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

Source: Elcoteq

Source: Intel
Technical University Tallinn, ESTONIA

**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 ↑

Embedded Tester

Test Controller

Tester Memory

Core 1

BIST

Test access mechanism

Core 3

BIST

Core 4

BIST

Core 5

BIST

System on Chip

Technical University Tallinn, ESTONIA
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**

<table>
<thead>
<tr>
<th>Level</th>
<th>Design and test</th>
<th>Fabrication</th>
<th>Manuf. Test</th>
<th>Maintenance test</th>
<th>Diagnosis and repair</th>
<th>Service interruption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chips</td>
<td>+ / -</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boards</td>
<td>+ / -</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>System</td>
<td>+ / -</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

+  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

Source: VLSI Test: Bushnell-Agrawal
Technical University Tallinn, ESTONIA
Built-In Self-Test

Test per Scan:

**Initial test set:**

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

**Test application:**

1100 T 1010 T 0101 T 1001 T

Number of clocks = $4 \times 4 + 4 = 20$

- Assumes existing scan architecture
- Drawback:
  - Long test application time
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 register 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
  - T1 T4 T3 T2
  - Number of clocks = 10
**BILBO BIST Architecture**

*Working modes:*

<table>
<thead>
<tr>
<th>B1</th>
<th>B2</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Reset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Flip-flop (normal)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Scan mode</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Test mode</td>
</tr>
</tbody>
</table>

*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

Source: VLSI Test: Bushnell-Agrawal
Pattern Generation

- Store in ROM – too expensive
- Exhaustive
- Pseudo-exhaustive
- \textit{Pseudo-random (LFSR)} – Preferred method
- Binary counters – use more hardware than LFSR
- Modified counters
- Test pattern \textit{augmentation}
  - LFSR combined with a few patterns in ROM
  - \textit{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:

Polynomial: \( P(x) = x^4 + x^3 + 1 \)
Theory of LFSR

\[ y_j(t) = y_{j-1}(t-1) \quad \text{for} \quad j \neq 0 \]
\[ y_j(t) = y_0(t - j) \]
\[ y_j(t) = y_0(t)x^j \]

where \( j \) represents the time translation units

\[ y_0(t) = \sum_{j=1}^{j=n} c_j y_j(t) \]
\[ y_0(t) = \sum_{j=1}^{j=n} c_j y_0(t)x^j \]
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

Characteristic polynomial:

\[ y_0(t)P_n(x) = 0 \]

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

where \( P_n(x) = 1 + \sum_{j=1}^{j=n} c_j x^j \)
Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:

Polynomial: \( P(x) = x^4 + x^3 + 1 \)
Matrix Equation for Standard LFSR

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

\(X(t + 1) = T_s X(t) \quad (T_s \text{ is companion matrix})\)
Pseudorandom Test Generation

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

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

<table>
<thead>
<tr>
<th>t</th>
<th>x</th>
<th>x^2</th>
<th>x^3</th>
<th>x^4</th>
<th>t</th>
<th>x</th>
<th>x^2</th>
<th>x^3</th>
<th>x^4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
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 $k = 2^n - 1$
- 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:

$x^3 + x^2 + 1$

The polynomials will divide evenly the polynomial $x^7 + 1$, but not any one of $k<7$, hence, they are primitive

$x^3 + x + 1$

They are also reciprocal: coefficients are 1011 and 1101

Reducible polynomials (non-primitive):

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

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

$x^3 + x^2 + x + 1 = (x + 1)(x^2 + 1)$
### Theory of LFSR: Examples

**Comparison of test sequences generated:**

<table>
<thead>
<tr>
<th>Primitive polynomials</th>
<th>Non-primitive polynomials</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^3 + x + 1$</td>
<td>$x^3 + 1$</td>
</tr>
<tr>
<td>$x^3 + x^2 + 1$</td>
<td>$x^3 + x^2 + x + 1$</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>110</td>
<td>100</td>
</tr>
<tr>
<td>111</td>
<td>100</td>
</tr>
<tr>
<td>110</td>
<td>100</td>
</tr>
<tr>
<td>111</td>
<td>100</td>
</tr>
<tr>
<td>110</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td>111</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td>010</td>
<td>100</td>
</tr>
<tr>
<td>011</td>
<td>100</td>
</tr>
<tr>
<td>001</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

The table compares sequences generated from primitive and non-primitive polynomials.
Theory of LFSR: Examples

Reducible polynomial (non-primitive):

\[ x^3 + 1 = (x + 1)(x^2 + x + 1) \]

\[ x^3 + 1 \quad x + 1 \quad x^2 + x + 1 \]

<table>
<thead>
<tr>
<th>100</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \begin{array}{c}
100 \\
010 \\
001 \\
100
\end{array} \]

Primitive polynomial

Multiplication of two primitive polynomials:

\[ \frac{x^2 + x + 1}{x^2 + x + 1} \]
\[ \frac{x^3 + x^2 + x}{x^4 + x^3 + x^2} \]
\[ \frac{x^4 + x^2 + 1}{x^4 + x^2 + 1} \]

Is \[ x^4 + x^2 + 1 \]
a primitive polynomial?
Theory of LFSR: Examples

Is \( x^4 + x^2 + 1 \) a primitive polynomial?

Irreducible polynomial of degree \( n \) is characterized by:

- An odd number of terms including 1 term?
  Yes, it includes 3 terms

- Divisibility into \( 1 + x^k \), where \( k = 2^n - 1 \)
  No, there is remainder \( x^3 + 1 \)

\[ x^4 + x^2 + 1 \] is non-primitive?

Divisibility check:

\[
\begin{array}{c|c}
\hline
x^4 + x^2 + 1 & x^{11} + x^9 + x^5 + x^3 \\
\hline
x^{15} + 1 & x^{15} + x^{13} + x^{11} \\
& x^{13} + x^{11} + 1 \\
& x^{13} + x^{11} + x^9 \\
& x^9 + 1 \\
& x^9 + x^7 + x^5 \\
& x^7 + x^5 + 1 \\
& x^7 + x^5 + x^3 \\
& x^3 + 1 \\
\end{array}
\]
Theory of LFSR: Examples

Non-primitive polynomial
\[ x^4 + x^2 + 1 \]

<table>
<thead>
<tr>
<th>0001</th>
<th>1001</th>
<th>0110</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1100</td>
<td>1011</td>
</tr>
<tr>
<td>0100</td>
<td>1110</td>
<td>1101</td>
</tr>
<tr>
<td>1010</td>
<td>1111</td>
<td>0110</td>
</tr>
<tr>
<td>0101</td>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1001</td>
<td></td>
</tr>
</tbody>
</table>

Primitive polynomial
\[ x^4 + x + 1 \]

<table>
<thead>
<tr>
<th>0001</th>
<th>1011</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0101</td>
<td>0100</td>
</tr>
<tr>
<td>1100</td>
<td>1010</td>
<td>0010</td>
</tr>
<tr>
<td>1110</td>
<td>1101</td>
<td>0001</td>
</tr>
<tr>
<td>1111</td>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>0011</td>
<td></td>
</tr>
</tbody>
</table>
Theory of LFSR: Examples

Primitive polynomial
\[ x^4 + x + 1 \]

Zero generation:

<table>
<thead>
<tr>
<th></th>
<th>x^4</th>
<th>x^3</th>
<th>x^2</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>1011</td>
<td>1001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0101</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1010</td>
<td>0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>1101</td>
<td><strong>0001</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>0110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>x^4</th>
<th>x^3</th>
<th>x^2</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0000</strong></td>
<td>1011</td>
<td>1001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0101</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1010</td>
<td>0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>1101</td>
<td><strong>0001</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>0110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Technical University Tallinn, ESTONIA
Theory of LFSR: Reciprocal Polynomials

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

$$(X) = X^N P_N \left(\frac{1}{X}\right) = X^N \{1 + C_j X^{-j}\}$$

$$(X) = X^N + C_j X^{N-j} \quad \text{for } 1 \leq i \leq N$$

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 $N$

<table>
<thead>
<tr>
<th>$N$</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>2048</td>
</tr>
<tr>
<td>32</td>
<td>67108864</td>
</tr>
</tbody>
</table>

Table of primitive polynomials up to degree 31

<table>
<thead>
<tr>
<th>$N$</th>
<th>Primitive Polynomials</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,3,4,6,7,15,22</td>
<td>$1 + X + X^n$</td>
</tr>
<tr>
<td>5,11,21,29</td>
<td>$1 + X^2 + X^n$</td>
</tr>
<tr>
<td>10,17,20,25,28,31</td>
<td>$1 + X^3 + X^n$</td>
</tr>
<tr>
<td>9</td>
<td>$1 + X^4 + X^n$</td>
</tr>
<tr>
<td>23</td>
<td>$1 + X^5 + X^n$</td>
</tr>
<tr>
<td>18</td>
<td>$1 + X^7 + X^n$</td>
</tr>
<tr>
<td>8</td>
<td>$1 + X^2 + X^3 + X^4 + X^n$</td>
</tr>
<tr>
<td>12</td>
<td>$1 + X + X^3 + X^4 + X^n$</td>
</tr>
<tr>
<td>13</td>
<td>$1 + X + X^4 + X^6 + X^n$</td>
</tr>
<tr>
<td>14, 16</td>
<td>$1 + X + X^3 + X^4 + X^n$</td>
</tr>
</tbody>
</table>
# Theory of LFSR: Primitive Polynomials

## Number of PP of degree \( n \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>2048</td>
</tr>
<tr>
<td>32</td>
<td>67108864</td>
</tr>
</tbody>
</table>

## Examples of PP (exponents of terms):

<table>
<thead>
<tr>
<th>( n )</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1 0</td>
</tr>
<tr>
<td>3</td>
<td>1 0</td>
</tr>
<tr>
<td>4</td>
<td>1 0</td>
</tr>
<tr>
<td>5</td>
<td>2 0</td>
</tr>
<tr>
<td>6</td>
<td>1 0</td>
</tr>
<tr>
<td>7</td>
<td>1 0</td>
</tr>
<tr>
<td>8</td>
<td>6 5 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n )</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>4 0</td>
</tr>
<tr>
<td>10</td>
<td>3 0</td>
</tr>
<tr>
<td>11</td>
<td>2 0</td>
</tr>
<tr>
<td>12</td>
<td>7 4 3 0</td>
</tr>
<tr>
<td>13</td>
<td>4 3 1 0</td>
</tr>
<tr>
<td>14</td>
<td>12 11 1 0</td>
</tr>
<tr>
<td>15</td>
<td>1 0</td>
</tr>
<tr>
<td>16</td>
<td>5 3 2 0</td>
</tr>
</tbody>
</table>
Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence

Given test sequence:
(1) 100x0
(2) x1010
(3) 10101
(4) 01111

Creation of the shortest bit-stream:
10010 1 01111

Expected shortest LFSR sequence:
01111 (4)
10111
01011
10101 (3)
01010 (2)
00101
10010 (1)

Seed
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)

We are looking for values of $x_i$:

System of linear equations:

\[
\begin{bmatrix}
01111 \\
10111 \\
01011 \\
10101 \\
01010 \\
00101 \\
\end{bmatrix} \times \begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
\end{bmatrix} = \begin{bmatrix}
1 \\
0 \\
1 \\
0 \\
0 \\
1 \\
\end{bmatrix}
\]
### Deterministic Synthesis of LFSR

**Generation of the polynomial and seed for the given test sequence**

System of linear equations:

$$x_1 x_2 x_3 x_4 x_5 = 1, 0, 1, 0, 0, 1$$

Solving the equation by Gaussian elimination:

<table>
<thead>
<tr>
<th></th>
<th>01111</th>
<th>1</th>
<th>01000</th>
<th>10000</th>
<th>00100</th>
<th>00010</th>
<th>00001</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01111</td>
<td>1</td>
<td>01000</td>
<td>10000</td>
<td>00100</td>
<td>00010</td>
<td>00001</td>
</tr>
<tr>
<td>2</td>
<td>10111</td>
<td>0</td>
<td>1</td>
<td>4,6</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>01011</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>10101</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>01010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>00101</td>
<td>1</td>
<td>0</td>
<td>1,2,3,4,6</td>
<td>1,2,4,6</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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

![Graph showing fault coverage over time with breakpoints](image-url)
Fault coverage is rapidly growing:

- **100%**
- **93.75%**
- **87.5%**
- **75%**
- **50%**

**Truth table:**

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>00...000</td>
<td>01 01 01...101</td>
</tr>
<tr>
<td>00...001</td>
<td>00 11 00...011</td>
</tr>
<tr>
<td>00...010</td>
<td>00 00 11...111</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11...111</td>
<td>00 00 00...111</td>
</tr>
</tbody>
</table>

- **Number of patterns**
- **Number of functions**

Combinational circuit under test

\[
2^n \times 2^{2^n-1} \text{ tested} \quad 50%!
\]
BIST: Fault Coverage

Pseudorandom Test generation by LFSR:

Motivation of using LFSR:
- low generation cost
- high initial efficiency

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

$$\frac{1}{2^{2^n} - 1} \sum_{i=1}^{m} 2^{2^n - i}, \quad 1 \leq m \leq 2^n$$
BIST: Different Techniques

Pseudorandom Test generation by LFSR:

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

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

$$1 \leq m \leq 2^n$$

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:

- Faults covered by 1. pattern
- Faults covered by 2. pattern
- Faults covered by 3. pattern
- 4. pattern

Not tested faults

Fault coverage

100%

Number of patterns

Combination circuit under test

Technical University Tallinn, ESTONIA
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

Faulty functions covered by:
- 1. pattern
- 2. pattern
- 3. pattern
- 4. pattern

Faults covered by:
- 1. pattern
- 2. pattern
- 3. pattern
- 4. pattern
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

Exhaustive test

Pseudo-exhaustive sequential

Pseudo-exhaustive parallel

\( 2^{16} = 65536 \gg 4 \times 16 = 64 > 16 \)

Primitive polynomials
## 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

---

<table>
<thead>
<tr>
<th></th>
<th>c₀</th>
<th>a₀</th>
<th>b₀</th>
<th>c₁</th>
<th>a₁</th>
<th>b₁</th>
<th>c₂</th>
<th>a₂</th>
<th>b₂</th>
<th>c₃</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- 0-bit testing
- 1-bit testing
- 2-bit testing
- 3-bit testing ... etc
Testing carry-lookahead adder

General expressions:

\[ G_i = a_i b_i \quad P_i = a_i \overline{b_i} \lor a_i b_i \quad C_n = G_n \lor P_n C_{n-1} \]

\[ C_n = G_n \lor P_n (G_{n-1} \lor P_{n-1} C_{n-2}) = G_n \lor P_n G_{n-1} \lor P_n P_{n-1} C_{n-2} \]

n-bit carry-lookahead adder:

\[ C_1 = G_1 \lor P_1 C_0 = a_1 b_1 \lor a_1 \overline{b_1} C_0 \lor a_1 b_1 C_0 = f(a_1, b_1, C_0) \]

\[ C_3 = G_3 \lor P_3 G_2 \lor P_3 P_2 G_1 \lor \overline{P_3 P_2 P_1 C_0} \]

\[ P_3 P_2 P_1 C_0 = (a_3 \overline{b_3} \lor a_3 b_3)(a_2 \overline{b_2} \lor a_2 b_2)(a_1 \overline{b_1} \lor a_1 b_1) C_0 \]
Testing carry-lookahead adder

\[
P_3P_2P_1C_0 = (\overline{a_3b_3} \lor \overline{a_3b_3}) (\overline{a_2b_2} \lor \overline{a_2b_2}) (\overline{a_1b_1} \lor \overline{a_1b_1})C_0 \quad R
\]

<table>
<thead>
<tr>
<th>Testing (\equiv 0)</th>
<th>0 0 1 1</th>
<th>0 0 1 1</th>
<th>0 0 1 1</th>
<th>0 0 1 1</th>
<th>0 0 1 1</th>
<th>0 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td></td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Testing (\equiv 1)</th>
<th>0 1 1 0</th>
<th>0 1 1 0</th>
<th>0 1 1 0</th>
<th>0 1 1 0</th>
<th>0 1 1 0</th>
<th>0 1 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 0 0 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>0 1 1 0</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>0 1 1 0</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>1 0 0 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>1 0 0 1</td>
<td>0 1 1 0</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>0 1 1 0</td>
<td>0 1 1 0</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

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
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 efficiency

**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.
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

LFSR

Weight select

Desired weighted value

Scan-IN

1/2

1/4

1/8

1/16

MUX

&

&

&
**Problem:** random-pattern-resistant faults

**Solution:** weighted pseudorandom testing

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

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.
NCV - noncontrolling value

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

\[ R_I = \frac{NDI_G}{NDI_I} \]

- \( R_I \) - the desired ratio of the NCV to the controlling value for each gate input
BIST: Weighted pseudorandom test

Example:

R_1 = \frac{N\text{DI}_G}{N\text{DI}_I} = \frac{6}{1} = 6
R_2 = \frac{N\text{DI}_G}{N\text{DI}_I} = \frac{6}{2} = 3
R_3 = \frac{N\text{DI}_G}{N\text{DI}_I} = \frac{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

<table>
<thead>
<tr>
<th>Function</th>
<th>W0G</th>
<th>W1G</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>W0G</td>
<td>R_l * W1G</td>
</tr>
<tr>
<td>NAND</td>
<td>W1G</td>
<td>R_l * W0G</td>
</tr>
<tr>
<td>OR</td>
<td>R_l * W0G</td>
<td>W1G</td>
</tr>
<tr>
<td>NOR</td>
<td>R_l * W1G</td>
<td>W0G</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Function</th>
<th>$W_{O_i}$</th>
<th>$W_{I_i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>$R_i \cdot W_{O_G}$</td>
<td>$W_{1_G}$</td>
</tr>
<tr>
<td>NOR</td>
<td>$R_i \cdot W_{1_G}$</td>
<td>$W_{O_G}$</td>
</tr>
</tbody>
</table>
BIST: Weighted pseudorandom test

**Calculation of signal probabilities:**

For $\text{PI}_1$:
- $W_0 = 6$, $W_1 = 1$
- $P_1 = 1/7 = 0.15$

For $\text{PI}_2$ and $\text{PI}_3$:
- $W_0 = 2$, $W_1 = 3$
- $P_1 = 3/5 = 0.6$

For $\text{PI}_4$ - $\text{PI}_6$:
- $W_0 = 3$, $W_1 = 2$
- $P_1 = 2/5 = 0.4$

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

$WF = \max \{W_0, W_1\} / \min \{W_1, W_0\}$

**Probability:**

$P = WF / (WF + 1)$
BIST: Weighted pseudorandom test

Calculation of signal probabilities:

For $PI_1$:
$$P_1 = 0.15$$

For $PI_2$ and $PI_3$:
$$P_1 = 0.6$$

For $PI_4$ - $PI_6$:
$$P_1 = 0.4$$

Probability of detecting the fault $\equiv 1$ at the input 3 of the gate $G$:

1) equal probabilities ($p = 0.5$):
$$P = 0.5 \times (0.25 + 0.25 + 0.25) \times 0.5^3 = 0.5 \times 0.75 \times 0.125 = 0.046$$

2) weighted probabilities:
$$P = 0.85 \times (0.6 \times 0.4 + 0.4 \times 0.6 + 0.6^2) \times 0.6^3 = 0.85 \times 0.84 \times 0.22 = 0.16$$
BIST: Response Compression

1. Parity checking

\[ P(R) = \left( \sum_{i=1}^{m} r_i \right) \mod 2 \]

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:

![Diagram of LFSRs]

Polynomial: \( P(x) = x^4 + x^3 + 1 \)
BIST: Signature Analysis

Signature analyzer:

Response string

Response in compacted by LFSR

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

Polynomial: \( P(x) = 1 + x^3 + x^4 \)
The principles of CRC (Cyclic Redundancy Coding) are used in LFSR based test response compaction.

Coding theory treats binary strings as polynomials:

\[ R = r_{m-1} r_{m-2} \ldots r_1 r_0 \quad \text{m-bit binary sequence} \]

\[ R(x) = r_{m-1} x^{m-1} + r_{m-2} x^{m-2} + \ldots + r_1 x + r_0 \quad \text{polynomial in } x \]

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:

- linear 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:

\[ n = 2m + r, \quad r \in \{0,1\} \quad \text{or} \quad r \equiv n \pmod{2} \]

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

**Examples:**
\[
\begin{align*}
  x^4 + x^3 + x + 1 \\
  + x^4 + x^2 + x \\
  \hline
  x^3 + x^2 + 1
\end{align*}
\]
\[
\begin{align*}
  x^4 + x^3 + x + 1 \\
  * x + 1 \\
  \hline
  x^5 + x^4 + x^2 + x \\
  \hline
  x^4 + x^3 + x + 1 \\
  \hline
  x^5 + x^3 + x^2 + 1
\end{align*}
\]
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:
  
  \[
  \begin{array}{c}
  x^2 + x + 1 \\
  x^2 + 1 \\
  x^2 + x + 1 \\
  x^4 + x^3 + x^2 \\
  x^4 + x^3 + x + 1 \\
  \end{array}
  \]
**Theory of LFSR**

**Characteristic Polynomials:**

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

**Division of polynomials**

\[
\begin{array}{c|ccccc}
& x^2 + x + 1 & \multicolumn{4}{c}{x^4 + x^3} \\
\hline
x^2 + 1 & x^4 & + x^2 & x^3 + x^2 & + 1 \\
\hline
x^3 + x^2 & + x \\
x^2 & + 1 \\
x & + 1 \\
\end{array}
\]

Quotient: \(x^2 + x + 1\)  
Dividend: \(x^4 + x^3 + 1\)  
Remainder: \(x\)
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 signature of the test response string $P(x)$ from the UUT.

Signature is the CRC code word.

Example:

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

$P(x) = 1010111$  

$G(x) = 101011$  

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

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

$\text{Signature} = 001101$
BIST: Signature Analysis

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 $x^5 = x^3 + x + 1$.

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

$$010001$$

Shifted into LFSR

Response

$$001101$$

$R(x) = x^3 + x^2 + 1$
BIST: Signature Analysis

Aliasing:

\[ k = 2^L \]

\[ N < < L \]

\( L \) - test length

\( N \) - number of stages in Signature Analyzer

All possible responses

All possible signatures

Faulty response

Correct response
BiST: Signature Analysis

**Aliasing:**

- **UUT** → **Response** → **SA**

- \( k = 2^L \) - 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

**Probability of aliasing:**

\[
P = \frac{2^{L-N} - 1}{2^L - 1}
\]

For \( L \gg 1 \),

\[
P = \frac{1}{2^N}
\]
BIST: Signature Analysis

Parallel Signature Analyzer:

UUT

Single Input Signature Analyser

\[ \text{X}^4 \rightarrow \text{X}^3 \rightarrow \text{X}^2 \rightarrow \text{X} \rightarrow 1 \]

Multiple Input Signature Analyser (MISR)

UUT

\[ \text{X}^4 \rightarrow \text{X}^3 \rightarrow \text{X}^2 \rightarrow \text{X} \rightarrow 1 \]
BIST: Signature Analysis

Signature calculating for multiple outputs:

LFSR - Test Pattern Generator → Combinational circuit → Multiplexer → LFSR - Signature analyzer

LFSR - Test Pattern Generator → Combinational circuit → Multiplexer → LFSR - Signature analyzer
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

- LFSR - Test Pattern Generator
- Combinational circuit
- LFSR - Signature analyzer

Joint TPG and SA:

CSTP - Circular Self-Test Path:

- LFSR - Test Pattern Generator & Signature analyser
- Combinational circuit
BIST: Circular Self-Test Architecture
BIST: Circular Self-Test Path
Concurrent testing:

LFSR, CSTP $\rightarrow$ M2 $\rightarrow$ MISR1
M2 $\rightarrow$ M5 $\rightarrow$ MISR2 (Functional BIST)
CSTP $\rightarrow$ M3 $\rightarrow$ CSTP
LFSR2 $\rightarrow$ M4 $\rightarrow$ BILBO
BIST Architectures

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

**LOCST:** LSSD On-Chip Self-Test

- Test Pattern Generator
- Scan chain
- CUT
- ... Scan chain
- CUT
- MISR

- BS
- Scan Path
- CUT
- TPG
- SA
- Test Controller
- IC
- SO
- Error
Figure 1: Scan-based BIST for $n$-detection with weighted scan-enable signals and scan forest.

Copyright: D.Xiang 2003
Software BIST

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

Software based test generation:

```plaintext
load (LFSRj);
for (i=0; i<Nj; i++)
...
end;
```
Problems with BIST

The main motivations of using random patterns are:
- low generation cost
- high initial efficiency

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 efficiency

Problem: Low fault coverage

Patterns from LFSR:

Pseudorandom test window:

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?

Pseudorandom Test

Determ. Test
Optimization of Hybrid BIST

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

- **FAST estimation**: Number of remaining faults after applying \( k \) pseudorandom test patterns \( r_{\text{NOT}}(k) \)
- **SLOW analysis**: Number of pseudorandom test patterns applied, \( k \)
- **PR test length**: \( k \)
- **Det. Test**: Pseudorandom Test

**Table**

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

\[
\text{Faults detected by pseudo-random patterns} \\
\text{Faults to be detected by deterministic patterns} \\
\text{New detected faults}
\]
Calculation of the Deterministic Test Cost

Fault table based approach
A deterministic test set with fault table is calculated
For each breakpoint of P-sequence, the fault table is updated

Fault table based:

ATPG

Fault table update

All PR patterns?

No

Next PR pattern

Yes

End

Task for fault simulator

Updated fault table

Faults detected

By

pseudorandom patterns

Fault table for full deterministic test

To be detected faults
Experimental Data: HBIST Optimization

Finding optimal brakepoint in the pseudorandom sequence:

<table>
<thead>
<tr>
<th>Pseudorandom Test</th>
<th>Det. Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{\text{OPT}}$</td>
<td>$L_{\text{MAX}}$</td>
</tr>
<tr>
<td></td>
<td>$S_{\text{OPT}}$</td>
</tr>
<tr>
<td></td>
<td>$S_{\text{MAX}}$</td>
</tr>
</tbody>
</table>

Optimized hybrid test process:

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

The motivation of using random patterns is:
- low generation cost
- high initial efficiency

Problem: low fault coverage → long PR test

Solution: many seeds:

- Pseudorandom test:
  - 1
  - 2^{n-1}

- Hard to test faults:
  - 1
  - 2^{n-1}
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

Using many seeds:

Pseudorandom test:

1 \to 2^n-1

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
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 included into sequence and updated

The procedure ends when 100% fault coverage is achieved
Cost Curves for Hybrid BIST with Reseeding

Two possibilities for reseeding:

- **Constant block length** (less HW overhead)
- **Dynamic block length** (more HW overhead)

![Graph showing test length L and memory cost M against block size](image-url)
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

<table>
<thead>
<tr>
<th>Data</th>
<th>B1</th>
<th>B2</th>
<th>Total</th>
<th>B1</th>
<th>B2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>4/2</td>
<td>13.21</td>
<td>15.09</td>
<td>14.15</td>
<td>35.14</td>
<td>40.57</td>
<td>29.72</td>
</tr>
<tr>
<td>7/2</td>
<td>21.23</td>
<td>16.98</td>
<td>19.10</td>
<td>38.44</td>
<td>47.64</td>
<td>29.25</td>
</tr>
<tr>
<td>6/3</td>
<td>19.34</td>
<td>31.6</td>
<td>25.47</td>
<td>41.04</td>
<td>39.62</td>
<td>42.45</td>
</tr>
<tr>
<td>8/2</td>
<td>25.47</td>
<td>10.38</td>
<td>17.92</td>
<td>32.07</td>
<td>40.57</td>
<td>25.00</td>
</tr>
<tr>
<td>9/4</td>
<td>8.96</td>
<td>5.66</td>
<td>7.31</td>
<td>36.56</td>
<td>47.64</td>
<td>25.47</td>
</tr>
<tr>
<td>9/3</td>
<td>32.55</td>
<td>26.89</td>
<td>29.72</td>
<td>43.63</td>
<td>46.07</td>
<td>40.57</td>
</tr>
<tr>
<td>12/6</td>
<td>13.44</td>
<td>8.02</td>
<td>18.87</td>
<td>36.08</td>
<td>39.62</td>
<td>32.55</td>
</tr>
<tr>
<td>14/2</td>
<td>18.16</td>
<td>25.00</td>
<td>11.32</td>
<td>37.50</td>
<td>49.06</td>
<td>25.94</td>
</tr>
<tr>
<td>15/3</td>
<td>29.48</td>
<td>31.13</td>
<td>27.83</td>
<td>47.88</td>
<td>50.00</td>
<td>45.75</td>
</tr>
<tr>
<td>2/4</td>
<td>7.8</td>
<td>7.55</td>
<td>8.02</td>
<td>29.01</td>
<td>20.75</td>
<td>33.02</td>
</tr>
<tr>
<td>Aver.</td>
<td>18.96</td>
<td>17.83</td>
<td>17.97</td>
<td>37.74</td>
<td>42.15</td>
<td>32.97</td>
</tr>
<tr>
<td>Gain</td>
<td>1.0</td>
<td>1.03</td>
<td>1.0</td>
<td>2.0</td>
<td>2.4</td>
<td>1.8</td>
</tr>
</tbody>
</table>

FBIST: collection and analysis of samples during the working mode

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

Technical University Tallinn, ESTONIA
Example: Functional BIST

Functional BIST quality analysis

Samples from N=120 cycles

Register block
Control

ALU

Signature analyser

DB=64
K
Data

SB=105

K*N

Fault simulator

Fault coverage

Data compression:
N*SB / DB = 197

Functional test

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

- **Register block**
- **ALU**
- **MUX**
- **Signature analyser**
- **M**
- **Automatic Test Pattern Generator**
- **Data**
- **K**
- **Test patterns are stored in the memory**

Random resistant faults

Deterministic test set
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
- \( \alpha, \beta \) - weights of time and memory expenses

**Problem:** minimize \( C_{Total} \)
Functional Self-Test with DFT

Example: N-bit multiplier

- Register block
- ALU
- Signature analyser
- EXOR
- MUX
- F

N cycles

Improving controllability

Improving observability

Data K
Hybrid BIST for Multiple Cores

Embedded tester for testing multiple cores

SoC

Embedded Tester

Test Controller

Tester Memory

C2670

Test access mechanism

BIST

C3540

BIST

C1908

BIST

C880

BIST

C1355

BIST
Hybrid BIST for Multiple Cores

Deterministic test (DT)

Pseudorandom test (PT)

How to pack knapsack?

How to compress the test sequence?

The optimal test set for each core

<table>
<thead>
<tr>
<th>Core</th>
<th>Random</th>
<th>Det.</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1908</td>
<td>105</td>
<td>123</td>
</tr>
<tr>
<td>C880</td>
<td>121</td>
<td>48</td>
</tr>
<tr>
<td>C2670</td>
<td>444</td>
<td>77</td>
</tr>
<tr>
<td>C1355</td>
<td>121</td>
<td>52</td>
</tr>
<tr>
<td>C3540</td>
<td>297</td>
<td>110</td>
</tr>
</tbody>
</table>

Technical University Tallinn, ESTONIA
**Cost of BIST:**  
\[ C_{\text{TOTAL}} = \alpha k + \beta t(k) \]

**FAST estimation**  
Number of remaining faults after applying k pseudorandom test patterns \( r_{\text{NOT}}(k) \)

**SLOW analysis**  
Number of pseudorandom test patterns applied, \( k \)

**Det. Test**

**Pseudorandom Test**

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

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 \( TDE_k(i) \) will be the estimation of the DT length

Deterministic test length estimation for a single core
Deterministic Test Cost Estimation

Total cost calculation of core costs:

<table>
<thead>
<tr>
<th>Core name</th>
<th>Memory usage</th>
<th>Deterministic time</th>
</tr>
</thead>
<tbody>
<tr>
<td>c499</td>
<td>1353</td>
<td>33</td>
</tr>
<tr>
<td>c880</td>
<td>480</td>
<td>8</td>
</tr>
<tr>
<td>c1355</td>
<td>1025</td>
<td>25</td>
</tr>
<tr>
<td>c1908</td>
<td>363</td>
<td>11</td>
</tr>
<tr>
<td>c5315</td>
<td>2136</td>
<td>12</td>
</tr>
<tr>
<td>c6288</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c432</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Memory usage: 5357 bits

Solution
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

Source: VLSI Test: Bushnell-Agrawal
Broadcasting Test Patterns in BIST

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

Traditional single scan design

Broadcast test architecture

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

Examples of connection possibilities in Broadcasting BIST:

- **j-to-j connections**
  - CUT 1
  - CUT 2

- **Random connections**
  - CUT 1
  - CUT 2
Broadcasting Test Patterns in BIST

Scan configurations in Broadcasting BIST:

- Common MISR
- Individual and multiple MISRs
Fault-Model Free Fault 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:**

- Test Pattern Generator (TPG)
- Circuit Under Diagnosis (CUD)
- Output Response Analyser (ORA)
- BISD Control Unit
- Test patterns

**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

Diagram:

- Test Pattern Generator (TPG)
- Circuit Under Test (CUT)
- Output Response Analyser (ORA)
- BIST Control Unit
- Test patterns
  - Number
  - Signature
  - Faults

Diagnosis procedure:

1. test
2. test
3. test

- Faulty signature
- Correct signature

Technical University Tallinn, ESTONIA
Built-In Fault Diagnosis

<table>
<thead>
<tr>
<th>№</th>
<th>All faults</th>
<th>New faults</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>16.67%</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>10</td>
<td>50.00%</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>1</td>
<td>53.33%</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>1</td>
<td>56.67%</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>3</td>
<td>66.67%</td>
</tr>
<tr>
<td>6</td>
<td>21</td>
<td>1</td>
<td>70.00%</td>
</tr>
<tr>
<td>7</td>
<td>25</td>
<td>4</td>
<td>83.33%</td>
</tr>
<tr>
<td>8</td>
<td>26</td>
<td>1</td>
<td>86.67%</td>
</tr>
<tr>
<td>9</td>
<td>29</td>
<td>3</td>
<td>96.67%</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>1</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

Pseudorandom test fault simulation

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

<table>
<thead>
<tr>
<th>№</th>
<th>All faults</th>
<th>New faults</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>16.67%</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>10</td>
<td>50.00%</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>1</td>
<td>53.33%</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>1</td>
<td>56.67%</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>3</td>
<td>66.67%</td>
</tr>
<tr>
<td>6</td>
<td>21</td>
<td>1</td>
<td>70.00%</td>
</tr>
<tr>
<td>7</td>
<td>25</td>
<td>4</td>
<td>83.33%</td>
</tr>
<tr>
<td>8</td>
<td>26</td>
<td>1</td>
<td>86.67%</td>
</tr>
<tr>
<td>9</td>
<td>29</td>
<td>3</td>
<td>96.67%</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>1</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

## 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:

Test pattern generator

Fault

CUD

SA₁  SA₂  SA₃

SA₁  SA₂  SA₃

SA₁  SA₂  SA₃

D₁  D₂  D₃  D₄  D₅  D₆  D₇
Built-In Fault Diagnosis

Diagnosis with multiple signatures:

<table>
<thead>
<tr>
<th>No</th>
<th>Codeword</th>
<th>Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h$</td>
<td>0 0 1</td>
<td>$R_1$</td>
</tr>
<tr>
<td>$i$</td>
<td>0 0 1</td>
<td>$R_1''$</td>
</tr>
<tr>
<td>$j$</td>
<td>0 1 1</td>
<td>$R_2$</td>
</tr>
<tr>
<td>$k$</td>
<td>0 1 1</td>
<td>$R_1''$, $R_2''$</td>
</tr>
<tr>
<td>$l$</td>
<td>1 1 1</td>
<td>$R_3$</td>
</tr>
<tr>
<td>$v$</td>
<td>1 1 1</td>
<td>$R_1'$, $R_2'$, $R_3'$</td>
</tr>
</tbody>
</table>

Diagnostic tree
Built-In Fault Diagnosis

BIST with multiple signature analyzers

Test pattern generator

Fault

CUD

Intersection using tests

Faulty signature

Correct signature

Intersection using SA-s

Faulty signature

SA1

SA2

SA3

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:

- Conditional fault
- Pattern fault
- Constrained SAF
- Single faulty signal

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

- Resistive bridge fault
Fault evidence:
for test pattern $t$

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

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

for full test $T$ (sum)

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

Copyright: H.J.Wunderlich 2007
Diagnosis of Fault Model Free Defects

Real test experiment

Circuit Under Diagnosis

Δτₜ

Δσₜ

ΔIₜ

Faulty machine FM(f)

Fault

Simulation

Test pattern t

Different classical fault cases

<table>
<thead>
<tr>
<th>Classic model</th>
<th>Iₜ</th>
<th>τₜ</th>
<th>γₜ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single SAF</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Multiple SAF</td>
<td>0</td>
<td>&gt;0</td>
<td>0</td>
</tr>
<tr>
<td>Single conditional SAF</td>
<td>&gt;0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Multiple cond. SAF</td>
<td>&gt;0</td>
<td>&gt;0</td>
<td>0</td>
</tr>
<tr>
<td>Delay fault</td>
<td>&gt;0</td>
<td>0</td>
<td>&gt;0</td>
</tr>
<tr>
<td>General case</td>
<td>&gt;0</td>
<td>&gt;0</td>
<td>&gt;0</td>
</tr>
</tbody>
</table>

Copyright: H.J. Wunderlich 2007

Technical University Tallinn, ESTONIA
Diagnosis of Fault Model Free Defects

Ranking
(on the top the most suspicious faults):

1. By increasing $\gamma_T$ (single SAF on top)
2. If $\gamma_T$ are equal then by decreasing $\sigma_T$
3. If $\gamma_T$ and $\sigma_T$ are equal then by increasing $l_T$

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

Example:

<table>
<thead>
<tr>
<th>SAF</th>
<th>$\gamma_T$</th>
<th>$\sigma_T$</th>
<th>$l_T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>0</td>
<td>42</td>
<td>0</td>
</tr>
<tr>
<td>$f_2$</td>
<td>30</td>
<td>42</td>
<td>15</td>
</tr>
<tr>
<td>$f_3$</td>
<td>30</td>
<td>42</td>
<td>25</td>
</tr>
<tr>
<td>$f_4$</td>
<td>30</td>
<td>42</td>
<td>30</td>
</tr>
<tr>
<td>$f_5$</td>
<td>30</td>
<td>36</td>
<td>38</td>
</tr>
<tr>
<td>$f_6$</td>
<td>38</td>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>$f_7$</td>
<td>38</td>
<td>23</td>
<td>23</td>
</tr>
</tbody>
</table>
Fault Diagnosis Without Fault Models

- Transistor level
- Logic level
- System level

Reverse defect mapping

Defective area
dy

Logic level

Error detection

Error (defective area) diagnosis

\[
x_1, x_2, x_3, x_4, x_5
\]

IN

\[
R_1, M_1, M_2, M_3, R_2, R_3
\]
Fault Model Free Fault Diagnosis

System network graph

Test response: 1 0 1 0

No match

Because of the unidirectional “distortions” of test responses, rectification is possible.

Diagnosis: s11

Rectified code: 1 0 1 0

Diagnostic table

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Fault Model Free Fault Diagnosis

System network graph

Test response: 1 0 1 0

No match

Because of the unidirectional “distortions” of test responses, rectification is possible.

Diagnosis: s11

Rectified code: 1 0 1 0

Diagnostic table

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Fault Tolerance: Error Detecting Codes

Examples:

Decimal digits:
- Eligible: 0, 1, 2, ..., 9
- Not eligible: 10, 11, ..., 15

Parity check:
- Parity bit
- Eligible
- Not eligible

<table>
<thead>
<tr>
<th>Parity check</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>
Error Detecting/Correcting Codes

Hamming distance between codes: Minimal number of bits how two codes differ from each other

Eligible codes

Parity check:

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Parity bit

d = 2

Eligible codes

Not eligible codes

Technical University Tallinn, ESTONIA
Error Detecting/Correcting Codes

Error detecting codes:
- Eligible codes
- Detection not possible
- Not eligible codes

Error correcting codes:
- Error correction is possible: direction is known

Eligible codes

$d=2$
- Error detection: direction unknown

$d=3$
- Correction not possible
Fault Tolerance: Error Correcting Codes

\[ d = 2e + 1 \]

- \(2e\) - error detection
- \(e\) - error correction

One error correction code:

\[ 2^c \geq q + c + 1 \]

- For addressing of the erroneous bit
- Error free

- Check bits
- Information bits
Fault Tolerance: One Error Correcting Code

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

Calculation of check sums:
\[
\sum_{k \in P_i} b_k = 0, \quad i = 1, \ldots, c
\]

\(P_i\) – 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:

Check bits

\[ b_2^i, i = 1, \ldots, c \]

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

Check bits have to be independently assigned

Analogy with fault diagnosis by using fault table:

<table>
<thead>
<tr>
<th>Received code</th>
<th>Test</th>
<th>Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 0 0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1 0 0 0 0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Initial code

Technical University Tallinn, ESTONIA
Fault Tolerant Communication System

- Initial code
- Check-bits generator
- Sender
- Receiver
- Checker
- Error indication
- Error correction code
- Received correct code
Error Detection in Arithmetic Operations

Residue codes

N – information bits
C = (N) mod m - check bits
m – residue of the code
p = ⌊log₂ m⌋ – number of check bits

Example

Information bits: I₂, I₁, I₀
m = 3, p = 2
Check bits: c₁, c₀
## Error Detection in Arithmetic Operations

### Addition:

<table>
<thead>
<tr>
<th>Information bits</th>
<th>Check bits</th>
<th>Result</th>
<th>Check</th>
<th>Mod3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>1 0</td>
<td>2.2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1</td>
<td>4.1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0 1 1 0</td>
<td>6.3</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

(6)mod3 = 0

### Multiplication:

<table>
<thead>
<tr>
<th>Information bits</th>
<th>Check bits</th>
<th>Result</th>
<th>Check</th>
<th>Mod3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>1 0</td>
<td>2.2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1</td>
<td>4.1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
<td>8.2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

(8)mod3 = 2

<table>
<thead>
<tr>
<th>Information bits</th>
<th>Check bits</th>
<th>Result</th>
<th>Check</th>
<th>Mod3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>1 0</td>
<td>2.2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1</td>
<td>4.1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>4.3</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

(4)mod3 = 1

(3)mod3 = 0

Error!
Error Detection in Arithmetic Operations

A + B → Adder

A + B → Residue calculator

A + B → Check bit generator

C(A) C(B) → Adder mod m

(C(A) + C(B)) mod m → Comparator

Error indicator
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 $3 \lceil \log_2(2mn+2) \rceil$ 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

<table>
<thead>
<tr>
<th>Bus</th>
<th>Test</th>
<th>Detected faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0</td>
<td>Stuck-at-1</td>
</tr>
<tr>
<td>1</td>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0 1 0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0 1 1</td>
<td>All opens and shorts</td>
</tr>
<tr>
<td>4</td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 0 1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 1 0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1 1 1</td>
<td>Stuck-at-0</td>
</tr>
</tbody>
</table>

6 wires tested

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

6 wires tested

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

6 wires tested
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 responses
  - **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

Source/Sink
(Stimuli/Responses)

User-defined test access mechanism (TAM)

P1500 wrapper
Core 1
WIR
WSI
WSO
WPI
WPO

P1500 Wrapper interface port (WIP)

P1500 wrapper
Core n
WIR
WSI
WSO
WPI
WPO

System chip
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