Built-In Self-Test

Outline

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

Cores have to be tested on chip

Source: Elcoteq

Source: Intel
Self-Test in Complex Digital Systems

Test architecture components:
- Test pattern source & sink
- Test Access Mechanism
- Core test wrapper

Solutions:
- Off-chip solution
  - need for external ATE
- Combined solution
  - mostly on-chip, ATE needed for control
- On-chip solution
  - BIST
Self-Test in Complex Digital Systems

Test architecture components:
• Test pattern source & sink
• Test Access Mechanism
• Core test wrapper

Solutions:
• Off-chip solution
  – need for external ATE
• Combined solution
  – mostly on-chip, ATE needed for control
• On-chip solution
  – BIST
What is BIST

• On circuit
  – Test pattern generation
  – Response verification
• Random pattern generation, very long tests
• Response compression
SoC BIST

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

Diagram showing Embedded Tester, Test Controller, Tester Memory, and various Cores with BIST mechanisms.
Built-In Self-Test

• **Motivations for BIST:**
  – Need for a cost-efficient testing (general motivation)
  – Doubts about the stuck-at fault model
  – Increasing difficulties with TPG (Test Pattern Generation)
  – Growing volume of test pattern data
  – Cost of ATE (Automatic Test Equipment)
  – Test application time
  – Gap between tester and UUT (Unit Under Test) speeds

• **Drawbacks of BIST:**
  – Additional pins and silicon area needed
  – Decreased reliability due to increased silicon area
  – Performance impact due to additional circuitry
  – Additional design time and cost
BIST in Maintenance and Repair

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

• To overcome the **disadvantages** of software tests for field test and diagnosis (nonBIST):
  – Low hardware fault coverage
  – Low diagnostic resolution
  – Slow to operate

• **Hardware BIST benefits:**
  – Lower system test effort
  – Improved system maintenance and repair
  – Improved component repair
  – Better diagnosis
  – Possibility to use the functionality of microprocessors
BIST Techniques

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

Universal test sets

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

Properties of exhaustive tests

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

2. **Shortcomings:**
   - long test length ($2^n$ patterns are needed, $n$ - is the number of inputs)
   - CMOS stuck-open fault problem
Exhaustive and Pseudo-Exhaustive Testing

Exhaustive combinational fault model:
- exhaustive test patterns
- pseudoexhaustive test patterns
  - exhaustive output line oriented test patterns
  - exhaustive module oriented test patterns
**BIST: Pseudoexhaustive Testing**

**Pseudo-exhaustive test sets:**

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

**Module function verification**

**Exhaustive test**
\[ 2^{16} = 65536 \]

**Pseudo-exhaustive sequential**
\[ 4 \times 16 = 64 \]

**Pseudo-exhaustive parallel**
\[ > 16 \]

---

Technical University Tallinn, ESTONIA
Testing ripple-carry adder

Output function verification (maximum parallelity)

Exhaustive test generation for n-bit adder:

*Good news:* Bit number n - arbitrary
*Bad news:* The method is correct only for ripple-carry adder

*Test length - always 8 (!)*

<table>
<thead>
<tr>
<th>a0</th>
<th>b0</th>
<th>a1</th>
<th>b1</th>
<th>a2</th>
<th>b2</th>
<th>c3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

0-bit testing 1-bit testing 2-bit testing 3-bit testing ... etc
Pseudo-Exhaustive Test for Multiplier

Selectable multiplicands

Multiplier array

Two examples

Multiplication with traditional “paper and pencil” method

Technical University Tallinn, ESTONIA
# Pseudo-Exhaustive Test for Multiplier

Replication of columns with pseudo-exhaustive patterns for **Adder** and **Multiplier**.

This table is replicated and all replications are repeated for all shifted $b = (\ldots11\ldots)$.

<table>
<thead>
<tr>
<th>N</th>
<th>6-bit</th>
<th>5-bit</th>
<th>4-bit</th>
<th>3-bit</th>
<th>2-bit</th>
<th>1-bit</th>
<th>0-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$c_6a_7a_6$</td>
<td>$c_5a_6a_5$</td>
<td>$c_4a_5a_4$</td>
<td>$c_3a_4a_3$</td>
<td>$c_2a_3a_2$</td>
<td>$c_1a_2a_1$</td>
<td>$a_1a_0$</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 1 0</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>4</td>
<td>1 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>5</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>6</td>
<td>1 0 1</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1</td>
<td>1 0 1</td>
<td>1 0 1</td>
<td>1 0 1</td>
<td>1 0 1</td>
<td>1 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>9</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>10</td>
<td>0 1 0</td>
<td>1 0 1</td>
<td>1 0 1</td>
<td>1 0 1</td>
<td>1 0 1</td>
<td>1 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>11</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

**carry multiplier array**
Exhaustively Self-Testing Multiplier

BIST
Built-in Self-Test

Multiplier operands:
Shifted 11
00000011
00000110
11000000

Multiplicand operands: generated with FSM and replicated

Test length: (n-1) × 11
Pseudoexhaustive Test Optimization

Simple iterative algorithm for test pattern generation:

Output function verification

Partial parallelism

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

Segmented LFSR → Pseudo-Random Test → Combined PE-PR Test → Circuit with 4 cones

A set of Partial Pseudo-Exhaustive tests can be combined with
(1) Pseudorandom BIST or
(2) Stored Deterministic test set
Problems with Exhaustive Testing

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

**NOR gate test:**

**Stuck-off (open)**

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

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
<th>$y^d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$Y'$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Test sequence is needed: **00, 10**
Problems with Exhaustive Testing

Problem: Sequential fault class - Bridging Fault Sequentiality

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

\[ 2^4 = 16 \] input patterns
Instead of \[ 2^3 = 8 \]
**General Architecture of BIST**

- **BIST components:**
  - Test pattern generator (TPG)
  - Test response analyzer (TRA)

- TPG & TRA are usually implemented as linear feedback shift registers (LFSR)

- Two widespread schemes:
  - test-per-scan
  - test-per-clock
## Built-In Self-Test

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

### Test per Scan:

**Initial test set:**

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

**Test application:**

1100 1010 T 0101 T 1001 T

Number of clocks = \((4 \times 4) + 4 = 20\)
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

- **Number of clocks = 8 < 20**
Pattern Generation

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

• *Exhaustive testing* – Apply all possible $2^n$ patterns to a circuit with $n$ inputs

• *Pseudo-exhaustive testing* – Break circuit into small blocks (overlapping if needed) and test each exhaustively

• *Pseudo-random testing* – Algorithmic pattern generator that produces a subset of all possible tests with most of the properties of randomly-generated patterns

• *LFSR* – *Linear feedback shift register*, hardware that generates pseudo-random pattern sequence

• *BILBO* – *Built-in logic block observer*, extra hardware added to flip-flops so they can be reconfigured as an LFSR pattern generator or response compacter, a scan chain, or as flip-flops
Pattern Generation

Pseudorandom test generation by LFSR:

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

LFSR – Linear Feedback Shift Register:

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

LFSR – Linear Feedback Shift Register:

Why modular LFSR is useful for BIST?

Polynomial: \( P(x) = x^4 + x^3 + 1 \)
Problems with BIST: Hard to Test Faults

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

Problem: Low fault coverage

Patterns from LFSR:

Pseudorandom test window:

Dream solution: Find LFSR such that:
Pseudorandom Test Generation

Scan-based BIST

Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:

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

Why modular LFSR is useful for BIST?
BILBO BIST Architecture

Working modes:

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

Testing modes:

CC1: LFSR 1 - TPG
     LFSR 2 - SA

CC2: LFSR 2 - TPG
     LFSR 1 - SA
BILBO BIST Architecture

Working modes:
B1 B2
0 0 Normal mode
0 1 Reset
1 0 Test mode
1 1 Scan mode

Testing modes:
CC1, CC2 Tested in parallel:
LFSR 1
LFSR 2
TPG + SA
Reconfiguration of the LFSR

Unit Under Test

LFSR FEEDBACK

From $T_{i-1}$

$T_i$

&

Reset

Test

Scan

Signature analyzer mode

B1

B2

MUX

Normal

OR

4 working modes

4 working modes

$T_i$ & $T_{i+1}$ & $T_{i+2}$

Test

Signature analyzer mode

From $T_{i-1}$

To $T_{i+2}$

Technical University Tallinn, ESTONIA
Pseudorandom Test Generation - LFSR

Two approaches to LFSR simulation:

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

Matrix calculation:

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

Shift

Feedback

<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>1</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>
Theory of LFSR: Primitive Polynomials

Properties of Polynomials:

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

Polynomials of degree n=3 (examples):

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 \textit{reciprocal}: coefficients are 1011 and 1101.

Reducible polynomials (non-primitive):

\[ x^3 + 1 = (x + 1)(x^2 + x + 1) \]
\[ x^3 + x^2 + x + 1 = (x + 1)(x^2 + 1) \]
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?
  
  \( \text{Yes, it includes 3 terms} \)

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

\( x^4 + x^2 + 1 \) is non-primitive?

Divisibility check:

\[
\begin{align*}
\frac{x^4 + x^2 + 1}{x^{15} + x^{13} + x^{11}} & = x^{11} + x^9 + x^5 + x^3 \\
\frac{x^{15} + 1}{x^{13} + x^{11} + 1} & = x^{13} + x^{11} + 1 \\
\frac{x^{13} + 1}{x^9 + 1} & = x^9 + 1 \\
\frac{x^9 + x^7 + x^5}{x^7 + x^5 + 1} & = x^7 + x^5 + 1 \\
\frac{x^7 + x^5 + x^3}{x^3 + 1} & = x^3 + 1
\end{align*}
\]
Theory of LFSR: Examples

Simulation of the behaviour of LFSR by polynomial:

Primitive polynomials

\[ x^3 + x + 1 \]
## Theory of LFSR: Examples

### Comparison of test sequences generated:

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

Non-primitive polynomial
$x^4 + x^2 + 1$

$x \rightarrow x^2 \rightarrow x^3 \rightarrow x^4$

<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
$x^4 + x + 1$

$x \rightarrow x^2 \rightarrow x^3 \rightarrow x^4$

<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:

The code 0000 is missing
Pseudorandom Testing with LFSR

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

**Red patterns are test patterns**
- \( 0001 \)
- \( 1000 \)
- \( 1100 \)
- \( 1110 \)
- \( 1111 \)
- \( 0111 \)

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

**No match in the blue sequence**
Pseudorandom Testing with LFSR

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

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

Blue patterns are not testing the fault

For testing the fault \( x_{21} \equiv 1 \), the test patterns 0001, 0101 and 1001 can be used.
## Theory of LFSR: Primitive Polynomials

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>

<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>
Theory of LFSR: Primitive Polynomials

Examples of PP (exponents of terms):

<table>
<thead>
<tr>
<th>n</th>
<th>other</th>
<th>n</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

$x^3 + x + 1$

$x^{13} + x^4 + x^3 + x + 1$
BIST: Fault Coverage

Pseudorandom Test generation by LFSR:

Motivation for LFSR:
- low generation cost
- high initial efficiency

Drawback: 100% fault coverage is difficult to achieve
BIST: Fault Coverage

Pseudorandom Test generation by LFSR:

Motivation for 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

$$\frac{1}{2^{2^n} - 1} \sum_{i=1}^{m} 2^{2^n - i}, \quad 1 \leq m \leq 2^n$$
Fault Coverage: Functional View

Why we need $2^{64}$ patterns

Truth table for adder:

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Possible functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>00...000</td>
<td>01 0 1 0 1...101</td>
</tr>
<tr>
<td>00...001</td>
<td>00 1 1 0 0...011</td>
</tr>
<tr>
<td>00...010</td>
<td>00 1 0 0 1...101</td>
</tr>
<tr>
<td>11...11</td>
<td>00 0 0 0 0...111</td>
</tr>
</tbody>
</table>

First pattern

Test quality:

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

Fault coverage vs. Number of patterns:

- 0% to 100%

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Test Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>50% tested</td>
</tr>
<tr>
<td>Second</td>
<td>50%</td>
</tr>
<tr>
<td>3rd</td>
<td>75%</td>
</tr>
<tr>
<td>4th</td>
<td>87.5%</td>
</tr>
<tr>
<td>5th</td>
<td>93.75%</td>
</tr>
<tr>
<td>6th</td>
<td>100%</td>
</tr>
</tbody>
</table>
BIST: Fault Coverage

Explanation of the formula of fault coverage:

1) General case:

\[ \frac{1}{2^{2^n} - 1} \sum_{i=1}^{m} 2^{2^n - i} \leq m \leq 2^n \]

# tested functions

# all functions

2) Example:

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

\[ \frac{1}{2^{2^2} - 1} \sum_{i=1}^{1} 2^{2^2 - i} = \frac{2^3}{2^4 - 1} = \frac{8}{15} \]

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

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

Faulty functions covered by 1. pattern

Faulty functions covered by 2. pattern

50%

87,5%

75%

93,75%

0%
Deterministic test approach:

Testing of structural faults:

1. pattern
2. pattern
3. pattern
4. pattern

Faults covered by 1. pattern
Not tested faults

Combinational circuit under test

Fault coverage

Number of patterns

100%

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

Deterministic test approach:

Faults covered by 1. pattern

3. pattern

4. pattern

Not tested faults

Faulty functions covered by 1. pattern

Faulty functions covered by 2. pattern

Faulty functions covered by 3. pattern

Faulty functions covered by 4. pattern

0% 50% 75% 87.5% 93.75%

100%
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:

1 2^{n-1}

Hard to test faults

Dream solution: Find LFSR such that:

1 2^{n-1}

Hard to test faults
Deterministic Scan-Path Test

Test per Clock:

Combinational Circuit Under Test

Scan-Path Register

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

How to generate the shortest sequence by LFSR

• Test application:
  • 1 10 0 1 0 1 0 01 01 1001

• Number of clocks = 8 < 20
Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence

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

2) Creation of the shortest bit-stream:
   $10010 \ 10111$

3) Expected shortest LFSR sequence:
   $01111$ (4)

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

Generation of the polynomial and seed for the given test sequence

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

Expected shortest LFSR sequence:

<table>
<thead>
<tr>
<th>x_i</th>
<th>f_k</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111 (4)</td>
<td>1</td>
</tr>
<tr>
<td>10111</td>
<td>0</td>
</tr>
<tr>
<td>01011</td>
<td>0</td>
</tr>
<tr>
<td>10101 (3)</td>
<td>0</td>
</tr>
<tr>
<td>01010</td>
<td>0</td>
</tr>
<tr>
<td>00101</td>
<td>1</td>
</tr>
<tr>
<td>10010 (1)</td>
<td>1</td>
</tr>
</tbody>
</table>

We are looking for the values of $x_i$
## Deterministic Synthesis of LFSR

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

System of linear equations:

<table>
<thead>
<tr>
<th></th>
<th>01111</th>
<th>x₁</th>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10111</td>
<td>x₂</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>01011</td>
<td>x₃</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>10101</td>
<td>x₄</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>01010</td>
<td>x₅</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>00101</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solving the equation by Gaussian elimination with swapping of rows

<table>
<thead>
<tr>
<th>Rows:</th>
<th>Results:</th>
<th>f_k</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₁ 1,2,4,6</td>
<td>01000</td>
<td>0</td>
</tr>
<tr>
<td>x₂ 4,6</td>
<td>10000</td>
<td>f₂</td>
</tr>
<tr>
<td>x₃ 1,3</td>
<td>00100</td>
<td>1</td>
</tr>
<tr>
<td>x₄ 2,4</td>
<td>00010</td>
<td>f₃</td>
</tr>
<tr>
<td>x₅ 1,3,6</td>
<td>00001</td>
<td></td>
</tr>
</tbody>
</table>

\[ aₖx₁ ⊕ bₖx₂ ⊕ cₖx₃ ⊕ dₖx₄ ⊕ eₖx₅ = fₖ, \ k = 1,2,..,6 \]

Examples:

1. (4) 10101 0
2. (6) 00101 1
3. (1) 01111 1
4. (3) 01011 1

4) Solution:

\[ x₁ x₂ x₃ x₄ x₅ = 1 0 0 0 0 1 \]
Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence

Solving the equation by Gaussian elimination with swapping of rows

\[ a_k x_1 \oplus b_k x_2 \oplus c_k x_3 \oplus d_k x_4 \oplus e_k x_5 = f_k, \quad k = 1, 2, \ldots, 6 \]

\[ \begin{align*}
\wedge 01000 &= 0 \quad \Rightarrow \quad x_2 = 0 \\
\wedge 10000 &= 1 \quad \Rightarrow \quad x_1 = 1 \\
\wedge 00100 &= 0 \quad \Rightarrow \quad x_3 = 0 \\
\wedge 00010 &= 0 \quad \Rightarrow \quad x_4 = 0 \\
\wedge 00001 &= 1 \quad \Rightarrow \quad x_5 = 1
\end{align*} \]

4) Solution: \[ x_1 x_2 x_3 x_4 x_5 = 100001 \]
Deterministic Synthesis of LFSR

Embedding deterministic test patterns into LFSR sequence:

4) Solution: \( x_1 x_2 x_3 x_4 x_5 \)
   \[ 1 \ 0 \ 0 \ 0 \ 1 \]

5) Polynomial: \( x^5 + x + 1 \)  Seed: 01111

Given deterministic test sequence:

- (1) 10000
- (2) x1010
- (3) 10101
- (4) 01111

LFSR sequence:

- (1) 01111 (4)
- (2) 10111
- (3) 01011
- (4) 10101 (3)
- (5) 01010 (2)
- (6) 00101
- (7) 10010 (1)
Which Test Patterns to Select for as HTF?

Frequencies of fault detection

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

Easily detectable faults

Hard-to-test faults (HTF)

Different faults
Other Problems with Pseudorandom Test

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

**Problem:** low fault coverage

If $\text{Reset} = 1$ signal has probability 0.5 then counter will not work and 1 for AND gate may never be produced.
Sequential BIST

A DFT technique of BIST for sequential circuits is proposed

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

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

Control Part

Data Part

Digital System
- FSM
- Datapath
- primary inputs
- masked status bits
- primary outputs
- observation points
- control signals
- status signals
- test/normal mode (TM)
- clock
- reset

Technical University Tallinn, ESTONIA
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 are chosen randomly

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

Calculation of signal probabilities:

For $PI_1$:
\[ P = 0.15 \]

For $PI_2$ and $PI_3$:
\[ P = 0.6 \]

For $PI_4$ - $PI_6$:
\[ P = 0.4 \]

Probability of detecting the fault \( \equiv 1 \) 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: Weighted pseudorandom test

Hardware implementation of weight generator

LFSR

Weight select

Desired weighted value

Scan-IN

1/2

1/4

1/8

1/16
BIST: Weighted pseudorandom test

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

**Solution:** weighted pseudorandom testing

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

NCV – non-controlling value

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

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

NDI - relative measure of the number of faults to be detected through the gate.
BIST: Weighted pseudorandom test

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 (1) to the controlling value (0) for each gate input
BIST: Weighted pseudorandom test

Example:

R_1 = \frac{\text{NDI}_G}{\text{NDI}_I} = \frac{6}{1} = 6
R_2 = \frac{\text{NDI}_G}{\text{NDI}_I} = \frac{6}{2} = 3
R_3 = \frac{\text{NDI}_G}{\text{NDI}_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 are calculated by backtracking.

<table>
<thead>
<tr>
<th>Function</th>
<th>W0_{IN}</th>
<th>W1_{IN}</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>W0_G</td>
<td>R_I \times W1_G</td>
</tr>
<tr>
<td>NAND</td>
<td>W1_G</td>
<td>R_I \times W0_G</td>
</tr>
<tr>
<td>OR</td>
<td>R_I \times W0_G</td>
<td>W1_G</td>
</tr>
<tr>
<td>NOR</td>
<td>R_I \times W1_G</td>
<td>W0_G</td>
</tr>
</tbody>
</table>

Calculation of W0, W1 for inputs:

- R_2 = 3
  - W0_2 = 1
  - W1_2 = 3
- R_1 = 6
  - W0_1 = 1
  - W1_1 = 6
- R_3 = 2
  - W0_3 = 1
  - W1_3 = 2
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 inputs

<table>
<thead>
<tr>
<th>Function</th>
<th>$W_{0I}$</th>
<th>$W_{1I}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>$R_I \times W_{0G}$</td>
<td>$W_{1G}$</td>
</tr>
<tr>
<td>NOR</td>
<td>$R_I \times W_{1G}$</td>
<td>$W_{0G}$</td>
</tr>
</tbody>
</table>
BIST: Weighted pseudorandom test

Calculation of signal probabilities:

- $P_{I_1}$:
  - $W_0 = 6$  $W_1 = 1$  $P_1 = 1/7 = 0.15$

- $P_{I_2}$ and $P_{I_3}$:
  - $W_0 = 2$  $W_1 = 3$  $P_1 = 3/5 = 0.6$

- $P_{I_4}$ - $P_{I_6}$:
  - $W_0 = 3$  $W_1 = 2$  $P_1 = 2/5 = 0.4$
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 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:

$P = 0.85 \times (0.6 \times 0.4 + 0.4 \times 0.6 + 0.6^2) \times 0.6^3 = 0.85 \times 0.84 \times 0.22 = 0.16$
The Main BIST Problems

- **On circuit**
  - Test pattern generation
  - Response verification
- **Random pattern generation,**
  - Very long tests
  - Hard-to-test faults
- **Response compression**
  - Aliasing of results
Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:

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

Signature analyzer:

Response in compacted by LFSR

The content of LFSR after test is called *signature*

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

Parallel Signature Analyzer:

Single Input Signature Analyser

UUT

\[ x^4 \rightarrow x^3 \rightarrow x^2 \rightarrow x \rightarrow 1 \]

Multiplication Input Signature Analyser (MISR)

UUT

\[ x^4 \rightarrow x^3 \rightarrow x^2 \rightarrow x \rightarrow 1 \]
Special Cases of 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

\[ P(R) = \sum_{i=1}^{m} \overline{r_i} \]
Special Cases of Response Compression

4. Transition counting

a) Transition 0→1

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

b) Transition 1→0

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

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

Coding theory treats binary strings as polynomials:

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

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

**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.
Theory of LFSR

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

$$n = 2m + r, \; r \in \{0,1\} \quad \text{or} \quad r = n \pmod{2}$$

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

Examples (addition, multiplication):

$$\begin{align*}
x^4 + x^3 & \quad + \quad x & \quad + & \quad 1 \\
+ x^4 & \quad + \quad x^2 & \quad + & \quad x \\
\hline
\quad x^3 & \quad + \quad x^2 & \quad + & \quad 1
\end{align*}$$

$$\begin{align*}
x^4 + x^3 & \quad + \quad x & \quad + & \quad 1 \\
\times x & \quad + & \quad 1 \\
\hline
\quad x^5 & \quad + \quad x^4 & \quad + \quad x^2 & \quad + \quad x \\
\quad x^4 & \quad + \quad x^3 & \quad + & \quad x & \quad + & \quad 1 \\
\hline
\quad x^5 & \quad + \quad x^3 & \quad + \quad x^2 & \quad + & \quad 1
\end{align*}$$
Theory of LFSR

Characteristic Polynomials:

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

Divisor \[ x^2 + 1 \]

\[ \begin{array}{ccc}
   & x^2 + x + 1 & \text{Quotient} \\
\hline
x^4 + x^3 & +1 & \text{Dividend} \\
  \hline
x^4 & + x^2 & \\
  \hline
x^3 & + x^2 & +1 \\
  \hline
x^3 & + x & \\
  \hline
x^2 & + x + 1 & \\
  \hline
x^2 & + 1 & \\
  \hline
x & \text{Remainder} \\
\end{array} \]
BIST: Signature Analysis

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

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

**Example:**

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

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

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

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

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

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

$$\begin{array}{c}
\text{P(x)} \\
00100110110110111001101000110101100101011
\end{array}$$

$$\text{Signature}$$
BIST: Hardware for Signature Analysis

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

\( G(x) \) is shifted into LFSR with divisor polynomial defined by feedback connections.

Response:

\( P(x) \)

\( x^5 \) is divided by \( G(x) \) and the remainder is used as the signature.

Signature:

\( 001101 \)

Division process can be mechanized using LFSR with the divisor polynomial defined by the feedback connections.

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

Aliasing:

\[ k = 2^L \]

All possible responses

Faulty response

Correct response

\[ L \] - test length

\[ N \] - number of stages in Signature Analyzer

\[ k = 2^N \]

All possible signatures

\[ N << L \]
BIST: Signature Analysis

Aliasing:

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.

\[
k = 2^L \quad \text{- number of different possible responses}
\]

\[
2^{L-N} - 1 \quad \text{---- Aliasing is possible}
\]

Probability of aliasing:

\[
P = \frac{2^{L-N} - 1}{2^L - 1} \quad \text{or} \quad P = \frac{1}{2^N} \quad \text{for} \quad L \gg 1
\]
BIST: Signature Analysis

Parallel Signature Analyzer:

Single Input Signature Analyser

Multiple Input Signature Analyser (MISR)
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 - SA
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
Two functionalities of LFSR:

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

Response string for
Signature Analysis
Pseudorandom Test Generation

LFSR – Linear Feedback Shift Register:

Why modular LFSR is useful for BIST?

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

Instead of BILBO we have now CSTP architecture
BIST Architectures

Test per Clock:

Disjoint TPG and SA:
BILBO

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

Joint TPG and SA:
CSTP - Circular Self-Test Path:

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

Circuit Under Test

 FF  FF  FF
BIST: Circular Self-Test Path

Diagram showing the circular self-test path with nodes labeled CSTP, CC, and R connected in a loop.

Technical University Tallinn, ESTONIA
BIST Embedding Example

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

Diagram of BIST Architectures with components labeled as:
- **CUT:** Core Under Test
- **Scan chain:** Scan chain path through IC
- **MISR:** Multi-Input Signature Register
- **TPG:** Test Pattern Generator
- **SA:** Scan Angle
- **BS:** Boundary Scan
- **Test Controller**
- **IC:** Integrated Circuit
- **SI:** Input Signal
- **SO:** Output Signal
- **Error**
Scan-Based BIST Architecture

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

PS – Phase shifter
Scan-Forest
Scan-Trees
Scan-Segments (SC)
Weighted scan-enables for SS
Compactor - EXORs

Copyright: D. Xiang 2003
Problems with BIST

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

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

Possible solutions
- Weighted pseudorandom test
- Combining pseudorandom test with deterministic data
  - Multiple seed
  - Bit flipping
- Hybrid BIST
Problems with BIST: Hard to Test Faults

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

Problem: **Low fault coverage**

Patterns from LFSR:

Dream solution: Find LFSR such that:
Deterministic Synthesis of LFSR

Generation of the polynomial and seed for the given test sequence

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

2) Creation of the shortest bit-stream:
   10010 1 01111

3) Expected shortest LFSR sequence:
   01111 (4)

This deterministic test set is generated by ATPG. However, only patterns which detect the hard-to-test faults can be chosen.
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: \( C_{\text{TOTAL}} = \alpha k + \beta t(k) \)

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

**SLOW analysis**
Cost of stored test \( C_{\text{MEM}} \)
Number of pseudorandom test patterns applied, \( k \)

Pseudorandom Test  Det. Test

How to convert #faults to #tests

<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>

Figure 2: Cost calculation for hybrid BIST
Deterministic Test Length Estimation

Fault coverage

F

100%

FD_k(i)

FPE_k(i)

F*

Brake point search

j

i*

|TD^F_k|

Number of patterns

|TD^E_k(i)|

Deterministic test length estimation

Fast estimation for the length of deterministic test:

For each PT length i* we determine
- PT fault coverage F*, and
- the imaginable part of DT FD_k(i) to be needed for the same fault coverage

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

Second idea for estimation: estimating number of patterns
Deterministic Test Length Estimation

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

- **FAST estimation**
  - Number of remaining faults after applying \( k \) pseudorandom test patterns \( \beta r_{\text{NOT}}(k) \)
  - Cost of pseudorandom test patterns \( C_{\text{GEN}} \)
  - Cost of stored test \( C_{\text{MEM}} \)
  - PR test length \( k \)
  - \( \alpha k \)
  - \( \beta t(k) \)

- **SLOW analysis**
  - Number of pseudorandom test patterns applied, \( k \)
  - Brake point \( \min C_{\text{TOTAL}} \)

**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>

**Figure 2:** Cost calculation for hybrid BIST

**How to convert \# faults to \# tests**
Calculation of the Deterministic Test Cost

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

**ATPG based approach**
For each breakpoint of P-sequence, ATPG is used

**Fault table based approach**
A deterministic test set with fault table is calculated
For each breakpoint of P-sequence, the fault table is updated for not yet detected faults

**FAST estimation**
Only fault coverage is calculated

---

### ATPG based:

1. **ATPG**
2. **Detected Faults**
3. **All PR patterns?**
   - **No** → **Next PR pattern**
   - **Yes** → **End**

### Fault table based:

1. **ATPG**
2. **Fault table update**
3. **All PR patterns?**
   - **No** → **Next PR pattern**
   - **Yes** → **End**

---

Technical University Tallinn, ESTONIA
Calculation of the Deterministic Test Cost

**ATPG based approach**
For each breakpoint of P-sequence, ATPG is used

ATPG based:

- **ATPG**
- Detected Faults
- All PR patterns?
  - No: Next PR pattern
  - Yes: End

**Faults detected by pseudo-random patterns**

- Task for fault simulator
- Brake point
- Task for ATPG
- T_p

**Faults to be detected by deterministic patterns**

- New detected faults

---

Technical University Tallinn, ESTONIA
Calculation of the Deterministic Test Cost

Fault table based approach
A deterministic test set with fault table is calculated
For each breakpoint of P-sequence, the fault table is updated and remaining det. patterns are determined

Fault table based:

A deterministic test set with fault table is calculated
For each breakpoint of P-sequence, the fault table is updated and remaining det. patterns are determined

Fault table based approach:
A deterministic test set with fault table is calculated
For each breakpoint of P-sequence, the fault table is updated and remaining det. patterns are determined
Calculation of the Deterministic Test Cost

Fault table based approach
A deterministic test set with fault table is calculated
For each breakpoint of P-sequence, the fault table is updated and remaining det. patterns are determined

Fault table based:

1. ATPG
2. Fault table update
3. All PR patterns?
   - No: Next PR pattern
   - Yes: End

Pseudo-random patterns

Fault table coverage

Deterministic patterns

D1
D2
D3
Experimental Data: HybBIST Optimization

Finding optimal brakepoint in the pseudorandom sequence:

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

Optimized hybrid test process:

<table>
<thead>
<tr>
<th>Circuit</th>
<th>$L_{MAX}$</th>
<th>$L_{OPT}$</th>
<th>$S_{MAX}$</th>
<th>$S_{OPT}$</th>
<th>$B_k$</th>
<th>$C_{TOTAL}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C432</td>
<td>780</td>
<td>91</td>
<td>80</td>
<td>21</td>
<td>4</td>
<td>175</td>
</tr>
<tr>
<td>C499</td>
<td>2036</td>
<td>78</td>
<td>132</td>
<td>60</td>
<td>6</td>
<td>438</td>
</tr>
<tr>
<td>C880</td>
<td>5589</td>
<td>121</td>
<td>77</td>
<td>48</td>
<td>8</td>
<td>505</td>
</tr>
<tr>
<td>C1355</td>
<td>1522</td>
<td>121</td>
<td>126</td>
<td>52</td>
<td>6</td>
<td>433</td>
</tr>
<tr>
<td>C1908</td>
<td>5803</td>
<td>105</td>
<td>143</td>
<td>123</td>
<td>5</td>
<td>720</td>
</tr>
<tr>
<td>C2670</td>
<td>6581</td>
<td>444</td>
<td>155</td>
<td>77</td>
<td>30</td>
<td>2754</td>
</tr>
<tr>
<td>C3540</td>
<td>8734</td>
<td>297</td>
<td>211</td>
<td>110</td>
<td>7</td>
<td>1067</td>
</tr>
<tr>
<td>C5315</td>
<td>2318</td>
<td>711</td>
<td>171</td>
<td>12</td>
<td>23</td>
<td>987</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>3694</td>
</tr>
</tbody>
</table>

Technical University Tallinn, ESTONIA
Hybrid BIST with Reseeding

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

Problem: low fault coverage $\rightarrow$ long PR test

Solution: many seeds:

Pseudorandom test:
Hybrid BIST with Reseeding

Using many seeds:

Pseudorandom test (with different polynomials):

Creation of the shortest bit-stream:
1) Given test sequence:
   (1) $100 \times 0$
   (2) $x1010$
   (3) $10101$
   (4) $01111$

   $10010 \ 10111$

Expected shortest LFSR sequence:
   Seed $\rightarrow$ Shift

   $01111 \ (4)$

States of the LFSR:
   $1 \ 0111$
   $0 \ 1011$
   $1 \ 0101 \ (3)$
   $0 \ 0100 \ (2)$
   $0 \ 0101$
   $1 \ 0010 \ (1)$

Problems:
Which polynomials and seeds should be used for the blocks?
• **ROM** contains deterministic data for BIST control to target **hard-to-test-faults**
• Each pattern $P_k$ in ROM serves as an initial state of the LFSR for test pattern generation (TPG) - **seeds**
• Counter 1 counts the number of pseudorandom patterns generated starting from $P_k$ - **width of the windows**
• After finishing the cycle for Counter 2 is incremented for reading the next pattern $P_{k+1}$ – for **starting the new window**
Store-and-Generate vs. Hybrid BIST

Deterministic test pattern

Pseudorandom patterns

Store and generate method (Reseeding)

Deterministic test pattern

Hybrid BIST method

ROM
LFSR
Seed

ROM

TPG

UUT

Counter 1

Counter 2

ADR

RD

Window

CL

Seeds
HBIST Optimization Problem

Using many seeds:

Pseudorandom test:

1

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
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 included into sequence and updated

The procedure ends when 100% fault coverage is achieved
Two possibilities for reseeding:

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

![Diagram showing test length L vs memory cost M for different block sizes](image)
Functional Self-Test

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

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

**Functional BIST quality analysis for**

Samples from $N=120$ cycles

- Register block
- Control
- ALU
- Signature analyser

**Fault simulator**

**Fault coverage**

Data compression:

$N \times SB / DB = 197$

K pairs of operands B1, B2

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

DB=64

SB=105

K

Data

$K \times N$
Example: Functional BIST Quality for Divider

Fault coverage of FBIST compared to Functional test

<table>
<thead>
<tr>
<th>Data</th>
<th>Functional testing</th>
<th>Functional BIST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B1</td>
<td>B2</td>
</tr>
<tr>
<td>4/2</td>
<td>13.21</td>
<td>15.09</td>
</tr>
<tr>
<td>7/2</td>
<td>21.23</td>
<td>16.98</td>
</tr>
<tr>
<td>6/3</td>
<td>19.34</td>
<td>31.6</td>
</tr>
<tr>
<td>8/2</td>
<td>25.47</td>
<td>10.38</td>
</tr>
<tr>
<td>9/4</td>
<td>8.96</td>
<td>5.66</td>
</tr>
<tr>
<td>9/3</td>
<td>32.55</td>
<td>26.89</td>
</tr>
<tr>
<td>12/6</td>
<td>13.44</td>
<td>8.02</td>
</tr>
<tr>
<td>14/2</td>
<td>18.16</td>
<td>25.00</td>
</tr>
<tr>
<td>15/3</td>
<td>29.48</td>
<td><strong>31.13</strong></td>
</tr>
<tr>
<td>2/4</td>
<td>7.8</td>
<td>7.55</td>
</tr>
<tr>
<td>Aver.</td>
<td>18.96</td>
<td>17.83</td>
</tr>
<tr>
<td>Gain</td>
<td>1.0</td>
<td>1.0</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
Hybrid test set contains pseudorandom and deterministic vectors

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

*Optimization problem:* Where should be this breakpoint?

| Pseudorandom Test | Determ. Test |
Hybrid Functional BIST for Divider

Hybrid Functional BIST implementation

Where should be this breakpoint?
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 \( C_{Total} \)
## Hybrid Functional BIST Quality

### Hyb FBIST with multiple seeds (data operands)

<table>
<thead>
<tr>
<th>$k$</th>
<th>$N_j$</th>
<th>$N$</th>
<th>$FC%$</th>
<th>Total cost</th>
<th>Determin. test part</th>
<th>Total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>58</td>
<td>6148</td>
</tr>
<tr>
<td>1</td>
<td>108</td>
<td>108</td>
<td>66,8</td>
<td>140</td>
<td>24</td>
<td>2544</td>
</tr>
<tr>
<td>2</td>
<td>105</td>
<td>213</td>
<td>76,7</td>
<td>277</td>
<td>18</td>
<td>2185</td>
</tr>
<tr>
<td>3</td>
<td>113</td>
<td>326</td>
<td>83,3</td>
<td>518</td>
<td>17</td>
<td>2320</td>
</tr>
<tr>
<td>4</td>
<td>108</td>
<td>434</td>
<td>85,5</td>
<td>690</td>
<td>16</td>
<td>2386</td>
</tr>
<tr>
<td>5</td>
<td>110</td>
<td>544</td>
<td>88,4</td>
<td>864</td>
<td>15</td>
<td>2454</td>
</tr>
</tbody>
</table>

$k$ – number of operands used in the FBIST

The fault coverage increases if $k$ increases
Functional Self-Test with DFT

Example: N-bit multiplier

Register block

ALU

EXOR

Signature analyser

N cycles

Improving controllability

Improving observability

Data

K

T

MUX

F
Hybrid BIST for Multiple Cores

Embedded tester for testing multiple cores

- Deterministic test patterns: C880, C1355, C1908, C2670, C3540
- Pseudorandom or functional test patterns generated on-line
Hybrid BIST for Multiple Cores

Deterministic test (DT)

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>

How to pack knapsack?
How to compress the test sequence?

Pseudorandom test (PT)
Total Test Cost Estimation

Using total cost solution we find the PT length:

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


c1908

c880

c2670

c1355

c3540

0 100 200 300 400 500

Using total cost solution we find the PT length:

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


c432

c6288

c880

c980

c1355

c1355

c499

0 50 100 150 200

c499

c1355

c1355

c432

Deterministic

Pseudorandom

Total Test

Using total cost solution we find the PT length:

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


c432

c6288

c880

c980

c1355

c1355

c499

0 50 100 150 200

c499

c1355

c1355

c432

Deterministic

Pseudorandom

Total Test

Using total cost solution we find the PT length:

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


c432

c6288

c880

c980

c1355

c1355

c499

0 50 100 150 200

c499

c1355

c1355

c432

Deterministic

Pseudorandom

Total Test

Using total cost solution we find the PT length:

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


c432

c6288

c880

c980

c1355

c1355

c499

0 50 100 150 200

c499

c1355

c1355

c432

Deterministic

Pseudorandom

Total Test

Using total cost solution we find the PT length:

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


c432

c6288

c880

c980

c1355

c1355

c499

0 50 100 150 200

c499

c1355

c1355

c432

Deterministic

Pseudorandom

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

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

**SLOW analysis**
- Number of pseudorandom test patterns applied, \( k \)
- Cost of stored test \( C_{\text{MEM}} \)

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

**Two problems:**
1) Calculation of DT \( \beta t(k) \) cost is difficult
2) We have to optimize \( n (!) \) processes
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 $TD_k^E(i)$ 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:

- Memory usage: 5357 bits
- Core costs
- Estimated cost
- Real cost
- Total test length

Solution

Core name: c499, c880, c1355, c1908, c5315, c6288, c432

<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>

Total test length calculation of core costs:
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
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.
• 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:

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:

Scan-In → CUT 1 → ... → MISR → Scan-Out

Scan-In → CUT n → ... → MISR → Scan-Out

- Common MISR
- Individual and multiple MISRs
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.
Embedded Built-in Self-Diagnosis (BISD)

• Introduction to Fault Diagnosis
  – Combinational diagnosis (effect-cause approach)
  – Sequential (adaptive) diagnosis (cause-effect approach)

• General conception of embedded BISD

• Diagnostic resolution
  – Intersection based on test subsequences
  – Intersection based on using signature analyzers

• Fault model free diagnosis

• Fault evidence based diagnosis
# Why Fault Masking is Important Issue?

<table>
<thead>
<tr>
<th>Diagnosis method</th>
<th>Fault table</th>
<th>Test result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Devil’s advocate approach</td>
<td>Tested faults</td>
<td>Passed</td>
</tr>
<tr>
<td></td>
<td>Tested faults</td>
<td>Failed</td>
</tr>
<tr>
<td></td>
<td>Tested faults</td>
<td>Failed</td>
</tr>
<tr>
<td>Single fault assumption</td>
<td>Fault candidates</td>
<td>Diagnosis</td>
</tr>
<tr>
<td>Multiple faults allowed</td>
<td>?</td>
<td>Fault candidates</td>
</tr>
<tr>
<td>Angel’s advocate</td>
<td>Proved OK</td>
<td>Fault candidates</td>
</tr>
</tbody>
</table>
Fault Diagnosis

Fault table

<table>
<thead>
<tr>
<th></th>
<th>F_1</th>
<th>F_2</th>
<th>F_3</th>
<th>F_4</th>
<th>F_5</th>
<th>F_6</th>
<th>F_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T_2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T_3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T_4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T_5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T_6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Test experiment

<table>
<thead>
<tr>
<th></th>
<th>E_1</th>
<th>E_2</th>
<th>E_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T_2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T_3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T_4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T_5</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T_6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

Fault simulation

Test generation

Fault F_5 located

Fault diagnosis

Testing

Fault modeling
Sequential Fault Diagnosis

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

<table>
<thead>
<tr>
<th></th>
<th>F₁</th>
<th>F₂</th>
<th>F₃</th>
<th>F₄</th>
<th>F₅</th>
<th>F₆</th>
<th>F₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T₂</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T₃</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T₄</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T₅</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T₆</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Diagnostic tree

Two faults F₁,F₄ remain indistinguishable
Not all test patterns used in the fault table are needed
Different faults need for identifying test sequences with different lengths
The shortest test contains two patterns, the longest four patterns
Embedded BIST Based Fault Diagnosis

BISD scheme:

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

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
- Diagnosis procedure:
  1. test
  2. test
  3. test

Faulty signature
Correct signature
Pseudorandom test sequence
**Introduction to Information Theory**

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

\[
H = - \sum p_i \log_2 (p_i)
\]

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

\[
H_X = - p \log_2 p - (1-p) \log_2 (1-p)
\]

\[
I = - p \log_2 p - (1-p) \log_2 (1-p)
\]

\( p \) – probability of detecting a fault
Build-In Fault Diagnosis

Measuring of information we get from the test:

\[ I = -p \log_2 p - (1-p) \log_2 (1-p) \]

\( p \) – probability of detecting a fault

Pseudorandom test fault simulation (detected faults)

<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 test patterns

Average number of test sessions: 3,3
Average number of clocks: 8,67
Built-In Fault Diagnosis

### Pseudorandom test fault simulation (detected faults)

<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
(based on reasoning of spacial information):

Test pattern generator

Fault

CUD

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>
Built-In Fault Diagnosis

BIST with multiple signature analyzers

Optimization in time dimension

Optimization in space dimension

Intersection using tests

Intersection using SA-s

Faulty signature

Correct signature

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

Diagram:
- Defect
- SAF
- Multiple fault
- Single faulty signal
Fault-Model Free Fault Diagnosis

Combined cause-effect and effect-cause diagnosis

1) Cause-Effect Fault Diagnosis
Suspected faulty area is located based on the fault table (dictionary)

2) Effect-Cause Fault Diagnosis
Faulty block is located in the suspected faulty area

3) Fault Reasoning
Failing test patterns are mapped into the suspected defect or into a set of suspected defects in the faulty block
Practical Use of Boolean Differences

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

Correct function: \[ y = x_1 x_2 x_3 \lor x_4 x_5 \]

Faulty function: \[ y^d = (x_1 \lor x_4)(x_2 x_3 \lor x_5) \]

Defect variable: \[ d = \begin{cases} 0 & \text{defect } d \text{ is missing} \\ 1 & \text{defect } d \text{ is present} \end{cases} \]

Generic function with defect:
\[ y^* = (y \land \bar{d}) \lor (y^d \land d) \]

Mapping the physical defect onto the logic level by solving the equation:
\[ \frac{\partial y^*}{\partial d} = 1 \]
### Fault Table: Mapping Defects to Faults

<table>
<thead>
<tr>
<th>$i$</th>
<th>Fault $d_i$</th>
<th>Erroneous function $f^{di}$</th>
<th>$p_i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B/C</td>
<td>not((B<em>C)</em>(A+D))</td>
<td>0.010307065</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>B/D</td>
<td>not((B<em>D)</em>(A+C))</td>
<td>0.000858922</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>B/N9</td>
<td>B*(not(A))</td>
<td>0.043375564</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B/Q</td>
<td>B*(not(C*D))</td>
<td>0.007515568</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>B/VDD</td>
<td>not(A+(C*D))</td>
<td>0.001717844</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>B/VSS</td>
<td>not(C*D)</td>
<td>0.035645265</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>A/C</td>
<td>not((A<em>C)</em>(B+D))</td>
<td>0.098990767</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>A/D</td>
<td>not((A<em>D)</em>(B+C))</td>
<td>0.013098561</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>A/N9</td>
<td>A*(not(B))</td>
<td>0.038651492</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>A/Q</td>
<td>A*(not(C*D))</td>
<td>0.025982392</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>A/VDD</td>
<td>not(B+(C*D))</td>
<td>0.000214731</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>C/N9</td>
<td>not(A+B+D)+(C*(not((A*B)+D)))</td>
<td>0.020399399</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>C/Q</td>
<td>C*(not((A*B)))</td>
<td>0.033927421</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>C/VSS</td>
<td>not(A*B)</td>
<td>0.005153532</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>D/N9</td>
<td>not((A<em>B)+(D</em>(not((A*B)+C)))</td>
<td>0.007730298</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>D/Q</td>
<td>D*(not(A*B))</td>
<td>0.149452437</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>N9/Q</td>
<td>not((A<em>B)+(B</em>C<em>D)+(A</em>C*D))</td>
<td>0.148654713</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>N9/VDD</td>
<td>not((C<em>D)+(A</em>B<em>D)+(A</em>B*C))</td>
<td>0.253382006</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>Q/VDD</td>
<td>SA1 at Q</td>
<td>0.014386944</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>Q/VSS</td>
<td>SA0 at Q</td>
<td>0.095555078</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Generalization: Functional Fault Model

Conditional Stuck-at-Fault model
Constrained SAF

Component $F(x_1, x_2, \ldots, x_n)$
Defect

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

Fault model:
$(dy, W^d)$
$(dy, \{W_k^d\})$

Constraints calculation:
$y^* = F^*(x_1, x_2, \ldots, x_n, d) = dF \lor dF^d$

Fault-free
Faulty

$d = 1$, if the defect is present
Diagnosis of Fault Model Free Defects

Fault evidence:
for test pattern \( t \)
\[ e(f,t) = (\Delta \tau_t, \Delta \sigma_t, \Delta l_t, \Delta \gamma_t) \]
\[ \Delta \gamma_t = \min (\Delta \sigma_t, \Delta l_t) \]
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

Δlₜ

Simulation

Faulty machine FM(t)

f

Fault

Different classical fault cases

<table>
<thead>
<tr>
<th>Classic model</th>
<th>lₜ</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
Diagnosis of Fault Model Free Defects

Different classical fault cases

<table>
<thead>
<tr>
<th>Classic model</th>
<th>$l_t$</th>
<th>$\tau_t$</th>
<th>$\gamma_t$</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
Diagnosis of Fault Model Free Defects

Different classical fault cases

<table>
<thead>
<tr>
<th>Classic model</th>
<th>$I_t$</th>
<th>$\tau_t$</th>
<th>$\gamma_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single SAF</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Multiple SAF (defects)</strong></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
Diagnosis of Fault Model Free Defects

Real test experiment

Circuit Under Diagnosis

Condition

Defect

Test pattern $t$

Simulation

Faulty machine $FM(t)$

$\Delta \sigma_t$

$\Delta l_t$

Fault

Different classical fault cases

<table>
<thead>
<tr>
<th>Classic model</th>
<th>$l_t$</th>
<th>$\tau_t$</th>
<th>$\gamma_t$</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
Fault Diagnosis Without Fault Models

Reverse defect mapping

Defective area

Error (defective area) diagnosis

Error detection
Diagnosis of Fault Model Free Defects

Real test experiment

Circuit Under Diagnosis

Faulty machine

FM(τ)

Fault

Simulation

Test pattern t

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 Tolerance: Error Detecting Codes

Examples:

Decimal digits:

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

Parity check:

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

Eligible
Not
Error Detecting/Correcting Codes

Hamming distance between codes:

- Eligible codes
- Not eligible codes

Minimal number of bits how two codes differ from each other

Parity check:

\[ d = 2 \]

Eligible codes

- 000
- 010
- 100
- 110
- 101
- 111
- 001
- 011

Parity bit

- 00
- 01
- 10
- 11

Eligible

- 00
- 01
- 10
- 11

Not eligible

- 01
- 10
- 11

eligible
Error Detecting/Correcting Codes

Error detecting codes:

- Eligible codes
- Error detection: direction unknown
- Detection not possible
- Not eligible codes

Error correcting codes:

- Eligible codes
- Error correction is possible: direction is known
- Correction not possible
Fault Tolerance: Error Correcting Codes

\[ d = 2e + 1 \quad - \quad 2e \text{ - error detection} \\
\quad e \text{ - error correction} \]

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

Check bits

Information bits

Error free

For addressing of the erroneous bit
Fault Tolerance: One Error Correcting Code

Location of erroneous bit:

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

Check bits have to be independently assigned

\[
\begin{align*}
P_1 &= b_1 \oplus b_3 \oplus b_5 \oplus b_7 = 0 \\
P_2 &= b_2 \oplus b_3 \oplus b_6 \oplus b_7 = 0 \\
P_3 &= b_4 \oplus b_5 \oplus b_6 \oplus b_7 = 0
\end{align*}
\]

Analogy with fault diagnosis by using fault table:

Initial code

Received code

Diagnosis

Check bits

Test

Technical University Tallinn, ESTONIA
Fault Tolerant Communication System

Initial code

Check-bits generator → Sender

Error correction code

Error indication

Receiver → Checker

Error correction (restoring)

Received correct code
Error Detection in Arithmetic Operations

Residue codes

- N – information code
- \( C = (N) \mod m \) - check code
- \( m \) – residue of the code
- \( p = \lceil \log_2 m \rceil \) – number of check bits

**Example**

<table>
<thead>
<tr>
<th>Information bits</th>
<th>Check bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_2 ), ( I_1 ), ( I_0 )</td>
<td>( c ), ( c_1 ), ( c_0 )</td>
</tr>
<tr>
<td>0, 0, 0, 0</td>
<td>0, 0, 0</td>
</tr>
<tr>
<td>0, 0, 1, 1</td>
<td>1, 0, 1</td>
</tr>
<tr>
<td>0, 1, 0, 2</td>
<td>2, 1, 0</td>
</tr>
<tr>
<td>0, 1, 1, 3</td>
<td>0, 0, 0</td>
</tr>
<tr>
<td>1, 0, 0, 4</td>
<td>1, 0, 1</td>
</tr>
<tr>
<td>1, 0, 1, 5</td>
<td>2, 1, 0</td>
</tr>
<tr>
<td>1, 1, 0, 6</td>
<td>0, 0, 0</td>
</tr>
<tr>
<td>1, 1, 1, 7</td>
<td>1, 0, 1</td>
</tr>
</tbody>
</table>
## Error Detection in Arithmetic Operations

### Addition:

<table>
<thead>
<tr>
<th>Information bits</th>
<th>Check bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>1 0</td>
<td>2.2</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1</td>
<td>4.1</td>
</tr>
<tr>
<td><strong>0 1 1 0</strong></td>
<td><strong>1 1</strong></td>
<td><strong>6.3</strong></td>
</tr>
</tbody>
</table>

(6) mod 3 = 0 (3) mod 3 = 0

### Multiplication:

<table>
<thead>
<tr>
<th>Information bits</th>
<th>Check bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>1 0</td>
<td>2.2</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1</td>
<td>4.1</td>
</tr>
<tr>
<td><strong>1 0 0 0</strong></td>
<td><strong>1 0</strong></td>
<td><strong>8.2</strong></td>
</tr>
</tbody>
</table>

(8) mod 3 = 2 (2) mod 3 = 2

<table>
<thead>
<tr>
<th>Information bits</th>
<th>Check bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>1 0</td>
<td>2.2</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1</td>
<td>4.1</td>
</tr>
<tr>
<td><strong>0 1 0 0</strong></td>
<td><strong>1 1</strong></td>
<td><strong>4.3</strong></td>
</tr>
</tbody>
</table>

(4) mod 3 = 1 (3) mod 3 = 0

Error!

<table>
<thead>
<tr>
<th>Information bits</th>
<th>Check bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>1 0</td>
<td>2.2</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1</td>
<td>4.1</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 0</td>
<td>9.2</td>
</tr>
</tbody>
</table>

(9) mod 3 = 0 (2) mod 3 = 2

Error!
Error Detection in Arithmetic Operations

- **A** + **B**
- **A** + **B**
- **C(A)** + **C(B)**
- **C(A) + B**
- **C(C(A) + C(B))**
- Error indicator

Diagram:

1. **Adder**
2. **Check bit generator**
3. **Adder mod m**
4. **Residue calculator**
5. **Comparator**
6. **Error indicator**
Fault Tolerance: One Error Correcting Code

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

Calculation of check sums:

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

Parity bits for $c = 3$:

- $P_1 = b_1 \oplus b_3 \oplus b_5 \oplus b_7 = 0$
- $P_2 = b_2 \oplus b_3 \oplus b_6 \oplus b_7 = 0$
- $P_3 = b_4 \oplus b_5 \oplus b_6 \oplus b_7 = 0$
Theory of LFSR

Characteristic Polynomials:

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

Multiplication of polynomials

\[
\begin{align*}
x^2 + x + 1 \\
x^2 + 1 \\
x^2 + x + 1 \\
\frac{x^4 + x^3 + x^2}{x^4 + x^3} + x + 1
\end{align*}
\]
Fault Tolerant Communication System

Initial code $P(x)$

Check-bits generator

Sender

$R(x)$

Error correction code

Receiver

$P'(x).R(X)$

Checker

$P'(x)/G(X)$

Error indication $\neq R(x)$

$P'(x)$

Error correction (restoring)

$P'(x)$

$P(x)$

Received correct code

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