Hazards and Testing

ENGI 9865 Advanced Digital Systems

1 Overview of Hazards and Testing

In this unit we are looking at the notions of hazards, reliability, and design for testing. Now, we have been doing functional testing all along to verify our designs, but when designing digital systems we also need to consider designing so that manufactured systems can be tested as well.

First we briefly review combinational hazards and then move on to notions of testing for manufacturing defects. Specifically, we will introduce design for testability (DFT), automatic test-pattern generators (ATPGs), scan design, and built-in self-test (BIST).

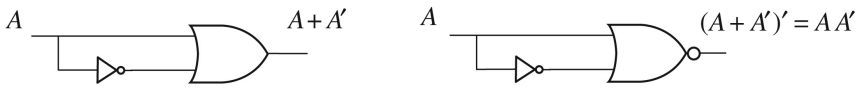
2 Hazards in Combinational Circuits

Let us start by thinking back to just combinational circuits. We know that as inputs change, unwanted transients (from switching, propagation delays, gate delays, wire delays, etc.) may appear on the output.

- If a circuit output briefly goes 0 when it should have stayed constant at 1, the circuit has a **static 1-hazard**.
- If a circuit output briefly goes 1 when it should have stayed constant at 0, the circuit has a **static 0-hazard**.
- When the circuit output is supposed to transition (either 0 to 1 or 1 to 0) and the output briefly changes three or more times, the circuit has a **dynamic hazard**.

In Figure 1 below, we show some circuits that illustrate 1-hazards and 0-hazards.

FIGURE 1-9: Simple Circuits Containing Hazards



(a) Simple circuit with static 1-hazard

(b) Simple circuit with static 0-hazard

Figure 1: Simple circuits containing (a) 1-hazard and (b) 0-hazard (Figure 1-9 from text)

Static 1-hazards occur in sum-of-product implementations when two minterms differing by only 1 input variable are not covered by the same product term. For example, see the circuit in Figure 2 (a) below. The timing chart in subfigure (b) assumes a 10 ns gate delay, which is what causes the 1-hazard.

FIGURE 1-10: Elimination of 1-Hazard

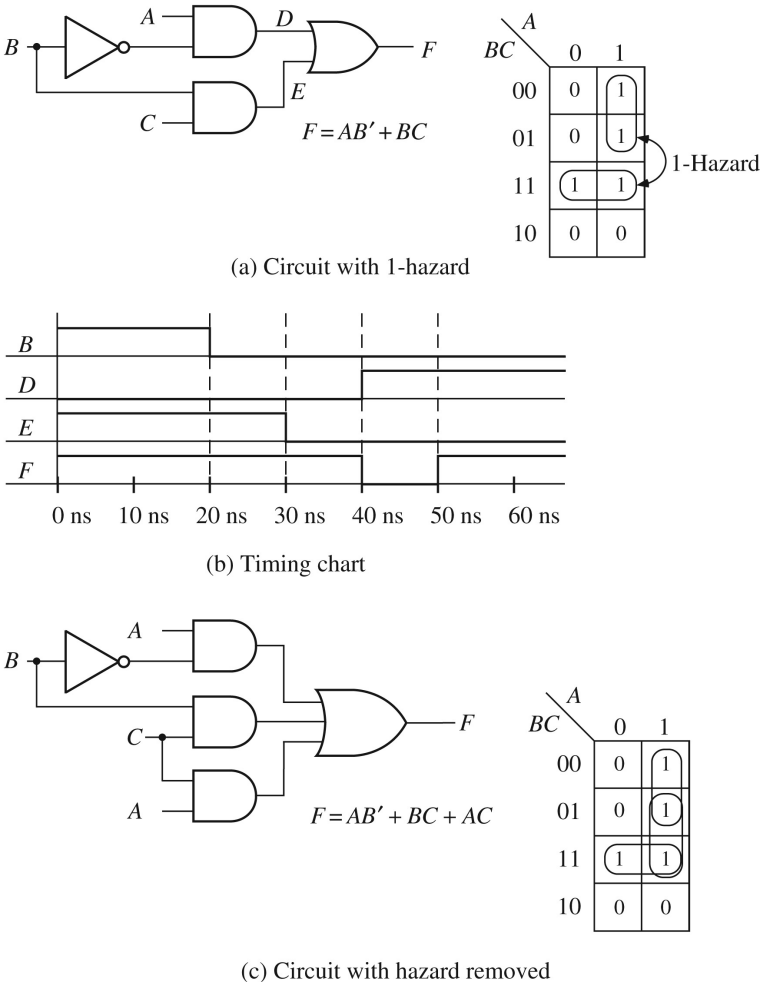


Figure 2: Elimination of 1-hazard (Figure 1-10 from text)

If we used a non-minimal circuit, we could eliminate the hazard as shown in subfigure (c) above, but we could generally only want to do this if we have combinational logic that exists on its own. One of the reasons we use sequential logic is that the use of memory elements and time allows us to ignore transient hazards and minimize combinational logic as much as we possibly can. However, if we do wish to design a circuit free of static and dynamic hazards, we can find a sum-of-products representation for the output such that every pair of adjacent 1's is covered by a 1-term. A two-level AND-OR circuit based on this representation will be free of 1-, 0-, and dynamic hazards.

2.1 Testing Combinational Logic

Two common types of faults in manufactured circuits are short circuits and open circuits.

- A short to ground acts as 0, short to power supply voltage acts as 1—thus, shorts can cause gate inputs to act as if they are stuck at 0 or 1.
- An open circuit (i.e. unconnected) input may act as either a 1 or 0 depending on the logic type being used.
- Thus, faults in logic circuits are typically modelled as **stuck-at-1 (s-a-1)** or **stuck-at-0 (s-a-0)** faults.

We can test gates for stuck-at faults by applying carefully chosen inputs and observing the output. For example, in Figure 3 (a) below, we apply all ones to the input of an AND gate to test for s-a-0 faults—if any input is stuck at 0, the output will be 0 when it should be 1.

FIGURE 10-1: Testing AND and OR Gates for Stuck-At Faults

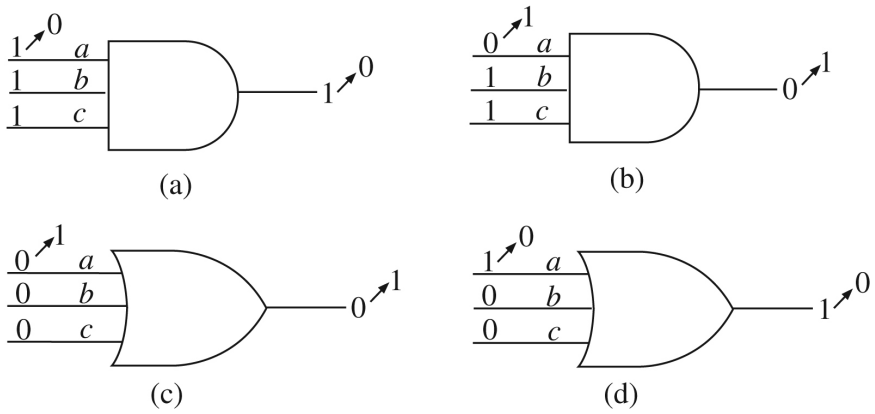


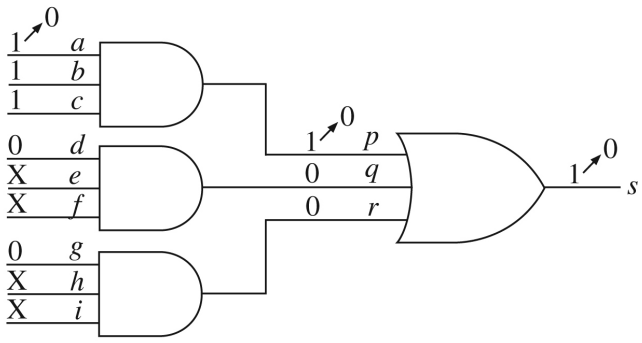
Figure 3: Testing AND and OR gates for stuck-at faults (Figure 10-1 from text)

In Figure 3 above, the arrow notation (eg. $1 \rightarrow 0$) means the normal value in correct operation (source of arrow) is changed to a different value (destination of the arrow) due to the fault.

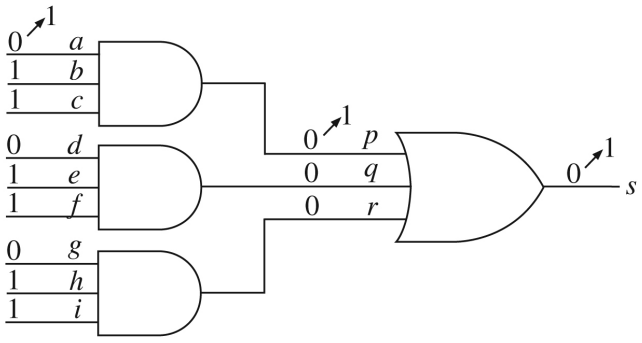
- The AND gate can be tested for stuck-at-1 faults by applying 0 to each input in turn and checking whether the output is 0.
- The OR gate is tested for s-a-1 by applying all 0s to the inputs.
- The OR gate is tested for s-a-0 by applying 1 to the input under test and 0 to the others.

This above seems easy enough, but now consider the circuit below in Figure 4.

FIGURE 10-2: Testing an AND-OR Circuit



(a) stuck-at-0 test



(b) stuck-at-1 test

Figure 4: Testing AND-OR circuit for stuck-at faults (Figure 10-2 from text)

In this case, assume the OR gate inputs (i.e. the internal signals) are not accessible, so gates cannot be tested individually. How do we test this?

- We could apply all $2^9 = 512$ possible input combinations and observe the output for the correct results, but this is horribly inefficient and doesn't scale well.
- A better approach is to test for all s-a-0 and s-a-1 faults using the test vectors from the table below.

TABLE 10-1: Test Vectors for Figure 10-2

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	Faults Tested
1	1	1	0	X	X	0	X	X	*a*0, *b*0, *c*0, *p*0
0	X	X	1	1	1	0	X	X	*d*0, *e*0, *f*0, *q*0
0	X	X	0	X	X	1	1	1	*g*0, *h*0, *i*0, *r*0
0	1	1	0	1	1	0	1	1	*a*1, *d*1, *g*1, *p*1, *q*1, *r*1
1	0	1	1	0	1	1	0	1	*b*1, *e*1, *h*1, *p*1, *q*1, *r*1
1	1	0	1	1	0	1	1	0	*c*1, *f*1, *i*1, *p*1, *q*1, *r*1

Figure 5: Test vectors for AND-OR circuit (Table 10-1 from text)

The basic idea is to carefully select test vectors and observe which faults they "cover". Through careful selection in the example above, we need only 6 test vectors to test for

all s-a-0 and s-a-1 faults, as opposed to the 512 vectors needed for an exhaustive test. Though these tests are developed assuming only a single fault, they may also detect the presence of multiple faults as well. However, these test vectors don't tell us where the fault is.

Unfortunately, while this sort of two-level circuit is easy enough to test using these kinds of test vectors, multilevel (i.e. more than 2-level) circuits are considerably more complex. To test for an internal fault, we have to choose a set of inputs that will excite that fault and then propagate the effect to the output, as illustrated in Figure 6 below.

FIGURE 10-3: Fault Detection Using Path Sensitization

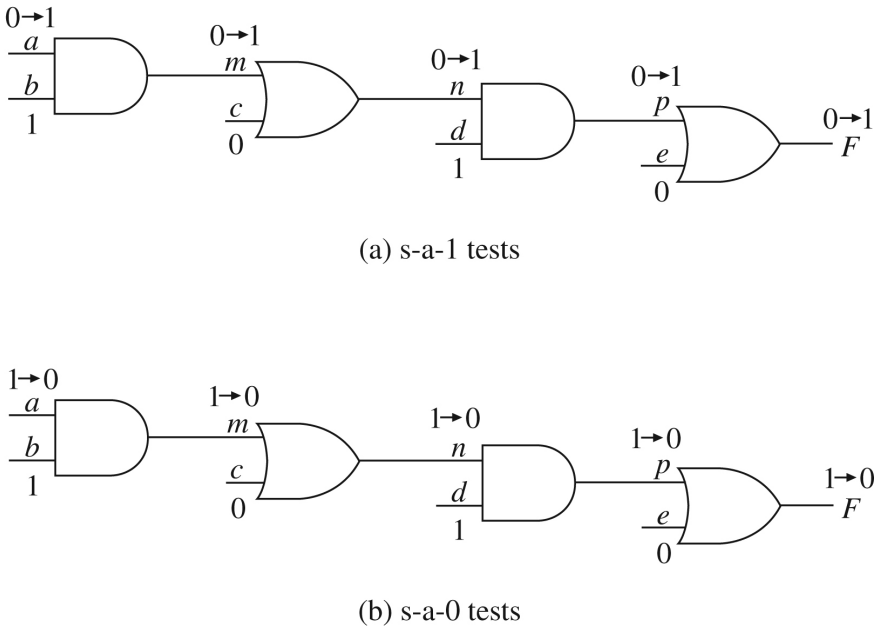


Figure 6: Fault detection using path sensitization (Figure 10-3 from text)

- *a, b, c, d, e* are inputs to the circuit.
- To test for, say, s-a-1 at gate input *n*, the input value must be 0.
- We can achieve this by setting inputs $a = 0, b = 1, c = 0$ as shown in (a).
- To ensure the fault value will propagate to the output *F*, we must set the inputs $d = 1$ and $e = 0$.
- This will, in fact, detect s-a-1 faults at any of *a, m, n,* or *p*.
- Changing input *a* to 1 will also test for s-a-0 faults at any of *a, m, n,* or *p*.
- The path through *a, m, n,* and *p* is said to be **sensitized**.

Path sensitization allows us to test for a variety of stuck-at faults using one set of circuit inputs.

Consider Figure 7 above. We wish to devise a minimum set of test vectors for all single s-a-1 and s-a-0 faults. We can apply inputs to A, B, C, D and observe the output F.

FIGURE 10-4: Example Circuit for Stuck-At Fault Testing (*p* stuck at 1)

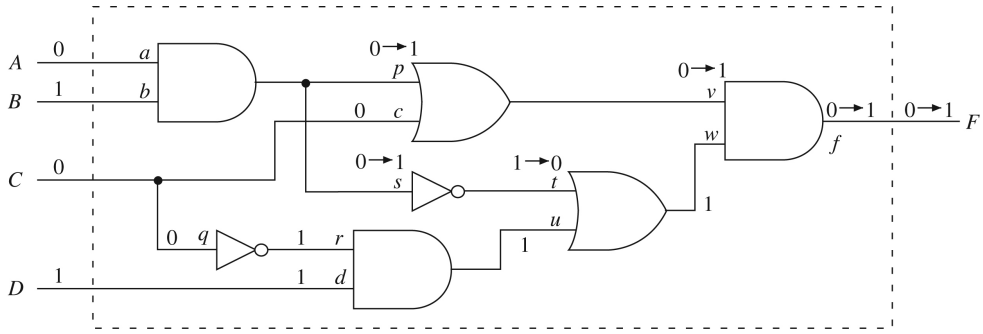


Figure 7: Example circuit for stuck-at fault testing (Figure 10-4 from text)

A general procedure to determine test vectors is as follows:

1. Select an untested fault.
2. Determine the required *ABCD* inputs.
3. Determine the additional faults that are tested.
4. Repeat this process until tests are found for all of the faults.

As an example, consider testing point *p* for a s-a-1 fault.

- We need to choose our inputs such that *p* should be 0, so let's say $A = 0, B = 1$.
- For a s-a-1 to appear on the output of that OR gate (*v*), we must set $C = 0$.
- For our result on *v* to propagate to the output, the final AND gate must have input $w = 1$.
- To get $w = 1$, we can work backward through the circuit and find that we need to set input $D = 1$.
- Note that our initial choice of $B = 1$ is what sensitized the path to test for s-a-1 faults at any of *a, p, v, f* with test vector 0101.
- To test that path for s-a-0 faults, use test vector 1101.

We can next pick an untested fault point and develop a new test vector to check it, and so on. This should result in a table like that in Figure 8 below.

There are many other kinds of faults that can occur (eg. bridging faults, when 2 unconnected signal lines are shorted together). For large combinational circuits, finding

TABLE 10-2: Tests for Stuck-At Faults in Figure 10-4

Test Vectors				Normal Gate Inputs													Faults Tested
A	*B*	*C*	*D*	*a*	*b*	*p*	*c*	*q*	*r*	*d*	*s*	*t*	*u*	*v*	*w*	*F*	
0	1	0	1	0	1	0	0	0	1	1	0	1	1	0	1	0	*a*1 *p*1 *c*1 *v*1 *f*1
1	1	0	1	1	1	1	0	0	1	1	1	0	1	1	1	1	*a*0 *b*0 *p*0 *q*1 *r*0 *d*0 *u*0 *v*0 *w*0 *f*0
1	0	1	1	1	0	0	1	1	0	1	0	1	0	1	1	1	*b*1 *c*0 *s*1 *t*0 *v*0 *w*0 *f*0
1	1	0	0	1	1	1	0	0	1	0	1	0	0	1	0	0	*a*0 *b*0 *d*1 *s*0 *t*1 *u*1 *w*1 *f*1
1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	0	0	*a*0 *b*0 *q*0 *r*1 *s*0 *t*1 *u*1 *w*1 *f*1

Figure 8: Test vectors for stuck-at faults in figure above (Table 10-2 from text)

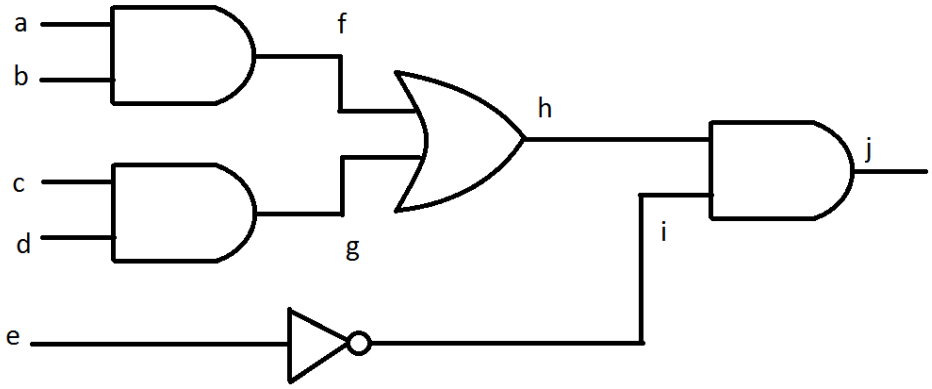
a minimum set of test vectors can be hard. For circuits containing redundant gates, it may be impossible to test for some faults. And even if we can find a good set of test vectors, it may take too long to run/simulate them.

Thus, it is common in practice to use a comparatively small set of test vectors that tries to test for most of the faults. There are many algorithms and software tools that try to efficiently generate these test patterns and estimate the percentage of possible fault points are tested by a set of test vectors (a.k.a. the coverage). However, as a little heads-up, finding test vectors for a simple circuit might make a nice exam question.

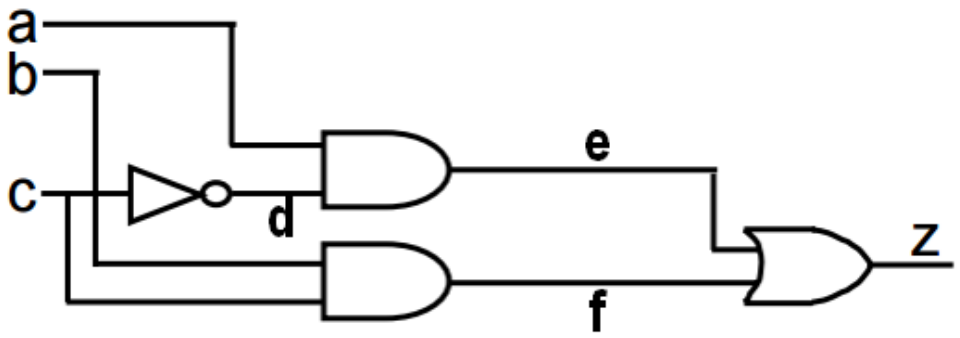
2.2 Practice Problems in Testing for Stuck-at-Faults

Try out these practice problems to ensure you understand the concept.

(a) Generate test vectors for s-a-0, s-a-1 at f:



(b) Generate a complete set of test vectors (points a–f) for the following circuit:



3 Hazards and Sequential Logic

Sequential logic is generally even harder to test for faults than combinational logic since test vectors would generally be sequences of inputs. If we can observe only input and output sequences and not internal states of flip-flops, we will need a large number of test sequences to determine if the circuit under test is equivalent to a correctly functioning circuit.

One of the reasons we should always design sequential circuits with a reset is to that it can be tested from a known initial state. We could use this known starting point in a brute-force, exhaustive test where we reset the system, apply a set of input sequences, and observe a set of output sequences to see if they are as expected. However, it should be obvious that this is a totally impractical approach. We should try to derive a small set of test sequences for the circuit.

One way to derive possible test sequences is to imagine the sequential circuit as an iterative circuit. An example is shown in Figure 9 below.

- Sequential circuits operate in time—each clock cycle brings new input as well as new values in flip-flops.
- An iterative version of the circuit operates in space—the combinational part of the sequential circuit is replicated such that the output of the first "iteration" of the combinational logic is used alongside the second input vector as input into the second "iteration" of the logic, and so on.
- You can think of this as something like loop unrolling in software, when you get rid of a loop statement by replicating its body the number of times the loop would have iterated.

Once we have envisioned the circuit in iterative form, it consists of combinational logic. We can develop test vectors as before. After these test vectors have been dervied, they can become the input sequences used to test the actual sequential circuit.

Even with this technique finding an adequate set of test vectors is hard. For example, consider that state machine shown in Figure 10 below.

The state transitions are summarized in the following table:

If S_0 is the reset state, we can start from there and develop input sequences to test all possible state transitions. This is a necessary test, but it is not sufficient. This can be most easily illustrated with an example.

- Assume the input sequence $X = 010110011$.
- The preceding input sequence will traverse all the transitionis in the state graph.
- The observed output sequence will be $Z = 001011110$.

FIGURE 10-5: **Sequential and Iterative Circuits**

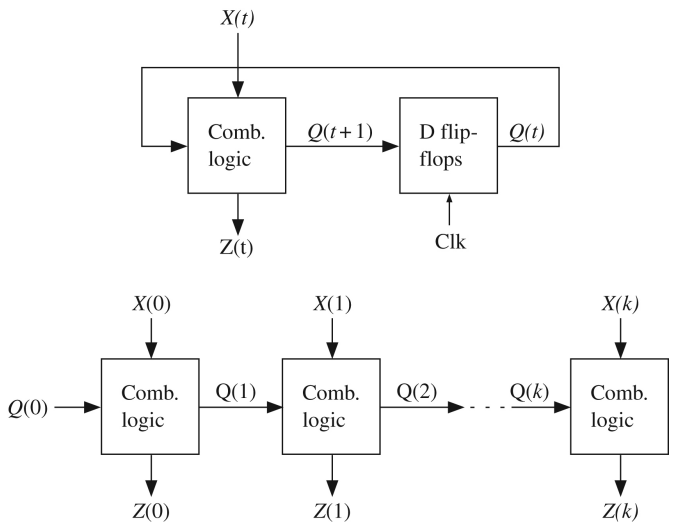


Figure 9: Sequential as iterative circuit (Figure 10-5 from text)

FIGURE 10-6: **State Graph for Test Example**

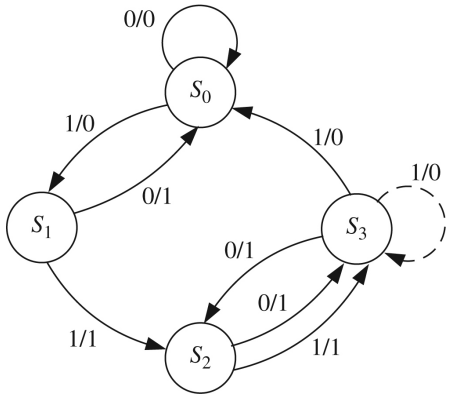


Figure 10: Example state machine (Figure 10-6 from text)

- However, now imagine that the 0/1 arc from S_3 to S_0 is gone, replaced with a self loop on S_3 (shown as a dashed transition in the figure above).
- In this case, the output sequence would be the same, but this new version of the state machine is not equivalent to the old one.

A state graph in which every state can be reached from every other state is known as **strongly connected**.

- Can try to test a sequential circuit with a strongly connected state graph and no equivalent states by finding a sequence that will distinguish each state from the others, a.k.a. a **distinguishing sequence**.
- We can only distinguish between states if there is at leaset one finite input sequence that will produce different output sequences.

TABLE 10-3: State Table for Figure 10-6

$Q1Q2$	State	Next State $X=0$	Next State 1	Output $X=0$	Output 1
00	S_0	S_0	S_1	0	0
10	S_1	S_0	S_2	1	1
01	S_2	S_3	S_3	1	1
11	S_3	S_2	S_0	1	0

Figure 11: Example state table (Table 10-3 from text)

- It has been proven that if two states of a state machine *M* are distinguishable, they can be distinguished by a sequence of length $\leq n-1$ where n is the number of states in *M*.
- In our example above, a distinguishing sequence is 11.

We could also try to derive test sequences based on testing for stuck-at faults. For example, in Figure 12 below we see a circuit implementing the state graph shown in the figure above with the following state assignments: $S_0 = 00, S_1 = 10, S_2 = 01, S_3 = 11$

FIGURE 10-7: Realization of Figure 10-6

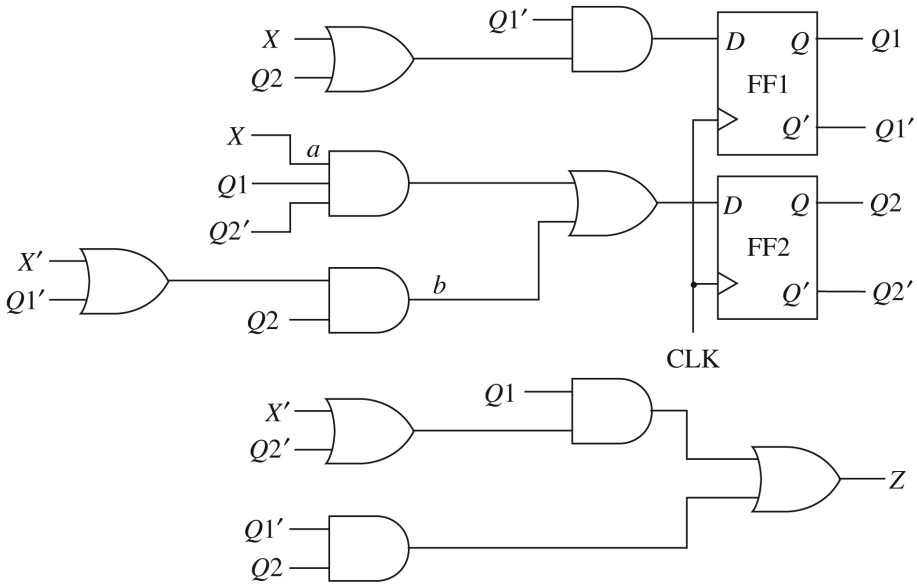


Figure 12: Example state machine realization (Figure 10-7 from text)

To test gate input *a* for s-a-1, we need to get into state S_1 and then set the input to 0. In normal operation, the next state would be S_0, but if a fault is present, the next state would be S_2. We can build a full test sequence as follows:

- To get to state S_1, reset followed by $X = 1$.
- To test point *a* for s-a-1: input $X = 0$.

- To distinguish the resulting state that is reached: input $X = 11$.
- So the full sequence is R1011. Normal output would be 0101; faulty output would be 0110.

None of the above techniques are especially good or easy to use, however. Things get a lot easier if we can peek inside a sequential circuit....

3.1 Scan Testing

As previously mentioned, sequential circuits tend to exist as pools of combinational logic with flip-flops distributed throughout. Testing that knowing overall inputs and ouputs is hard, but if we can observe the state of the internal flip-flops, testing becomes much easier.

- Observing the internal state of flip-flops means our tests can be simpler, since we can observe both overall system output as well as correct internal state behaviour without having to resort to finding distinguishing sequences.
- A first pass approach at this might be to connect the output of all internal flip-flops to pins on the device, but this is impractical as the number of available pins on an IC is typically limited.
- But if the flip-flops inside the chip are configured to form one giant shift register, then we could access internal state bit-by-bit using only one serial output pin on the IC.
- This is called **scan path testing**.

Consider Figure 13 below.

- The sequential circuit is separated into a combinational logic part and a state register composed of flip-flops.
- (The "state register" does not just refer to flip-flops used in a state machine, but to all values stored in flip-flops that comprise the current "state" of the system.)
- In the above figure, however, each flip-flop has two input ports and two clocks.
- Clock C1 is the system clock (SCK) that governs normal operation; on the edge of C1 input D1 is stored to Q.
- Clock C2 is a test clock (TCK) used for shifting test values; on the edge of C2, input D2 is stored to Q.
- The Q output of each flip-flop is connected both to the combinational logic that requires it as well as the D2 input of the next flip-flop in the scan chain.

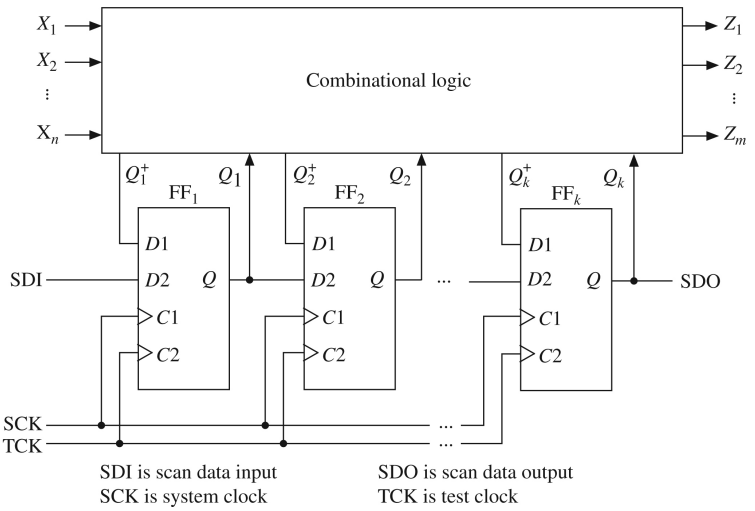


FIGURE 10-8: Scan Path Test Circuit Using Two-Port Flip-Flops

Figure 13: Example of a scan path test circuit using 2-port flip-flops (Figure 10-8 from text)

- The first flip-flop in the chain has the SDI pin (scan data in) connected to its D2 port; the last flip-flop has its Q output also connected to the SDO pin (scan data out).

During testing, a desired state is shifted in to the flip-flops using SDI and TCK. Then, an input test vector is applied, the outputs are verified, and SCK is pulsed to shift the system into the next state. The next state can be verified by using TCK to shift the new state value out of the scan data register via SDO.

Using this technique and infrastructure, we reduce the problem of testing a sequential circuit to testing a combinational circuit, and test vectors can be developed using any of the previously-discussed methods. We can formalize this a little better as shown below.

1. Scan in test vector state values Q_i via SDI using test clock TCK.
2. Apply corresponding test values to X_i inputs.
3. After sufficient time for signals to propagate through combinational logic, verify the output Z_i values.
4. Apply one pulse to system clock SCK to store the new values of Q_i^+ into corresponding flip-flops.
5. Scan out and verify Q_i values by pulsing the test clock TCK.
6. Repeat steps 1 through 5 for each test vector.

Note that you can overlap steps 1 and 5, and scan in the next test vector while scanning out the result of the previous test.

We can consider an example circuit of the form shown in the figure above, with 2 inputs X_1, X_2, three flip-flops Q_1, Q_2, Q_3, and 2 outputs Z_1, Z_2. One row of its state transition table can be represented as:

Current State $Q_1Q_2Q_3$	Current Input X_1X_2	Next State $Q_1^+Q_2^+Q_3^+$	Output Z_1Z_2
101	00	010	10
101	01	110	11
101	10	111	01
101	11	011	00

Figure 14 below shows a set of tests for these transitions. (Note that this is not an exhaustive test.)

FIGURE 10-9: Timing Chart for Scan Test

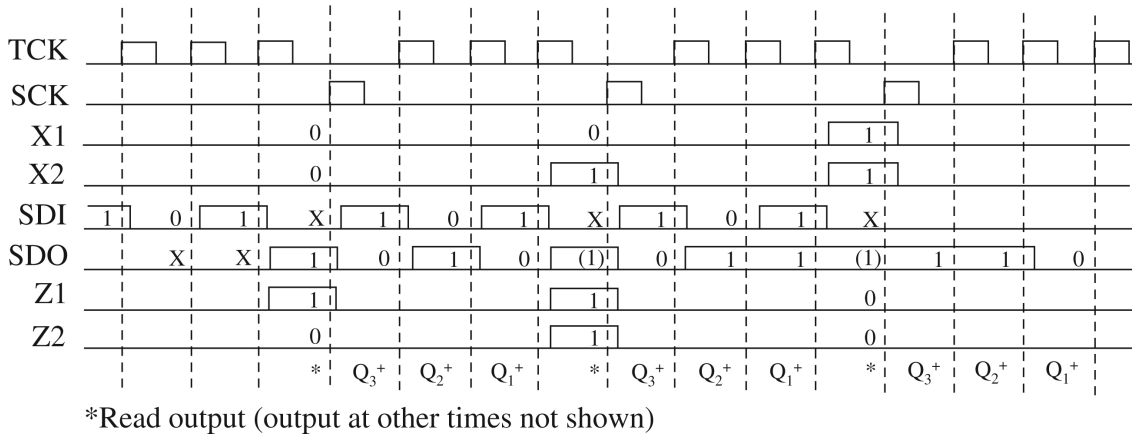


Figure 14: Example timing chart for scan test example (Figure 10-9 from text)

So, for a general system we design that consists of flip-flop registers separate by blocks of combinational logic, as shown in Figure 15 (a) below, we can convert it to make use of scan testing by replacing our regular flip-flops with 2-port flip-flops (or other scannable flip-flops) and linking them in in a scan chain as shown in subfigure (b). Good CAD tools for ASIC design can (try to) insert these sorts of scan chains automatically.

FIGURE 10-10: **System with Flip-Flop Registers and Combinational Logic Blocks**

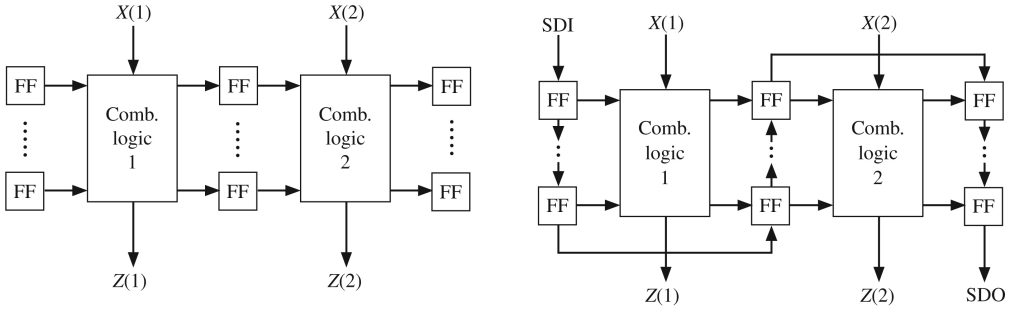


Figure 15: Converting basic design to use scan chain (Figure 10-10 from text)

If we are designing a system using multiple ICs on a board, it is possible to chain together the scan registers in each IC so the entire board can be tested using a single serial access port, as illustrated in Figure 16 below.

FIGURE 10-11: **Scan Test Configuration with Multiple ICs**

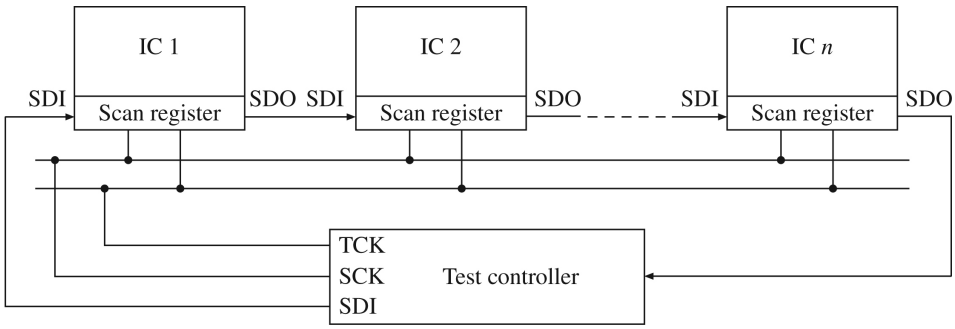


Figure 16: Multi-IC scan test configuration (Figure 10-11 from text)

3.2 Boundary Scan

As ICs and printed circuit boards have become more complex, traditional testing methods (eg. “bed-of-nails” test fixture) became impractical, so boundary scan test methodology has been developed by the Joint Test Action Group (JTAG) and adopted as ANSI/IEEE Standard 1149.1, “Standard Test Access Port and Boundary-Scan Architecture”. According to the standard, a boundary scan register (BSR) is placed on each IC between each input or output pin and the internal core logic, and 4–5 pins are devoted to being the test access port, or TAP. The TAP controller and other test logic are also part of the IC. This is shown in Figure 17 below.

The TAP interface is as follows:

- TDI is the test data input port; this data is shifted serially into the BSR.
- TCK is the test clock.

FIGURE 10-12: IC with Boundary Scan Register and Test-Access Port

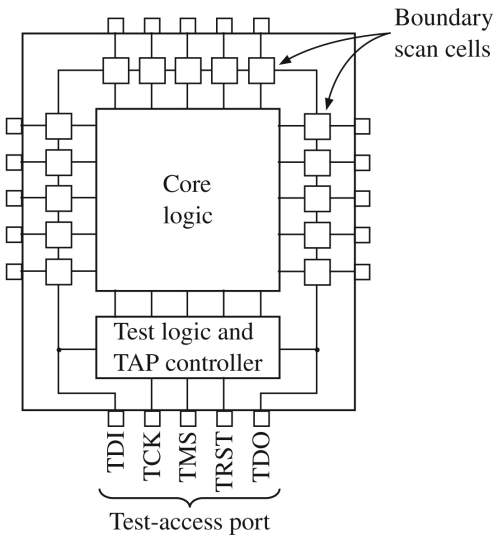


Figure 17: IC with BSR and TAP (Figure 10-12 from text)

- TMS is the test mode select.
- TDO is the test data output port; this data is shifted serially out of the BSR.
- TRST is an optional test reset input; if present, can be used to reset TAP controller and test logic.

A board with multiple ICs following this standard can be tested with only a few test connector pins/traces as shown in Figure 18 below.

FIGURE 10-13: PC Board with Boundary Scan ICs

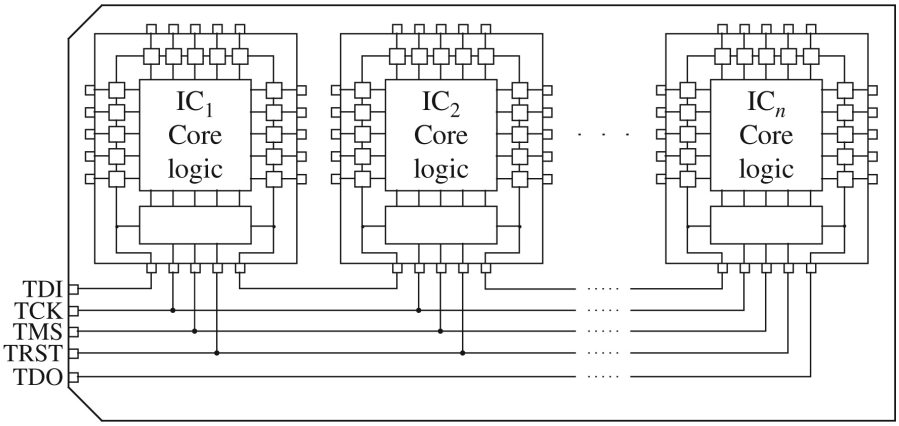


Figure 18: Circuit board with boundary scan ICs (Figure 10-13 from text)

The individual boundary scan cells (shown on the next page in Figure 19) have two inputs: the serial TDI as well as parallel in from core logic or input pins. They also have two outputs, TDO and the parallel out to core logic or an output pin.

FIGURE 10-14:
Typical Boundary Scan Cell

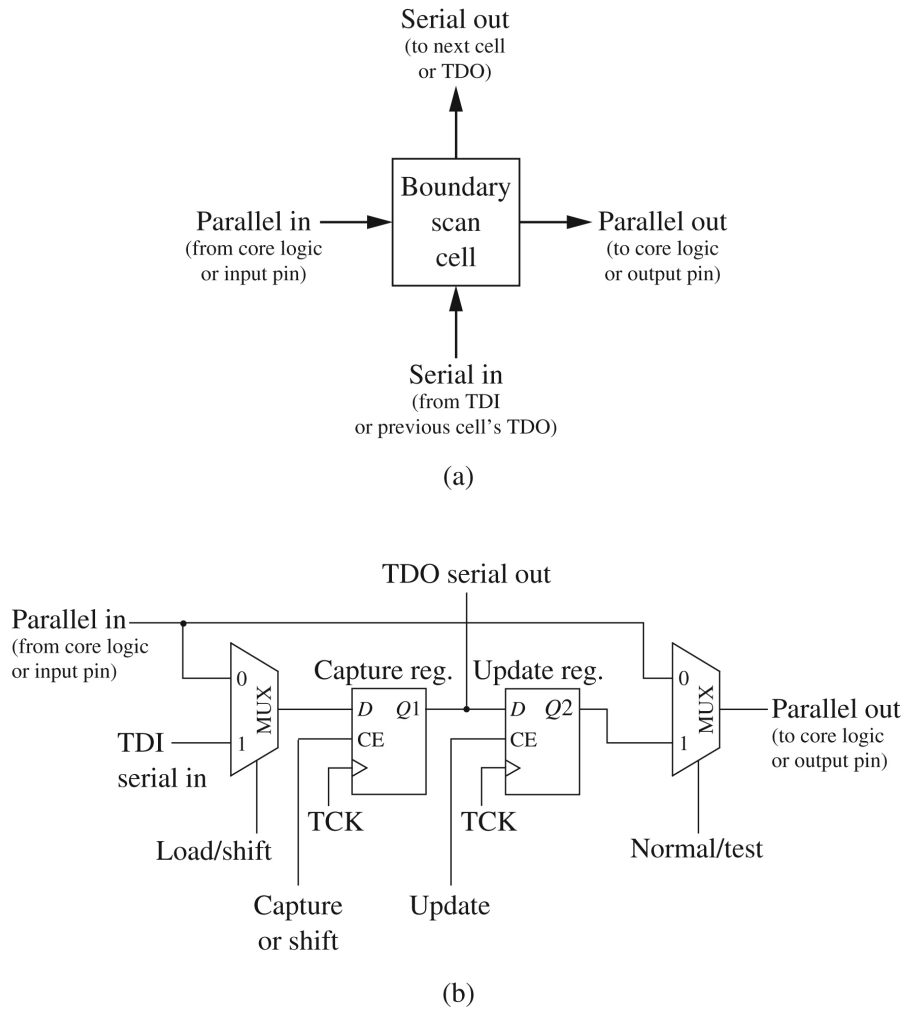


Figure 19: Typical boundary scan cell (Figure 10-14 from text)

4 Built-In Self Test

JTAG was standardized in 1990, but systems have continued to grow in complexity since then, and thus become much harder and more expensive to test. One solution is to take advantage of the additional transistors available on an IC as technology improves to add logic that allows the chip to test itself. This is referred to as **built-in self-test**, or BIST.

The basic idea is than an on-chip test generator applies test patterns to the circuit under test, and the resulting output is observed by a response monitor. If the monitor detects an error, it can produce an error signal. An example of this is shown below in Figure 20.

FIGURE 10-23: Generic BIST Scheme

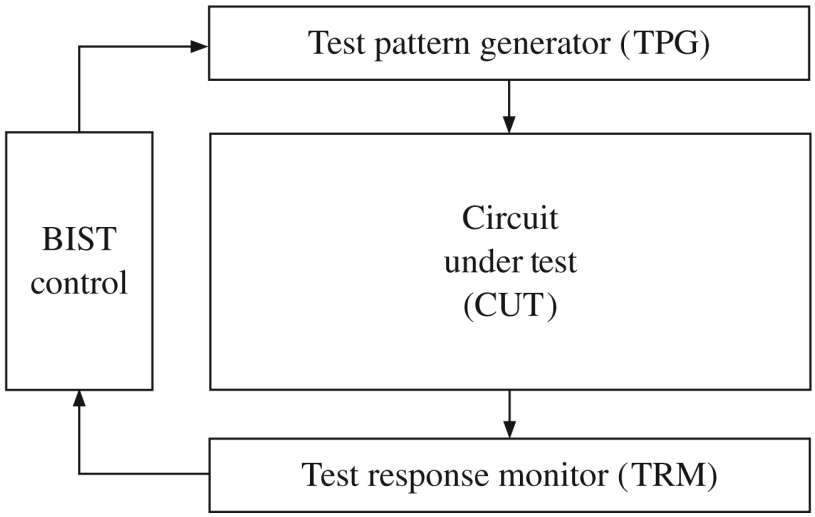


Figure 20: Generic BIST scheme (Figure 10-23 from text)

BIST is often used for testing memory, since the regular structure of a memory chip makes it easy to generate test patterns. An example of such a system is shown in Figure 21. The basic idea is that the write-data generator and address counter are enabled to write to each memory location, then the read-data generator and address counter are enable to read it back, with the data read back from the cell being compared against the expected value produced by the read-data generator.

FIGURE 10-24: Self-Test Circuit for RAM

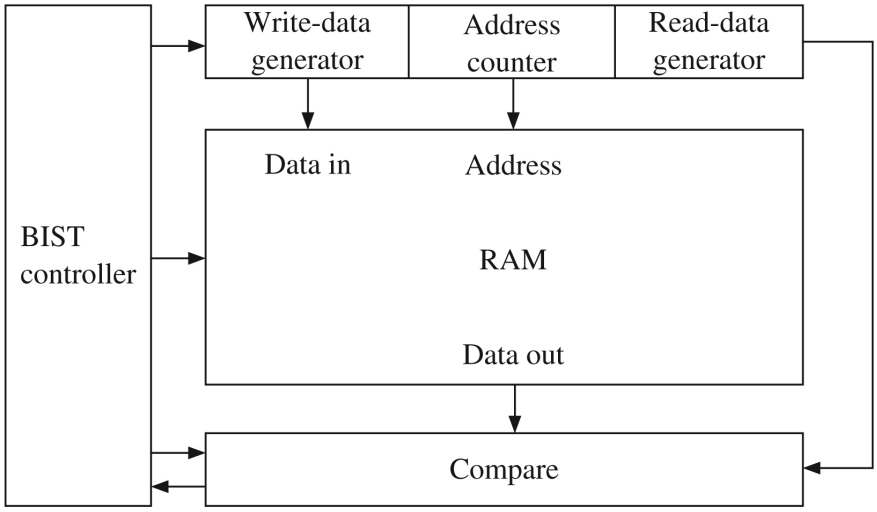


Figure 21: RAM self-test (Figure 10-24 from text)

- Memory often tested by writing checkerboard patterns. Eg. 01010101 on even addresses and 10101010 on odd, and then the reverse in a second test. This tests every bit for every possible value.

- The March test reads each cell and then writes its complement back to it. This is done to the whole array, and then repeated in reverse order of addresses.
- Testing can be simplified by using a signature register—output data gets compressed into a shorter string called a **signature** which is then compared to signature of correctly functioning component.

A **multiple-input signature register (MISR)** combines and compresses several output streams into a single signature, and makes a test circuit much simpler as shown in Figure 22 below.

FIGURE 10-25: Self-Test Circuit for RAM with Signature Register

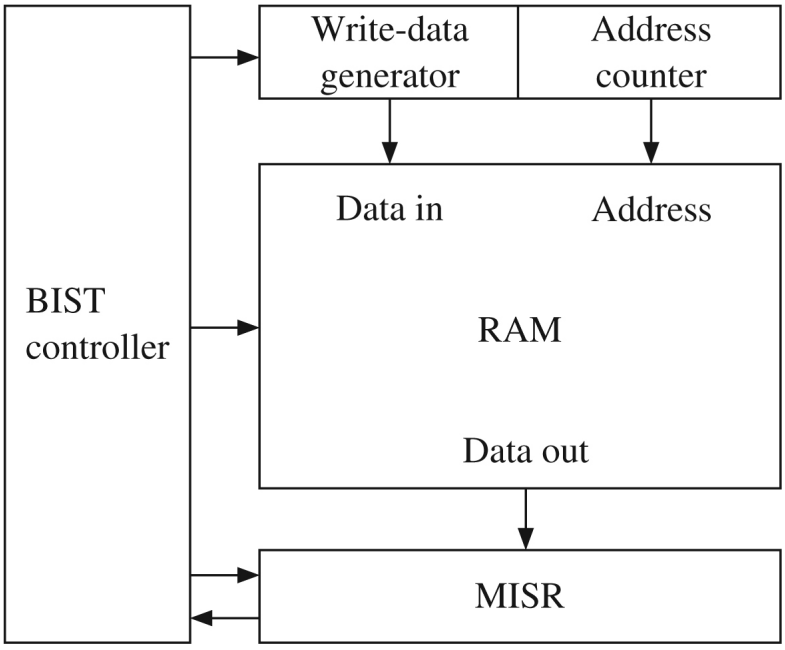


Figure 22: RAM self-test with signature (Figure 10-25 from text)

Linear feedback shift registers (LFSRs) are often used to generate test patterns and compress test outputs into signatures. LFSRs have the serial input bit as a function of the shift register content. By careful choice of which outputs that are fed back through an XOR gate to generate the input, it is possible to generate $2^n - 1$ different bit patterns (except for all 0) using an n-bit shift register.

Such an LFSR is often called a pseudo-random pattern generator, and is useful because they can generate a large number of test patterns with a small amount of circuitry. Some examples of feedback that will generate the full range of values for some LFSRs are shown in the table below (in Figure 23).

TABLE 10-4: Feedback for Maximum-Length LFSR Sequence

n	Feedback
4, 6, 7	$Q_1 \oplus Q_n$
5	$Q_2 \oplus Q_5$
8	$Q_2 \oplus Q_3 \oplus Q_4 \oplus Q_8$
12	$Q_1 \oplus Q_4 \oplus Q_6 \oplus Q_{12}$
14, 16	$Q_3 \oplus Q_4 \oplus Q_5 \oplus Q_n$
24	$Q_1 \oplus Q_2 \oplus Q_7 \oplus Q_{24}$
32	$Q_1 \oplus Q_2 \oplus Q_{22} \oplus Q_{32}$

Figure 23: Feedback taps for maximum-length LFSR sequences (Table 10-4 from text)

An LFSR can be modified to produce an all-0 state as well by adding an AND gate with $n-1$ inputs, as shown in Figure 24 below. In that example, state 0001 will produce state 0000, and then then next state will be 1000. Other than that, the sequence would be the same as if the AND gate wasn't present.

FIGURE 10-27: Modified LFSR with 0000 State

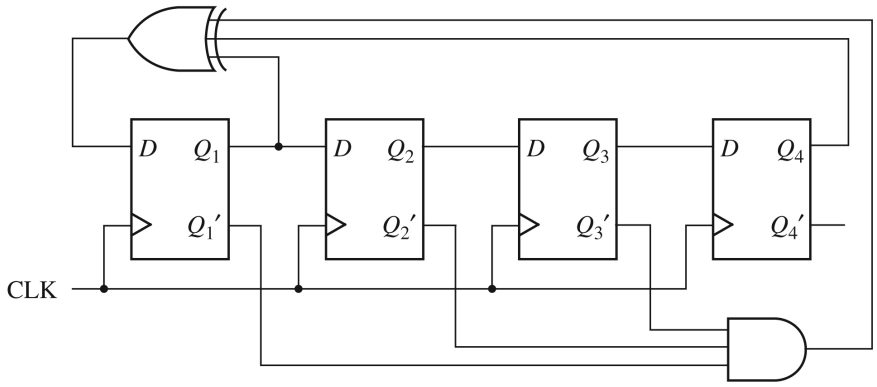


Figure 24: Modified LFSR with 0000 state (Figure 10-27 from text)

We can use an LFSR to construct an MISR by adding XOR gates to the input of each flip-flop as shown in Figure 25 below.

FIGURE 10-28: Multiple-Input Signature Register (MISR)

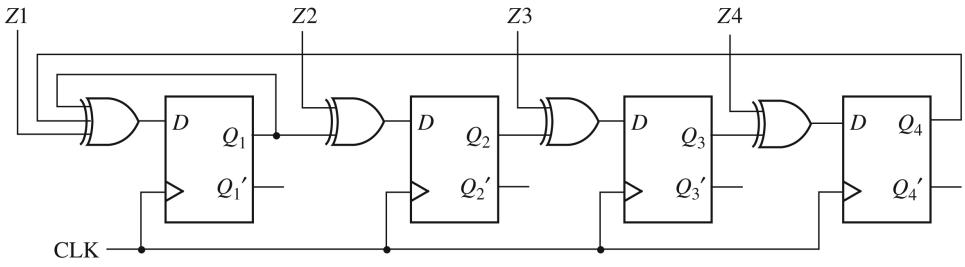


Figure 25: MISR from LFSR (Figure 10-28 from text)

By XORing the test data into the LFSR state, the final result is a signature that can be compared against what the correct signature should be if everything is working fine. This will catch many—but not all—possible errors. The probability that an incorrect input sequence will produce a correct signature is on the order of $\frac{1}{2}^n$.

Consider the following example using the circuit above:

- Assume an input sequence of: 1010, 0001, 1110, 1111, 0100, 1011, 1001, 1000, 0101, 0110, 0011, 1101, 0111, 0010, 1100.
- This should produce a signature of 0010 if the initial contents of the MISR were 0000.
- Any input that differs in 1 bit will map to a different signature. (Eg. if 0001 was changed to 1001, the resulting sequence would be 0000.)
- Most sequences that differ in 2 bits will also be detected, but there are certain cases (eg. 0001 to 1001 and 0010 to 0110) that would produce a signature of 0010, and thus the errors won't be detected.

4.1 STUMPS

STUMPS stands for "Self-Testing Using an MISR and Parallel SRSG", and is an example of a popular architecture for BIST. The SRSG in its name refers to "Shift Register Sequence Generator". STUMPS uses scan chains to perform testing, as illustrated in Figure 26 below.

FIGURE 10-29: The STUMPS Architecture

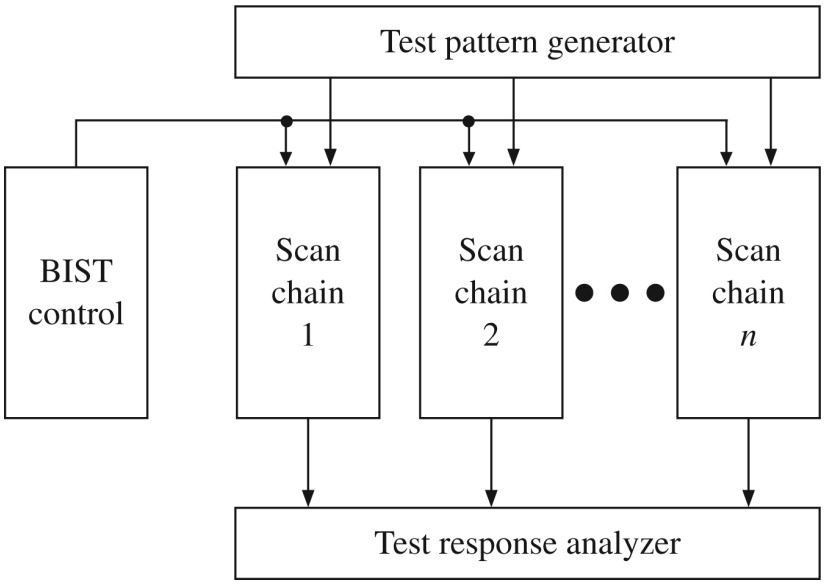


Figure 26: STUMPS architecture (Figure 10-29 from text)

STUMPS using a pseudo-random pattern generator to feed test stimulus to the scan chain, runs a capture cycle, and then has a test response analyzer to receive and evaluate the responses:

1. Scan in patterns from the test pattern generator (LFSR) into all scan chains.
2. Switch to normal function mode and clock once with system clock.
3. Shift out scan chain into test response analyzer (MISR) where test signature is generated.

If a device has a long scan chain, it will take many clock cycles to read data in and out to run one test. (This is a drawback of a test-per-scan scheme.) However, we can reduce testing time by using a larger number of parallel scan chains, so different parts of the chip can be tested independently in parallel.

4.2 BILBO

BILBO stands for "Built-In Logic Block Observer", and is a test-per-clock scheme that offers faster testing. Under this scheme, the scan register is modified to that it can serve as a state register, pattern generator, signature register, or shift register. Thus, portions of it can be configured to shift in data as in a normal scan, generate test patterns and signatures, or function normally as the situation requires; when one area of the device is tested, we can more on to another. However, after the registers are initialized for a test, there is no time wasted to scan values in or out—a test can be applied each clock cycle in a test-per-clock scheme.

An example of this can be seen in Figure 27 below, which shows a circuit with two combinational blocks. Figure 27 (a) shows a test for the first block; subfigure (b) shows the test for the second. In each case, the input register to the block is configured as a pattern generator (PRPG) and the output register acts as a signature register (MISR).

Figure 28 below shows the detailed structure of a 4-bit BILBO register. Control inputs B1 and B2 control the mode, and Z1–Z4 are the inputs from the combinational logic. When B1 and B2 are 00, it operates as a shift register; 01 as a PRPG; 10 is normal register operation; and 11 is MISR.

Figure 29 below shows how BILBO registers can be used for testing.

- Registers A and B are loaded from the data bus using LDA and LDB signals.
- The register values are added together and the sum and carry are stored in register C following the LDC signal.
- For testing, set B1 and B2 to 00 and scan in initial values for A, B, C.
- Then, set B1 B1 to 01, which places A and B in PRPG mode and C in MISR mode (due to XOR gate).
- Clock for 15 cycles.
- Set B1 B2 to 00 and scan out the signature.

So, while there is some scanning in to initialize the BILBO registers, when the test is running, each cycle has A and B generating the next test input without need new data shifted in. This is why this is a "test-per-clock" approach. We can speed up testing at the cost of more complex test hardware.

FIGURE 10-30: BIST Using BILBO Registers

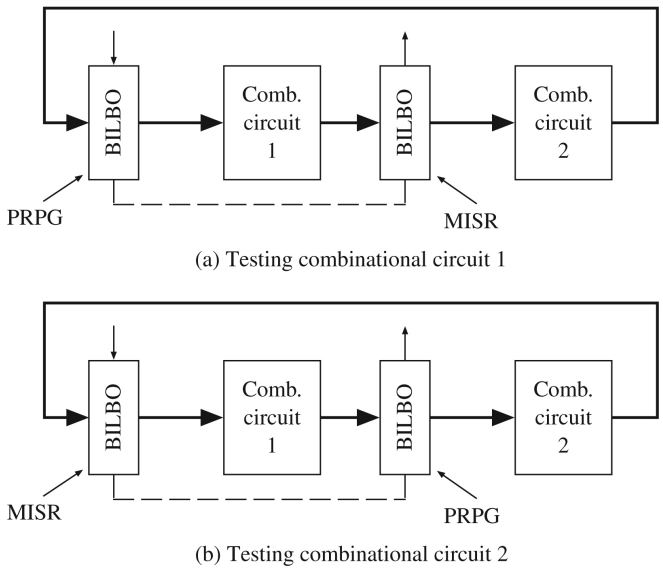


Figure 27: BIST using BILBO architecture (Figure 10-30 from text)

FIGURE 10-31:
Four-Bit BILBO Register

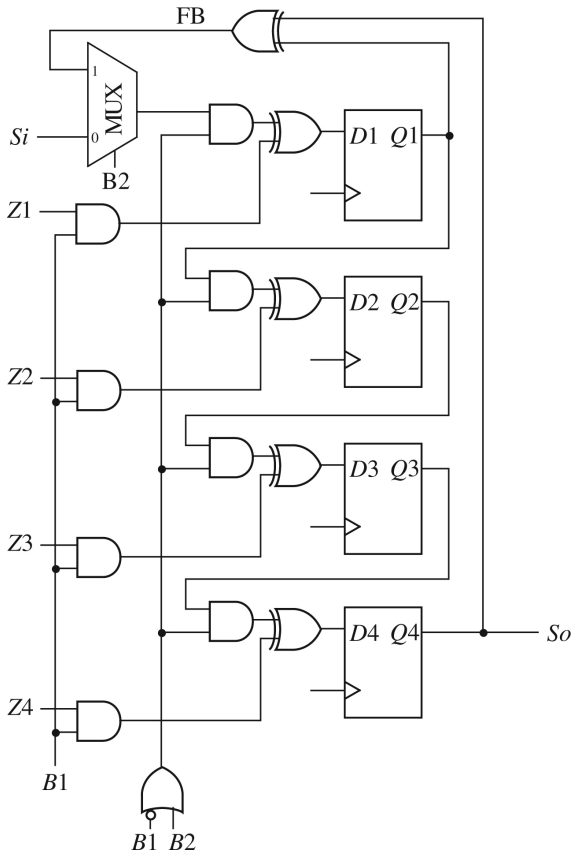


Figure 28: 4-bit BILBO register (Figure 10-31 from text)

FIGURE 10-33:
System with BILBO Registers and Tester

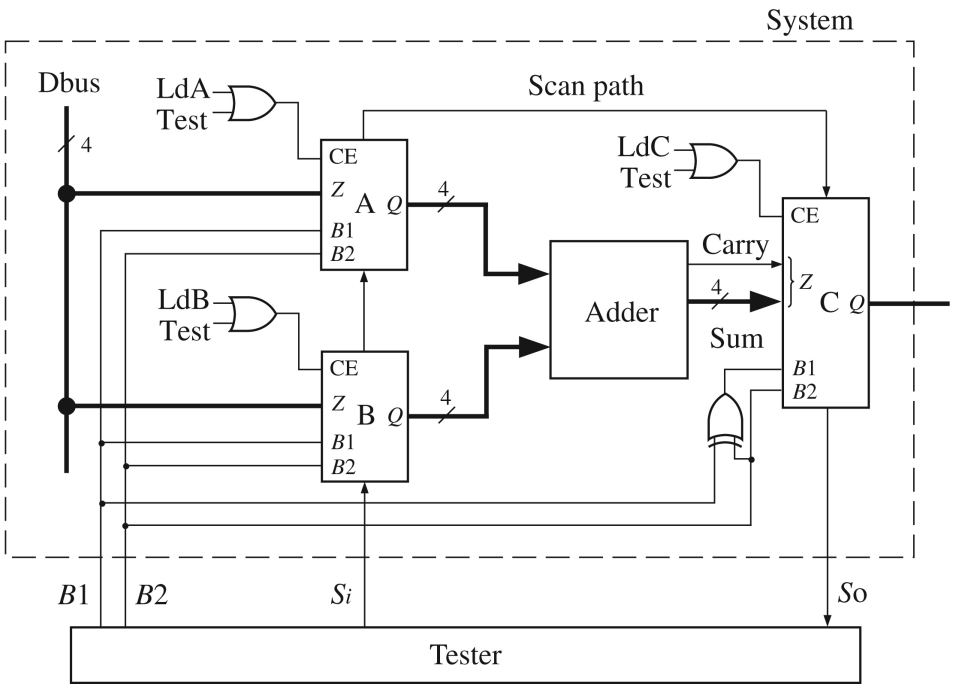


Figure 29: System with BILBO registers and tester (Figure 10-33 from text)