Chapter 7

Logic Diagnosis
Outline

- Introduction
- Combinational Logic Diagnosis
- Scan Chain Diagnosis
- Logic BIST Diagnosis
- Conclusion
What would you do when chips fail?

- Is it due to design bugs?
  - If most chip fails with the same syndrome when running an application
- Is it due to parametric yield loss?
  - Timing-related failure?
    - Insufficient silicon speed?
  - Noise-induced failure?
    - supply noise, cross-talk, leakage, etc.?
  - Lack of manufacturability?
    - inappropriate layout?
- Is it due to random defects?
  - Via misalignment, Via/Contact void, Mouse bite,
  - Unintentional short/open wires, etc.
Problem: Fault Diagnosis

This chapter focuses more on diagnosis of defects or faults, not design bugs

Circuit Under Diagnosis (CUD)

expected response

not equal!

faulty response

given test patterns

a chip with defects inside

Question: Where are the fault locations?
Diagnosis For Yield Improvement

- Golden Reference Model
- Logic Diagnosis
- A Set Of Potential Defect Locations

Physical Failure Analysis
- Scanning Electronic Microscope (SEM)
- Focused Ion Beam (FOB)

Defect Mechanisms

Tune the Manufacturing Process or Design for Yield Improvement

Via void, Mouse bite, etc.
Quality Metrics of Diagnosis

- **Success rate**
  - The percentage of hitting at least one defect in the physical failure analysis
  - This is the ultimate goal of failure analysis

- **Diagnostic resolution**
  - Total number of fault candidates reported by a tool
  - The perfect diagnostic resolution is 1
  - Though perfect resolution does not necessarily imply high hit rate

- **First-hit index**
  - Used for a tool that reports a ranked list of candidates
  - Refers to the index of the first candidate in the ranked list that turns out to be a true defect site
  - Smaller first-hit index indicates higher accuracy

- **Top-10 hit**
  - Used when there are multiple defects in the failing chip
  - The number of true defects in the top 10 candidates
Do whatever you want, but give me that damn bug(s) in less than 5 candidates.

failure analysis people under time-to-market pressure
Supporting Circuitry:
Makes Logic’s inputs controllable and outputs observable
Design For Diagnosis

Complexity Of Diagnosis

Original Design

Separated Logic & Memory

interface circuitry

Scan-chain

Logic Design With Full-Scan

More Supporting Circuitry

VLSI Test Principles and Architectures  Ch. 7 - Logic Diagnosis  - P. 9
Possible Assumptions Used in Diagnosis

- **Stuck-At Fault Model Assumption**
  - The defect behaves like a stuck-at fault

- **Single Fault Assumption**
  - Only one fault affecting any faulty output

- **Logical Fault Assumption**
  - A fault manifests itself as a logical error

- **Full-Scan Assumption**
  - The chip under diagnosis has to be full-scanned

Note: A diagnosis approach less dependent on the fault assumptions is more capable of dealing with practical situations.
Examples of Faults

- **Node Fault**
- **Short Fault (Bridging)**

Most diagnosis algorithms perform at the gate level, trying to identify the troubling signals or cells.
Byzantine Open Fault

Definition of Byzantine Fault:
- A fault that causes an ambiguous voltage level

\[ \alpha \sim 2.5 \text{ V} \]

- open fault

- pseudo ‘0’

- pseudo ‘1’

- ‘0’

- ‘1’

G1

G2

G3
A Byzantine Node Type

Truth Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Z</th>
<th>Z'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>~0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The faulty output could be ambiguous

VLSI Test Principles and Architectures

Ch. 7 - Logic Diagnosis - P. 13
Fault Classification

Fault in Logic IC

- **Functional Fault**
  - Node Fault
  - Open Fault
  - Short Fault
  - Byzantine Fault

- **Delay Fault**
  - Gate-Delay Fault
  - Path-Delay Fault

affects functionality

affects timing
Outline

- Introduction
- **Combinational Logic Diagnosis**
  - Cause-Effect Analysis
  - Effect-Cause Analysis
  - Chip-Level Strategy
  - Diagnostic Test Pattern Generation
- Scan Chain Diagnosis
- Logic BIST Diagnosis
- Conclusion
Terminology

- **Device Under Diagnosis (DUD):** The Failing Chip
- **Circuit Under Diagnosis (CUD):** The Circuit Model
- **Failing Input Vector:** Causes Mismatches

![Diagram of Failing chip and Gate-level CUD with input vector and output mismatches]
**Cause-Effect Analysis**

- **Fault dictionary (pre-analysis of all causes)**
  - Records test response of every fault under the applied test set
  - Built by intensive fault simulation process
- **A chip is diagnosed (effect matching)**
  - By matching up the failing syndromes observed at the tester with the pre-stored fault dictionary
Fault Dictionary Example

(a) Circuit under diagnosis

(b) Full-response dictionary

<table>
<thead>
<tr>
<th>Circuits</th>
<th>Test vectors in terms of (a, b, c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(v_1)</td>
</tr>
<tr>
<td>fault-free</td>
<td></td>
</tr>
<tr>
<td>(f_1)</td>
<td>0</td>
</tr>
<tr>
<td>(f_2)</td>
<td>0</td>
</tr>
<tr>
<td>(f_3)</td>
<td>1</td>
</tr>
<tr>
<td>(f_4)</td>
<td>1</td>
</tr>
<tr>
<td>(f_5)</td>
<td>0</td>
</tr>
</tbody>
</table>

(c) Diagnostic tree

A diagnosis session: traverse from a path from root to a leaf
## Fault Dictionary Reduction – P&R

### (a) Full-response table

<table>
<thead>
<tr>
<th>Fault</th>
<th>Output Response ((z_1, z_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_1)</td>
<td>(t_2)</td>
</tr>
<tr>
<td>(f_1)</td>
<td>10</td>
</tr>
<tr>
<td>(f_2)</td>
<td>00</td>
</tr>
<tr>
<td>(f_3)</td>
<td>00</td>
</tr>
<tr>
<td>(f_4)</td>
<td>01</td>
</tr>
<tr>
<td>(f_5)</td>
<td>01</td>
</tr>
<tr>
<td>(f_6)</td>
<td>01</td>
</tr>
<tr>
<td>(f_7)</td>
<td>10</td>
</tr>
<tr>
<td>(f_8)</td>
<td>11</td>
</tr>
</tbody>
</table>

### (b) Pass-fail dictionary

<table>
<thead>
<tr>
<th>Fault</th>
<th>Pass (0) or Fail (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_1)</td>
<td>(t_2)</td>
</tr>
<tr>
<td>(f_1)</td>
<td>1</td>
</tr>
<tr>
<td>(f_2)</td>
<td>1</td>
</tr>
<tr>
<td>(f_3)</td>
<td>1</td>
</tr>
<tr>
<td>(f_4)</td>
<td>1</td>
</tr>
<tr>
<td>(f_5)</td>
<td>1</td>
</tr>
<tr>
<td>(f_6)</td>
<td>1</td>
</tr>
<tr>
<td>(f_7)</td>
<td>1</td>
</tr>
<tr>
<td>(f_8)</td>
<td>0</td>
</tr>
</tbody>
</table>

### (c) P&R compression dictionary

<table>
<thead>
<tr>
<th>Fault ID</th>
<th>Pass-fail + Extra outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(t_1)</td>
</tr>
<tr>
<td>(f_1)</td>
<td>1</td>
</tr>
<tr>
<td>(f_2)</td>
<td>1</td>
</tr>
<tr>
<td>(f_3)</td>
<td>1</td>
</tr>
<tr>
<td>(f_4)</td>
<td>1</td>
</tr>
<tr>
<td>(f_5)</td>
<td>1</td>
</tr>
<tr>
<td>(f_6)</td>
<td>1</td>
</tr>
<tr>
<td>(f_7)</td>
<td>1</td>
</tr>
<tr>
<td>(f_8)</td>
<td>0</td>
</tr>
</tbody>
</table>

Response of \(z_1\)  
Response of \(z_2\)
### Detection Fault Dictionary

#### (a) Full-response table

<table>
<thead>
<tr>
<th>Fault ID</th>
<th>Output Response ((z_1, z_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_1)</td>
<td>(10) (10) (11) (10)</td>
</tr>
<tr>
<td>(f_2)</td>
<td>(00) (00) (11) (00)</td>
</tr>
<tr>
<td>(f_3)</td>
<td>(00) (00) (00) (00)</td>
</tr>
<tr>
<td>(f_4)</td>
<td>(01) (00) (00) (01)</td>
</tr>
<tr>
<td>(f_5)</td>
<td>(01) (00) (01) (01)</td>
</tr>
<tr>
<td>(f_6)</td>
<td>(01) (00) (01) (01)</td>
</tr>
<tr>
<td>(f_7)</td>
<td>(10) (00) (10) (00)</td>
</tr>
<tr>
<td>(f_8)</td>
<td>(11) (11) (11) (11)</td>
</tr>
</tbody>
</table>

#### (b) Pass-fail dictionary

<table>
<thead>
<tr>
<th>Fault ID</th>
<th>Pass (1) or Fail (0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_1)</td>
<td>(1) (1) (0) (1)</td>
</tr>
<tr>
<td>(f_2)</td>
<td>(1) (0) (0) (1)</td>
</tr>
<tr>
<td>(f_3)</td>
<td>(1) (0) (1) (1)</td>
</tr>
<tr>
<td>(f_4)</td>
<td>(1) (0) (1) (0)</td>
</tr>
<tr>
<td>(f_5)</td>
<td>(1) (0) (1) (0)</td>
</tr>
<tr>
<td>(f_6)</td>
<td>(1) (0) (1) (0)</td>
</tr>
<tr>
<td>(f_7)</td>
<td>(1) (0) (1) (1)</td>
</tr>
<tr>
<td>(f_8)</td>
<td>(0) (1) (0) (1)</td>
</tr>
</tbody>
</table>

#### (c) Detection dictionary

<table>
<thead>
<tr>
<th>Fault ID</th>
<th>Detection information ((\text{Test ID : Output Vector}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_1)</td>
<td>(t_1:10) (t_2:10) (t_4:10);</td>
</tr>
<tr>
<td>(f_2)</td>
<td>(t_1:00) (t_4:00);</td>
</tr>
<tr>
<td>(f_3)</td>
<td>(t_1:00) (t_3:00) (t_4:00);</td>
</tr>
<tr>
<td>(f_4)</td>
<td>(t_1:01) (t_3:00);</td>
</tr>
<tr>
<td>(f_5)</td>
<td>(t_1:01) (t_3:01);</td>
</tr>
<tr>
<td>(f_6)</td>
<td>(t_1:01) (t_3:01);</td>
</tr>
<tr>
<td>(f_7)</td>
<td>(t_1:10) (t_3:10) (t_4:00);</td>
</tr>
<tr>
<td>(f_8)</td>
<td>(t_2:10) (t_4:11);</td>
</tr>
</tbody>
</table>

Detection information:
- \(f_1\): \(t_1:10\) \(t_2:10\) \(t_4:10\)
- \(f_2\): \(t_1:00\) \(t_4:00\)
- \(f_3\): \(t_1:00\) \(t_3:00\) \(t_4:00\)
- \(f_4\): \(t_1:01\) \(t_3:00\)
- \(f_5\): \(t_1:01\) \(t_3:01\)
- \(f_6\): \(t_1:01\) \(t_3:01\)
- \(f_7\): \(t_1:10\) \(t_3:10\) \(t_4:00\)
- \(f_8\): \(t_2:10\) \(t_4:11\)
Outline

- Introduction
- Combinational Logic Diagnosis
  - Cause-Effect Analysis
  - Effect-Cause Analysis
  - Chip-Level Strategy
  - Diagnostic Test Pattern Generation
- Scan Chain Diagnosis
- Logic BIST Diagnosis
- Conclusion
Effect-cause analysis does not build fault dictionary
It predicts fault locations by analyzing CUD from mismatch PO’s
Structural Pruning – Intersection or Union?

(a) Cone intersection.

(b) Cone union when there are multiple faults.
**Backtrace Algorithm**

- **Trace back from each mismatched PO**
  - To find out suspicious faulty locations

- **Functional Pruning**
  - During the traceback, some signals can be disqualified from the fault candidate set based on their signal values.

- **Rules**
  - (1) At a controlling case (i.e., 0 for a NAND gate): Its fanin signals with non-controlling values (i.e., 1) are excluded from the candidate set.
  - (2) At a non-controlling case (i.e., 1 for a NAND gate): Every fanin signal remains in the candidate set.
Backtrace Example

All suspicious fault locations are marked in red.

Target mismatched output

VLSI Test Principles and Architectures

Ch. 7 - Logic Diagnosis - P. 25
**Terminology – Injection**

An injection at a signal \( f \) flips its current value which could create value-change events downstream.

O: correct output  
X: failing output

A mismatched output could be fixed by the injection!
Terminology – Curable Output

- **Diagnosis Criterion**
  - A signal is more suspicious if it has more curable outputs

  - An injection at f fixes two mismatched outputs
  - Thus, f has two curable outputs!
**Terminology – Curable Vectors**

v is a curable vector by f

⇒ because an injection at f exists such that it cures all mismatches without creating new one

*Curable vector is a stronger diagnosis indicator than curable output!*

---

**Diagram:**

- For \( f = '1' \): The vector v is applied and cures all mismatches.
- For \( f = '0' \): The vector v is applied and cures all mismatches.

---

VLSI Test Principles and Architectures

Ch. 7 - Logic Diagnosis - P. 28
Example of Curable Vector

(a) Failing Chip

\[
\begin{align*}
\text{x}_1 &= 0 \\
\text{x}_2 &= 1 \\
\text{x}_3 &= 1 \\
\text{x}_4 &= 1 \\
\text{failing} &\quad 0 \\
\text{cured} &\quad 1
\end{align*}
\]

(b) Circuit Under Diagnosis

VLSI Test Principles and Architectures

Ch. 7 - Logic Diagnosis - P. 29
Why Curable Vector?

- Information theory
  - A less probable event contains more information
  - Curable output is an easy-to-satisfy criterion, high aliasing
  - Curable vector is a hard-to-satisfy criterion, low aliasing

- Not all failing input vectors are equal!

- Niche input vector
  - Is an failing input vector that activates only one fault
  - Likely to be a curable vector of certain signals
  - Few, but tells more about the real fault locations
**Inject-and-Evaluate Paradigm**

1. **input vectors**
2. **design model**
3. **failing chip response**

- Calculate the no. of “curable vectors” of each signal
- Calculate the no. of “curable outputs” of each signal

**Sorting Criteria**
- Sort the signals by the no. of “correctable vectors”, if tied, sort by the no. of “correctable outputs”

 ranking of each signal’s possibility of being a defect location
for each failing input vector $v$ 

Step 1: perform logic simulation;
Step 2: for each candidate signal $f$ 

Step 2.1: flip the value at $f$; /* injection */
Step 2.2: run event-driven fault simulation; /* evaluation */
Step 2.3: calculate certain metrics /* ranking */

} 

Sort the candidate signals by the calculated metrics;

a list of ranked candidate signals
**Reward-and-Penalty Heuristic**

*Rank1: curable vector count*
*Rank2 = (curable output count – 0.5 * new mismatched output count)*

(a) Failing Chip.

(b) Circuit Under Diagnosis.
Even in a realistic bridging fault, there is only one victim at any time. This victim will expose his location by owning some curable vectors.
SLAT Paradigm

Ref: SLAT (Single Location At a Time) paradigm [Bartenstein 2001]
Note: A SLAT vector is a curable vector

Phase 1: Finding SLAT (Single Location at A Time) vectors:
(1) Fault simulation, (2) Output matching

Phase 2: Finding valid fault multiplets
(1) Finding single-fix candidates
(2) Finding double-fix candidates
(3) Finding triple-fix candidates, etc.

A number of valid fault multiplets

SLAT Paradigm

Phase 1: Finding SLAT (Single Location at A Time) vectors:
(1) Fault simulation, (2) Output matching

Phase 2: Finding valid fault multiplets
(1) Finding single-fix candidates
(2) Finding double-fix candidates
(3) Finding triple-fix candidates, etc.

A number of valid fault multiplets
### Example: SLAT Paradigm

<table>
<thead>
<tr>
<th>Failing Input Vectors</th>
<th>Signals in the CUD</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>( \nu_1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \nu_2 )</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \nu_3 )</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \nu_4 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>( \nu_5 )</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \nu_6 )</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \nu_7 )</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \nu_8 )</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>( \nu_9 )</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>( \nu_{10} )</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

A mark * means the corresponding vector is a SLAT vector of the corresponding signal.

(f_3 and f_5) is a valid fault multiplet.
Outline

- Introduction
- Combinational Logic Diagnosis
  - Cause-Effect Analysis
  - Effect-Cause Analysis
  - Chip-Level Strategy
  - Diagnostic Test Pattern Generation
- Scan Chain Diagnosis
- Logic BIST Diagnosis
- Conclusion
Fault $f_1$ is an independent fault. Faults $f_2$ and $f_3$ are dependent faults.
Two independent faults, $f_1$ and $f_2$, lead to one diagnosis block.

*Direct divide-and-Conquer does not work well!*

Dependency Graph

one connected component
Main Strategy: Detach-Divide-and-then-Conquer

- Phase 1: Isolate Independent Faults
  - Search for prime candidates
  - Use word-level information
- Phase 2: Locate Dependent Faults As Well
  - Perform partitioning
  - Aim at finding one fault in each block
A signal $f$ is a prime candidate if
(1) All failing input vectors are partially curable by $f$
(2) Curable-Output-Set($f$) is not covered by any other’s

$f_1$ & $f_2$ are prime!
Fake Prime Candidates

- Structurally Independent Faults
  - are often prime candidates
- Fake Prime Candidates
  - are prime candidates that are NOT really faults - aliasing

Example: Dependent Double Faults $f_1$ & $f_2$
May create fake prime candidates \{f_1, f_2, f_3\}. 
Word-Level Registers and Outputs

Signals in a design are often defined in words. This property can be used to differentiate fake prime candidates from the real ones.

Word-Level Output: O1
Word-Level Registers: R1, R2, State

```verilog
module design( O1, ...)
    output[31:0] O1;
    reg[31:0] R1, R2;
    reg[5:0] State
    ...
endmodule
```
**Word-Level Prime Candidates**

Note: Z and R are two word-level output groups.

Original prime candidates: \{f_1, f_2\}
Word-level prime candidates \{f_1, f_2\}

Assumed original prime candidates: \{f_3, f_4, f_5\}
\{f_4, f_5\} will be identified as fake
⇒ Final Word-level prime candidates \{f_3\}
Efficiency of Using Word-Level Info.

- Without word-level Information
  - 2.4 real faults out of 72.3 candidates
- With word-level Information
  - 1.23 real faults out of 3.65 candidates

<table>
<thead>
<tr>
<th># of candidates</th>
<th>Original</th>
<th>After Filtering</th>
<th>Filtering Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prime Candidates</td>
<td>2.375</td>
<td>1.23</td>
<td>48.2 %</td>
</tr>
<tr>
<td>Fake Prime Candidates</td>
<td>69.96</td>
<td>2.42</td>
<td>96.5 %</td>
</tr>
</tbody>
</table>
**Overall Flow**

Phase 1:
(1) Find Word-Level Prime Candidates

Phase 2:
(1) Remove *explained outputs and their fanin cones*
(2) Partition the rest model into blocks
(3) Perform diagnosis for each block

Rank candidates produced in phases 1 & 2
An example with five faults
One of them is identified as the prime candidate

VLSI Test Principles and Architectures  Ch. 7 - Logic Diagnosis  - P. 47
removed explained faulty outputs

prime candidates

syndromes at y and z are fully explained
Grouping Example

Two independent diagnosis blocks Are successfully derived!
Summary

- **Strategy**
  - (1) Search For Word-Level Prime Candidates
  - (2) Identify Independent Faults First
  - (3) Locate Dependent Faults As Well

- **Effectiveness**
  - identify 2.98 faults in 5 signal inspections
  - find 3.8 faults in 10 signal inspections
Diagnostic Test Pattern Generation

- Fault-free circuit
- DTPG helps to increase diagnostic resolution

Model for differentiating vector generation

VLSI Test Principles and Architectures
Ch. 7 - Logic Diagnosis - P. 51
Outline

- Introduction
- Combinational Logic Diagnosis
- Scan Chain Diagnosis
  - Preliminaries
  - Hardware-Assisted Method
  - Signal-Profiling Based Method
- Logic BIST Diagnosis
- Conclusion
Flush test of scan chains
(pumping random patterns and checking response)

Pass or Fail?

Pass
Test Combinational Logic

Fail
Find failing scan chain(s)
Classify fault types
Scan Chain Diagnosis
Commonly Used Fault Types in Scan Chains

- **Scan Chain Faults**
  - **Functional Faults**
    - Stuck-at
    - Bridging
  - **Timing Faults**
    - Setup-Time Violation Fault
    - Hold-Time Violation Fault
    - Slow-To-Rise Fault
    - Slow-To-Fall Fault

Each fault could be permanent or intermittent.
A Stuck-At Fault In the Chain

Effect: A killer of the scan-test sequence

VLSI Test Principles and Architectures Ch. 7 - Logic Diagnosis - P. 55
A Realistic Bridging Fault Model

(a) Bridging between a flip-flop and a logic cell.

\[ \text{If}(\beta = 1) \quad \alpha^\text{faulty} = \alpha \]
\[ \text{else} \quad \alpha^\text{faulty} = F_2 \]

(b) Our bridging fault model.
Potential Hold-Time Fault?

(Negative Edge-Triggered Flip-Flop)

CLK = high

Master Slave

CLK = low

normal

Master Slave

faulty

shut down too slowly

CLK = low

Master Slave
**Example: Faulty Syndrome of a Scan Chain**

A scan chain

SI (scan input pin)  SO (scan output pin)

A faulty flip-flop

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Scan-In Pattern</th>
<th>Observed Syndrome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuck-at-0</td>
<td>1100110011001100</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>Stuck-at-1</td>
<td>1100110011001100</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>Slow-to-Rise</td>
<td>1100110011001100</td>
<td>1000100010001000</td>
</tr>
<tr>
<td>Slow-to-Fall</td>
<td>1100110011001100</td>
<td>1101110111011100</td>
</tr>
</tbody>
</table>

The rightmost bit goes into the scan first

The rightmost bit gets out of the scan first

A underlined bit in the observed image is failing.
Augmentation of a Flip-Flop for Easy Diagnosis

(a) A normal scan flip-flop.

(b) A modified scan flip-flop for easy inversion.
Fault Location via Inversion Operation

(1) Original bitstream pattern = (1111111111111111)
(2) After scan-in: snapshot image = (1111000000000000)
(3) After inversion: snapshot image = (0000011111111111)
(4) After scan-out: observed image = (0000011111111111)

The fault location is at the edge between 0’s and 1’s
Scan Chain Diagnosis Flow

- Circuit Under Diagnosis
- Diagnostic Test Sequence Generator
- Diagnostic Test Sequences
- Test Application
- Fault-Free Observed Images
- Diagnosis
- Faulty FF’s location
- Signal Profiling Based Diagnosis Program
- Observed Images Of Failing Chip
**Definition: Snapshot Image**

Def: A snapshot image is the combination of flip-flop values at certain time instance.

Snapshot image: \( \{ (F_1, F_2, F_3, F_4) \mid (0, 1, 0, 1) \} \)
**Definition: Observed Image**

Def: An observed image is the scanned-out version of a snapshot image.

**Snapshot image:** \( \{(F_1, F_2, F_3, F_4) \mid (0, 1, 0, 1)\} \)

**Observed image:** \( \{(F_1, F_2, F_3, F_4) \mid (0, 0, 0, 1)\} \)
A stuck-at-0 fault is assumed at the output of the 2\textsuperscript{nd} FF from SI.
Test Application: Run-and-Scan

Step 1: Apply a test sequence from PI’s

Setting up a snapshot image at FF’s

Step 2: Scan-out an observed image

Less distorted image

up-stream part will be distorted

The fault location is embedded in the observed image
Signal Profiling

A profile is the distribution of certain statistics of the flip-flops.

Comparing failing profile with the fault-free profile ➔ Could reveal the fault location
Profile Analysis

Fault-free images (say 100 of them)

Failing images (say 100 of them)

Collected from tester

Derive the fault-free profile

Derive the failing profile

Derive the difference profile

A difference image = fault-free image ⊕ failing image

Perform filtering on the difference profile

Perform edge detection to derive ranking profile

report a ranked list of fault locations
Example: Filtering & Edge Detection

**Difference Profile**

![Difference Profile Graph]

**Filtered Difference Profile**

![Filtered Difference Profile Graph]

**Ranking (or suspicion) Profile**

![Ranking Profile Graph]

VLSI Test Principles and Architectures  
Ch. 7 - Logic Diagnosis  
- P. 68
(Average-sum filtering) Assume that the difference profile is given and denoted as $D[i]$, where $i$ is the index of a flip-flop. We use the following formula to compute a smoothed difference profile, $SD[i]$:

$$SD[i] = 0.2*(D[i-2] + D[i-1] + D[i] + D[i+1] + D[i+2])$$
The true location of the faulty flip-flop is likely to be the *left-boundary of the transition region in the difference profile*. To detect this boundary, we can use a simply *edge detection formula* defined below.

(Edge detection) On the smoothed difference profile $SD[i]$, the following formula can be used to compute the faulty frequency of each flip-flop as a suspicious profile.

$$\text{suspicion}[i] = [-1, -1, -1, 1, 1, 1, 1, 1, 1] \cdot \begin{bmatrix} SD[i] - SD[i - 3] \\ SD[i] - SD[i - 2] \\ SD[i] - SD[i - 1] \\ SD[i] - SD[i + 1] \\ SD[i] - SD[i + 2] \\ SD[i] - SD[i + 3] \end{bmatrix}$$
Summary of Scan Chain Diagnosis

- **Hardware Assisted**
  - Extra logic on the scan chain
  - Good for stuck-at fault

- **Fault Simulation Based**
  - To find a faulty circuit matching the syndromes [Kundu 1993] [Cheney 2000] [Stanley 2000]
  - Tightening heuristic → upper & lower bound [Guo 2001][Y. Huang 2005]
  - Use single-excitation pattern for better resolution [Li 2005]

- **Profiling-Based Method**
  - Locate the fault directly from the difference profiles obtained by run-and-scan test
  - Applicable to bridging faults
  - Use signal processing techniques such as filtering and edge detection
Outline

- Introduction
- Combinational Logic Diagnosis
- Scan Chain Diagnosis
- Logic BIST Diagnosis
  - Overview
  - Interval-Based Method
  - Masking-Based Method
- Conclusion
A Logic BIST Architecture

PRPG (Pseudo-Random Pattern Generator)

Core Logic

MISR (Multiple-Input Signal Analyzer)

All flip-flops are assumed to be observable through scan chains.

scan out (as the signature)
Diagnosis for BISTed Logic

- Diagnosis in a BIST environment requires:
  - determining from compacted output responses which test vectors have produced a faulty response (time information)
  - determining from compacted output responses which scan cells have captured errors (space information)

- The true fault location inside the logic
  - Can then be inferred from the above space and time information using previously discussed combinational logic diagnosis
Binary Search To Locate 1st Failing Vector

Time (or test vector index)

Space (or scan cell index)

1st BIST session

BIST session length:
14 → 7 → 4 → 2 → 3

First failing at vector #4
Interval Unloading-Based Diagnosis

A signature is scanned out to the tester for comparison at the end of each interval.
Deterministic Masking-Based Diagnosis

(a) STUMP-based BIST architecture

VLSI Test Principles and Architectures

(b) Scan cell matrix

Ch. 7 - Logic Diagnosis - P. 77
Circuitry to Support Deterministic Masking

PRPG (Pseudo-Random Pattern Generator)

Core Logic

MISR (Multiple-Input Signal Analyzer)

VLSI Test Principles and Architectures  Ch. 7 - Logic Diagnosis - P. 78
A Search for Scan Cells Capturing Errors

(a) Scan cells capturing errors in the fourth scan chain

(b) The search tree

(Y, Z)=(1, 7)

(Y, Z)=(1, 4)

(Y, Z)=(1, 2)

(Y, Z)=(3, 3)

(Y, Z)=(3, 4)

(Y, Z)=(4, 4)

(Y, Z)=(5, 7)

(Y, Z)=(5, 6)

(Y, Z)=(7, 7)

9 BIST sessions
Conclusions

- Logic diagnosis for combinational logic
  - Has been mature
  - Good for not just stuck-at faults, but also bridging faults

- Scan chain diagnosis
  - Making good progress …
  - Fault-simulation-based, or signal-profiling based

- Diagnosis of scan-based logic BIST
  - Hardware support is often required
  - Interval-unloading, or masking-based

- Future challenges
  - Performance (speed) debug
  - Diagnosis for logic with on-chip test compression and decompression
  - Diagnosis for parametric yield loss due to nanometer effects