Fault Modeling

- Suppose that a given design has been fully verified. However, some of the chips may not be functional due to defects that occur during the manufacturing process. Such defects are known as *faults*.
- Faults may be modeled at different levels:
  - A physical fault is the actual defect that occurs.
    - Examples: poor gate oxide, high resistance contacts, shorted or open metal lines, etc.
    - However, there are so many possibilities that it is not feasible to try to analyze them at this level of detail.
  - A logical fault is the effect of a physical fault on the logical operation of the chip.
    - By working at a higher level of abstraction, it becomes possible to fully analyze a small number of possible effects, thereby making the problem tractable.
- The most common logical fault model is based on *stuck-at faults*:
  - We assume that a physical fault manifests itself by having an input or output of a logic gate permanently stuck at either logic-0 or logic 1.
  - It turns out that most physical faults can be modeled in this fashion.

© 2006 by Gerald E. Sobelman

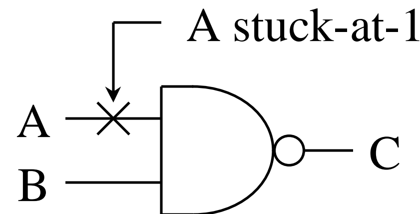
Stimulus/Response Testing

- In testing, we apply stimulus to the chip and observe its response:
  - We apply a set of values to the chip's input pins (called the primary inputs, or PIs). Such a set of values is called an *input vector*.
  - We observe a set of values a the chip's output pins (called the primary outputs, or POs). Such a set of values is called an *output vector*.
  - The set of all input vectors to be applied is called a *test set* or a *test sequence*.
  - A test set together with the corresponding output vectors is called the *test data*.
- A major simplification that is made is called the single stuck fault model: We assume that one fault occurs at a time on a chip. This drastically reduces the complexity of the testing problem. Suppose that, considering all of the logic gates on the chip, there are a total of k input and output ports of these logic gates.
  - With the single fault simplification there are a total of 2k single stuck-at faults to consider (i.e., each input or output port stuck-at-0 or stuck-at-1).
  - On the other hand, if we allowed all possible combinations of single and multiple faults, then there are $3^k - 1$ fault situations to consider (i.e., each line is either good, stuck-at-0 or stuck-at-1, and we exclude the one case in which all of the lines are good.)
- Fortunately, testing for single stuck faults will also detect most of the multiple faults.

© 2006 by Gerald E. Sobelman

Single Stuck Fault Testing Example

- As an extreme simplification, suppose that our chip consists of a single 2-input NAND gate. Suppose that we wanted to find a test vector for the fault A stuck-at-1.
- There are two basic rules to follow:
  - For the line with the stuck-at fault, we have to apply the value opposite to the fault (otherwise, there would be no way to tell if the circuit is faulty or not). Hence, in this example, we must assign A = 0.
  - For the other input, we must apply the value that will allow the faulty value to be propagated through the gate. Here, we must choose B = 1 (otherwise, B = 0 forces the output to 1 regardless of the condition of the A input).
- Thus, applying the test vector (A, B) = (0, 1) allows us to distinguish a good chip from a faulty chip:
  - In the case of the good chip: C = 1
  - In the case where input A is stuck-at-1: C = 0
  - Since these two output values are *different*, this is a valid test vector for this fault.



© 2006 by Gerald E. Sobelman

Fault Collapsing

- Consider a combinational logic function $f(\mathbf{x}) = f(x_1, x_2, \ldots, x_n)$. A given stuck-fault α causes f to change to some other function $f_\alpha(x_1, x_2, \ldots, x_n)$.
- An input vector **x** detects the stuck-fault α if $f(\mathbf{x})$ and $f_\alpha(\mathbf{x})$ are different, i.e. either:
  (i) $f(\mathbf{x}) = 0$ and $f_\alpha(\mathbf{x}) = 1$
  (ii) $f(\mathbf{x}) = 1$ and $f_\alpha(\mathbf{x}) = 0$
- In other words, if we form the Boolean function $T_\alpha(\mathbf{x}) = f(\mathbf{x}) \oplus f_\alpha(\mathbf{x})$, then the values of **x** for which $T_\alpha(\mathbf{x}) = 1$ are the test vectors for the fault α.
- Now consider two faults, α and β. Then, form: $T_\alpha(\mathbf{x}) = f(\mathbf{x}) \oplus f_\alpha(\mathbf{x})$, $T_\beta(\mathbf{x}) = f(\mathbf{x}) \oplus f_\beta(\mathbf{x})$. If $f_\alpha(\mathbf{x}) = f_\beta(\mathbf{x})$, then $T_\alpha(\mathbf{x}) = T_\beta(\mathbf{x})$ and there is no test that can distinguish between the two faults α and β. In such a case, the α and β are said to be *equivalent faults*.
- This can be extended to a larger set of faults: If a set of faults are all (pairwise) equivalent to each other, then they are said to form a *fault equivalence class*. Any test that detects one of the faults in the equivalence class automatically will detect all of those faults, and no test vector can distinguish between any of the faults in the equivalence class.
- Therefore, in selecting test vectors, one only needs to consider any one fault from the entire class to be a representative for that class. This reduces the number of test vectors that are required for the chip, and this process is known as *fault collapsing*.

© 2006 by Gerald E. Sobelman

Example of Fault Collapsing

- Consider the 2-input AND gate shown below (i.e., $f = x_1 x_2$) and the following 3 faults:

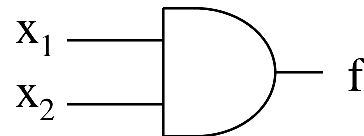
  $\alpha = x_1$ stuck-at-0 , $\beta = x_2$ stuck-at-0 , $\gamma = f$ stuck-at-0
- It is obvious that $f_\alpha = f_\beta = f_\gamma = 0$. Therefore, $\{\alpha , \beta, \gamma\}$ form a fault equivalence class. Choose α as the representative fault. We have:

$$T_a = f \oplus f_a = x_1 x_2 \oplus 0$$

$$T_a = 1 \text{ if and only if } x_1 x_2 = 1$$

$$\Rightarrow x_1 = x_2 = 1$$

- For the input vector $(x_1, x_2) = (1, 1)$, we will obtain different values at the output for the good and faulty circuits:
  - For the good circuit: $f = 1$.
  - For the circuit with fault α: $f = 0$.
- Therefore, the test vector $(x_1, x_2) = (1, 1)$ detects the faults α , β, and γ, and there is no test vector that can distinguish between α , β, and γ.



General Rules for Fault Collapsing

- For an n-input AND gate:
  - If any input is 0, the output will be 0. Therefore, any input stuck-at-0 will be equivalent to any other input stuck-at-0. It will also be equivalent to the output stuck-at-0. Thus, we have a fault equivalence class containing n+1 faults:

    $\{x_1$ stuck-at-0, x_2 stuck-at-0, … , x_n stuck-at-0, f stuck-at-0$\}$
  - In addition, there are the following n+1 additional fault equivalence classes, each containing only a single fault:

    $\{x_1$ stuck-at-1$\}$, $\{x_2$ stuck-at-1$\}$, … , $\{x_n$ stuck-at-1$\}$, $\{$f stuck-at-1$\}$
  - This gives a total of n+2 non-equivalent faults that must be considered.
- The situation is similar for an n-input NAND gate:
  - If any input is 0, the output will be 1. Therefore, any input stuck-at-0 will be equivalent to any other input stuck-at-0. It will also be equivalent to the output stuck-at-1. Thus, we have a fault equivalence class containing n+1 faults:

    $\{x_1$ stuck-at-0, x_2 stuck-at-0, … , x_n stuck-at-0, f stuck-at-1$\}$
  - In addition, there are the following n+1 additional fault equivalence classes, each containing only a single fault:

    $\{x_1$ stuck-at-1$\}$, $\{x_2$ stuck-at-1$\}$, … , $\{x_n$ stuck-at-1$\}$, $\{$f stuck-at-0$\}$
  - Again, we have a total of n+2 non-equivalent faults that must be considered.

© 2006 by Gerald E. Sobelman

General Rules for Fault Collapsing - Continued

- For an n-input OR gate:
  - If any input is 1, the output will be one. Therefore, any input stuck-at-1 will be equivalent to any other input stuck-at-1. It will also be equivalent to the output stuck-at-1. Thus, we have a fault equivalence class containing n+1 faults:

    $\{x_1$ stuck-at-1, x_2 stuck-at-1, … , x_n stuck-at-1, f stuck-at-1$\}$
  - In addition, there are the following n+1 additional fault equivalence classes, each containing only a single fault:

    $\{x_1$ stuck-at-0$\}$, $\{x_2$ stuck-at-0$\}$, … , $\{x_n$ stuck-at-0$\}$, {f stuck-at-0}
  - This gives a total of n+2 non-equivalent faults that must be considered.
- For an n-input NOR gate:
  - If any input is 1, the output will be 0. Therefore, any input stuck-at-1 will be equivalent to any other input stuck-at-1. It will also be equivalent to the output stuck-at-0. Thus, we have a fault equivalence class containing n+1 faults:

    $\{x_1$ stuck-at-1, x_2 stuck-at-1, … , x_n stuck-at-1, f stuck-at-0$\}$
  - In addition, there are the following n+1 additional fault equivalence classes, each containing only a single fault:

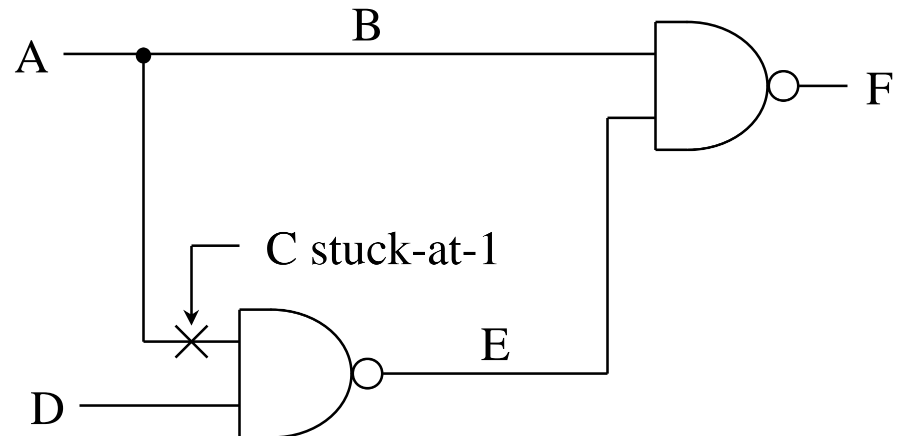
    $\{x_1$ stuck-at-0$\}$, $\{x_2$ stuck-at-0$\}$, … , $\{x_n$ stuck-at-0$\}$, {f stuck-at-1}
  - This gives a total of n+2 non-equivalent faults that must be considered.

© 2006 by Gerald E. Sobelman

Redundant Faults

- A redundant fault is one in which the circuit behaves normally. (They can arise from redundant logic, which may exist for various reasons. For example, certain types of high-speed carry chains contain redundant logic.) Therefore, we can consider redundant faults to be in the same equivalence class as the good circuit. This means that redundant faults cannot be tested, and they should be removed from the list of faults to be considered.
- As an example, consider the circuit shown below. The output F in the case of the good circuit or the case where the fault C stuck-at-1 exists both produce the same output values for all input combinations. Therefore, C stuck-at-1 is a redundant fault.

A	D	F (good circuit)	F (C s-a-1)
0	0	1	1
0	1	1	1
1	0	0	0
1	1	1	1



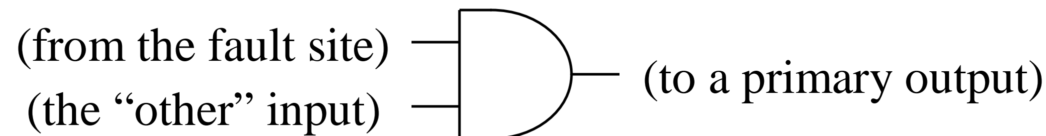
Test Generation

- The basic problem in test generation is to find a set of test vectors that will result in an observable difference (i.e., at one of the primary outputs of the chip) between the fault-free circuit and the various faulty circuits that can exist in the single stuck-fault model.
- Test generation is <u>fault-specific</u>: Given a fault, find a test for that fault.
- Normally, the site of the fault will be at some inaccessible internal node in the chip. We will need to do the following:
  - Find a set of chip input values that causes the site of the fault to take on the logical value that is opposite to that of the fault. This is known a *fault activation*.
  - Find a set of chip input values that allows the faulty value to propagate to one of the output pins where it can be observed.
- For purely combinational circuits, algorithms exist which can successfully construct a test vector for any non-redundant fault.
- For arbitrary sequential circuits, however, the situation is much more complex and no general algorithms exist. The difficulty comes from the fact that we need to propagate values not only through the circuit (i.e., in space) but also through a sequence of states in the registers (i.e., in time). As a result, design-for-testability techniques (e.g., scan design, built-in self-test) are used to simplify the testing problem.
- First, let’s consider some examples of test generation for combinational circuits.

© 2006 by Gerald E. Sobelman

Single Path Sensitization

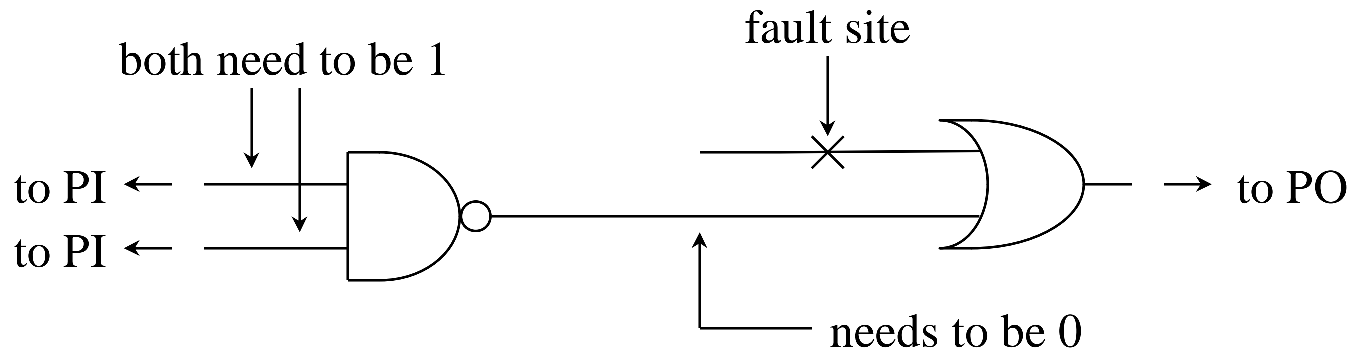
- While not a completely general solution, single path sensitization illustrates the basic concepts involved in test generation. There are three phases:
  - Fault activation: Select value at the site of the fault to be the one opposite to the stuck-at value. For example, for a stuck-at-0 fault, we need to make sure that we produce a logic-1 at that node so that there will be a difference created between the good and faulty circuits.
  - Error propagation (also called forward trace): Create a path from the site of the fault to a primary output where the difference between the good and faulty circuit can be observed. This requires the following:
    - Choose a path through intermediate logic gates to one of the primary outputs.
    - At each logic gate along this path, choose values for the *other* inputs to those logic gates that allows the fault information to propagate through each gate. For example, if the 2-input AND gate below is along this path, the other input to the AND gate must have be at logic-1. (If it were at logic-0, the output of that gate would be 0, and the information from the fault site would be lost.)



© 2006 by Gerald E. Sobelman

Single Path Sensitization - Continued

– Line justification (also called backward trace): Find values at the primary inputs such that all of the intermediate node values required in fault activation and error propagation can be established at those nodes. For example, suppose that we need to propagate a fault through the circuit segment shown below.
  - The other input to the OR gate would have to be 0.
  - This, in turn, means that both of the inputs to the NAND gate driving that node would have to be 1.
  - This line of argument is continued all the way back to the primary inputs.
  - The same must be done for all gates along the path to the PO that was chosen during error propagation as well as for the gates on a path from the fault site back to the primary inputs.



© 2006 by Gerald E. Sobelman

Issues in Single Path Sensitization

- We may or may not be successful in carrying out the 3-step process:
  - For example, the choice of path in the error propagation phase may create a set of values required on other nodes that may not be consistent with each other during the line justification phase.
  - If this occurs, then we should backtrack, meaning that we should undo one or more of our previous choices and select new ones. This, in general, creates an iterative process.
  - Furthermore, in some situations, it may be *impossible* using this technique to find a test for some non-redundant fault. The essential difficulty is that fault effects may propagate along multiple paths and then reconverge at certain logic gates, creating cancellations and other effects that are not properly accounted for by the single path sensitization technique.
  - A complete solution was found by J. P. Roth in 1966 in a procedure known as the *D-Algorithm.* This algorithm properly accounts for multiple path effects, and it is guaranteed to find a test vector for any non-redundant fault in a general combinational logic circuit.
  - Since then, other complete algorithms have been developed (e.g., PODEM, FAN) that may be faster and/or more efficient than the D-Algorithm.

© 2006 by Gerald E. Sobelman

Fault Simulation

- Fault simulation is conceptually the opposite of test generation. It is test-specific: Given a test, find all faults that are detected by it. Remember that a test vector was constructed to expose some specific fault. However, there are two categories of additional faults that may be detected by this same test vector:
  - Trivial faults, i.e. faults in the same equivalence class.
  - Accidental faults, i.e. non-equivalent faults that just happen to be detected by this test vector. (This may sound contradictory, but it is not. Equivalent faults are those that cannot be distinguished by any means. Non-equivalent accidental faults behave differently under some test vectors but just happen to behave the same under the one specific test vector under consideration.)
- Fault simulation is very computationally intensive. For each test vector, we must:
  - Simulate the fault-free circuit
  - Simulate all associated circuits that are obtained by inserting single stuck-at faults, one at a time. (Of course, we only need to consider one representative fault from each equivalence class.
- The above process is repeated for all test vectors in the test set to obtain the *fault coverage* of the test set, i.e. the percentage of all faults that are detected by the test set. Hence, we may need to simulate many thousands of circuits (each of which is large) for each test vector.

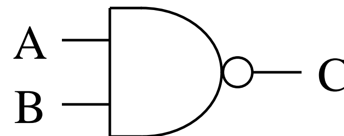
© 2006 by Gerald E. Sobelman

Fault Simulation Example

- Consider fault simulation of a 2-input NAND gate. This has the following 4 fault equivalent classes: {A s-a-0, B s-a-0, C s-a-1}, {A s-a-1}, {B s-a-1}, {C s-a-0}.
- A hand-simulation of the 4 possible test vectors for the good and faulty cases gives:

		C for:				
A	B	good	A s-a-1	Bs-a-1	C s-a-0	C s-a-1
0	0	1	1	1	0	1
0	1	1	0	1	0	1
1	0	1	1	0	0	1
1	1	0	0	0	0	1

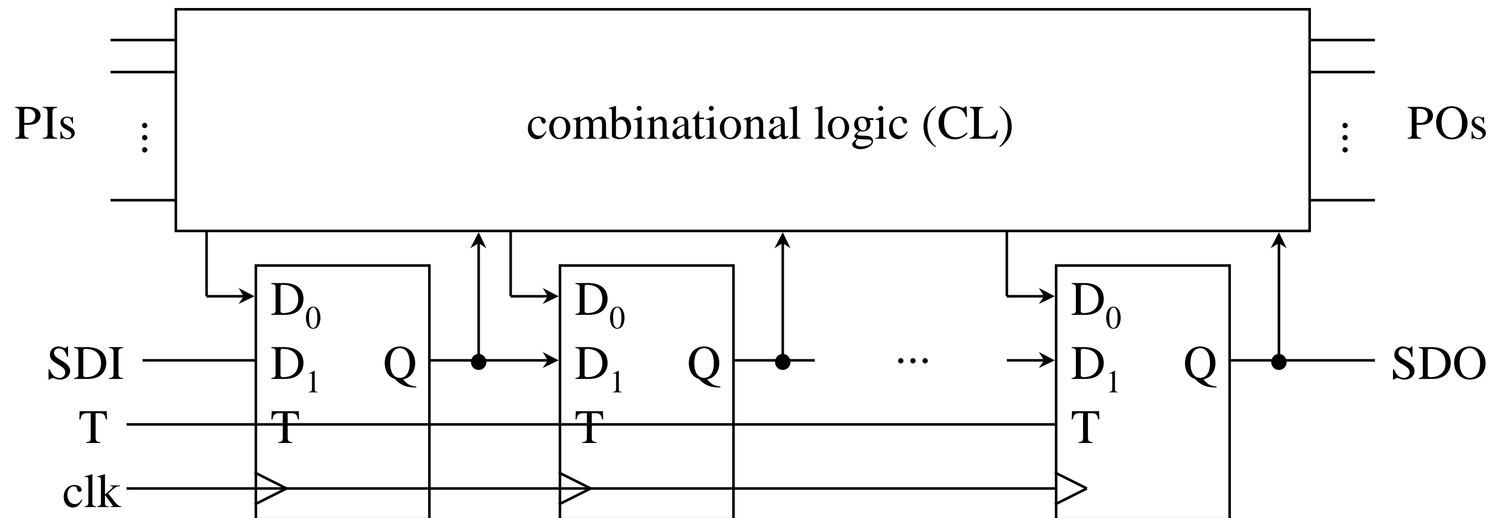
- Comparing the output for the good circuit and each of the faulty circuits gives:
  - (0, 0) detects C s-a-0.
  - (0, 1) detects A s-a-1 and C s-a-0.
  - (1, 0) detects B s-a-1 and C s-a-0.
  - (1, 1) detects C s-a-1.
- Therefore, the test set {(0, 1), (1, 0), (1, 1)} detects all of the faults.



© 2006 by Gerald E. Sobelman

Scan Design Technique

- We have seen that the algorithms for test generation do not apply to general sequential circuits. Therefore, in order to make these circuits testable, we should use a design methodology that allows them to be tested. Scan design is the main technique for achieving this.
- The basic concept is to convert each flip-flop into a *multiplexed-data flip-flop* having two data inputs, D_0 and D_1, and a control input, T:
  - One of the data inputs, D_0, is used for normal system operation.
  - The other data input, D_1, is connected so the flip-flop acts as one stage of a shift register. When configured in this way, the flip-flops are said to form a *scan chain*.



Scan Design Technique - Continued

- Note that some additional pins are used in this set-up:
  - SDI = serial data in (used for shifting in a set of values into the flip-flops)
  - SDO = serial data out (used for shifting out the state of the flip-flop Q outputs)
  - T = mode select (T = 0 for normal mode, T = 1 for test mode)
- With this configuration, we have complete controllability and observability of the internal state of the system. The problem is thus reduced to testing purely combinational logic, for which we can apply one of the known test generation algorithms.
  - A test vector is the concatenation of the values applied at the PIs with the values shifted into the scan chain.
  - The corresponding output vector is the concatenation of the values produced at the POs with the values shifted out of the scan chain.
- The basic testing procedure for a test vector is as follows:
  - Put the system into test mode.
  - Shift in the desired flip-flop contents through the SDI pin
  - Put the system into normal mode.
  - Operate the system for 1 clock cycle (PIs + present state -> CL -> POs + next state).
  - Put the system into test mode.
  - Shift out the flip-flop contents through the SDO pin.

Built-In Self-Test (BIST)

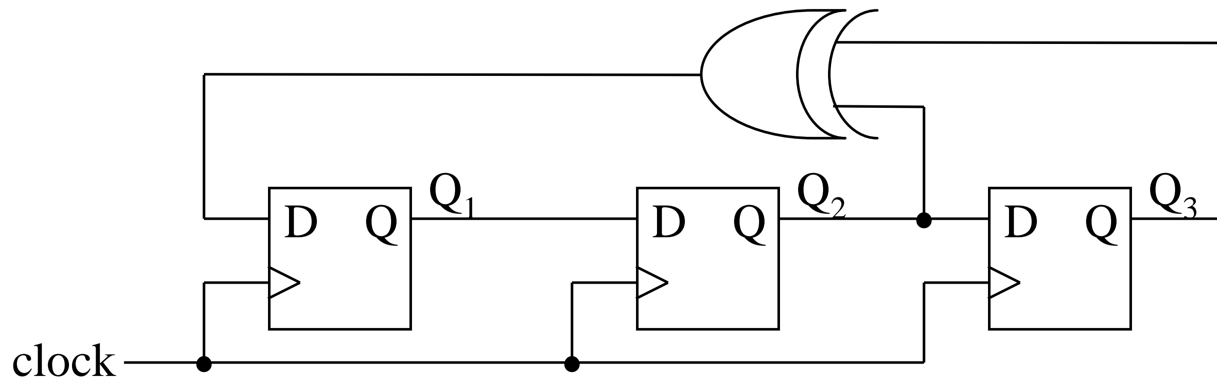
- We have been working under the assumption that the test vectors are applied and the responses analyzed using an *external* test system. Another possibility is to move the tester onto the chip itself. In other words, a portion of the chip will be used to generate test patterns, apply them to the rest of the chip and then analyze the responses from the rest of the chip.
- Clearly this is only feasible if both the test generation and response analysis procedures can be enormously simplified. This is done as follows:
  - Rather than calculating specific test vectors for a specific set of faults, we will use a linear feedback shift register (LFSR) to generate pseudo-random test vectors. The fault coverage of these vectors can still be obtained by fault simulation, so we can keep increasing the number of pseudo-random vectors until the fault coverage is deemed to be high enough.
  - Rather than checking each individual response vector for correctness, we will use a form of data compression to reduce all of the output data to a single quantity called the *signature*. We can then just compare the signature obtained with the value that should be obtained for a good circuit, which can be easily found by doing a simulation of the system. The data compression is performed in a multiple-input signature register (MISR), which is a special form of LFSR.
- With BIST, we get a good/bad indication without diagnostic information.

© 2006 by Gerald E. Sobelman

Generation of Pseudo-Random Test Vectors Using an LFSR

- Consider the 3-stage LFSR shown below. Starting from any non-zero state, it cycles in a pseudo-random order through all $2^3 - 1 = 7$ states other than the all-zero state:

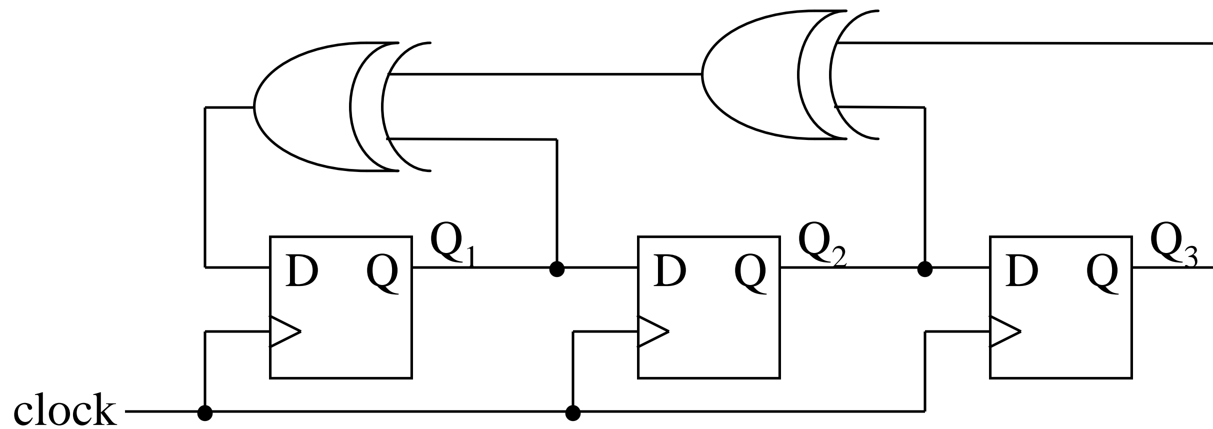
Q_1	Q_2	Q_3
1	0	0
0	1	0
1	0	1
1	1	0
1	1	1
0	1	1
0	0	1
1	0	0



© 2006 by Gerald E. Sobelman

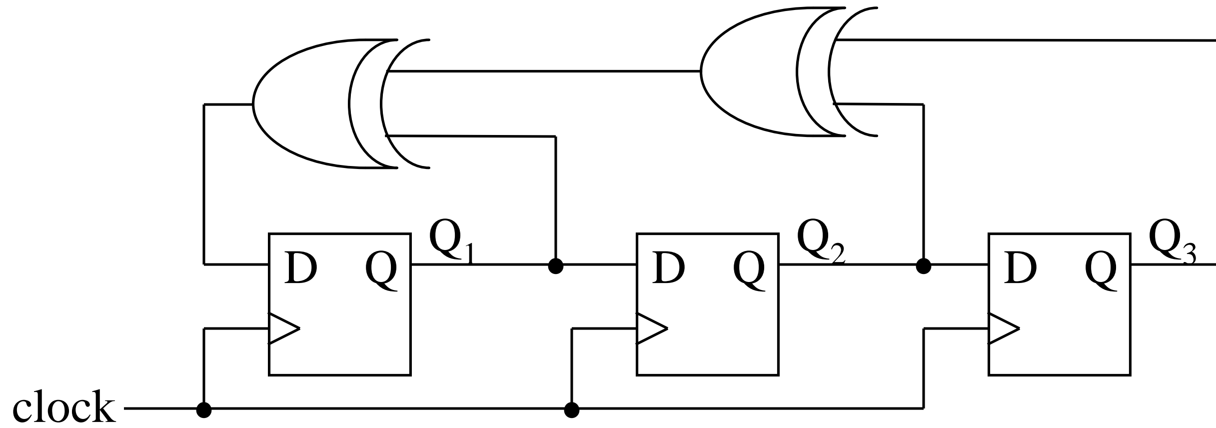
Analysis Problem

- Analyze the sequential behavior of the LFSR shown below. (Hint: You will need to consider several different initial states.)



© 2006 by Gerald E. Sobelman

Solution



Q_1	Q_2	Q_3
1	0	0
1	1	0
0	1	1
0	0	1
1	0	0

Q_1	Q_2	Q_3
0	1	0
1	0	1
0	1	0

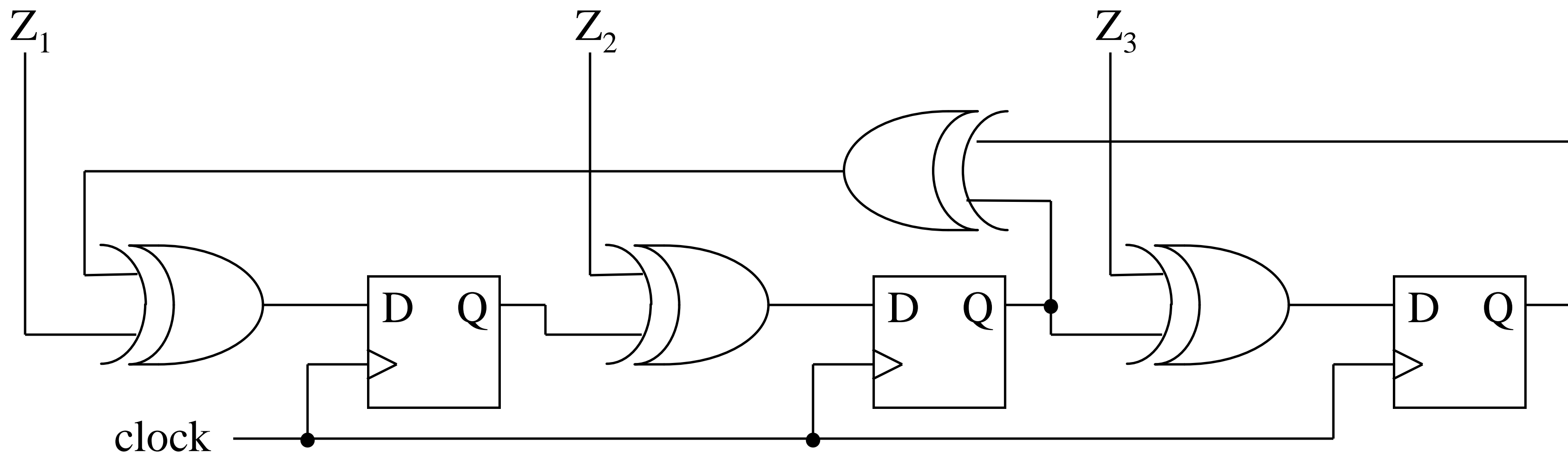
Q_1	Q_2	Q_3
1	1	1
1	1	1

Q_1	Q_2	Q_3
0	0	0
0	0	0

© 2006 by Gerald E. Sobelman

Compression of Output Data into a Signature Using a MISR

- The structure of a MISR is illustrated in the 3-stage design given below. The Z_i signals are the output signals from the remainder of the chip.
- Each time the MISR is clocked, a set of Z_i values gets compressed. After all of the output vectors have been produced and compressed, the final set of Q-output values constitutes the signature. The signature is then tested for equality with the prestored, precomputed value for a good circuit using a comparator (not shown).
- There is a small probability that the outputs produced by a faulty circuit will, by accident, get compressed into the signature for the good circuit. The probability of this happening in an n-stage MISR is proportional to 2^{-n}, so a large number of stages should be used to reduce this to a negligible chance.



© 2006 by Gerald E. Sobelman

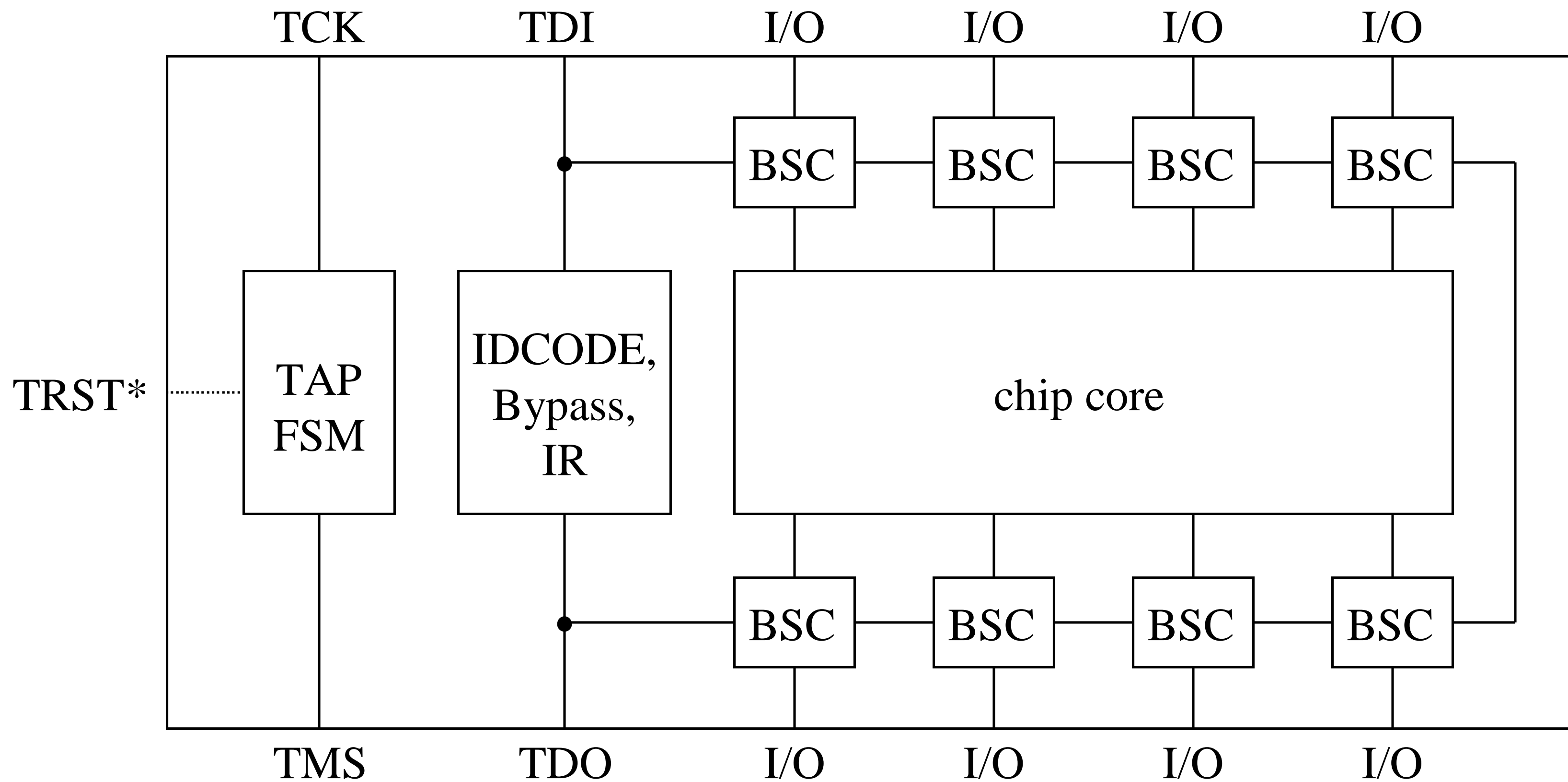
Boundary-Scan Concept

- The basic idea of boundary-scan is to provide a way to test components and their interconnections in an assembled system (e.g., chips and traces on a printed circuit board). The concept was first established as the JTAG standard, and is now also IEEE Standard 1149.1-2001 (Standard Test Access Port and Boundary-Scan Architecture).
- We want to be able to gain access to the input and output ports of components through a separate path. The standard approach is to set up a scan chain composed of D-type flip-flops and 2:1 MUXs. Control signals are used to determine if the system will operate normally (i.e., independently of the shift register) or in a test mode where the shift-register determines access to the component pins.
- Boundary-scan uses the following components and signals:
  - Boundary-Scan Cells (BSCs) between each signal pin and the chip's core. These cells either allow normal transmission of signals between the pins and the core, or else they provide a shift-register function for the I/O signals. The set of all BSCs forms the Boundary Register.
  - Additional registers (e.g., IDCODE, Bypass, Instruction Register (IR))
  - Test Access Port (TAP) Controller, which determines the operation of the boundary-scan system. It is a 16-state finite-state machine (FSM) with inputs TCK (the boundary-scan clock), TMS (Test Mode Select, the primary control input for the FSM) and TRST* (an optional reset signal).

© 2006 by Gerald E. Sobelman

Simplified View of Boundary-Scan

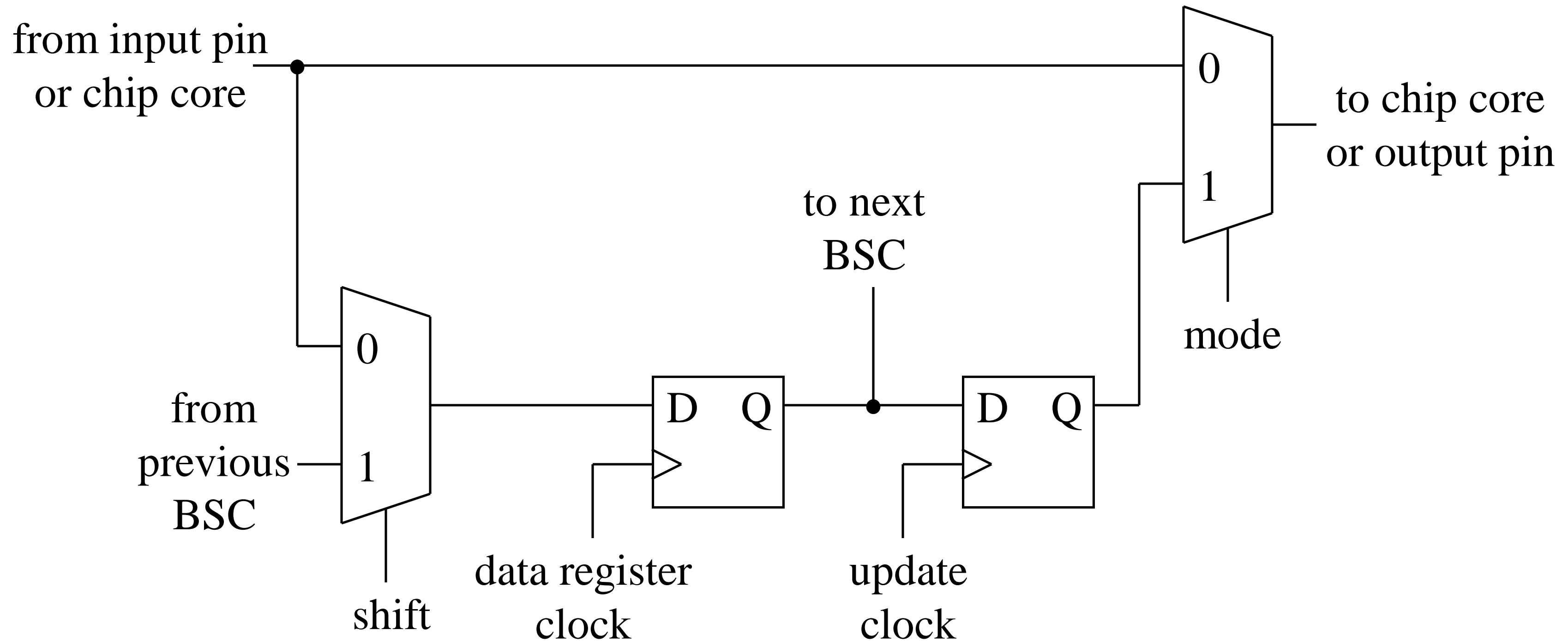
- A simplified view of the boundary-scan cells, registers and controller is shown below. The TDI pin is used to load all of the registers and BSCs, while their contents may be shifted out through the TDO pin. The TAP FSM controls the functions of the entire boundary-scan system.



© 2006 by Gerald E. Sobelman

Boundary-Scan Cell Design

- A typical design for a BSC is shown below. It is composed of two 2:1 MUXs and two D-type flip-flops. It can be used at either an input pin or an output pin.
  - *mode* controls whether normal or test value is sent to the core or output pin.
  - *shift* controls whether the input/core or scan value is loaded into the data register.



© 2006 by Gerald E. Sobelman

Using the Boundary-Scan Cell

- Consider the operation when the BSC is used at an input pin:
  - During normal mode, the input pin is connected to the core of the chip.
  - During test mode, *shift* is set to 1, so that the data register flip-flop in each BSC would be configured into a shift register:
    - First, all of these cells are loaded serially through the TDI pin.
    - Then, the update clock is pulsed and the values are transferred (in parallel) to the second flip-flop in each cell.
    - These values are then sent to the chip core through the second MUX.
- Next, consider the operation when the BSC is used at an output pin:
  - During normal mode, an output from the chip core is connected to an output pin.
  - During test mode:
    - First, *shift* is set to 0, so that the data register flip-flop in each of these cells can be loaded (in parallel) when the data register clock is pulsed.
    - Then, *shift* is set to 1, so that the data register flip-flops in the BSCs are configured into a shift register. As the data register clock is subsequently pulsed, these values are shifted out through the TDO pin.
- The BSCs can also be used to test the interconnect between chips on a board or to send/receive values to/from other non-boundary-scan chips that may be on the board.

© 2006 by Gerald E. Sobelman