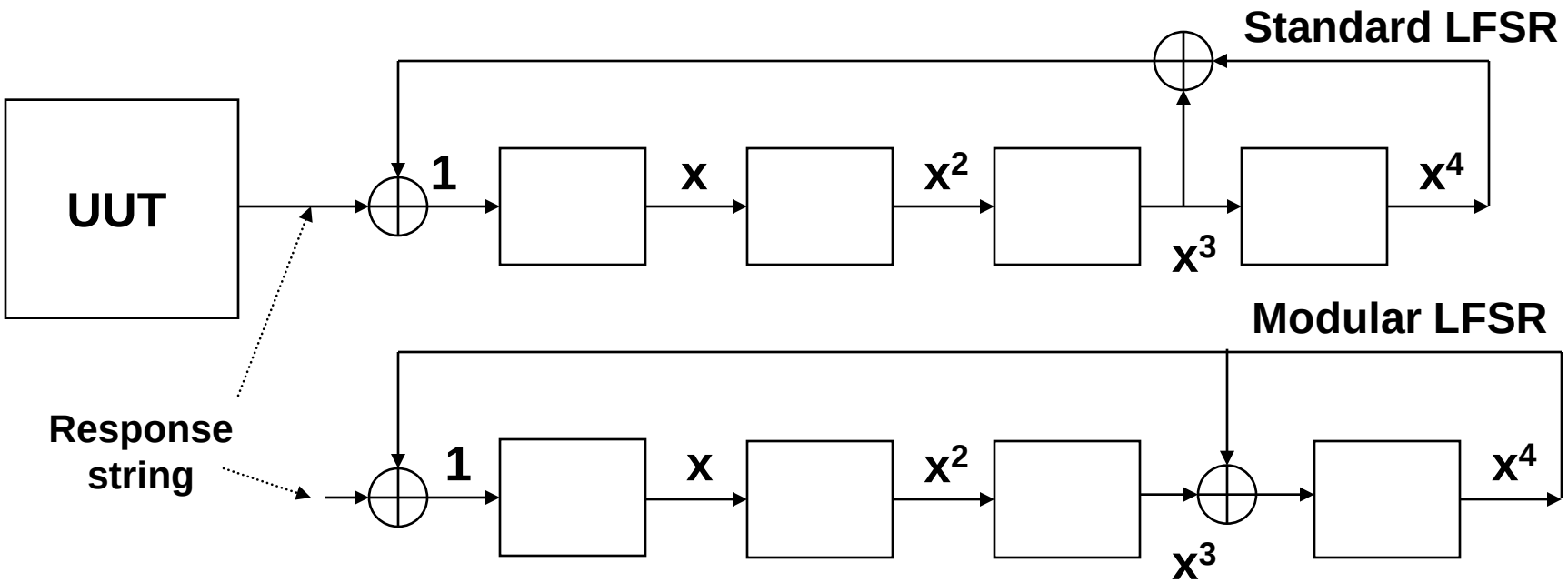
Modular LFSR



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

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

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

Example:

$11001 \rightarrow R(x) = x^4 + x^3 + 1$

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

BIST: Signature Analysis

Arithmetic of coefficients:

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

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

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

Examples:

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

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

Theory of LFSR

Characteristic Polynomials:

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

Multiplication of polynomials

$$\begin{array}{r} x^2 + x + 1 \\ x^2 + 1 \\ \hline x^2 + x + 1 \\ x^4 + x^3 + x^2 \qquad\qquad \\ \hline x^4 + x^3 \qquad\quad + x + 1 \end{array}$$

Theory of LFSR

Characteristic Polynomials:

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

Quotient

Divider $x^2 + 1$

Dividend

$$\begin{array}{l|l}
 & x^2 + x + 1 \\
\hline
x^2 + 1 & x^4 + x^3 \qquad\quad + 1 \\
 & x^4 \qquad + x^2 \\
\hline
 & \quad x^3 + x^2 \qquad + 1 \\
 & \quad x^3 \qquad + x \\
\hline
 & \qquad x^2 + x + 1 \\
 & \qquad x^2 \qquad\quad + 1 \\
\hline
 & \qquad\qquad x
\end{array}$$

Division of polynomials

Remainder

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 unaltered 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 <u>signature</u> of the <u>test response</u> string P(x) from the UUT

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

Signature is the CRC code word

<u>Example:</u>

G(x)

1 0 1 0 1 1

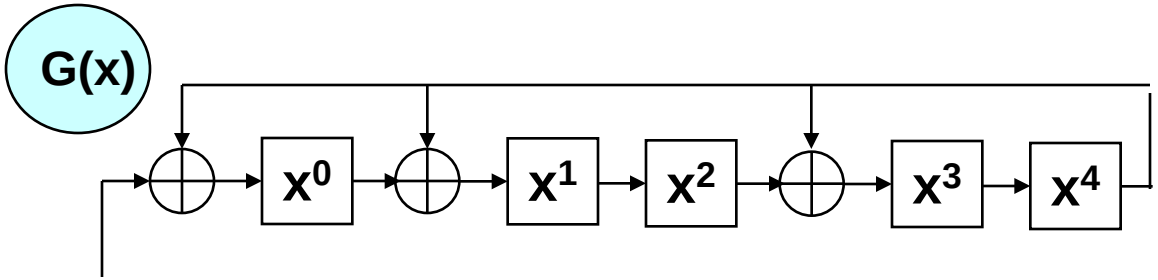
P(x)

```
1 0 1                = Q(x) = x^2 + 1
1 0 1 0 1 1 | 1 0 0 0 1 0 1 0
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

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

BIST: Signature Analysis



IN: 01 **010001** → Shifted into LFSR

P(x)

The division process can be mechanized using LFSR

The divisor polynomial G(x) is defined by the feedback connections

Shift creates x^5 which is replaced by $x^5 = x^3 + x + 1$

Compressor

G(x) → 1 0 1 0 1 1

```
      1 0 1
    ---------
1 | 1 0 0 0 1 0 1 0
  | 1 0 1 0 1 1
    ---------
0 | 0 0 1 0 0 1 1 0
      1 0 1 0 1 1
    ---------
  | 0 0 1 1 0 1
```

x^5

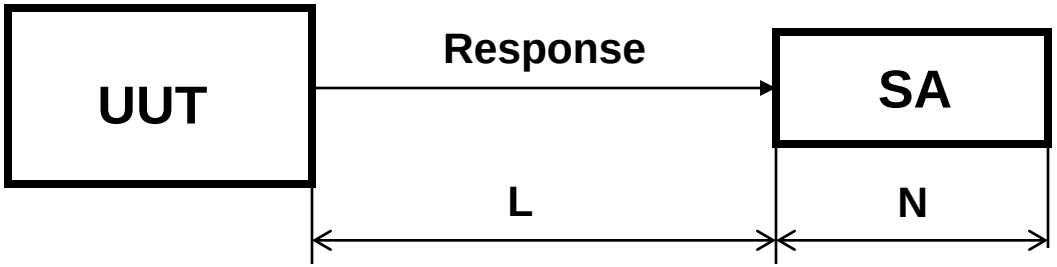
P(x) → **Response**

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

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

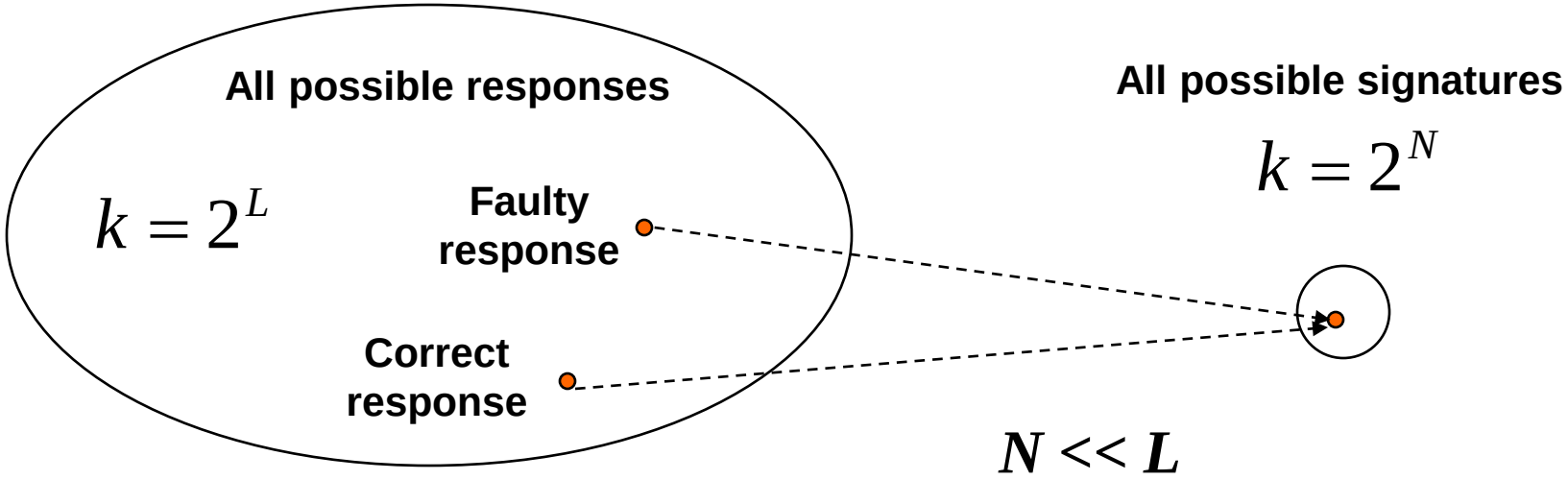
BIST: Signature Analysis

Aliasing:



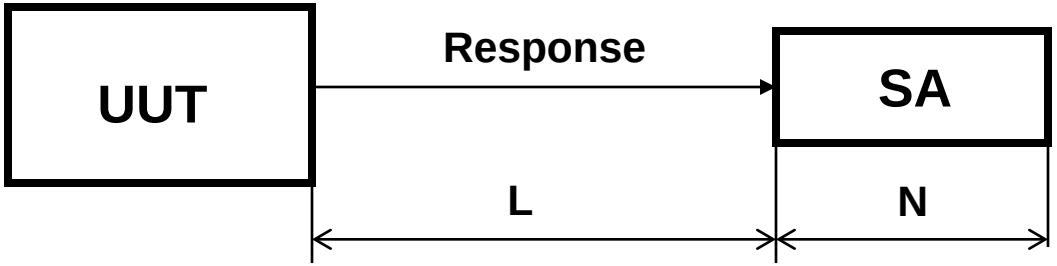
L - test length

N - number of stages in Signature Analyzer



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

$2^{L-N} - 1$ - aliasing is possible

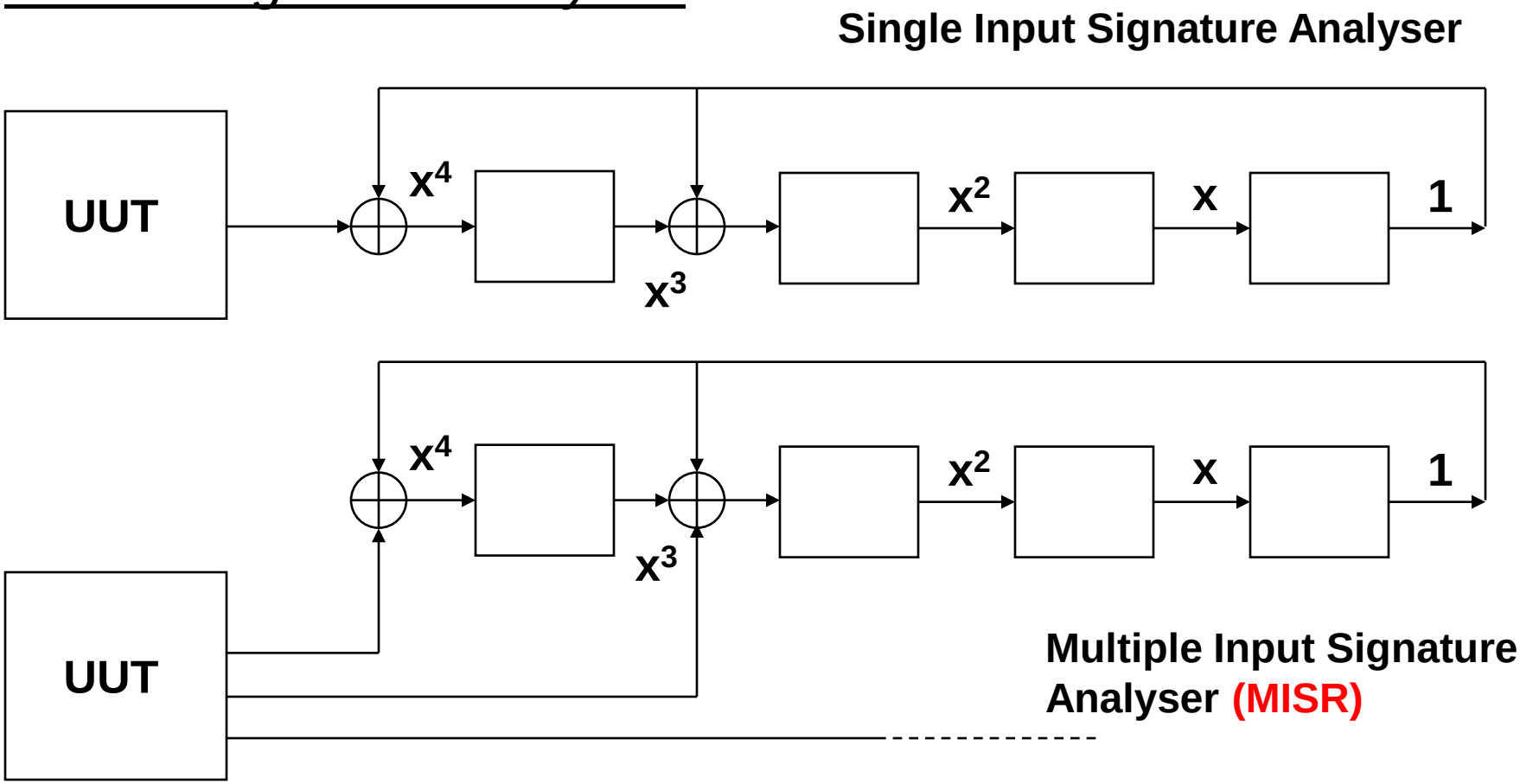
000000000000000 ... 00000 XXXXX

L N

Probability of aliasing: $P = \dfrac{2^{L-N} - 1}{2^L - 1} \xrightarrow{L \gg 1} P = \dfrac{1}{2^N}$

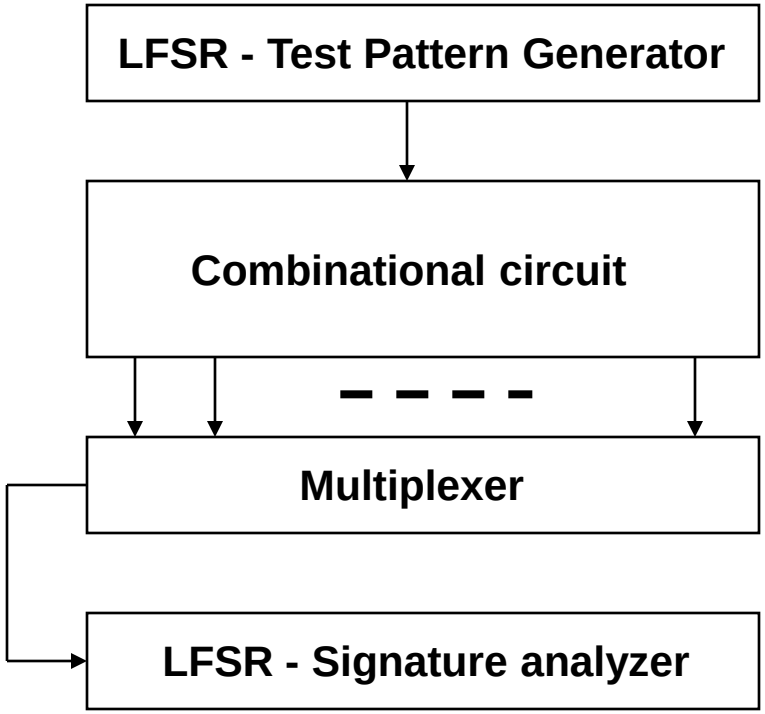
BIST: Signature Analysis

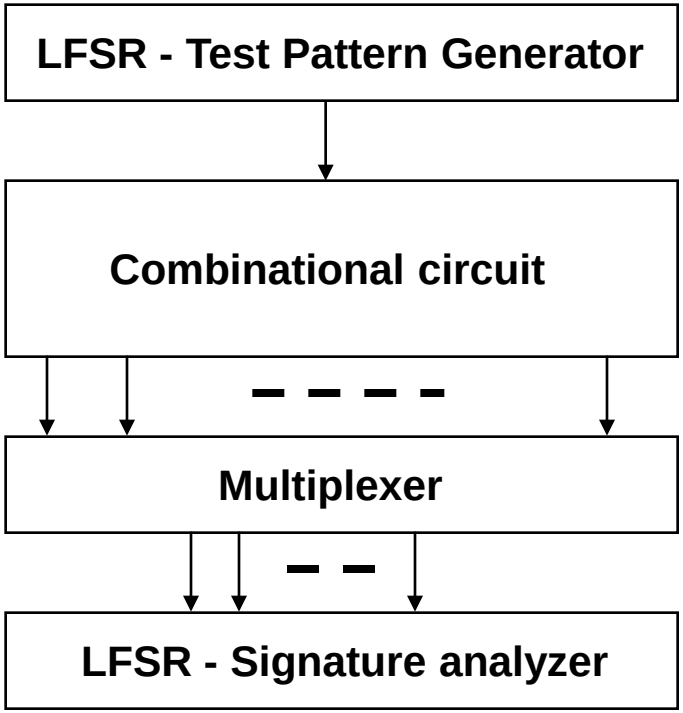
Parallel Signature Analyzer:



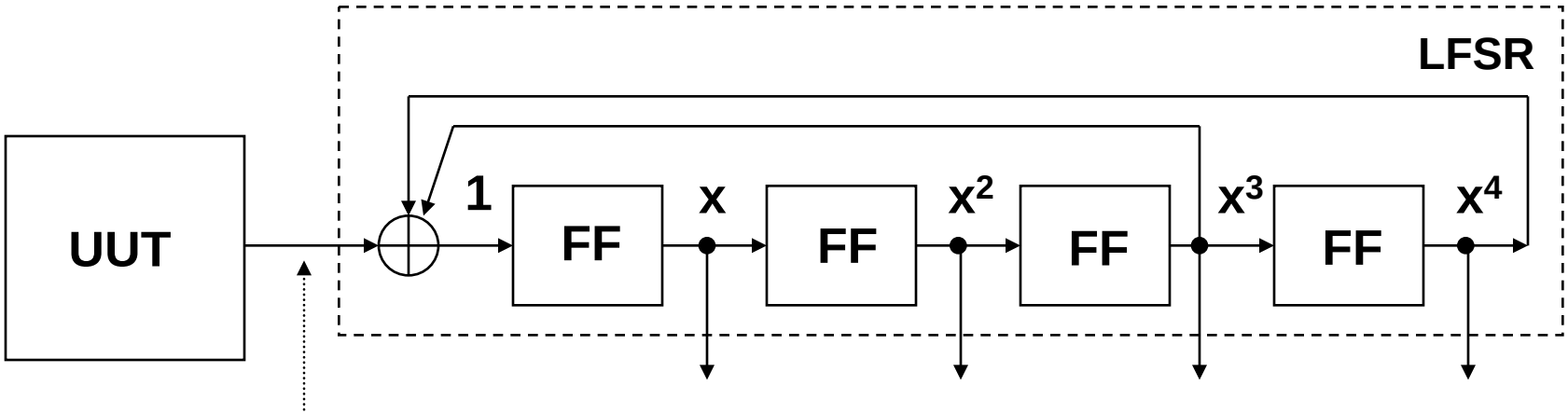
BIST: Signature Analysis

Signature calculating for multiple outputs:





BIST: Joining TPG and SA

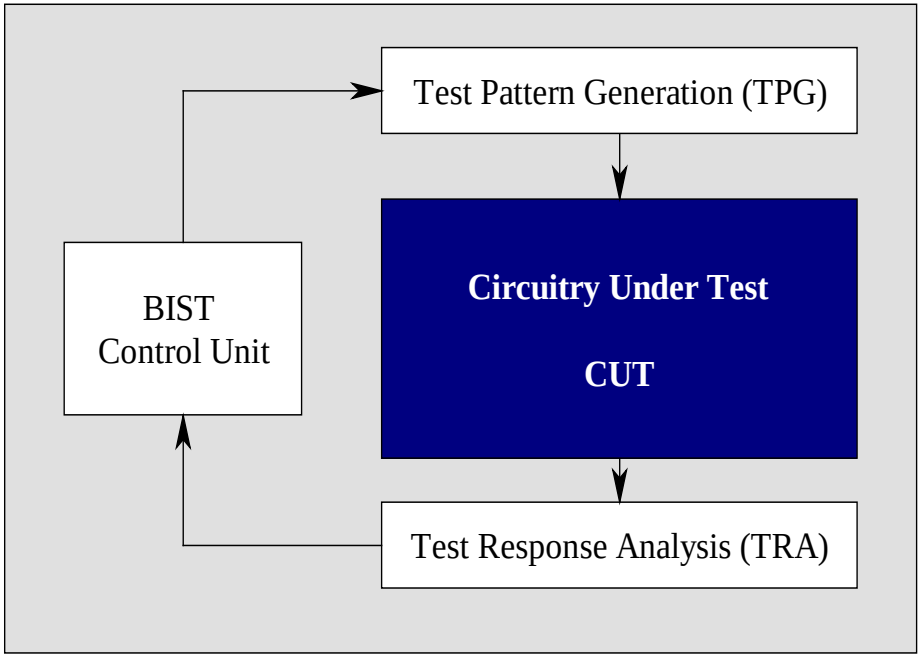


Response string for
Signature Analysis

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

BIST Architectures

General Architecture of BIST

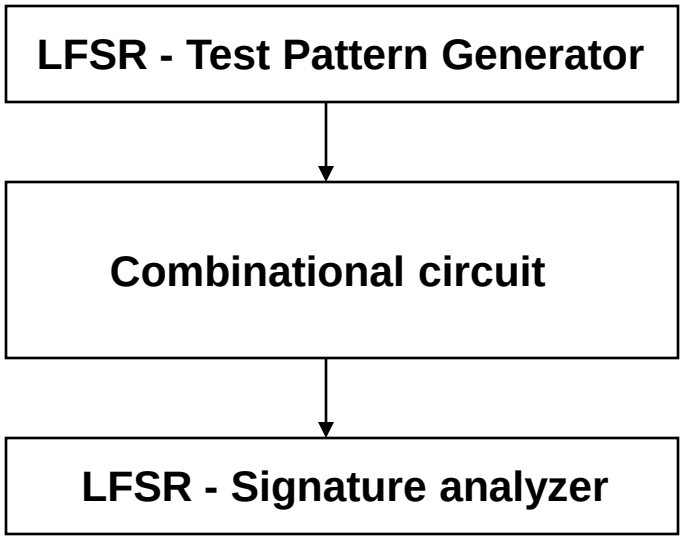


- **BIST components:**
  - **Test pattern generator (TPG)**
  - **Test response analyzer (TRA)**
  - **BIST controller**
- **A part of a system (*hardcore*) must be operational to execute a self-test**
- **At minimum the hardcore usually includes *power*, *ground*, and *clock* circuitry**
- **Hardcore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

BIST Architectures

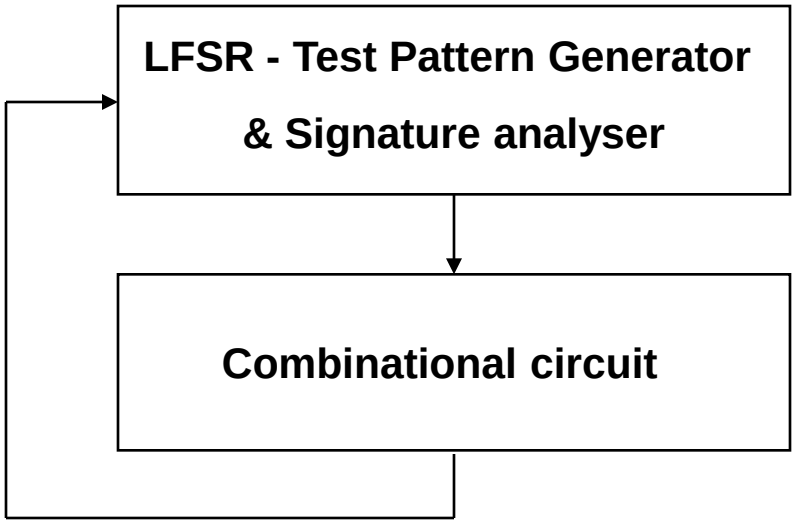
Test per Clock:

Disjoint TPG and SA: BILBO

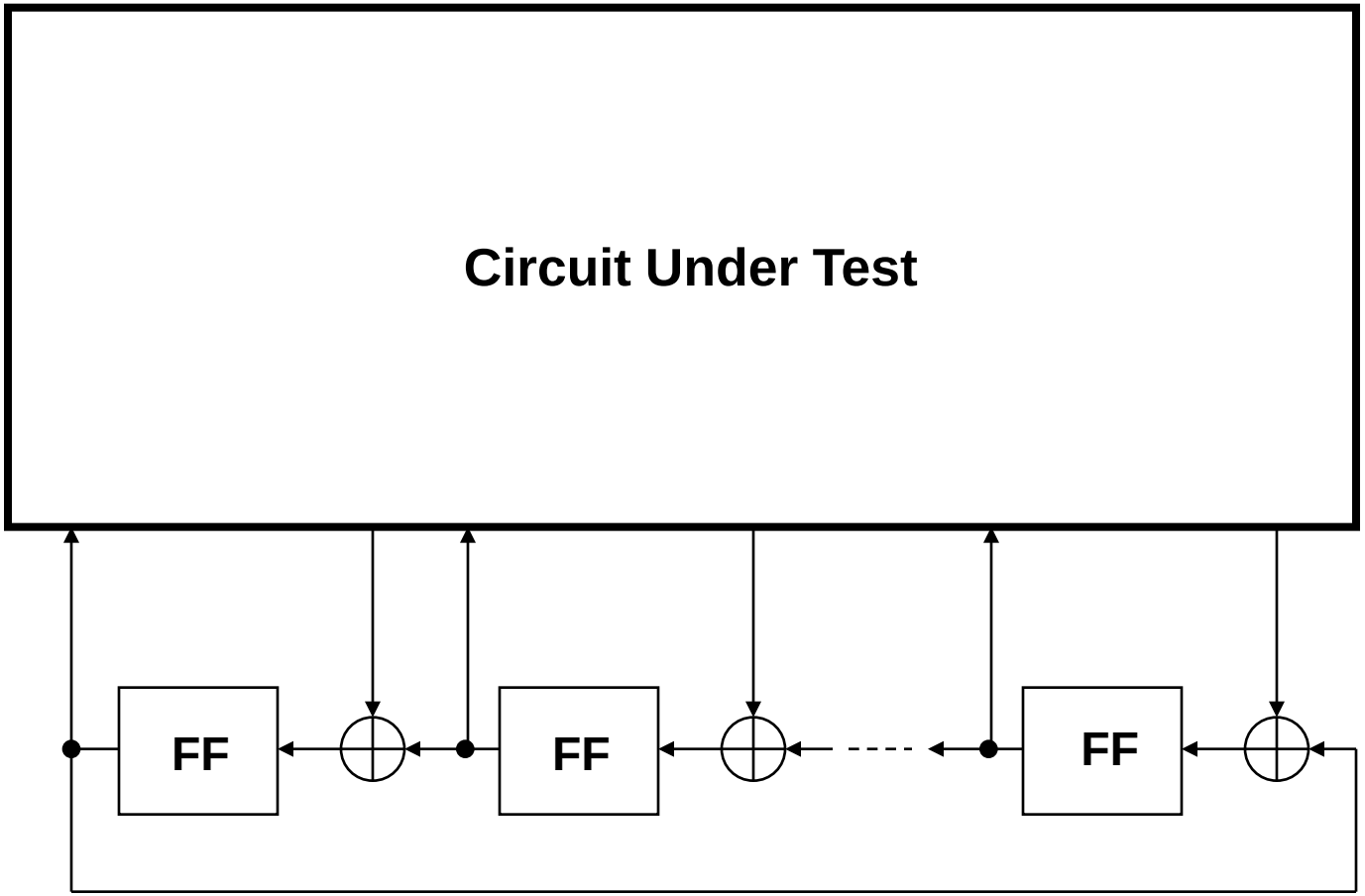


Joint TPG and SA:

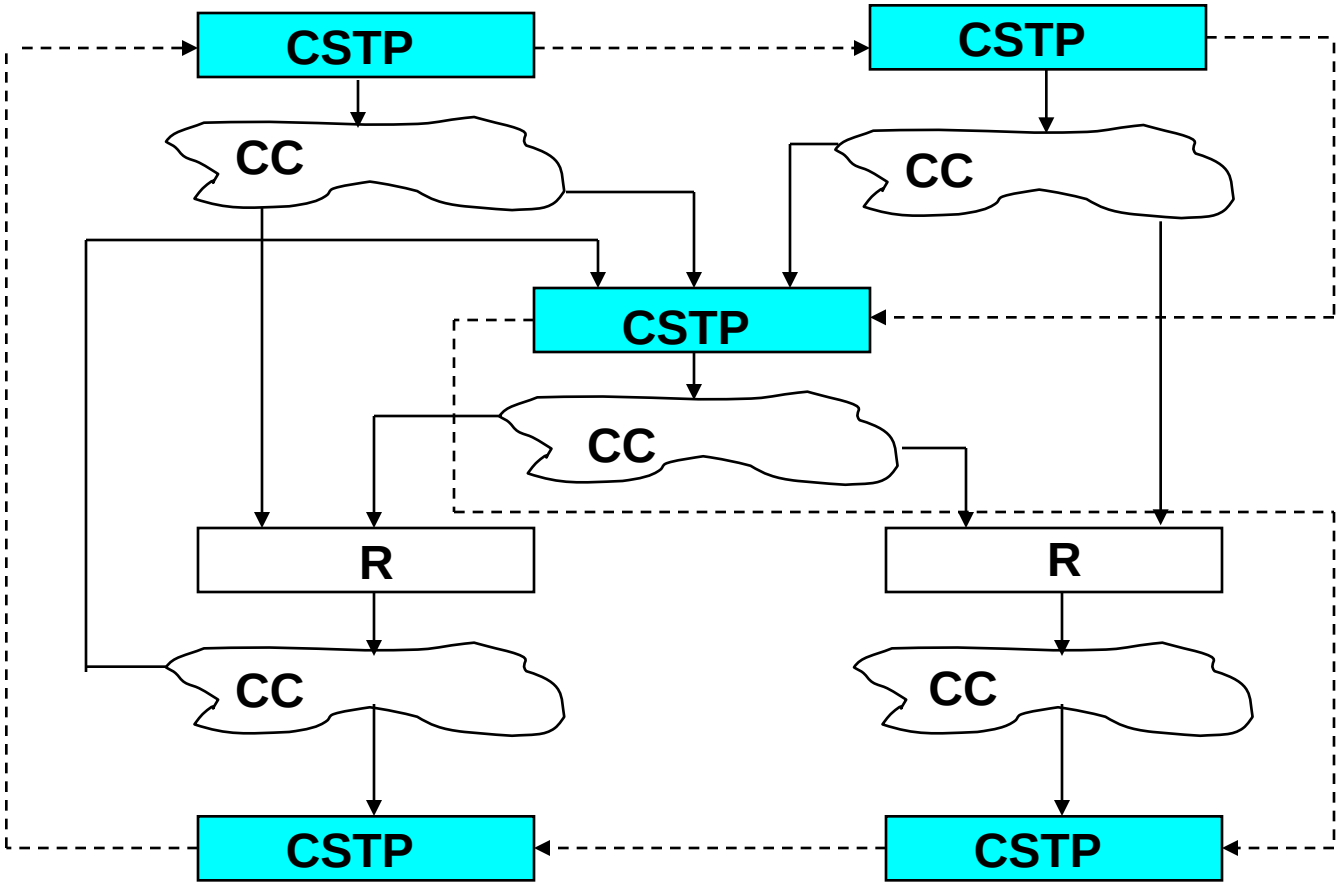
CSTP - Circular Self-Test Path:



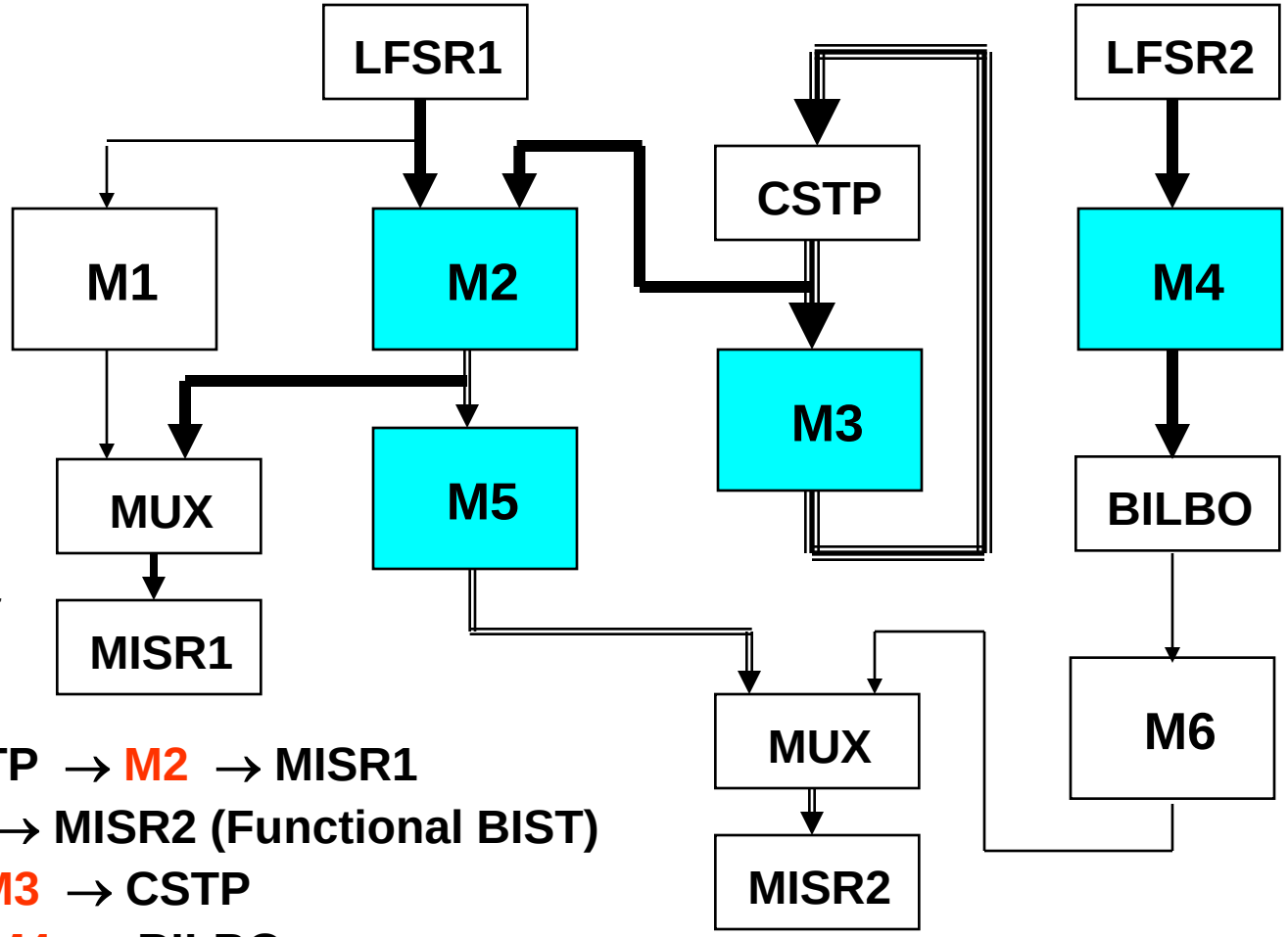
BIST: Circular Self-Test Architecture



BIST: Circular Self-Test Path



BIST Embedding Example



Concurrent testing:

LFSR, CSTP → M2 → MISR1

M2 → M5 → MISR2 (Functional BIST)

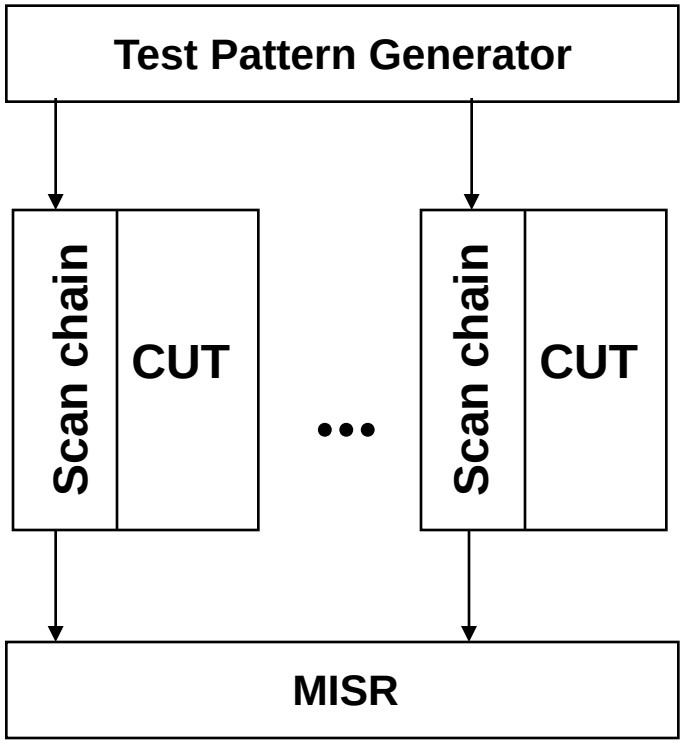
CSTP → M3 → CSTP

LFSR2 → M4 → BILBO

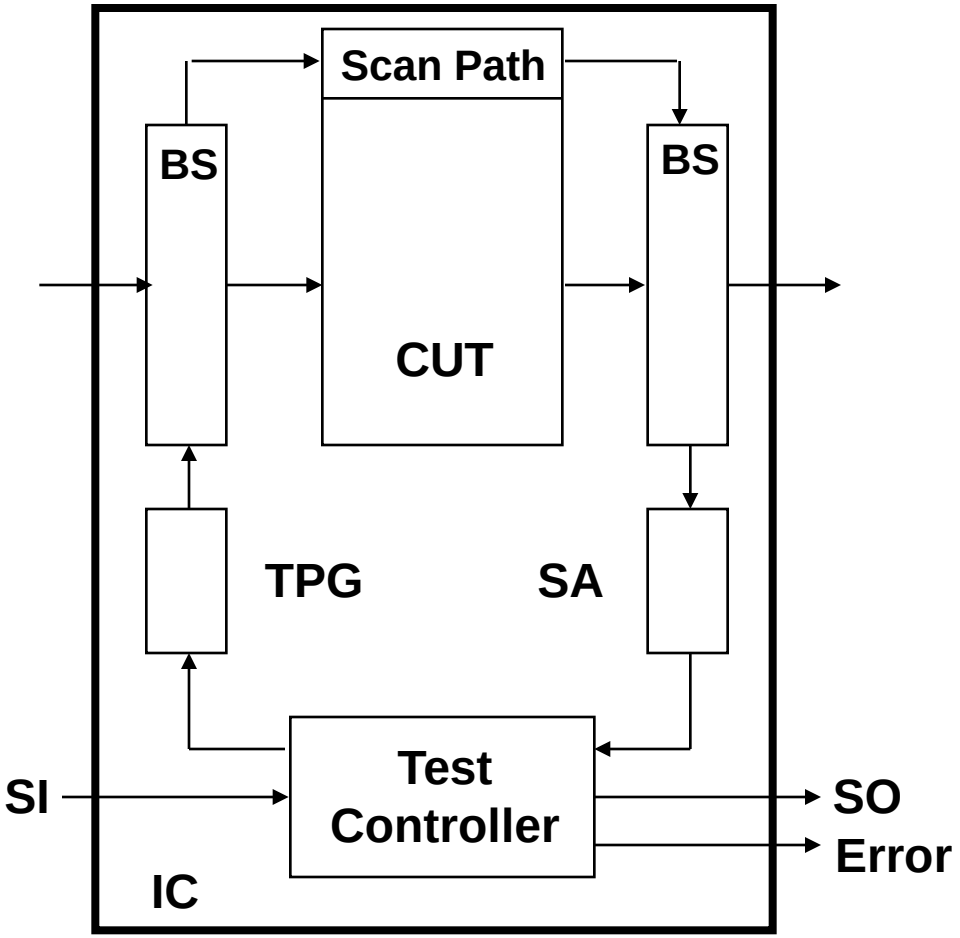
BIST Architectures

STUMPS:

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



LOCST: LSSD On-Chip Self-Test



Scan-Based BIST Architecture

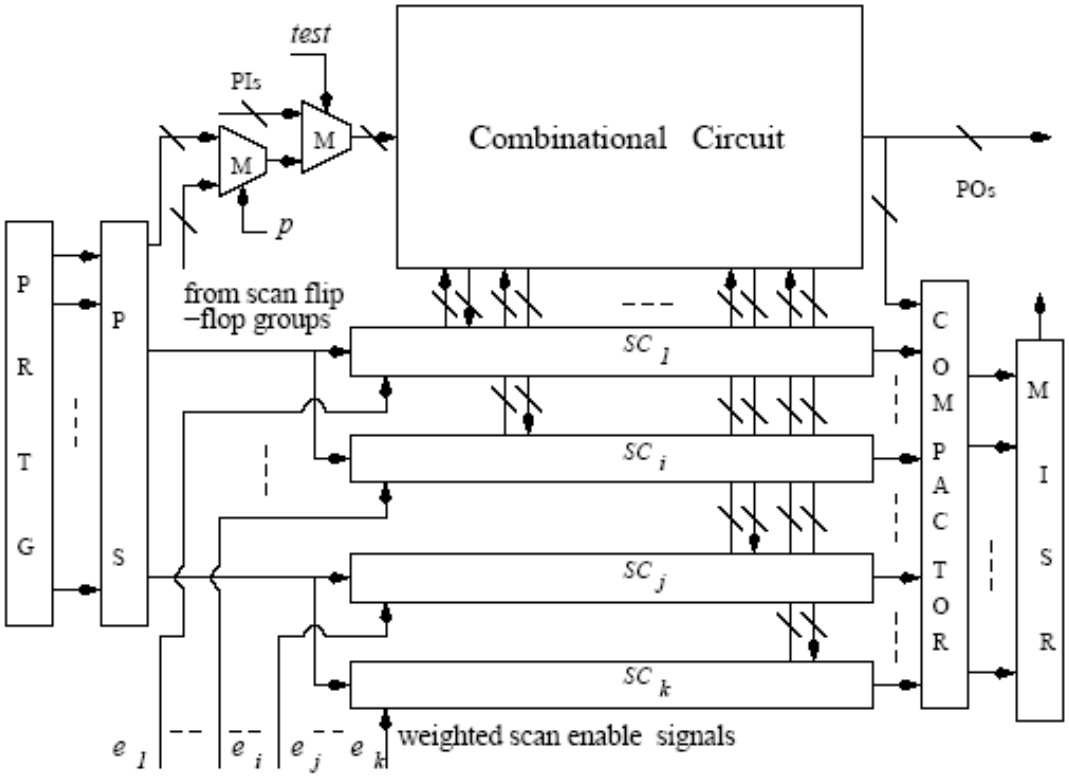


Figure 1: Scan-based BIST for n-detection with weighted scan-enable signals and scan forest.

PS – Phase shifter

Scan-Forest

Scan-Trees

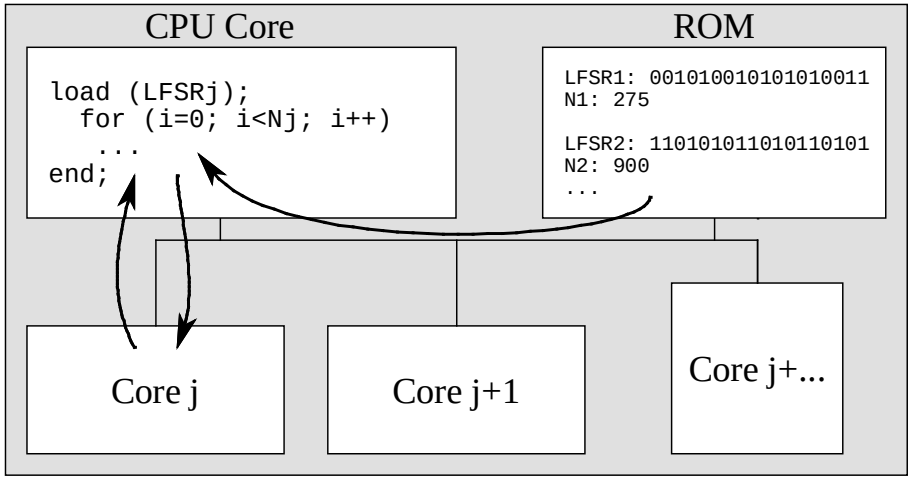
Scan-Segments (SC)

Weighted scan-enables for SS

Compactor - EXORs

Software BIST

Software based test generation:



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

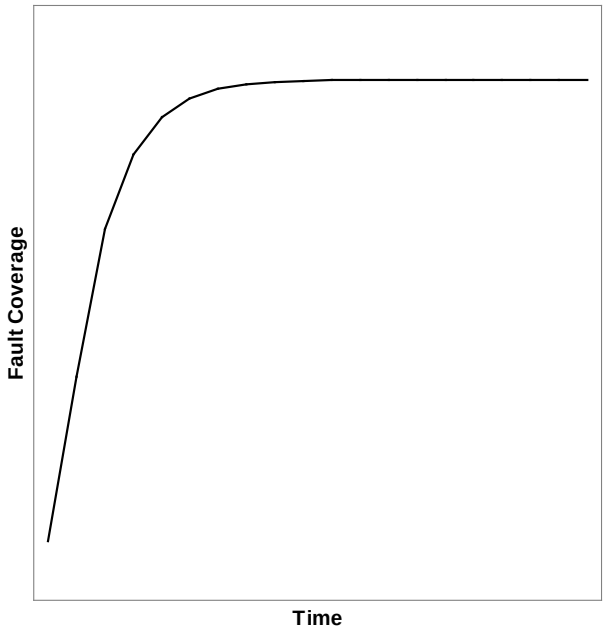
Software BIST is especially attractive to test SoCs, because of the availability of computing resources directly in the system (a typical SoC usually contains at least one processor core)

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

Problems with BIST

The main motivations of using random patterns are:

- low generation cost
- high initial efeciency

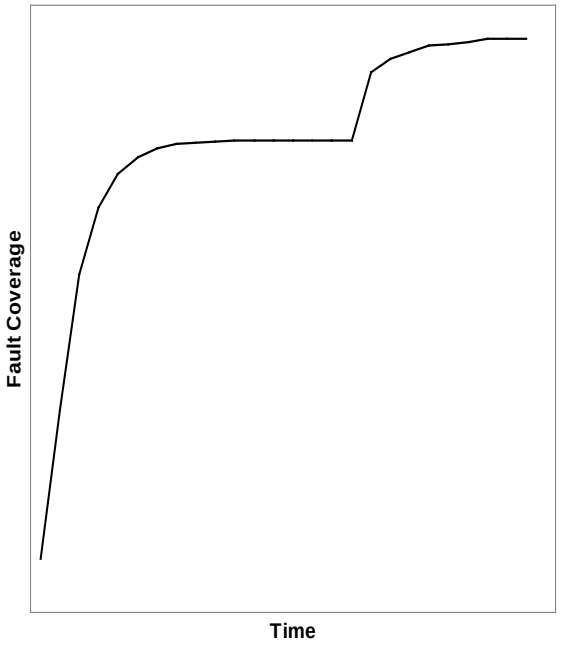


Problems:

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

Possible solutions

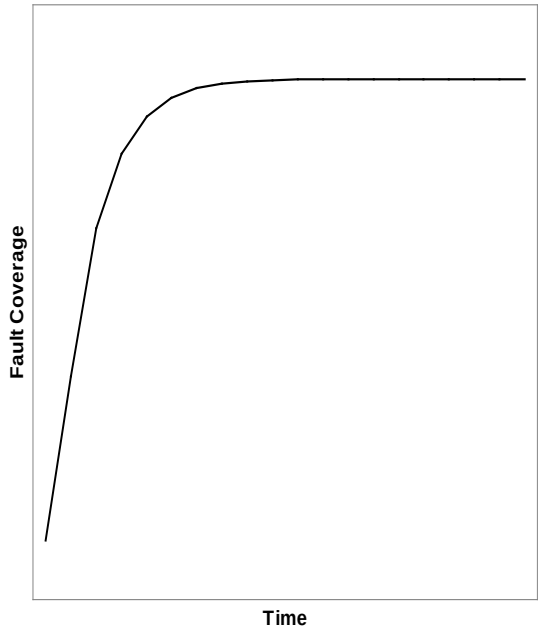
- Weighted pseudorandom test
- Combining pseudorandom test with deterministic test
  - Multiple seed
  - Bit flipping
- Hybrid BIST



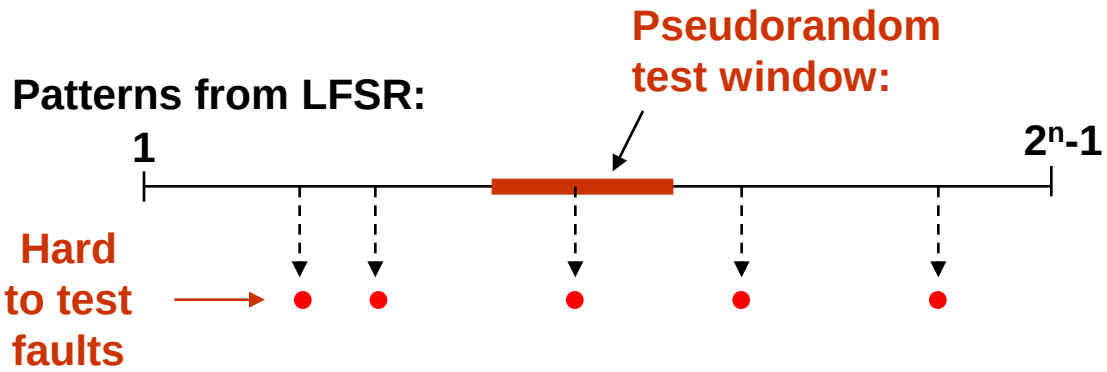
Problems with BIST: Hard to Test Faults

The main motivations of using random patterns are:

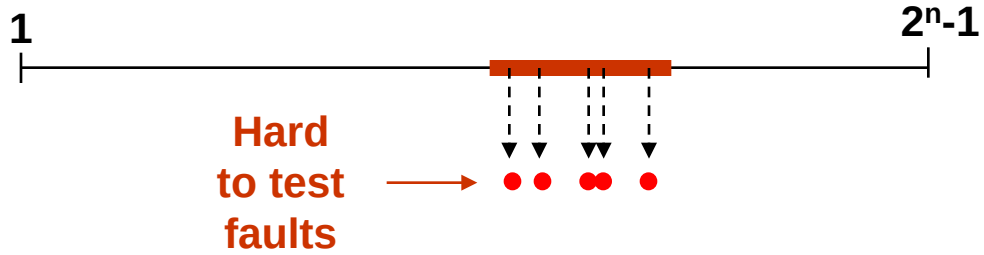
- low generation cost
- high initial efeciency



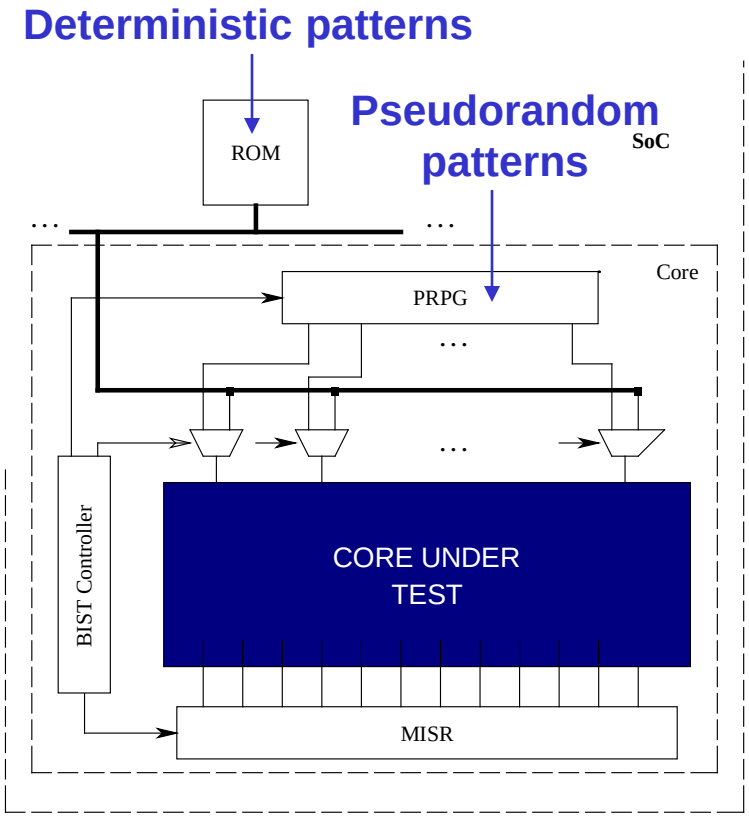
Problem: Low fault coverage



Dream solution: Find LFSR such that:



Hybrid Built-In Self-Test

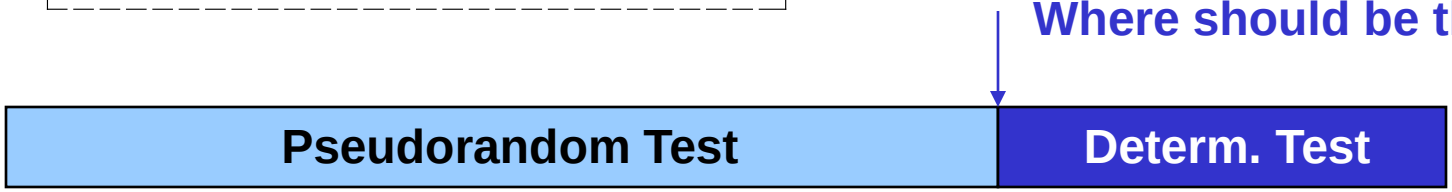


Hybrid test set contains pseudorandom and deterministic vectors

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

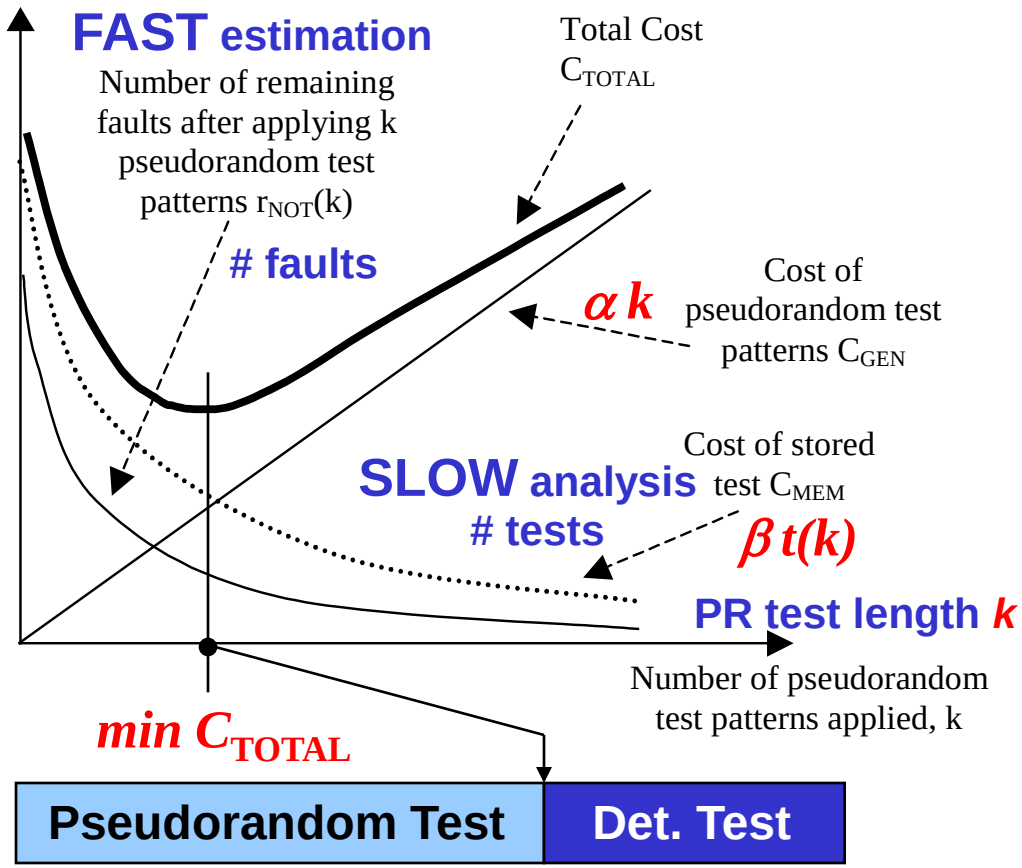
Optimization problem:

Where should be this breakpoint?



Optimization of Hybrid BIST

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



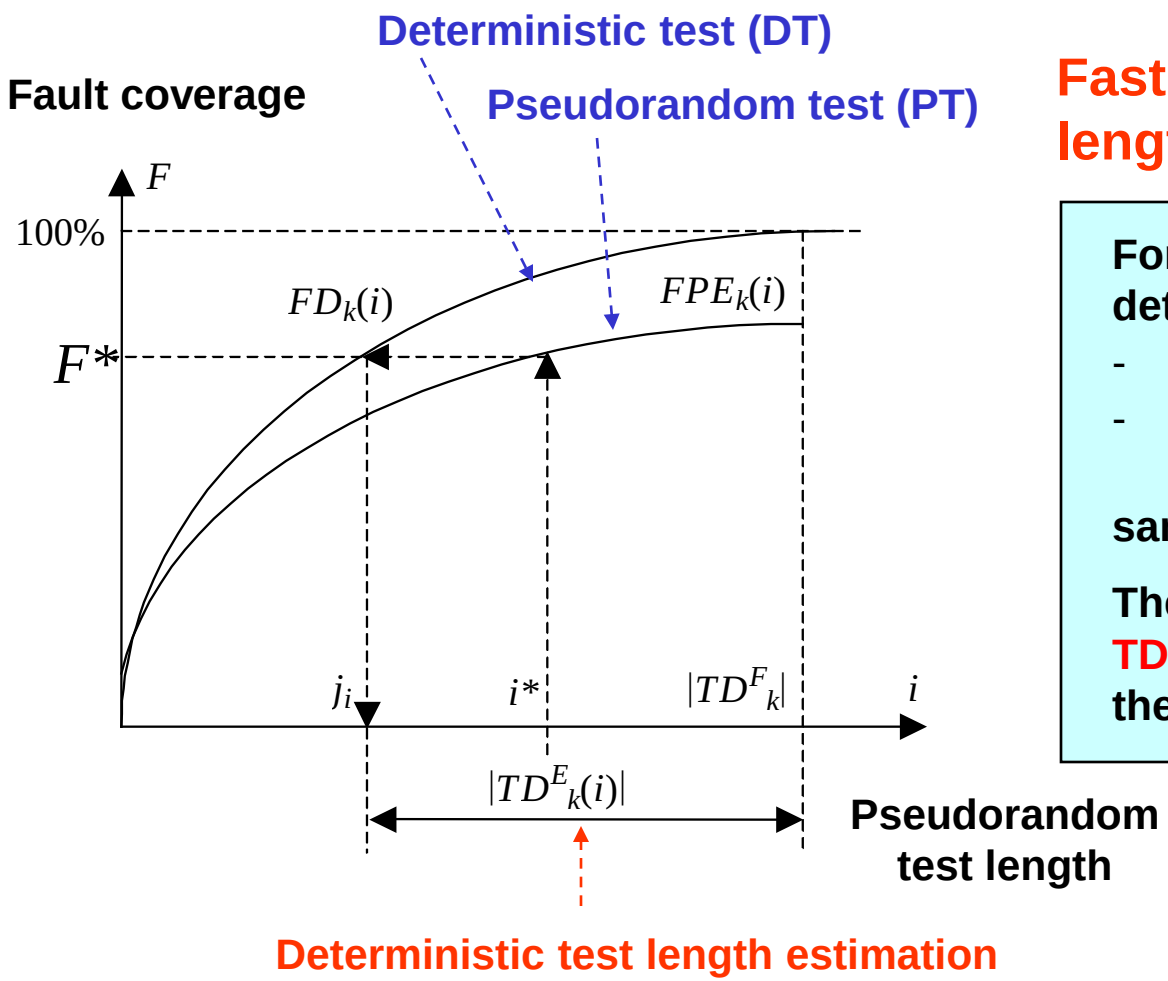
PR test length

faults not detected

tests needed

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

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 used for the same fault coverage

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

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

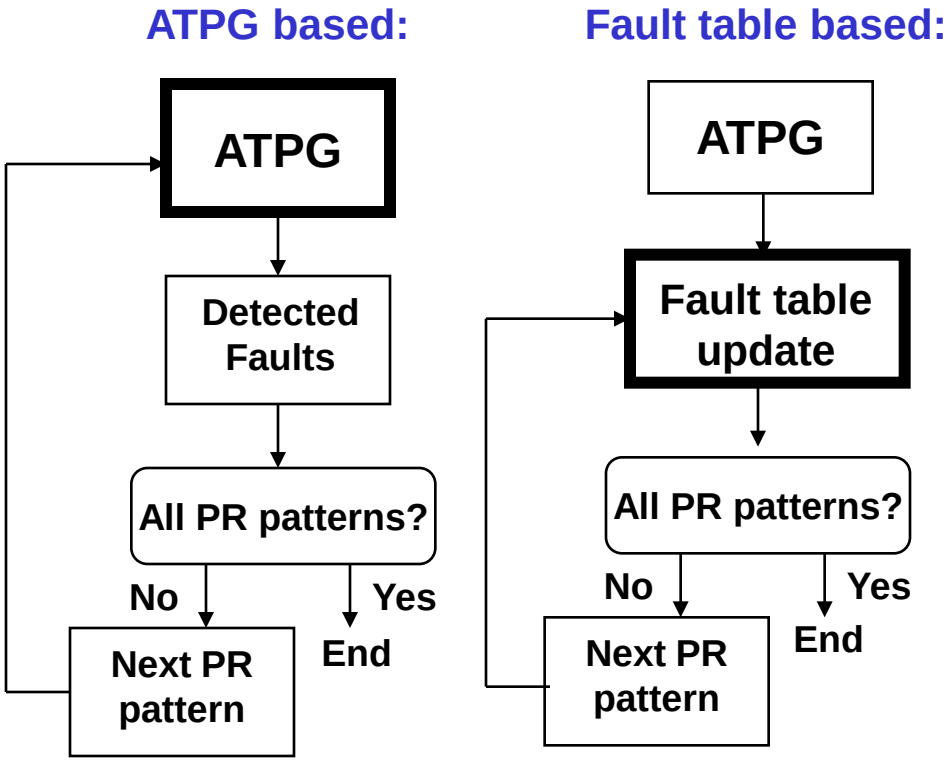
Fault table based approach

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

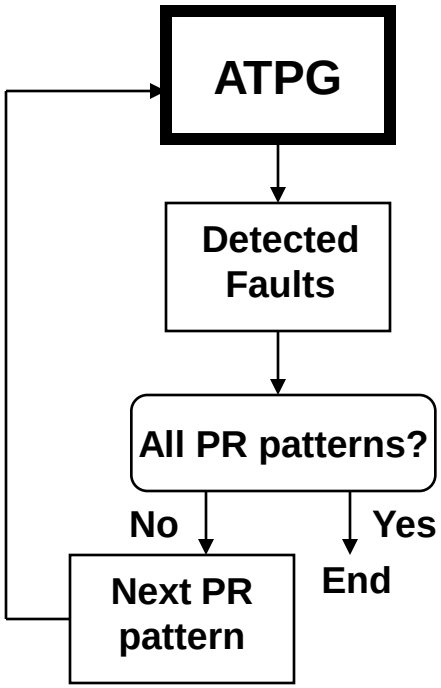


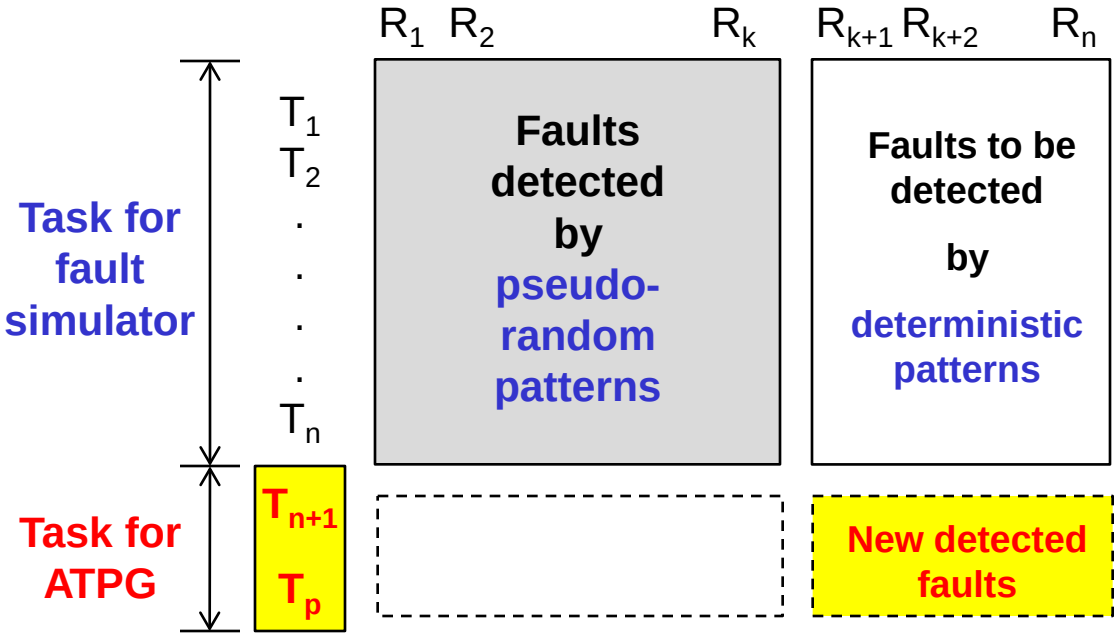
Calculation of the Deterministic Test Cost

ATPG based approach

For each breakpoint of P-sequence, ATPG is used

ATPG based:





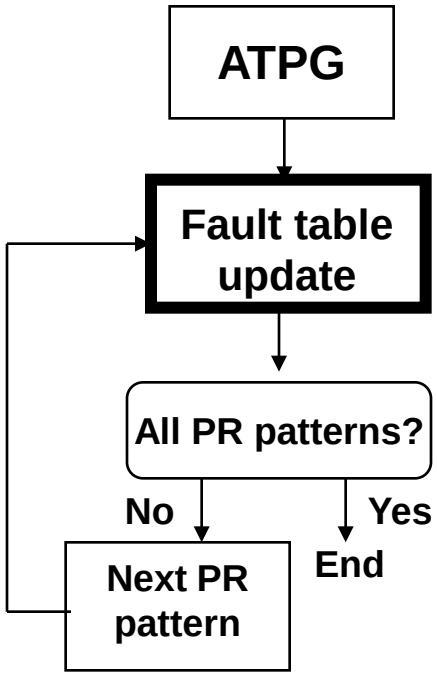
Calculation of the Deterministic Test Cost

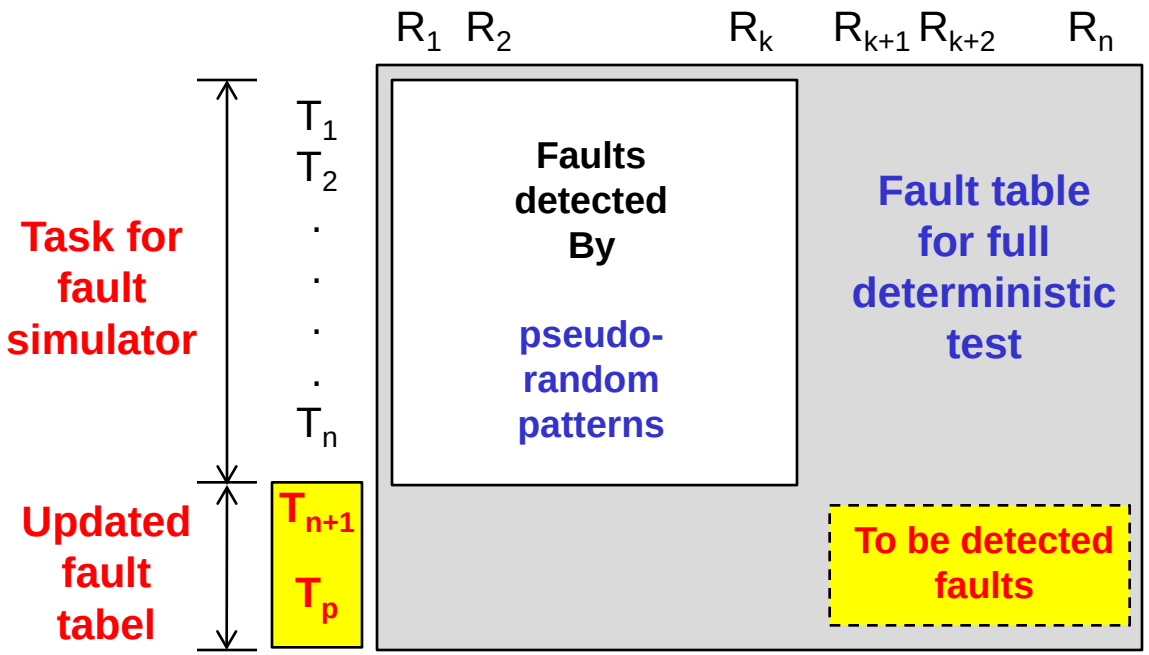
Fault table based approach

A deterministic test set with fault table is calculated

For each breakpoint of P-sequence, the fault table is updated

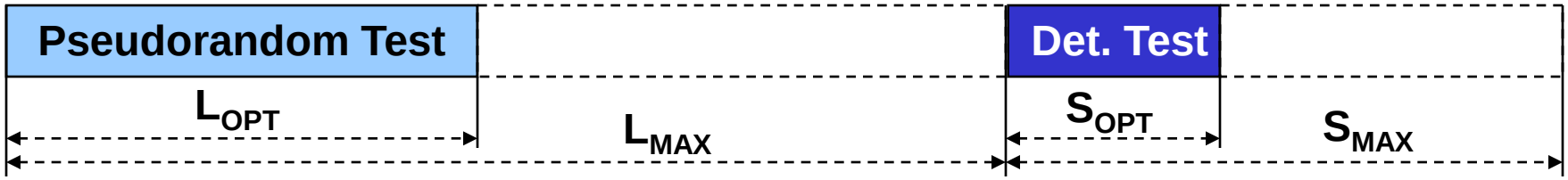
Fault table based:





Experimental Data: HBIST Optimization

Finding optimal brakepoint in the pseudorandom sequence:



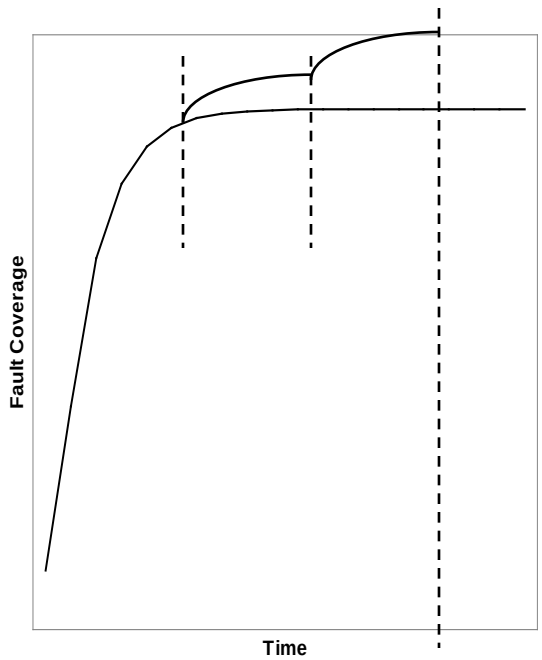
Optimized hybrid test process:

Circuit	L_{MAX}	L_{OPT}	S_{MAX}	S_{OPT}	B_k	C_{TOTAL}
C432	**780**	**91**	**80**	**21**	**4**	**186**
C499	**2036**	**78**	**132**	**60**	**6**	**386**
C880	**5589**	**121**	**77**	**48**	**8**	**481**
C1355	**1522**	**121**	**126**	**52**	**6**	**388**
C1908	**5803**	**105**	**143**	**123**	**5**	**612**
C2670	**6581**	**444**	**155**	**77**	**30**	**26867**
C3540	**8734**	**297**	**211**	**110**	**7**	**889**
C5315	**2318**	**711**	**171**	**12**	**23**	**985**
C6288	**210**	**20**	**45**	**20**	**4**	**100**
C7552	**18704**	**583**	**267**	**61**	**51**	**2161**

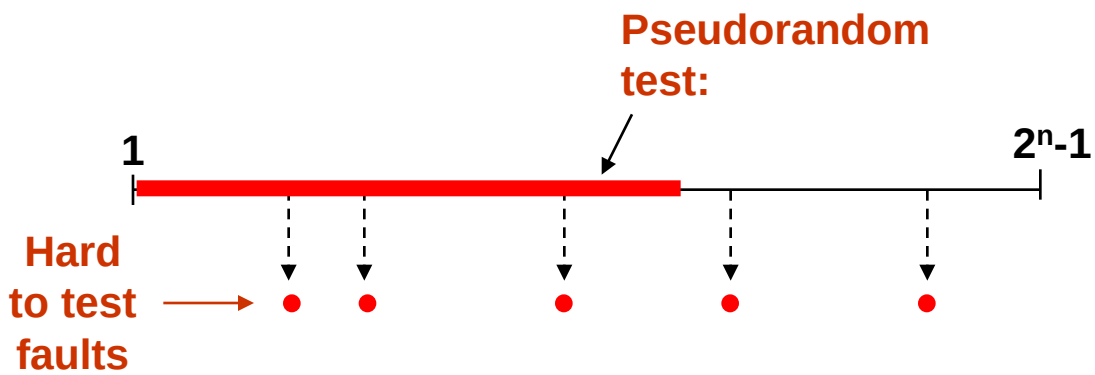
Hybrid BIST with Reseeding

The motivation of using random patterns is:

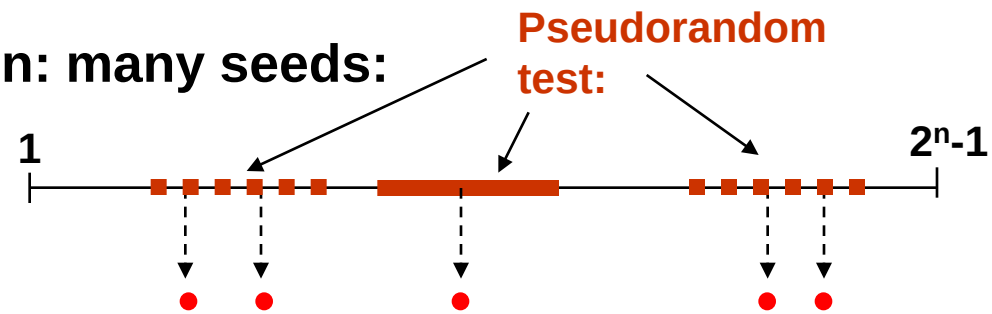
- low generation cost
- high initial efeciency



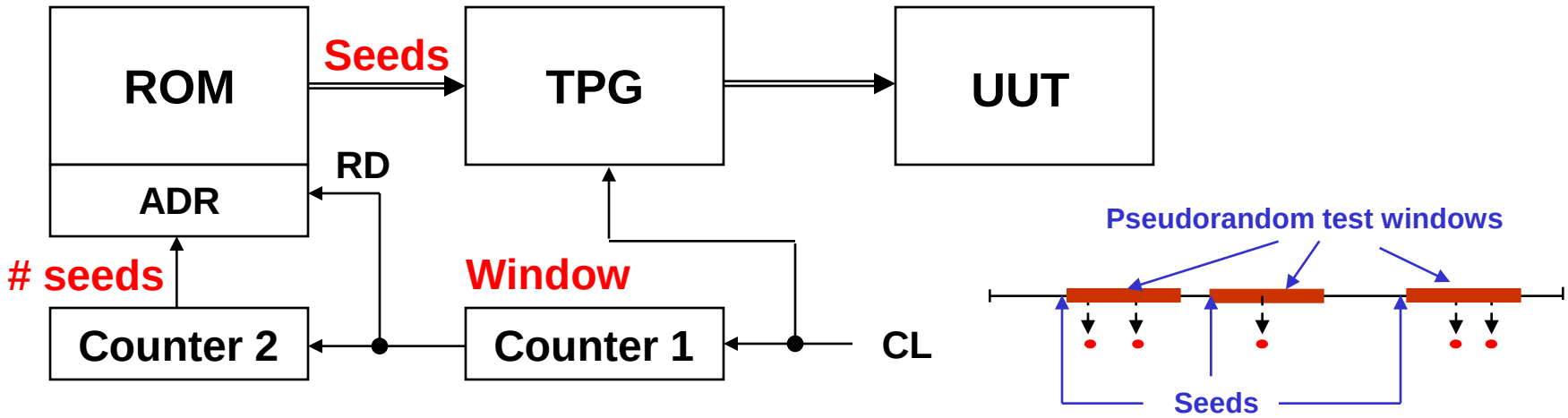
Problem: low fault coverage → long PR test



Solution: many seeds:

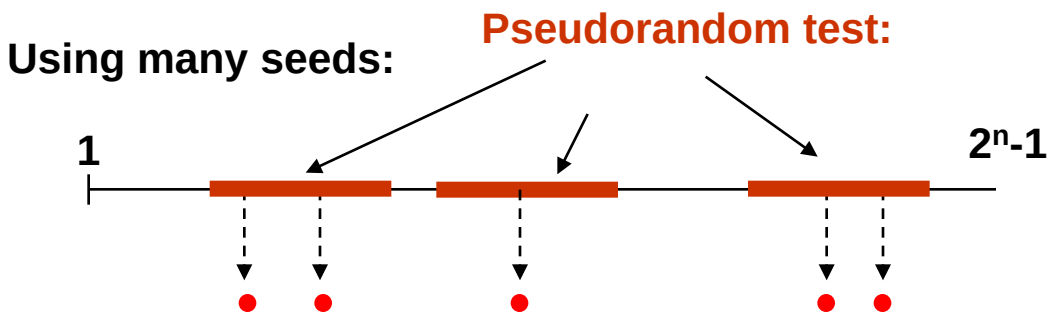


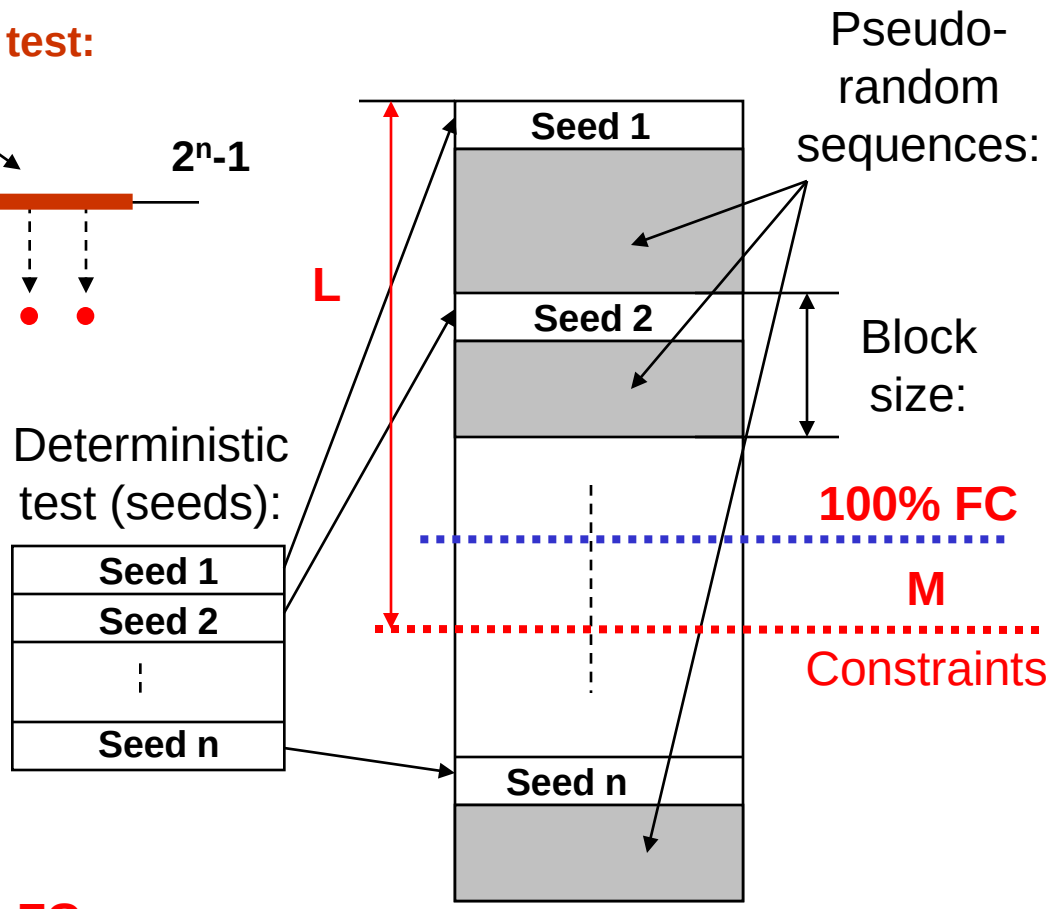
Store-and-Generate Test Architecture



- ROM contains test patterns for 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} – **beginning of the new window**

HBIST Optimization Problem





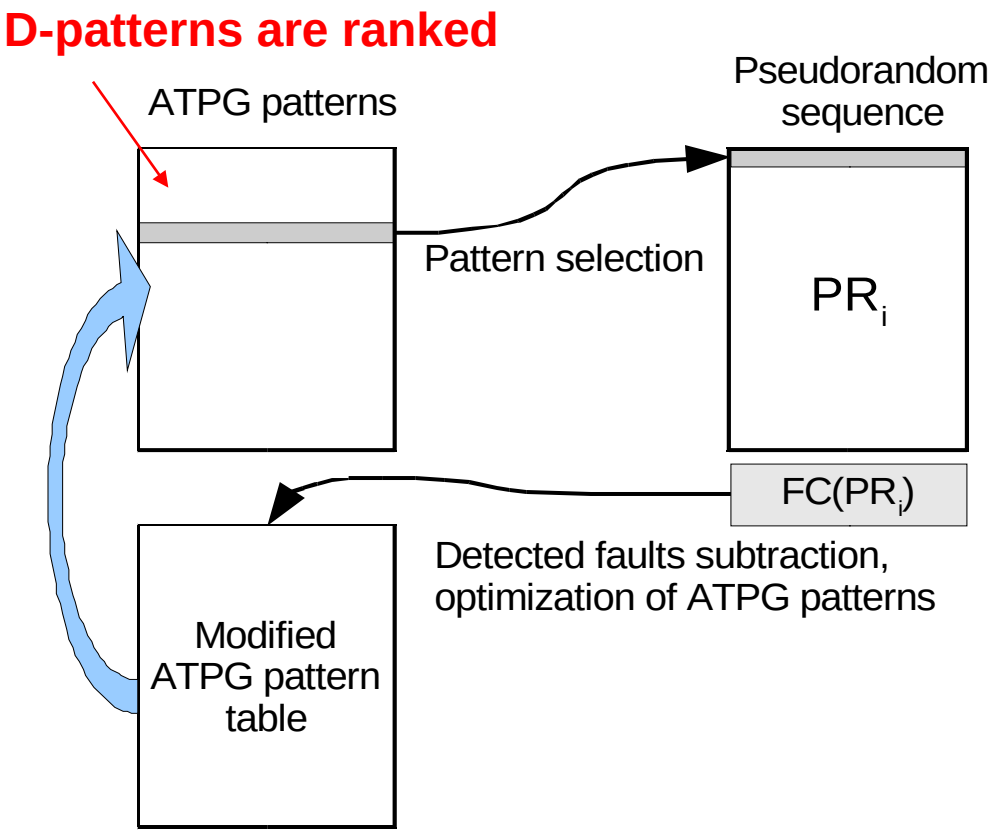
Problems:

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

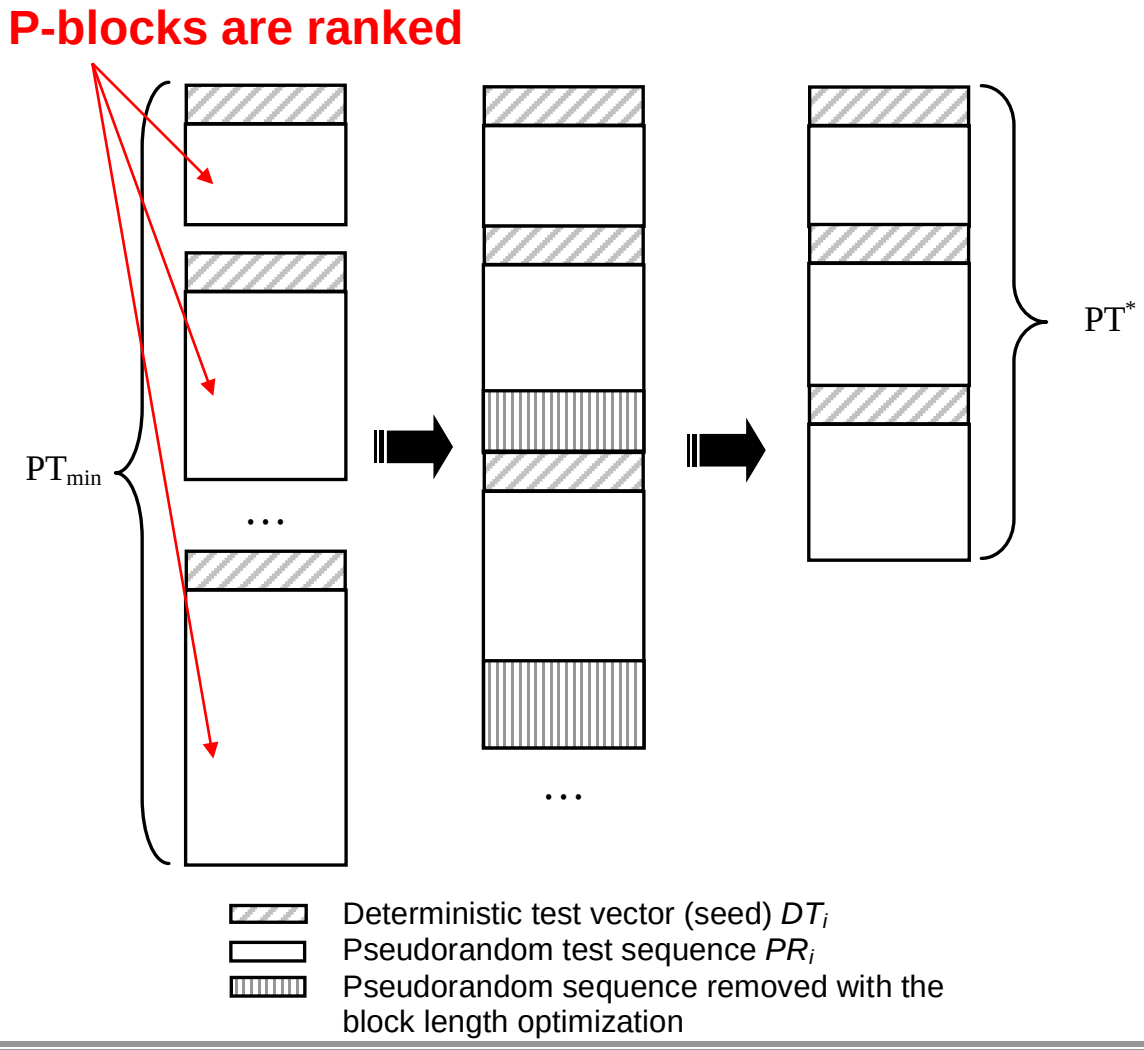
Deterministic test patterns with 100% quality are generated by ATPG

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



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 includeed into sequence and updated

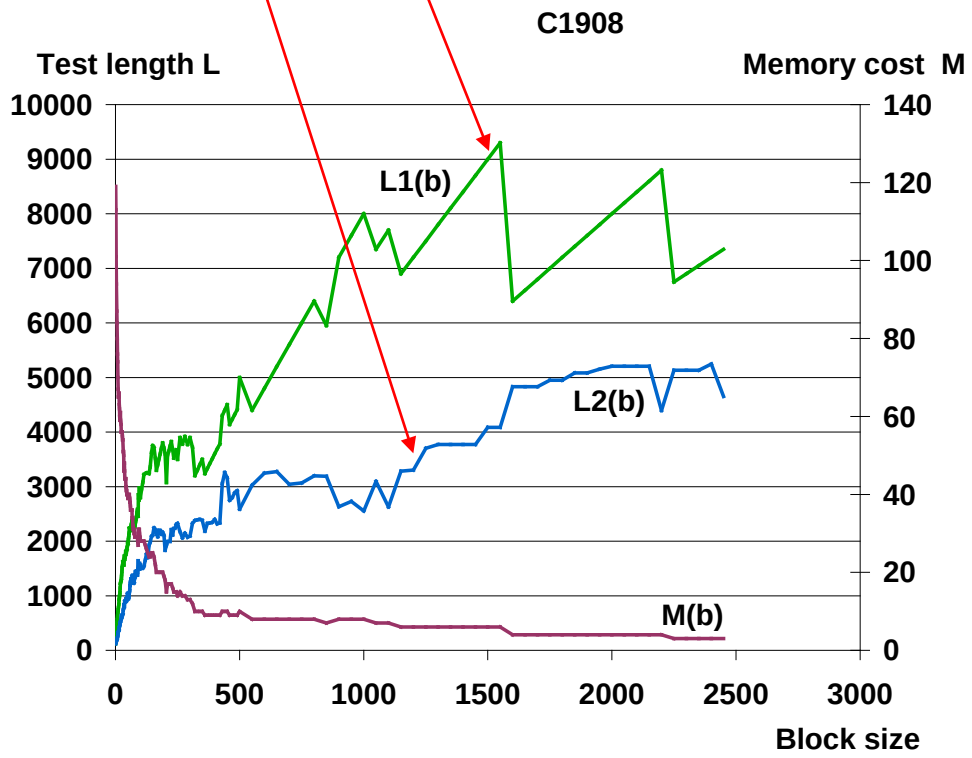
The procedure ends when 100% fault coverage is achieved

Cost Curves for Hybrid BIST with Reseeding

Two possibilities for reseeding:

Constant block length (less HW overhead)

Dynamic block length (more HW overhead)



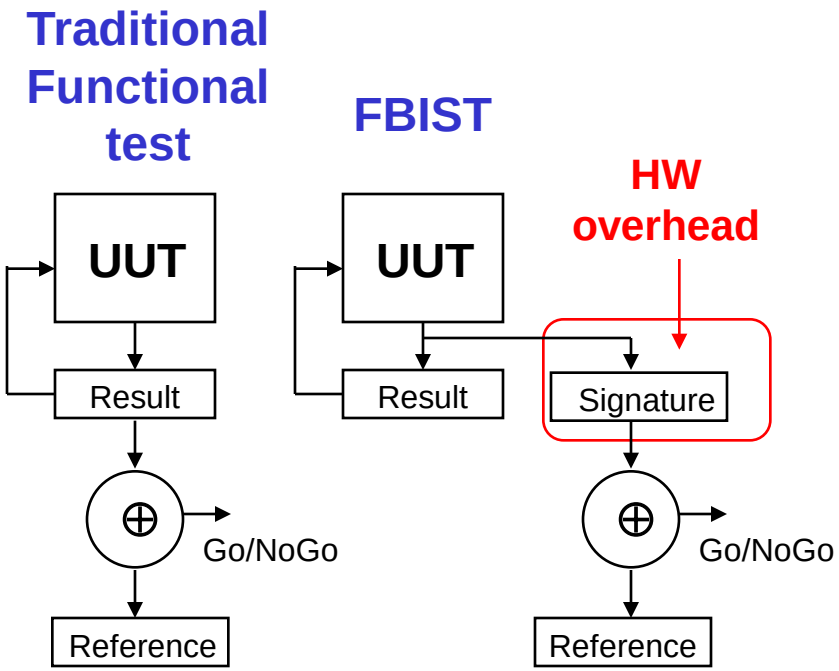
Functional Self-Test

- 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

Functional BIST Quality

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

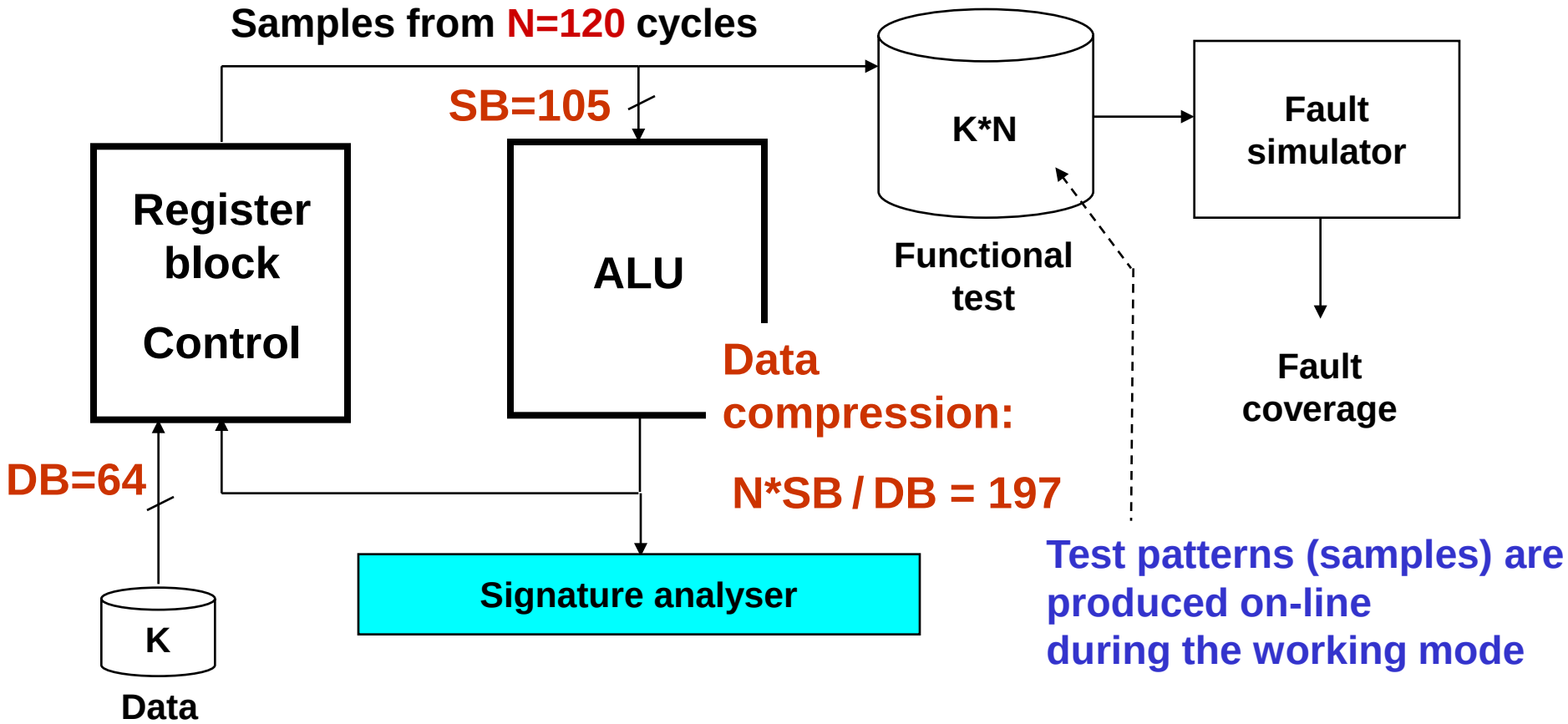


FBIST: collection and analysis of samples during the working mode

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

Example: Functional BIST

Functional BIST quality analysis

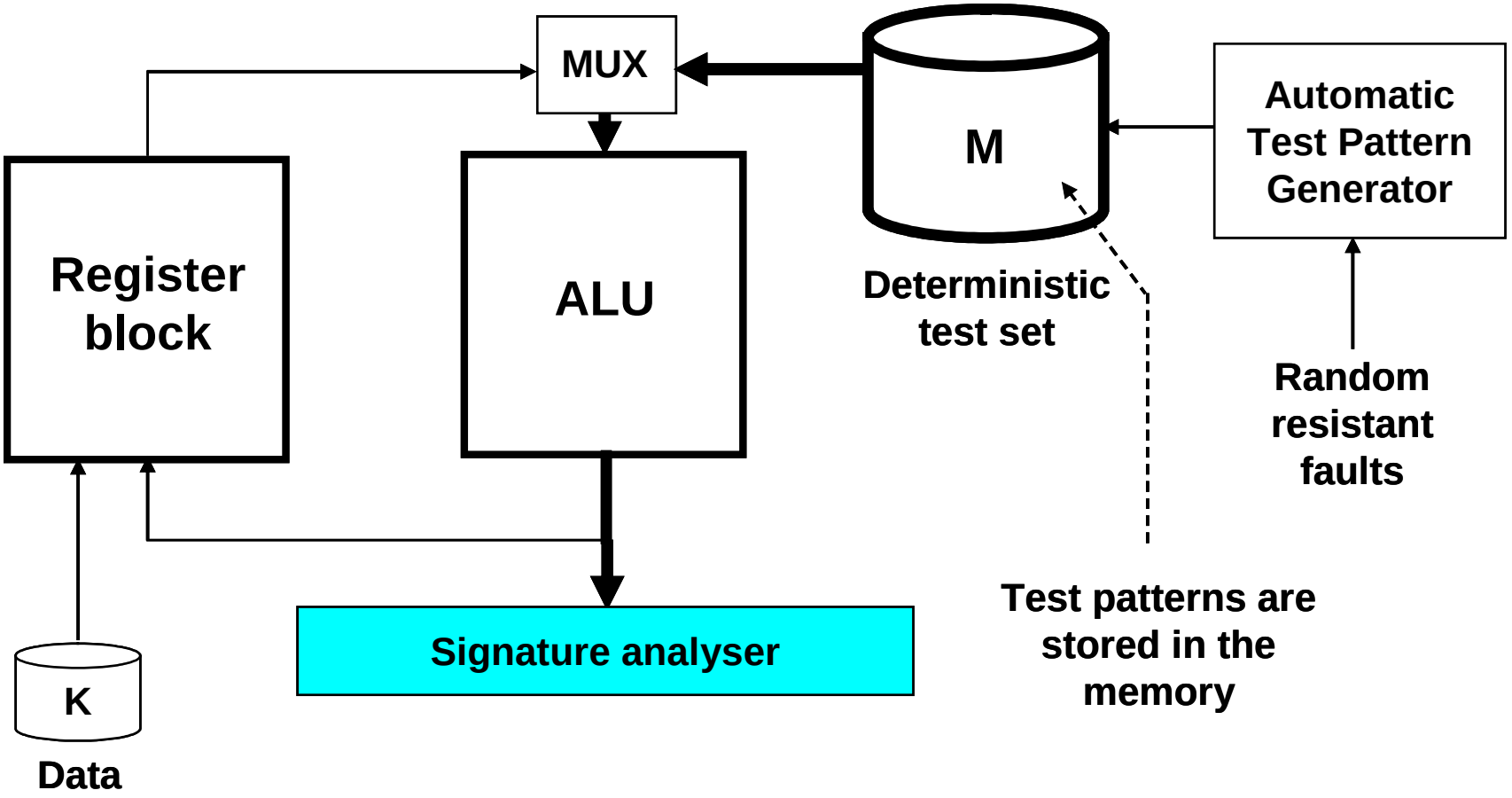


Hybrid Functional BIST

- To improve the quality of FBIST we introduce the method of Hybrid FBIST
- The idea of Hybrid FBIST consists in using for test purposes 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)
- Tradeoff should be found between
  - the testing time and
  - the HW/SW overhead cost

Functional Hybrid Self-Test

Functional BIST implementation



Cost Functions for Hybrid Functional BIST

Total cost:

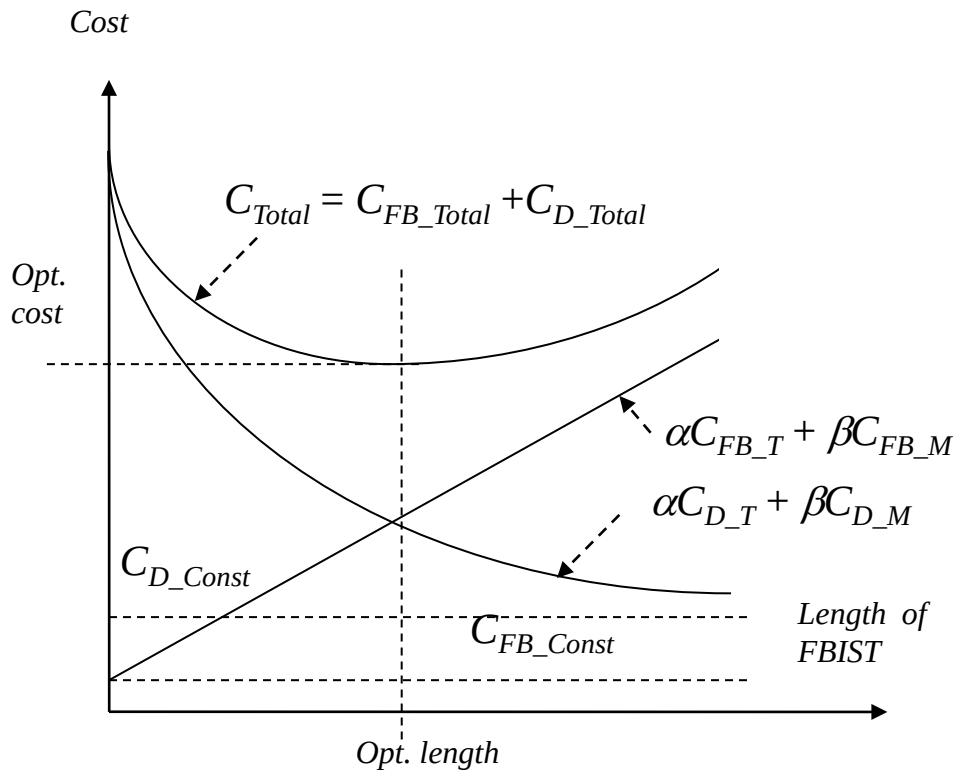
$$C_{Total} = C_{FB\_Total} + C_{D\_Total}$$

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

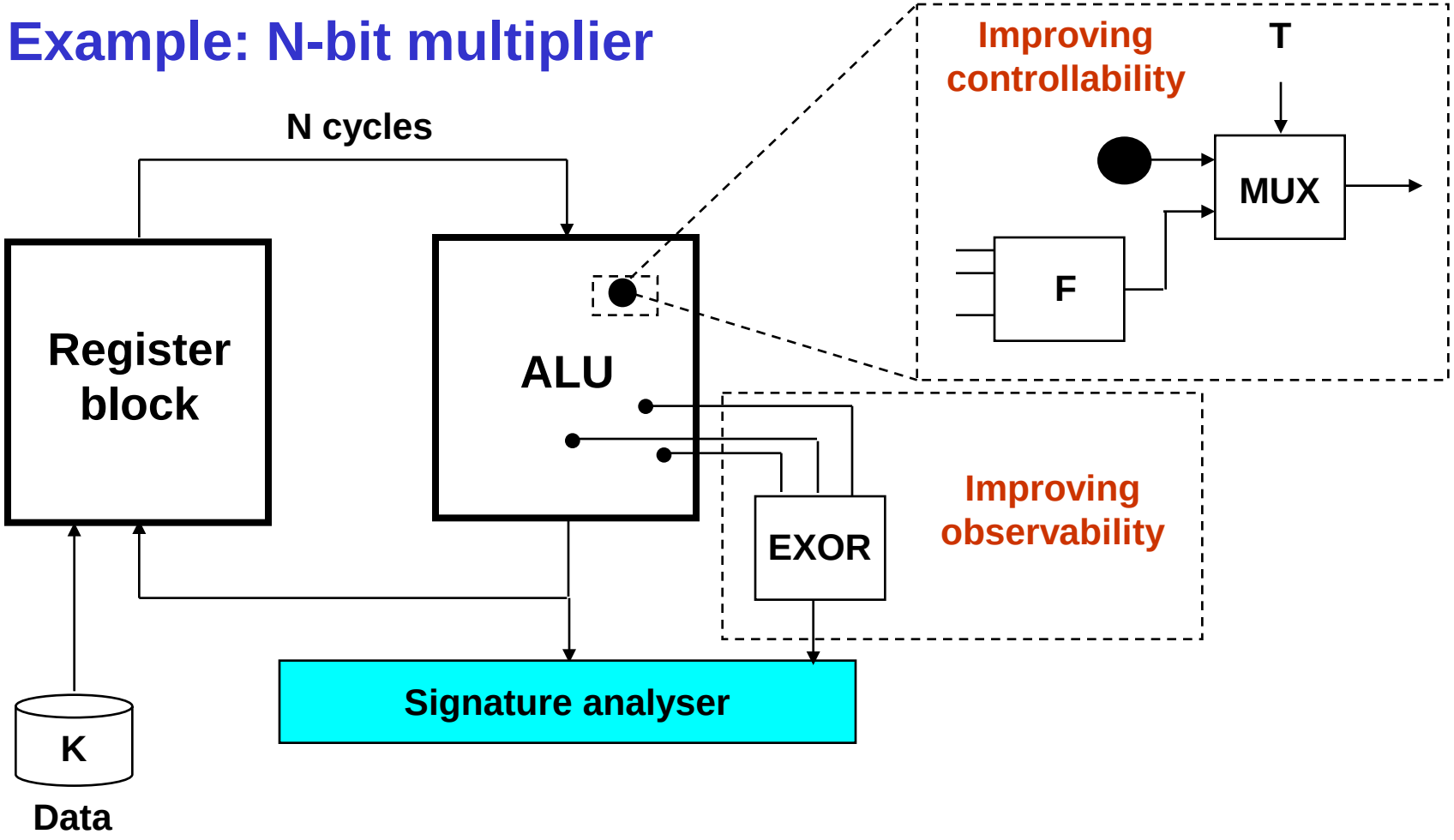
C_{FB_T}, C_{D_T} - testing time cost

α, β - weights of time and memory expenses

Problem: minimize C_{Total}

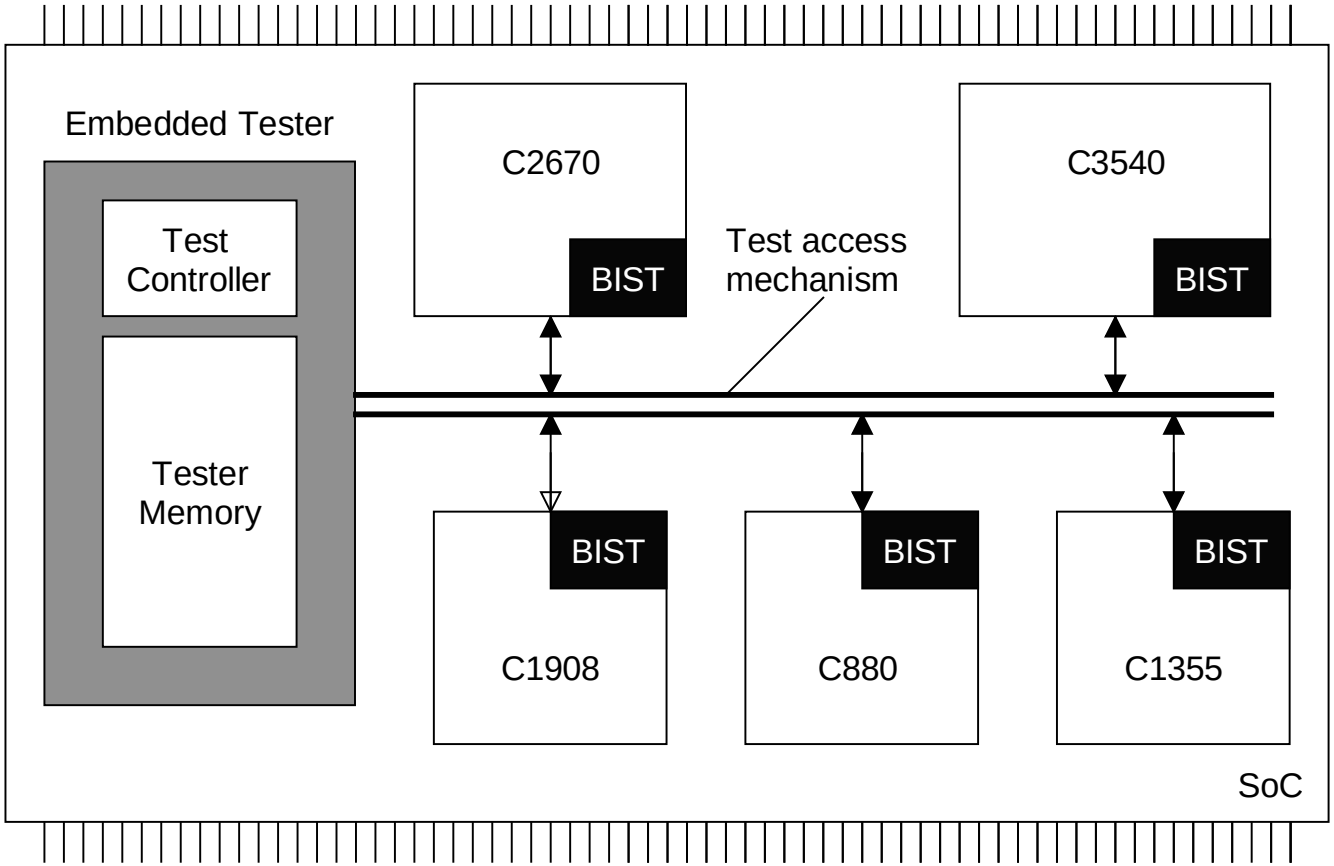
Functional Self-Test with DFT

Example: N-bit multiplier

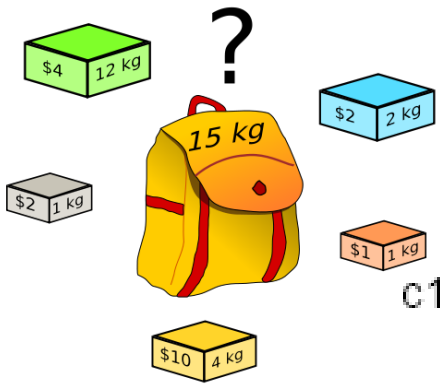


Hybrid BIST for Multiple Cores

Embedded tester for testing multiple cores



Hybrid BIST for Multiple Cores

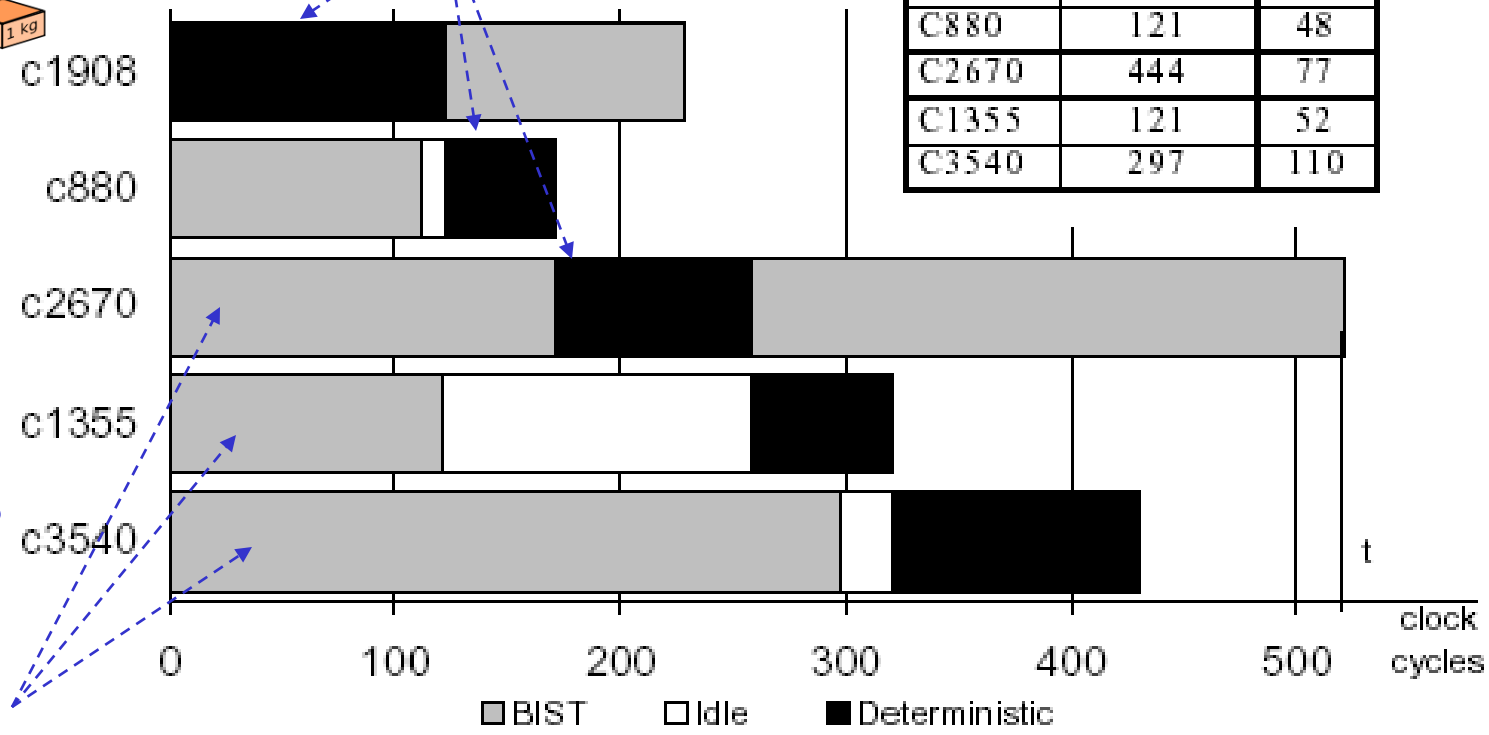


How to pack knapsack?

How to compress the test sequence?

Deterministic test (DT)

Pseudorandom test (PT)

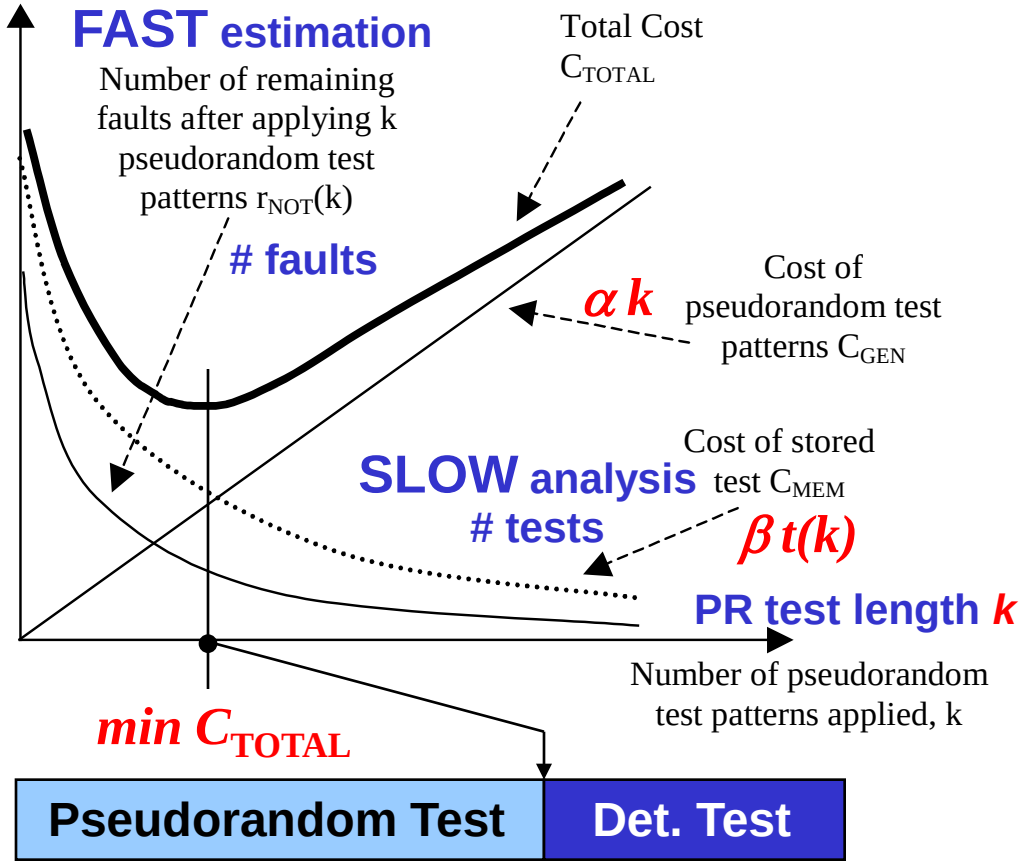


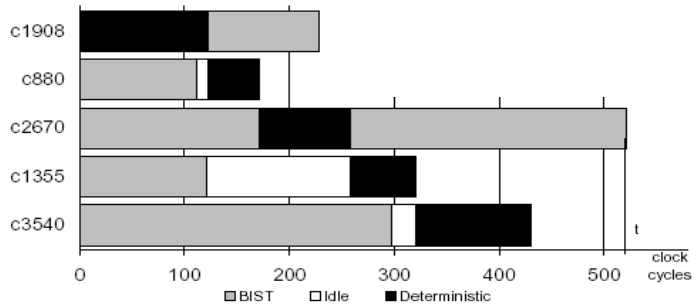
The optimal test set for each core

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

Multi-Core Hybrid BIST Optimization

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



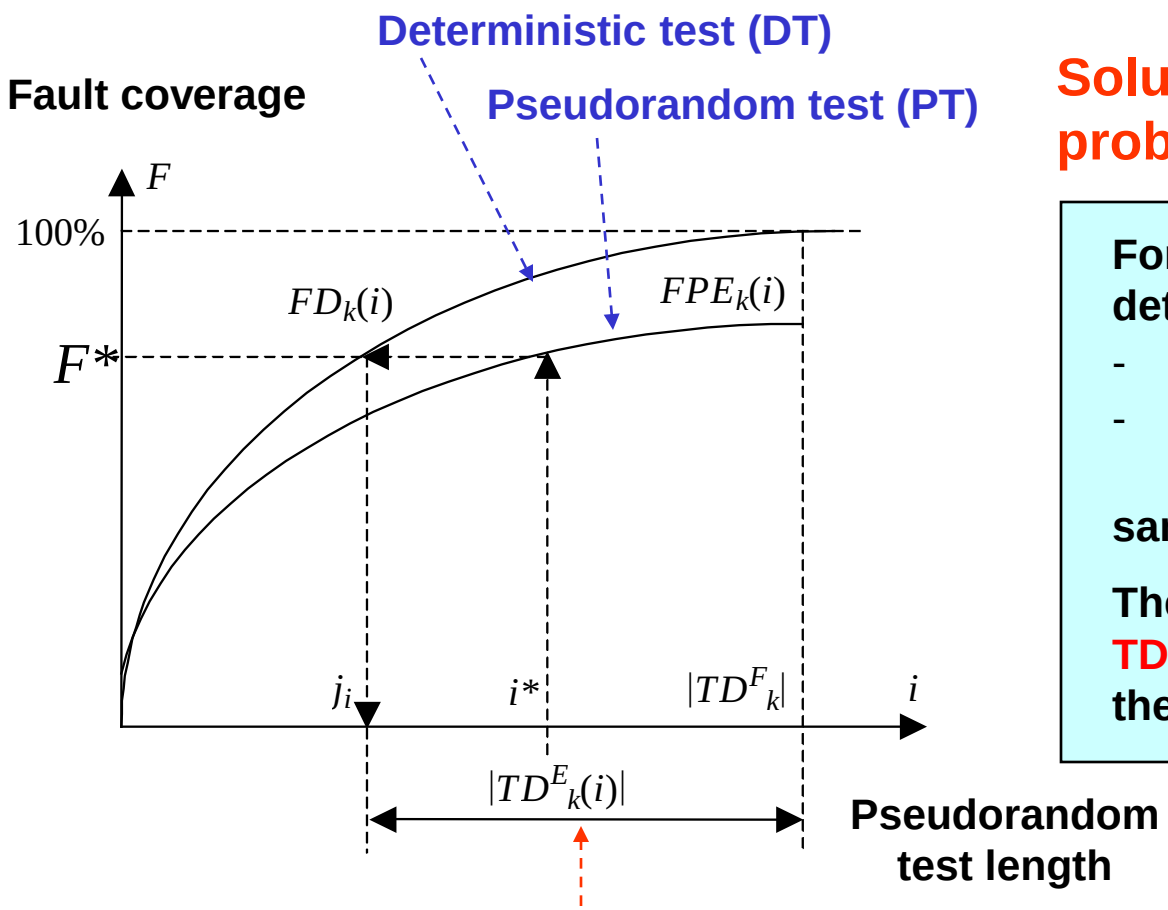


Two problems:

1) **Calculation of DT cost is difficult**
2) **We have to optimize n (!) processes**

How to avoid the calculation of the very expensive full DT cost curve?

Deterministic Test Length Estimation



Deterministic test length estimation for a single core

Solution of the first problem:

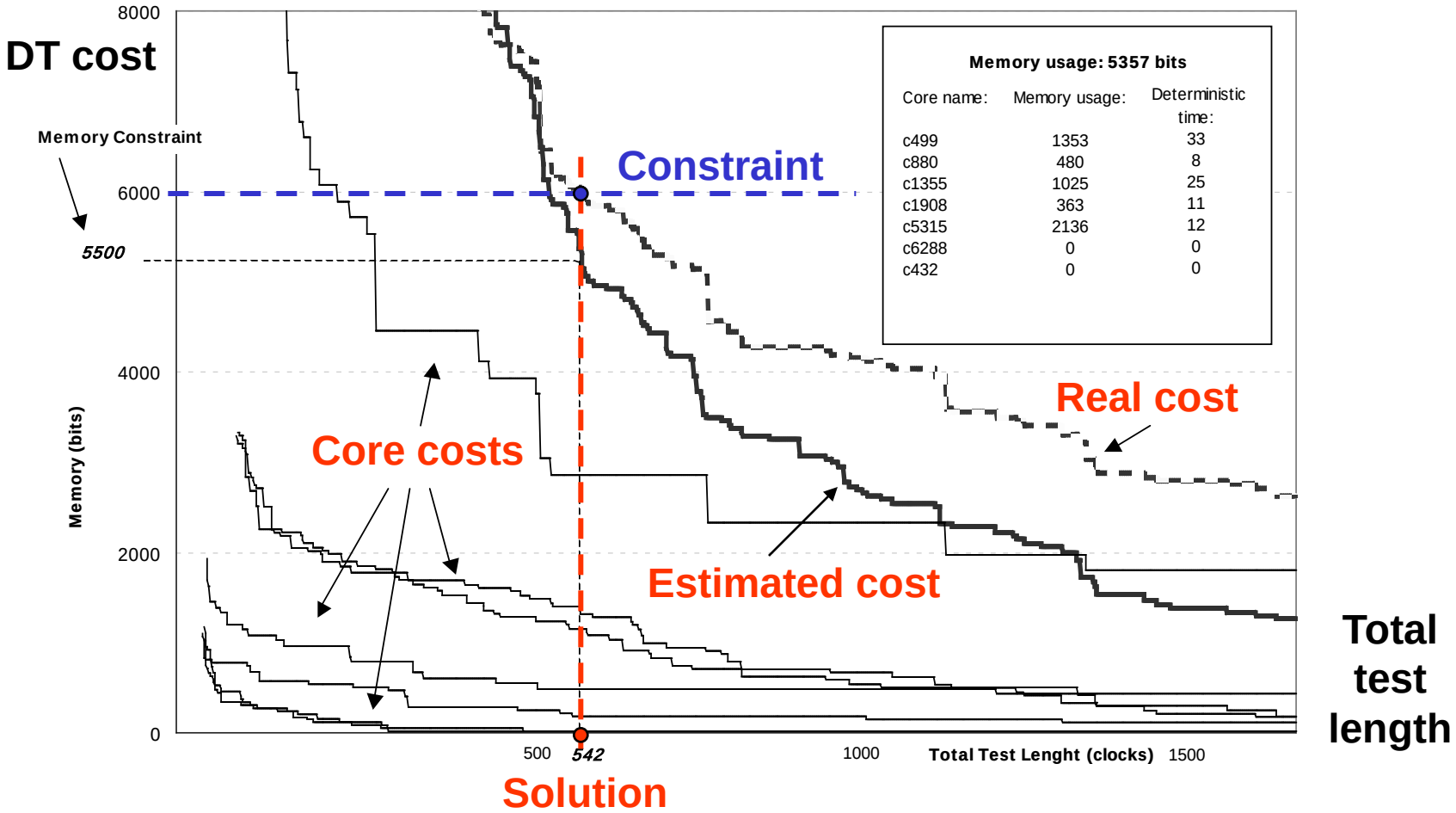
For each PT length **i*** we determine

- 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

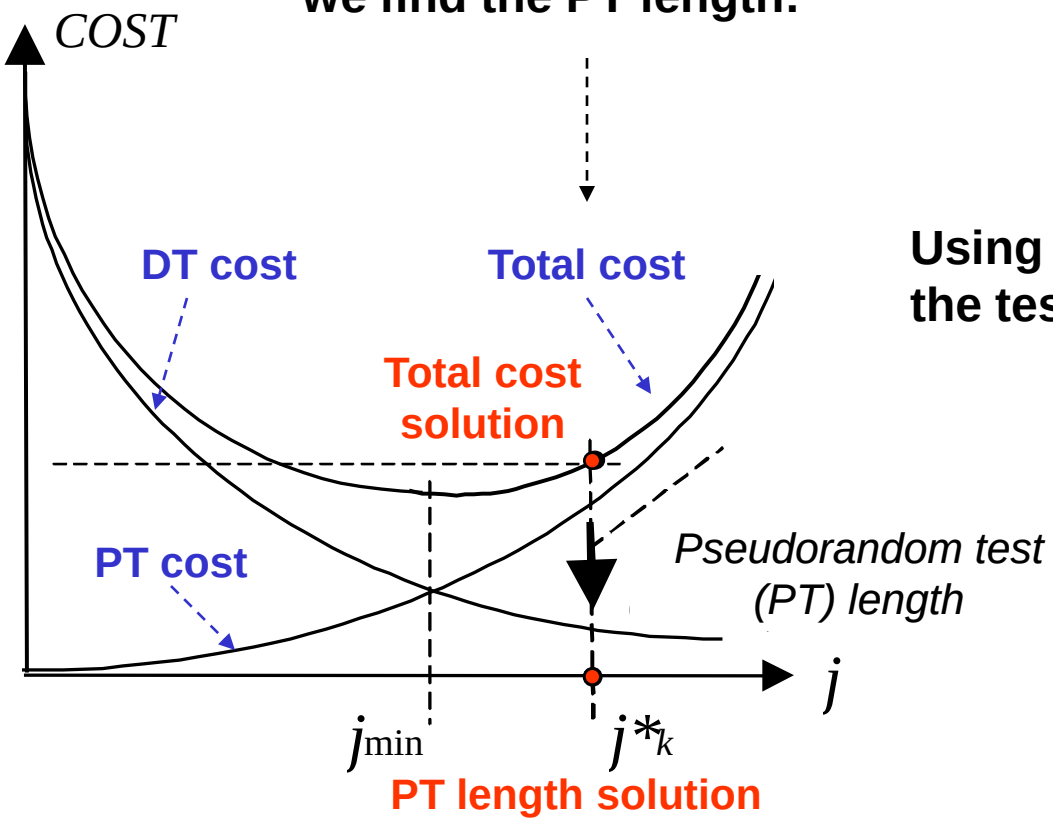
Deterministic Test Cost Estimation

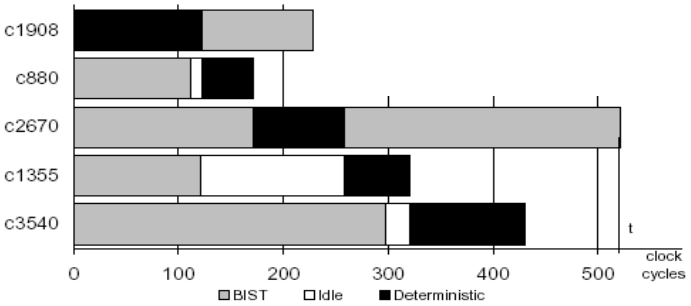
Total cost calculation of core costs:



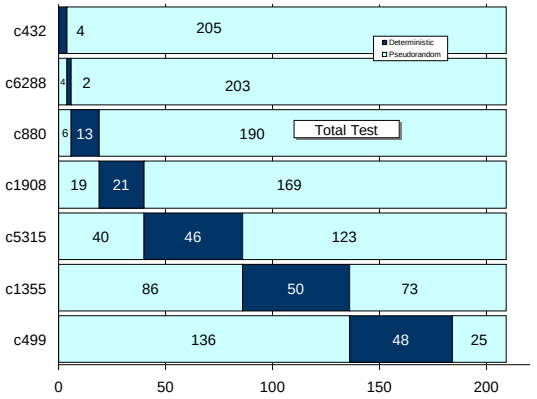
Total Test Cost Estimation

Using total cost solution we find the PT length:



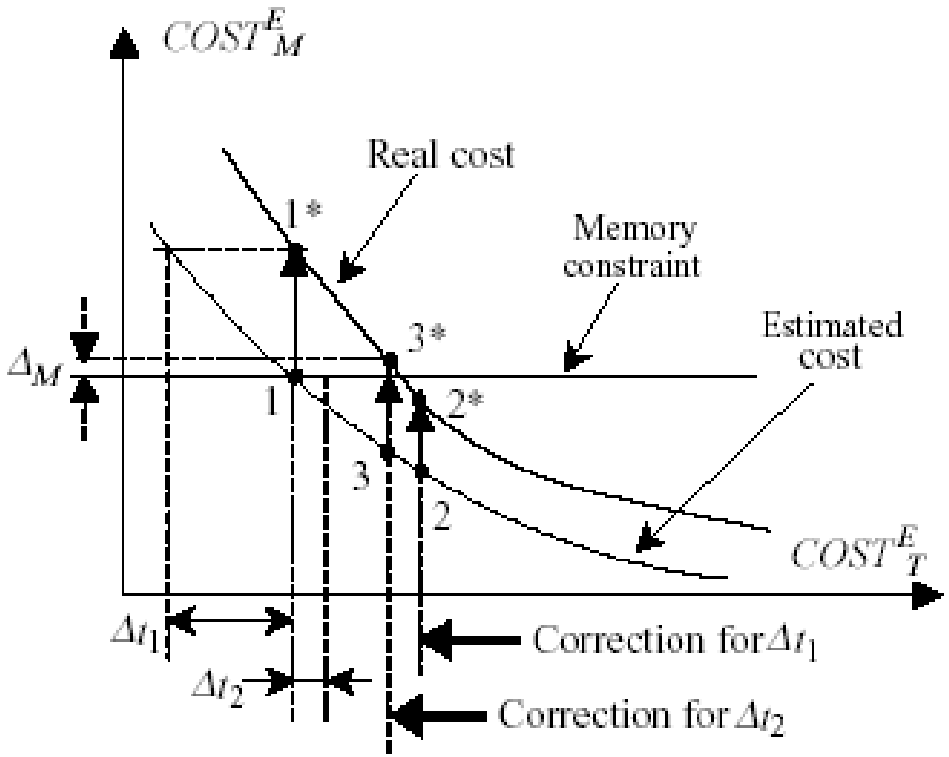


Using PT length, we calculate the test processes for all cores:



Multi-Core Hybrid BIST Optimization

Iterative optimization process:



1 - First estimation

1* - Real cost calculation

2 - Correction of the estimation

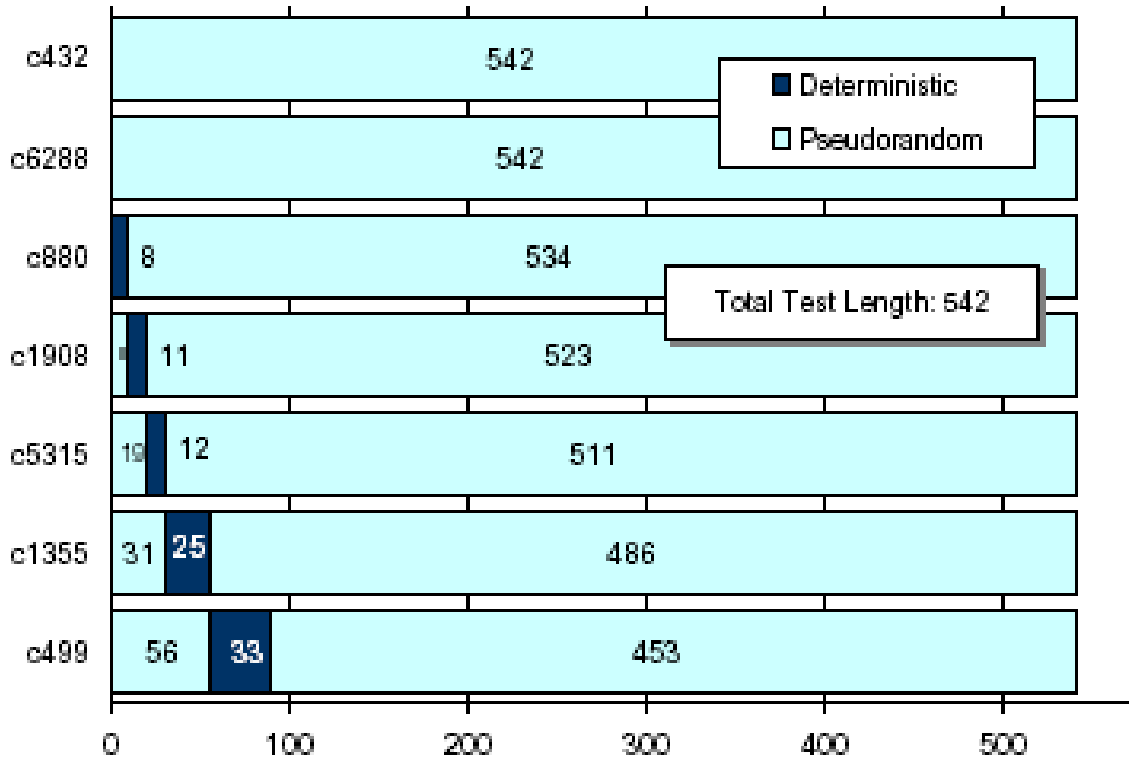
2* - Real cost calculation

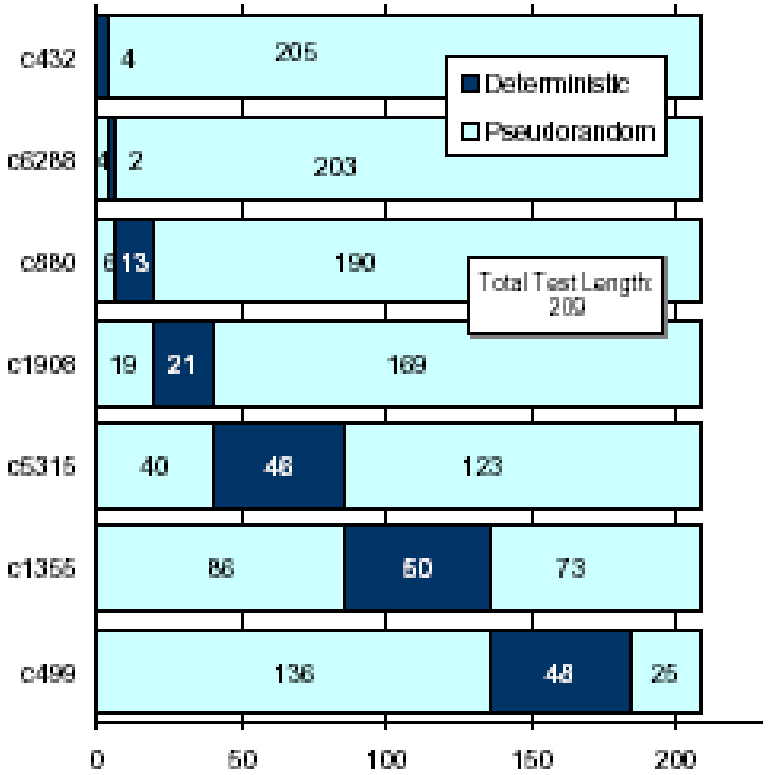
3 - Correction of the estimation

3* - Final real cost

Optimized Multi-Core Hybrid BIST

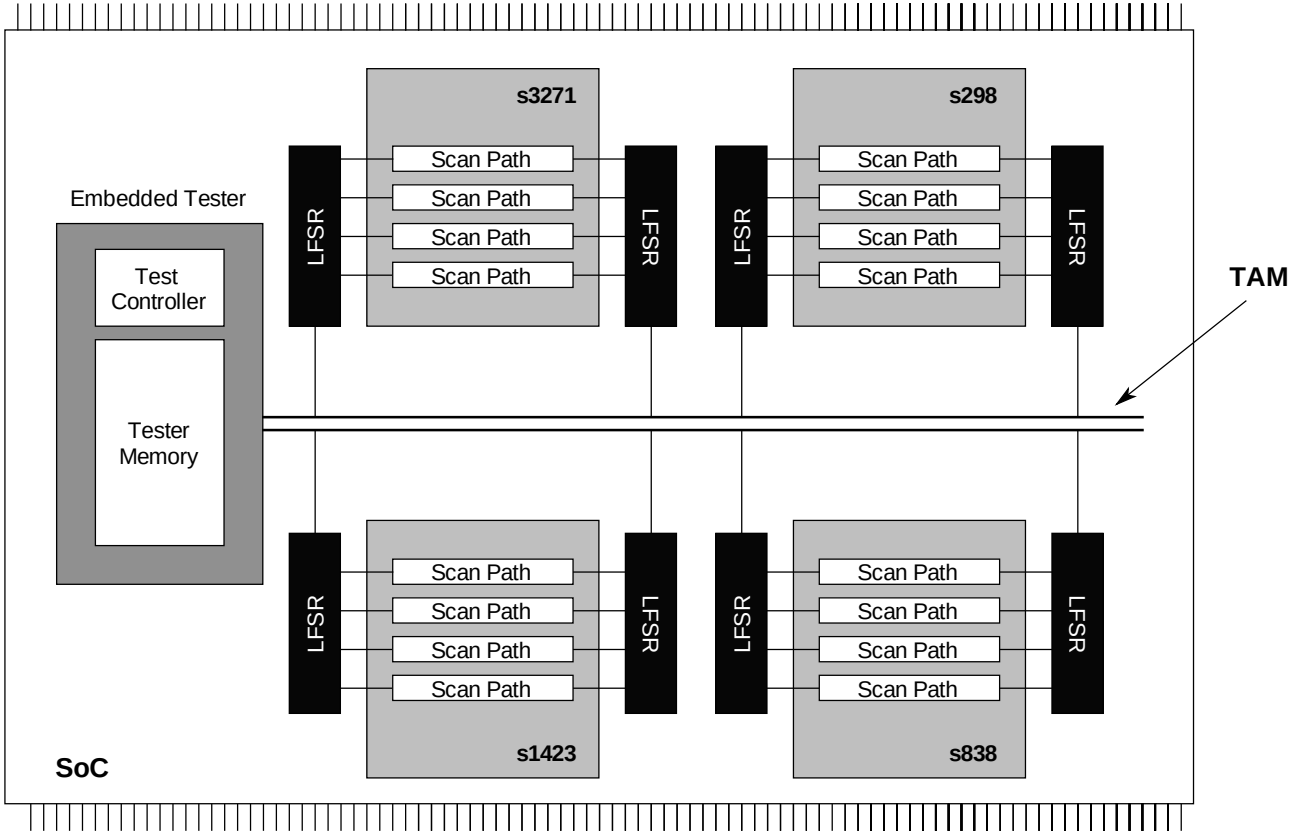
Pseudorandom test is carried out in parallel, deterministic test - sequentially





Test-per-Scan Hybrid BIST

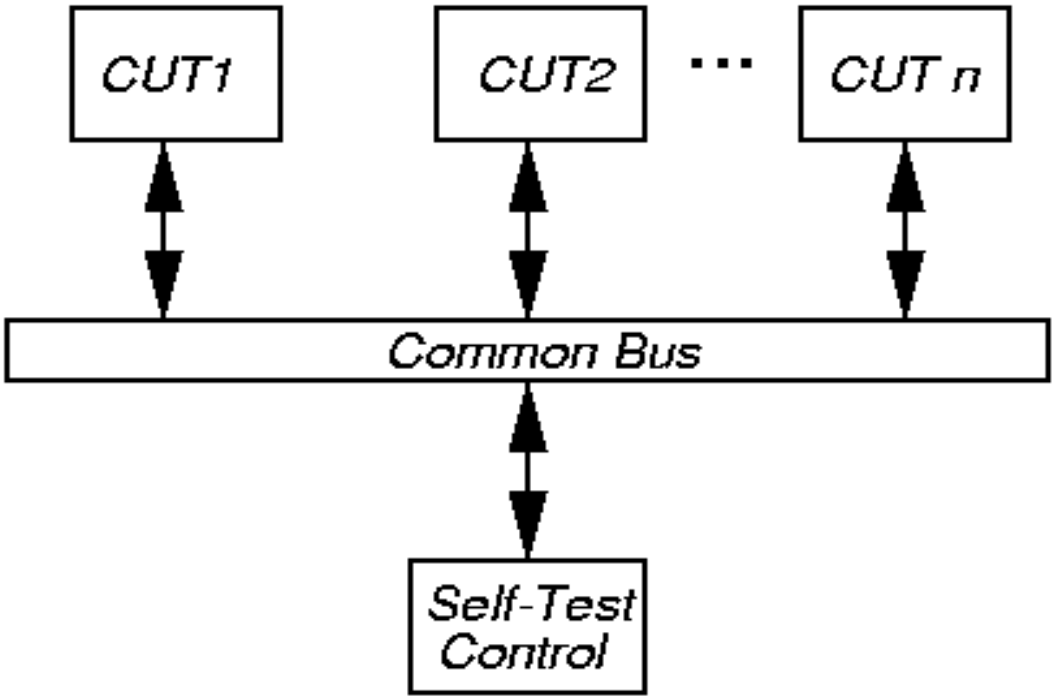
Every core's BIST logic is capable to produce a set of independent pseudorandom test
The pseudorandom test sets for all the cores can be carried out simultaneously



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

Only one test access bus at the system level is needed.

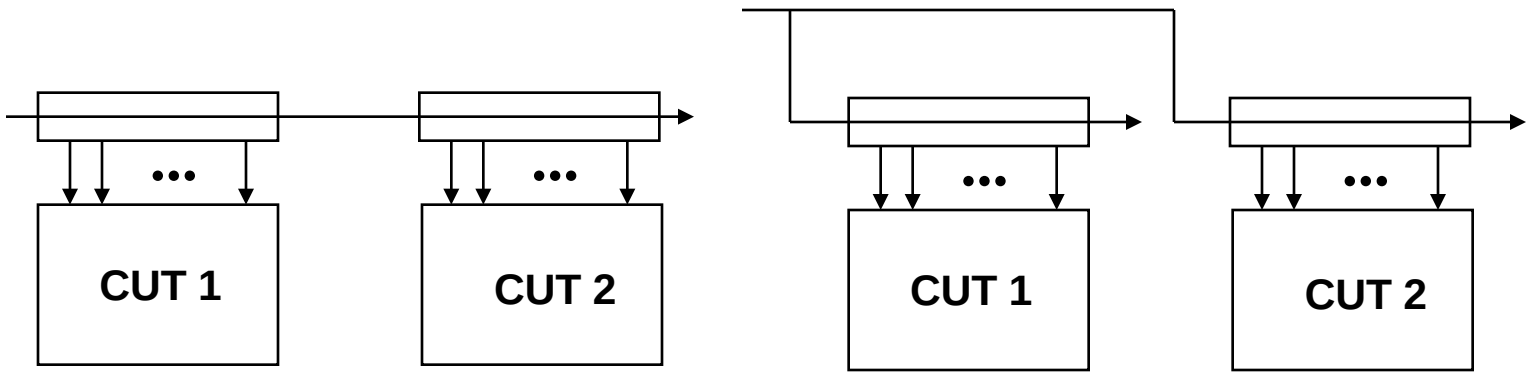
Bus-Based BIST Architecture



- ***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**

Broadcasting Test Patterns in BIST

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

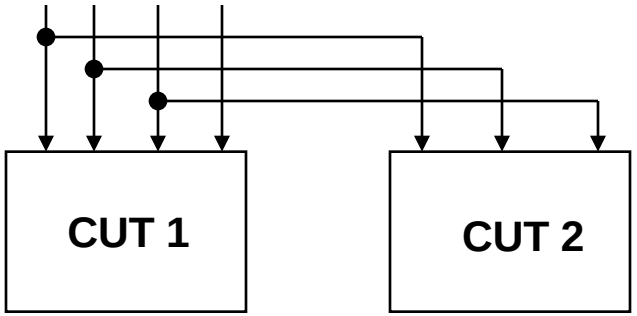


Traditional single scan design **Broadcast test architecture**

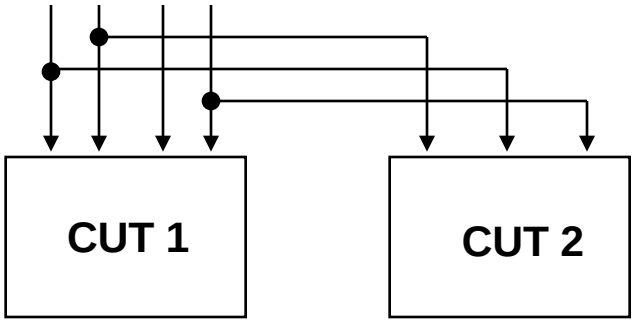
While one module is tested by its test patterns, the same test patterns can be applied simultaneously to other modules in the manner of pseudorandom testing

Broadcasting Test Patterns in BIST

Examples of connection possibilities in Broadcasting BIST:



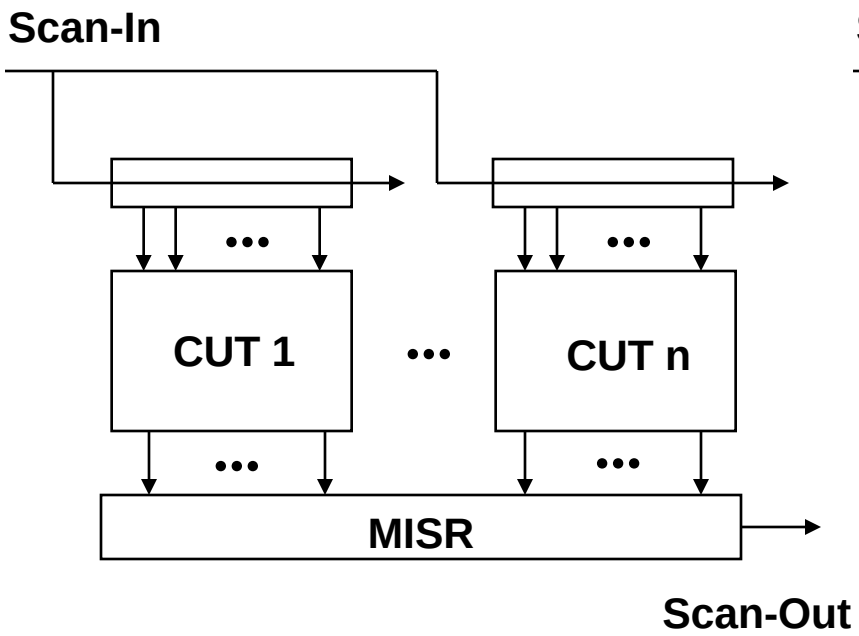
***j*-to-*j* connections**



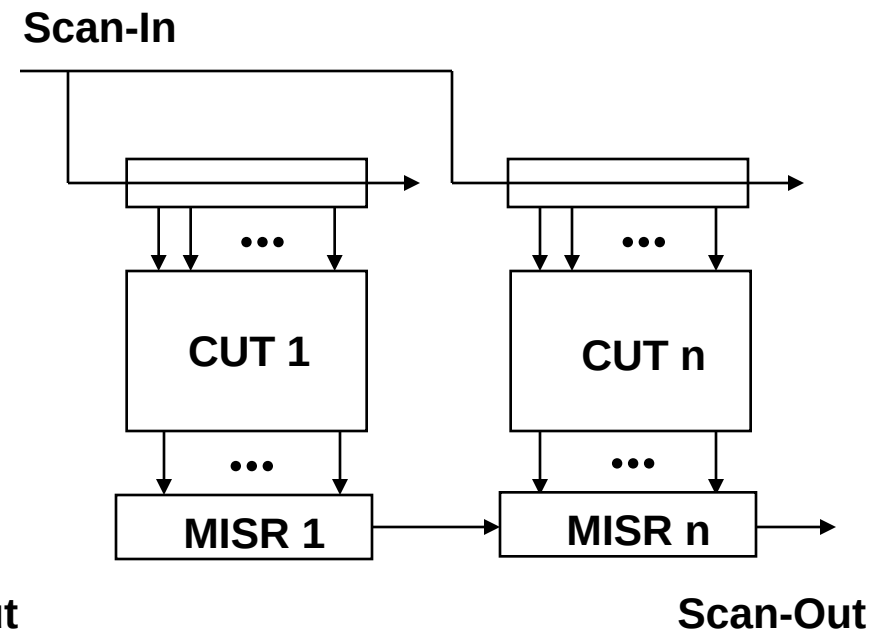
Random connections

Broadcasting Test Patterns in BIST

Scan configurations in Broadcasting BIST:



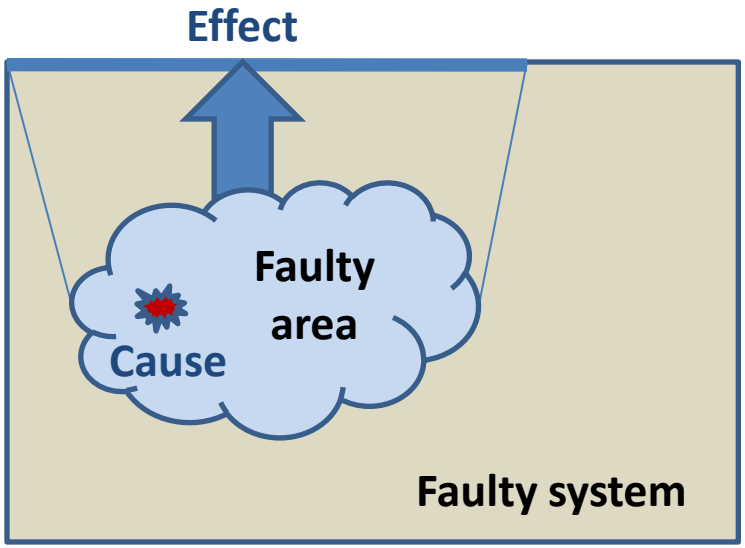
Common MISR

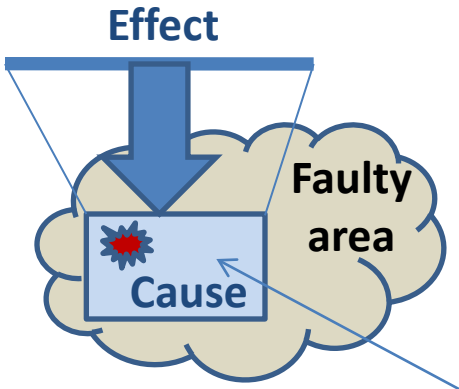


Individual and multiple MISRs

Fault-Model Free Fault Diagnosis

Combined cause-effect and effect-cause diagnosis





1) Cause-Effect Fault Diagnosis

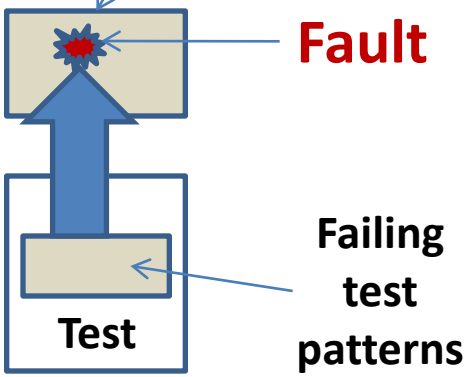
Suspected faulty area is located

2) Effect-Cause Fault Diagnosis

Faulty block is located in the suspected faulty area

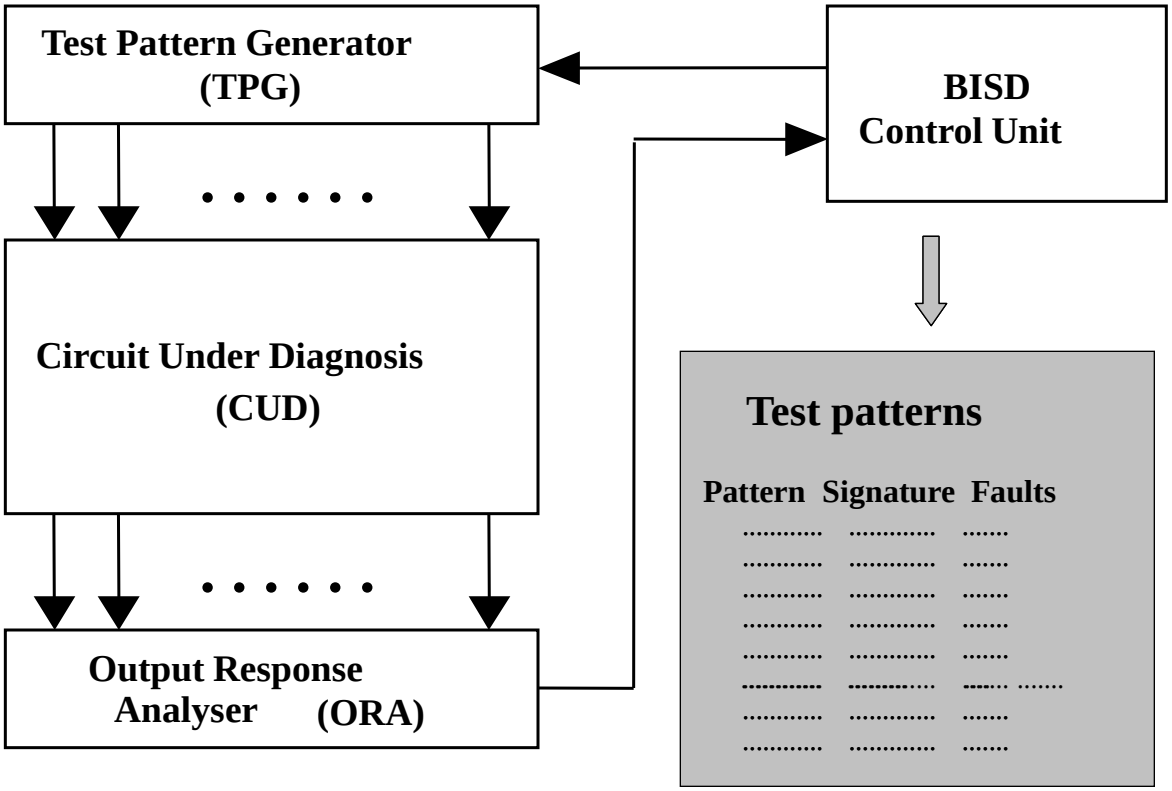
3) Fault Reasoning

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

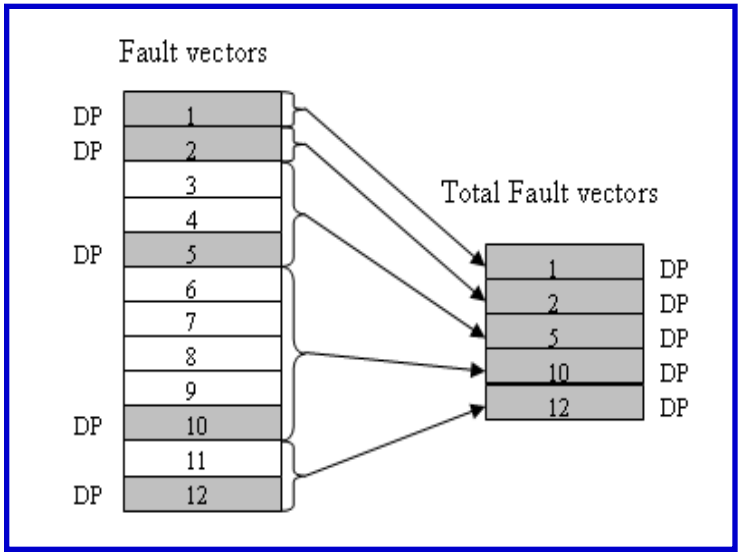


Embedded BIST Based Fault Diagnosis

BISD scheme:



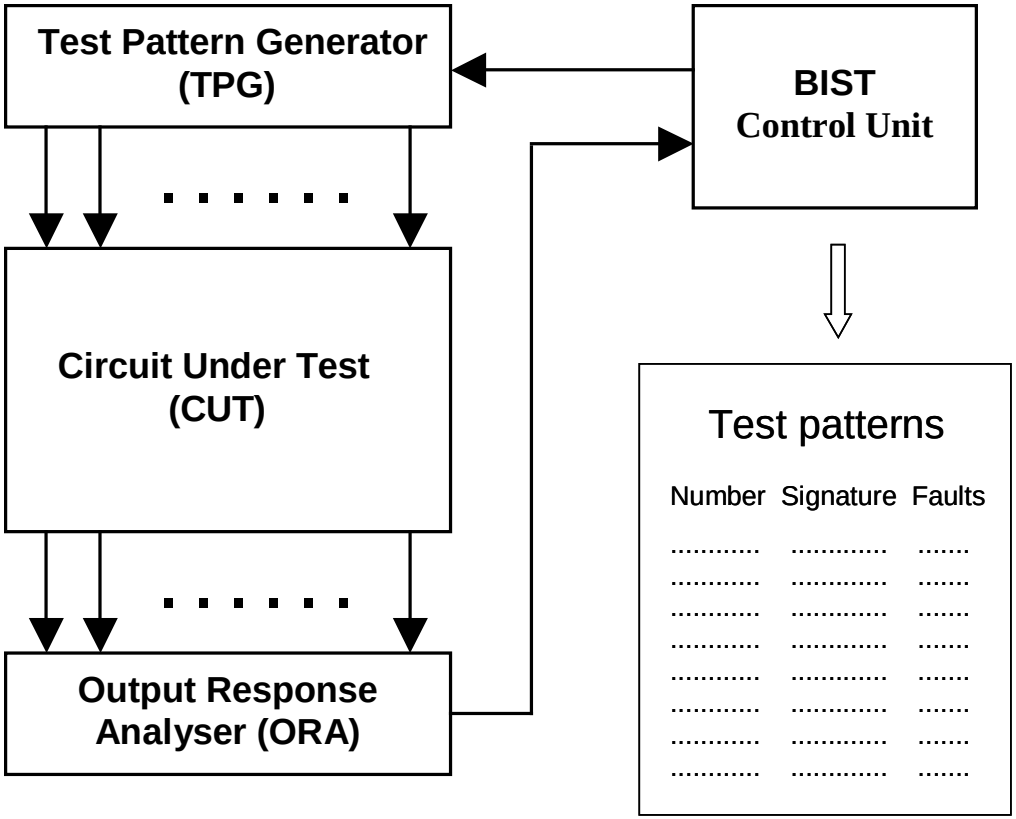
Pseudorandom test sequence:



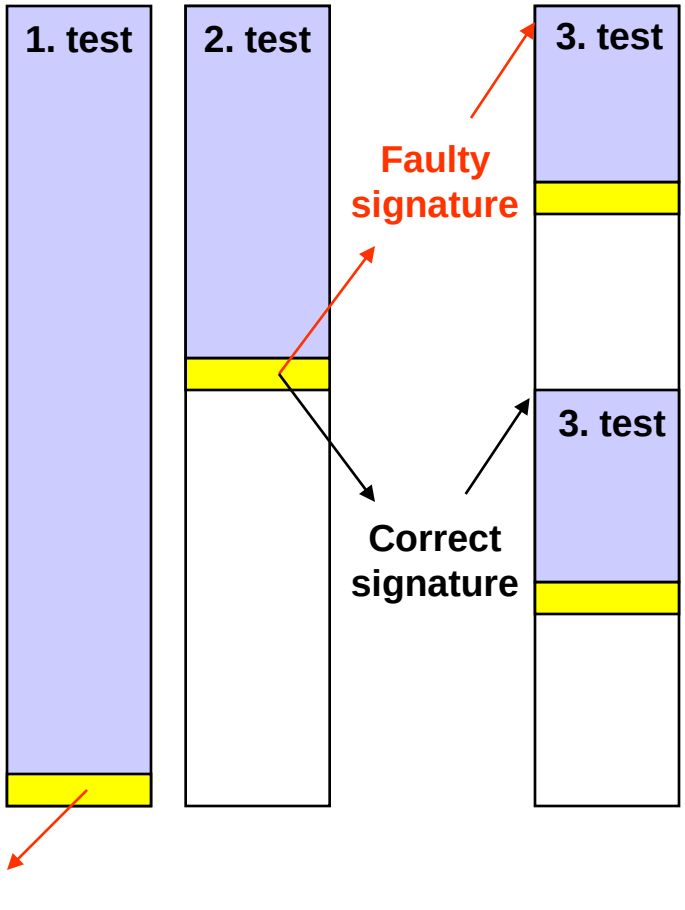
Diagnostic Points (DPs) – patterns that detect new faults

Further minimization of DPs – as a tradeoff with diagnostic resolution

Built-In Fault Diagnosis



Diagnosis procedure:

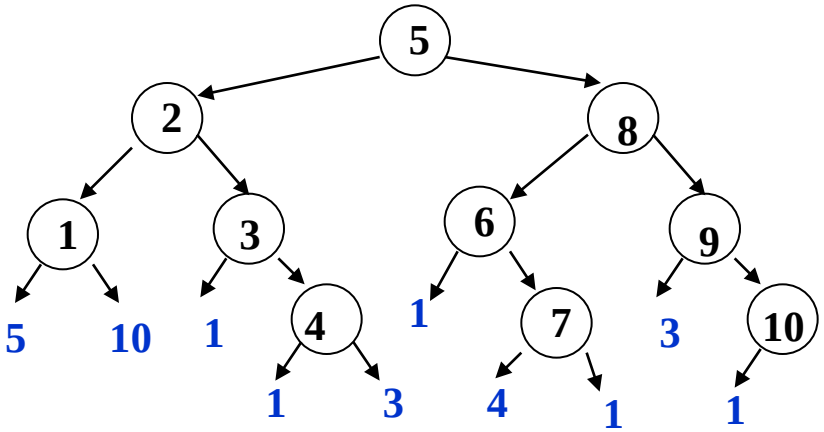


Built-In Fault Diagnosis

Pseudorandom test fault simulation

№	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

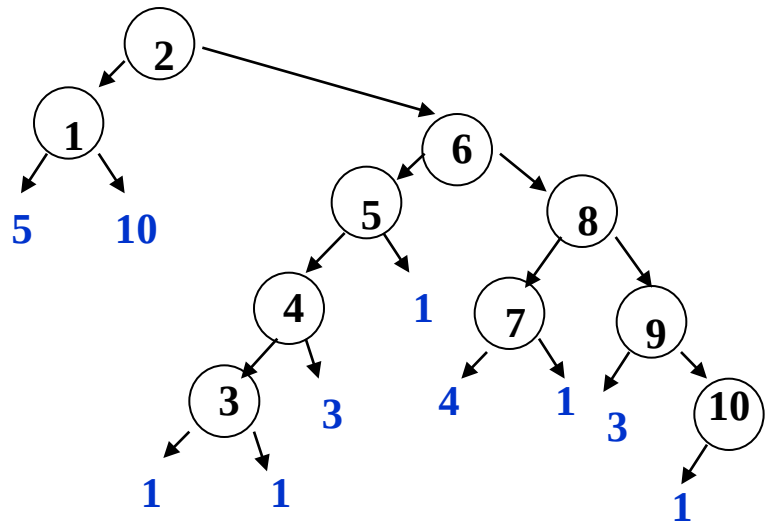
Average number of clocks: 8,67

Built-In Fault Diagnosis

Pseudorandom test fault simulation

№	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

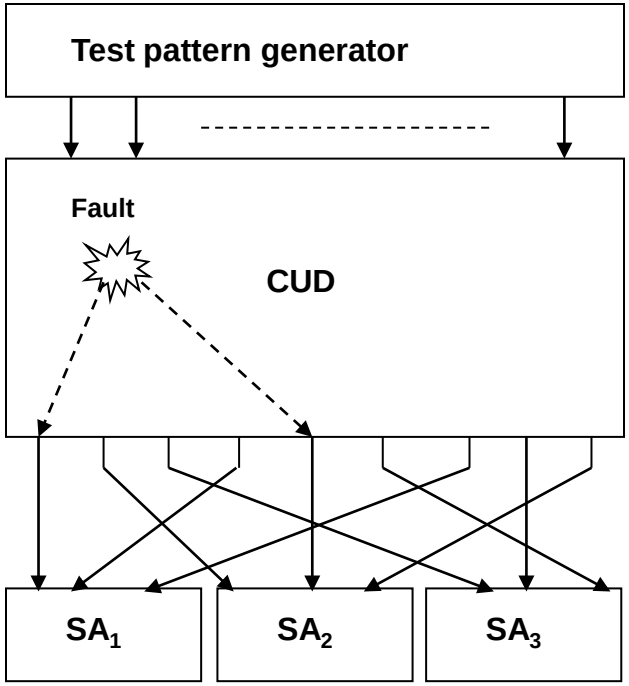


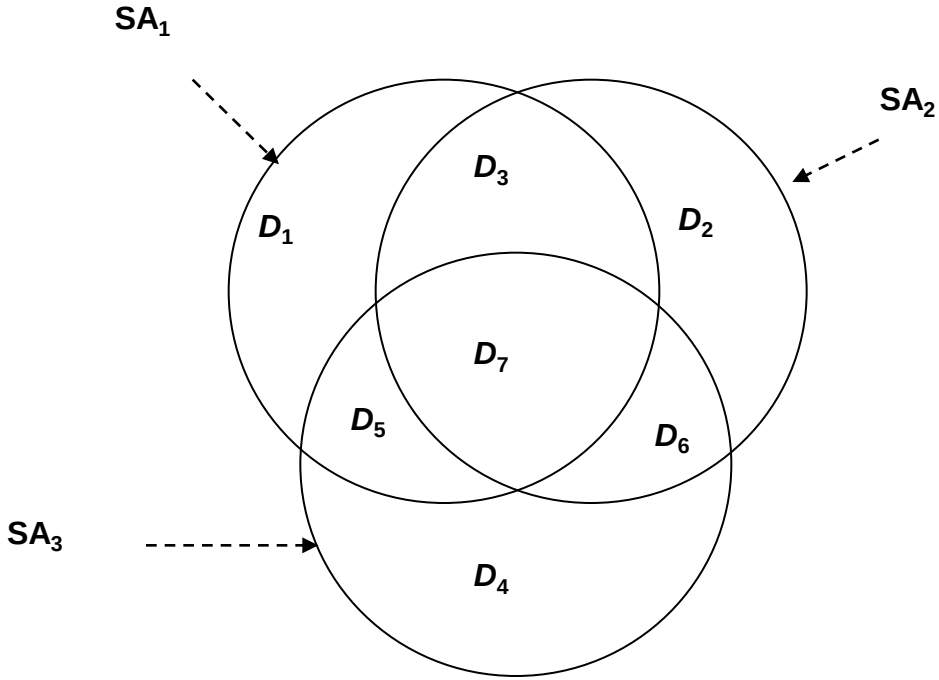
Average number of test sessions: 3,06

Average number of clocks: 6,43

Built-In Fault Diagnosis

Diagnosis with multiple signatures:



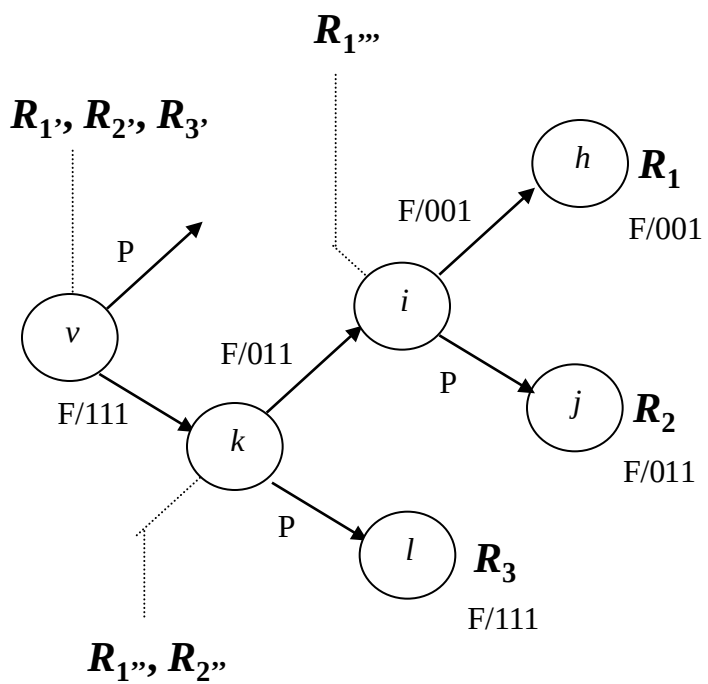


Built-In Fault Diagnosis

Diagnosis with multiple signatures:

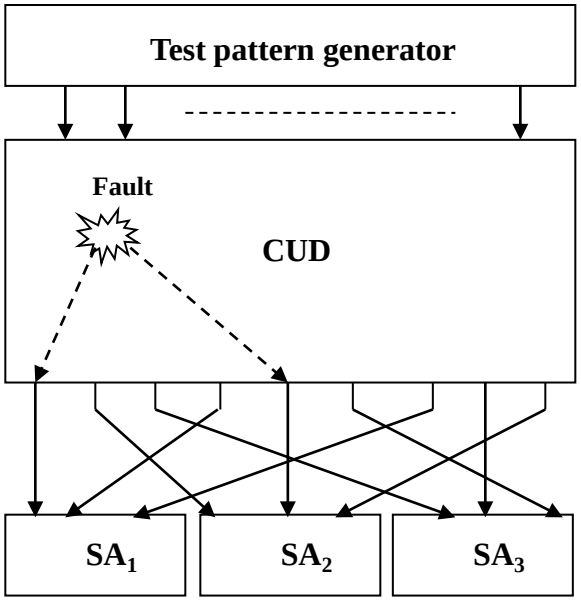
No	Codeword			Diagnosis
h	0	0	1	R_1
i	0	0	1	$R_{1'''}$
j	0	1	1	R_2
k	0	1	1	$R_{1''}, R_{2''}$
l	1	1	1	R_3
v	1	1	1	$R_{1'}, R_{2'}, R_{3'}$

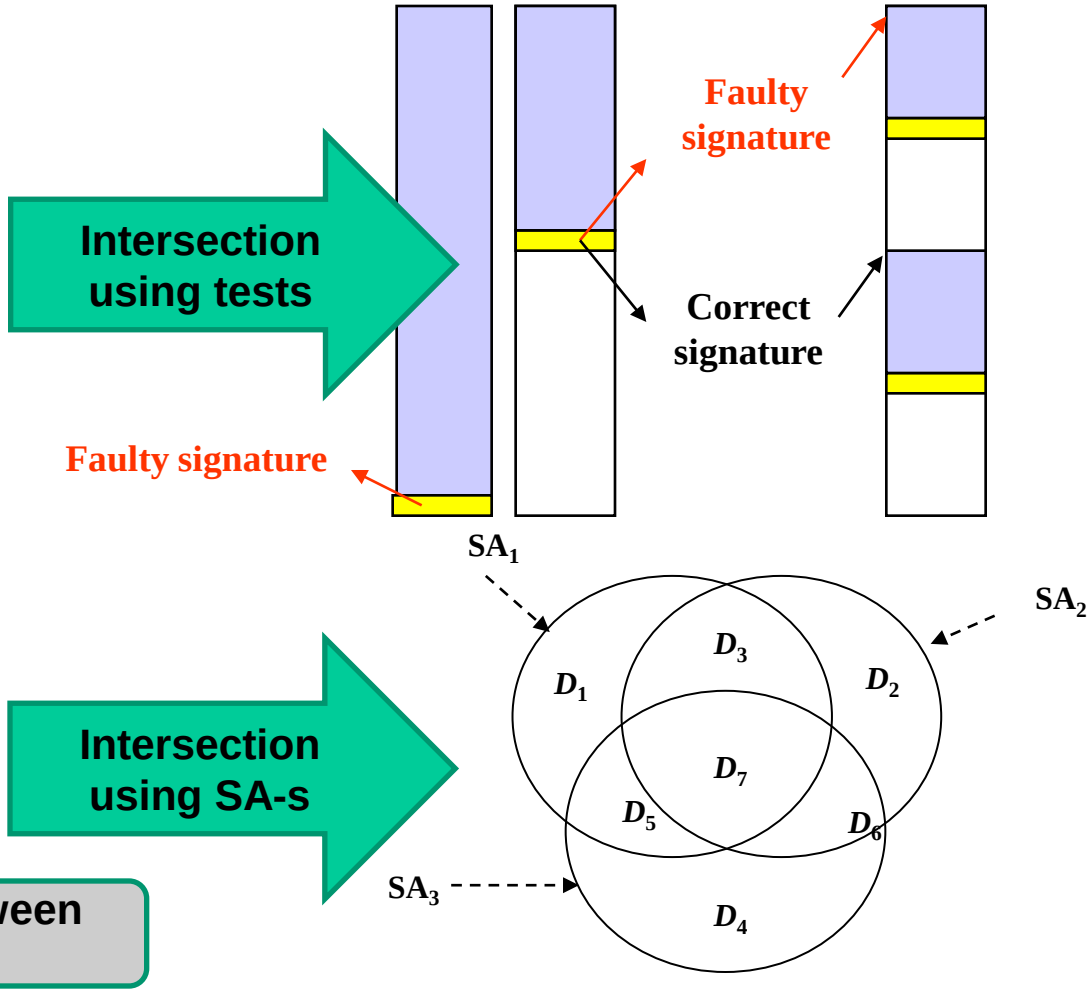
Diagnostic tree



Built-In Fault Diagnosis

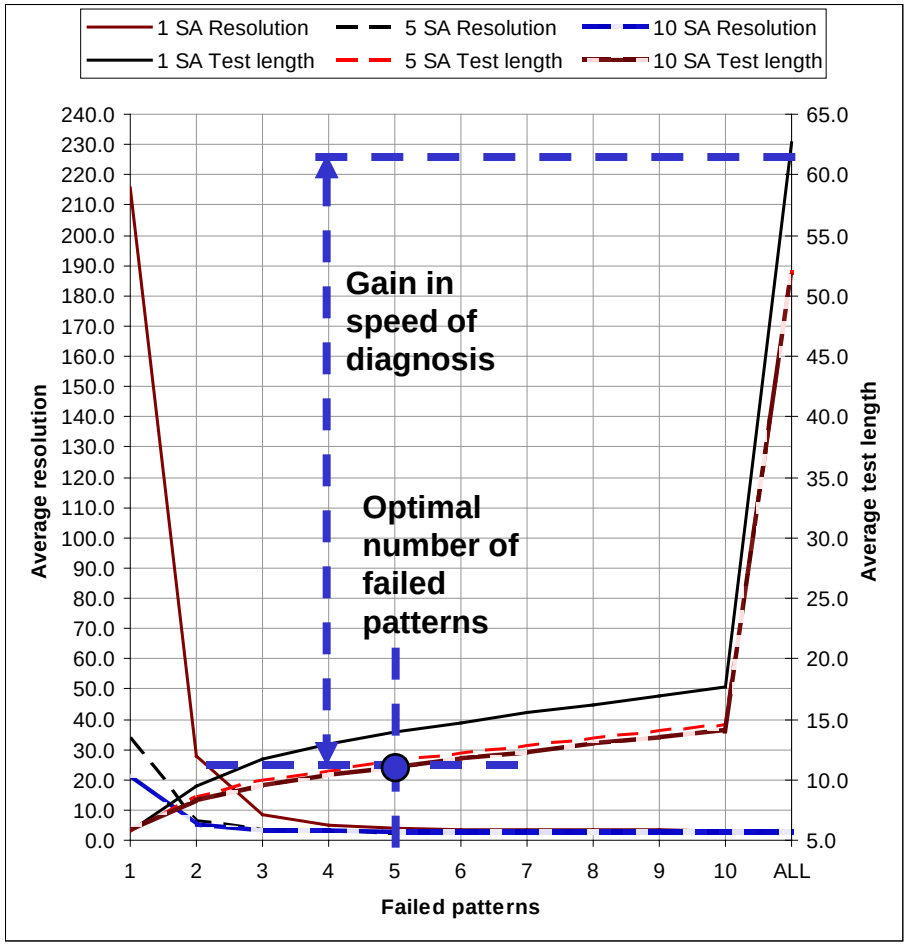
BIST with multiple signature analyzers





Optimization of the interface between CUD and SA-s

Built-In Fault Diagnosis



Diagnosis with multiple signatures:

Measured:

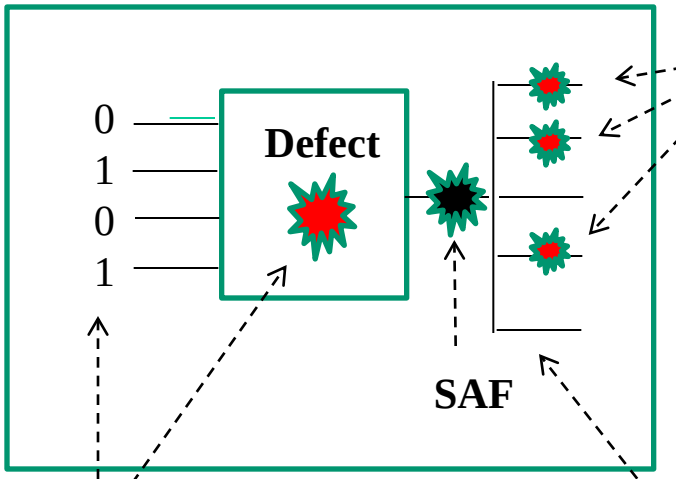
- **average resolution**
- **average test length**

Compared: 1SA, 5SA, 10SA

Gain in test length: 6 times

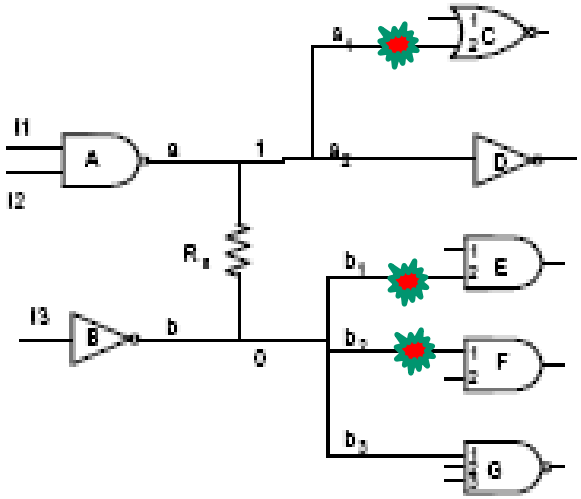
Extended Fault Models

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



Multiple fault

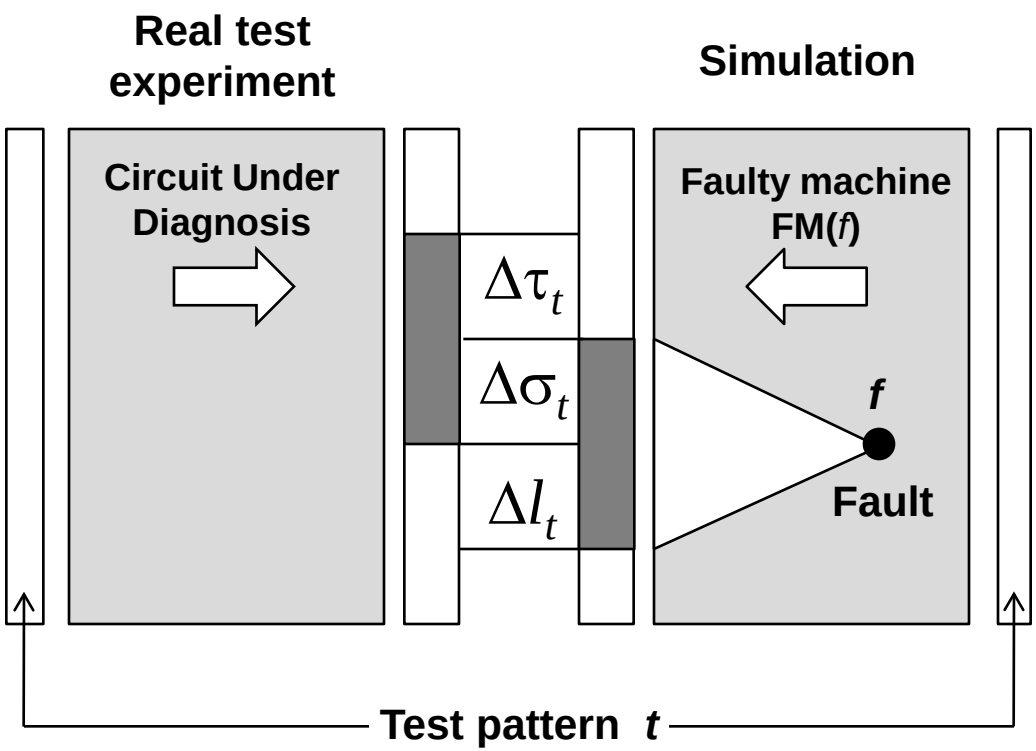
Resistive bridge fault



Conditional fault
Pattern fault
Constrained SAF
Single faulty signal

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

Diagnosis of Fault Model Free Defects



Fault evidence:

for test pattern ***t***

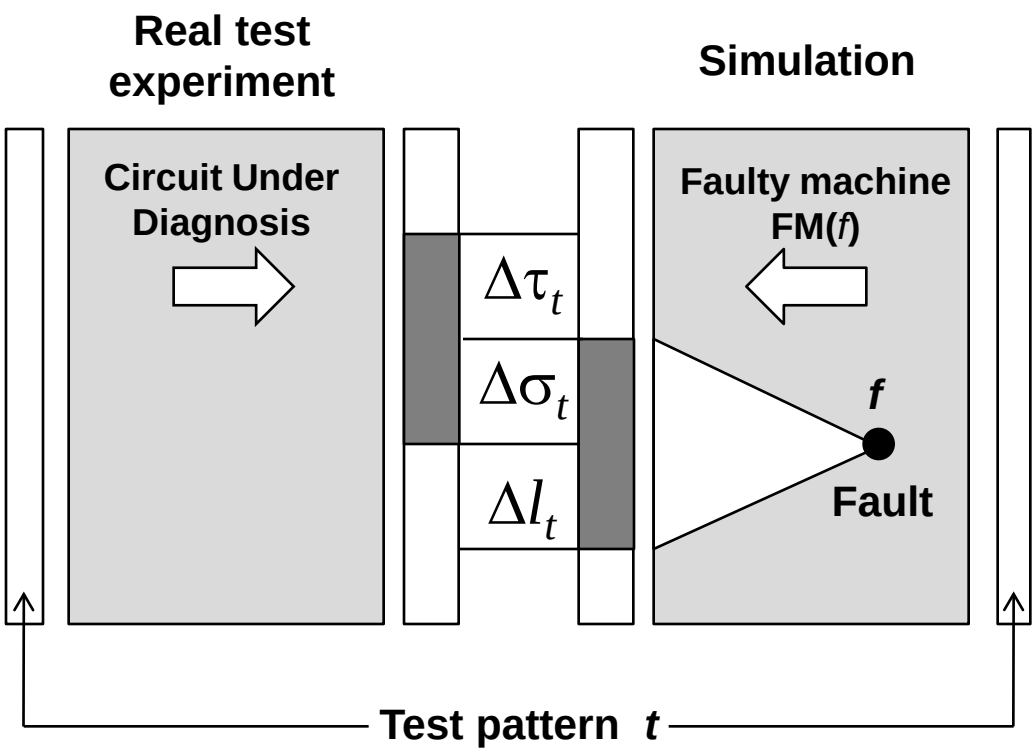
$e(f,t) = (\Delta\tau_t, \Delta\sigma_t, \Delta l_t, \Delta\gamma_t)$

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

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

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

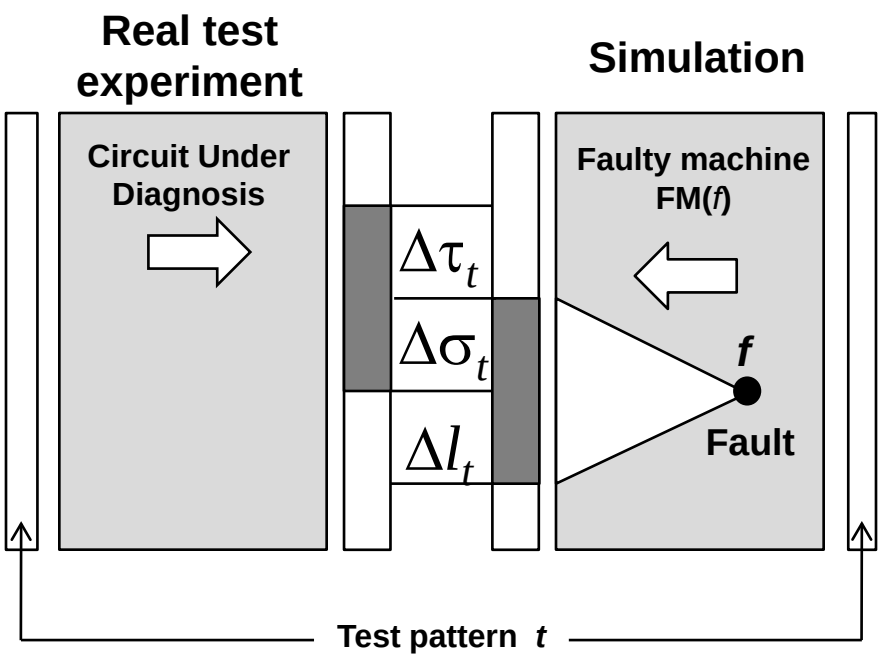
Diagnosis of Fault Model Free Defects



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



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

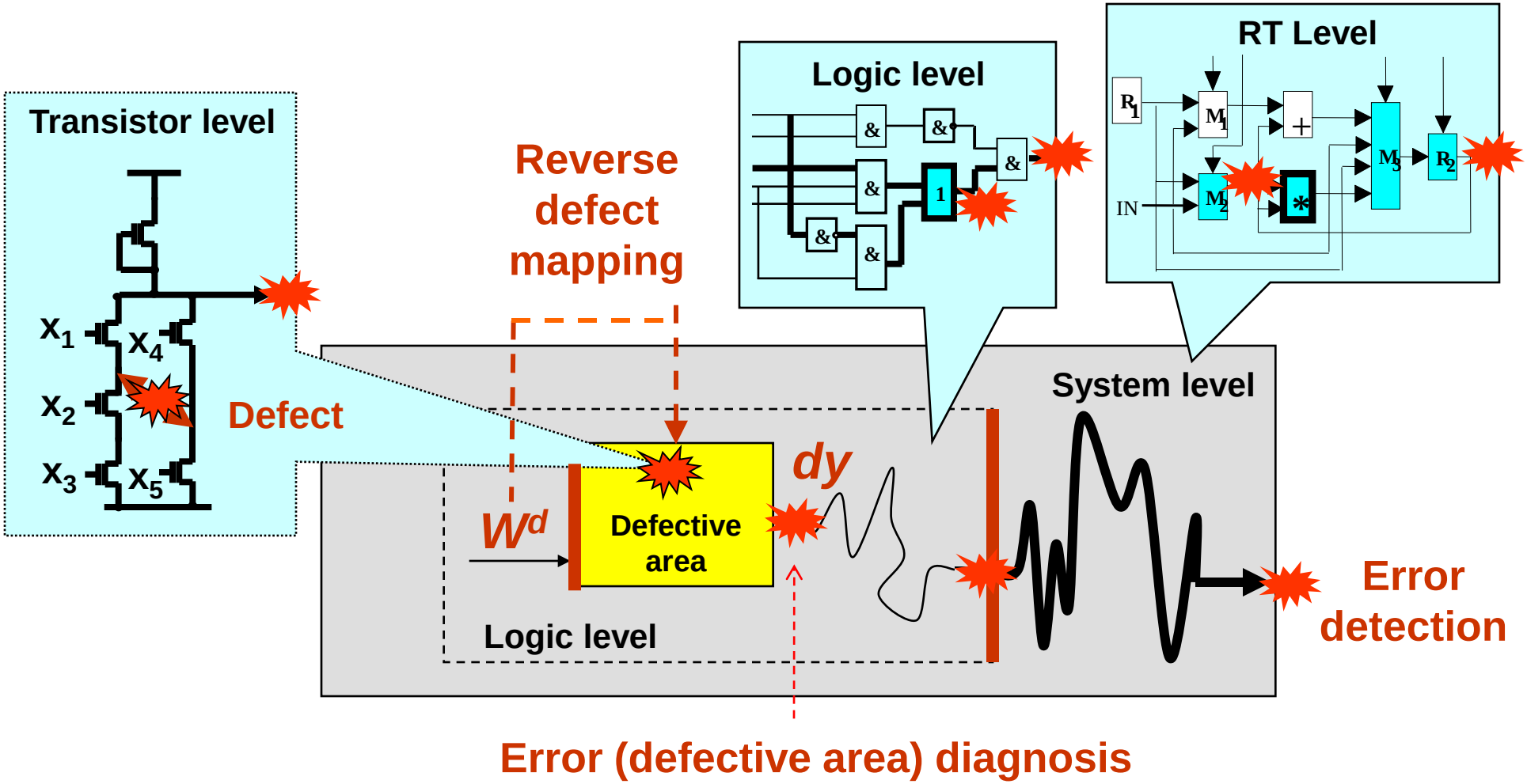
(3) If γ_T and σ_T are equal then by increasing l_T

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

Example:

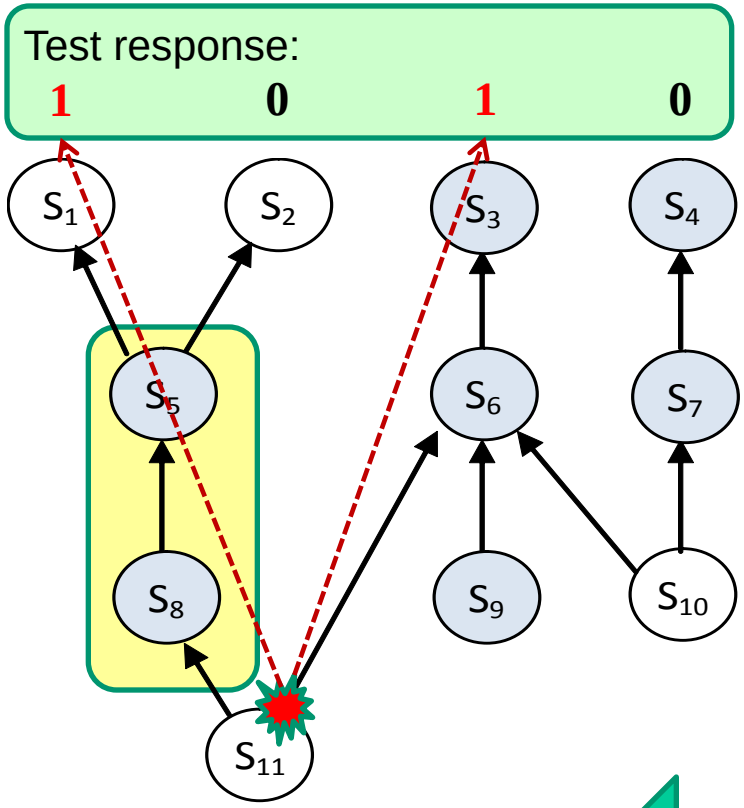
SAF	γ_T	σ_T	l_T
f_1	0	42	0
f_2	30	42	15
f_3	30	42	25
f_4	30	42	30
f_5	30	36	38
f_6	38	23	22
f_7	38	23	23

Fault Diagnosis Without Fault Models



Fault Model Free Fault Diagnosis

System network graph



No match

Because of the unidirectional "distortions" of test responses, rectification is possible

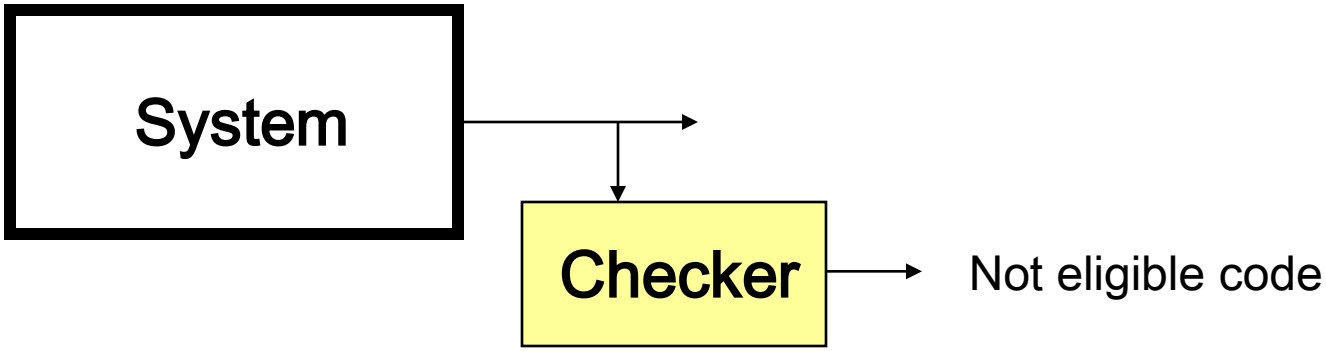
Rectified code

Diagnosis: s11

Diagnostic table

	1	2	3	4
1	1			
2		1		
3			1	
4				1
5	1	1	0	
6			1	
7				1
8	1	1	0	
9			1	
10			1	1
11	1	1	1	0
	1	0	1	0

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**

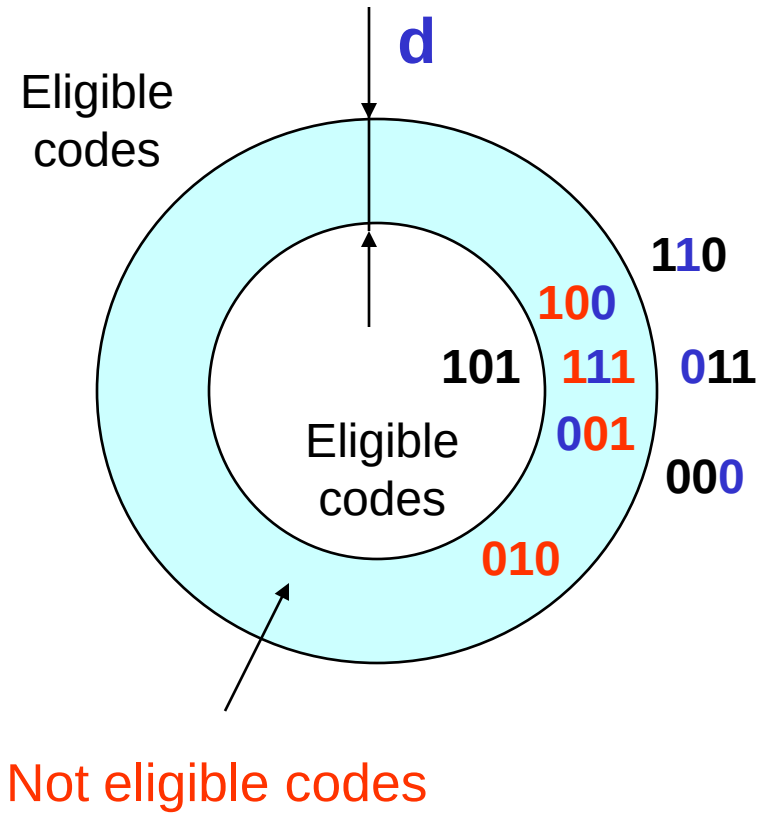
Eligible

Not eligible

Error Detecting/Correcting Codes

Hamming distance between codes:

Minimal number of bits how two codes differ from each other



Parity check:

d = 2

Parity bit

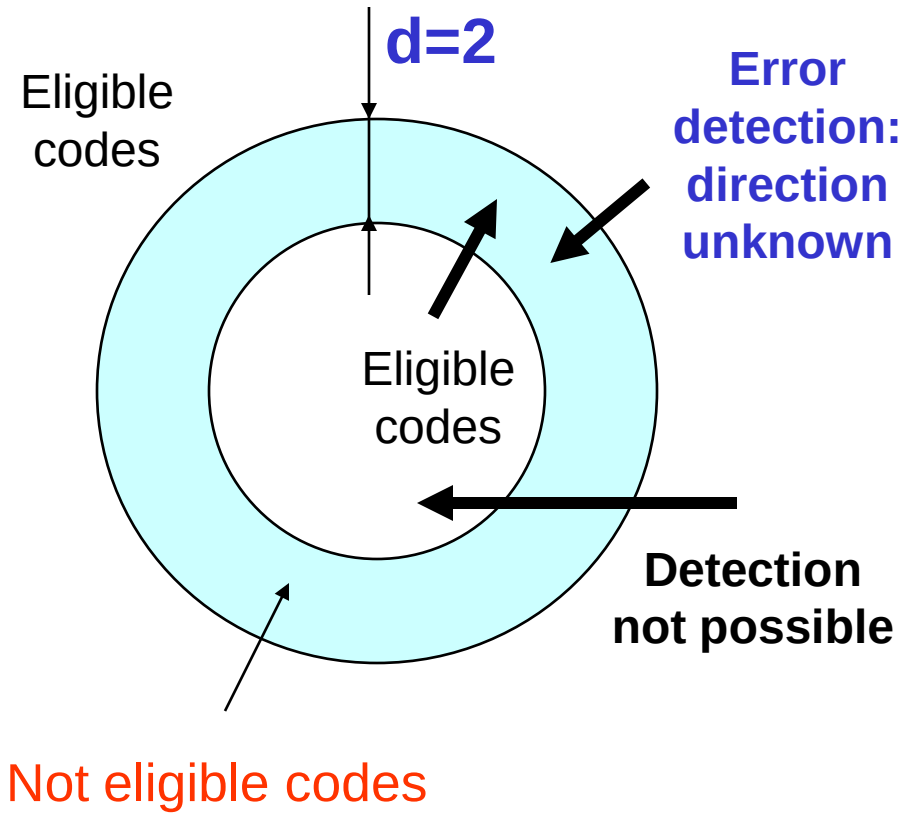
00	0	0	1
01	1	3	2
10	1	5	4
11	0	6	7

Eligible

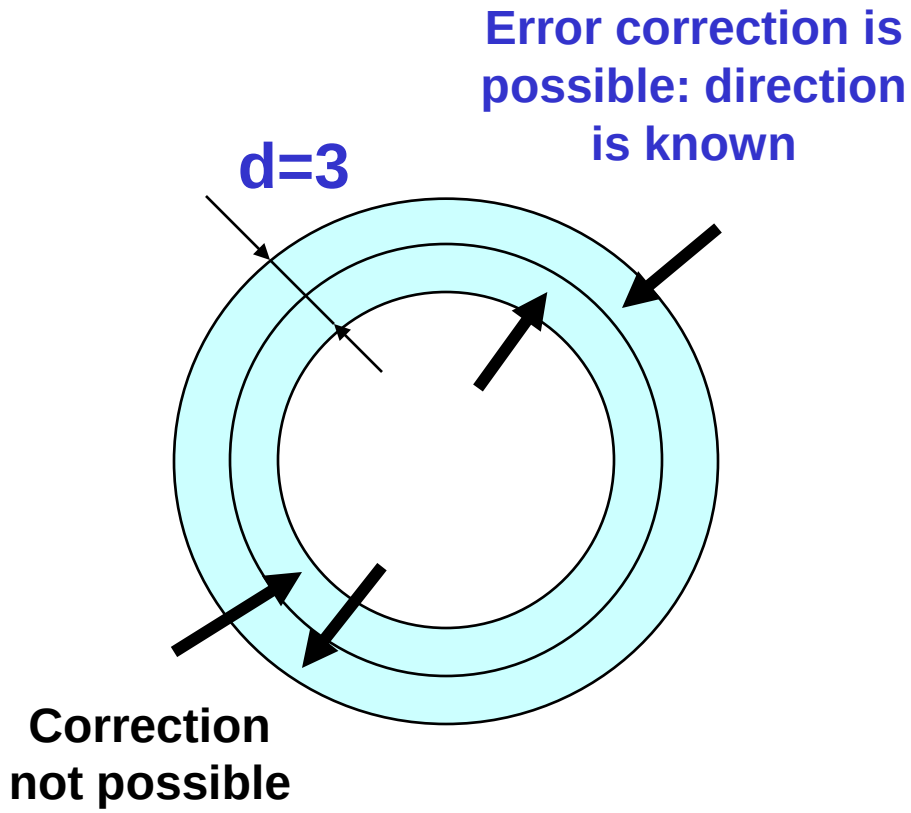
Not eligible

Error Detecting/Correcting Codes

Error detecting codes:



Error correcting codes:



Fault Tolerance: Error Correcting Codes

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

e - error correction

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

Check bits

q	c

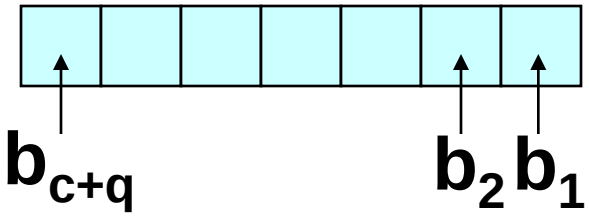
Information bits

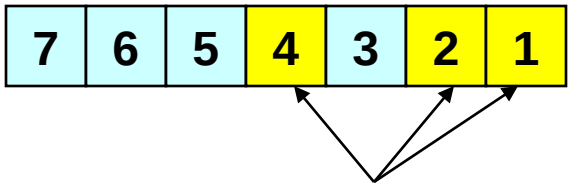
Error free

For addressing of the erroneous bit

Fault Tolerance: One Error Correcting Code

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





Check bits

b_2^i, i = 0,1,,...,c-1

Calculation of check sums:

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

P_i – number of bits where $b_i = 1$

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

Fault Tolerance: One Error Correcting Code

Location of erroneous bit:

7	6	5	4	3	2	1

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

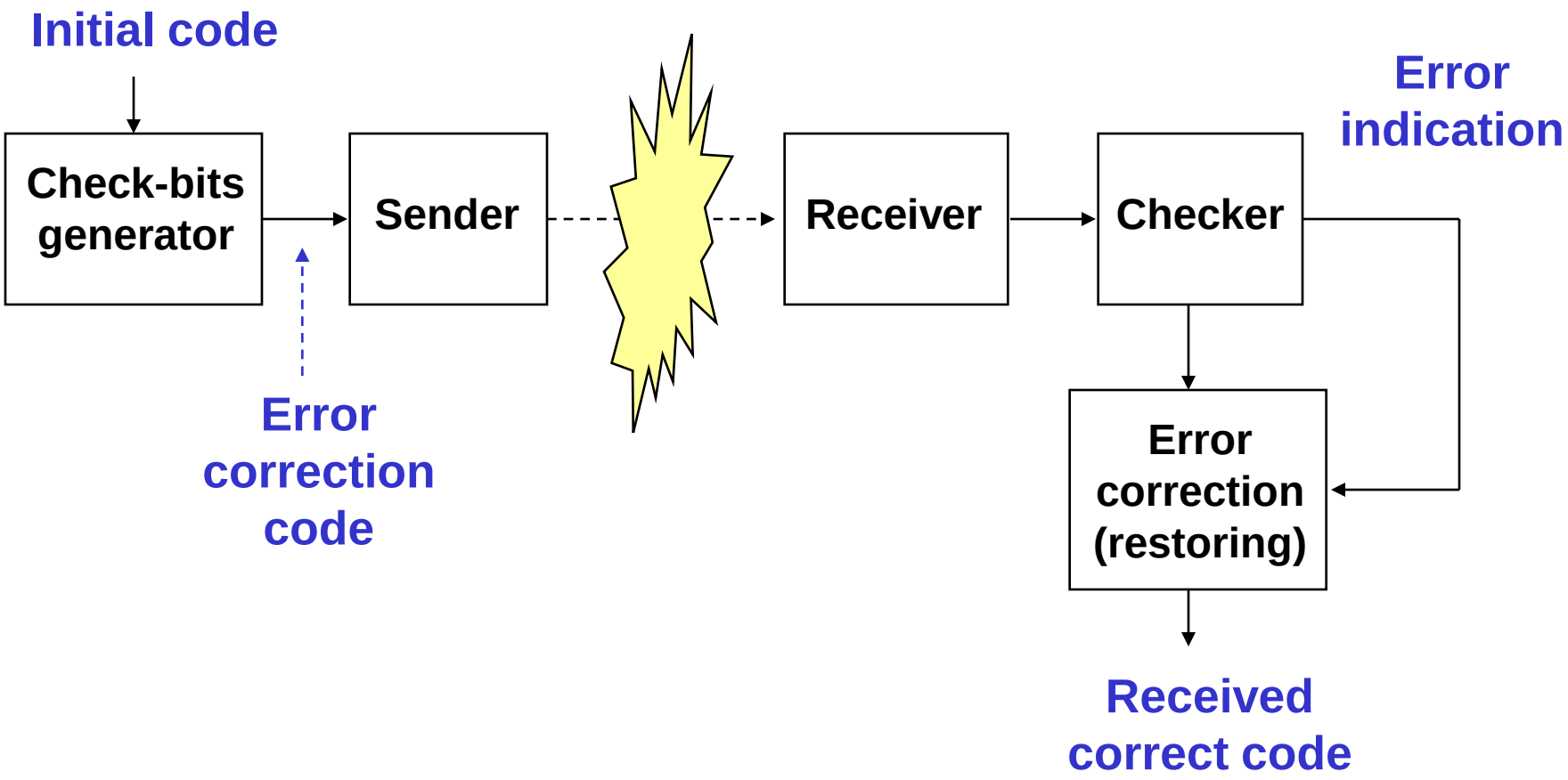
Analogy with fault diagnosis by using fault table:

1	0	1	0	1	0	0	Initial code
1	0	1	1	1	0	0	Received code

	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 bits

C = (N) mod m - check bits

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

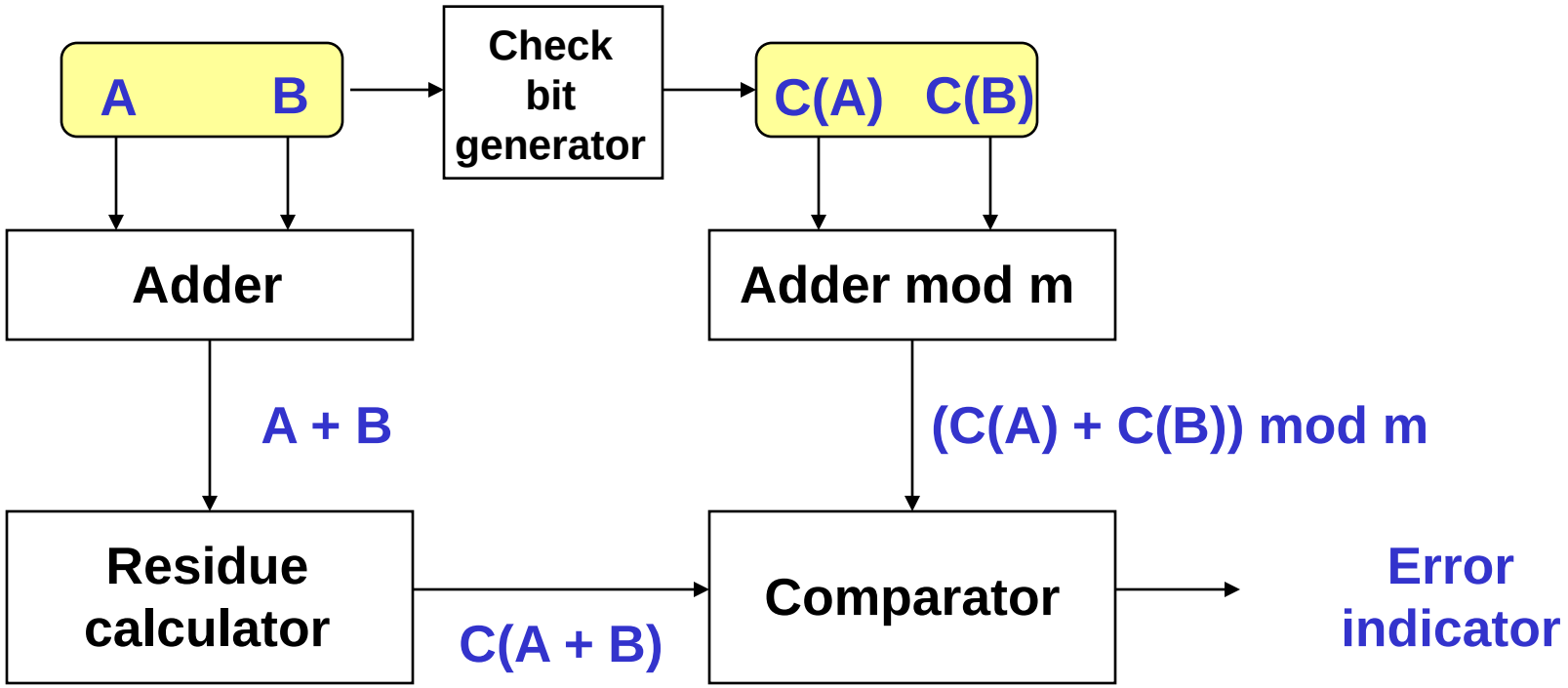
Error!

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



Summary

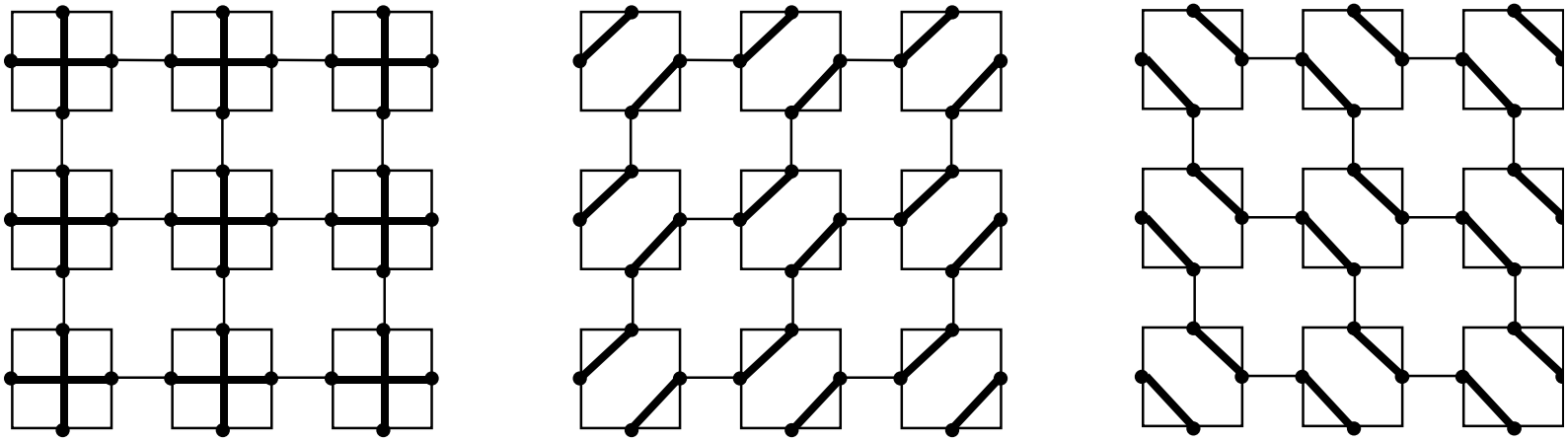
- LFSR pattern generator and MISR response compactor – preferred BIST methods
- BIST has overheads: test controller, extra circuit delay, Input MUX, pattern generator, response compactor, DFT to initialize circuit & test the test hardware
- BIST benefits:
  - At-speed testing for delay & stuck-at faults
  - Drastic ATE cost reduction
  - Field test capability
  - Faster diagnosis during system test
  - Less effort to design testing process
  - Shorter test application times

Testing of Networks-on-Chip (NoC)

- Consider a mesh-like topology of NoC consisting of
  - switches (routers),
  - wire connections between them and
  - slots for SoC resources, also referred to as tiles.
- Other types of topological architectures, e.g. honeycomb and torus may be implemented and their choice depends on the constraints for low-power, area, speed, testability
- The resource can be a processor, memory, ASIC core etc.
- The network switch contains buffers, or queues, for the incoming data and the selection logic to determine the output direction, where the data is passed (upward, downward, leftward and rightward neighbours)

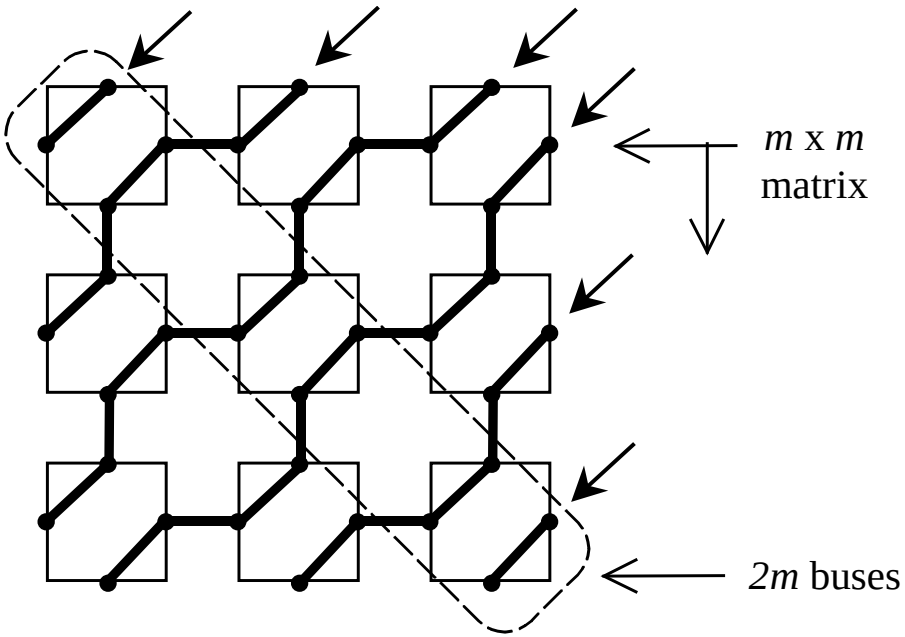
Testing of Networks-on-Chip

- Useful knowledge for testing NoC network structures can be obtained from the interconnect testing of other regular topological structures
- The test of wires and switches is to some extent analogous to testing of interconnects of an FPGA
- a switch in a mesh-like communication structure can be tested by using only three different configurations



Testing of Networks-on-Chip

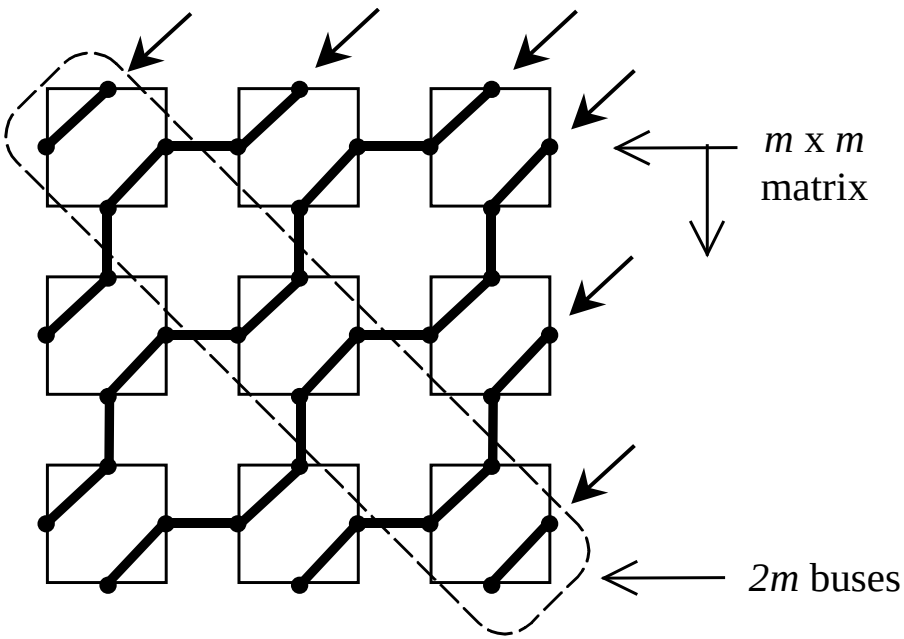
Concatenated bus concept



- **Arbitrary short and open in an *n*-bit bus can be tested by $\log_2(n)$ test patterns**
- **When testing the NoC interconnects we can regard different paths through the interconnect structures as one single concatenated bus**
- **Assuming we have a NoC, whose mesh consists of $m \times m$ switches, we can view the test paths through the matrix as a wide bus of $2mn$ wires**

Testing of Networks-on-Chip

Concatenated bus concept



- **The stuck-at-0 and stuck-at-1 faults are modeled as shorts to Vdd and ground**
- **Thus we need two extra wires, which makes the total bitwidth of the bus $2mn + 2$ wires.**
- **From the above facts we can find that $3[\log_2(2mn+2)]$ test patterns are needed in order to test the switches and the wiring in the NoC**

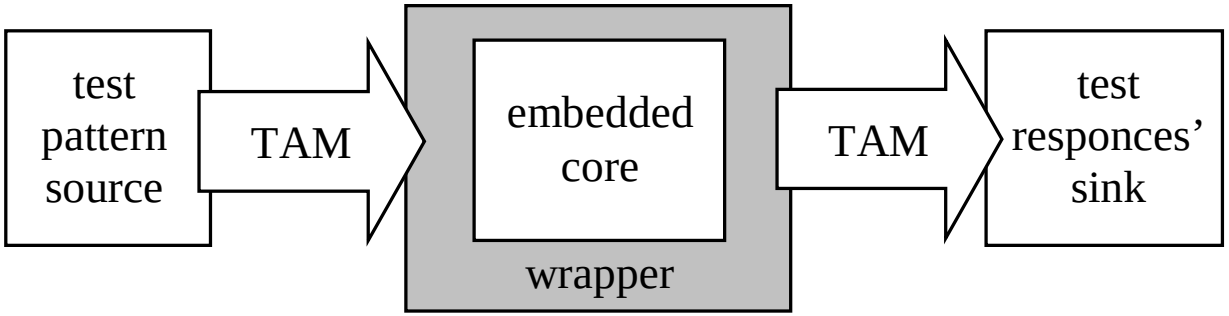
Testing of Networks-on-Chip

$3[\log_2(2mn+2)]$ test patterns needed

	Bus	*Test*	*Detected faults*
	0	0 0 0	Stuck-at-1
6 wires tested	1	0 0 1	All opens and shorts
	2	0 1 0	
	3	0 1 1	
	4	1 0 0	
	5	1 0 1	
	6	1 1 0	
	7	1 1 1	Stuck-at-0

IEEE P1500 standard for core test

- **The following components are generally required to test embedded cores**
  - ***Source*** **for application of test stimuli and a *sink* for observing the responces**
  - ***Test Access Mechanisms*** **(TAM) to move the test data from the source to the core inputs and from the core outputs to the sink**
  - ***Wrapper*** **around the embedded core**



IEEE P1500 standard for core test

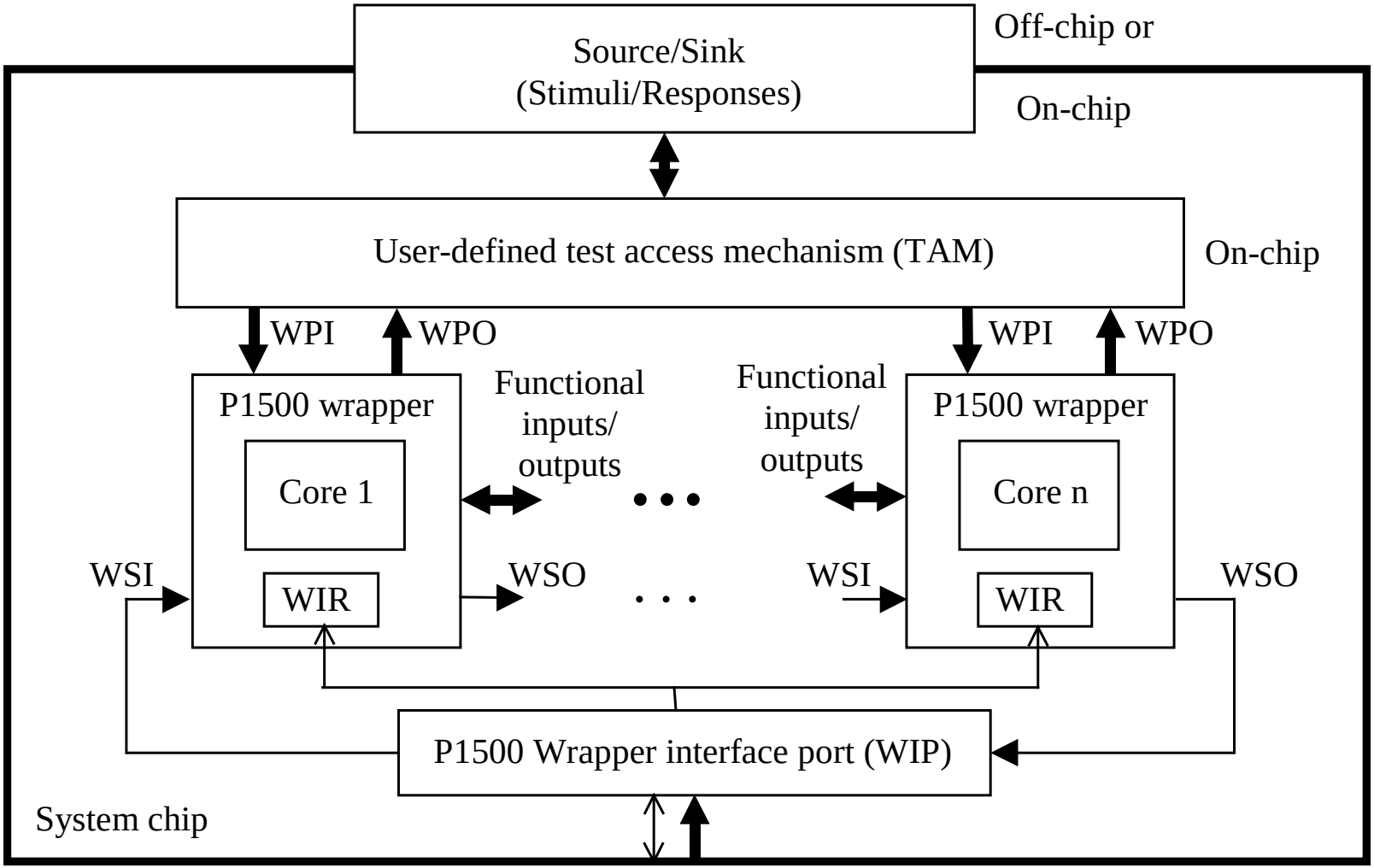
- **The two most important components of the P1500 standard are**
  - ***Core test language* (CTL) and**
  - ***Scalable core test architecture***
- **Core Test Language**
  - **The purpose of it is to standardize the core test knowledge transfer**
  - **The CTL file of a core must be supplied by the core provider**
  - **This file contains information on how to**
    - **instanciate a wrapper,**
    - **map core ports to wrapper ports,**
    - **and reuse core test data**

IEEE P1500 standard for core test

Core test architecture

- It standardizes only the wrapper and the interface between the wrapper and TAM, called Wrapper Interface Port or (WIP)
- The P1500 TAM interface and wrapper can be viewed as an extension to IEEE Std. 1149.1, since
  - the 1149.1 TAP controller is a P1500-compliant TAM interface,
  - and the boundary-scan register is a P1500-compliant wrapper
- Wrapper contains
  - an instruction register (WIR),
  - a wrapper boundary register consisting of wrapper cells,
  - a bypass register and some additional logic.
- Wrapper has to allow normal functional operation of the core plus it has to include a 1-bit serial TAM.
- In addition to the serial test access, parallel TAMs may be used.

IEEE P1500 standard for core test



Theory of LFSR: Galois Field

LFSR as a Galois field:

- ***Galois field*** **(mathematical system) G(p^n):**
  - **Multiplication by *x* same as right shift of LFSR**
  - **Addition operator is XOR ($\oplus$)**
- **T_S companion matrix:**
  - **1st column 0, except *n*-th element which is always 1 (X_0 always feeds X_{n-1})**
  - **Rest of row *n* – feedback coefficients h_i**
  - **Rest is identity matrix *I* – means a right shift**
- **Near-exhaustive (maximal length) LFSR**
  - **Cycles through 2^n – 1 states (excluding all-0)**
  - **one pattern of *n* 1's, two of *n*-1 consecutive 0's**