IHS 3: Test of Digital Systems

R. Ubar, A. Jutman, H-D. Wuttke
RT-Level Design

- data path and control path on RT-level
- RT level simulation
- Functional units (F1,...,F4)
- Register
- Multiplexer / Demultiplexer
Fault Modeling on High Level DDs

High-level DDs (RT-level):

<table>
<thead>
<tr>
<th>( y_1 )</th>
<th>Function</th>
<th>( M_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( M_1 = R_1 )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( M_1 = IN )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( y_2 )</th>
<th>Function</th>
<th>( M_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( M_2 = R_1 )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( M_2 = IN )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( y_3 )</th>
<th>Function</th>
<th>( M_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( M_3 = M_1 + R_2 )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( M_3 = IN )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( M_3 = R_1 )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( M_3 = M_2 * R_2 )</td>
<td></td>
</tr>
</tbody>
</table>

\( R_2 \):

<table>
<thead>
<tr>
<th>( y_4 )</th>
<th>Operation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reset</td>
<td>( R_2 = 0 )</td>
</tr>
<tr>
<td>1</td>
<td>Hold</td>
<td>( R_2 = R_2' )</td>
</tr>
<tr>
<td></td>
<td>Load</td>
<td>( R_2 = M_3 )</td>
</tr>
</tbody>
</table>
Fault Modeling on High Level DDs

**High-level DDs (RT-level):**

Terminal nodes represent:
- RTL-statement faults: data storage, (e.g. #0)
- data transfer, (e.g. IN)
- data manipulation faults (e.g. \( R_1 \cdot R_2 \))

Nonterminal nodes represent:
- RTL-statement faults: label, timing condition, logical condition, register decoding, operation decoding, control faults

Terminal nodes represent:
- RTL-statement faults: data storage, (e.g. #0)
- data transfer, (e.g. IN)
- data manipulation faults (e.g. \( R_1 \cdot R_2 \))
High-Level DDs for a microprocessor (example):

Instruction set:

\[ I_1: \text{MVI A,D} \rightarrow A \leftarrow \text{IN} \]
\[ I_2: \text{MOV R,A} \rightarrow R \leftarrow A \]
\[ I_3: \text{MOV M,R} \rightarrow \text{OUT} \leftarrow R \]
\[ I_4: \text{MOV M,A} \rightarrow \text{OUT} \leftarrow A \]
\[ I_5: \text{MOV R,M} \rightarrow R \leftarrow \text{IN} \]
\[ I_6: \text{MOV A,M} \rightarrow A \leftarrow \text{IN} \]
\[ I_7: \text{ADD R} \rightarrow A \leftarrow A + R \]
\[ I_8: \text{ORA R} \rightarrow A \leftarrow A \lor R \]
\[ I_9: \text{ANA R} \rightarrow A \leftarrow A \land R \]
\[ I_{10}: \text{CMA A} \rightarrow A \leftarrow \lnot A \]

DD-model of the microprocessor:
Hierarchical Test Generation

Hierarchical test (fault propagation)

Defect mapping

Transistor level

Logic level

System level

Error

Component level

dy

y*

Logic level

RT Level

IN

M1

M2

M3

M4

M5

R1

R2

R3

Defect
Experiments

• Use the example A+B/2 (average value)
• Find for each test method best parameters
  – Functional test
  – Deterministic test
  – Functional BIST
  – Logical BIST
  – Circular BIST
• Note them and compare results
High-Level Decision Diagrams Applet

- Algorithm: average
- \((A+B)/2\)

- a) high speed
  - 2 operation units (F2, F3)
  - Add and Div in 1 clock cycle
  - 4 steps
  - Cost: 83
High-Level Decision Diagrams Applet

- Algorithm: average
- \((A+B)/2\)

- b) low cost
  - Only 1 Unit (F4)
  - 2 clock cycles for operation
  - 5 clocks
  - Cost: 73
High-Level Decision Diagrams Applet

- Algorithm: high speed average
- \((A+B)/2\) (4 steps Cost: 83)

- Simulation
  - Load from Input Reg1
  - Load from Input Reg2
  - Add, Div and store in 1 clock cycle
  - Output and End

<table>
<thead>
<tr>
<th>Ad</th>
<th>Next</th>
<th>F1(in)</th>
<th>F2(in)</th>
<th>F4</th>
<th>IN</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F3(out)</th>
<th>F4</th>
<th>OUT</th>
<th>Input</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>REG1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>REG2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>REG1</td>
<td>REG2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>END</td>
<td>REG1</td>
<td>REG2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Applets for Learning RT- Level Test

**Functional Test mode:**
The test vectors are operands, results can be observed at the output, no extra test parts.

**Deterministic Test mode:**
Gate level test for each FU separately, generate test vectors, local test panel, cumulative FC calculation.

**BIST mode:**
Functional-, Logical- Circular Built In Self Test via extra test part: Test Pattern Generator and Signature analyzer.
Functional Test

- The test vectors are operands, results can be observed at the output,
- no extra test parts needed
Functional Test

- **Algorithm:** average
- **(A+B)/2**

- **Funct.**
- **Test**

<table>
<thead>
<tr>
<th>Test Nr.</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Nr.</th>
<th>Clock</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>IN</th>
<th>OUT</th>
<th>State</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Deterministic Test

- Adder (F2)
  - Schematic level
  - Gate level test
  - Boolean derivation
  - SSBDDs
  - Path activation
  - Fault propagation
  - Fault observation
Deterministic Test

- Insert test vectors manually
- simulate the fault coverage of the vector (FC vec) and the sequence (FC)
Overview

1. Introduction
2. Theory: Boolean differential algebra
3. Theory: Decision diagrams
4. Fault modelling
5. Test generation
6. Fault simulation
7. Fault diagnosis
8. Testability measuring
9. Design for testability
10. Built in Self-Test
Built-In Self-Test

• **Motivations for BIST:**
  – Need for a cost-efficient testing
  – 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
Built-In Self-Test in SoC

System-on-Chip testing

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
Built-In Self-Test Components

- 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
Linear Feedback Shift Register (LFSR)

Pseudorandom Test generation by LFSR:

**Standard LFSR**

**Modular LFSR**

Polynomial: \( P(x) = 1 + x^3 + x^4 \)
Built In Self Test (BIST)

- Functional
  - No additional part

- Logical
  - Additional TPG

- Circular
  - Additional TPG/SA
Built In Self Test (BIST)

- Functional
  - No additional part
- Logical
  - Additional TPG
  - Test Pattern Generator
- Circular
  - Additional TPG/SA
  - Signature Analyzer
LFSR

- Find configurations of the LFSR
  - (Linear Feedback Shift Register) that generates optimal test vectors (TPG/SA)
- Parameters:
  - Feedback polynomial
  - Initial State
BIST: Signature Analysis

- **TPG / SA**
  - Registers for LFSR

- **Configure**
  - Initial state
  - Polynom

- **Parameters**
  - TPG LFSR
  - SA LFSR
Logical BIST

- Find a configuration of the LFSR
  - (Linear Feedback Shift Register) that generates optimal operands (= test vektors)

- LFSR is generator and analyzer in one
LFSR: Signature Analyser

Response string for Signature Analysis

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

Stimuli
BIST: Signature Analysis

Aliasing:

UUT \rightarrow \text{Response} \rightarrow \text{SA}

\( L \) - test length

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

All possible responses

\[ k = 2^L \]

Faulty response

Correct response

All possible signatures

\[ k = 2^N \]

\( N << L \)
BIST: Signature Analysis

**Aliasing:**

![Diagram](image)

\[ k = 2^L \] - number of different possible responses

No aliasing is possible for those strings with \( L - N \) leading zeros since they are represented by polynomials of degree \( N - 1 \) that are not divisible by characteristic polynomial of LFSR. There are \( 2^{L-N} \) such strings

**Probability of aliasing:**

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

Parallel Signature Analyzer:

Single Input Signature Analyser

Multiple Input Signature Analyser (MISR)
Built-In Self-Test

Signature calculating for multiple outputs:

LFSR - Test Pattern Generator

Combinational circuit

Multiplexer

LFSR - Signature analyzer

LFSR - Test Pattern Generator

Combinational circuit

Multiplexer

LFSR - Signature analyzer
Test-per-Clock BIST Architectures

BILBO - Built-In Logic Block Observer:

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

CSTP - Circular Self-Test Path:

- LFSR - Test Pattern Generator & Signature analyser
- Combinational circuit
Reconfiguration for Self-Test

Module

Scan-IN

Pseudorandom test generator

Module

Scan-OUT

Test controller

Signature Analyser
Test-per-Scan BIST Architectures

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

LOCST: LSSD On-Chip Self-Test

Test Pattern Generator

R1 CC1

... Rn CCn

MISR

Scan Path

BS TPG SA

Test Controller

SI SO

CUT

BS

SO Error
Problems with BIST

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

**Problem:** low fault coverage

If Reset = 1 signal has probability 0.5 then counter will not work and 1 for AND gate may never be produced
Problems with BIST

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

**Problem:** low fault coverage

Possible patterns from LFSR:

Hard to test faults

**Dream solution:** find LFSR so that:
Store-and-Generate test architecture

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

Problem: low fault coverage

Long PR test:

Using many seeds:

Pseudorandom (PR) test:

Hard to test faults
Store-and-Generate test architecture

- ROM contains test patterns for hard-to-test faults
- Each pattern $P_k$ in ROM serves as an initial state of the LFSR for test pattern generation (TPG)
- Counter 1 counts the number of pseudorandom patterns generated starting from $P_k$
- After finishing the cycle for Counter 1, Counter 2 is incremented for reading the next pattern $P_{k+1}$
Hybrid Built-In Self-Test

Single test processes:

- Hybrid test set contains a limited number of pseudorandom and deterministic vectors
- Pseudorandom test is improved by a stored test set which is specially generated to target the random resistant faults
- Optimization

Pseudorandom Test  Deterministic Test
Hybrid BIST for Multiple Cores

Embedded tester for testing multiple cores
Functional Self-Test

Test patterns are produced on-line during the working mode of the system.
Functional Hybrid Self-Test

- Register block
- ALU
- MUX
- M
- Automatic Test Pattern Generator
- Deterministic test set
- Random resistant faults
- Test patterns are stored in the memory
- Signature analyser
- Data
- K
TURBO-TESTER: Low-Level TPG Tools

http://www.pld.ttu.ee/tt/

**Levels:**
- Gate
- Macro

**Methods:**
- Deterministic
- Random
- Genetic

**Test Generation**

**Methods:**
- BILBO
- CSTP
- Store/Generate

**Test**

**Fault models:**
- Stuck-at-faults
- Stuck-opens
- Delay faults

**Fault Location**

**Methods:**
- Single fault
- Parallel
- Deductive

**Fault Table**

**Test Optimization**

**Fault Diagnosis**
Overview

1. Introduction
2. Theory: Boolean differential algebra
3. Theory: Decision diagrams
4. Fault modelling
5. Test generation
6. Fault simulation
7. Fault diagnosis
8. Testability measuring
9. Design for testability
10. Built in Self-Test
Ad Hoc Design for Testability Techniques

- Find
  - good additional observation points that improve the Fault Coverage (FC)

- Find
  - optimal operands (= test vektors)

- Parameters:
  - Observation points
  - Operands
Ad Hoc Design for Testability Techniques

**Method of Test Points:**

Block 1 → Block 2

**Problem:**

Block 1 is not observable, Block 2 is not controllable

*Improving controllability and observability:*

1- controllability:
CP = 0 - normal working mode
CP = 1 - controlling Block 2 with signal 1

0- controllability:
CP = 1 - normal working mode
CP = 0 - controlling Block 2 with signal 0
Ad Hoc Design for Testability Techniques

Method of Test Points:

Block 1 ➔ Block 2

Block 1 is not observable,
Block 2 is not controllable

Improving controllability:

Normal working mode:
CP1 = 0, CP2 = 1
Controlling Block 2 with 1:
CP1 = 1, CP2 = 1
Controlling Block 2 with 0:
CP2 = 0

Normal working mode:
CP2 = 0
Controlling Block 2 with 1:
CP1 = 1, CP2 = 1
Controlling Block 2 with 0:
CP1 = 0, CP2 = 1
Ad Hoc Design for Testability Techniques

Multiplexing monitor points:

To reduce the number of output pins for observing monitor points, multiplexer can be used:

$2^n$ observation points are replaced by a single output and $n$ inputs to address a selected observation point.

**Disadvantage:**

only one observation point can be observed at a time
Ad Hoc Design for Testability Techniques

Demultiplexer for implementing control points:

To reduce the number of input pins for controlling testpoints, demultiplexer and a latch register can be used.

N clock times are required between test vectors to set up the proper control values.

\[ N = 2^n \]
Time-sharing of outputs for monitoring

To reduce the number of output pins for observing monitor points, time-sharing of working outputs can be introduced: no additional outputs are needed.

To reduce the number of inputs, again counter or shift register can be used.

Disadvantage:
only one observation point can be observed at a time.
Time-sharing of inputs for controlling

To reduce the number of input pins for controlling test points, time-sharing of working inputs can be introduced.

To reduce the number of inputs for driving the address lines of demultiplexer, counter or shift register can be used.
Ad Hoc Design for Testability Techniques

Partitioning of large combinational circuits:

The time complexity of test generation and fault simulation grows faster than a linear function of circuit size.

Partitioning of large circuits reduces these costs.

I/O sharing of normal and testing modes is used.

Three modes can be chosen:
- normal mode
- testing C1
- testing C2 (bolded lines)
Scan-Path Design

Scan-Path design allows to control and observe internal flip-flops, which means that the task of sequential testing has been transformed to the task of testing a combinational circuit.

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

### Diagram

- **IN**
- **Scan-IN**
- **Combinational circuit**
- **OUT**
- **Scan-OUT**
- **R**
- **q’**
- **q**
- **T**
- **Scan-IN**
- **& 1**
- **D T**
- **q’**
- **Test mode**: flip-flops are disconnected from the combinational circuit and connected to each other to form a shift register.
Scan-Path Design and Testability

Two possibilities for improving controllability/observability
Parallel Scan-Path

In parallel scan path flip-flops can be organized in more than one scan chain.
Hierarchical test generation with Scan-Path:

**Partial Scan Path**

**Scan-In**

Control Part

Data Part

**Scan-Out**

Bus
Logical BIST

- Test Microprogram
- Parameters
  - Program steps
  - Used registers
  - Tested functions
In random access scan each flip-flop in a logic network is selected individually by an address for control and observation of its state.
Boundary Scan Standard
Conclusions

• **Self-Test** of modules (components) is important, but is not the *Panacea*
• **Hierarchical and Functional Test** have the same importance
• **Functional Fault Model** is a universal means for mapping defects, faults and errors from level to level, providing a formal basis for hierarchical approach
• **Decision Diagrams** can be used successfully not only at logical levels, but also at higher RTL or behavioral levels
References

• **Contact data:**
  – Tallinn Technical University
  – Computer Engineering Department
  – Address: Raja tee 15, 12618 Tallinn, Estonia
  – Tel.: +372 620 2252, Fax: +372 620 2253
  – E-mail: raiub@pld.ttu.ee
  – www.ttu.ee/~raiub/
Overview

1. Introduction
2. Theory: Boolean differential algebra
3. Theory: Decision diagrams
4. Fault modelling
5. Test generation
6. Fault simulation
7. Fault diagnosis
8. Testability measuring
9. Design for testability
10. Built in Self-Test
Testability of Design Types

General important relationships:

- T (Sequential logic) < T (Combinational logic)
  Solutions: Scan-Path design strategy

- T (Control logic) < T (Data path)
  Solutions: Data-Flow design, Scan-Path design strategies

- T (Random logic) < T (Structured logic)
  Solutions: Bus-oriented design, Core-oriented design

- T (Asynchronous design) < T (Synchronous design)

- T: Testability
# Testability Estimations for Circuit Types

<table>
<thead>
<tr>
<th>Circuits less controllable</th>
<th>Circuits less observable</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Decoders</td>
<td>• Circuits with feedback</td>
</tr>
<tr>
<td>• Circuits with feedback</td>
<td>• Embedded</td>
</tr>
<tr>
<td>• Counters</td>
<td>– RAMs</td>
</tr>
<tr>
<td>• Clock generators</td>
<td>– ROMs</td>
</tr>
<tr>
<td>• Oscillators</td>
<td>– PLAs</td>
</tr>
<tr>
<td>• Self-timing circuits</td>
<td>• Error-checking circuits</td>
</tr>
<tr>
<td>• Self-resetting circuits</td>
<td>• Circuits with redundant</td>
</tr>
<tr>
<td></td>
<td>nodes</td>
</tr>
</tbody>
</table>
Testability Measures

Evaluation of testability:

- Controllability
  - $C_0 (i)$
  - $C_1 (j)$
- Observability
  - $O_Y (k)$
  - $O_Z (k)$
- Testability

Controllability for 1 needed

Defect
Probability of detecting $1/p^{60}$
Probabilistic Testability Measures

Controllability calculation:
Value: minimum number of nodes that must be set in order to produce 0 or 1
For inputs: \( C_0(i) = p(x_i=0) \quad C_1(i) = p(x_i=1) = 1 - p(x_i=0) \)
For other signals: recursive calculation rules:

\[
\begin{align*}
\text{For } x_1 \text{ and } x_2, & \\
& \quad p_y = p_{x1} p_{x2} \\
\text{For } x_1 \text{ and } 1, & \\
& \quad p_y = 1 - (1 - p_{x1})(1 - p_{x2}) \\
\text{For } x_1 \text{ to } x_n \text{ with } p_y = & \\
& \quad p_y = 1 - \prod_{i=1}^{n} (1 - p_{x_i}) \\
\end{align*}
\]
Probabilistic Testability Measures

Probabilities of reconverging fanouts:

\[
p_y = (1 - p_{x_1}) \cdot p_{x_2} + (1 - p_{x_2}) \cdot p_{x_1}
= 0.25 + 0.25 = 0.5
\]

\[
p_y = 1 - (1 - p_a) \cdot (1 - p_b)
= 1 - 0.75 \cdot 0.75 = 0.44
\]

Signal correlations:

\[
p_y = p_x \cdot p_x = p_x^2
\]
Calculation of Signal Probabilities

**Parker - McCluskey algorithm:**

\[ p_y = 1 - (1 - p_a)(1 - p_b) = \]
\[ = 1 - (1 - p_{x1}(1 - p_{x2}))(1 - p_{x2}(1 - p_{x1})) = \]
\[ = 1 - (1 - p_{x1} + p_{x1}p_{x2})(1 - p_{x2} + p_{x1}p_{x2}) = \]
\[ = 1 - (1 - p_{x2} + p_{x1}p_{x2} - p_{x1}p_{x2} + p_{x1}^2p_{x2} + p_{x2}^2p_{x1}) = \]
\[ = p_{x2} - p_{x1}p_{x2} + p_{x1} - p_{x1}p_{x2} + p_{x1}p_{x2} - p_{x1}p_{x2} + p_{x1}p_{x2} - p_{x1}p_{x2} = 0.5 \]
Calculation of Signal Probabilities

**Straightforward methods:**

\[\begin{align*}
\text{Calculation gate by gate:} \\
p_a &= 1 - p_1 p_2 = 0.75, \\
p_b &= 0.75, \quad p_c = 0.4375, \quad p_y = 0.22 \\
\text{Parker - McCluskey algorithm:} \\
p_y &= p_c p_2 = (1 - p_a p_b) p_2 = \\
&= (1 - (1 - p_1 p_2) (1 - p_2 p_3)) p_2 = \\
&= p_1 p_2^2 + p_2^2 p_3 - p_1 p_2^3 p_3 = \\
&= p_1 p_2 + p_2 p_3 - p_1 p_2 p_3 = 0.38
\end{align*}\]

For all inputs: \( p_k = 1/2 \)