Chapter 6  Digital System Testing

1. **Goal**
   Separate the good chips from the bad ones.

2. **Why?**
   The fabrication/manufacturing process of the VLSI chips is not a perfect process.
   ➔ A certain percentage of chips will not be manufactured correctly.

3. **Some definitions:**
   (i) A defect is the physical flaw that results in a bad chip.
   (ii) Defects are modeled as stuck-at-faults at the gate circuit level.
   (iii) An error has occurred when a circuit’s output is opposite to its correct output.

4. **Testing strategies**
   (i) **Exhaustive Testing**
       - All input patterns are generated and tested!
       - Prohibitively expensive.
   (ii) **Functional Testing**
       - Circuit’s verification vectors are used.
       - Incomplete coverage.
   (iii) **Structural Testing**
       - Model faults and generate test vectors for those faults.
       - Work in practice.

5. **Fault Model in Structural Testing**
   (i) **Single stuck at faults**
   (ii) Multiple stuck at faults
   (iii) Delay (transition) faults: slow to rise / fall.

§ 6.1 Combinational Logic Circuit Testing

1. **Control input for gates**
   (i) One input gate – Can’t have a control input.
   (ii) Two input gates
       - AND gate – ‘1’  OR gate – ‘0’
       - XOR gate – ‘X’  NAND gate – ‘1’
2. Example

<table>
<thead>
<tr>
<th>Wire</th>
<th>SA0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>x</th>
<th>0 √</th>
<th>11010 (SA0: f, h, i)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wire</th>
<th>SA0</th>
<th>1</th>
<th>1</th>
<th>x</th>
<th>x</th>
<th>0 √</th>
<th>01010 (SA1: f, h, g)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wire</th>
<th>SA0</th>
<th>1</th>
<th>1</th>
<th>x</th>
<th>x</th>
<th>0 √</th>
<th>11001 (SA1: i)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wire</th>
<th>SA0</th>
<th>0</th>
<th>x</th>
<th>1</th>
<th>1</th>
<th>0 √</th>
<th>01110 (SA0: g)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Minimum (Optional) Test Vector = \{11010, 01110, 11001, 01010\}

Maximum of Test Vectors = \{00000, 00001, \ldots, 11111\}
3. **Sequential Logic Circuit Testing**

   (i) Basic concept of a scan chain

   Any sequential circuits like this:
   The outputs not only based on the inputs, but also the states \( (y_1, y_2, \ldots, y_s) \)

   ![Combinational Logic Circuit Diagram](image_url)

   The basic idea: configure all memory elements as units in a shift register (scan) chain.

   FFs are replaced by MUX-FFs
   Extra pins are required: Scan\(_\text{in}\), Scan\(_\text{out}\), Test.
(ii) A way to test this: Scan-Path design

When T=0, D1 input is the D-input, system operates as usual
When T=1, All FFs turn into one long shift register and the combinational logic is by-passed

Test Procedure:
1. Set T=1, shift in test pattern, y_i
2. Set X inputs as required
3. Set T=0, after sufficient delay, check Z
4. Apply a clock signal to CLK
5. Set T=1, shift out the FF outputs via Z_m, at the same time, shift in the next y_i

(iii)Level Sensitive Scan Design (LSSD) – (IBM approach)
Instead of using FFs, IBM use two latches build in a master-slave form and two phase clock (Non-overlapping clock)
Test procedure:
1. Scan in test vector via SDI by applying pulses alternatively to the test clock input TCK and the system clock input CK2
2. Set the corresponding test values on the $X_i$ input. After sufficient propagation delay, check $Z_j$
3. Apply one clock pulse to system clock CK1 to enter new values of $y_j$ into corresponding master latches
4. Scan out and check the $y_j$ values by applying clock pulses alternatively to TCK and CK2

For most designs, scan-path / LSSD is a built-in part.
4. **ATPG – Automatic Test Pattern Generators**

   (i) D – Calculus

   Two steps:
   - Justify: to detect the fault
   - Propagate: set other unrelated pins to make it through

   D (can be ‘0’ or ‘1’), indicates that the fault free circuit should be 1 and the faulty circuit is 0. D’ is the inverse.

   For any gates, for example the two-input AND gate, we have

<table>
<thead>
<tr>
<th>AND</th>
<th>0</th>
<th>1</th>
<th>D</th>
<th>D'</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>D'</td>
<td>X</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>D</td>
<td>D</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>D'</td>
<td>0</td>
<td>D'</td>
<td>0</td>
<td>D</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

   Algorithm:
   ```
   /* Generate a test for a fault s-a-v on line l */
   {
   Set all input to x;
   Justify (l, v');
   If (v == 0) Propagate (l, D);
   Else Propagate (l, D');
   }
   -- v = hypothesized faulty level
   -- l = point
   
   Justify (l, Val);
   /* Determine primary input (PI) that justify a line setting */
   {
   If l is a PI, return;
   c = gate’s controlling value;
   i = gate’s inversion value;
   inval := c xor i;
   If (val == inval) select one of the gate inputs (j);
   Justify (j, c);
   ```
Else for each $j$ that is an input to the gate

\[
\text{Justfy (} j, \text{ val xor } i)\;
\]

Propagate ($l$, err)

\{
    \text{Set } l \text{ to err;}
    \text{If } l \text{ is a PO return;}
    c = \text{controlling value of the gate driven by } l;
    l = \text{inversion of the gate;}
    \text{For the gate's inputs (} j \text{)}
    \text{Justfy (} j, c'\text{)}
    k = \text{the output line of the gate;}
    \text{Propagate (} k, \text{ err xor } i)\;
\}

(ii) Some exceptional cases for this algorithm

- Wired logic should be avoided
- Dynamic logic should be avoided
- No one-shots, since these circuit are only external testable
- Redundant logic should be avoided
  Since it is not testable, and can cause other faults to be masked
- Analog & digital mixed will cause problem
  So, should be tested separately.

5. Examples

(i) Example 1: Case with Fanout that make system intestable

Stuck-at-1 at point F
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

To Justify

To Propagate

Problem: the fanout at point E

Justify needs E to be 0, while propagate needs E to be 1

Whenever you have fanout, this is likely to happen

Solution: Add more controllable or observable points to the circuit.

(ii) Example 2: Case with redundancy circuit

To eliminate static hazard, we need to add one more redundancy state

6. Three ways to solve ATPG problem

D-algorithm, PODEM, and FAM

Other than ATPG, there’s another way to test the circuit:

RTPG: Random Test Pattern Generation (actually, pseudo-random binary sequence)

Usually, both ATPG & RTPG are used to test devices.