Outline

1. Combinational testing
2. Sequential testing
Introduction to testing

- After fabrication, some circuits don’t work correctly
- Determining which circuits contain faults requires testing
- Testing some types of circuits is easy, testing others is difficult
  - One can use automation to help solve this problem
Determining the best way to test large circuits requires automation.

Building large circuits that are easy to test also requires automation.

Testing it spans all design, from system to physical level.

Today, I’ll introduce some classical ideas at the logic level.
Section outline

1. Combinational testing
   Yield
   Fault models
   Combinational test generation
Yield

- Yield, $y$, is the fraction of fault-free products.
- Test fault coverage, $c$, is the fraction of faults that the set of applied tests detects.
  - Low-yield is expensive because fabrication capacity is wasted.
- Defect level, $d$, is the fraction of parts containing undetected faults.
  - $d = 1 - y^{1-c}$
  - A high defect level is extremely expensive because it means circuits make their way into products, or worse, to consumers, before faults are discovered.
Defect level

- Example acceptable defect level: \( \sim 0.0002 \)
- Either yield must be very high or fault coverage must be very high
Section outline

1. Combinational testing
   Yield
   Fault models
   Combinational test generation
Faults and failures

- A *fault* is a physical defect in a circuit
- A *failure* is the deviation of a circuit from its specified behavior
- Faults can cause failures but they don’t always cause failures
Functional testing

- No assumptions about types of fault
- No assumptions about structure or properties of Circuit Under Test (CUT)
- The CUT is a black box that is checked to determine whether it responds to all (or most) input (sequences) as specified
- However, ignoring structural information makes testing unnecessarily slow
Structural testing

- Use information about the specific CUT
  - Types of faults that are likely to occur
  - Structure of circuit
Fault models

Combinational testing
Sequential testing

Yield
Fault models
Combinational test generation

Fault models

- Combinational test generation
- Fault models

Diagram of a digital logic circuit with inputs a and b, output z, and voltage sources $V_{DD}$ and $V_{SS}$.
Fault models

Combinational testing
Sequential testing
Yield
Fault models
Combinational test generation

Fault models

Combinational test generation
Fault models

- **Combinational testing**
- **Sequential testing**

**Yield**
**Fault models**
**Combinational test generation**

*Fault models*

**Stuck-open fault**

![Diagram showing a stuck-open fault](image)

- $V_{DD}$
- $V_{SS}$
- $a$
- $b$
- $z$

Diagram of a stuck-open fault in a circuit with inputs $a$ and $b$ and output $z$. The circuit is shown with voltage levels $V_{DD}$ and $V_{SS}$.
Fault models
Fault models

Combinational testing
Sequential testing

Yield
Fault models
Combinational test generation

Fault models

Combinational test generation

stuck-at fault

V_{DD}

a

b

z

V_{SS}

V_{DD}

a

b

z

stuck-at fault

12 Robert Dick  Advanced Digital Logic Design
Fault models
Singe stuck-at faults

- One of the simplest and most common fault models
  - S-a-0: Stuck-at 0
  - S-a-1: Stuck-at 1
- Relies on digital reinforcement
  - $0.8 \cdot V_{DD}$ is $1 \cdot V_{DD}$ one logic stage later
  - $S-a-0.8 \approx s-a-1$
- For two-level logic, exhaustive test set for single stuck-at faults will detect all multiple stuck-at faults, too
  - Doesn’t hold for multi-level logic
## Single stuck-at faults

Consider a NAND3 gate

<table>
<thead>
<tr>
<th>Inputs (A, B, C)</th>
<th>fault free</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
</tbody>
</table>
## Single stuck-at faults

Consider a NAND3 gate

<table>
<thead>
<tr>
<th>Inputs A B C</th>
<th>fault free</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Too expensive to use $2^n$ inputs ($2^3 = 8$ in this case)
## Single stuck-at faults

Consider a NAND3 gate

<table>
<thead>
<tr>
<th>Inputs A B C</th>
<th>fault free</th>
<th>fault a</th>
<th>fault b</th>
<th>fault c</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1</td>
<td>1 0</td>
<td>1 1</td>
<td>1 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1</td>
<td>1 1</td>
<td>1 0</td>
<td>1 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0</td>
<td>1 0</td>
<td>1 0</td>
<td>1 0</td>
<td>0 1</td>
</tr>
</tbody>
</table>

Unate covering again
Fault coverage

- A test of all possible inputs for a NAND\(n\) gate will require \(2^n\) tests
- Applying all possible tests (or sequences of tests) for complex circuits isn’t practical
- However, for all NAND\(n\) gates, all single stuck-at faults (and implicitly multiple stuck-at faults) can be tested using only \(n + 1\) tests
Fault equivalence

- The function of the circuit is identical in the presence of either fault.
- E.g., any input of a NAND gate being s-a-0 or the output being s-a-1 → output of 1.
- In general, identifying equivalent faults, or fault collapsing is difficult.
- However, can use knowledge about gates, and connectivities, to find most equivalent faults.
### Fault equivalence

Consider a NAND3 gate

<table>
<thead>
<tr>
<th>Inputs</th>
<th>fault free</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Consider a NAND3 gate

<table>
<thead>
<tr>
<th>Inputs</th>
<th>fault free</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fault equivalence

Consider a NAND3 gate

<table>
<thead>
<tr>
<th>Inputs</th>
<th>fault free</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</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>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Equivalent faults

<table>
<thead>
<tr>
<th>Inputs</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Robert Dick

Advanced Digital Logic Design
Other fault models

- ~2/3 CMOS faults are not stuck-at faults
  - Luckily, stuck-at fault tests often detect them
- More advanced faults models
  - Allow higher theoretical coverage
  - Are more complicated
- In this lecture, we only have time to cover stuck-at-faults
Section outline

1. Combinational testing
   Yield
   Fault models
   Combinational test generation
Test generation

- Can use algorithms to automatically generate a test for a specific fault
- Problem: Identify some test, \( t \), such that the output of the circuit will deviate from its specified value if a particular fault, \( f \), is present
- Excitation: Need to propagate a value to stuck-at fault that is contrary to the fault value
- Sensitization: Need to propagate the faulty value to an output of the circuit
Excitation and sensitization

s−a−0

\[ \text{Excitation and sensitization} \]

\[ s\rightarrow a\rightarrow 0 \]

\[ \text{sensitization} \]

\[ s\rightarrow a\rightarrow 0 \]

\[ \text{sensitization} \]

\[ s\rightarrow a\rightarrow 0 \]
Excitation and sensitization
Excitation and sensitization

\[ s-a-0 \]

sensitization
Excitation and sensitization
Excitation and sensitization

s–a–0

sensitization

1 1 1
0 0 1

0
Backtracking

not sensitized

s—a—0

1 1 1
1 1
1 1
0 0
1 1
Backtracking

Combinational testing
Sequential testing
Yield
Fault models
Combinational test generation

s-a-0

1 1 1
1 1

not sensitized
0 1 0

sensitized
0 1 1

Robert Dick
Advanced Digital Logic Design
Backtracking

---

sensitized
Backtracking

s-a-0

sensitized
Automatic Test Pattern Generation (ATPG)

- Can automatically determine test for a particular fault
- Boolean difference
  - Generally considered too slow for use on large circuits
  - Easy to understand
- Excitation/sensitization based methods (D-algorithm)
Boolean difference

\[ i_1 \oplus i_2 = z \]

\[ i_3 \]

In the presence of the fault, function is XOR faulty and specified functions
In the presence of the fault, function is $\overline{i_3}$
Boolean difference

XOR faulty and specified functions

\( \delta \)
Boolean difference

- If the resulting function is 0, the fault can not be identified
- If the resulting function is 1, the fault can be identified
  - Use a test that satisfies (sets to 1) the function
Boolean difference

\[
\begin{array}{cccc}
00 & 01 & 11 & 10 \\
0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
00 & 01 & 11 & 10 \\
0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 \\
\end{array}
\]
Boolean difference

\[
\begin{array}{cccc}
00 & 01 & 11 & 10 \\
0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0
\end{array}
\]

\[
\begin{array}{cccc}
00 & 01 & 11 & 10 \\
0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0
\end{array}
\]
Boolean difference

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>00</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>00</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>00</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
Boolean difference

\[
\begin{array}{c|c|c|c|c}
00 & 01 & 11 & 10 \\
\hline
0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
00 & 01 & 11 & 10 \\
\hline
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 \\
\end{array}
\]
**D-algorithm**

- We have seen that, in order to test for a particular fault, we must
  - Excite the fault
  - Sensitize a path from the fault to the output
- The $D$-algorithm is an automated method of finding a test that excites a fault and sensitizes a path to the circuit output
- J.P. Roth, IBM, 1966
Roth’s extension to Boolean algebra

- $\alpha/\beta$
  - $\alpha$ is the fault-free value
  - $\beta$ is the value in the presence of the fault
- $0/0 = 0$
- $1/1 = 1$
- $1/0 = \mathcal{D}$
- $0/1 = \overline{\mathcal{D}}$
- $X/X = X$
## (N)AND \( \mathcal{D} \)-cubes of failure

<table>
<thead>
<tr>
<th>gate</th>
<th>detection input</th>
<th>( \mathcal{D} )-cube of failure</th>
<th>( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( a )</td>
<td>( b )</td>
<td>( z )</td>
</tr>
<tr>
<td>AND</td>
<td>1</td>
<td>1</td>
<td>s-a-0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>( X )</td>
<td>s-a-1</td>
</tr>
<tr>
<td></td>
<td>( X )</td>
<td>0</td>
<td>s-a-1</td>
</tr>
<tr>
<td>NAND</td>
<td>1</td>
<td>1</td>
<td>s-a-1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>( X )</td>
<td>s-a-0</td>
</tr>
<tr>
<td></td>
<td>( X )</td>
<td>0</td>
<td>s-a-0</td>
</tr>
</tbody>
</table>
## (N)OR $D$-cubes of failure

<table>
<thead>
<tr>
<th>gate</th>
<th>detection input</th>
<th>$D$-cube of failure $z$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a$</td>
<td>$b$</td>
<td>$a$</td>
</tr>
<tr>
<td>OR</td>
<td>0</td>
<td>0</td>
<td>s-a-1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>X</td>
<td>s-a-0</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>1</td>
<td>s-a-0</td>
</tr>
<tr>
<td>NOR</td>
<td>0</td>
<td>0</td>
<td>s-a-0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>X</td>
<td>s-a-1</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>1</td>
<td>s-a-1</td>
</tr>
</tbody>
</table>
(N)AND $\mathcal{D}$-cubes of propagation

<table>
<thead>
<tr>
<th>input</th>
<th>AND</th>
<th>NAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>z</td>
</tr>
<tr>
<td>1</td>
<td>$\mathcal{D}$</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td>1</td>
<td>$\overline{D}$</td>
<td>$\overline{D}$</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>1</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td>$\overline{D}$</td>
<td>1</td>
<td>$\overline{D}$</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>$\mathcal{D}$</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td>$\overline{D}$</td>
<td>$\overline{D}$</td>
<td>$\overline{D}$</td>
</tr>
</tbody>
</table>
(N)OR $D$-cubes of propagation

<table>
<thead>
<tr>
<th>input</th>
<th>OR</th>
<th>NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>z</td>
</tr>
<tr>
<td>0</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>0</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

Robert Dick
Advanced Digital Logic Design
$D$-algorithm

1. Initially, all the lines in the circuit are $X$ *(DON’T CARE)*
   - Left blank in example

2. A gate with an input of $D$ or $\overline{D}$ and an output of $X$ belongs to the *frontier*

3. An element whose assigned inputs do not imply the output is *unjustified*

4. We want to drive the frontier to a circuit output (sensitize)...

5. ...and justify the gate inputs (excite)
$D$-algorithm overview

1. Choose $d$-cube of failure to excite fault
2. Sensitize path to output
3. Justify gates
4. Conflicts?
   - Yes: Backtrack
   - No: Done
Select a $D$-cube of failure to excite the fault

while frontier has not yet reached an output do
  Perform the implications of the most recent assignment
  if the frontier is empty then
    BACKTRACK
  end if
  Choose a signal, $s$, s.t. $s$ can’t reached from the fault
  Assign $s$ to a value in order to propagate the fault forward
end while

for each unjustified line, $l$ do
  if $l$ can be justified then
    Justify $l$
  else
    BACKTRACK
  end if
end for
A test has been found
There are a few details not explained in the pseudocode

- Signals should be selected in order to drive the frontier forward
- Backtracking is the action of backing up and taking the closest previously unexplored branch in the decision tree
$D$-algorithm example
D-algorithm example
\( D \)-algorithm example
D-algorithm example
$D$-algorithm example

\[ \begin{array}{c}
\overline{D} \\
\overline{Z} \\
\overline{i_5} \\
\overline{i_4} \\
\overline{i_3} \\
\overline{i_2} \\
\overline{i_1}
\end{array} \]
$D$-algorithm example

\[ \begin{aligned}
\bar{D} & \quad \bar{D} \\
i_1 & \quad 0 \\
i_2 & \quad \cdot \\
i_3 & \quad \cdot \\
i_4 & \quad \cdot \\
i_5 & \quad 1
\end{aligned} \]

$z$
$D$-algorithm example

\[ D \]

\[ D_i^1 \quad D_i^2 \quad D_i^3 \quad D_i^4 \quad D_i^5 \]

\[ 0 \quad 1 \]

\[ i_1 \quad i_2 \quad i_3 \quad i_4 \quad i_5 \]

\[ z \]

\[ \overline{D} \]
$D$-algorithm example

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]

\[ D \]
$D$-algorithm example

\[ D \]

\[ i_1 \ 0 \]
\[ i_2 \]
\[ i_3 \]
\[ i_4 \ 1 \]
\[ i_5 \ 1 \]

\[ D \]

\[ \overline{D} \]

\[ z \]
$D$-algorithm example
$D$-algorithm example

\begin{figure}
\centering
\includegraphics[width=\textwidth]{d_algorithm_example.png}
\end{figure}
$D$-algorithm example

```
\begin{array}{c}
i_1 & 0 \\
i_2 \\
i_3 \\
i_4 & 1 \\
i_5 & 1 \\
\end{array}
```

```
\begin{array}{c}
\text{AND} & \text{OR} & \text{AND} & \text{OR} & \text{AND} \\
\text{AND} & \text{OR} & \text{AND} & \text{OR} & \text{AND} \\
\text{AND} & \text{OR} & \text{AND} & \text{OR} & \text{AND} \\
\text{AND} & \text{OR} & \text{AND} & \text{OR} & \text{AND} \\
\text{AND} & \text{OR} & \text{AND} & \text{OR} & \text{AND} \\
\end{array}
```

\[ D \rightarrow z \]
$D$-algorithm example

\begin{figure}[h]
\centering
\begin{circuitikz}
\draw (0,0) node_AND [name=i1]{$i_1$} ++(0,1) node_NOT [name=i1'] {$\bar{i}_1$};
\draw (i1.east) ++(1,0) node_AND [name=i2]{$i_2$} ++(0,1) node_NOT [name=i2'] {$\bar{i}_2$};
\draw (i2.east) ++(1,0) node_AND [name=i3]{$i_3$} ++(0,1) node_NOT [name=i3'] {$\bar{i}_3$};
\draw (i3.east) ++(1,0) node_AND [name=i4]{$i_4$} ++(0,1) node_NOT [name=i4'] {$\bar{i}_4$};
\draw (i4.east) ++(1,0) node_AND [name=i5]{$i_5$} ++(0,1) node_NOT [name=i5'] {$\bar{i}_5$};
\draw (i5.east) ++(1,0) node_AND [name=Z]{$z$} ++(0,1) node_NOT [name=Z'] {$\bar{z}$};
\draw (Z.east) ++(1,0) node_XOR [name=D]{$D$} ++(0,1) node_NOT [name=D'] {$\bar{D}$};
\draw (D.east) ++(1,0) node_D [name=ZD]{$d$} ++(0,1) node_NOT [name=ZD'] {$\bar{d}$};
\draw (ZD.east) ++(1,0) node_AND [name=ZD'] {$\bar{D}$};
\end{circuitikz}
\end{figure}
$D$-algorithm example

\[ i_1 = 0, \quad i_2, \quad i_3, \quad i_4, \quad i_5 = 1 \]

\[ D = 1 \rightarrow \overline{D} = 0 \]

\[ D = 0 \rightarrow \overline{D} = 1 \]

\[ z = \overline{D} \]
$D$-algorithm example

Conflict!

0 1 0 1 0

$D$ 0 $z$

1 $\overline{D}$ $\overline{D}$

Robert Dick  Advanced Digital Logic Design
$D$-algorithm example

- $i_1 = 0$
- $i_2$
- $i_3$
- $i_4$
- $i_5 = 1$

Output $D$ and $z$
$D$-algorithm example
$D$-algorithm example

![Diagram of a digital logic circuit](image-url)
D-algorithm example

\[ D \]

\begin{align*}
 i_1 & \rightarrow \quad D \\
 i_2 & \rightarrow \\
 i_3 & \rightarrow \\
 i_4 & \rightarrow \\
 i_5 & \rightarrow \\
 \end{align*}

\[ \overline{D} \rightarrow Z \]
D-algorithm example
D-algorithm example
D-algorithm example
$D$-algorithm example
$D$-algorithm example

\[ D \]

\[ 1 \]

\[ 0 \]

\[ i_1 \]

\[ i_2 \]

\[ i_3 \]

\[ i_4 \]

\[ i_5 \]

\[ z \]

\[ D \]

\[ \overline{D} \]

\[ 1 \]
\( \mathcal{D} \)-algorithm example
\( D \)-algorithm example

\[
\begin{align*}
\cdot \quad & i_1 \quad 1 \\
\cdot \quad & i_2 \\
\cdot \quad & i_3 \\
\cdot \quad & i_4 \\
\cdot \quad & i_5 \quad 0 \\
\end{align*}
\]

\[
\begin{align*}
\cdot \quad & \overline{D} \\
\cdot \quad & D \\
\end{align*}
\]

\[
\begin{align*}
\cdot \quad & 1 \\
\cdot \quad & \overline{D} \\
\cdot \quad & \\
\cdot \quad & Z \\
\end{align*}
\]
**D-algorithm example**

The diagram shows a logic circuit with inputs $i_1$, $i_2$, $i_3$, $i_4$, and $i_5$. The outputs are $1$ and $0$ for $i_1$, $i_2$, and $i_3$, respectively. The circuit includes AND, OR, and NOT gates, with the final output $z$ and the inverters $\bar{D}$ and $\overline{\bar{D}}$.
$D$-algorithm example

\[ \begin{array}{c}
\cdot i_1 & 1 \\
\cdot i_2 \\
\cdot i_3 \\
\cdot i_4 \\
\cdot i_5 & 0 \\
\cdot z & 0 \\
\end{array} \]
$\mathcal{D}$-algorithm example

$\mathcal{D}$-algorithm example
$D$-algorithm example

$D$-algorithm

\[ \begin{array}{lllll}
i_1 & 1 & 1 & 1 & 1 \\
i_2 & 1 & 0 & 0 & 0 \\
i_3 & 1 & 0 & 0 & 0 \\
i_4 & 1 & 1 & 0 & 0 \\
i_5 & 0 & 0 & 0 & 0 \\
\end{array} \]
$D$-algorithm example

\[ i_1 \quad 1 \]
\[ i_2 \quad 1 \]
\[ i_3 \quad 0 \]
\[ i_4 \quad 1 \]
\[ i_5 \quad 0 \]

\[ D \quad 0 \]
\[ \overline{D} \quad \overline{D} \]
\[ z \]
$D$-algorithm example

\[
\begin{array}{c}
\begin{array}{c}
 i_1 \\
 i_2 \\
 i_3 \\
 i_4 \\
 i_5 \\
\end{array}
\end{array}
\begin{array}{c}
 1 \\
 1 \\
 0 \\
 0 \\
 0 \\
\end{array}
\begin{array}{c}
\begin{array}{c}
 1 \\
 0 \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
 0 \\
 0 \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
 D \\
 z \\
 D \\
\end{array}
\end{array}
\end{array}
\]
$D$-algorithm example

![D-algorithm example](image)
$D$-algorithm example

The diagram shows a combinational logic circuit with inputs $i_1, i_2, i_3, i_4, i_5$ and output $z$. The circuit includes AND and OR gates with specified input values. The circuit is designed to illustrate the $D$-algorithm example in the context of combinational test generation.
## NOR $\mathcal{D}$-cubes of failure

<table>
<thead>
<tr>
<th>gate</th>
<th>detection input</th>
<th>$\mathcal{D}$-cube of failure $z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOR</td>
<td>a b</td>
<td>a b</td>
</tr>
<tr>
<td>0 0</td>
<td>s-a-0</td>
<td>0 0</td>
</tr>
</tbody>
</table>
## NOR $\mathcal{D}$-cubes of failure

<table>
<thead>
<tr>
<th>gate</th>
<th>detection input</th>
<th>$\mathcal{D}$-cube of failure</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>NOR</td>
<td>0</td>
<td>0</td>
<td>s-a-0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>s-a-1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>s-a-1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>s-a-1</td>
</tr>
</tbody>
</table>
AND $\mathcal{D}$-cubes of propagation

<table>
<thead>
<tr>
<th>input</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td>1</td>
<td>$\bar{\mathcal{D}}$</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>1</td>
</tr>
<tr>
<td>$\bar{\mathcal{D}}$</td>
<td>1</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td>$\bar{\mathcal{D}}$</td>
<td>$\bar{\mathcal{D}}$</td>
</tr>
</tbody>
</table>
### AND $D$-cubes of propagation

<table>
<thead>
<tr>
<th>input</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>$D$</td>
</tr>
<tr>
<td>1</td>
<td>$D$</td>
</tr>
<tr>
<td>$D$</td>
<td>1</td>
</tr>
<tr>
<td>$\overline{D}$</td>
<td>1</td>
</tr>
<tr>
<td>$D$</td>
<td>$D$</td>
</tr>
<tr>
<td>$\overline{D}$</td>
<td>$\overline{D}$</td>
</tr>
</tbody>
</table>
## NOR $\mathcal{D}$-cubes of propagation

<table>
<thead>
<tr>
<th>input</th>
<th>NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>0</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td>0</td>
<td>$\overline{\mathcal{D}}$</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>0</td>
</tr>
<tr>
<td>$\overline{\mathcal{D}}$</td>
<td>0</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td>$\overline{\mathcal{D}}$</td>
<td>$\overline{\mathcal{D}}$</td>
</tr>
</tbody>
</table>
### NOR $\mathcal{D}$-cubes of propagation

<table>
<thead>
<tr>
<th>input</th>
<th>NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>0</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td>0</td>
<td>$\overline{\mathcal{D}}$</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>0</td>
</tr>
<tr>
<td>$\overline{\mathcal{D}}$</td>
<td>0</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td>$\overline{\mathcal{D}}$</td>
<td>$\overline{\mathcal{D}}$</td>
</tr>
</tbody>
</table>
NOR $\mathcal{D}$-cubes of failure

<table>
<thead>
<tr>
<th>gate</th>
<th>detection input</th>
<th>$\mathcal{D}$-cube of failure</th>
<th>z</th>
<th>a</th>
<th>b</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOR</td>
<td>a b</td>
<td>s-a-0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 0</td>
<td>s-a-1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>s-a-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>s-a-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
## NOR $\mathcal{D}$-cubes of failure

<table>
<thead>
<tr>
<th>gate</th>
<th>detection input</th>
<th>$\mathcal{D}$-cube of failure z</th>
<th>a</th>
<th>b</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOR</td>
<td>0 0</td>
<td>s-a-0</td>
<td>0</td>
<td>0</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>s-a-1</td>
<td>0</td>
<td>1</td>
<td>$\overline{\mathcal{D}}$</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>s-a-1</td>
<td>1</td>
<td>0</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>s-a-1</td>
<td>1</td>
<td>1</td>
<td>$\overline{\mathcal{D}}$</td>
</tr>
</tbody>
</table>
**D-algorithm summary**

- If a test for a fault exists, guaranteed to generate it
  - Full backtracking
- However, much faster than Boolean difference
Caveats

- This lecture has only introduced some topics in testing
  - If you intend to do research in this area, take a few courses on testing
- Single stuck-at fault model is becoming obsolete
Caveats

- $D$-algorithm is actually more complicated
  - Avoiding backtracking is important for performance
    - Implications
    - Heuristics to make best decisions first
  - Should understand derivation of $D$-cubes for arbitrary fault models

- This is one of the better references in the library
- Beware: The $D$-algorithm examples contain small errors

- Recommended by a student
- Introduces a number of important logic-level and physical-level CAD algorithms
- A useful supplement to the material in Hachtel and Somenzi’s, and Sherwani’s books
- Introduces the $D$-algorithm
Outline

1. Combinational testing

2. Sequential testing
Section outline

2. Sequential testing
   \( D \)-algorithm review
   Redundancy elimination
   Sequential testing
   Design/synthesis for testability
**D-algorithm review**

- Given a fault, the $D$-algorithm will generate a test for that fault.
- Selects a $D$-cube of failure to excite the fault.
- Propagates a $D$ value to the circuit outputs by selecting $D$-cubes of propagation for gates.
- Justifies gate inputs.
- Uses full backtracking on $D$-cubes of failure and gate justification.
Fault simulator

- Fault simulation is the process of determining which faults a test detects
- Test generation is complicated and time-consuming
- Fault simulation is quick
Test sequence generation

The $D$-algorithm gives us a way to generate a test for a given fault. However, we need a sequence of tests for all (or most) faults.

Determine all stuck-at faults for the circuit
Eliminate equivalent faults

**while** untested faults remain **do**
- Pick a fault and use an ATPG to generate a test
- Append the test to a list
- Use a fault simulator to determine all faults the test detects
- Eliminate detected faults

**end while**
Test sequence generation

- However, this approach can still be too expensive
- Recall
  - Fault simulation is fast
  - ATPG is slow
- Take advantage of fault simulator
Test sequence generation

\[\textbf{while} \text{ fault coverage is less than } 90\% \textbf{ do} \]
Generate a random pattern – extremely fast
Use a fault simulator to determine if new faults are detected
\[\textbf{if} \text{ so } \textbf{then} \]
Append pattern to a list
Eliminate detected faults
\[\textbf{end if} \]
\[\textbf{end while} \]

\[\textbf{while} \text{ fault coverage is less than } 99.5\% \textbf{ do} \]
Pick a fault and use an ATPG to generate a test
Append the test to a list
Use a fault simulator to determine all faults the test detects
Eliminate detected faults
\[\textbf{end while} \]
Section outline

2. Sequential testing
   \( D \)-algorithm review
   Redundancy elimination
   Sequential testing
   Design/synthesis for testability
Redundancies

- Sometimes a s-at-0 fault can’t be detected
  - Implication: Given any set of inputs to the circuit, that node can be set to 0 and the output will match the specifications
- Converse for s-a-1 faults
- Even if a portion of the circuit is not able the switch, the same function is realized
- The presence of an undetectable fault implies redundancy
Redundancies

- Sometimes a s-at-0 fault can’t be detected
  - Implication: Given any set of inputs to the circuit, that node can be set to 0 and the output will match the specifications

- Converse for s-a-1 faults
- Even if a portion of the circuit is not able the switch, the same function is realized
- The presence of an undetectable fault implies redundancy
- How can the $D$-algorithm can be used to simplify logic?
Redundancies

Logic terminating in the location of the stuck-at fault can be removed from the circuit and the same function will be realized.

- Conventional wisdom suggests removal
  - Simplify logic
  - Make circuit more fully testable
Redundancy example

![Diagram of a digital logic circuit with inputs a and b and output z, illustrating redundancy elimination.](image)
Redundancy example

\[ a \quad s-a-0 \quad b \quad z \]
Redundancy example

Excite: \( a \neq b \)
Redundancy example

Excite: \(a \neq b\)
Sensitize: \(a = b = 0\)
Redundancy example

Excite: $a \neq b$
Sensitize: $a = b = 0$
Can’t excite and propagate faulty value to $z$!
Redundancy example

Excite:  $a \neq b$
Sensitize:  $a = b = 0$
Can’t excite and propagate faulty value to $z$!
Replace s-a-0 logic with 0
Redundancy example

Excite: \( a \neq b \)
Sensitize: \( a = b = 0 \)
Can’t excite and propagate faulty value to \( z \)!
Replace s-a-0 logic with 0
Eliminate unused logic
Redundancy example

Excite: \( a \neq b \)
Sensitize: \( a = b = 0 \)
Can’t excite and propagate faulty value to \( z \)!
Replace s-a-0 logic with 0
Eliminate unused logic
Simplify logic
ATPG for logic simplification

Can use ATPG algorithm to simplify logic

1. Find a fault that is untestable due to redundancy
2. Remove the redundant logic
3. Repeat
Redundancy removal not always safe

- Logical equivalence may not imply true equivalence
- What happens if redundancy is intentional?
  - To reduce power consumption
  - To ensure signal stability for use in asynchronous circuits
- Removal can
  - Increase power
  - Introduce races
Section outline

2. **Sequential testing**
   - $D$-algorithm review
   - Redundancy elimination
   - Sequential testing
   - Design/synthesis for testability
Sequential testing

- Can convert to a version of combinational testing
  - Iterative array expansion
- Simulation-based
Iterative array expansion

- Make a copy of the combinational logic for each time step
- However, now the problem converts ATPG for multiple faults
- Iterative array expansion increases the size of the combinational problem
- More importantly, changes the required fault model
  - Requires ATPG for multiple faults
Iterative array expansion

Diagrams showing logic circuits with inputs $i_1, i_2, i_3, i_4$ and output $z$.
Iterative array expansion

\[ i_1 \quad i_2 \quad i_3 \quad i_4 \]

\[ D \quad Q \]

\[ s-a-0 \]

\[ z \]

\[ c \]
Iterative array expansion

\[ i_1, i_2, i_3, i_4 \rightarrow z \]

\[ D \text{-algorithm review} \]

\[ \text{Redundancy elimination} \]

\[ \text{Sequential testing} \]

\[ \text{Design/synthesis for testability} \]
Iterative array expansion

\[ D \text{-algorithm review} \]
\[ \text{Redundancy elimination} \]
\[ \text{Sequential testing} \]
\[ \text{Design/synthesis for testability} \]

\[ \text{Combinational testing} \]
\[ \text{Sequential testing} \]
Iterative array expansion

\[ \text{Diagram with iterative array expansion logic gates and inputs and outputs} \]
Modified $D$-algorithm for iterative array

- Propagate $D$-frontier forward
  - Generate new time frames as necessary
- Propagate gate justification backward
  - Generate new time frames as necessary
- However, this algorithm is not certain to find a test if one exists
  - Roth’s algebra not suited to sequential circuits
Muth’s 9-valued logic

- $\mathcal{D}$-algorithm can’t adequately model the possible states in sequential circuits
  - Repeated effect of fault can’t be modeled
- Roth’s values
  - $0/0$, $0/1$ ($\overline{\mathcal{D}}$), $1/0$ ($\mathcal{D}$), $1/1$, $X/X$
- Muth’s values
  - $0/0$, $0/1$, $0/X$, $1/0$, $1/1$, $1/X$, $X/0$, $X/1$, $X/X$
Simulation-based sequential testing

- Generate a vector
- Determine whether that vector brought the circuit state nearer to or farther from fault excitation and sensitization
- If the state moved near to detection, append the pattern to a list
- Many of the probabilistic optimization algorithms in the third lecture can be applied for simulation-based testing
Simulation-based sequential testing
Simulation-based sequential testing

![Logic diagram with inputs a, b, c and output s-a-1]
Simulation-based sequential testing

What if \( a, b, \) and \( c \) are state variables?
Simulation-based sequential testing

What if $a$, $b$, and $c$ are state variables?
Simulation-based sequential testing

What if $a$, $b$, and $c$ are state variables?
Simulation-based sequential testing

What if $a$, $b$, and $c$ are state variables?
Sequential test generation problems

- Sequential test generation intractable for large circuits
- Some continue to work on improving test generation
- Others modify circuit structure to make test generation easier...
2. Sequential testing
   $D$-algorithm review
   Redundancy elimination
   Sequential testing
   Design/synthesis for testability
Scan-based design for test (DFT)

- If testability is considered during design or synthesis, can be made easy
- Instead of forcing ATPG to deal with synchronous testing, insert a scan chain
- Chain together (all) flip-flops and scan through an input if and only if a testing input is activated
Scan D flip-flop
Scan D flip-flop
Scan D flip-flop

![Scan D flip-flop diagram]
Scan D flip-flop

Diagram showing a scan D flip-flop circuit with inputs labeled $D$ and $T$, connecting from previous flip-flop to next flip-flop.
Scan D flip-flop

testing logic overhead
from previous flip-flop
to next flip-flop
Scan D flip-flop

designating logic overhead

designating routing overhead

designating logic overhead

designating routing overhead

designating routing overhead

designating routing overhead

designating routing overhead

designating routing overhead

designating routing overhead

designating routing overhead

designating routing overhead

designating routing overhead

from previous flip-flop

to next flip-flop

D-algorithm review
Redundancy elimination
Sequential testing
Design/synthesis for testability

Combinational testing
Sequential testing

Robert Dick
Advanced Digital Logic Design
Full scan

- Greatly simplifies testing
  - Converts sequential testing to combinational testing
- Testing can be slow if I/O pins limited
- Significantly increases chip area and price
  - More complicated flip-flops
  - More complicated routing
  - I/O requirements, especially if speed required
Scan chain

D-algorithm review
Redundancy elimination
Sequential testing
Design/synthesis for testability

Combinational testing
Sequential testing

DQCDTS QCDTS QCDTS QCDTS

Combinational logic

Serial scan input
Test mode
Clock

Serial scan output

73 Robert Dick Advanced Digital Logic Design
Level-sensitive scan design (LSSD)

- Less delay than scan-path during normal operation
- Used commercially, especially within IBM
- Pre-processing step replaces all latches with LSSD latches
- Other companies have related testing approaches
  - Scan path – NEC
  - Scan/set – Sperry Univac
  - Random access scan – Fujitsu
LSSD

Combinational testing
Sequential testing

D-algorithm review
Redundancy elimination
Sequential testing
Design/synthesis for testability

\[ D \]
\[ C_{\text{normal}} \]
\[ C_{\text{test1}} \]
\[ C_{\text{test2}} \]
\[ T \]

\[ L_1 \]
\[ L_2 \]
\[ \overline{L}_2 \]
LSSD

level sensitive D flip-flop

$D$

$C_{normal}$

$C_{test1}$

$T$

$C_{test2}$

$L_1$

$L_1$

$L_2$

$\overline{L_2}$

$L_1/L_1$

$D$-algorithm review
Redundancy elimination
Sequential testing
Design/synthesis for testability
LSSD

level sensitive D flip-flop

L1

L1

L2

L2

level sensitive D flip-flop

D

C_{normal}

C_{test 1}

C_{test 2}

T
LSSD

level sensitive D flip-flop

mode control

$D$

$L_1$

$L_1$

$L_2$

$T$

$L_2$

$c_{\text{normal}}$

$c_{\text{test1}}$

$c_{\text{test2}}$
LSSD

level sensitive D flip-flop

$D$

$C_{\text{normal}}$

$C_{\text{test1}}$

$C_{\text{test2}}$

mode control

level sensitive D flip-flop

$L_1$

$L_1$

$L_2$

$\bar{L}_2$
Full scan disadvantages

- Increased area
  - 5%-15%
- Increased delay
  - Depends on critical path average combinational logic depth
- Slow when implemented serially
  - Need to serially clock through every register in IC
- High pin requirements to speed up
Partial scan

- Instead of scanning all latches, scan a subset
- Need to carefully select scanned latches based on
  - Test coverage effects
  - Testing speed
Advanced DFT/SFT techniques

- Testing core-based systems-on-chip (SoC)
  - Related to board-level boundary-scan
- Fault injection
  - Reliability evaluation
  - Insert fault into system and determine reaction
- RTL synthesis for testability
- Software fault modeling and testing
  - Extremely difficult problem
Personal observations on testing

- Despite continued research on advanced testing techniques, industry continues to use full-scan
  - In order for industry to accept a more complicated testing technique, the advantages must be tremendous
  - Companies don’t like complexity
- Testing is a fairly mature field
- Some people in academia have shifted their interest away from testing
  - It remains a huge practical problem for industry
Personal observations on testing

- Companies don’t want increased complexity
- Researchers want to see their ideas used
- Therefore, reduction in interest in testing among researchers
- Therefore, reduction in interest in testing among companies
- However, companies still have huge testing problems
- Therefore, high demand for MS and Ph.D. graduates with testing experience

One option: target a research problems involving testing and another area, e.g., synthesis
Testing summary

- Low defect rate requires high test coverage
- Functional testing requires too many patterns
- Use structural testing based on a fault model, e.g., single stuck-at
- Can use fault equivalence to reduce patterns
- Can automatically generate a test for a specific fault
- Combine ATPG with fault simulator to generate a set of tests with high coverage
Testing summary

- Can use ATPG for logic simplification – redundancy
- Sequential testing difficult for large circuits
- Use DFT to make testing easier
- Use design automation to make DFT easier
Sequential testing references

  - Survey on high-level design/synthesis for test techniques
  - Survey on sequential test generation