Center for Reliable Computing

TECHNICAL REPORT

Checking Experiments For Scan Chain Latches and Flip-Flops

Samy Makar

(CSL TR # =

July 1996=

Center for Reliable Computing
Gates Room # 235, MC 9020
Gates Building 2A
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract:

This technical report contains the text of Samy Makar's thesis "Checking Experiments for Scan Chain Latches and Flip-Flops".

Funding:

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant No. N00014-92-J-1782, in part by the Advanced Research Projects Agency under Contract No. DABT63-94-C-0045, and in part by the National Science Foundation under Grant No. MIP-9107760. It was also funded in part by Cirrus Logic.

ABSTRACT

New digital designs often include scan chains; high quality economical test is the reason. A scan chain allows easy access to internal combinational logic by converting bistable elements, latches and flip-flops, into a shift register. Test patterns are scanned in, applied to the internal circuitry, and the results are scanned out for comparison. While many techniques exist for testing the combinational circuitry, little attention has been paid to testing the bistable elements themselves. The bistable elements are typically tested by shifting in a sequence of zeroes and ones. This test can miss many defects inside the bistable elements. A checking experiment is a sequence of inputs and outputs that contains enough information to extract the functionality of the circuit. A new approach, based on such sequences, can significantly reduce the number of defects missed. Simulation results show that as many as 20 percent of the faults in bistable elements can be missed by typical tests; essentially all of these missed faults are detected by checking experiments. Since the checking experiment is a functional test, it is independent of the implementation of the bistable element. This is especially useful since designers often use different implementations of bistable elements to optimize their circuits for area and performance. Another benefit of a functional test is that it avoids the need for generating test patterns at the transistor level.

Applying a complete checking experiment to a bistable element embedded inside a circuit can be very difficult, if not impossible. The new approach breaks up the checking experiment into a set of small sub-sequences. For each of these sub-sequences a test pattern is generated. These test patterns are scanned in, as in the case of the tests for combinational logic, appropriate changes to the control inputs of the bistable elements are applied, and the results are scanned out. The process of generating the patterns is automated by modifying an existing stuck-at test generator. A designer or test engineer need only provide a gate level description of the circuit to generate tests that guarantee a checking experiment for each bistable element in the design.

Test size is an important economic factor in circuit design. The size of the checking-experiment-based test increases with circuit size at about the same rate as the traditional test, indicating that it is practical for large circuits. Checking-experiment-based tests are an effective economic means for testing the bistable elements in scan chain designs.

ACKNOWLEDGMENT

I am deeply grateful to my advisor, Prof. Edward J. McCluskey, for his constant guidance, support, and encouragement throughout my years at Stanford. I would like to thank Prof. John Gill, my associate advisor, and Prof. Leonard Tyler, my committee chairman, for reading my dissertation, and Prof. Mike Flynn for agreeing to be the fourth member of my committee.

I would like to thank my colleagues at the Center for Reliable Computing for interesting discussions, helpful comments, and companionship through the years: Khader Abdel-Hafez, LaNae Avra, Jonathan Chang, Hong Hao, Rajesh Kamath, Erin Kan, Wern-Yan Koe, Sunil Kosalge, Siyad Ma, Samiha Mourad, Shridhar Mukund, Rob Norwood, Nirmal Saxena, Philip Shirvani, Alice Tokarnia, Nur Touba and Sanjay Wattal. I also thank Siegrid Munda for her administrative support.

I wish to thank other friends at Stanford for their support over the years: Edgar Holmann, Margaret Martonosi, Tony Todesco and Josep Torrelas. I would also like to thank all members of the Stanford Ballroom Dance Club for all the fun during my years at Stanford. Thanks also go to Judith Shimizu, for reading the many versions of this thesis, and other publications, and for preparing the wonderful food for my oral presentation, and to Sanjay Wattal for his valuable comments on reading this thesis. Special thanks also go to Suhas Patil, founder and chairman of Cirrus Logic Inc., for encouraging me to complete my degree after joining Cirrus Logic.

I am very grateful my parents for their continued support and encouragement during the pursuit of not just my PhD, but to all my education. I dedicate this thesis to them. I also thank my brother and sisters, Rosemary, Ramsey and Hoda, for their continuing encouragement and support.

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant No. N00014-92-J-1782, in part by the Advanced Research Projects Agency under Contract No. DABT63-94-C-0045, and in part by the National Science Foundation under Grant No. MIP-9107760. It was also funded in part by Cirrus Logic.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. Introduction

Chips are tested to ensure that defective ones are not shipped to customers. This is important for economic reasons. Using a defective chip in a system may render the entire system defective. In some cases, such as space equipment, it may not be even possible to replace a defective chip once it is operational. Thus, it is important to test chips before they are used, and it is important to test them thoroughly. The test process is illustrated in Fig. 1-1. A chip is placed on the tester, and a test is applied to it. Only chips that pass the manufacturer's test are shipped to the customer. Failed chips are sometimes analyzed to identify manufacturing process problems [Miczo 86]. Depending on how good the test is, some of the chips sent to the customer may be defective. A more thorough test would minimize the number of defective chips shipped to the customer.

A chip is faulty if it contains one or more defects. *Test generation* is the process of producing the test patterns that will identify chips with defects.

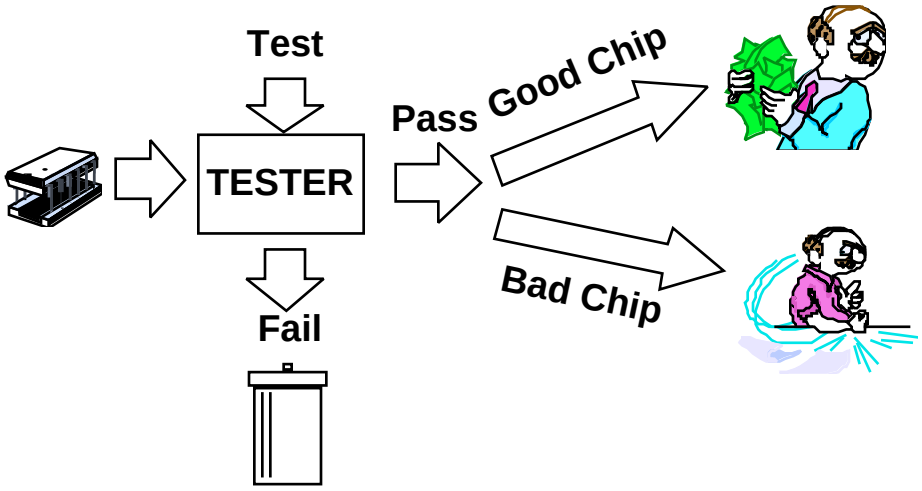
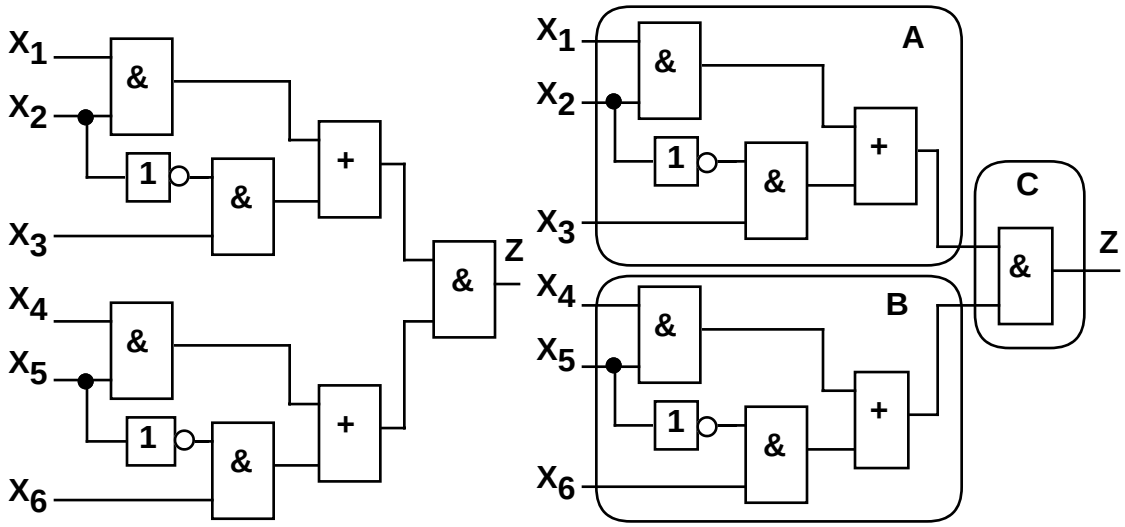


Figure 1-1 Test Process.

Test generation depends on the type of circuit being tested. Digital circuits are of two types: combinational and sequential. In combinational circuits the outputs depend only on the present inputs; outputs of sequential circuits depend on past as well as present inputs. There are many tests that can be used for combinational logic. An *exhaustive test* applies all possible patterns to the inputs of the circuit. This approach guarantees that any defect that would change the functionality of the combinational logic, without making it sequential, is detected. However, due to the large number of patterns needed by this technique, other approaches have been used. In pseudo-exhaustive test [McCluskey 86], the circuit is partitioned into smaller blocks, and each of the blocks is tested exhaustively.



(a) Circuit Under Test **(b) Partitioned Circuit**

Figure 1-2 Example of Exhaustive and Pseudo-Exhaustive Test.

For example, in Fig. 1-2a, a small circuit with 6 inputs is shown. An exhaustive test requires 64 (2^6) patterns. With the partitions shown in Fig. 1-2b, only 20 patterns are needed, 8 (2^3) for partition A and 8 (2^3) for partition B, and 4 (2^2) for partition C.

Another approach to reduce the size of an exhaustive test is pseudo-random test [Savir and Bardell 84; Williams 85; Chin and McCluskey 87; McCluskey et. al. 88]. In this test, non-repeating and reproducible random patterns are applied. The fact that the patterns are non-repeating and reproducible makes them not true random patterns. The non-repeating nature has two advantages over true random patterns: saving time since repeating a pattern won't detect more defects, and a small circuit can be used for implementing it. The reproducibility of the patterns makes it possible to “fault grade” the patterns. Typically, a linear feedback shift register (LFSR) is used to generate pseudo-random patterns [McCluskey 86]. An example of such a circuit is shown in Fig. 1-3.

A more direct approach to test generation is to generate patterns for the defects expected in the circuit. However, the difficulty of generating patterns from defects, and the

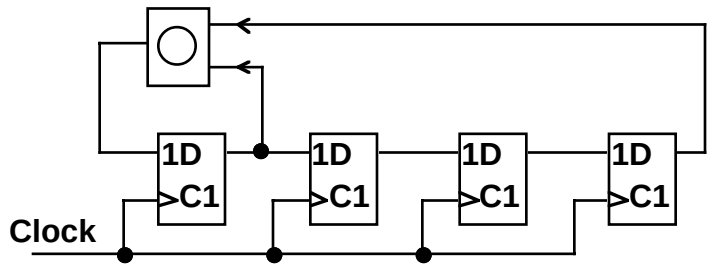


Figure 1-3 Example of LFSR (Linear Feedback Shift Register).

continuing change of technology used for digital design, has lead to the use of fault models as an abstraction of defects. The most commonly used fault model is the *single-stuck fault* model [Abramovici 90]. In this model, one line in the circuit is assumed to be always stuck at 0 or 1. The goal of test generation is to find a pattern that would produce different output when applied to a good and faulty circuit. For example, in the circuit of Fig. 1-4, suppose we need to find a test for line *a* stuck-at 1. First, the input pattern should set line *a* to 0 so that the faulty and fault-free circuits have different behavior. The good circuit will have 0 while the faulty circuit will have 1 due to the fault. Setting line *a* to 0 requires either $X_1 = 0$ or $X_2 = 0$. We also have to make sure that the error in the faulty circuit (i.e., the 1 instead of 0) appears at the primary output (Z). If X_3 is set to 1, then the output of both the good and faulty circuits would be 1 and we will not detect the fault. By setting X_3 to 0 the output will depend on the value at line *a.* For the good circuit we should see 0 at the output, and for the faulty circuit we should see 1. Therefore, a test for line *a* stuck-at 1 is $X_1X_2X_3$ = -00 or 0-0. A "-" in a pattern means either 0 or 1 would work.

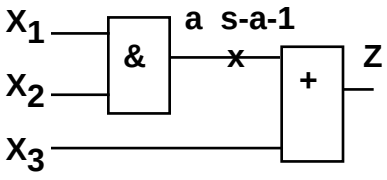


Figure 1-4 Example of Test Generation Using Stuck-At Fault Model.

The problem becomes more complicated when we have reconvergent fanout. For example, consider the circuit in Fig. 1-5. As in the previous example, we want a test for line *a* stuck-at 1. We need to set line *a* to 0, which can be done with $X_1 = 0$ or $X_2 = 0$. Suppose we select X_2 to be 0. This would imply that line *b* will be 1. The output of the good and faulty circuit would then be 1, and we would not detect the fault. So, we need to use $X_1 = 0$. We also need line *b* to be 0 so that the effect of the fault appears at the primary output. This requires $X_2 = 1$ and $X_3 = 0$.

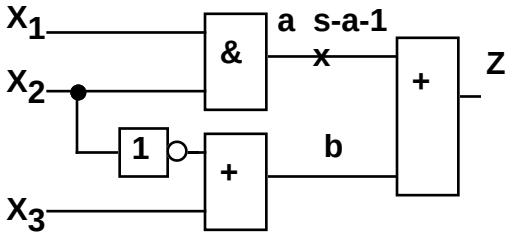


Figure 1-5 Example of Test Generation for Circuit With Reconvergent Fanout.

There are several algorithms for generating patterns for combinational circuits based on the single-stuck fault model. The boolean difference is an algebraic method that finds an equation representing tests for a fault [Sellers et. al. 68]. The difficulty of algebraic methods to handle large circuits efficiently has given way to structural methods. One algebraic approach that has proven to be successful for large circuits is boolean satisfiability [Larrabee 89]. This method takes advantage of the fact that only one pattern is really needed to satisfy the boolean equation generated and does not attempt to reduce the algebraic expressions. The first of the structural methods, the D-Algorithm [Roth 66] assigns a D to the faulty line in the circuits, and propagates its effect towards the output, while "justifying" the values on gates that drive the line, similar to what we did in the above example. In the search for a pattern, there are often many choices. In the above example, we had a choice of which AND gate input to set to 0. Our first choice resulted in a conflict with another part of the test. In this case, the other choice was taken, and the process repeated. The process of canceling a choice, and trying another is called *backtracking*. The efficiency of a test generation algorithm depends heavily on the amount of backtracking required before a test is found. PODEM (path oriented decision analysis) was introduced to reduce the amount of backtracking encountered with the D-Algorithm [Goel 81] . In PODEM, values are assigned to the primary inputs successively, until either a pattern is found, or the fault is proven undetectable. An incorrect assignment at the primary input will cause backtracking. However, since there are generally fewer primary inputs than lines in the circuit, the number of backtracks are generally fewer. FAN (fanout oriented) further improved on PODEM by allowing assignments to some fanout nodes as well as primary inputs [Fujiwara and Shimono 83].

Generating a test for a sequential circuit is much more difficult than for a combinational one, because the output response depends not just on the input but on a sequence of inputs. Mourad [90] gives an excellent survey of the different approaches for sequential test generation. There have been a few attempts ([Kubuo 68; Putzolu and Roth 71; Muth 76]) at extending the combinational test generation algorithms to handle sequential circuits. The basic idea is to replicate the combinational logic multiple times, treating each flip-flop output as a pseudo-input, and flip-flop inputs as pseudo-outputs. For example, consider the circuit in Fig. 1-6. Again we want to find a test for line *a* stuck-at 1. Fig. 1-7 shows the combinational circuit replicated twice. The notation used here is taken from Fujiwara's book [85]. The combinational logic is the same as that of Fig. 1-5, so we would like to use the same pattern $X_1X_2X_3 = 010$. However, instead of X_2 as a primary input we have a bistable element. Therefore, starting in time frame 2, we need $X_1(2)y(2)X_3(2) = 010$. This implies that we need $Y(1) = 1$. $Y(1)$ can be set to 1 by

setting $X_3(1)$ to 1. So, our test requires two consecutive patterns. The first one X_1X_3 = -1 will make the flip-flop output become 1, and the second pattern X_1X_3 = 00 will detect our fault as before. In this example, we only needed two time frames. In general, many time frames could be needed.

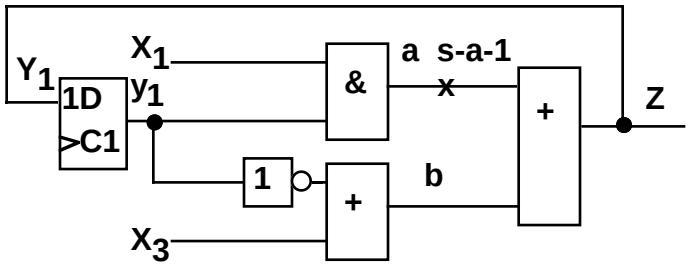


Figure 1-6 Example of Test Generation for Sequential Circuits.

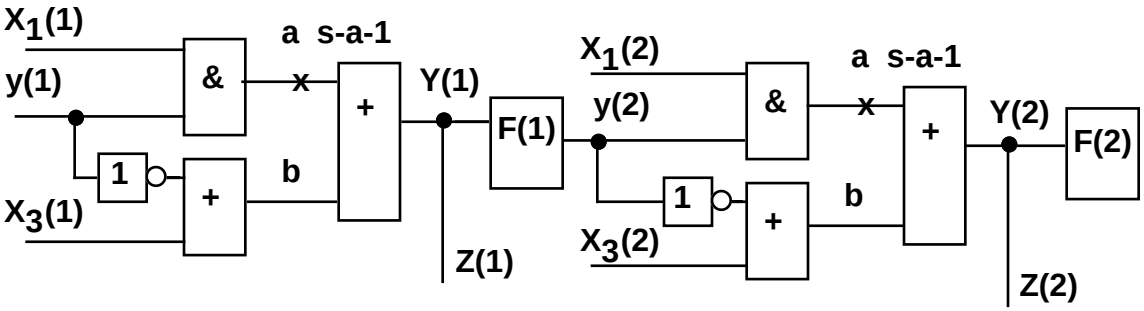


Figure 1-7 Replicating Combinational Logic for Sequential Test Generation.

Another approach for testing sequential circuits is to use the state table. Here, a functional description of a sequential circuit is used, rather than the circuit, to find a test. For example, Table 1-1 shows the state table for the circuit in Fig. 1-6. The idea is to find a sequence of inputs that when applied to the circuit would give different output than any other circuit with the same input, outputs, and same number of or fewer states. Such a sequence, a *checking experiment*, contains enough information to reconstruct the state table from the observed output when the sequence is applied. The sequence in Table 1-2 is a checking experiment for the state machine described in Table 1-1. We can prove this by deriving the state table from the sequence. Looking at patterns 2 and 3 of this sequence, we see that the circuit has different outputs when X_1X_3 = 00 is applied. This indicates that there must be two states in the state machine. We call these states A and B and create Table 1-3. States are shown slightly offset from the pattern in Table 1-2, to show the states before and after a pattern has been applied. After pattern 2, we are in state A and after pattern 3 we are in state B. From patterns 2, 3 and 4 we see that applying X_1X_3 = 00 will

switch between the two states. We can thus fill the first column of Table 1-3. Since we assume that there must be at most two states in the circuit, $X_1X_3 = 00$ can be used as a distinguishing sequence. A *distinguishing sequence* is a sequence that gives a different output sequence for each state. In our example if we apply X $X_3 = 00$ and see a 1 at the output then we were in state A before applying the distinguishing sequence, and if we see a 0 then we were in state B. In Table 1-2, states followed by the distinguishing sequence $X_1X_3 = 00$ are highlighted. In pattern 5, we have $X_1X_3 = 01$, and see an output of 1. Since we were in state A before this pattern was applied, we can put 1 as the output value in the entry marked (1) in Table 1-3. The pattern is followed by the distinguishing sequences ($X_1X_3 = 00$) with an output of 0. This implies that the next state in entry (1) should be B. The rest of the table is filled in the same way. If we map state A to 0 and state B to 1, we get the same table as Table 1-1, proving that the sequence in Table 1-2 really is a checking experiment. A systematic approach for generating a checking experiment is given in Hennie [64] and Friedman and Menon [71]. The main problem with checking experiments is that the number of patterns can be very large for even small circuits, making it impractical for real designs.

Table 1-1 State Table for Circuit in Fig. 1-6.

	X_1X_3			
y_1	00	01	11	10
0	1,1	1,1	1,1	1,1
1	0,0	1,1	1,1	1,1

Y_1Z

Table 1-2 Checking Experiment for State Table of Table 1-1.

Pattern	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
X_1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0
X_3	1	0	0	0	1	0	0	1	0	1	0	0	1	0	0	0	0	0	0
Z	1	0	1	0	1	0	1	1	0	1	0	1	1	0	1	0	1	1	0
State	B	A	B	A	B	A	B	B	A	B	A	B	B	A	B	A	B	B	
Position					1			2			3		4			5		6	

Table 1-3 State Table From Checking Experiment in Table 1-2.

	X_1X_3			
	00	01	11	10
A	B,1	B,1 (1)	B,1 (3)	B,1 (5)
B	A,0	B,1 (2)	B,1 (4)	B,1 (6)

Scan was introduced to overcome the difficulties of sequential test generation described above [Williams and Angel 73; Eichelberger and Williams 77]. The basic idea of scan is to allow easy access to the flip-flops in the design so that patterns can be applied directly to the inputs of the internal combinational logic, and the outputs of the internal combinational logic can be observed from the primary output. This makes it possible to use the methods discussed for combinational circuits on sequential ones. There are several architectures for scan designs [McCluskey 86]. These are shown in Figs. 1-8 through 1-11. There are two modes of operation in any scan design: shift mode and normal mode. In *shift mode*, the scan chain is configured as a shift register to scan data in and out of the bistable elements. In *normal mode*, the bistable elements are configured to get their inputs from the combinational logic and perform normal functional operation of the circuit. In the MD-latch and MD flip-flop architectures, the shift mode is activated by setting T = 1, and normal mode is activated by setting T = 0. In the other two architectures, TCK is used as the clock for shift mode, and CK_1 is used for normal mode.

The combinational logic is tested by scanning in a pattern with the scan chain in shift mode. Values at the primary input are then applied, and values at the primary output are checked. The scan chain is switched to normal mode for one cycle to capture the output of the combinational logic in the bistable elements. Then the scan chain is switched back to shift mode, and the contents of the bistable elements are scanned out and checked at SDO. The next pattern is scanned in while the bistable element contents are being scanned out. The bistable elements themselves are tested by shifting a pattern that applies all four transitions (0->0, 0->1, 1->1 and 1->0) to the bistable elements. An example of such a pattern is 00110.

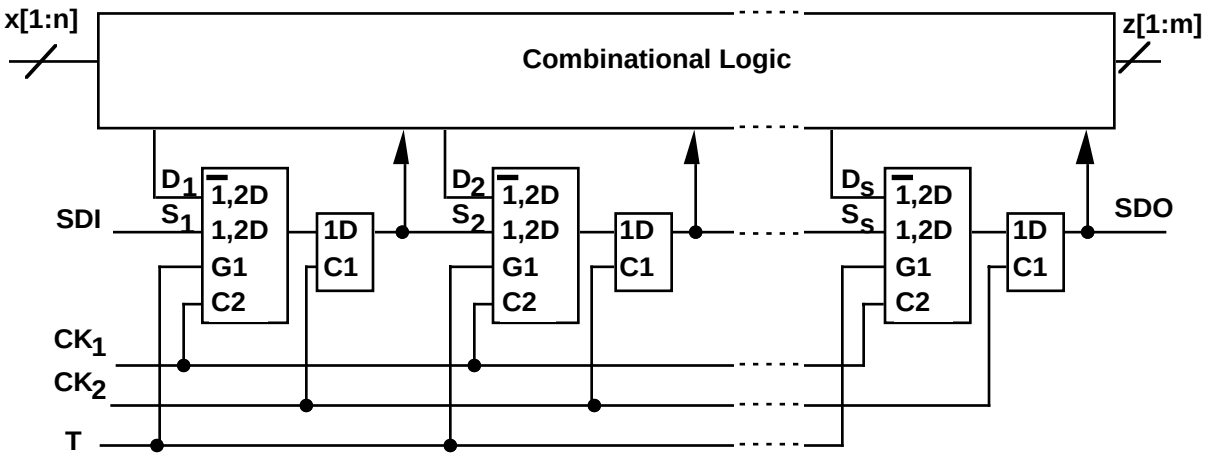


Figure 1-8 MD-Latch Based Scan Architecture.

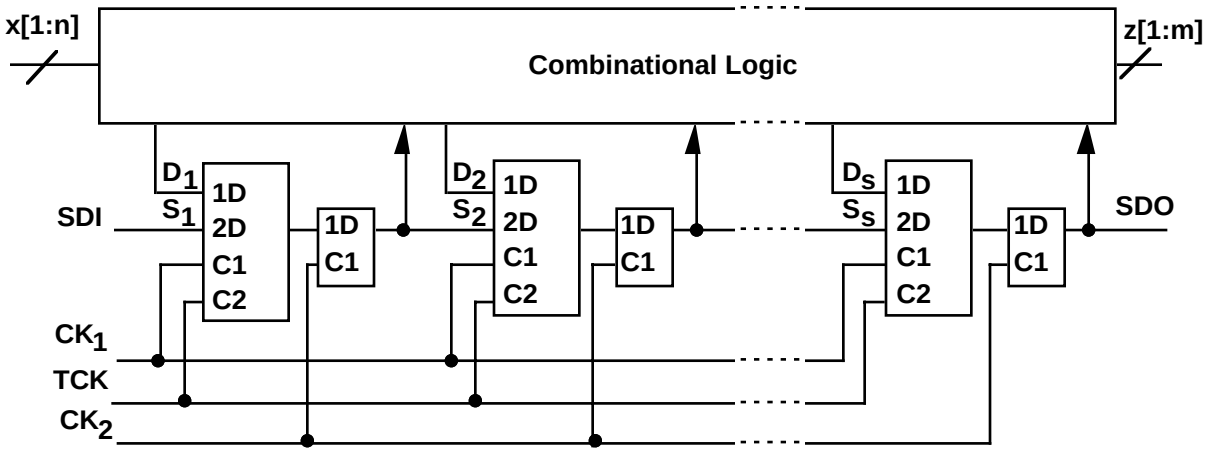


Figure 1-9 Level Sensitive Scan Design (LSSD) Architecture.

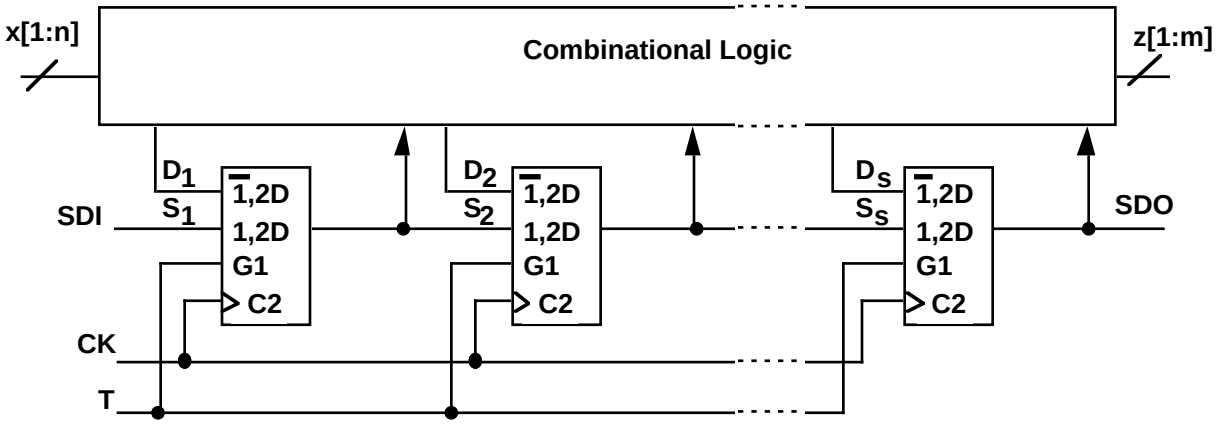


Figure 1-10 MD Flip-Flop Based Scan Architecture.

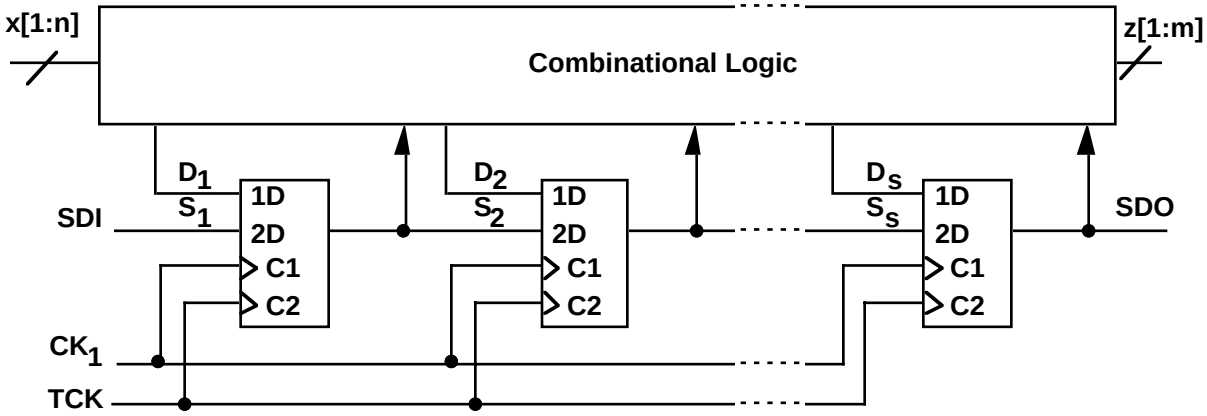


Figure 1-11 TP Flip-Flop Based Scan Architecture.

The problem with this method of testing the bistable elements is that it does not address faults within the bistable elements. Experimental results shown in Chapter 5 of this thesis show that this can be a serious problem. However, this is not the first work to show that faults inside bistable elements can be missed by traditional scan tests. Reddy and Reddy [86] derived tests for stuck-open faults in different latch implementations. Lee and Breuer [90] analyzed bridging faults in scan registers, and combined the use of current and voltage tests. Al-Assadi [93] mapped many, but not all, of the internal faults to functional fault models. He also showed that some of the internal faults cannot be mapped to functional fault models.

What is missing from the previous work is an implementation independent test that will detect all the faults in latches and flip-flops, and a method to automatically generate such a test for bistable elements used in scan chains. This is where the work in this thesis begins. We combine the work of checking experiments mentioned earlier, with the work on scan and structural test generation to create a new algorithm that can generate tests for the latches and flip-flops in the scan chain. As mentioned earlier, checking experiments were not practical for real designs. However, we use a checking experiment only for the bistable elements themselves. From the requirements of the checking experiment for the bistable element, an algorithm that generates test patterns was developed. These patterns are scanned in as in the case of tests for the combinational logic, but rather than switching to normal mode for one cycle, the circuit is kept in normal mode for several cycles depending on the type of pattern. This will be discussed in detail in Chapter 4.

There are two types of bistable elements: latches and flip-flops. In latches, when an input value is changed, any effect on the output appears right after the input changes. This property is often called the *transparency property* of latches [McCluskey 86]. Chapter 2 describes the method of generating checking experiments for two-state latches, and shows how checking experiments can be applied to latches embedded in shift registers or scan chains. The other type of bistable elements, flip-flops, do not have the transparency property of the latch. A flip-flop output changes only in response to transition on a control input or a change in an asynchronous input [McCluskey 86]. Chapter 3 describes the method of generating checking experiments for flip-flops, and shows how checking experiments can be applied to flip-flops embedded in shift registers or scan chains. Chapter 4 describes the new test generation algorithm that will apply checking experiments to all the bistable elements in the circuit. The algorithm was implemented in C by modifying an existing combinational test generation program. The reason for selecting an existing test generation program was to show that this new algorithm can be easily implemented by enhancing existing commercial test generation program for real designs.

The effectiveness of the new method can be measured by fault simulating a circuit. The results of fault simulation of individual bistable elements presented in Chapter 5. In Chapter 6 we fault simulate a three-bit binary counter using traditional test patterns and patterns for an MD flip-flop architecture. The results show that many faults are missed by the traditional test. In the same chapter we generate patterns for the ISCAS-89 benchmarks [Brglez et. al. 89]. The number of patterns generated for these circuits indicate that our algorithm will generate a reasonable number of test patterns for real circuits.

Contributions

Contributions to knowledge, described in this thesis, are summarized as follows:

1. Development of a new approach for generating test patterns for bistable elements in a scan chain design. The test patterns guarantee the detection of all faults that do not increase the number of states of the bistable element.
2. Implementation of this new approach by modifying an existing stuck-at automatic test pattern generation tool. The implementation was run on all the ISCAS 89 benchmarks, and the results indicate that this new approach is practical for large circuits.
3. Introduction of a new method for easily deriving checking experiments for two-state latches, and derivation of a lower bounds on the number of patterns needed for checking experiments of two-state latches.
4. Introduction of a new method for deriving checking experiments for bistable elements that are embedded in a circuit, taking into account controllability of the inputs and observability of the outputs.
5. Derivation of simple tests for bistable elements in shift registers and demonstration that any data sequence that applies all four transitions will apply a checking experiment to a double-rank shift register.

Chapter 2. Checking Experiments for Latches

Latches are memory elements that have the *transparency property*. When an asynchronous input of a latch is changed, any effect on the output appears shortly after the input changes, and when a synchronous input is changed, the output changes shortly after the input changes if the control input is active [McCluskey 86]. This chapter begins with a summary of various latches and checking experiments for them in Section 2.1. The theoretical details for generating checking experiments for two-state latches are given in Section 2.2. Checking experiments for some of the latches are derived in the same section, and checking experiments for the rest are derived in Makar and McCluskey [94]. As most circuits consist of more than just a single latch, we show what happens to a D-latch once it is used in a shift register in Section 2.3. In that section, we show that any data sequence, such as 01100, that applies all four transitions (0->0, 0->1, 1->0 and 1->1) will apply a checking experiment to all the latches in the shift register. In Section 2.4 we apply the same technique of Section 2.3 to MD-latches and TP-latches used in scan chains. We show that there is no simple test that can be applied, and present a technique to generate patterns to test the latches in the scan chain.

2.1 Latches and Their Minimum-Length Checking Experiments

Various latch types are discussed in this section (see Table 2.1-1). The simplest latch type is the SR-latch. An *SR-latch* is a sequential element that can be set or reset by activating the appropriate input. Even though the SR-latch is still occasionally used, the most commonly used latch today is the D-latch. A *D-latch* is a sequential element, in which the data input is propagated to the output when the clock is active, otherwise it holds the stored value. A *D-latch with Asynchronous Set/Reset* is a D-latch that can be set or reset when the clock is not active. Scan-paths require latches with two different data sources. These can be either Multiplexed-Data latches or Two-Port latches. A *Multiplexed-Data latch* (*MD-latch*) is a D-latch with multiplexed data inputs; a *Two-Port latch (TP-latch)* has two control inputs with the data source determined by the active control input [McCluskey 86]. A *Load Enable latch* is a D-latch with a gated control input, and a *D-Enable latch* is a D-latch with gated data. An *XOR Input latch* performs an exclusive-or operation on its two data inputs. This latch is commonly used in an LFSR (linear feedback shift register) to generate pseudo-random vectors, and to compress results. Other latches commonly used for BIST (built-in self test) are the *Built-In Logic Block Observer latch (BILBO latch)*, and the *Concurrent Built-In Logic Block Observer latches (CBILBO latches).* The BILBO latch has two data inputs. It can be configured to

load either of the two inputs (one a scan input, and the other for normal operation), load the exclusive-or of the two inputs (for signature analysis), or load 0. The CBILBO latches, an extension of the BILBO latch, are two latches that can operate simultaneously as a pseudo-random pattern generator and a signature analyzer. The two latches are treated separately, with outputs Q_1 and Q_2. Table 2.1-1 shows the excitation function of each of these latches and the minimum-length of a checking experiment. Minimum-length checking experiments for each of these latches are shown in Tables 2.1-2 through 2.1-11. Details for deriving checking experiments for some of the latches are given in Section 2.2. Details for the rest are given in Makar and McCluskey [94].

Table 2.1-1 Latches and Their Excitation Functions.

Latch Type	Excitation Function	Assumptions	M*
SR-Latch	$Q = S + \overline{R}q$	SR = 0	6
D-Latch	$Q = CD + \overline{C}q$		7
D-Latch with Asynchronous Set/Reset	$Q = \overline{R}\,(S + CD + \overline{C}q)$	SR = 0	14
MD-Latch	$Q = C\,(TS + \overline{T}D) + \overline{C}\,q$		26
Two-Port Latch	$Q = C_1D_1 + C_2D_2 + \overline{C_1}\,\overline{C_2}q$	$C_1C_2 = 0$	23
Load Enable Latch	$Q = CLD + \overline{(LC)}q$		15
D-Enable Latch	$Q = CDE + \overline{C}q$		16
XOR Input Latch	$Q = C\,(D \oplus S) + \overline{C}q$		13
BILBO Latch	$Q = C\,(B_1D \oplus \overline{B}_2S) + \overline{C}\,q$		58
CBILBO Latch	$Q_1 = C\,(\overline{B}_1D \oplus S) + \overline{C}\,q_1$ $Q_2 = C\,(B_2S + \overline{B}_2D) + \overline{C}\,q_2$		25 26

*M - minimum length (number of test vectors) of checking experiment.

Table 2.1-2 A Minimum-Length (6) Checking Experiment for an SR-Latch.

S	0	0	1	0	0	0
R	1	0	0	0	1	0
Q	0	0	1	1	0	0

Table 2.1-3 A Minimum-Length (7) Checking Experiment for a D-Latch.

C	1	1	0	0	1	0	0
D	1	0	0	1	1	1	0
Q	1	0	0	0	1	1	1

Table 2.1-4 A Minimum-Length (14) Checking Experiment for an Asynchronous Set/Reset Latch.

C	1	1	0	0	0	0	0	0	0	0	0	0	1	0
D	1	0	0	0	0	0	0	1	1	1	1	1	1	1
R	0	0	0	0	0	1	0	0	0	0	1	0	0	0
S	0	0	0	1	0	0	0	0	1	0	0	0	0	0
Q	1	0	0	1	1	0	0	0	1	1	0	0	1	1

Table 2.1-5 A Minimum-Length (26) Checking Experiment for an MD-Latch.

T	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
C	1	1	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0
D	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
S	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0
Q	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	1	0	0

Table 2.1-6 A Minimum-Length (23) Checking Experiment for a TP-Latch.

C_1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0
D_1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1
C_2	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
D_2	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
Q	1	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1

Table 2.1-7 A Minimum-Length (15) Checking Experiment for a Load Enable Latch.

L	1	1	0	0	1	1	0	0	1	0	0	1	1	0	0
D	1	0	0	0	0	1	1	1	1	1	1	1	0	0	0
C	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1
Q	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1

Table 2.1-8 A Minimum-Length (16) Checking Experiment for a D-Enable Latch.

D	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0
E	1	1	1	1	1	0	0	1	1	1	0	0	1	0	0	0
C	1	1	0	0	1	1	0	0	1	0	0	0	0	0	1	0
Q	1	0	0	0	1	0	0	0	1	1	1	1	1	1	0	0

Table 2.1-9 A Minimum-Length (13) Checking Experiment for an XOR Input Latch.

D	0	0	0	0	0	0	1	1	1	1	1	1	0
S	1	0	0	1	1	1	1	1	1	0	0	0	0
C	1	1	0	0	1	0	0	1	0	0	1	0	0
Q	1	0	0	0	1	1	1	0	0	0	1	1	1

Table 2.1-10 A Minimum-Length (58) Checking Experiment for a BILBO Latch.

D	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1
S	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
B_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
B_1	0	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
C	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0
Q	1	0	0	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1
D	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0
S	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1
B_2	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
B_1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
C	0	1	0	0	1	0	0	1	0	0	1	1	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0
Q	1	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	1	1	1	1	0	0

Table 2.1-11a A Minimum-Length (25) Checking Experiment for a CBILBO Latch 1.

D	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0
S	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0
B_1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
C	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0
Q_1	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	1	0	0

Table 2.1-11b A Minimum-Length (26) Checking Experiment for a CBILBO Latch 2.

D	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
S	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0
B_2	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
C	1	1	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0
Q_2	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	1	0	0

2.2 Deriving Checking Experiments for Two-State Latches

All latches described in Table 2.1-1 are typically implemented as two-state latches; the flow tables of these latches have only two rows. In this section, special properties of two-state flow tables are analyzed and a method for generating checking experiments for them is developed. This method was used to generate the checking experiments in Tables 2.1-2 through 2.1-11.

Each cell in a flow table, a *total state*, corresponds to an assignment of values to the circuit inputs and internal states. A total state is an *unstable state* if it causes a change in internal state of the machine. A total state is a *stable state* if the next internal state is the same as the current internal state. In the flow table, a stable state is represented with a number in a circle if its output is 0 and with a number in a square if its output is 1. The numbers of the states start with 2 (0 and 1 are not used to avoid confusion with logic values). For some states the output value may not be observable due to design constraints

(more on this in Section 2.3). For states with observable output, the output value is included after the state. Some total states cannot be reached because of the single-input change restriction on fundamental mode circuits. Such total states are *unspecified states,* and are shown with "–" in the flow tables. A sequence *visits* a total state when the sequence applies the input of the total state while the machine is in the internal state of the total state. A total state is *identified* by a sequence if the sequence provides enough information to reconstruct the corresponding entry in the flow table.

For a state machine to have sequential behavior, there must be at least one column in the flow table with different outputs. Otherwise, the machine acts as pure combinational logic. The total states in a column of a two-state flow table with differing outputs are called *distinguishing states*, and the input is called a *distinguishing input*. A state machine must also have two columns that change the internal state, so that both internal states are reachable. Inputs of such columns are called *synchronizing inputs* because they force the machine into a known state. These definitions are shown in Fig. 2.2-1.

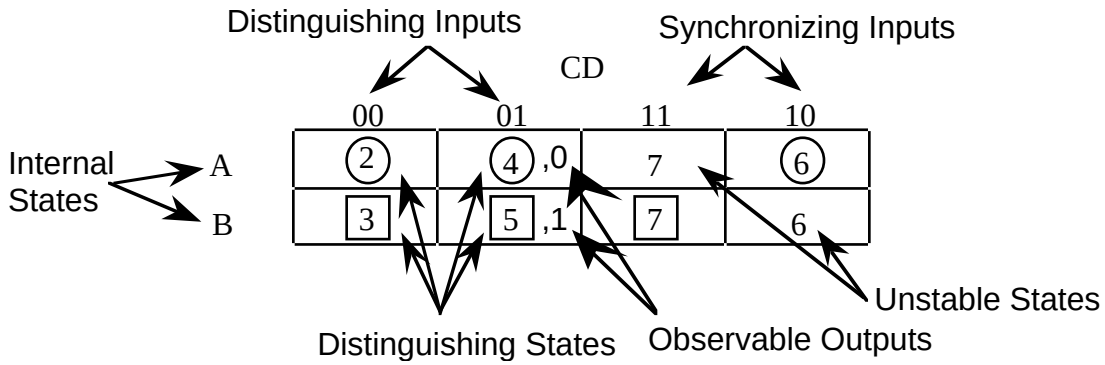


Figure 2.2-1 Definitions in Flow Table.

As mentioned earlier, a checking experiment contains enough information to reconstruct the flow table. Therefore, it must identify all total states in the flow table. In this section, we analyze two-state flow tables that have distinguishing inputs and synchronizing inputs only (i.e., no columns in the table have two stable states with the same output), because all latches studied here have flow tables that fall into this class of flow tables. The requirement for reconstructing the flow table can be refined into three simpler requirements: all total states must be visited, all unstable states must be identified, and all distinguishing states must be identified. If the flow table has columns with two stable non-distinguishing states (i.e., the two total states are stable and have the same output), then they will have to be identified too. Since none of the latch flow tables

we have seen contain such columns, we restrict our analysis to identifying distinguishing and unstable states.

We start with the first requirement, i.e., all total states must be visited. If a total state is not visited by the sequence, then the effect of applying the input of the total state when the machine is in the internal state of the total state is not known. This requirement is proven in Lemma 1.

Lemma 1: A checking experiment for a two-state flow table with only distinguishing and synchronizing inputs must visit all total states in the flow table.

Proof: Suppose that a sequence does not visit one of the stable total states. Create a second flow table by copying the original flow table and changing the output of the state not visited by the sequence. The output response of the sequence when applied to the second flow table would be the same as that of the original one because the sequence never visits the only total state that differs in the two flow tables. Since the two flow tables give the same response to the same input sequence, the sequence cannot be a checking experiment. Now suppose that the sequence does not visit one of the unstable states. In this case, the sequence would have the same response for a flow table that had the unstable total state replaced by a stable total state (the output does not matter). Since the input sequence has the same response for two flow tables, it cannot be a checking experiment. Therefore, a checking experiment must visit all total states (stable and unstable) in the flow table.

Even though visiting all the total states is a necessary requirement for a checking experiment, it is not a sufficient one. Consider the flow tables in Table 2.2-1. The sequence in Table 2.2-1c visits all the states of the flow table in Tables 2.2-1a. However, the output response is the same for the two flow tables in Table 2.2-1. Since the two flow tables are different, the sequence is not a checking experiment. The sequence did not identify the unstable states 8 and 9 in Table 2.2-1a. An unstable state is identified, when the sequence shows that the input caused a change in internal state. To show that an input causes a change in internal state, we need to show that the total states before and after the application of the input have different internal states. The internal state after the application of the input can be identified by following the input with a distinguishing input. The total state would then be a distinguishing state. The total state before the application of the input does not have to be a distinguishing state, but its internal state must be known. This analysis suggests the creation of state triples.

Table 2.2-1a Two-State Flow Table.

	CD							
	00	01	11	10	00	01	11	10
	A=0				A=1			
	(2),0	(4),0	7	(6),0	9	(8),0	–	–
	[3],1	[5],1	[7],1	6	[9],1	8	–	–

Table 2.2-1b Another Flow Table That Produces Same Output Sequence When Table 2.2-1c Sequence is Applied.

	CD							
	00	01	11	10	00	01	11	10
	A=0				A=1			
	(2),0	(4),0	7	(6),0	[9],1	(8),0	–	–
	[3],1	[5],1	[7],1	6	[11],1	(10),0	–	–

Table 2.2-1c Sequence That Visits All Total States, But is Not a Checking Experiment.

A	0	0	0	0	0	0	0	1	1	1
C	1	1	0	0	1	0	0	0	0	0
D	0	1	1	0	0	0	1	1	0	1
Q	0	1	1	1	0	0	0	0	1	0
State	6	7	5	3	6	2	4	8	9	8

A *state triple* is a set of three total states that contains a *setup state* (same internal state as unstable state and input unit distance from unstable state input), an unstable state, and a distinguishing state. The setup state and the distinguishing state must have different internal states. A sequence *visits a state triple* if it visits the setup state, then visits the unstable state right after the setup state, then visits the distinguishing state right after the unstable state. An unstable state can have many state triples, as there can be several candidates for setup states and for distinguishing states.

State triples are best described as graphs. For example, the graphs for the three unstable states in the flow table in Table 2.2-2 are shown in Fig. 2.2-2. Each of these graphs shows all the triples for one of the unstable states. In these graphs, any path represents one of the triples. For example, the first graph of Fig. 2.2-2 has four triples: 7,8,6; 7,8,2; 3,8,6 and 3,8,2. In the graphs of triples, setup states that are distinguishing states are shown on the left of the unstable state (in Fig. 2.2-2, states 7 and 3 for the first graph, state 2 for the second graph, and state 5 for the third graph). Setup states that are

not distinguishing states are shown on the top of the unstable state (in Fig. 2.2-2, state 10 for the second graph, and state 9 for the third graph). The unstable states are shown in the middle (in Fig. 2.2-2, state 8 for the first graph, state 9 for the second graph, and state 10 for the third). The distinguishing states of the triples are shown on the right (in Fig. 2.2-2, state 6 and state 2 for the first graph, state 3 for the second graph, and state 4 for the third graph). The triples for the unstable states in Table 2.2-1a are shown in Fig. 2.2-3.

Table 2.2-2 Flow Table Fragment Explaining State Triples.

	CD							
	00	01	11	10	00	01	11	10
	A=0				A=1			
	(2),0	(4),0	(6),0	(8),0	9	(10),0	–	–
	[3],1	[5],1	[7],1	8	[9],1	10	–	–

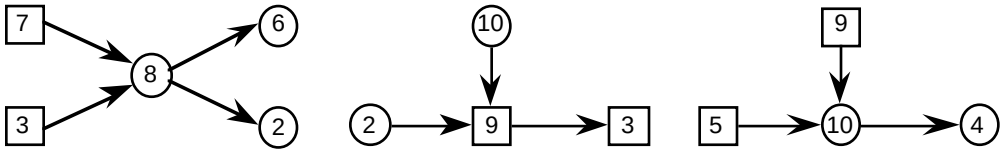


Figure 2.2-2 Graphs of State Triples for Flow Table in Table 2.2-2.

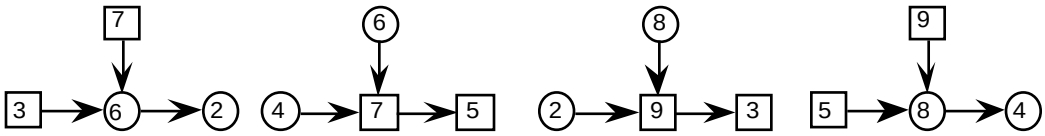


Figure 2.2-3 Graphs of State Triples for Flow Table in Table 2.2-1a.

Lemma 2: A checking experiment for a two-state flow table with only distinguishing and synchronizing inputs must visit at least one state triple of each unstable state to identify all unstable states. Triples can overlap, the distinguishing state of one triple can be the setup state of another triple.

Proof: We have already shown that a checking experiment must visit all the total states, including the unstable states. Visiting an unstable state requires visiting the setup state of a triple of the unstable state before visiting the unstable state itself. Therefore, a sequence that visits an unstable state *u* but does not identify it is not visiting a distinguishing state after visiting the unstable state (i.e., it only visits the first two states of the triple). In this case, the input applied by the sequence after visiting the unstable state *u* is a synchronizing input. Create a second flow table by copying the original flow table and changing the unstable state *u* to a

stable total state and give it the same output of the other total state in the same column. When the input corresponding to our unstable state is applied to either flow table, we get the same output. Since the next input is a synchronizing input, the next output and internal state will be the same for both flow tables. Therefore, a checking experiment must visit at least one state triple of each unstable state to identify all the unstable states.

Consider the flow tables in Table 2.2-3. Graphs for the state triples of Table

Table 2.2-3a Two-State Flow Table.

	CD							
	00	01	11	10	00	01	11	10
	A=0				A=1			
	(2),0	(4),0	7	(6),0	9	(8),0	–	–
	[3],1	[5],1	[7],1	6	[9],1	8	–	–

Table 2.2-3b Another Flow Table That Produces the Same Output Sequence When Table 2.2-3c Sequence is Applied.

	CD							
	00	01	11	10	00	01	11	10
	A=0				A=1			
	(2),0	[5],1	[7],1	(6),0	9	8	–	
	[3],1	(4),0	7	6	[9],1	(8),0	–	–

Table 2.2-3c Sequence That Visits All Total States, Identifies Unstable States, But is Not a Checking Experiment.

A	0	0	1	0	0	0	0	0	0	1	0
C	1	0	0	0	1	0	1	1	0	0	0
D	1	1	1	1	1	1	1	0	0	0	0
Q	1	1	0	0	1	1	1	0	0	1	1
State	7	5	8	4	7	5	7	6	2	9	3
Triples		A							D		
				B			C				

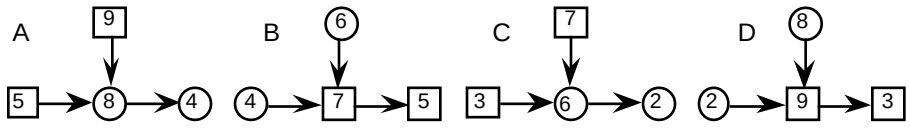


Figure 2.2-4 Graphs of State Triples for Flow Table in Table 2.2-3a.

2.2-3a are shown in Fig. 2.2-4. The sequence shown in Table 2.2-3c visits all the total states, and identifies all the unstable states of Table 2.2-3a. However, both flow tables in Tables 2.2-3a and 2.2-3b, produce the same output response for the input sequence of Table 2.2-2c.

The sequence in Table 2.2-3c produces different outputs for both 00 and 01, indicating that they are distinguishing inputs. However, there are two possible permutations (barring isomorphism) for the distinguishing states in the flow table. These are shown in the first two columns of Tables 2.2-3a and 2.2-3b. To distinguish between the two flow tables, a sequence must have CD = 00, 01 or CD = 01, 00 as sub-sequences.

This brings us to the third requirement: identifying the distinguishing states. As shown in this example, it is not enough to visit the distinguishing states. Two distinguishing inputs are said to be *linked* in a sequence, if the sequence provides enough information to determine the distinguishing states in their two columns. If a distinguishing input follows another distinguishing input in the sequence, then the two are linked.

There is another way that two states can be linked. Consider the flow table fragment in Table 2.2-4. The graph of the state triples for state 5 are shown in Fig. 2.2-5. Now suppose that a sequence visits a state triple that ends with state 3, then we know that the internal state of state 3 is the same as the internal state of stable state 5. If the sequence also visits a state triple that ends with state 7, then we know that the internal state of state 7 is the same as the internal state of stable state 5, and therefore the same as the internal state of state 3. Since state 3 and state 7 have the same internal state, their corresponding inputs are linked. Therefore, if an unstable state has several triples with different distinguishing states and a sequence visits triples with different distinguishing

Table 2.2-4 Flow Table Fragment Explaining Links.

	CD			
	00	01	11	10
	(2),0	5	(4),0	
	[3],1	[5],1	[7],1	

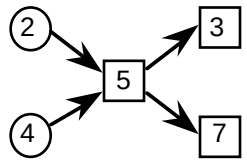


Figure 2.2-5 Graph of State Triples for Flow Table Fragment in Table 2.2-4.

states, then the distinguishing inputs corresponding to these distinguishing states are linked.

There can be many distinguishing inputs. Every distinguishing input can have one of two possible permutations of distinguishing states. Therefore, for n distinguishing inputs there are 2^n possible column permutations. Half of these permutations are nothing more than other permutations with the rows exchanged. Thus they do not need to be considered, and there are 2^{n-1} possible unique permutations. For a single distinguishing input, there is one unique permutation. Therefore, any distinguishing input is linked to itself, making the link relation reflexive. From the definition of link, link is a symmetric property. If *a* is linked to *b*, then *b* is linked to *a*. Now suppose that there are three distinguishing inputs *a, b* and *c*. If *a* is linked to *b*, then there is only one unique permutation for the distinguishing states in columns of *a* and *b*. Similarly, if *a* is linked to *c*, then there is only one unique permutation for the distinguishing states in columns of *a* and *c*. Therefore, there is one unique permutation for the all three columns, and so the link relationship is transitive. Since the link relationship is reflexive, symmetric, and transitive, it must be an equivalence relationship.

Lemma 3: A checking experiment for a two-state flow table with only distinguishing and synchronizing inputs must link all distinguishing inputs to identify all distinguishing states.

Proof: We have already shown that a checking experiment must visit all the total states, including the distinguishing states. Suppose that two distinguishing inputs are not linked in a sequence. Since link is an equivalence relation between distinguishing inputs, the distinguishing inputs would fall into two equivalence classes. Within each of the classes, there is only one unique permutation of distinguishing states. Create a second flow table by copying the original flow table, and swapping the rows in the distinguishing input columns of one of the equivalence classes. Also, swap the rows of any synchronizing inputs that are a unit distance from any of the distinguishing inputs in that class, if the synchronizing input is not a unit distance from a distinguishing input of the other class. Applying the sequence to the new flow table would give the same response as when applied to the original flow table. Therefore, if the distinguishing inputs are not linked in a sequence, the sequence is not a checking experiment.

Now that we have shown that the three conditions (visit all states, visit at least one state triple for each unstable state, and link all distinguishing states) are necessary for a

sequence to be a checking experiment, we show that if all three conditions are satisfied that the sequence is a checking experiment. In other words, given a two-state flow table with distinguishing and synchronizing inputs, any sequence that satisfies all three conditions is guaranteed to be a checking experiment. These conditions are necessary and sufficient. The proof is given in Theorem 1.

Theorem 1: A sequence for a two-state machine with distinguishing and synchronizing inputs is a checking experiment if and only if it satisfies the following properties:
1. Visits all the total states.
2. Visits at least one state triple for each unstable state.
3. Links all the distinguishing inputs.

Proof: From Lemmas 1, 2 and 3, a checking experiment must satisfy all the above conditions. Now, we need to show that if a sequence satisfies the three conditions, then it is a checking experiment. If a sequence links all the distinguishing inputs, then there can only be one permutation of distinguishing states in the flow table. If the sequence also visits at least one triple for each unstable state, then all entries in the flow table are identified. Therefore only one flow table can be constructed from the response of the sequence, making it a checking experiment.

An important consequence of Theorem 1 is that a checking experiment does not require all possible transitions in a two-state flow table. For example, consider the flow table in Table 2.2-5. The graphs of the triples are shown in Fig. 2.2-6. A sequence formed by combining the triples is 6,7,5,3,6,2. Since state 3 follows state 5, the distinguishing inputs are linked. State 4 can be added to the end of the sequence to satisfy the first requirement of Theorem 1. Therefore, the state sequence becomes:

Table 2.2-5 Flow Table Marked With Checking Sequence.

	CD 00	01	11	10	Q
A →	(2), 0	(4), 0	7	(6), 0	0
B ←	[3], 1	[5], 1	[7], 1	6	1

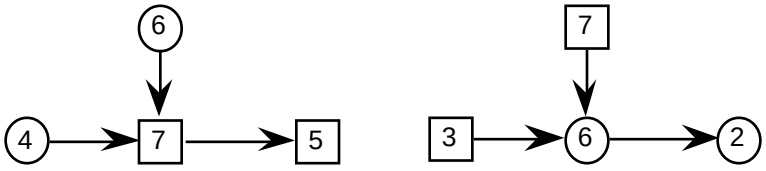


Figure 2.2-6 Graph of State Triples for Flow Table in Table 2.2-5.

6,7,5,3,6,2,4. The transitions through this sequence are shown graphically in Table 2.2-5. Thick arrows are used to indicate the beginning and end of the sequence. The following six transitions are not included in this sequence: 4->2, 2->6, 4->7, 3->5, 5->7 and 7->6.

The above example suggests the following procedure to generate a checking experiment for a two-state flow table with only distinguishing and synchronizing inputs. More examples for using this procedure are given in the following subsections.

Procedure for Deriving Checking Experiments from Two-State Flow Tables.

1. Determine all the state triples for the unstable states.
2. Select one triple for each unstable state.
3. Combine the triples of step 2. As in the example above, step 2 and 3 may be performed simultaneously (i.e., select the triples that best fit together), as long as a triple for each unstable state is used.
4. If the sequence resulting from step 3 does not form a link among all the distinguishing inputs then modify the sequence by adding extra states so that it forms a link among all distinguishing inputs. This should be done without destroying the triples.
5. Add any missing total states to the sequence.
6. Convert the state sequence into an input sequence, adding a synchronizing input, if necessary (i.e., if the first input of the first triple is not a synchronizing input).

The first pattern in the sequence for a two-state flow table should force the machine into a known state. Therefore, in step 2 of the procedure, the setup state of the first triple should correspond to a synchronizing input whenever possible.

Combining triples would be most efficient if the distinguishing state of one triple is the setup state of another. Since triples cause a change in the internal state, the final state of one triple can be the setup state of another if the two triples cause internal state changes in the opposite directions (i.e., if the first triple causes the machine to change from internal state A to internal state B, then the second triple should cause the machine to change from internal state B back to internal state A). If there are more triples that cause state changes in one direction than in the other, then some of the state changes will need to be repeated to get to the setup states of all the triples.

Lower Bound on Checking Experiment Length for Two-State Latches: The length of a checking experiment (L) is bounded by the following equation.

$$L \geq S + 1 + \sum_{i=0}^{S} \max(n_i - 1, 0) \quad \text{if } v = 0$$

$$L \geq S + v + \sum_{i=0}^{S} \max(n_i - 1,0) \quad \text{if } v > 0$$

where S = number of total stable states

n_i = number of unstable states for which s_i is the only distinguishing state of all its triples

v = difference between the number of unstable states in the two rows

Proof: As seen from Lemma 1, a checking experiment must visit every total state. A machine can only be in one stable total state for every input pattern. Therefore, there must be at least as many patterns in the sequence as there are total stable states. There will always be at least one extra pattern for initialization. To be useful, the first pattern should force the machine into a known state (a synchronizing input). If there are n_i unstable states that require s_i as the only distinguishing state of all their triples, then s_i must appear at least n_i times. One of the occurrences of state s_i is accounted for in S (the total number of stable states). Therefore s_i must appear an additional n_i - 1 times. If n_i is zero, then nothing should be added. Hence the term max(n_i - 1,0) is added for each total state. If the number of unstable states in one row differs from the number of unstable states in the other row by *v*, then at least *v* -1 extra internal state changes have to be applied. Each extra state change requires at least one more pattern. Adding *v* -1 to the initialization pattern gives *v*.

In many of the latch flow tables, half the inputs are distinguishing inputs, and the other half are synchronizing inputs. A single input variable determines whether an input is a distinguishing input or a synchronizing input. This class of state machines will be referred to as *single-input control state machines*, and the variable that determines the input type will be called a *control input*. For example, in Table 2.2-5, all inputs are distinguishing inputs when C = 0, and all inputs are synchronizing inputs when C = 1. Therefore, the flow table describes a single-input control state machine with C as the control input. An interesting property of such state machines is that the distinguishing state of one triple cannot be a setup state of another triple, because there are no unstable states adjacent to a distinguishing state of a triple. As before, if there are more triples that cause state changes in one direction than in the other, then some of the state changes will need to be repeated to get to the setup states of all the triples. In this case, since the distinguishing state of a triple is not a setup state of another triple, we will need two extra

inputs instead of just one. Therefore, if there are *v* more unstable states in one row than in the other, then 2(*v* -1) extra patterns are needed. This is used to derive a tighter lower bound on the length of the checking experiment.

Lower Bound on Checking Experiment Length for Single-Input Control State Machine: The length of a checking experiment (L) of a Single-Input Control State Machine is bounded by

$$L \geq S + 1 \qquad \text{if } v = 0$$

$$L \geq S + 2v - 1 \qquad \text{if } v > 0$$

where S = number of stable total states

v = difference between the number of unstable states in the two rows

Proof: In a single-input control state machine each distinguishing input is a unit distance from only one synchronizing input. This implies that no two triples can have the same distinguishing state. Therefore, the summation term in the original bound will always be 0. If v = 0, then the arguments for the previous bound apply. If v > 1, then two patterns are needed for each additional transition because the distinguishing state of a triple cannot be used as a setup state for another triple. Therefore 2(v -1) extra patterns are needed. Combining this with the initialization pattern gives 2v -1.

Another property of single-input control state machines is that a sequence that uses triples with distinguishing states as setup states except for the first triple, will identify the distinguishing states as well as the unstable states. This property is proved in Theorem 2.

Theorem 2: If a sequence is applied to a single-input control state machine, and the setup states of all but the first triple are distinguishing states, then the sequence links all the distinguishing inputs and is a checking experiment.

Proof: In a single-input control state machine each distinguishing input has a distinguishing state that is a setup state of a triple, and another distinguishing state that is a distinguishing state of the same triple. Therefore, there is a one-to-one correspondence between distinguishing inputs and triples. Now, if triple B is applied after triple A, the distinguishing input corresponding to triple A is linked to the one corresponding to triple B. Suppose triple C is applied after triple B, then

the distinguishing input corresponding to triple B is linked to the one corresponding to triple C. Since link is an equivalence relation, the distinguishing input corresponding to triple A is linked to the one corresponding to triple C. Using the same argument, it can be shown that the distinguishing input corresponding to triple A is linked to all the distinguishing inputs. Therefore, the sequence links all the distinguishing inputs, identifying all the stable states. Since the sequence identifies all the stable states and the unstable states, it is a checking experiment.

In the rest of Section 2.2, state triples and minimum-length checking experiments for SR-latch, D-latch, MD-latch, and TP-latch are derived. Triples and checking experiments for the other latches are presented in Makar and McCluskey [94].

2.2.1 Two-State SR-Latch

The equation for the SR-latch is $Q = S + \overline{R}q$. The latch is set when S = 1 and reset when R = 1. S and R should not be 1 at the same time (a *set dominant SR-latch* would be set if SR = 1, and a *reset dominant SR-latch* would be reset if SR = 1). The flow table for the SR-latch is shown in Table 2.2.1-1. The graphs of the state triples are shown in Fig. 2.2.1-1.

Table 2.2.1-1 Flow Table for Two-State SR-Latch.

	SR				Q
	00	01	11	10	
	(2),0	(4),0	-	5	0
	[3],1	4	-	[5],1	1

Figure 2.2.1-1 Graphs of State Triples for Two-State SR-Latch.

The setup state of one triple is the distinguishing state of the other triple. Therefore, the two triples can be combined without adding any states between them. Two possible state sequences are 2,5,3,4,2 and 3,4,2,5,3. Since there is only one distinguishing input, the distinguishing states are identified by simply visiting them. The two sequences contain all the total states in the flow table. The input sequence that would generate these state sequences is shown in Table 2.2.1-2. Since the sequences satisfy Theorem 1, they are checking experiments. Since the setup state in both triples is a distinguishing state, a synchronizing input needs to be added to the beginning of either sequence. The sequences in Table 2.2.1-2 are minimum length. The two tests shown in Table 2.2.1-2 are symmetric, the sequence of values on the S input of the first test is the same as that of the R input on the second test, and vice versa.

Table 2.2.1-2 Minimum-Length (6) Checking Experiments for Two-State SR-Latch.

S	0	0	1	0	0	0	S	1	0	0	0	1	0
R	1	0	0	0	1	0	R	0	0	1	0	0	0
Q	0	0	1	1	0	0	Q	1	1	0	0	1	1
State	4	2	5	3	4	2	State	5	3	4	2	5	3
Triples		B					Triples		A				
				A							B		

2.2.2 Two-State D-Latch

The equation for a D-latch is Q = CD + $\overline{C}$q, and the flow table for the D-latch is shown in Table 2.2.2-1. The graphs of the state triples are shown in Fig. 2.2.2-1. We need to combine a triple from graph A and a triple from graph B. Suppose we start with the triples in graph A. State 6 is selected as the setup state because it a synchronizing input, giving the state triple 6,7,5. Now, looking at graph B, there are two triples: 7,6,2 and 3,6,2. State 7 has already been visited in the first sequence, thus visiting it again has no benefit. Choosing state 3 as the setup state would make state 3 follow state 5, linking the two distinguishing inputs. Thus the sequence becomes 6,7,5,3,6,2. The sequence is missing state 4, which can be added to the end of the sequence. This makes the final sequence 6,7,5,3,6,2,4. The same approach can be used starting with graph B in Fig. 2.2.2-1. The sequence in that case would be 7,6,2,4,7,5,3. These two state sequences, and the checking experiments that would generate them, are shown in Table 2.2.2-2. Since the lengths of these sequences meet the lower bound, these sequences are minimum length. The two tests shown in Table 2.2.2-2 are symmetric, the sequence of values on the C input are the same for both tests, and the values on the D input of the first test are complements of the values on the D input of the second test.

Table 2.2.2-1 Flow Table for Two-State D-Latch.

	CD				Q
	00	01	11	10	
	(2), 0	(4), 0	7	(6), 0	0
	[3], 1	[5], 1	[7], 1	6	1

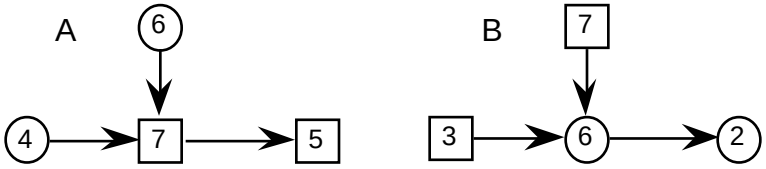


Figure 2.2.2-1 Graphs of State Triples for Two-State D-Latch.

Table 2.2.2-2 Minimum-Length (7) Checking Experiments for Two-State D-Latch.

C	1	1	0	0	1	0	0
D	1	0	0	1	1	1	0
Q	1	0	0	0	1	1	1
State	7	6	2	4	7	5	3
Triples	B			A			

C	1	1	0	0	1	0	0
D	0	1	1	0	0	0	1
Q	0	1	1	1	0	0	0
State	6	7	5	3	6	2	4
Triples	A			B			

2.2.3 Two-State MD-Latch

The equation for an MD-latch (Multiplexed-Data latch) is $Q = C(TS + \overline{T}D) + \overline{C}q$. When T = 0, the latch operates in normal mode (D is used as the input), and when T = 1 it uses S as input. The flow table for the MD-latch is given in Table 2.2.3-1. The graphs of the triples are shown in Fig. 2.2.3-1.

Looking at the flow table, all the distinguishing states occur when C = 0, and all the unstable states occur when C = 1. Therefore, the MD-latch is a single-input control state machine, and C is the control input. There are 24 total stable states, and the number of unstable states in each of the rows is 4. Therefore, the length of a checking experiment must be at least 25. Makar and McCluskey [94] shows that the length must be at least 26. The details of combining the triples to derive minimum length sequences also appear in Makar and McCluskey [94]. One such sequence is shown in Table 2.2.3-2.

Table 2.2.3-1 Flow Table for Two-State MD-Latch.

	DS																
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
	C = 0								C = 1								
	T = 0				T = 1				T = 0				T = 1				Q
	(2),0	(4),0	(6),0	(8),0	(10),0	(12),0	(14),0	(16),0	(18),0	(20),0	19	21	(22),0	23	25	(24),0	0
	[3],1	[5],1	[7],1	[9],1	[11],1	[13],1	[15],1	[17],1	18	20	[19],1	[21],1	22	[23],1	[25],1	24	1

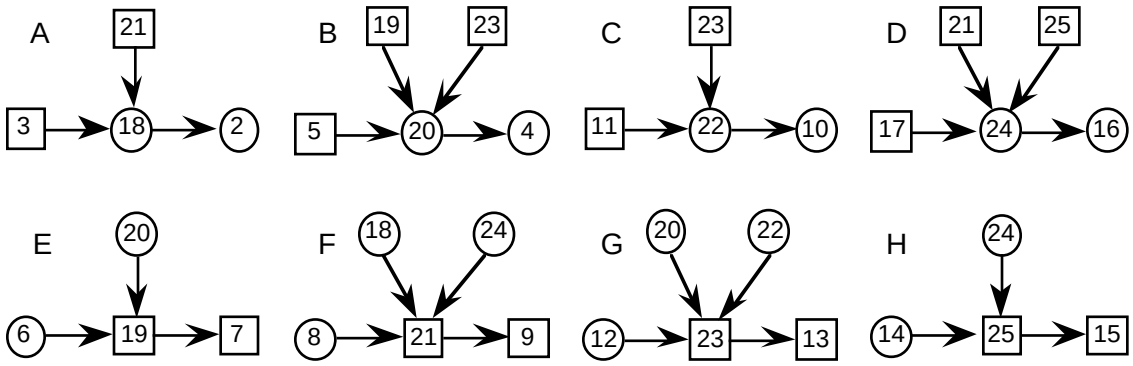


Figure 2.2.3-1 Graphs of State Triples for Two-State MD-Latch.

Table 2.2.3-2 A Minimum-Length (26) Checking Experiment for MD-Latch.

T	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
C	1	1	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0
D	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
S	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0
Q	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	1	0	0
State	24	21	9	3	18	2	8	6	19	7	5	20	4	12	23	13	11	22	10	12	14	25	15	17	24	16
Triples	F			A				E			B			G			C				H			D		

2.2.4 Two-State TP-Latch

The equation for a TP-latch (Two-Port latch) is $Q = C_1D_1 + C_2D_2 + \overline{C}_1\overline{C}_2q$. C_1 and C_2 should not be both active at the same time. The TP-latch loads the data input corresponding to the active control input. The flow table is shown in Table 2.2.4-1.

The last four columns are marked as don't cares because the operation of the latch is not defined when both control lines are active. Graphs of the triples are shown in Fig. 2.2.4-1. Since states 2 and 7 appear twice as the only distinguishing states of triples, and there are 16 total states, the lower bound on the test length is 19. Makar and McCluskey

Table 2.2.4-1 Flow Table for Two-State TP-Latch.

	D_1D_2																
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
	$C_2 = 0$								$C_2 = 1$								
	$C_1 = 0$				$C_1 = 1$				$C_1 = 0$				$C_1 = 1$				Q
	(2),0	(4),0	(6),0	(8),0	(10),0	(12),0	11	13	(14),0	15	17	(16),0	-	-	-	-	0
	[3],1	[5],1	[7],1	[9],1	10	12	[11],1	[13],1	14	[15],1	[17],1	16	-	-	-	-	1

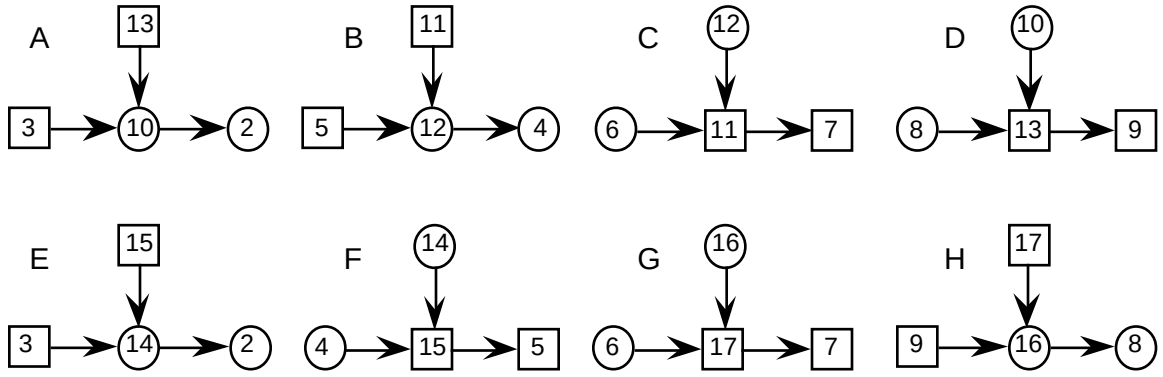


Figure 2.2.4-1 Graphs of State Triples for Two-State TP-Latch.

[94] shows that the length must be at least 23. The details of combining the triples to derive minimum length sequences also appear in Makar and McCluskey [94]. One such sequence is shown in Table 2.2.4-2.

Table 2.2.4-2 A Minimum-Length (23) Checking Experiment for TP-Latch.

C_1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0
C_2	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
D_1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1
D_2	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
Q	1	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1
State	13	10	2	4	15	5	3	14	2	8	13	9	16	8	6	17	7	5	12	4	6	11	7
Triples	A			F			E			D													
												H			G			B			C		

2.3 Shift Register

A shift register is a circuit that, once every cycle, shifts its stored contents . A double-rank design of a shift register using D-latches is shown in Fig. 2.3-1. In this circuit, a *cycle* consists of a CK_1 pulse, followed by a CK_2 pulse. When CK_1 is 1 the data on d1 is transferred to d2, the data on d3 is transferred to d4, and the data on d5 is transferred to d6. When CK_2 is 1 the data on d2 is transferred to d3, the data on d4 is transferred to d5, and the data on d6 is transferred to d7. If CK_1 and CK_2 are 1 at the same time, the data at d1 shifts all the way to d7. Therefore, for proper operation, CK_1 and CK_2 must be non-overlapping clocks.

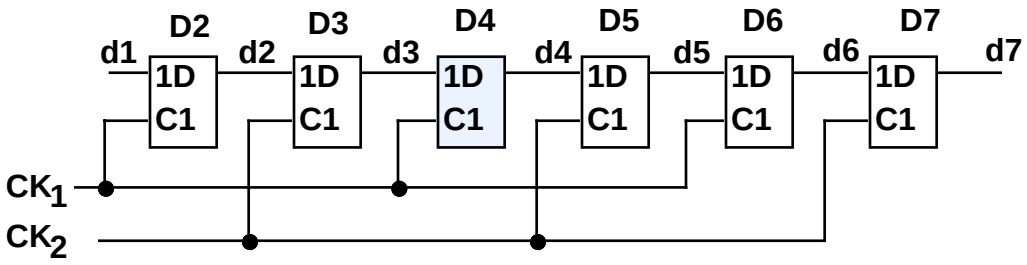


Figure 2.3-1 Double Rank Shift Register.

In the rest of this section, we analyze the input and output constraints imposed on the latch by embedding it in a shift register. From these constraints, we derive a new flow table and a new set of state triples. From the new flow table and state triples we derive a new checking experiment. The checking experiment we derive for the latch in a double-rank shift register can be described as the data sequence 01100 (in a data sequence one value is applied to the data input (d1) every cycle). Extra cycles are needed after applying the data sequence so that the data values reach the internal flip-flop, and the output of the internal flip-flop reaches the primary output, d7. We also show that any sequence that contains all four possible transitions (0->1, 1->1, 1->0 and 0->0), applies a checking experiment to all the latches.

The non-overlapping constraint and the interconnection of the latches impose controllability and observability restrictions on the latches in the shift register. There are two issues here. First, the appropriate patterns must be applied to the latch under test. Second, the error at the output of the faulty latch must be captured and transferred to the primary output (d7) of the shift register. During this test, it is assumed that only one latch is faulty. In this section we show that despite these constraints, a checking experiment can be applied to all the latches in the scan chain.

The *application point* of a signal is the point in the cycle where it can change. The signal on d2 can only change when CK_1 is high. Similarly, d3 can only change

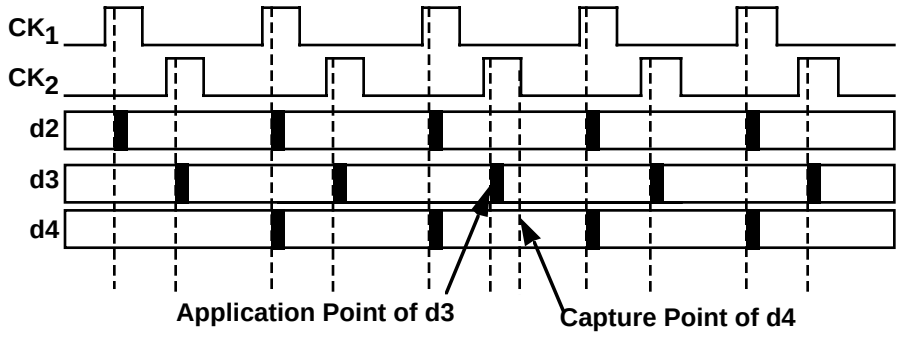


Figure 2.3-2 Restrictions in Shift Register Operation.

when CK_2 is high, but during that time d2 is not changing (because CK_1 is low when CK_2 is high). Therefore, the application point of d3 is the rising edge of CK_2, and there can only be one change per cycle (see Fig. 2.3-2).

The output of the latch under test must be propagated to the primary output of the circuit. The only path from the output of the highlighted latch (D4) in Fig. 2.3-1 is from d4 through d5, and on to d6 and d7, the primary output. The D5 latch "captures" the value on d4 on the falling edge of CK_2. This point in the cycle, called *capture point*, is the only point in the cycle when the value on d4 can be latched by D5. Once captured by D5, the value can be propagated to the primary output by applying the appropriate number of cycles (in this case one more cycle is needed). Since CK_1 and CK_2 cannot be 1 at the same time, and CK_1 is connected to the C input of the latch, then we can only capture outputs of states in the C = 0 columns of Table 2.3-1a.

The restrictions on a D-latch in a double-rank shift register can be summarized as follows:

- the D input can change only when the C input is 0.
- the D input can change only once per cycle.
- the output of the latch is "observed" only for states with C = 0.

The first restriction implies that the 6->7 and 7->6 transitions in Table 2.3-1a cannot occur. This means that some of the triples cannot be used and we can reduce the graphs of the state triples as in Fig. 2.3-3. The output restriction implies that the output

(a) Original Graph of State Triples **(b) Reduced Graph of State Triples**

Figure 2.3-3 Reduced Graph of State Triples for D-Latch in Shift Register.

Table 2.3-1 Reduced Flow Table for Two-State D-Latch in Shift Register.

(a) Original Flow Table

CD / Q	00	01	11	10
0	(2), 0	(4), 0	7	(6), 0
1	[3], 1	[5], 1	[7], 1	6

(b) Reduced Flow Table

CD / Q	00	01	11	10
0	(2), 0	(4), 0	7	(6)
1	[3], 1	[5], 1	[7]	6

cannot be observed for the right two columns of the flow table. The reduced flow table is shown in Table 2.3-1b. In the reduced table, the circles and squares indicate that the output of the latch is 0 or 1 for the corresponding state. States with explicitly listed outputs have observable outputs, and states with no explicit output listed have no observable output.

The reduced flow table and state triples can be used to derive a checking experiment. Graph A has one state triple 4,7,5, and thus requires the input sequence CD = 01, 11, 01. After this sequence, we can have D change to 0, since C = 0. However, doing so would change the total state of the latch under test to state 3 before the output is "captured" by D5. Therefore, the sequence must be directly followed by CD = 11. Thus the sequence becomes CD = 01, 11, 01, 11. To get to state 4 (the setup state of the triple) requires the sequence CD = 10, 00, 01. Thus the complete sequence becomes CD = 10, 00, 01, 11, 01, 11. Similarly, the sequence for the triple in graph B is CD = 11, 01, 00, 10, 00, 10. The last input of the sequence of triple A is the first input of the sequence of triple B, and vice versa. Therefore, the two sequences can be easily combined. An example of a combined sequence is shown in Table 2.3-2. A * in the Q entry of this table indicates outputs that are not captured, and thus never observed. Since the test in Table 2.3-2 contains both triples, and the D-latch is a single-input control state machine (a single input determines whether an input is distinguishing input or a synchronizing input), then, by Theorem 2, the distinguishing inputs are linked, and the sequence is a checking

Table 2.3-2 Test Sequence for Embedded D-Latch.

Pattern No.	1	2	3	4	5	6	7	8	9	10	11
C	1	0	0	1	0	1	0	0	1	0	1
D	0	0	1	1	1	1	1	0	0	0	0
Q	*	*	0	*	1	*	*	1	*	0	*
State		2	4	7	5			3	6	2	
Triple			A					B			

experiment.

One of the restrictions on the checking experiment for the D-latch in the scan chain is that the D input can only change once every cycle, and we therefore have one data value every cycle. The test in Table 2.3-2 has five data values 01100. The first four values are the captured outputs, and the last value is needed to ensure capture of the output of state 2 (pattern 10). Without the last 0, the value of D in pattern 10 could be 1, putting the circuit in state 4 instead of state 2.

We will now show that any test sequence that contains all four transitions (0->1, 1->1, 1->0 and 0->0) applies a checking experiment to the internal latches of the shift register. First, we show that a sequence with all four transitions must contain certain sub-sequences. Then we show that the responses to these sub-sequences are sufficient to construct the flow table.

For the first step, a test that contains all four transitions can either start with 00 or 11, end with 00 or 11, or not start or end with either one. Table 2.3-3 shows the five possible cases. Consider the first case in Table 2.3-3. If a sequence starts with 00, then the initial string of 0s must be followed by 1, implying that the sequence contains 001. Since a sequence starts with 00, it cannot start with 11. The string of 1s for the 1->1 transition must be preceded by 0, implying that 011 is part of the sequence. The 10 sequence is needed for the 1->0 transition. Similar arguments can be made for cases 2, 3 and 4 in Table 2.3-3.

Table 2.3-3 Five Cases of Sequences With All Four Transitions.

Case	Sequence	Implied sub-sequences
1. Start with 00	00.....	001,011,10
2. Start with 11	11.....	110,100,01
3. End with 00	00	110,100,01
4. End with 11	11	001,011,10
5. Start and end with neither	...0...11...0... ....1...00...1...	110,011 001,100

For the last case, since the string of 1s for the 1->1 transition is not at the beginning or end of the sequence, the string must be preceded and followed by 0, implying that the sequence contains 011 and 110. Similarly, since the string of 0s for the 0->0 transition is not at the beginning or the end of the sequence, the sequence contains 100 and 001. Therefore, any sequence that has all four transitions must contain either: 100, 110, 01 or 011, 001, 10. These two sets of sub-sequences are complements of each other (one can be derived from the other by complementing all the entries). We next

show that the responses to the first set of sub-sequences (100, 110, 01) are sufficient to construct the flow table. A similar analysis can be applied to the other set of sub-sequences.

Tables 2.3-4 shows how the three sequences (100, 110 and 01) are applied to an internal latch. In this table, x indicates an unknown value, either 0 or 1, and * indicates that the output is not captured. From Table 2.3-4a, patterns 5a and 7a show two different outputs for the input CD = 00, indicating different internal states. Pattern 6a causes a change in internal state. Thus the flow table in Table 2.3-5a can be formed. In the flow tables of Table 2.3-5, we show the internal states and outputs rather than the total states. All stable states are shown in a circle. Now, pattern 4b and pattern 5c show two different values for input CD = 01, indicating different internal states. This is shown in Table 2.3-5b. Here, the output value is not known, since the connection between the states has not yet been identified. According to the flow table, pattern 3c would set the machine to

Table 2.3-4 Test Sub-Sequences Applied to Embedded D-Latch.

a) Test Sub-Sequence 100

Pattern No.	1a	2a	3a	4a	5a	6a	7a	8a	9a	10a
Sequence			1			0		0		
C	0	0	1	0	0	1	0	1	0	0
D	x	1	1	1	0	0	0	0	0	x
Q	*	x	*	*	1	*	0	*	*	0

b) Test Sub-Sequence 110

Pattern No.	1b	2b	3b	4b	5b	6b	7b	8b	9b	10b
Sequence			1		1			0		
C	0	0	1	0	1	0	0	1	0	0
D	x	1	1	1	0	0	0	0	0	x
Q	*	x	*	1	*	*	0	*	*	0

c) Test Sub-Sequence 01

Pattern No.	1c	2c	3c	4c	5c	6c	7c	8c	9c	10c
Sequence			0			1				
C	0	0	1	0	0	1	0	0		
D	x	0	0	0	1	1	1	x		
Q	*	x	*	*	0	*	*	1		

internal state A even though no output is observed. Since patterns 4c and 5c do not cause a change in state, the machine must be in state A after pattern 5c is applied, making the A,01 entry of the flow table 0, which in turn makes the B,01 entry of the flow table 1 (see Table 2.3-5c). Now, pattern 8c must keep the machine in state B since the output is 1. Thus, after the application of pattern 7c the machine must also be in internal state B. Since we have already established that the machine is in state A after pattern 5c, then pattern 6c must cause a transition from state A to state B. The final flow table is shown in Table 2.3-5d.

Table 2.3-5 Flow Table Fragments.

a) From Table 2.3-4 (a)

	CD 00	01	11	10
A	(A) ,0			(A)
B	(B) ,1			A

b) Adding CD = 01 Column

	CD 00	01	11	10
A	(A) ,0	(A) ,v		(A)
B	(B) ,1	(B) ,$\bar{v}$		A

c) Output When CD = 01

	CD 00	01	11	10
A	(A) ,0	(A) ,0		(A)
B	(B) ,1	(B) ,1		A

d) Final Table

	CD 00	01	11	10
A	(A) ,0	(A) ,0	B	(A)
B	(B) ,1	(B) ,1	(B)	A

From the definition of the double rank shift register, data applied at the d1 input is applied to all the D inputs of the latches if enough clock cycles are applied. Therefore, the 01100 data sequence can be applied to all the latches of the shift register using the waveform in Fig. 2.3-4. If we have a shift register of only two latches, then we need five cycles for the five patterns to go through the shift register. For each additional pair of latches in the shift register we need one extra cycle. Therefore we need 4 + N/2 cycles, for a shift register of length N.

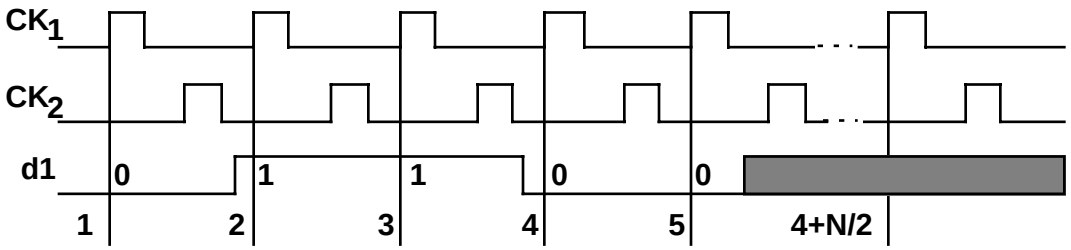


Figure 2.3-4 Test for Shift Register Latches (N = Number of Latches).

2.4 Latch Based Scan Chains

Chapter 1 introduced scan chains as a method of simplifying test generation for sequential circuits. In this section, we focus on generating tests for the latches in the two latch-based architectures, MD-latch and LSSD. For each of the architectures, we describe a method to generate test patterns that apply a checking experiment to the latches in the circuit. In Section 2.4.1 we analyze the MD-latch architecture, and in Section 2.4.2 we analyze LSSD. In Chapter 4, we use the information developed here to describe an algorithm to generate test patterns to test all the latches in a scan design.

2.4.1 MD-Latch Based Scan Architecture

The MD-latch based scan architecture is shown in Fig. 2.4.1-1. The scan chain consists of MD-latches as well as D-latches. All the latches need to be tested. We first show how the shift register test of Section 2.3 can be used to test the D-latches in the scan chain. The same test cannot be used for the MD-latches because they have inputs from the combinational logic. As with the shift register, the circuit imposes constraints on the control of the inputs, and observation of the outputs of the MD-latches in the scan chain. We first analyze the effect of these constraints on the state triples and flow table of the MD-latch. We then show how test patterns can be generated for each of the state triples.

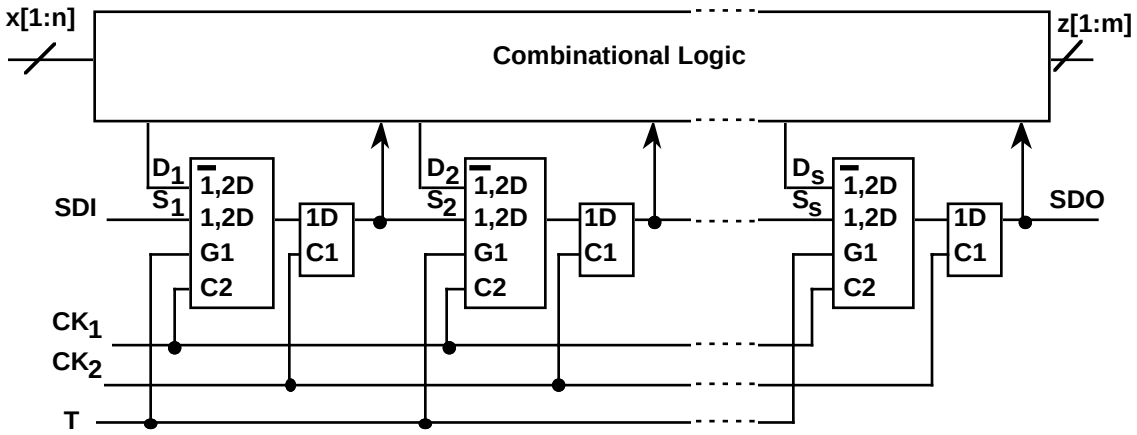


Figure 2.4.1-1 MD-Latch Based Scan Chain Architecture.

In the MD-latch architecture, when T is set to 1, the scan chain behaves as the double-rank shift register in Section 2.3. For normal operation T is set to 0. The two clocks, CK_1 and CK_2, are two-phase clocks.

In Section 2.3, we showed that any sequence that has all four transitions (0->0, 0->1, 1->1 and 1->0) would apply a checking experiment to all the D-latches in a shift register. The same approach can be used here to test the D-latches in the scan chain. T is set to 1, the sequence is applied to SDI, and the output is observed at SDO. With T

set to 1, the MD-latches behave like D-latches, and the whole scan chain behaves as a shift register.

The MD-latches have inputs from the combinational logic as well as from D-latches. Therefore, there is no simple sequence as in the case of the shift register than can apply a checking experiment to all the MD-latches in the scan chain. In this section, we show how patterns can be generated for testing the MD-latches in the scan chain.

The S input of the MD-latch is the output of a D-latch. Therefore, the S input can only change when CK_2 is 1, which implies that CK_1 is 0. From the MD-latch perspective, this means that S can only change when its C input is 0. This means that some of the triples cannot be used and we can reduce the graphs of the state triples as in Fig. 2.4.1-2. The reduced flow table is shown in Table 2.4.1-1. In this table, the outputs in the C = 1 columns are unobservable, since, as with the shift register, we can only observe outputs of states with C = 0.

Since the MD-latch is a single-input controlling state machine, from Theorem 2, a sequence that visits all state triples, and use a distinguishing state as a setup state for each triple, is a checking experiment. Graphs of these triples are shown in Fig. 2.4.1-3. Therefore, we need to find a test pattern for each triple in Fig. 2.4.1-3. A test pattern consists of bits, each bit corresponding to a value of an MD-latch or a primary input. The

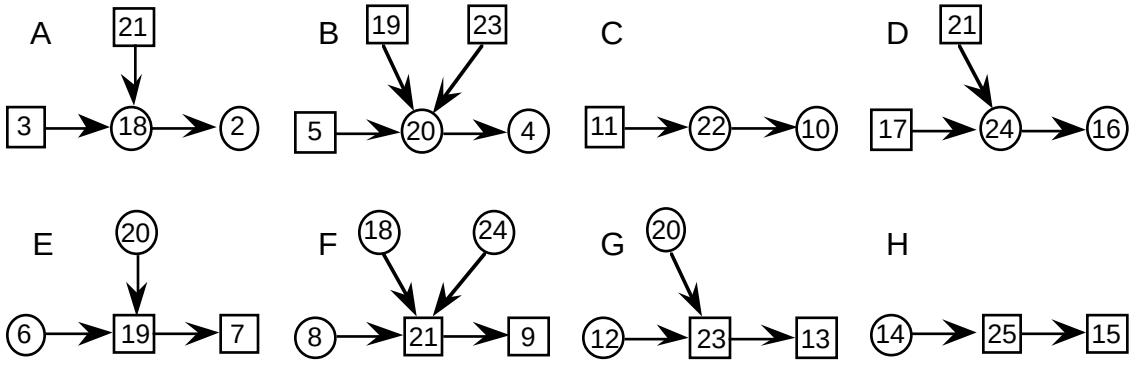


Figure 2.4.1-2 Reduced State Triples for MD-Latch in Scan Chain.

Table 2.4.1-1 Reduced Flow Table for Two-State MD-Latch in Scan Chain.

	DS																
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
	C = 0								C = 1								
	T = 0				T = 1				T = 0				T = 1				Q
	(2),0	(4),0	(6),0	(8),0	(10),0	(12),0	(14),0	(16),0	(18)	(20)	19	21	(22)	23	25	(24)	0
	[3],1	[5],1	[7],1	[9],1	[11],1	[13],1	[15],1	[17],1	18	20	[19]	[21]	22	[23]	[25]	24	1

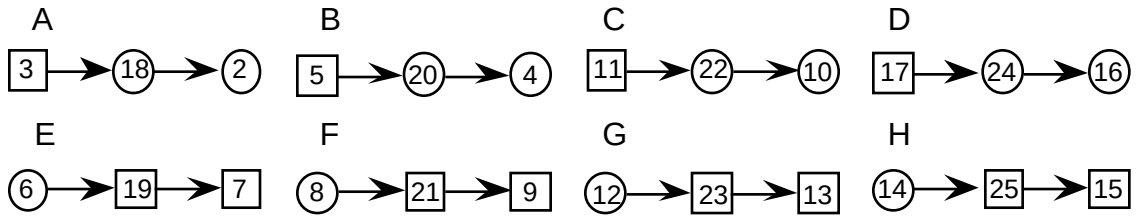


Figure 2.4.1-3 Graphs of State Triples with Distinguishing States as Setup States.

test pattern should:

1- Put the MD-latch under test in the setup state of the state triple. This includes the internal state of the latch under test, and its inputs.

2- Make the MD-latch under test visit the three states of the triple when a pulse is applied to C.

The state triples can be divided into two groups: those that have T = 0 (ABCD) and those that have T = 1 (EFGH). We first show how a test can be derived for a triple with T = 1, then for T = 0. Consider triple G (12, 23, 13) with T = 1. The waveforms at the inputs and output of the MD-latch under test are shown in Fig. 2.4.1-4. Dashed lines in these and all waveforms show when state outputs are captured.

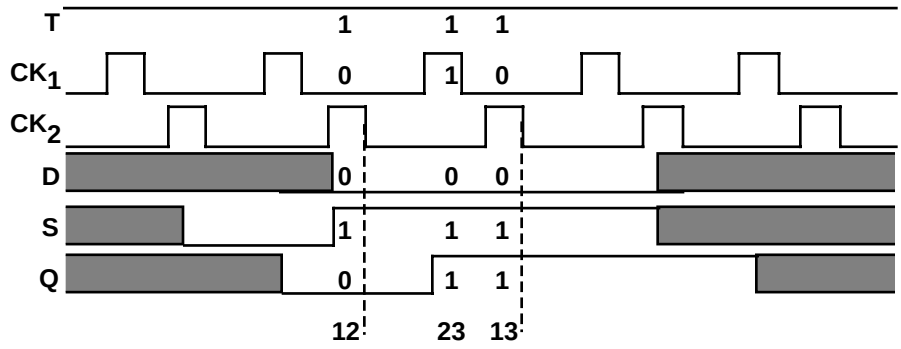


Figure 2.4.1-4 Waveforms for Triple G of MD-Latch Under Test.

Since Q is the output of the MD-latch under test, its value changes slightly after a positive transition of CK_1. The S input is the output of a D-latch, and therefore its value changes after a positive transition of CK_2. The Q waveform is the S waveform shifted by half a cycle, since T = 1 for the entire waveform.

Fig. 2.4.1-5 shows part of the circuit that is involved in testing the latch under test L_t. In this circuit, some of the MD-latches are paired with the D-latches following them for ease of discussion. The latch pairs are numbered based on their order in the scan chain. For example, the latch pair preceding the latch under test is L_{t-1}. The D input of

the latch is the output of combinational logic with *n* inputs. *p* of the inputs come from other latches and the rest come from primary inputs. The inputs of the combinational logic are $X_1,X_2,...,X_n$. If an input X_i is driven by a latch, the latch is called L_{X_i}.

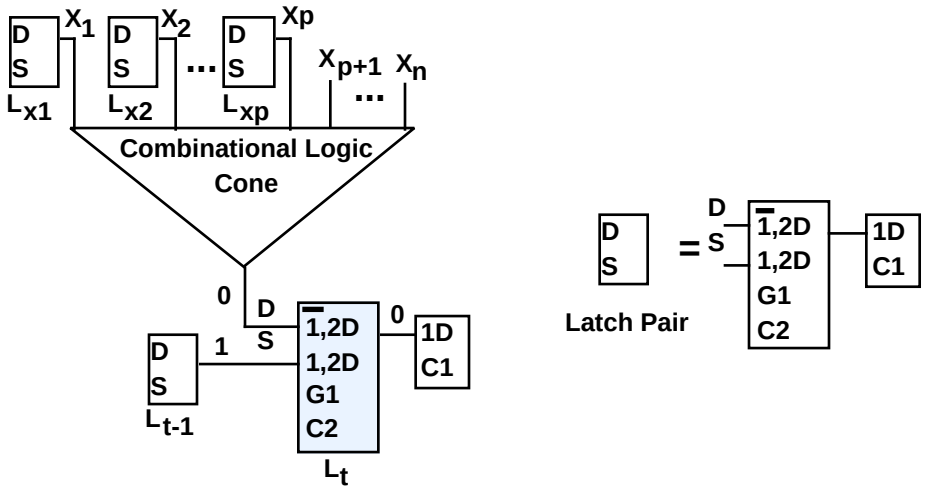


Figure 2.4.1-5 Part of Circuit Involved in Testing L_t.

The first requirement of the test pattern is to put the MD-latch in the setup state of the triple. In our example, since we require Q to be 0 in our setup state, the bit corresponding to the latch under test (L_t), should be 0. Since we require a 1 on the S input, the bit corresponding to L_{t-1} should also be set to 1. We also need to set the D input to 0. Therefore, the bits of the pattern corresponding to $L_{x_1},L_{x_2},...,L_{x_p}$ and $X_{p+1},X_{p+2},...,X_n$ should be selected such that the output of the combinational logic is 0.

From the waveforms in Fig. 2.4.1-4, we see that the distinguishing state of the triple also requires that D = 0, and S = 1 after the scan chain has shifted once. The circuit in Fig. 2.4.1-5 is expanded slightly in Fig. 2.4.1-6 to include the logic that would affect L_t after the scan chain has shifted once. In this figure, L_{t-2} is the latch that precedes L_{t-1}. After the shift, the value in L_{t-2} is transferred to L_{t-1}, which as we mentioned earlier is the S input to L_t. Therefore, the bit corresponding to L_{t-2} should also be set to 1. Also, after the shift, values in $L_{x_1}-1,L_{x_2}-1,...,L_{x_p}-1$ are transferred to $L_{x_1},L_{x_2},...,L_{x_p}$ respectively. Therefore, the bits corresponding to $L_{x_1}-1,L_{x_2}-1,...,L_{x_p}-1$ and $X_{p+1},X_{p+2},...,X_n$ should be selected such the output of the combinational logic is 0 after the shift. Formally, if the function of the combinational logic cone is $f(L_{x_1},L_{x_2},...,L_{x_p},X_{p+1},X_{p+2},...,X_n)$, then we need $f(L_{x_1},L_{x_2},...,L_{x_p},X_{p+1},X_{p+2},...,X_n) = 0$ to satisfy setup state requirements, and $f(L_{x_1}-1,L_{x_2}-1,...,L_{x_p}-1,X_{p+1},X_{p+2},...,X_n) = 0$ to satisfy distinguishing state requirements.

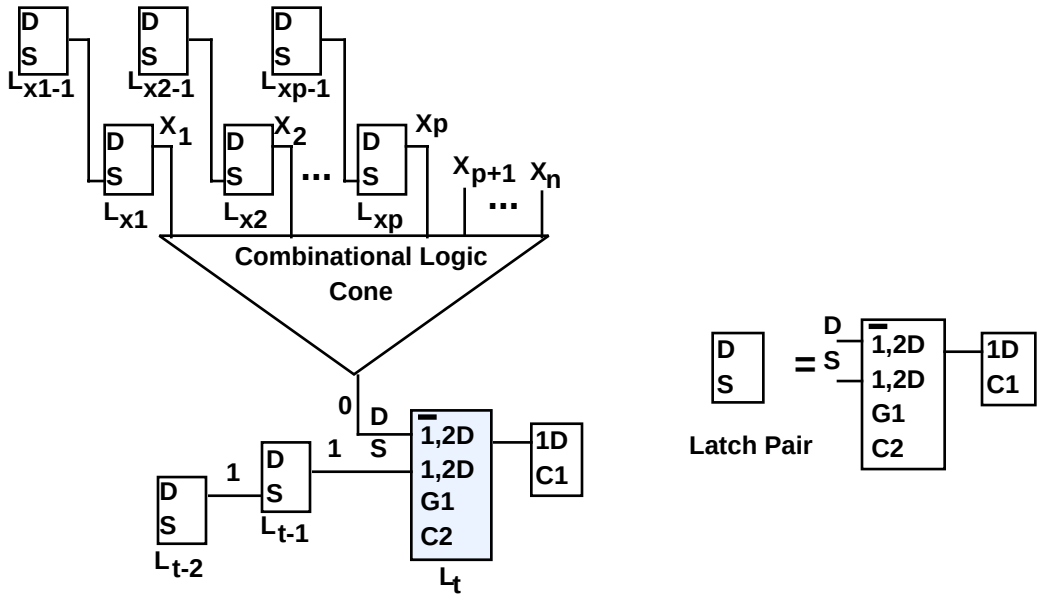


Figure 2.4.1-6 Part of Circuit Involved With Distinguishing State for L_t.

In Fig. 2.4.1-6, all the latches had only one role in the circuit. This was done only to keep the explanation simple. It is possible, for example that the latch preceding the latch under test is also one of the latches that drive the combinational logic (see Fig. 2.4.1-7). In that case, the input driving the combinational logic should be assumed to be 1 when trying to find a pattern to set D to 0. A test pattern that satisfies all the bit

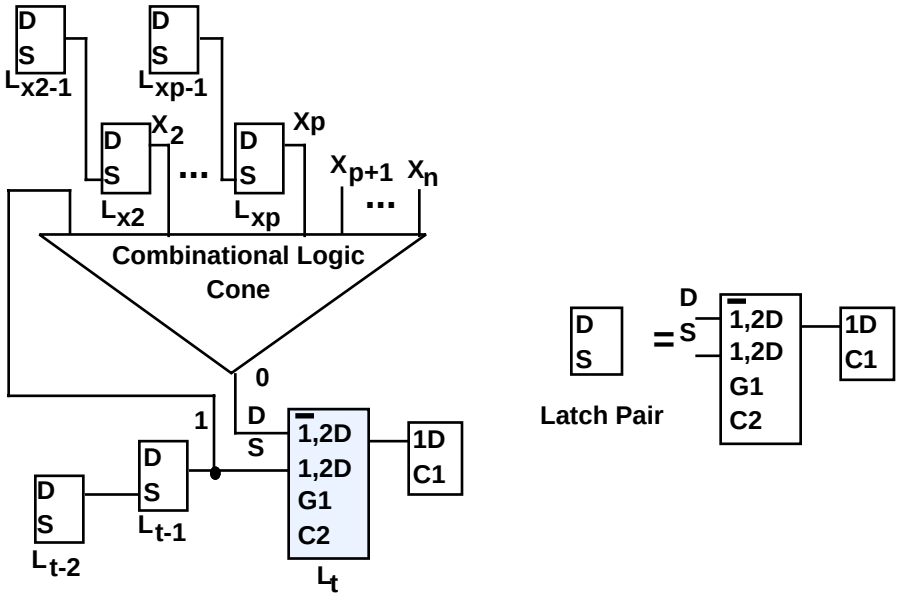


Figure 2.4.1-7 Circuit With L_{t-1} Driving the Combinational Logic of D.

requirements for the setup and distinguishing states of a triple would make the latch visit the three states of the triple consecutively when a pulse on C is applied. Since T = 1 for the entire test, the outputs of the setup and distinguishing states are captured in the scan chain.

A similar approach can be used for triples with T = 0. The waveforms for triple E (6, 19, 7) are shown in Fig. 2.4.1-8. The difference between these waveforms and the ones of Fig. 2.4.1-4 is that T = 0 for one cycle. The approach used above for deriving the test pattern bits for the setup state is exactly the same, because it only involves L_{t-1} and the combinational logic driving D (see Fig. 2.4.1-5). The distinguishing state is reached

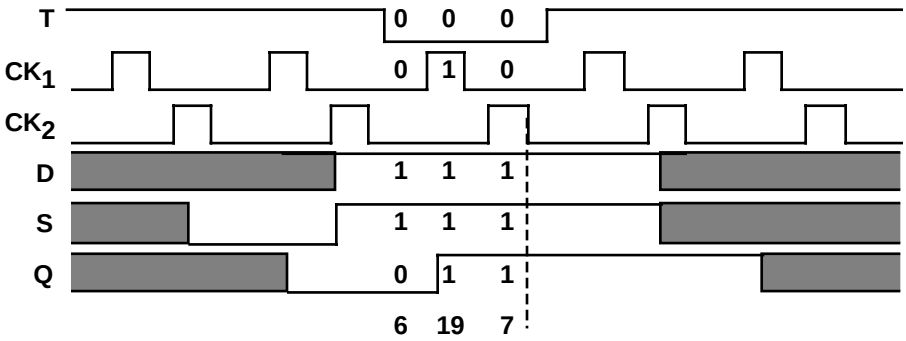


Figure 2.4.1-8 Waveforms for Triple E of MD-Latch Under Test.

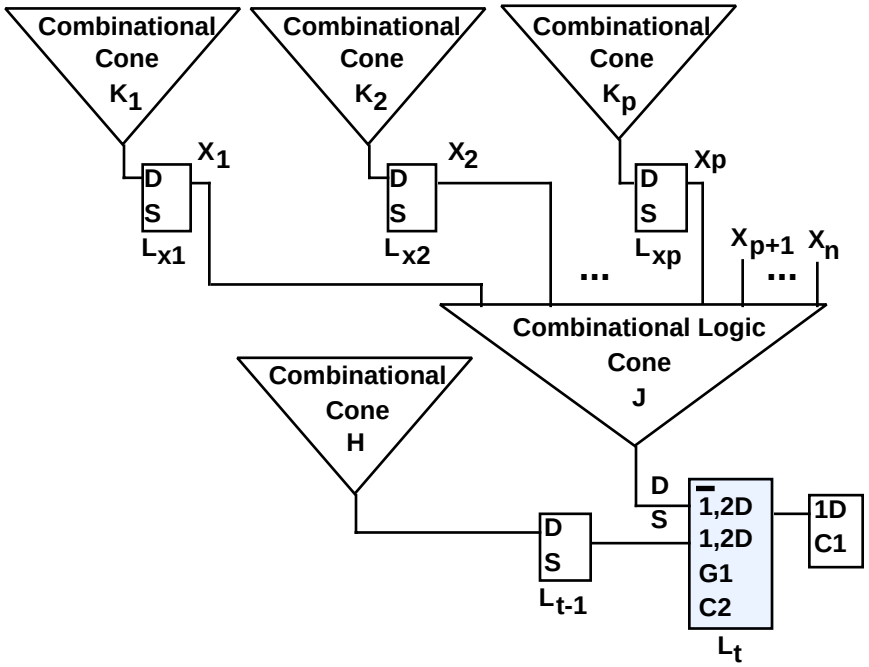


Figure 2.4.1-9 Part of Circuit Involved With Distinguishing State of Triple E for L_t.

after a cycle with T = 0. The circuit in Fig. 2.4.1-9 includes the logic that would affect L_t after one cycle with T = 0. Here, instead of L_{t-2}, the value of S for the distinguishing state is determined by the combinational logic cone H. Therefore, the bits corresponding to the inputs of H should be selected to set the output of the combinational logic to 1. Similarly, the bits corresponding to the inputs of combinational logic cones $K_1,K_2,...,K_p$ should be selected such that they supply a pattern to combinational logic cone J that would set its output to 0. Formally, suppose that the function of combinational logic cone J is f_J. Then we need $f_J(L_{x_1-1},L_{x_2-1},...,L_{x_p-1},X_{p+1},X_{p+2},...,X_n) = 0$ to satisfy the setup state requirements, and $f_J(f_{K_1},f_{K_2},...,f_{K_p},X_{p+1},X_{p+2},...,X_n) = 0$ (where f_{K_i} is the function of combinational logic cone K_i) to satisfy the distinguishing state requirements.

In the example of triple G (T = 1), we captured the output of both the setup and the distinguishing state since we are always shifting values in the scan chain. However in the example of triple E (T = 0), since T = 0 for a cycle, the captured output of the setup state is lost. To capture the setup state, we repeat the same pattern but change the waveform of T as in Fig. 2.4.1-10. We need to capture the output of this state because it is not captured by a test of any other triple.

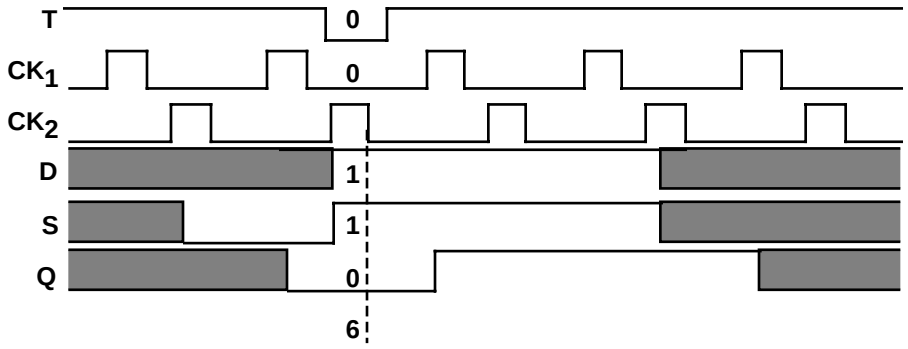


Figure 2.4.1-10 Waveforms for Capturing Output of Setup State of Triple E.

2.4.2 LSSD Architecture

The LSSD architecture is shown in Fig. 2.4.2-1. The scan chain consists of TP-latches as well as D-latches. All latches need to be tested. As with the MD-latch architecture, we show how the shift register test of Section 2.3 can be used to test the D-latches in the scan chain. The same test cannot be used for the TP-latches because they have inputs from the combinational logic. As with the shift register, the circuit imposes constraints on the control of the inputs, and observation of the outputs of the TP-latches in the scan chain. We first analyze the effect of these constraints on the state triples and flow table of the TP-latch. We then show how test patterns can be generated for each of the state triples.

In the LSSD architecture, if CK_1 is set to 0, TCK and CK_2 can be used to make the scan chain behave as the double-rank shift register in Section 2.3. For normal operation TCK is set to 0, and CK_1 and CK_2 are used. Only one of the three clocks, CK_1, CK_2 and TCK, is 1 at any time.

In Section 2.3, we showed that any sequence that has all four transitions (0->0, 0->1, 1->1 and 1->0) would apply a checking experiment to all the D-latches in a shift register. The same approach can be used here to test the D-latches in the scan chain. CK_1 is set to 0, the sequence is applied to SDI with TCK and CK_2 cycling in an non-overlapping fashion, and the output is observed at SDO. In this mode, the TP-latches behave like D-latches, and the whole scan chain behaves like a shift register.

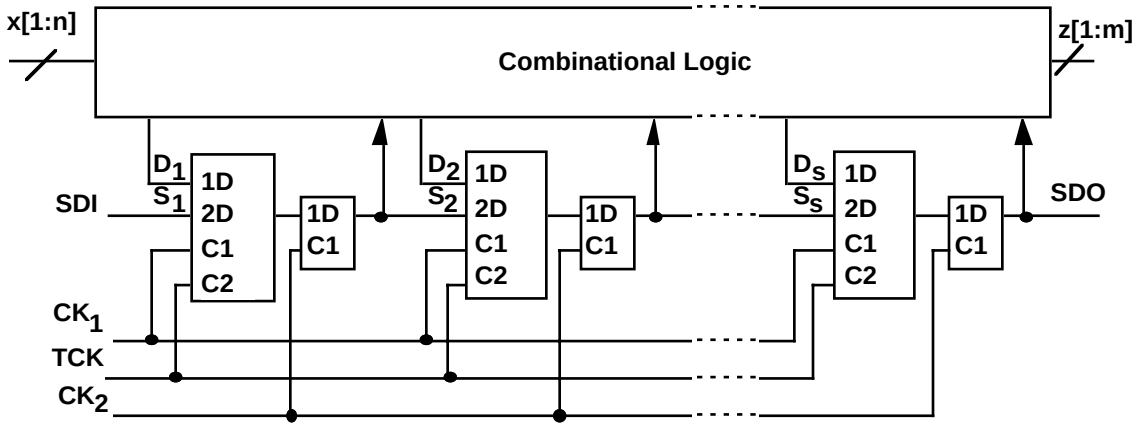


Figure 2.4.2-1 LSSD Scan Chain Architecture.

Just like the MD-latches in the previous section, the TP-latches have inputs from the combinational logic as well as from D-latches. Therefore, there is no simple sequence as in the case of the shift register that can apply a checking experiment to all the TP-latches in the scan chain. Formally, the data inputs of a TP-latch are D_1 and D_2. Since we use D_1 as the data input from the combinational logic, and D_2 as the data input from the preceding latch in the scan chain, we refer to D_1 as the D input, and D_2 as the S input of the latch in the following discussion. In the rest of this section, we derive a procedure, similar to that of the MD-latches, for testing the TP-latches in the scan chain.

The S input of the TP-latch is the output of a D-latch. Therefore, the S input can only change when CK_2 is 1, which implies that CK_1 (C_1 of the latch) and TCK (C_2 of the latch) are 0. From the TP-latch perspective, this means that S can only change when $C_1 = C_2 = 0$. This means that some of the triples cannot be used and we can reduce the graphs of the state triples as in Fig. 2.4.2-2. The reduced flow table is shown in Table 2.4.2-1. In this table, the outputs in the $C_1 = 1$ and $C_2 = 1$ columns are unobservable,

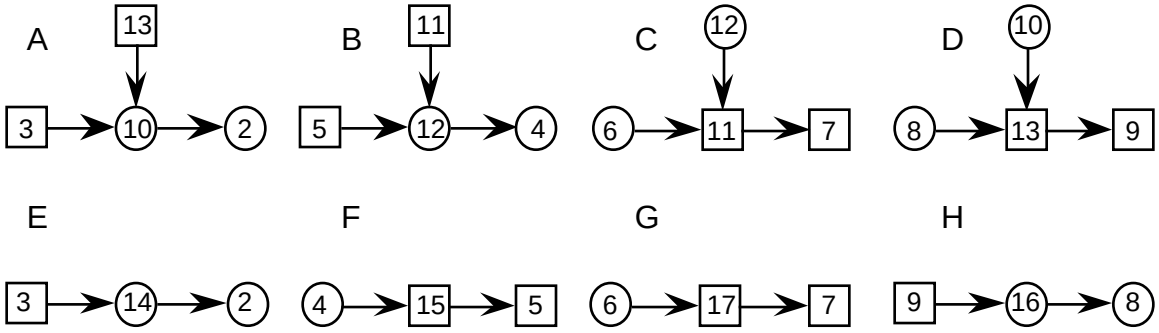


Figure 2.4.2-2 Reduced State Triples for TP-Latch in Scan Chain.

Table 2.4.2-1 Reduced Flow Table for Two-State TP-Latch in Scan Chain.

DS																	
00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10		
$C_2 = 0$								$C_2 = 1$									
$C_1 = 0$				$C_1 = 1$				$C_1 = 0$				$C_1 = 1$				Q	
(2),0	(4),0	(6),0	(8),0	(10)	(12)	11	13	(14)	15	17	(16)	-	-	-	-	0	
[3],1	[5],1	[7],1	[9],1	10	12	[11]	[13]	14	[15]	[17]	16	-	-	-	-	1	

since the D-latches capture values on the negative transition of CK_2, at which point, TCK = CK_1 = 0.

As with the MD-latch architecture, we need to find a test pattern for each triple that uses a distinguishing state as a setup state. The graphs for these triples are shown in Fig. 2.4.2-3. Each bit in the test pattern corresponds to a value of an TP-latch or a primary input. The test pattern should:

1- Put the TP-latch under test in the setup state of the state triple. This includes the internal state of the latch under test, and its inputs.

2- Make the TP-latch under test visit the three states of the triple by applying a pulse on C_1 (CK_1) or C_2 (TCK) depending on the triple.

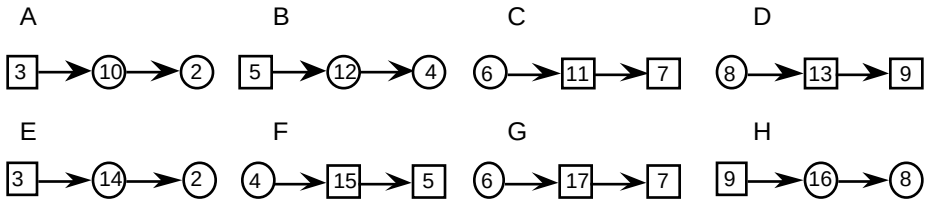


Figure 2.4.2-3 Graphs of State Triples with Distinguishing States as Setup States.

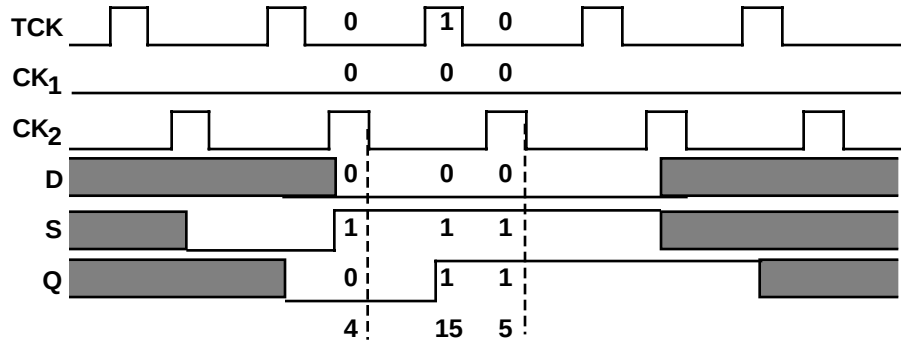


Figure 2.4.2-4 Waveforms for Triple F of TP-Latch Under Test.

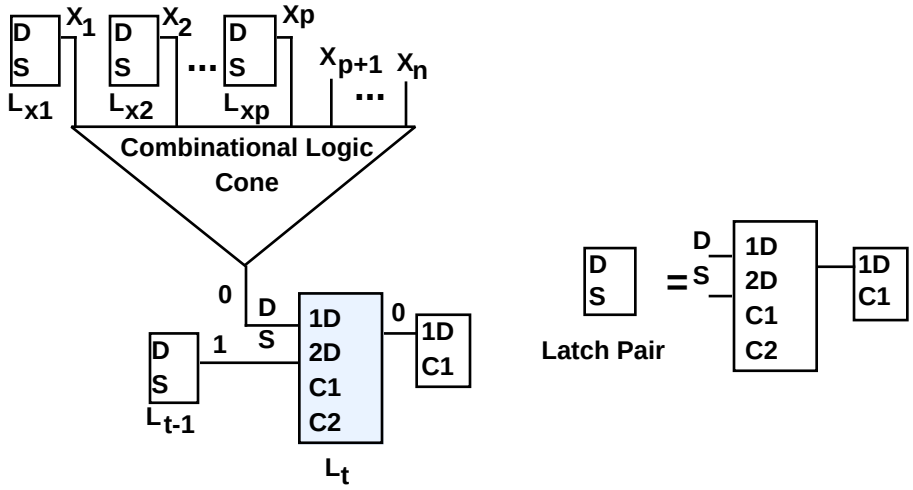


Figure 2.4.2-5 Part of Circuit Involved in Testing L_t.

The state triples can be divided into two groups: those that have TCK changing (ABCD) and those that have CK_1 changing (EFGH). We first show how a test can be derived for a triple with TCK changing, then for CK_1 changing. Consider triple F (4, 15, 5) with TCK changing. The waveforms at the inputs and output of the TP-latch under test are shown in Fig. 2.4.2-4.

Fig. 2.4.2-5 shows part of the circuit that is involved in testing the latch under test L_t. In the circuit, some of the TP-latches are paired with the D-latches following them for ease of discussion. The latch pairs are numbered based on their order in the scan chain. For example, the latch pair preceding the latch under test is L_{t-1}. The D input of L_t is the output of combinational logic with *n* inputs. *p* of the inputs come from other latches and the rest come from primary inputs. The inputs of the combinational logic are $X_1, X_2, ..., X_n$. If an input X_i is driven by a latch, the latch is called L_{X_i}.

The first requirement of the test pattern is to put the TP-latch under test in the setup state of the triple. In our example, since we require Q to be 0 in our setup state, the bit corresponding to the latch under test (L_t), should be 0. Since we require a 1 on the S input, the bit corresponding to L_{t-1} should also be set to 1. We also need to set the D input to 0. Therefore, the bits of the pattern corresponding to $L_{x_1},L_{x_2},...,L_{x_p}$ and $X_{p+1},X_{p+2},...,X_n$ should be selected such that the output of the combinational logic is 0.

From the waveforms in Fig. 2.4.2-1 we see that the distinguishing state of the triple also requires that D = 0, and S = 1 after the scan chain has shifted once. The circuit in Fig. 2.4.2-5 is expanded slightly in Fig. 2.4.2-6 to include the logic that would affect L_t after the scan chain has shifted once. In this figure, L_{t-2} is the latch that precedes L_{t-1}. After the shift, the value in L_{t-2} is transferred to L_{t-1}, which as we mentioned earlier is the S input to L_t. Therefore, the bit corresponding to L_{t-2} should also be set to 1. Also, after the shift, values in $L_{x_1}-1,L_{x_2}-1,...,L_{x_p}-1$ are transferred to $L_{x_1},L_{x_2},...,L_{x_p}$ respectively. Therefore, the bits corresponding to $L_{x_1}-1,L_{x_2}-1,...,L_{x_p}-1$ and $X_{p+1},X_{p+2},...,X_n$ should be selected such that the output of the combinational logic is 0 after the shift. Formally, if the function of the combinational logic cone is $f(L_{x_1},L_{x_2},...,L_{x_p},X_{p+1},X_{p+2},...,X_n)$, then we need $f(L_{x_1},L_{x_2},...,L_{x_p},X_{p+1},X_{p+2},...,X_n) = 0$ to satisfy setup state requirements, and $f(L_{x_1}-1,L_{x_2}-1,...,L_{x_p}-1,X_{p+1},X_{p+2},...,X_n) = 0$ to satisfy distinguishing state requirements. Again, this is similar to the analysis for the MD-latch in the previous section. As with the MD-latch architecture, the latches in Fig. 2.4.2-6 may not be independent. This may add constraints to the bit values of the pattern.

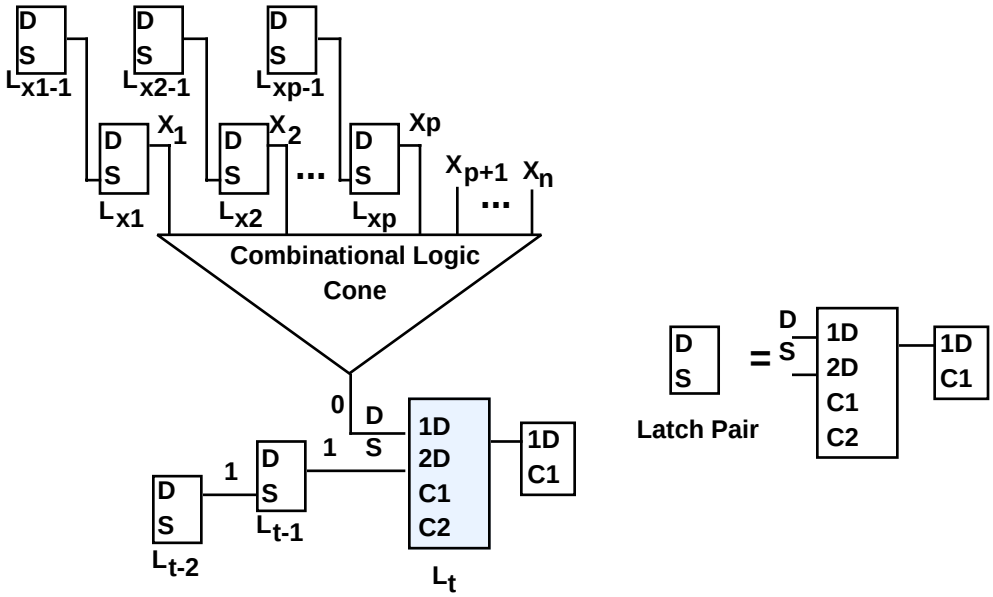


Figure 2.4.2-6 Part of Circuit Involved With Distinguishing State for L_t.

A similar approach can be used for triples with CK_1 changing. The waveforms for triple B are shown in Fig. 2.4.2-7. The difference between these waveforms and the ones of Fig. 2.4.2-4 is the pulse on CK_1 instead of TCK. The approach used above for deriving the test pattern bits for the setup state is exactly the same, because it only involves L_{t-1} and the combinational logic driving D (see Fig. 2.4.2-5). The distinguishing state is reached after a cycle with CK_1. The circuit in Fig. 2.4.2-8 includes the logic that would affect L_t after a CK_1 cycle. Here, instead of L_{t-2}, the value of S for the distinguishing state is determined by the combinational logic cone H. Therefore, the bits corresponding to the inputs of H should be selected to set the output of the

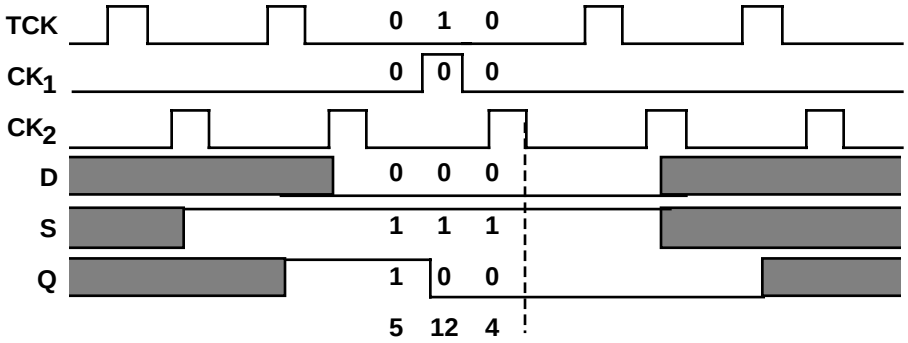


Figure 2.4.2-7 Waveforms for Triple B of TP-Latch Under Test.

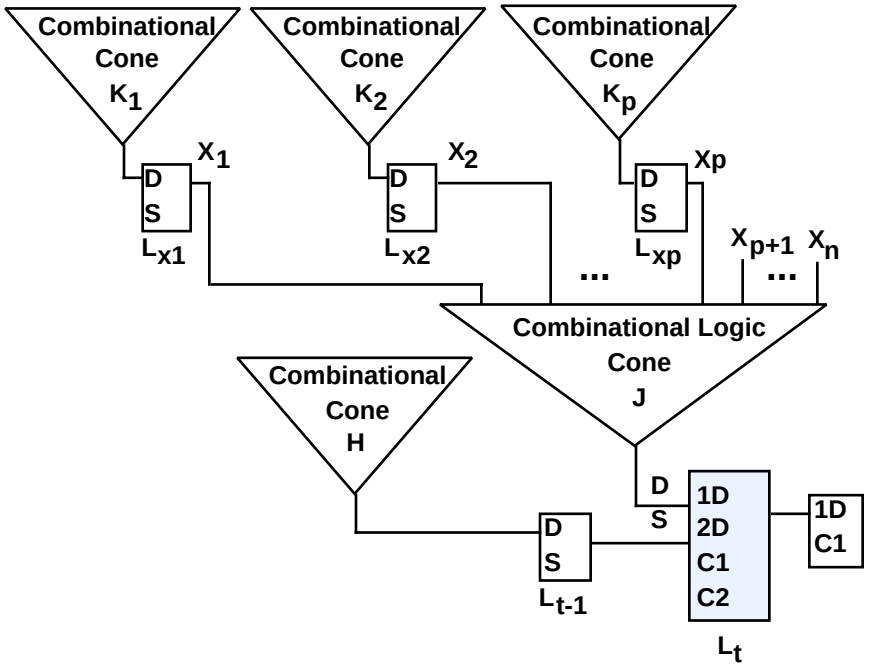


Figure 2.4.2-8 Part of Circuit Involved With Distinguishing State of Triple B for L_t.

combinational logic to 1. Similarly, the bits corresponding to the inputs of combinational logic cones $K_1,K_2,...,K_n$ should be selected such that they supply a pattern to combinational logic cone J that would set its output to 0. Formally, suppose that the function of combinational logic cone J is f_J. Then we need $f_J(L_{X_1}\text{-}1,L_{X_2}\text{-}1,...,L_{X_p}\text{-}1, X_{p+1},X_{p+2},...,X_n) = 0$ to satisfy the setup state requirements, and $f_J(f_{K_1},f_{K_2},...,f_{K_p},X_{p+1},X_{p+2},...,X_n) = 0$ (where f_{K_i} is the function of combinational logic cone K_i) to satisfy the distinguishing state requirements.

In the example of triple F (TCK changing), we captured the output of both the setup and the distinguishing state since we are always shifting values in the scan chain. However in the example of triple B (CK_1 changing), the captured output of the setup state is lost. For the MD-latch we had to repeat the pattern with another waveform to capture the output of the setup state, since the output of the setup state was not captured by a test for any of the triples. However, in this case, state 5 is the distinguishing state of triple F, so we do not need to repeat the pattern. In general, all the setup states of triples with CK_1 changing are setup or distinguishing states of triples with TCK changing, and thus their outputs are always captured.

2.5 Summary

This chapter began with a theoretical analysis of checking experiments for two-state latches, and minimum-length checking experiments for several latches were derived. When latches are used in a circuit, the circuit imposes constraints on the control of the inputs, and observation of the outputs. In Section 2.3, we showed that a simple test can be used to apply checking experiments to all the latches in a shift register. This test can also be used for the D-latches in scan chains. We also showed that it is not possible to derive such a simple test for MD-latches and TP-latches in scan chains. Section 2.4 gave a detailed theoretical analysis of the type of tests that are needed for these memory elements, and how they can be generated. In Chapter 4, we show how the process of generating patterns for MD-latches and TP-latches is automated.

Chapter 3. Checking Experiments for Flip-Flops

Flip-flops are memory elements that do not have the transparency property of latches (discussed in Chapter 2). Flip-flop outputs change only in response to a transition on a control input or a change in an asynchronous input. In Section 3.1 checking experiments for a D flip-flop, an MD flip-flop and a TP flip-flop are derived. As most circuits consist of more than just a single flip-flop, we show what happens to a D flip-flop once it is used in a shift register in Section 3.2. In that section, we show that a simple test will apply a checking experiment to all the flip-flops in the shift register. In Section 3.3, we analyze MD flip-flops and TP flip-flops used in scan chains. We show that there is no simple test that can be applied because the data inputs of the flip-flops depend on the combinational logic in the circuit, and present a technique to generate patterns to test the flip-flops in the scan chain.

3.1 Checking Experiments for Flip-Flops

There are many types of flip-flops [McCluskey 86]. In this section we focus on three that are commonly used in shift registers and in scan chain designs. A *D flip-flop* is a sequential element, in which the data is propagated to the output on a positive transition of the control input, otherwise it holds the stored value. Scan-paths require flip-flops with two different data sources. These can be either Multiplexed-Data flip-flops or Two-Port flip-flops. A *Multiplexed-Data flip-flop (MD flip-flop)* is a D flip-flop with multiplexed data inputs; a *Two-Port flip-flop (TP flip-flop)* has two control inputs with the data source determined by the transitioning control input [McCluskey 86].

Flip-flops have many flow tables, but there is only one primitive flow table for each flip-flop type. Therefore we use primitive flow tables in our analysis. In a *primitive flow table*, each row contains only one stable state. A checking experiment for a primitive flow table will detect all defects that do not increase the number of states in any column. Each cell in the flow table, a *total state*, corresponds to an assignment of values to the circuit inputs and internal states. A total state is an *unstable state* if it causes a change in internal state of the machine. A total state is a *stable state* if the next internal state is the same as the current internal state. In the flow table, a stable state is represented with a number in a circle if its output is 0 and with a number in a square if its output is 1. The numbers of the states start with 2 (0 and 1 are not used to avoid confusion with logic values). States that cannot be reached (because of the single-input change restriction on fundamental mode circuits or because some inputs are not allowed) are called *unspecified states*. Unspecified states are shown with "–" in the flow tables. Stable state S_j is the *successor* of stable state

S_i, and stable state S_i is the *predecessor* of stable state S_j if there is an input that takes the machine from S_i to S_j. A sequence *visits* a total state when the sequence applies the input of the total state while the machine is in the internal state of the total state. A total state is *identified* by a sequence if the sequence provides enough information to reconstruct the corresponding entry in the flow table. A *distinguishing sequence* gives a different output sequence for each state. A *transition tour* of a flow table is a sequence that applies all transitions in the flow table (i.e., it visits all the unstable states). In the following sub-sections, we show how to derive checking experiments for a D flip-flop, an MD flip-flop and a TP flip-flop. Minimum-length checking experiments are derived for each.

3.1.1 D Flip-Flop

A *D flip-flop* is a sequential element, in which the data is propagated to the output on any positive transition of the control input, otherwise it holds the stored value. The primitive flow table for a D flip-flop is shown in Table 3.1.1-1. For a sequence to be a checking experiment, it must identify all the stable states and all the unstable states.

Table 3.1.1-1 Primitive Flow Table of D Flip-Flop.

	C D 00	01	11	10
2	(2),0	4	–	8
3	[3],1	5	–	8
4	2	(4),0	7	–
5	3	[5],1	7	–
6	–	4	(6),0	8
7	–	5	[7],1	9
8	2	–	6	(8),0
9	3	–	7	[9],1

The primitive flow table in Table 3.1.1-1 has two stable states in each column, each with different output values. Therefore, applying an input and observing the value at the output is sufficient to identify which stable state the machine is in. No distinguishing sequence is needed. Since each stable state is the successor of another stable state, a sequence that visits all unstable states also visits all the stable states. In other words, a transition tour of the primitive flow table, is a checking experiment. This is very different from our analysis of the latches in Chapter 2. There, not all transitions were necessary because we had a two state flow table.

Theorem 1: A transition tour of its primitive flow table is a checking experiment for a D Flip-Flop.

Proof: In a primitive flow table all transitions are needed for forming a checking experiment. If a transition is left out, then the flow table entry corresponding to the transition can be replaced by another state. There will always be another state, because there are two stable total states in each column. Since all states are successors of some other state, applying all transitions must visit all the states. Visiting the stable total states is sufficient to identify them as there are only two states per column, and they have different output values.

Now that we established the conditions for a sequence to be a checking experiment, we show how to derive a checking experiment. The transitions in the flow table can be described graphically as shown in Fig. 3.1.1-1. In this digraph (a *digraph* or directed graph is a graph with directed edges) a node represents a stable state, and an edge represents a transition from one stable state to another (an unstable state). Any sequence that goes through all the edges is a checking experiment. Of course, the shortest sequence would be one that goes through each edge only once. Graphs in which a transition tour needs to visit each edge only once are called *euler digraphs*. In an euler digraph each node has as many incoming edges as it has outgoing edges [Wilson 72]. Looking at Fig. 3.1.1-1, we see that not all nodes satisfy this requirement, thus the graph is not an euler digraph. There are 16 edges in the graph, so a sequence must have at least 16 transitions. State 9 is the only predecessor for states 3 and 7, and state 7 is the only predecessor of state 9. Therefore, the transitions from state 7 to state 9 will have to appear twice in a sequence. A similar argument can be made for the transition from state 8 to state 6. This raises the number of required transitions to 18. At the beginning of operation, the state of the flip-flop is not known, and a *synchronizing sequence* is needed to put the flip-flop in a known state. There are two possible sequences for initialization: CD = 01, 11 ending in state 7 or

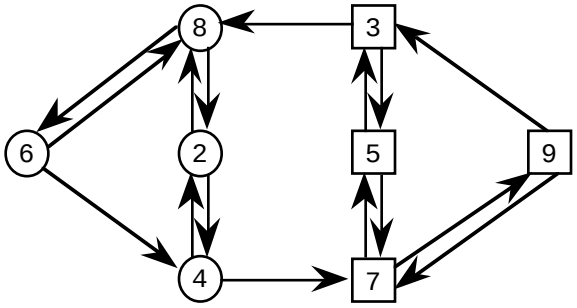


Figure 3.1.1-1 Transition Graph From Table 3.1.1-1.

Table 3.1.1-2 Example of Minimum-Length (20) Checking Experiment for D Flip-Flop.

Transition No.		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	0	1	1	0	0	1	0	0	1	0	0	0	1	1	1	1	0	1	1	1
D	1	1	0	0	1	1	1	0	0	0	1	0	0	1	0	1	1	1	0	1
Q	-	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1
State	4/5	7	9	3	5	7	5	3	8	2	4	2	8	6	8	6	4	7	9	7

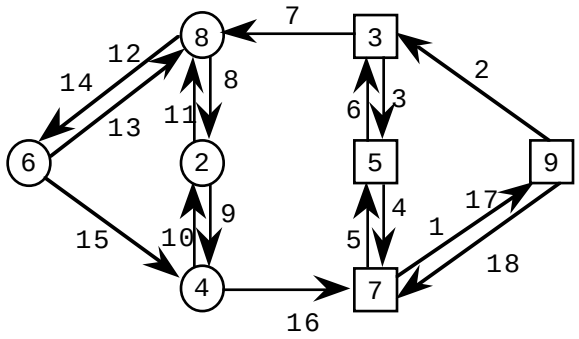


Figure 3.1.1-2 Sequence of Transitions in Table 3.1.1-2.

CD = 00, 10 ending in state 8. This raises the number of transitions to 19, and the number of patterns to 20. One minimum-length checking experiment is shown in Table 3.1.1-2. Figure 3.1.1-2 shows the transition graph annotated with the order of the sequence in Table 3.1.1-2.

3.1.2 MD Flip-Flop

A *Multiplexed-Data flip-flop (MD flip-flop)* is a D flip-flop with multiplexed data inputs. The primitive flow table for an MD flip-flop is shown in Table 3.1.2-1. As in the D flip-flop primitive flow table, this table has two states in each column, each with different outputs, and each stable state is the successor of another stable state. Therefore, just as in the case of the D flip-flop, a transition tour of the primitive flow table for an MD flip-flop is a checking experiment.

Since a checking experiment must apply all transitions, a minimum-length checking experiment must include each transition at least once. The number of transitions in Table 3.1.2-1 equals the number of total states multiplied by the number of inputs (32 x 4 = 128). To determine if the minimum length is 128, the transitions of Table 3.1.2-1 are represented graphically in Fig. 3.1.2-1 through 3.1.2-3. These graphs are not disjoint graphs, since some states appear in multiple graphs. They are drawn this way for readability. They include all the transitions in Table 3.1.2-1. From Fig. 3.1.2-3, eight

states (19,20,22,25,27,29,30,32) have one more outgoing edge than incoming edge. From graph theory, each of these would require at least one more transition [Wilson 72]. This raises the minimum length to 136 transitions.

There are eight possible synchronizing sequences shown below:

CTSD = 0000, 1000 ending in state 18 CTSD = 0001, 1001 ending in state 21

CTSD = 0010, 1010 ending in state 23 CTSD = 0011, 1011 ending in state 24

CTSD = 0100, 1100 ending in state 26 CTSD = 0101, 1101 ending in state 28

CTSD = 0110, 1110 ending in state 31 CTSD = 0111, 1111 ending in state 33

Including the synchronizing sequence raises the minimum length to 137 transitions (which implies 138 patterns). A minimum-length checking experiment is shown in Table 3.1.2-2. The details of deriving this checking experiment are in Appendix B.

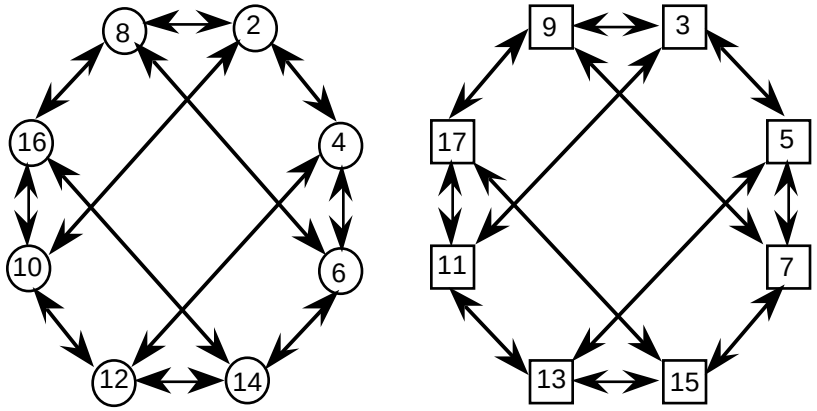


Figure 3.1.2-1 Graphs of Transitions Within First Quadrant for MD Flip-Flop.

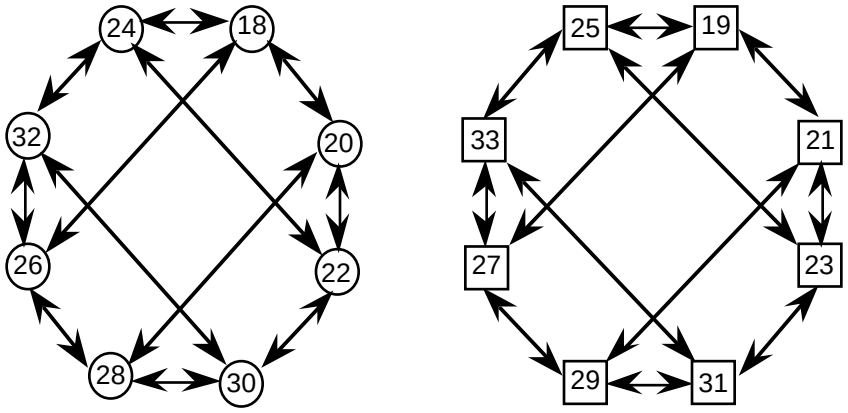


Figure 3.1.2-2 Graphs of Transitions Within Fourth Quadrant for MD Flip-Flop.

Table 3.1.2-1 Primitive Flow Table of MD-Flip-Flop.

	SD															
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10
	C=0								C=1							
	T=0				T=1				T=0				T=1			
2	(2) 0	4	–	8	10	–	–	–	18	–	–	–	–	–	–	–
3	[3] 1	5	–	9	11	–	–	–	18	–	–	–	–	–	–	–
4	2	(4) 0	6	–	–	12	–	–	–	21	–	–	–	–	–	–
5	3	[5] 1	7	–	–	13	–	–	–	21	–	–	–	–	–	–
6	–	4	(6) 0	8	–	–	14	–	–	–	23	–	–	–	–	–
7	–	5	[7] 1	9	–	–	15	–	–	–	23	–	–	–	–	–
8	2	–	6	(8) 0	–	–	–	16	–	–	–	24	–	–	–	–
9	3	–	7	[9] 1	–	–	–	17	–	–	–	24	–	–	–	–
10	2	–	–	–	(10) 0	12	–	16	–	–	–	–	26	–	–	–
11	3	–	–	–	[11] 1	13	–	17	–	–	–	–	26	–	–	–
12	–	4	–	–	10	(12) 0	14		–	–	–	–	–	28	–	–
13	–	5	–	–	11	[13] 1	15		–	–	–	–	–	28	–	–
14	–	–	6	–	–	12	(14) 0	16	–	–	–	–	–	–	31	–
15	–	–	7	–	–	13	[15] 1	17	–	–	–	–	–	–	31	–
16	–	–	–	8	10	–	14	(16) 0	–	–	–	–	–	–	–	33
17	–	–	–	9	11	–	15	[17] 1	–	–	–	–	–	–	–	33
18	2	–	–	–	–	–	–	–	(18) 0	20	–	24	26	–	–	–
19	3	–	–	–	–	–	–	–	[19] 1	21	–	25	27	–	–	–
20	–	4	–	–	–	–	–	–	18	(20) 0	22	–	–	28	–	–
21	–	5	–	–	–	–	–	–	19	[21] 1	23	–	–	29	–	–
22	–	–	6	–	–	–	–	–	–	20	(22) 0	24	–	–	30	–
23	–	–	7	–	–	–	–	–	–	21	[23] 1	25	–	–	31	–
24	–	–	–	8	–	–	–	–	18	–	22	(24) 0	–	–	–	32
25	–	–	–	9	–	–	–	–	19	–	23	[25] 1	–	–	–	33
26	–	–	–	–	10	–	–	–	18	–	–	–	(26) 0	28	–	32
27	–	–	–	–	11	–	–	–	19	–	–	–	[27] 1	29	–	33
28	–	–	–	–	–	12	–	–	–	20	–	–	26	(28) 0	30	–
29	–	–	–	–	–	13	–	–	–	21	–	–	27	[29] 1	31	–
30	–	–	–	–	–	–	14	–	–	–	22	–	–	28	(30) 0	32
31	–	–	–	–	–	–	15	–	–	–	23	–	–	29	[31] 1	33
32	–	–	–	–	–	–	–	16	–	–	–	24	26	–	30	(32) 0
33	–	–	–	–	–	–	–	17	–	–	–	25	27	–	31	[33] 1

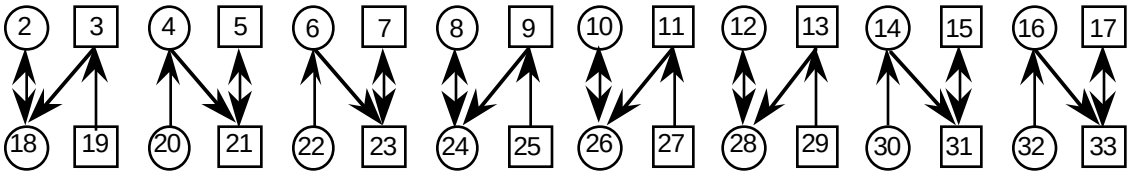


Figure 3.1.2-3 Graphs of Transitions Between Quadrants for MD Flip-Flop.

Table 3.1.2-2 Example of Minimum-Length (138) Checking Experiment for MD Flip-Flop.

C	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
T	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0
S	0	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	1	0	0	1
D	0	0	1	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1	0	0	1	0	0	1	1	1	1	1	1
Q		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State	-	18	20	22	30	28	26	32	24	18	20	28	20	18	26	18	24	22	24	32	30	32	26	28	30	22	20	4	6
C	1	1	0	1	1	0	0	0	0	1	0	0	1	1	0	1	1	0	0	0	1	0	0	0	0	1	0	0	0
T	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0	0
S	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1
D	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Q	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State	23	25	9	24	22	6	14	12	10	26	10	16	33	27	11	26	32	16	8	2	18	2	10	2	8	24	8	6	8
C	0	0	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
T	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0	0	0	0	1	1	1	1	0	0	0
S	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	1	1	0	0	0	0	1	1	0	0	1	1	0	0
D	0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	0	1
Q	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
State	16	14	31	29	13	28	30	14	16	10	12	4	12	28	12	14	6	4	2	4	21	23	31	29	27	33	25	19	21
C	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0
T	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1
S	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	0	0	0	1	1	0	0	1	1	1	1	0	0
D	1	1	0	0	0	0	1	0	0	1	0	0	1	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0
Q	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
State	29	21	19	27	19	25	23	25	33	31	33	27	29	31	23	21	19	3	5	7	15	13	11	17	33	17	9	3	11
C	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1							
T	0	0	0	0	1	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0							
S	0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0							
D	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0							
Q	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0							
State	3	9	7	9	17	15	31	15	17	11	13	5	13	15	7	23	7	5	21	5	3	18							

3.1.3 TP Flip-Flop

A *Two-Port flip-flop (TP flip-flop)* has two control inputs with the data source determined by the transitioning control input. C_1 and C_2 cannot be 1 at the same time. The primitive flow table for a TP flip-flops is shown in Table 3.1.3-1. As in the D flip-flop primitive flow table, this table has two states in each column, each with different outputs, and each stable state is the successor of another stable state. Therefore, just as in the case of the D flip-flop, a transition tour of the primitive flow table for a TP flip-flop is a checking experiment.

Since a checking experiment must apply all transitions, a minimum-length checking experiment must include each transition at least once. Normally the number of transitions equals the number of total states multiplied by the number of inputs. However, in Table 3.1.3-1, the last four columns are unspecified states because of the restriction that C_1 and C_2 cannot be 1 at the same time. Each of the first eight rows of the table has 4 transitions giving 32 (8 x 4) transitions for the first 8 rows, while the rest of the rows have only 3 transitions per row, giving 48 (16 x 3) for the rest of the rows. Therefore, the flow table has 80 (32 + 48) transitions. To determine if the minimum length is 80, the transitions of Table 3.1.3-1 are represented graphically in Fig. 3.1.3-1 and 3.1.3-2. These graphs are not disjoint graphs, since some states appear in multiple graphs. They are drawn this way for readability. They do include all the transitions in Table 3.1.3-1. From Fig. 3.1.3-2, eight states (11,12,14,16,19,20,22,25) have one more outgoing edge than incoming edge. From graph theory, each of these would require at least one more transition [Wilson 72]. This raises the minimum length to 88 transitions.

There are eight possible synchronizing sequences shown below:

$C_1C_2D_1D_2$ = 0000, 1000 end in state 10	$C_1C_2D_1D_2$ = 0000, 0100 end in state 18
$C_1C_2D_1D_2$ = 0001, 1001 end in state 12	$C_1C_2D_1D_2$ = 0001, 0101 end in state 21
$C_1C_2D_1D_2$ = 0010, 1010 end in state 17	$C_1C_2D_1D_2$ = 0010, 0110 end in state 24
$C_1C_2D_1D_2$ = 0011, 1011 end in state 15	$C_1C_2D_1D_2$ = 0011, 0111 end in state 23

Including the synchronizing inputs raises the minimum length to 89 transitions (which implies 90 patterns). A minimum-length checking experiment is shown in Table 3.1.3-2. The details of deriving this checking experiment are in Appendix B.

Table 3.1.3-1 Primitive Flow Table of TP Flip-Flop.

	D_1D_2															
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10
	C_1C_2=00				C_1C_2=10				C_1C_2=01				C_1C_2=11			
2	(2) 0	4	–	8	10	–	–	–	18	–	–	–	–	–	–	–
3	[3] 1	5	–	9	10	–	–	–	18	–	–	–	–	–	–	–
4	2	(4) 0	6	–	–	12	–	–	–	21	–	–	–	–	–	–
5	3	[5] 1	7	–	–	12	–	–	–	21	–	–	–	–	–	–
6	–	4	(6) 0	8	–	–	15	–	–	–	23	–	–	–	–	–
7	–	5	[7] 1	9	–	–	15	–	–	–	23	–	–	–	–	–
8	2	–	6	(8) 0	–	–	–	17	–	–	–	24	–	–	–	–
9	3	–	7	[9] 1	–	–	–	17	–	–	–	24	–	–	–	–
10	2	–	–	–	(10) 0	12	–	16	–	–	–	–	–	–	–	–
11	3	–	–	–	[11] 1	13	–	17	–	–	–	–	–	–	–	–
12	–	4	–	–	10	(12) 0	14		–	–	–	–	–	–	–	–
13	–	5	–	–	11	[13] 1	15		–	–	–	–	–	–	–	–
14	–	–	6	–	–	12	(14) 0	16	–	–	–	–	–	–	–	–
15	–	–	7	–	–	13	[15] 1	17	–	–	–	–	–	–	–	–
16	–	–	–	8	10	–	14	(16) 0	–	–	–	–	–	–	–	–
17	–	–	–	9	11	–	15	[17] 1	–	–	–	–	–	–	–	–
18	2	–	–	–	–	–	–	–	(18) 0	20	–	24	–	–	–	–
19	3	–	–	–	–	–	–	–	[19] 1	21	–	25	–	–	–	–
20	–	4	–	–	–	–	–	–	18	(20) 0	22	–	–	–	–	–
21	–	5	–	–	–	–	–	–	19	[21] 1	23	–	–	–	–	–
22	–	–	6	–	–	–	–	–	–	20	(22) 0	24	–	–	–	–
23	–	–	7	–	–	–	–	–	–	21	[23] 1	25	–	–	–	–
24	–	–	–	8	–	–	–	–	18	–	22	(24) 0	–	–	–	–
25	–	–	–	9	–	–	–	–	19	–	23	[25] 1	–	–	–	–

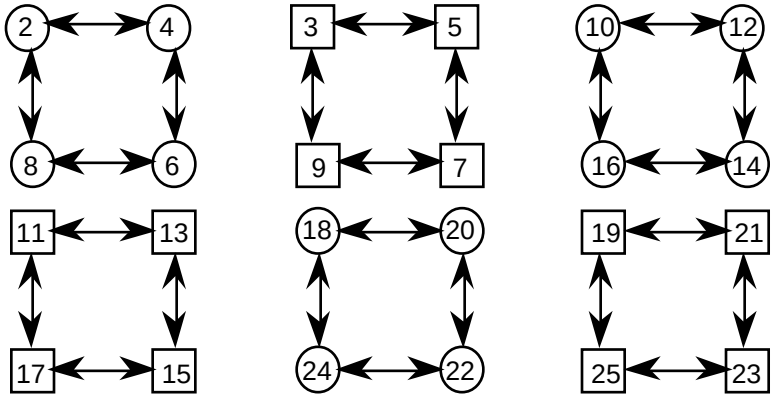


Figure 3.1.3-1 Graphs of Transitions Within Quadrants for TP Flip-Flop.

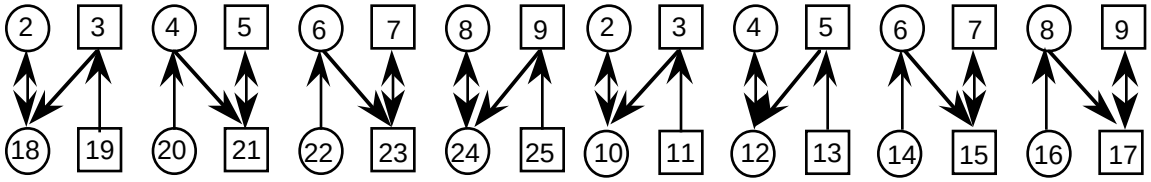


Figure 3.1.3-2 Graphs of Transitions Between Quadrants for TP Flip-Flop.

Table 3.1.3-2 Example of Minimum-Length (90) Checking Experiment for TP Flip-Flop.

C_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		1	0	1	1	1	1	1
D_1	0	0	0	1	1	0	0	0	1	1	0	0	1	1	1	0	0	1	1	0	1	1	0	0	0	0	0	1	1	0
C_2	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D_2	0	0	1	1	0	0	1	0	0	1	1	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0
Q		0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
State	-	18	20	22	24	18	20	18	24	22	20	4	6	8	17	11	13	15	17	11	17	15	13	11	3	10	12	14	16	10
C_1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0
D_1	1	1	0	0	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0
C_2	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	1	1	1
D_2	0	1	1	0	0	0	0	1	0	0	1	1	1	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	0	0
Q	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
State	16	14	12	10	16	8	2	4	2	8	6	4	21	19	21	23	25	9	24	22	6	15	13	5	12	14	6	23	25	19
C_1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
D_1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0
C_2	1	1	1	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1
D_2	0	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0
Q	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
State	25	23	21	19	3	5	21	5	13	5	7	9	25	9	17	9	3	9	7	23	7	15	7	5	3	19	3	11	3	18

3.2 Shift Register

A shift register is a circuit that, once every cycle, shifts its stored contents. A *cycle* is defined as the time between two rising edges of the clock. Fig. 3.2-1 shows an implementation using D flip-flops. Every time a positive transition appears on the C input of the circuit, the data on d1 is transferred to d2, the data on d2 is transferred to d3, etc. In Section 3.1.1, we derived a checking experiment for a D flip-flop with the assumption that the inputs can be changed at any time, as long as only one input changes at a time, and that the output is always observable. Now consider the highlighted flip-flop, D3, in the shift register of Fig. 3.2-1. Changing the value on d3 (the D input of the flip-flop) depends on C changing value. Since the output is not a primary output, the output of the flip-flop under test needs to be propagated to the primary output (d7). This can only be done by D4 "capturing" the output of D3, and then shifting the contents of D4 to the primary output. In this section, we will first analyze the effect of these restrictions on the operation of the flip-flop under test. From this analysis, we derive a new flow table for an embedded flip-flop in a shift register.

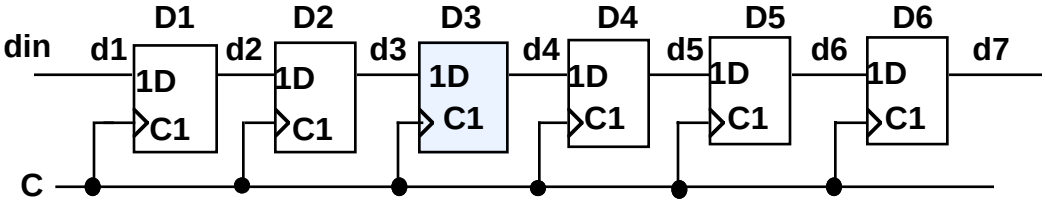


Figure 3.2-1 Shift Register Constructed from D Flip-Flops.

The simplest way to explain the constraints on the D flip-flop is to analyze the inputs and outputs of the flip-flop during one cycle. Fig. 3.2-2 shows the four events that occur in a sample cycle. In this sample cycle, we start with d2 = 0, d3 = 1, and d4 = 1. At the end of the cycle, by the definition of the shift register, d3 should be 0, and d4 should be 1. At t_1, C goes from 0 to 1. This causes the d3 input to change to 0 at time t_2, because d2 is 0 at t_1. At t_3, C goes back to 0, and at t_4 it goes back to 1, starting the next cycle. At that point, t_4, D4 "captures" the 1 on d4. This point in the cycle, called the *capture point*, is the only point in the cycle when the value on d4 can be "captured" by D4. The captured value can be observed at the shift register primary output by applying two more clock cycles. Within the cycle, d3 changed only once (at t_2), and it changed only when C = 1. Thus, one restriction on the operation of the flip-flop is that the D input can only change when C = 1, and it can change at most once. Since D4 "captures" the output of D3 when C is changing (t4), it may seem ambiguous as to whether D4 captured the output of state 3 or

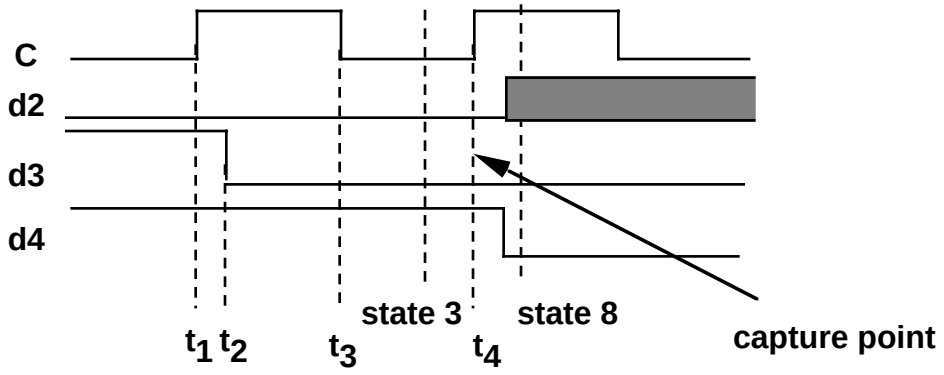


Figure 3.2-2 Sequence of Operations in Cycle.

state 8. Looking at the waveforms in Fig. 3.2-2, since the captured value is 1, we captured the output of state 3. In general, we can only capture the outputs of states when C = 0. This analysis implies three constraints on the operation of the D flip-flop in the shift register:

- the D input cannot change when C = 0.
- the D input can change only once when C = 1.
- the Output of the flip-flop is "observed" only for states with C = 0.

We can use these constraints to remove outputs that cannot be observed and impossible transitions from the flow table (Table 3.2-1). The reduced primitive flow table is shown in Table 3.2-2. Fig. 3.2-3 shows the transition diagram of this reduced flow table. In this figure, stable states that have observable outputs are shaded. Thick edges are used to indicate distinguishing sequences.

Table 3.2-1 Marked-Up Primitive Flow Table for D Flip-Flop in a Shift Register.

	C D 00	01	11	10
2	(2),0	~~4~~	–	8
3	[3],1	~~5~~	–	8
4	~~2~~	(4),0	7	–
5	~~3~~	[5],1	7	–
6	–	4	(6), ~~0~~	~~8~~
7	–	5	[7], ~~1~~	9
8	2	–	6	(8), ~~0~~
9	3	–	~~7~~	[9], ~~1~~

~~X~~ D cannot change when C = 0

D changes only once when C = 1

~~\~~ Outputs not observed when C = 1

Table 3.2-2 Reduced Primitive Flow Table for D Flip-Flop in a Shift Register.

	CD = 00	01	11	10
2	(2),0	–	–	8
3	[3],1	–	–	8
4	–	(4),0	7	–
5	–	[5],1	7	–
6	–	4	(6)	–
7	–	5	[7]	9
8	2	–	6	(8)
9	3	–	–	[9]

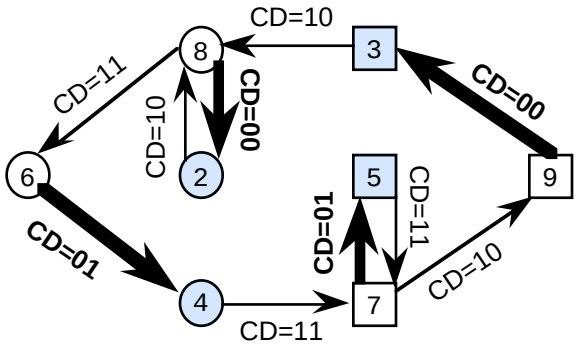


Figure 3.2-3 Transition Graph for Internal Flip-Flop (Table 3.2-2) Showing Distinguishing Sequences.

Now that we have a new flow table we need to derive a checking experiment for it. In general, since every stable state is the successor of at least one unstable state, we need to visit each unstable state (edge in the graph) and follow it with a distinguishing sequence to identify the stable successor state. In the original flow table, each column had two stable states with different outputs, and thus the distinguishing sequence was a null sequence (no extra inputs need to be applied). Visiting all the transitions once was sufficient for a checking experiment.

However, in this case, the states in the C = 1 columns do not have any observable outputs. For these states, the distinguishing sequence is one input that takes the machine to a state with C = 0, where the output can be observed. For example, state 8 has no observable output, so with an input of CD = 10 we cannot tell if the machine is in state 8 or state 9. Following state 8 with state 2 (by applying CD = 00) will identify state 8. If state

8 is followed by state 6 (by applying CD = 11) then it is not identified because the output is not observed. Thus, any path that traverses all edges, and has each edge into node 8 (7) followed by node 2 (5) corresponds to a checking experiment. This implies that the following sub-sequences are needed in any checking experiment: 2->8->2, 8->6->4, 4->7->5, 5->7->5, 7->9->3 and 3->8->2. These sub-sequences are combined into the single sequence shown in Table 3.2-3. In this table, the corresponding inputs and outputs are also included. A * indicates that the output cannot be observed.

Table 3.2-3 Example of Checking Experiment for D Flip-Flop in Shift Register.

Transition No.		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	0	1	0	1	0	1	1	0	1	0	1	0	1	1	0	1	0
D	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0
Q	*	*	0	*	0	*	*	0	*	1	*	1	*	*	1	*	0
State	2/3	8	2	8	2	8	6	4	7	5	7	5	7	9	3	8	2

Saxena [93], used a data transition tour test based on a pulse mode view of the D flip-flop. Such a test guarantees that all four data transitions (1->1, 1->0, 0->0 and 0->1) occur. One example of such a test is shown in Table 3.2-4. This table also shows the response of the circuit with the flow table in Table 3.2-5. Since the same input produces the same output when applied to the two flow tables, the sequence is not a checking experiment. The test fails to be a checking experiment because the unstable transition from state 2 to state 8 is never followed by a distinguishing sequence.

Table 3.2-4 Transition Tour Responses from D Flip-Flop and Table 3.2-5.

	Transition No.	1	2	3	4	5	6	7	8	9	10		
	C	0	1	0	1	1	0	1	0	1	1	0	1	0
	D	1	1	1	1	0	0	0	0	0	1	1	1	1
From Table 3.2-2	Output	-	*	1	*	*	1	*	0	*	*	0	*	1
	State	-	7	5	7	9	3	8	2	8	6	4	7	5
From Table 3.2-5	Output	-	*	1	*	*	1	*	0	*	*	0	*	1
	State	-	7	5	7	9	3	8	2	9	6	4	7	5

Table 3.2-5 Flow Table That Gives Same Response as Data Transition Tour to Flip-Flop.

	C D 00	01	11	10
2	(2),0	–	–	9
3	[3],1	–	–	8
4	–	(4),0	7	–
5	–	[5],1	7	–
6	–	4	(6)	9
7	–	5	[7]	9
8	2	–	6	(8)
9	3	–	6	[9]

The restrictions imposed by the shift register apply to all the internal flip-flops but not the first and last flip-flops in the shift register. For the first flip-flop, there are no constraints on how the input is applied, since the inputs are primary inputs to the circuit. For the last flip-flop, the output is always observed since it is a primary output. In the rest of this section, we look at what effect these reduced constraints have on the checking experiments.

For the last flip-flop in the shift register, the output is always observed, but the other two constraints still hold. The primitive flow table of the last flip-flop is shown in Table 3.2-6. The only difference between this table and the flow table of an internal flip-flop (Table 3.2-2) is that the C = 1 column has observable outputs. This implies that any sequence that is a checking experiment for an internal flip-flop is also a checking experiment for the last flip-flop. There are sequences that are checking experiments for the last flip-flop that are not checking experiments for the internal flip-flops, but since testing the internal flip-flops applies the same checking experiment to the last flip-flop, we need not generate such a test.

Table 3.2-6 Primitive Flow Table for Last D Flip-Flop in a Shift Register.

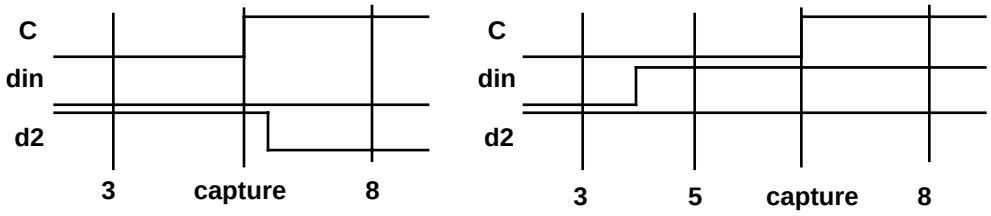
	CD 00	01	11	10
2	(2),0	–	–	8
3	[3],1	–	–	8
4	–	(4),0	7	–
5	–	[5],1	7	–
6	–	4	(6),0	–
7	–	5	[7],1	9
8	2	–	6	(8),0
9	3	–	–	[9],1

For the first flip-flop in the shift register, D and C are both primary inputs. Therefore, there are no constraints on when C and D change, as long as they do not change simultaneously. The output of the first flip-flop is captured by the second flip-flop when C changes from 0 to 1. Therefore, the output when C = 0 can only be observed by changing C to 1. The primitive flow table for the first flip-flop is shown in Table 3.2-7. The difference between the flow table of the first flip-flop (Table 3.2-7) and the internal flip-flop (Table 3.2-2) is that the first flip-flop has transitions than are not available to the internal flip-flop. The outputs are the same for both. We can identify the common transitions using the same sequence we derived for the internal flip-flop. Therefore, we will focus our attention on identifying the transitions of the first flip-flop that are not available for the internal flip-flop.

Since the second flip-flop captures the output of the first flip-flop when C changes from 0 to 1, we will only observe the output of a state if it has C = 0 and its successor state has C = 1. The transition between the two states is called the *identifying transition*. For example, if state 3 is followed by state 8, we will capture the output of state 3, and the transition from state 3 to state 8 is an identifying transition. If state 3 if followed by state 5, then the output of state 3 is not captured. This is illustrated in the waveforms of Fig. 3.2-4.

Table 3.2-7 Primitive Flow Table of First D Flip-Flop in Shift Register.

	CD 00	01	11	10
2	(2),0	4	–	8
3	[3],1	5	–	8
4	2	(4),0	7	–
5	3	[5],1	7	–
6	–	4	(6)	8
7	–	5	[7]	9
8	2	–	6	(8)
9	3	–	7	[9]



(a) State 3 Output Captured **(b) State 3 Output Not Captured**

Figure 3.2-4 Waveforms Illustrating Capture of Output of First D Flip-Flop.

The identifying transitions are shown in the transition graph, Fig. 3.2-5a, with thick arrows. There are two main differences between this transition graph and that of the internal flip-flops, Fig. 3.2-5b. First, in Fig. 3.2-5b there is only one edge out of the C = 0 states, and the successor states have C = 1. Therefore, all transitions out of the C = 0 states in Fig. 3.2-5b are identifying transitions. For the transition graph of Fig. 3.2-5a we need to ensure that every edge into 2,3 (4,5) is followed by node 8 (7). The second difference between Fig 3.2-5a and Fig. 3.2-5b is that Fig. 3.2-5a contains transitions that do not exist in Fig. 3.2-5b. Our main concern is identifying the transitions in Fig. 3.2-5a that are not in Fig. 3.2-5b. In Fig. 3.2-5a, these transitions are shown with a dashed edge. These are divided into two groups: 3->5, 5->3, 2->4, 4->2; and 9->7, 6->8.

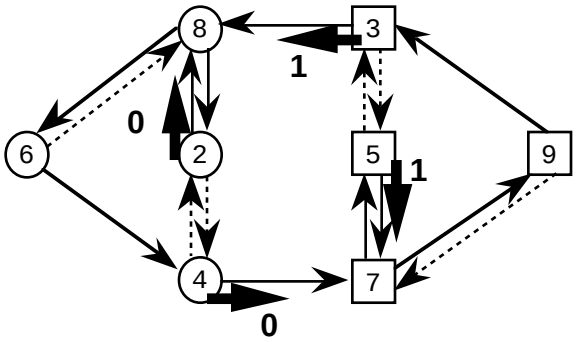


Figure 3.2-5a Transition Graph for the First D Flip-Flop of Shift Register.

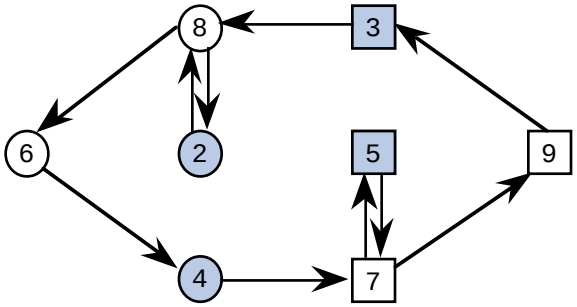


Figure 3.2-5b Transition Graph for the Internal D Flip-Flop of Shift Register.

The first group of transitions (3->5, 5->3, 2->4, 4->2) end in states that are followed by identifying transitions. The final states of these transitions can be identified by following them with the appropriate identifying transition. For example, consider the 3->5 transition. This transition ends in state 5, and should thus be followed by the identifying transition 5->7. Of course, we must also identify the starting state of the transitions (state S_i is the *starting state* of transition S_i -> S_j). Now, since all identifying transitions end in state 7 or state 8, we need to show how to get to the starting state from one of these two states. In our example, 3->5, there are two ways to get to state 3 from state 7:7->9->3 and 7->5->3. One of the sub-sequences that was used for the internal latch (and we will retain for the first latch) was 7->9->3->8. If we apply the first two inputs that correspond to this sequence, we know that we will be in state 3, without having to look at the output. Thus, the sub-sequence 7->9->3->5->7, will identify the 3->5 transition. A similar analysis can be applied for identifying the other three transitions. A checking experiment requires the following sub-sequences: 7->9->3->5->7, 8->6->4->2->8, 7->5->3->8, 8->2->4->7.

The second group of transitions (9->7 and 6->8) are not immediately followed by identifying transitions. Consider the 9->7 transition. Since there is no identifying transition out of state 7, we follow this transition with state 5 giving us 9->7->5. Following state 7 with state 5, will set the machine up for the 5->7 identifying transition.

Since the only predecessor of state 9 is state 7, we can extend the sequence to 7->9->7->5. Adding the identifying transition to this sequence gives 7->9->7->5->7. In this whole sequence, the only observed output is that of state 5. One of the sub-sequences that was used for the internal latch (and we will retain for the first latch) was 7->9->3->8. If we apply the first input that corresponds to this sequence, we know that we will be in state 9, without having to look at the output. The internal latch sub-sequences also included 5->7->5. This implies that state 7 is the only predecessor of state 5 with CD = 11. Therefore, if we apply CD = 11 followed by CD = 10 and observe a 1 (identifying state 5), then we know that the previous state was state 7. Since all the states in the sub-sequence (7->9->7->5->7) are identified, the 9->7 transition is identified. Similarly, the sub-sequence 8->6->8->2->8 would identify the 6->8 transition. In summary, a checking experiment requires the following sub-sequences: 7->9->7->5->7 and 8->6->8->2->8.

The sub-sequences required to identify the transitions of both groups are combined with the sub-sequences needed for the internal latch to form a checking experiment in Table 3.2-8. Of course, in some systems, even though C and din are primary inputs, the constraints discussed for the internal flip-flops may still apply to the first flip-flop. In such cases, the checking experiment of Table 3.2-3 can be used.

Table 3.2-8 Example of Checking Experiment for the First D Flip-Flop in Shift Register.

Transition No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	0	1	0	1	0	1	1	0	1	0	1	0	1	1	0	1	0	1	1	1
D	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	1	0
Q	*	*	0	*	0	*	*	0	*	1	*	1	*	*	1	*	0	*	*	*
State	2/3	8	2	8	2	8	6	4	7	5	7	5	7	9	3	8	2	8	6	8

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
C	0	1	1	0	0	1	0	0	1	1	0	0	1	1	1	0	1	0	0	1
D	0	0	1	1	0	0	0	1	1	0	0	1	1	0	1	1	1	1	0	0
Q	0	*	*	*	0	*	*	0	*	*	*	1	*	*	*	1	*	*	1	*
State	2	8	6	4	2	8	2	4	7	9	3	5	7	9	7	5	7	5	3	8

3.2.1 Primary Input/Output View

The analysis so far has been from the point of view of the individual D flip-flop. By picking an arbitrary flip-flip in the middle of the scan chain, the results obtained apply to any flip-flop within the scan chain. We now look at what these results imply at the primary inputs and output of the whole shift register. At that level, the data input is applied once per cycle. The waveforms of the checking experiment of Table 3.2-3 are shown in

Fig. 3.2.1-1. In this figure, the data values at the output of the flip-flop under test are marked on the rising edge of the clock. These are the data values that need to be applied to the din input of the shift register. One input is applied every cycle. Thus, the data sequence is din = 0001110.

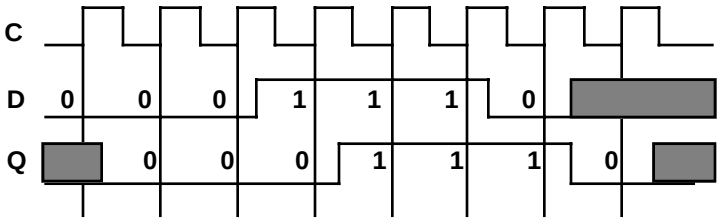


Figure 3.2.1-1 Waveforms of Checking Experiment in Table 3.2-3 for Flip-Flop in Shift Register Showing Data Values.

From the definition of the shift register, data applied at the din input is applied to all the D inputs of the flip-flops if enough clock cycles are applied. Therefore, the checking experiment data sequence, 0001110, can be applied to all flip-flops using the waveforms in Fig. 3.2.1-2. If we have a shift register with only one flip-flop, then we need seven cycles for the seven patterns to go through the shift register. For each additional flip-flop in the shift register we need one extra cycle. Therefore , we need N+6 cycles for a shift register of length N.

The waveform in Fig. 3.2.1-2 does not address the extra patterns we needed for the first flip-flop. In some designs, constraints may even exist on the primary inputs. For example, din may only be allowed to change when C is high. In that case, the same waveforms of Fig. 3.2.1-2 should be used. If there are no constraints on C and din, we

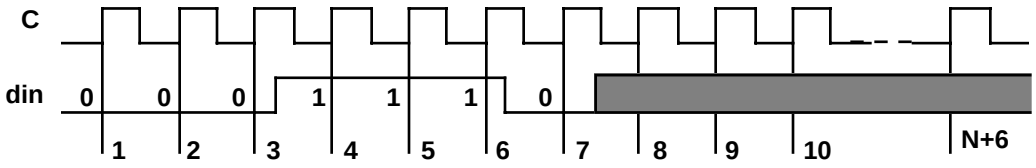


Figure 3.2.1-2 Test for Shift Register Flip-Flops (N = Number of Flip-Flops).

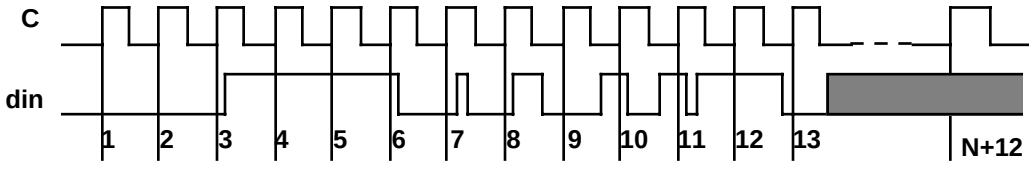


Figure 3.2.1-3 Test for Shift Register Flip-Flops With No Constraints on C and din.

need to add extra patterns to apply a checking experiment to the first flip-flop. This can be done using the waveforms in Fig. 3.2.1-3. Here, din changes multiple times in a cycle and changes at different times in the cycle, making it impossible to express the test as a cycle-based data sequence. An additional 6 cycles are needed for this test.

3.2.2 Reduced Flow Table for Embedded Flip-Flops

Since we have a new flow table for the flip-flip embedded in a shift register, it is interesting to see what the table can be reduced to. We know that, in general, a D flip-flop requires at least four states [McCluskey 86]. However, with the restricted flow table, we can derive a flip-flop with only three states. From the distinguishablity array in Fig. 3.2.2-1 (see [McCluskey 86] for definitions and procedures) the maximal compatibility classes are [6,8,2] [9,7,5] [3,4] [6,9] [4,9] [3,5] [3,6] [2,4] [2,5]. The first three classes include all the states in the primitive flow table. The resulting flow table is shown in Table 3.2.2-1. Unfortunately, there is no three state implementation that is more efficient than an ordinary four-state implementation.

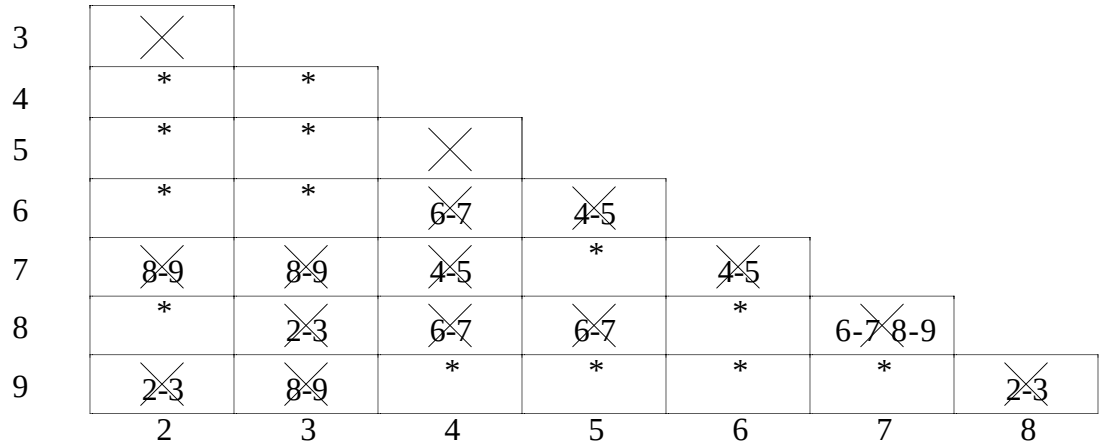


Figure 3.2.2-1 Distinguishablity Array for D Flip-Flop Restricted Flow Table.

Table 3.2.2-1 Reduced Flow Table for Shift Register D Flip-Flop.

	C D 00	01	11	10
[6,8,2] a	(a),0	c	(a),0	(a),0
[9,7,5] b	c	[b],1	[b],1	[b],1
[3,4] c	(c),0	[c],1	b	a

3.3 Flip-Flop Based Scan Chains

Chapter 1 introduced scan chains as a method of simplifying the test generation process for sequential circuits. In this section, we focus on generating tests for the flip-flops in the two flip-flop-based architectures, MD flip-flop architecture and TP flip-flop architecture. For each of the architectures, we describe a method to generate test patterns that apply a checking experiment to the flip-flops in the design. In Section 3.3.1 we analyze the MD flip-flop architecture, and in Section 3.3.2 we analyze the TP flip-flop architecture. In Chapter 4, we use the information developed here to describe an algorithm to generate test patterns to test the flip-flops in a scan design.

3.3.1 MD Flip-Flop Scan Chain

In Section 3.1.2, we described how a checking experiment can be derived for an MD flip-flop, with the assumptions that the inputs can be changed at any time, as long as only one input changes at a time, and that the output is always observable. When an MD flip-flop is used as part of a scan chain (see Fig. 3.3.1-1), the S input of the MD flip-flop is not directly controllable from the primary inputs. The S input of the MD flip-flop is the output of another flip-flop, and the outputs of flip-flops change only when CK changes to 1. Therefore the S input can only change when CK = 1. Since flip-flops are edge-triggered devices, the output of a flip-flop will change only once every cycle (A cycle is the time between successive positive transitions of CK). Therefore, the S input can change only once every cycle. The D input has similar restrictions to S if all the inputs of the combinational logic driving it are flip-flop outputs. If some of the inputs are primary inputs, then it may be possible to change the value on D without a positive transition on CK. The T and CK inputs can change at any time since they are primary inputs.

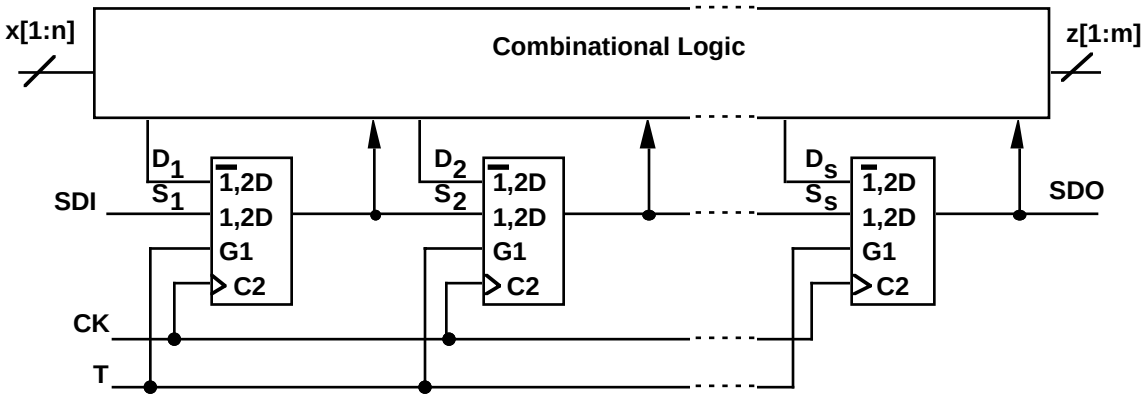


Figure 3.3.1-1 MD Flip-Flop Based Scan Chain.

The output of the flip-flop under test is captured by the next flip-flop in the scan chain when T = 1. When T = 0, the output of the flip-flop under test can be captured by one of the other flip-flops in the design if a path is sensitized from the output of the flip-flop under test to the D input of another flip-flop. Capturing the outputs is discussed in more detail later in this section. In both cases, the output is captured on a positive transition of CK. The captured output corresponds to the state with CK = 0, since the "capturing flip-flop" captures the output of the flip-flop under test before the clock has a chance to change the state of the flip-flop under test (assuming hold times are honored).

The restrictions on the MD flip-flop in the scan chain can be summarized as follows:

- the S input can only change when CK = 1.
- the S input can change only once per cycle.
- the output of the flip-flop is observable only for states with CK = 0.
- the output is captured on a positive transition of CK.
- the T input can change at any time since it is controlled from the primary input.

These restrictions make some of the total states unreachable. Table 3.3.1-1 shows the primitive flow table with unreachable states covered with X. Outputs that cannot be observed are covered with \. The reduced primitive flow table is shown in Table 3.3.1-2. If an MD flip-flop is replaced by a device that performs the operation described by this flow table, the circuit will still operate correctly. The transition diagrams for the reduced flow table are shown in Fig. 3.3.1-2 through 3.3.1-4.

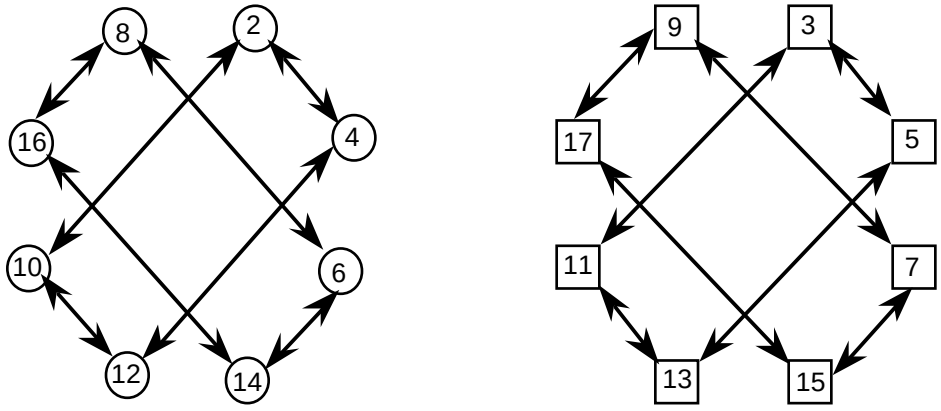


Figure 3.3.1-2 Graphs of Transitions Within First Quadrant for Scan Chain MD Flip-Flop.

Table 3.3.1-1 Marked-Up Primitive Flow Table of MD Flip-Flop in Scan Chain.

	SD															
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10
	C=0								C=1							
	T=0				T=1				T=0				T=1			
2	(2) 0	4	–	~~8~~	10	–	–	–	18	–	–	–	–	–	–	–
3	[3] 1	5	–	~~9~~	11	–	–	–	18	–	–	–	–	–	–	–
4	2	(4) 0	~~6~~	–	–	12	–	–	–	21	–	–	–	–	–	–
5	3	[5] 1	~~7~~	–	–	13	–	–	–	21	–	–	–	–	–	–
6	–	~~4~~	(6) 0	8	–	–	14	–	–	–	23	–	–	–	–	–
7	–	~~5~~	[7] 1	9	–	–	15	–	–	–	23	–	–	–	–	–
8	~~2~~	–	6	(8) 0	–	–	–	16	–	–	–	24	–	–	–	–
9	~~3~~	–	7	[9] 1	–	–	–	17	–	–	–	24	–	–	–	–
10	2	–	–	–	(10) 0	12	–	~~16~~	–	–	–	–	26	–	–	–
11	3	–	–	–	[11] 1	13	–	~~17~~	–	–	–	–	26	–	–	–
12	–	4	–	–	10	(12) 0	~~14~~	–	–	–	–	–	–	28	–	–
13	–	5	–	–	11	[13] 1	~~15~~	–	–	–	–	–	–	28	–	–
14	–	–	6	–	–	~~12~~	(14) 0	16	–	–	–	–	–	–	31	–
15	–	–	7	–	–	~~13~~	[15] 1	17	–	–	–	–	–	–	31	–
16	–	–	–	8	~~10~~	–	14	(16) 0	–	–	–	–	–	–	–	33
17	–	–	–	9	~~11~~	–	15	[17] 1	–	–	–	–	–	–	–	33
18	2	–	–	–	–	–	–	–	(18) 0	20	–	24	26	–	–	–
19	3	–	–	–	–	–	–	–	[19] 1	21	–	~~25~~	27	–	–	–
20	–	4	–	–	–	–	–	–	18	(20) 0	~~22~~	–	–	28	–	–
21	–	5	–	–	–	–	–	–	19	[21] 1	23	–	–	29	–	–
22	–	–	6	–	–	–	–	–	–	~~20~~	(22) 0	24	–	–	30	–
23	–	–	7	–	–	–	–	–	–	21	[23] 1	25	–	–	31	–
24	–	–	–	8	–	–	–	–	18	–	22	(24) 0	–	–	–	32
25	–	–	–	9	–	–	–	–	~~19~~	–	23	[25] 1	–	–	–	33
26	–	–	–	–	10	–	–	–	18	–	–	–	(26) 0	28	–	32
27	–	–	–	–	11	–	–	–	19	–	–	–	[27] 1	29	–	~~33~~
28	–	–	–	–	–	12	–	–	–	20	–	–	26	(28) 0	30	–
29	–	–	–	–	–	13	–	–	–	21	–	–	27	[29] 1	~~31~~	–
30	–	–	–	–	–	–	14	–	–	–	22	–	–	~~28~~	(30) 0	32
31	–	–	–	–	–	–	15	–	–	–	23	–	–	29	[31] 1	33
32	–	–	–	–	–	–	–	16	–	–	–	24	~~26~~	–	30	(32) 0
33	–	–	–	–	–	–	–	17	–	–	–	25	27	–	31	[33] 1

Table 3.3.1-2 Reduced Primitive Flow Table of MD Flip-Flop in Scan Chain.

	SD															
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10
	C=0								C=1							
	T=0				T=1				T=0				T=1			
2	(2) 0	4	–	–	10	–	–	–	18	–	–	–	–	–	–	–
3	[3] 1	5	–	–	11	–	–	–	18	–	–	–	–	–	–	–
4	2	(4) 0	–	–	–	12	–	–	–	21	–	–	–	–	–	–
5	3	[5] 1	–	–	–	13	–	–	–	21	–	–	–	–	–	–
6	–	–	(6) 0	8	–	–	14	–	–	–	23	–	–	–	–	–
7	–	–	[7] 1	9	–	–	15	–	–	–	23	–	–	–	–	–
8	–	–	6	(8) 0	–	–	–	16	–	–	–	24	–	–	–	–
9	–	–	7	[9] 1	–	–	–	17	–	–	–	24	–	–	–	–
10	2	–	–	–	(10) 0	12	–	–	–	–	–	–	26	–	–	–
11	3	–	–	–	[11] 1	13	–	–	–	–	–	–	26	–	–	–
12	–	4	–	–	10	(12) 0	–	–	–	–	–	–	–	28	–	–
13	–	5	–	–	11	[13] 1	–	–	–	–	–	–	–	28	–	–
14	–	–	6	–	–	–	(14) 0	16	–	–	–	–	–	–	31	–
15	–	–	7	–	–	–	[15] 1	17	–	–	–	–	–	–	31	–
16	–	–	–	8	–	–	14	(16) 0	–	–	–	–	–	–	–	33
17	–	–	–	9	–	–	15	[17] 1	–	–	–	–	–	–	–	33
18	2	–	–	–	–	–	–	–	(18)	20	–	24	26	–	–	–
19	3	–	–	–	–	–	–	–	[19]	21	–	–	27	–	–	–
20	–	4	–	–	–	–	–	–	18	(20)	–	–	–	28	–	–
21	–	5	–	–	–	–	–	–	19	[21]	23	–	–	29	–	–
22	–	–	6	–	–	–	–	–	–	–	(22)	24	–	–	30	–
23	–	–	7	–	–	–	–	–	–	21	[23]	25	–	–	31	–
24	–	–	–	8	–	–	–	–	18	–	22	(24)	–	–	–	32
25	–	–	–	9	–	–	–	–	–	–	23	[25]	–	–	–	33
26	–	–	–	–	10	–	–	–	18	–	–	–	(26)	28	–	32
27	–	–	–	–	11	–	–	–	19	–	–	–	[27]	29	–	–
28	–	–	–	–	–	12	–	–	–	20	–	–	26	(28)	30	–
29	–	–	–	–	–	13	–	–	–	21	–	–	27	[29]	–	–
30	–	–	–	–	–	–	14	–	–	–	22	–	–	–	(30)	32
31	–	–	–	–	–	–	15	–	–	–	23	–	–	29	[31]	33
32	–	–	–	–	–	–	–	16	–	–	–	24	–	–	30	(32)
33	–	–	–	–	–	–	–	17	–	–	–	25	27	–	31	[33]

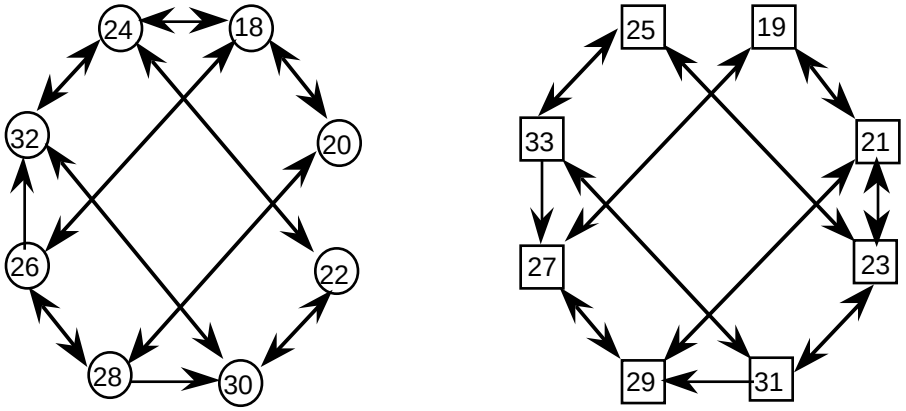


Figure 3.3.1-3 Graphs of Transitions Within Fourth Quadrant for Scan Chain MD Flip-Flop.

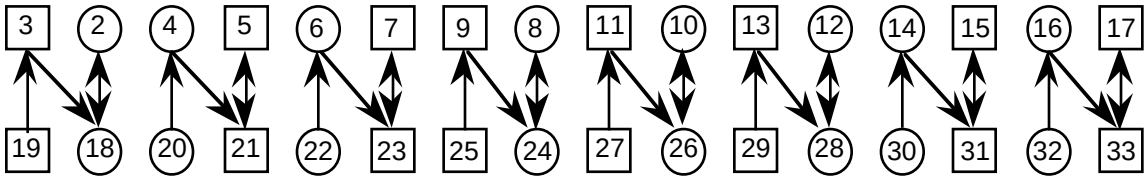


Figure 3.3.1-4 Graphs of Transitions Between Quadrants for Scan Chain MD Flip-Flop.

Now that we have a new flow table, we need to derive a checking experiment for it. In general, we need to visit each unstable state (edge in the graph) and follow it with a distinguishing sequence. In the original flow table of the MD flip-flop (see Table 3.1.2-1), each column had two stable states with different outputs, and thus the distinguishing sequence was a null sequence (no extra inputs need to be applied). Visiting all the transitions once was sufficient for a checking experiment.

However, in the case of Table 3.3.1-1 the outputs of states in the C = 1 columns cannot be observed. For these states, the distinguishing sequence is one input that takes the flip-flop to a state with C = 0, where the output can be observed. For example, state 18 has no observable output, so with an input of CTSD = 1000, we cannot tell if the flip-flop is in state 18 or state 19. Following state 18 with state 2 (by applying CTSD = 0000) will identify state 18.

To capture the output of state 2, the next transition should be CK changing back to 1 with T remaining 0. Such a transition, which will cause the capture of the output of the flip-flop, is called a *capturing transition.* Thus, every transition must be followed by a distinguishing sequence, and the distinguishing sequence should be followed by a capturing transition. The capturing transition is not part of the distinguishing sequence,

since we cannot observe the output after applying the capturing transition. In our example, after the capturing transition the flip-flop under test will be in state 18, which cannot be observed.

The unstable states are categorized into different groups, based on the kind of transitions that the unstable state makes in the flow table. These groups are selected to simplify the analysis for generating patterns for them, as we will show later in this section. For each of the groups, we select a member, and determine the state sub-sequence required to identify it. Other members of the group will require similar sub-sequences. The result of this analysis is a set of sub-sequences of states. A test that makes an MD flip-flop visit all the states of each of these state sub-sequences in the order of the sub-sequence, is a checking experiment for the MD flip-flop. The sub-sequences need not be connected to each other to form one long sequence. As we will show later in this section, for each sub-sequence we can find a corresponding test pattern. The test pattern can be scanned in, and T and CK changed appropriately to make the MD flip-flop visit the sub-sequence. The result can then be scanned out. These patterns are a checking experiment for the MD flip-flop under test because they make the flip-flop visit all the states of each required state sub-sequences, in the order of the sub-sequences. The unstable state groups are:

Group A: Unstable states corresponding to transitions between quadrants

Example: Unstable states corresponding to: 2-18, 3-18, 18-2

Required sub-sequences: 2-18-2-18 for 2-18

3-18-2-18 for 3-18

Since the 18-2 transition ends in distinguishing state 2, and state 2 is followed by a capturing transition, the unstable state corresponding to 18-2 is identified. Thus it can be identified by either 2-18-2-18 or 3-18-2-18. The 2-18 and 3-18 transitions are followed by distinguishing sequence 18-2 and that is followed by the capturing transition 2-18, thus the unstable states corresponding to 2-18 and 3-18 transitions are identified. Since we identified 2-18 and 3-18, we know from the observed outputs, that applying CTSD = 0000, 1000 will always put the flip-flop in state 18. This information is used to identify the predecessor states of many transitions in the other groups.

Group B: Unstable states corresponding to first quadrant transitions on T

Example: Unstable states corresponding to: 12-4

Required sub-sequence: 28-12-4-21

From identifying the unstable states in Group A, we know that applying CTSD = 0101, 1101, 0101 would put the flip-flop in state 12. State 4 is identified

because it is followed by a capturing transition. Since we know the starting state is 12, and the final state is 4, we have identified the unstable state corresponding to the 12-4 transition.

Group C: Unstable states corresponding to fourth quadrant transitions on T (except those in E1 and E2)

Example: Unstable states corresponding to: 18-26
Required sub-sequence: 18-26-10-26

From identifying the unstable states in Group A, we know that applying CTSD = 0000, 1000 would put the flip-flop in state 18. State 26 cannot be followed by a capturing transition. Therefore it must be followed by distinguishing state 10. This needs to be followed by the capturing transition 10-26. Since we know the starting state is 18, and the final state is 26, we have identified the 18-26 transition.

Group D1: Unstable states corresponding to fourth quadrant transition on D

Example: Unstable states corresponding to: 21-19
Required sub-sequence: 21-19-3-18

From identifying the unstable states in group A, we know that applying CTSD = 0001, 1001 would put the flip-flop in state 21. The unstable state corresponding to the 19-3 transition is identified in group A, so we know we were in state 19 before state 3. Since we know the starting state is 21 and the final state is 19, we have identified the 21-19 transition.

Group D2: Unstable states corresponding to fourth quadrant transition on S

Example: Unstable states corresponding to: 21-23
Required sub-sequence: 21-23-7-23

From identifying the unstable states in group A, we know that applying CTSD = 0001, 1001 would put the flip-flop in state 21. The unstable state corresponding to the 23-7 transition is identified in group A, so we know we were in state 23 before state 7. Since we know the starting state is 21 and the final state is 23, we have identified the 21-23 transition.

Group E1: Some states in the fourth quadrant cannot be directly reached from the first quadrant. They can only be reached by changing the D input while in another stable state in the fourth quadrant. In this case we need an extra transition while in the fourth quadrant to apply the T transitions.

Example: Unstable states corresponding to: 19-27

Required sub-sequence: 21-19-27-11-26

The unstable state corresponding to the 21-19 falls into group D1, so we know we are in state 19 after applying CTSD = 0001, 1001, 1000. The unstable state corresponding to the 27-11 transition falls into group A, so we know we were in state 27 before state 11. We identify state 6 using the capturing transition 6-23. Since we know the starting state is 19 and the final state is 27, we have identified the 19-27 transition.

Group E2: Some states in the fourth quadrant cannot be directly reached from the first quadrant. They can only be reached by changing the S input while in another stable state in the fourth quadrant. In this case we need an extra transition while in the fourth quadrant to apply the T transitions.

Example: Unstable states corresponding to: 30-22

Required sub-sequence: 28-30-22-6-23

The unstable state corresponding to the 28-30 falls into group D1, so we know we are in state 30 after applying CTSD = 0101, 1101, 1111. The unstable state corresponding to the 22-6 transition falls into group A, so we know we were in state 22 before state 6. We identify state 6 using the capturing transition 6-23. Since we know the starting state is 30 and the final state is 22, we have identified the 30-22 transition.

Group F: Unstable states corresponding to first quadrant transitions on D

Example: Unstable states corresponding to: 2-4

Required sub-sequence: 18-2-4-21

The unstable state corresponding to the 18-2 transition falls into group A (and we thus have a sub-sequence to identify the unstable state), so we know that we are in state 2 after applying CTSD = 0000, 1000, 0000. We identify state 4 by the capturing transition 4-21. Since we know the starting state is 2, and the final state is 4, we have identified the 2-4 transition.

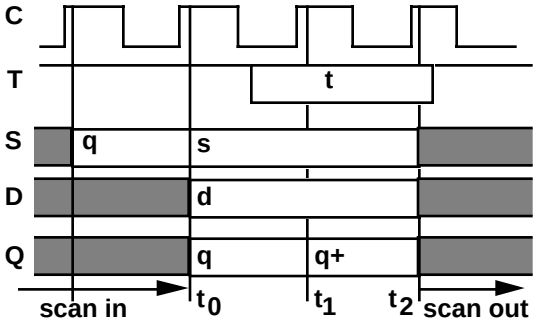
Table 3.3.1-3 shows all the required state sub-sequences. Each sub-sequence in Table 3.3.1-3 requires the initialization of the flip-flop under test to the first state of the sub-sequence, followed by a sequence of inputs that will visit the rest of the states of the sub-sequence. The flip-flop can be initialized by scanning in appropriate values for the first state in the sub-sequence, and the input sequence can be applied by changing the values on CK and T. The initialization and the input sequence can be best described by waveforms. The waveforms for all the groups are shown in Fig. 3.3.1-5. All waveforms start with T =

1 and end with T = 1 to indicate the end of shifting in initial values and beginning of shifting out the flip-flop contents. In these waveforms, values of T, S, D and Q are symbolically represented by t, s, d, q respectively. Each sub-sequence in a group has a waveform with corresponding values of t, s, d, and q. From Table 3.3.1-3, there are 16 sub-sequences in group A and there are 16 combinations of values of t, s, d, and q. Each sub-sequence in group A corresponds to one of the 16 combinations of t, s, d, and q. Groups B and F also have 16 sub-sequences each. Just as in group A, each sub-sequence corresponds to one of the 16 combinations of t, s, d and q. Groups C, D1 and D2 have only 8 sub-sequences each, and each sub-sequence in these groups corresponds to one of 8 combinations of t, s and d. Groups E1 and E2 have only 4 sub-sequences each, and each sub-sequence in these groups corresponds to one of 4 combinations of s and d.

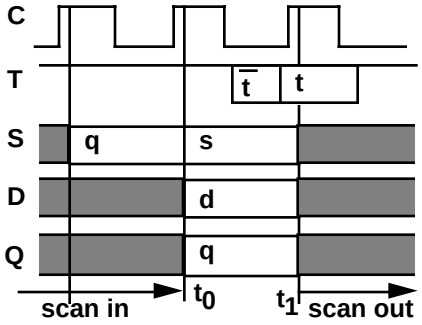
The value on Q, the output of the flip-flop under test, depends on the values at the inputs of the flip-flop under test. For groups A, B, and F, we required Q to have an initial value of q. Since initial values are set up by scanning in values, then the last value scanned in to the flip-flop under test should be q. Therefore, in the waveforms for groups A, B, and F, we show that S has q in the last cycle of scan in. The value of Q on the second cycle (for waveforms that have two cycles) will depend on the initial values of t, s, and d. In the waveforms, *value of Q in the second cycle* is called q+, and can be expressed as a function of t, s and d ($q+ = ts + t\overline{d}$). In waveforms E1 and E2, T has a fixed value, therefore the value of Q in the second cycle is d for E1 and s for E2.

Table 3.3.1-3 Sub-Sequences Required for MD Flip-Flop Checking Experiment.

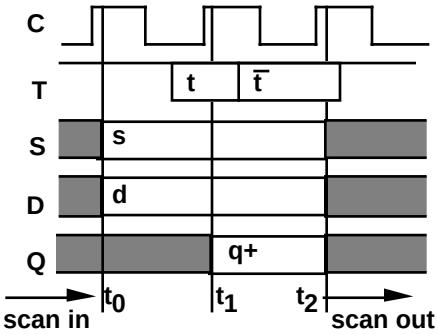
Group A		Group B		Group F	
A1	2-18-2-18	B1	18-2-10-26	F1	18-2-4-21
A2	3-18-2-18	B2	19-3-11-26	F2	19-3-5-21
A3	4-21-5-21	B3	20-4-12-28	F3	20-4-2-18
A4	5-21-5-21	B4	21-5-13-28	F4	21-5-3-18
A5	6-23-7-23	B5	22-6-14-31	F5	22-6-8-24
A6	7-23-7-23	B6	23-7-15-31	F6	23-7-9-24
A7	8-24-8-24	B7	24-8-16-33	F7	24-8-6-23
A8	9-24-8-24	B8	25-9-17-33	F8	25-9-7-23
A9	10-26-10-26	B9	26-10-2-18	F9	26-10-12-28
A10	11-26-10-26	B10	27-11-3-18	F10	27-11-13-28
A11	12-28-12-28	B11	28-12-4-21	F11	28-12-10-26
A12	13-28-12-28	B12	29-13-5-21	F12	29-13-11-26
A13	14-31-15-31	B13	30-14-6-23	F13	30-14-16-33
A14	15-31-15-31	B14	31-15-7-23	F14	31-15-17-33
A15	16-33-17-33	B15	32-16-8-24	F15	32-16-14-31
A16	17-33-17-33	B16	33-17-9-24	F16	33-17-15-31
Group C		Group D1		Group D2	
C1	18-26-10-26	D1	18-20-4-21	D9	18-24-8-24
C2	21-29-13-28	D2	21-19-3-18	D10	21-23-7-23
C3	23-31-15-31	D3	23-25-9-24	D11	23-21-5-21
C4	24-32-16-33	D4	24-22-6-23	D12	24-18-2-18
C5	26-18-2-18	D5	26-28-12-28	D13	26-32-16-33
C6	28-20-4-21	D6	28-26-10-26	D14	28-30-14-31
C7	31-23-7-23	D7	31-33-17-33	D15	31-29-13-28
C8	33-25-9-24	D8	33-31-15-31	D16	33-27-11-26
Group E1		Group E2			
E1	18-20-28-12-28	E5	26-32-24-8-24		
E2	21-19-27-11-26	E6	28-30-22-6-23		
E3	23-25-33-17-33	E7	31-29-21-5-21		
E4	24-22-30-14-31	E8	33-27-19-3-18		



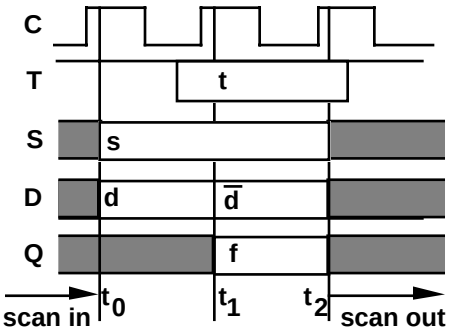
(a) Waveforms for Group A



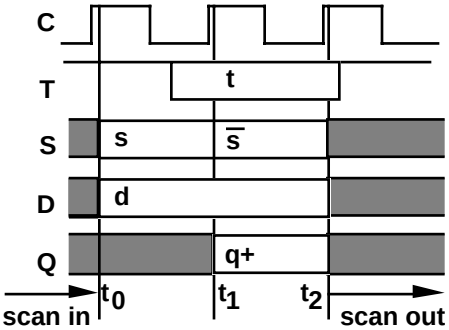
(b) Waveforms for Group B



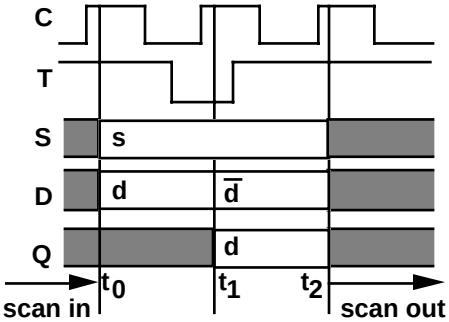
(c) Waveforms for Group C



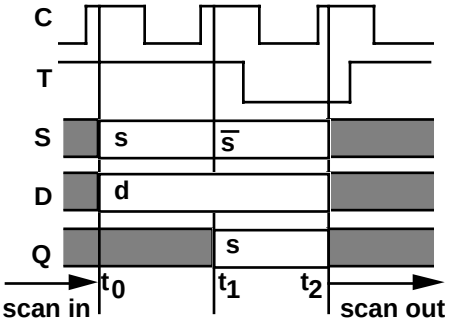
(d) Waveforms for Group D1



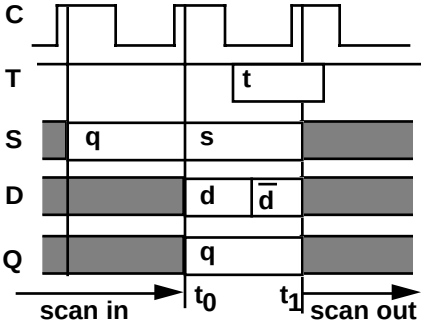
(e) Waveforms for Group D2



(f) Waveforms for Group E1



(g) Waveforms for Group E2



(h) Waveforms for Group F

Figure 3.3.1-5 Waveforms for Unstable State Groups for MD Flip-Flop.
($q+ = ts + \bar{t}d$).

We begin our analysis with group B since it has only one cycle. The waveforms for group B require T be set to t, S be set to s, D to d, and Q to q (where t, s, d, q ∈ {0,1}). Our goal is to find a test pattern that will satisfy these requirements. A test pattern consists of bits, each bit corresponding to a value of an MD flip-flop or a primary input. Fig. 3.3.1-6 shows a part of the design to help illustrate the test pattern generation process for the group B waveforms. In this figure, some signals and connections are left out for readability. The shaded flip-flop, F_t, is the flip-flop under test. Flip-flops are numbered based on their order in the scan chain. The flip-flop preceding the flip-flop under test is F_{t-1}. The D input of F_t is the output of combinational logic with *n* inputs. *p* of the inputs come from other flip-flops and the rest come from primary inputs. The inputs of the combinational logic are $X_1, X_2, \ldots, X_n$. If an input X_i is driven by a flip-flop, the flip-flop is called F_{X_i}. We have three requirements: Q should be set to q, S should be set to *s*, and D should be set to d. Since Q is the output of the flip-flop under test, the bit of the test pattern corresponding to F_t should be q. S comes directly from F_{t-1}, so the bit corresponding to F_{t-1} should be s. Since we need to set D to d, the bits corresponding to $F_{X_1}, F_{X_2}, \ldots, F_{X_p}$ and $X_{p+1}, X_{p+2}, \ldots, X_n$ should be selected such that the output of the combinational logic is d.

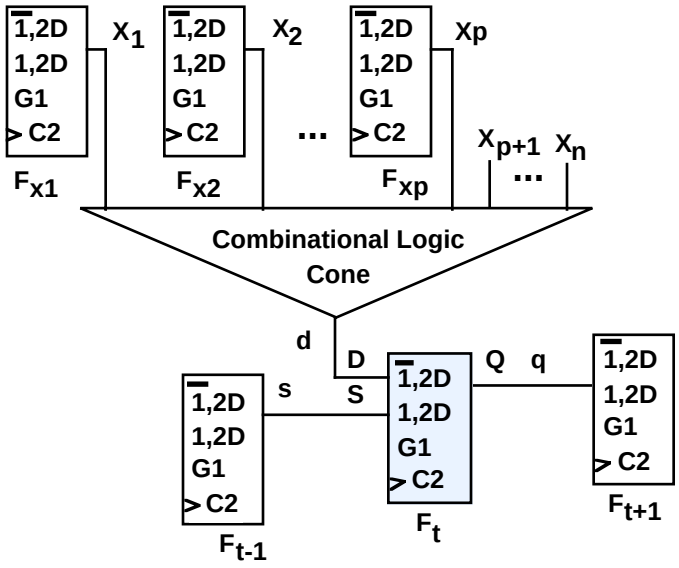


Figure 3.3.1-6 Part of Circuit Involved With Pattern Generation for Group B.

Group F has a single cycle as in the case of group B. However, group F requires a change on the D input of the flip-flop under test. If all the inputs of the combinational logic driving the D input of the flip-flop under test are primary inputs, then we can satisfy the requirements of group F using two patterns, one that makes the output of the combinational

logic 0, and the other 1. If some of the inputs of the combinational logic are flip-flops, these inputs cannot change in a single cycle, and only the bits corresponding to primary inputs can be used. However, the bits corresponding to flip-flops should be selected to allow changes in the primary inputs to change the D input of the flip-flop under test.

The waveforms for the other groups require D and S to have values for two consecutive cycles. In some cases, such as groups A, C, D2 and E2, the D input has to keep the same value for two cycles, while in other cases, such as groups D1 and E1, the D input should change. Similarly, in groups A, C, D1 and E1, the S input has to keep the same value for two cycles, and in the case of D2 and E2, the value on S needs to change. This implies that we need to perform test generation for two cycles. As mentioned earlier, the initial values on the inputs and output of the flip-flop under test are t, d, s, and q. In the following discussion the *value of D in the second cycle* will be called *d+*, and the *value of S in the second cycle* will be called *s+*. The analysis differs for the cases of T = 0 and for T = 1, but applies to groups A, C, D1, D2, E1 and E2.

The case for T = 1 is shown in Fig. 3.3.1-7. s, d and q will impose the same constraints on the test pattern as described for waveforms of group B. After the one cycle shift, the value in F_{t-2} is transferred to F_{t-1}, which as we mentioned earlier is the S input to F_t. Therefore, the bit corresponding to F_{t-2} should be set to s+, the value of S on the

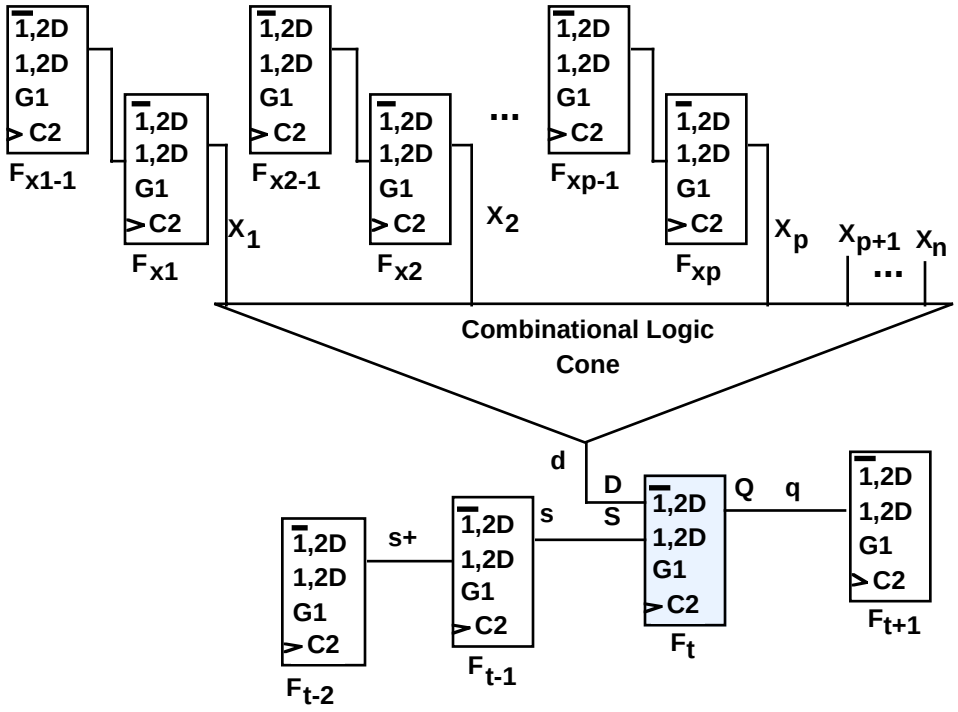


Figure 3.3.1-7 Part of Circuit Involved With Two-Cycle Pattern Generation (T = 1).

second cycle. After the shift, values in $F_{x_1}\text{-}1, F_{x_2}\text{-}1, \ldots, F_{x_p}\text{-}1$ are transferred to $F_{x_1}, F_{x_2}, \ldots, F_{x_p}$ respectively. Therefore, the bits corresponding to $F_{x_1}\text{-}1, F_{x_2}\text{-}1, \ldots, F_{x_p}\text{-}1$ and $X_{p+1}, X_{p+2}, \ldots, X_n$ should be selected such that the output of the combinational logic is d+, the value of D on the second cycle, after the shift. Formally, if the function of the combinational logic cone is $f(F_{x_1}, F_{x_2}, \ldots, F_{x_p}, X_{p+1}, X_{p+2}, \ldots, X_n)$, then we need $f(F_{x_1}, F_{x_2}, \ldots, F_{x_p}, X_{p+1}, X_{p+2}, \ldots, X_n) = d$ to satisfy requirements of the first cycle and $f(F_{x_1}\text{-}1, F_{x_2}\text{-}1, \ldots, F_{x_p}\text{-}1, X_{p+1}, X_{p+2}, \ldots, X_n) = d+$ to satisfy requirements of the second cycle.

In Fig. 3.3.1-7, all the flip-flops had only one role in the circuit. This was done only to keep the explanation simple. It is possible, for example, that the flip-flop preceding the flip-flop under test is also one of the flip-flops that drive the combinational logic (see Fig. 3.3.1-8). In that case, the input driving the combinational logic should be assumed to be s when trying to find a pattern to set D to d. A test pattern that satisfies all these requirements would identify the corresponding unstable state of the flip-flop under test.

The case for T = 0 is shown in Fig. 3.3.1-9. s, d and q will impose the same constraints on the test pattern as described earlier. Since T = 0, s+, the value of S on the second cycle, will be the output of the combinational logic H (combinational logic block driving D input of F_{t-1} in Fig. 3.3.1-8). Therefore, the bits corresponding to the inputs of

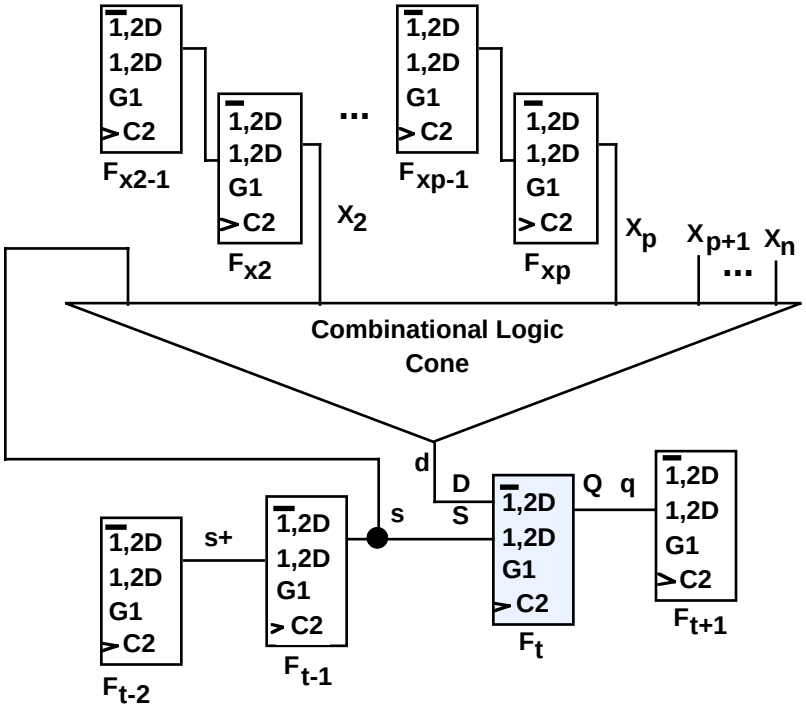


Figure 3.3.1-8 Circuit With F_{t-1} Driving the Combinational Logic of D.

H should be selected such that the output of H will be s+. Similarly, the bits corresponding to the inputs of combinational logic cones $K_1, K_2, \ldots, K_p$ should be selected such that they supply a pattern to combinational logic cone J that would set its output to d+. Formally, suppose that the function of combinational logic cone J is f_J. Then we need $f_J(L_{x_1}\text{-}1, L_{x_2}\text{-}1, \ldots, L_{x_p}\text{-}1, X_{p+1}, X_{p+2}, \ldots, X_n) = d$ to satisfy requirements of the first cycle and $f_J(f_{K_1}, f_{K_2}, \ldots, f_{K_p}, X_{p+1}, X_{p+2}, \ldots, X_n) = d+$, where f_{K_i} is the function of combinational logic cone K_i, to satisfy requirements of the second cycle.

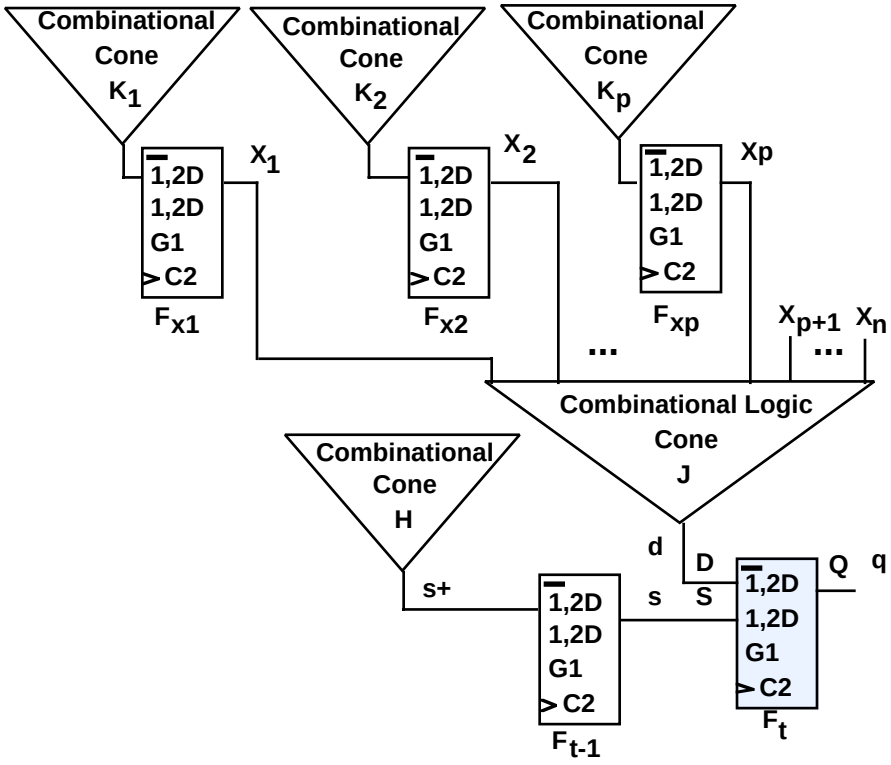


Figure 3.3.1-9 Part of Circuit Involved With Two-Cycle Pattern Generation (T = 0).

In the above analysis, we showed how to find a pattern that would apply a state sub-sequence to an MD flip-flop under test. It is just as important to capture the output of the flip-flop after the sequence is applied. If the sequence has T = 1, the output of a flip-flop is captured by the next flip-flop in scan chain. In such a case, the pattern is scanned in, T remains 1, and the pattern is scanned out. The mechanics of the operation are similar to that of the current test approach for scan chain flip-flops, except that several patterns are used.

For sequences with T = 0, the next flip-flop in the scan chain cannot capture the output of the flip-flop under test. This problem did not appear with MD-latch design in Chapter 2, because we captured the output of the MD-latch using the D-latch after it.

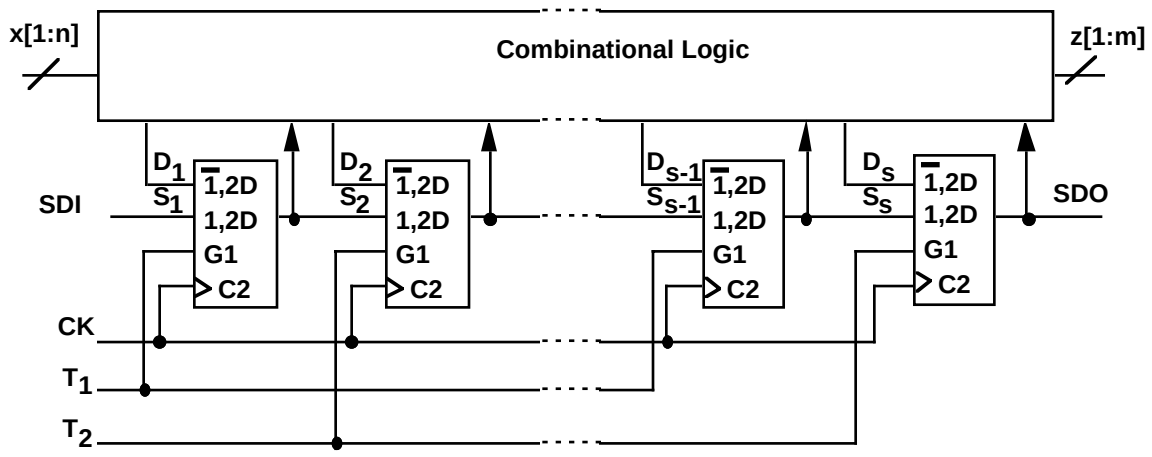


Figure 3.3.1-10 Multiple Test Mode Signals to Capture Flip-Flop Output.

One solution to this problem is to change the design slightly by using multiple test mode signals T1 and T2 instead of using a single T input. As shown in Fig. 3.3.1-10, T1 and T2 alternate as select inputs to the MD flip-flops. Now, suppose the flip-flop we are testing has T1 as input, and the sequence we are targeting requires T1 = 0. During scan-in phase, both T1 and T2 would be set to 1. T1 changes value as required by the waveform, while T2 remains 1. The output of the flip-flop under test is then captured by the next flip-flop since its G input is still set to 1. The problem with this approach is that it requires an extra primary input pin, and an additional wire to be routed throughout the chip.

Another solution is not to make any changes in the design, but sensitize the output of the flip-flop under test to the input of another flip-flop or a primary output. This is shown in Fig. 3.3.1-11. The advantage of this approach is that it requires no modification to the circuit. The disadvantage is that it reduces the number of possible patterns that can be used.

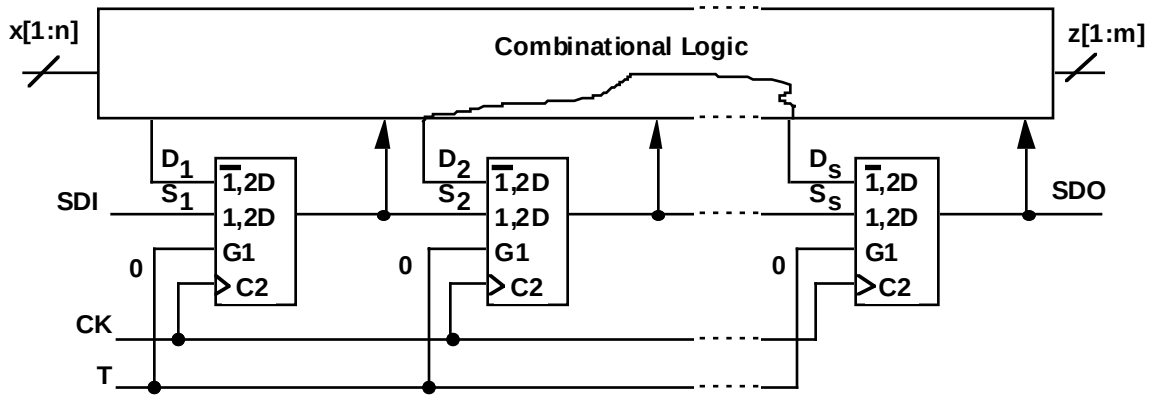


Figure 3.3.1-11 Path Sensitization to Capture Flip-Flop Output.

There may even be cases where it is not possible to apply a desired sequence and capture the output of the flip-flop through the combinational logic. We selected the second solution for implementation in our ATPG tool. This will be discussed in detail in Chapter 4.

3.3.2 TP Flip-Flop Scan Chain

In Section 3.1.3, we described how a checking experiment can be derived for a TP flip-flop, with the assumptions that the inputs can be changed at any time, as long as only one input changes at a time, and that the output is always observable. When the TP flip-flop is used as part of a scan chain (see Fig. 3.3.2-1), the data inputs of the TP flip-flop are not directly controllable from the primary inputs. Formally, the data inputs of a TP flip-flop are D_1 and D_2. Since we use D_1 as the data input from the combinational logic, and D_2 as the data input from the preceding latch in the scan chain, we refer to D_1 as the D input, and D_2 as the S input of the TP flip-flop in the following discussion.

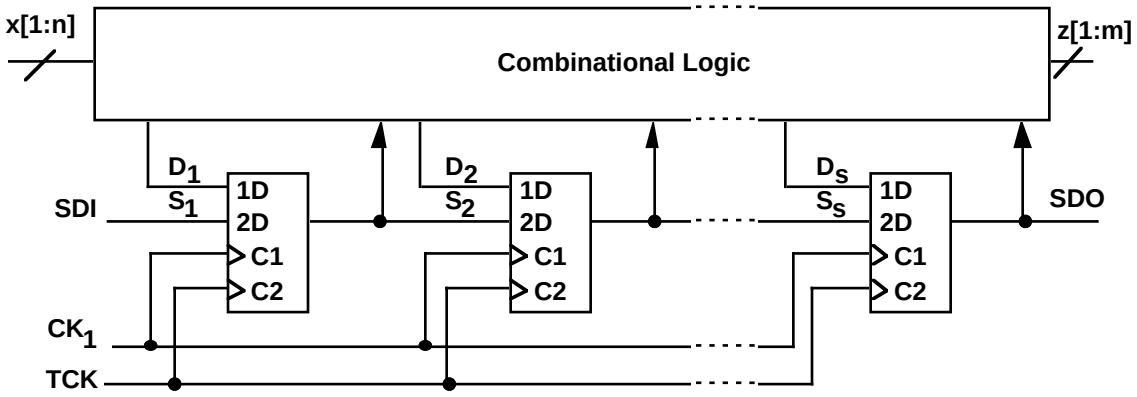


Figure 3.3.2-1 TP Flip-Flop Based Scan Chain.

The S input of the TP flip-flop is the output of another flip-flop, and the output of flip-flops change only when C_1 or C_2 changes to 1. Therefore, the S input can only change on a positive edge of CK_1 (C_1 of the flip-flop) or TCK (C_2 of the flip-flop). Since flip-flops are edge-triggered devices, the output of a flip-flop will change only once every cycle (A cycle is the time between successive positive transitions of C_1 or C_2). Therefore, the S input can change only once every cycle. The D input has the same property as S if the all the inputs of the combinational logic driving it are flip-flop outputs. If some of the inputs are primary inputs, then it may be possible to change the value on D without a positive transition on C_1 or C_2. The C_1 and C_2 inputs can change at any time since they are primary inputs.

The output of the flip-flop under test is captured by the next flip-flop in the scan chain by applying a positive transition on TCK. The captured output corresponds to the state with $C_1 = C_2 = 0$, since the "capturing flip-flop" captures the output of the flip-flop

under test before the transitioning control input has a chance to change the state of the flip-flop under test.

The restrictions on the TP flip-flop in the scan chain can be summarized as follows:

- the S input can only change when C_1 or C_2 = 1.
- the S input can change only once per cycle.
- the output of the flip-flop is "observed" only for states with C_1C_2 = 00.
- the output of the flip-flop is captured on positive transition of C_2.

These restrictions make some of the total states unreachable. Table 3.3.2-1 shows the primitive flow table of the TP flip-flop with these states covered with X. Outputs that cannot be observed are covered with \. The reduced primitive flow table is shown in Table 3.3.2-2. If a scan chain TP flip-flop is replaced by a device that performs the operation described by this flow table, the circuit will still operate correctly. The transition diagrams for the reduced flow table are shown in Fig. 3.3.2-2 and 3.3.2-3.

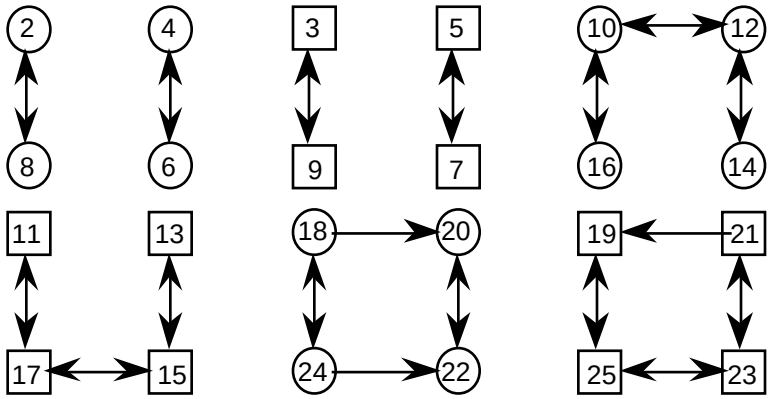


Figure 3.3.2-2 Graphs of Transitions Within Quadrants for Scan Chain TP Flip-Flop.

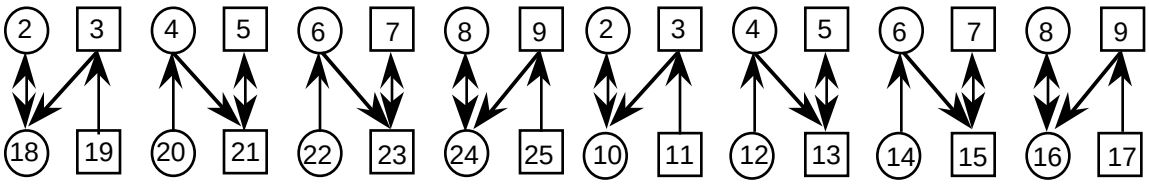


Figure 3.3.2-3 Graphs of Transitions Between Quadrants for Scan Chain TP Flip-Flop.

Table 3.3.2-1 Marked-Up Primitive Flow Table of TP Flip-Flop in Scan Chain.

	DS															
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10
	C_1C_2=00				C_1C_2=10				C_1C_2=01				C_1C_2=11			
2	(2) 0	~~4~~	–	8	10	–	–	–	18	–	–	–	–	–	–	–
3	[3] 1	~~5~~	–	9	10	–	–	–	18	–	–	–	–	–	–	–
4	~~2~~	(4) 0	6	–	–	12	–	–	–	21	–	–	–	–	–	–
5	~~3~~	[5] 1	7	–	–	12	–	–	–	21	–	–	–	–	–	–
6	–	4	(6) 0	~~8~~	–	–	15	–	–	–	23	–	–	–	–	–
7	–	5	[7] 1	~~9~~	–	–	15	–	–	–	23	–	–	–	–	–
8	2	–	~~6~~	(8) 0	–	–	–	17	–	–	–	24	–	–	–	–
9	3	–	~~7~~	[9] 1	–	–	–	17	–	–	–	24	–	–	–	–
10	2	–	–	–	(10) 0	12	–	16	–	–	–	–	–	–	–	–
11	3	–	–	–	[11] 1	~~13~~	–	17	–	–	–	–	–	–	–	–
12	–	4	–	–	10	(12) 0	14	–	–	–	–	–	–	–	–	–
13	–	5	–	–	~~11~~	[13] 1	15	–	–	–	–	–	–	–	–	–
14	–	–	6	–	–	12	(14) 0	~~16~~	–	–	–	–	–	–	–	–
15	–	–	7	–	–	13	[15] 1	17	–	–	–	–	–	–	–	–
16	–	–	–	8	10	–	~~14~~	(16) 0	–	–	–	–	–	–	–	–
17	–	–	–	9	11	–	15	[17] 1	–	–	–	–	–	–	–	–
18	2	–	–	–	–	–	–	–	(18) 0	20	–	24	–	–	–	–
19	3	–	–	–	–	–	–	–	[19] 1	~~21~~	–	25	–	–	–	–
20	–	4	–	–	–	–	–	–	~~18~~	(20) 0	22	–	–	–	–	–
21	–	5	–	–	–	–	–	–	19	[21] 1	23	–	–	–	–	–
22	–	–	6	–	–	–	–	–	–	20	(22) 0	~~24~~	–	–	–	–
23	–	–	7	–	–	–	–	–	–	21	[23] 1	25	–	–	–	–
24	–	–	–	8	–	–	–	–	18	–	22	(24) 0	–	–	–	–
25	–	–	–	9	–	–	–	–	19	–	~~23~~	[25] 1	–	–	–	–

Table 3.3.2-2 Reduced Primitive Flow Table of TP Flip-Flop in Scan Chain.

	DS 00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10
	C_1C_2=00				C_1C_2=10				C_1C_2=01				C_1C_2=11			
2	(2) 0	–	–	8	10	–	–	–	18	–	–	–	–	–	–	–
3	[3] 1	–	–	9	10	–	–	–	18	–	–	–	–	–	–	–
4	–	(4) 0	6	–	–	12	–	–	–	21	–	–	–	–	–	–
5	–	[5] 1	7	–	–	12	–	–	–	21	–	–	–	–	–	–
6	–	4	(6) 0	–	–	–	15	–	–	–	23	–	–	–	–	–
7	–	5	[7] 1	–	–	–	15	–	–	–	23	–	–	–	–	–
8	2	–	–	(8) 0	–	–	–	17	–	–	–	24	–	–	–	–
9	3	–	–	[9] 1	–	–	–	17	–	–	–	24	–	–	–	–
10	2	–	–	–	(10)	12	–	16	–	–	–	–	–	–	–	–
11	3	–	–	–	[11]	–	–	17	–	–	–	–	–	–	–	–
12	–	4	–	–	10	(12)	14	–	–	–	–	–	–	–	–	–
13	–	5	–	–	–	[13]	15	–	–	–	–	–	–	–	–	–
14	–	–	6	–	–	12	(14)	–	–	–	–	–	–	–	–	–
15	–	–	7	–	–	13	[15]	17	–	–	–	–	–	–	–	–
16	–	–	–	8	10	–	–	(16)	–	–	–	–	–	–	–	–
17	–	–	–	9	11	–	15	[17]	–	–	–	–	–	–	–	–
18	2	–	–	–	–	–	–	–	(18)	20	–	24	–	–	–	–
19	3	–	–	–	–	–	–	–	[19]	–	–	25	–	–	–	–
20	–	4	–	–	–	–	–	–	–	(20)	22	–	–	–	–	–
21	–	5	–	–	–	–	–	–	19	[21]	23	–	–	–	–	–
22	–	–	6	–	–	–	–	–	–	20	(22)	–	–	–	–	–
23	–	–	7	–	–	–	–	–	–	21	[23]	25	–	–	–	–
24	–	–	–	8	–	–	–	–	18	–	22	(24)	–	–	–	–
25	–	–	–	9	–	–	–	–	19	–	–	[25]	–	–	–	–

As with the MD flip-flop in Section 3.3.1, we can derive a checking experiment from the new reduced flow table. Again, we have columns with outputs that cannot be observed (C_1C_2 = 01 and C_1C_2 = 10). For states in these columns the distinguishing sequence is an input that takes the flip-flop to a state with C_1C_2 = 00 (i.e., a negative transition of C_1 or C_2). For the MD flip-flop we used a capturing transition after the distinguishing sequence to capture the output of the flip-flop under test. The capturing transition was a positive transition of C. In the case of the TP flip-flop, we will use a positive transition of C_2. This allows us to capture the output of the TP flip-flop in the next flip-flop of the scan chain.

For the MD flip-flop we categorized the unstable states into several groups. Unstable states in the same group had similar tests. We repeat the same procedure here, but with fewer groups:

Group A1 : Unstable states corresponding to transitions between C_1C_2 = 00 and C_1C_2 = 10 columns.

Example: Unstable states corresponding to: 2-10, 3-10, 10-2

Required sub-sequences: 2-10-2-18 for 2-10

3-10-2-18 for 3-10

Since the 10-2 transition ends in distinguishing state 2, and state 2 is followed by a capturing transition 2-18, the unstable state corresponding to 10-2 is identified. The 2-10 and 3-10 transitions are followed by distinguishing sequence 10-2 and that is followed by the capturing transition 2-18, thus the unstable states corresponding to 2-10 and 3-10 transitions are identified. Since we identified 2-10 and 3-10, we know from the observed outputs, that applying C_1C_2DS = 0000, 1000 will always put the flip-flop in state 10. This information is used to identify the predecessor states of many transitions in the other groups.

Group A2: Unstable states corresponding to transitions between C_1C_2 = 00 and C_1C_2 = 01 columns.

Example: Unstable states corresponding to: 2-18, 3-18, 18-2

Required sub-sequences: 2-18-2-18 3-18-2-18

Since the 18-2 transition ends in distinguishing state 2, and state 2 is followed by a capturing transition, the unstable state corresponding to 18-2 is identified. The 2-18 and 3-18 transitions are followed by distinguishing sequence 18-2 and that is followed by the capturing transition 2-18, thus the unstable states corresponding to 2-18 and 3-18 transitions are identified. Since we identified

2-18 and 3-18, we know from the observed outputs, that applying C_1C_2DS = 0000, 0100 will always put the flip-flop in state 18. This information is used to identify the predecessor states of many transitions in the other groups.

Group B1: Unstable states in C_1C_2 = 10 column corresponding to D transitions.

Example: Unstable states corresponding to: 12-14
Required sub-sequence: 12-14-6-23

From identifying the unstable states in group A1, we know that applying C_1C_2DS = 0001, 1001 would put the flip-flop in state 12. The unstable state corresponding to the 14-6 transition is identified in group A1, so we know we were in state 14 before state 6. Since we know the starting state is 12 and the final state is 14, we have identified the 12-14 transition.

Group B2: Unstable states in C_1C_2 = 01 column corresponding to D transitions.

Example: Unstable states corresponding to: 21-23
Required sub-sequence: 21-23-7-23

From identifying the unstable states in group A2, we know that applying C_1C_2DS = 0001, 0101 would put the flip-flop in state 21. The unstable state corresponding to the 23-7 transition is identified in group A2, so we know we were in state 23 before state 7. Since we know the starting state is 21 and the final state is 23, we have identified the 21-23 transition.

Group C1: Unstable states in C_1C_2 = 10 column corresponding to S transitions.

Example: Unstable states corresponding to: 12-10
Required sub-sequence: 12-10-2-18

From identifying the unstable states in group A1, we know that applying C_1C_2DS= 0001, 1001 would put the flip-flop in state 12. The unstable state corresponding to the 10-2 transition is identified in group A1, so we know we were in state 10 before state 2. Since we know the starting state is 12 and the final state is 10, we have identified the 12-10 transition.

Group C2: Unstable states in C_1C_2 = 01 column corresponding to S transitions.

Example: Unstable states corresponding to: 21-19
Required sub-sequence: 21-19-3-18

From identifying the unstable states in group A2, we know that applying C_1C_2DS = 0001, 0101 would put the flip-flop in state 21. The unstable state corresponding to the 19-3 transition is identified in group A2, so we know we

were in state 19 before state 3. Since we know the starting state is 21 and the final state is 19, we have identified the 21-19 transition.

Group D: Unstable states in C_1C_2 = 00 column corresponding to D transitions.

Example: Unstable states corresponding to: 2-8
Required sub-sequence: 18-2-8-24

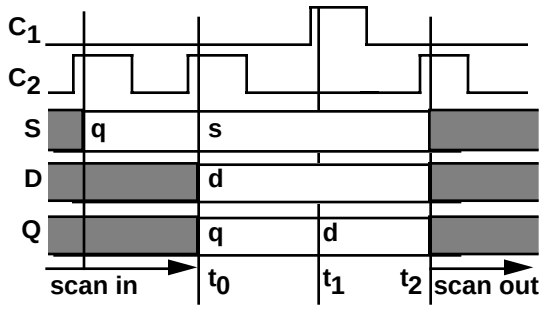
The unstable state corresponding to the 18-2 transition falls into group A2 (and we thus have a sub-sequence to identify the unstable state), so we know that we are in state 2 after applying C_1C_2DS= 0000, 1000, 0000. We identify state 8 by the capturing transition 8-24. Since we know the starting state is 2, and the final state is 8, we have identified the 2-8 transition.

Table 3.3.2-3 shows all the required state sub-sequences. Each sub-sequence in Table 3.3.2-3 requires the initialization of the flip-flop under test to the first state of the sub-sequence, followed by a sequence of inputs that will visit the rest of the states of the sub-sequence. The flip-flop can be initialized by scanning in appropriate values for the first state in the sub-sequence, and the input sequence can be applied by changing the values on TCK and C_1. The initialization and the input sequence can be best described by waveforms. The waveforms for all the groups are shown in Fig. 3.3.2-4. All waveforms start with at least one pulse of C_2 to indicate the end of shifting in initial values and beginning of shifting out the flip-flop output. In these waveforms, S, D and Q are symbolically represented by the boolean values s, d, q respectively. A waveform with a combination of s, d, q (or just s, d where applicable), identifies one of the unstable states in the group. From Table 3.3.2-3, there are 8 sub-sequences in group A1 and there are 8 combinations of values of s, d, and q. Each sub-sequence in group A1 corresponds to one of the 8 combinations of s, d, and q. Groups A2 and D also have 8 sub-sequences each. Just as in group A1, each sub-sequence corresponds to one of the 8 combinations of s, d and q. Groups B1, B2, C1 and C2 have only 4 sub-sequences each, and each sub-sequence in these groups corresponds to one of 4 combinations of s and d.

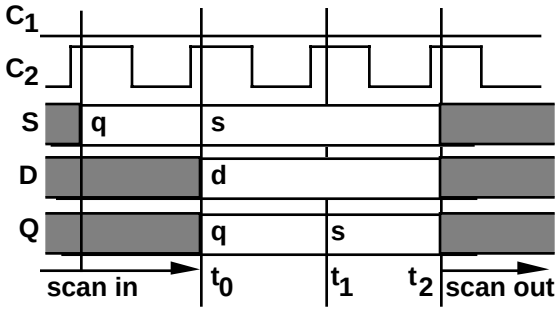
The value on Q, the output of the flip-flop under test, depends on the values at the inputs of the flip-flop under test. For groups A1, A2 and D, we required Q to have an initial value of q. Since initial values are set up by scanning in values, then the last value scanned in to the flip-flop under test should be q. Therefore, in the waveforms for groups A1, A2 and D, we show that S has q in the last cycle of scan in.

Table 3.3.2-3 Sub-Sequences Required for TP Flip-Flop Checking Experiment.

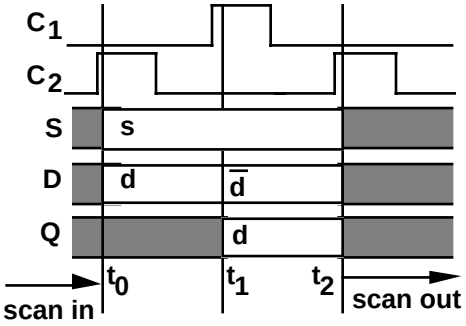
Group A1		Group A2		Group D	
A1	2-10-2-18	A9	2-18-2-18	D1	2-8-24
A2	3-10-2-18	A10	3-18-2-18	D2	3-9-24
A3	4-12-5-21	A11	4-21-5-21	D3	4-6-23
A4	5-12-5-21	A12	5-21-5-21	D4	5-7-23
A5	6-15-7-23	A13	6-23-7-23	D5	6-4-21
A6	7-15-7-23	A14	7-23-7-23	D6	7-5-21
A7	8-17-8-24	A15	8-24-8-24	D7	8-2-18
A8	9-17-8-24	A16	9-24-8-24	D8	9-3-18
Group B1		Group C1			
B1	10-16-8-24	C1	18-24-8-24		
B2	13-15-7-23	C2	21-23-7-23		
B3	15-13-5-21	C3	23-21-5-21		
B4	16-10-2-18	C4	24-18-2-18		
	Group B2		Group C2		
B5	10-12-4-21	C5	18-20-4-21		
B6	13-11-3-18	C6	21-19-3-18		
B7	15-17-9-24	C7	23-25-9-24		
B8	16-14-6-23	C8	24-22-6-23		



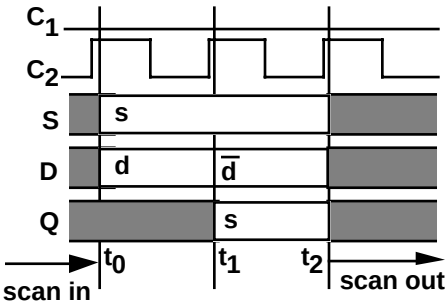
(a) Waveforms for Group A1



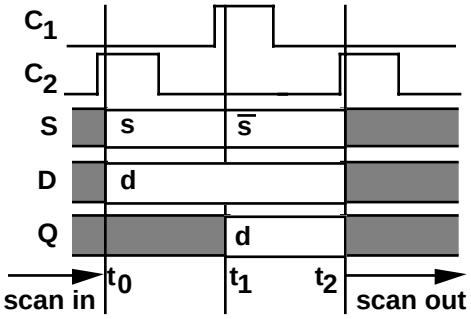
(b) Waveforms for Group A2



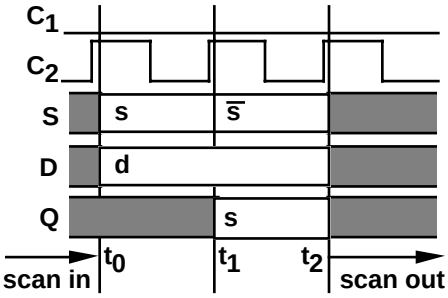
(c) Waveforms for Group B1



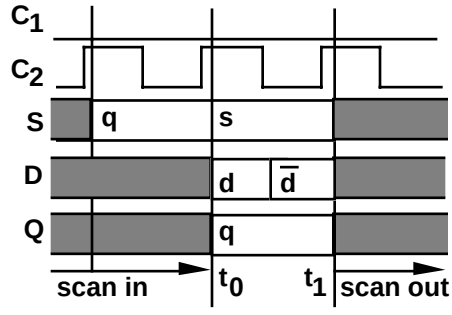
(d) Waveforms for Group B2



(e) Waveforms for Group C1



(f) Waveforms for Group C2



(g) Waveforms for Group D

Figure 3.3.2-4 Waveforms for Unstable State Groups for TP Flip-Flop.

The waveforms in Fig. 3.3.2-4 are similar to those that we had for the MD flip-flop. The approach for finding a pattern here is very similar to that presented in Section 3.3.1. The waveforms have values for two consecutive cycles. This implies that we need to perform test generation for two cycles. The analysis depends on whether the waveform has a C_1 pulse.

We begin our analysis with groups A2, B2 and C2. All these groups require two states for their test, and their waveforms do not have a C_1 pulse. Fig. 3.3.2-5 shows the part of the circuit involved in generating a test for a waveform with no pulse on C_1. In this figure, some signals and connections are left out for readability. The shaded flip-flop, F_t, is the flip-flop under test. Flip-flops are numbered based on their order in the scan chain. The flip-flop preceding the flip-flop under test is F_{t-1}. The D input of the flip-flop is the output of combinational logic with *n* inputs. *p* of the inputs come from other flip-flops and the rest come from primary inputs. The inputs of the combinational logic are $X_1, X_2, \ldots, X_n$. If an input X_i is driven by a flip-flop, the flip-flop is called F_{X_i}.

Since S is the output of F_{t-1}, the bit corresponding to F_{t-1} should be set to s. D is the output of the combinational logic. Therefore, the bits corresponding to $F_{X1}, F_{X2}, \ldots, F_{Xp}$ and $X_{p+1}, X_{p+2}, \ldots, X_n$ should be selected such that the output of the combinational logic is d. After the one cycle shift, the value in F_{t-2} will be transferred to

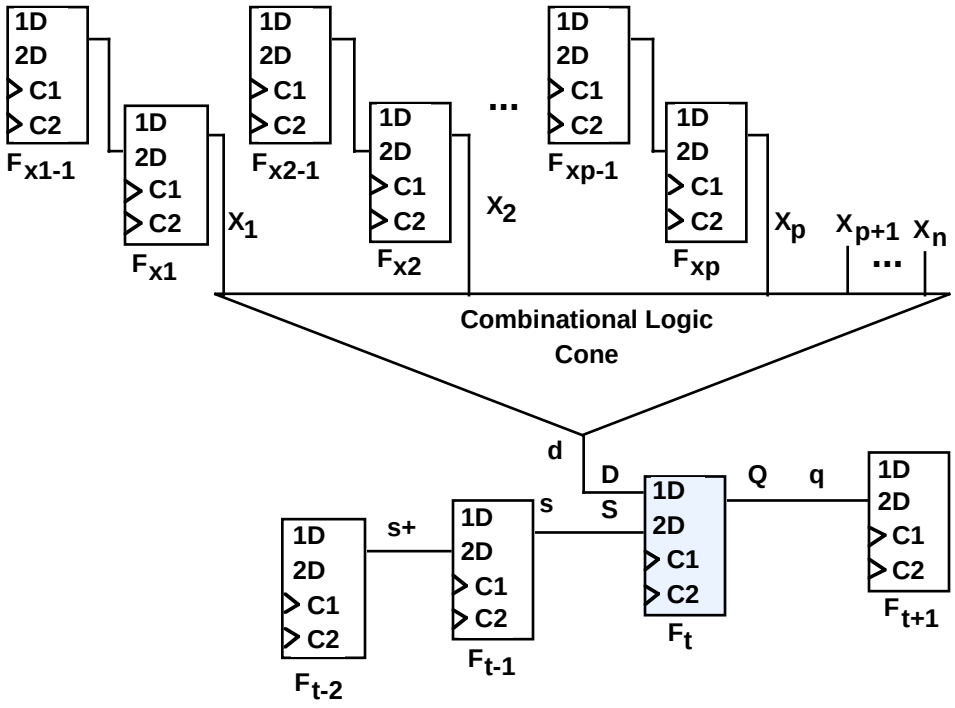


Figure 3.3.2-5 Part of Circuit Involved With Pattern Generation for Waveform With No C_1 Pulse.

F_{t-1}, which as we mentioned earlier is the S input to F_t. Therefore, the bit corresponding to F_{t-2} should also be set to s+, the second value of S. After the once cycle shift, values in $F_{x_1}-1, F_{x_2}-1, \ldots, F_{x_p}-1$ are transferred to $F_{x_1}, F_{x_2}, \ldots, F_{x_p}$. Therefore, the bits corresponding to $F_{x_1}-1, F_{x_2}-1, \ldots, F_{x_p}-1$ and $X_{p+1}, X_{p+2}, \ldots, X_n$ should be selected such that the output of the combinational logic is d+ after the shift. Formally, if the function of the combinational logic cone is $f(F_{x_1}, F_{x_2}, \ldots, F_{x_p}, X_{p+1}, X_{p+2}, \ldots, X_n)$, then we need $f(F_{x_1}, F_{x_2}, \ldots, F_{x_p}, X_{p+1}, X_{p+2}, \ldots, X_n) = d$ to satisfy requirements of the first cycle, and $f(F_{x_1}-1, F_{x_2}-1, \ldots, F_{x_p}-1, X_{p+1}, X_{p+2}, \ldots, X_n) = d+$ to satisfy requirements of the second cycle.

Fig. 3.3.2-6 shows the part of the circuit involved in generating a test for a waveform with a pulse on C_1. s and d will impose the same constraints on the test pattern as described earlier. Since we have a C_1 pulse, s^+, the second value applied to S will be the output of the combinational logic H (combinational logic block driving D input of F_{t-1} in Fig. 3.3.1-8). Therefore, the bits corresponding to the inputs of H should be selected such that the output of H is set to s+. Similarly, the bits corresponding to the inputs of combinational logic cones $K_1, K_2, \ldots, K_p$ should be selected such that they supply a pattern

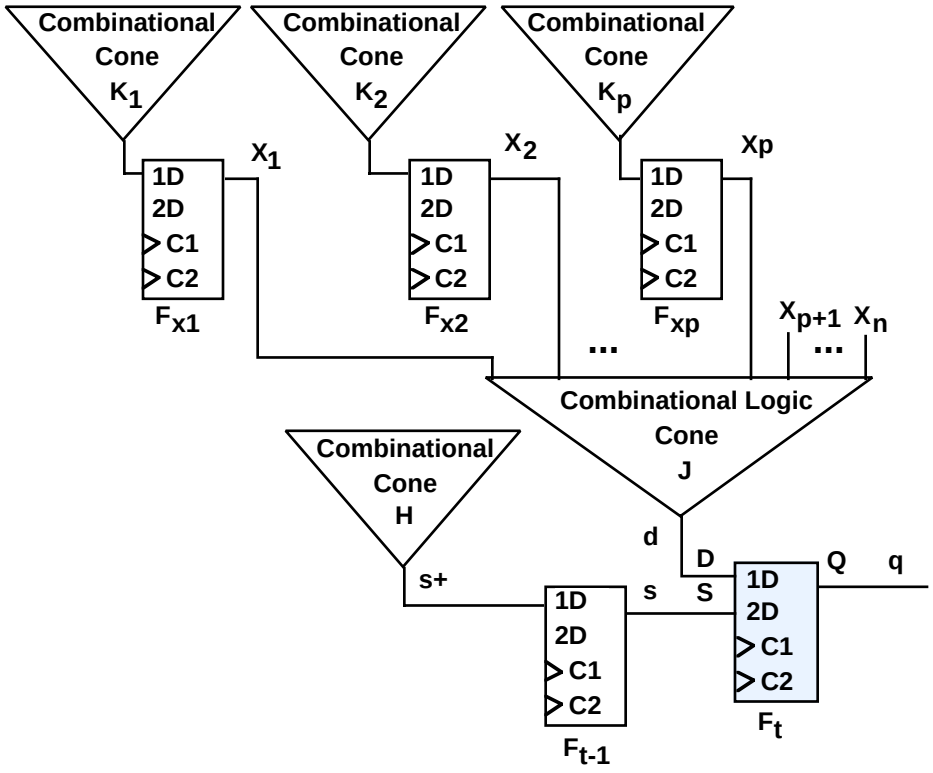


Figure 3.3.2-6 Part of Circuit Involved With Pattern Generation for Waveform With C_1 Pulse.

to combinational logic cone J that would set its output to d+, the second value of D. Formally, suppose that the function of combinational logic cone J is f_J. Then we need $f_J(L_{x_1}\text{-}1, L_{x_2}\text{-}1, \ldots, L_{x_p}\text{-}1, X_{p+1}, X_{p+2}, \ldots, X_n) = d$ to satisfy requirements of the first cycle, and $f_J(f_{K_1}, f_{K_2}, \ldots, f_{K_p}, X_{p+1}, X_{p+2}, \ldots, X_n) = d+$, where f_{K_i} is the function of combinational logic cone K_i, to satisfy requirements of the second cycle.

Group D has a single cycle and requires a change on the D input of the flip-flop just as in the case of group F of the MD flip-flop. If all the inputs of the combinational logic driving the D input of the flip-flop under test are primary inputs, then we can satisfy the requirements of group D using two patterns, one that makes the output of the combinational logic 0, and the other 1. If some of the inputs of the combinational logic are flip-flops, these inputs cannot change in a single cycle, and only the bits corresponding to primary inputs can be used. However, the bits corresponding to flip-flops should be selected to allow changes in the primary inputs to change the D input of the flip-flop under test.

Unlike the MD flip-flop scan chain, the outputs of the TP flip-flops are directly captured by the next flip-flop in the scan chain. This is because our capturing transition of the TP flip-flop is a positive transition on C_2, which always captures the output of the previous flip-flop in the scan chain. In the MD flip-flop, the capturing transition was a positive transition on C. As we saw in Section 3.3.1, if T was 0, then we could not capture directly in the scan chain. Of course, we could not change T to 1, as that would change the total state of the flip-flop under test.

3.4 Summary

This chapter began with the derivation of checking experiments for three common flip-flops, D flip-flop, MD flip-flop and TP flip-flop. When flip-flops are used in a circuit, the circuit imposes constraints on the control of the inputs, and observation of the outputs. In Section 3.2, we showed that a simple test can be used to test all the flip-flops in the shift register. We also showed that no simple test can be derived for scan chain flip-flops, because the D input of a scan flip-flop depends on the combinational logic driving it. In Section 3.3, we showed how a test for these memory elements embedded in the scan chain can be derived. In Chapter 4, we will show how the process of generating patterns for MD flip-flops and TP flip-flops is automated.

Chapter 4. Automatic Test Pattern Generation

In Chapters 2 and 3, we showed how patterns can be generated to apply a checking experiment to a latch or flip-flop embedded in a scan chain. In this Chapter, we describe an algorithm that automates the process of generating patterns for all the bistable elements in a full-scan circuit. These patterns guarantee the application of a checking experiment to each bistable element in the scan chain. We begin by reviewing the similarities and differences between generating patterns for bistable elements of the different scan architectures. This information is used to establish *elementary operations*, which are the building blocks of our algorithm. The algorithm is implemented by extending an existing combinational ATPG program. In doing so, we show the practicality of our algorithm. After briefly describing the combinational ATPG algorithm used, we show how it was extended to implement the elementary operations. The elementary operations are combined to form a procedure that generates patterns for a bistable element in the circuit. The procedure is repeated for each of the bistable elements in the circuit, generating patterns for all the bistable elements of the scan chain. The patterns are compacted to minimize their number.

There are some distinct similarities and differences among the pattern requirements for the four different architectures. All the architectures have patterns that require the analysis of two time frames. For the latch-based architectures, we need to hold values on the D input and S input of the latch under test for two cycles so that we can apply the state triples. For the flip-flop based architectures, we have waveforms that require two cycles. In all scan architectures, two different test generation approaches are required. In one approach, *shift operation*, bistable element values of the second cycle were a shifted version of the values in the first cycle, and in the second approach, *normal operation*, bistable element values of the second cycle are determined by the combinational logic driving the bistable elements. The methods for generating test patterns are similar for all four architectures. The main difference between the tests for different architectures is the timing on the clock and switching inputs, and the number of tests required for the checking experiment. The flip-flop architectures require more tests than the latch based ones. The MD flip-flop based architecture requires the most tests, and is the only one that requires sensitization through the combinational logic.

4.1 Elementary Operations

The operations for generating a test (applying a state triple for latch-based architecture, or a waveform for flip-flop based architecture) for a bistable element can be summarized in the following five elementary operations.

- Single Cycle: Determine bit values of a test pattern that would set lines in the circuit to desired values.
- Shift Operation: Determine bit values of a test pattern that would set lines in the circuit to desired values, and after the scan shifts by one, would again set some lines in the circuit to desired values. The values on the lines need not be the same for both cycles.
- Normal Operation: Determine bit values of a test pattern that would set lines in the circuit to desired values, and after a normal cycle (bistable element input selected from combinational logic), would again set some lines in the circuit to desired values. The values on the lines need not be the same for both cycles.
- Combinational Logic Sensitization: Determine bit values of a test pattern that would sensitize a line in the circuit to a primary output or an input of a bistable element.
- Single Cycle Change: Determine bit values of a test pattern that would set lines in the circuit to desired values, and by changing values only on the primary inputs would change the value of a line in the circuit.

Before explaining how these elementary operations are implemented, we need to explain the combinational ATPG algorithm that we are modifying. The algorithm we selected is used in SIS [Sentovich et. al. 92], and is based on the work of Larrabee [89]. The basic idea of this algorithm is to extract a formula that defines a set of test patterns that detect a fault, and then use boolean satisfiability to satisfy the formula. A formula is *satisfied* if it evaluates to 1.

Consider the circuit in Fig. 4.1-1. Every gate in the circuit corresponds to a formula that represents the function of the gate. The formula contains variables from incoming and outgoing wires and is represented in CNF (conjunctive normal form). Formulas for gates are often expressed as equations. For example, in Fig. 4.1-1 the formula for the AND gate is D = A • B. An equation P = Q is logically equivalent to $PQ + \overline{P}\,\overline{Q}$, because both expressions evaluate to 1 only when P and Q are the same value. $PQ + \overline{P}\,\overline{Q}$ can be transformed to CNF as $(P + \overline{Q})(\overline{P} + Q)$ [McCluskey 86]. In our example, D = A • B, P = D and Q = A • B. Using the above transformations, D = A • B is logically equivalent to $(\overline{D} + AB)(D + \overline{AB})$. This formula can be transformed to the CNF formula $(\overline{D} + A)(\overline{D} + B)(D + \overline{A} + \overline{B})$. The CNF clauses for several equations, and inequalities, that we will be using are shown in Table 4.1-1. A formula for the complete circuit can be formed by taking the conjunction of formulas of each gate in the circuit. In our example,

Table 4.1-1 CNF Clauses for Some Equations.

Equation	CNF Representation
$D = A \bullet B$	$(\overline{D} + A)(\overline{D} + B)(D + \overline{A} + \overline{B})$
$D = A + B$	$(D + \overline{A})(D + \overline{B})(\overline{D} + A + B)$
$A = B$	$(A + \overline{B})(\overline{A} + B)$
$A \neq B$	$(A + B)(\overline{A} + \overline{B})$

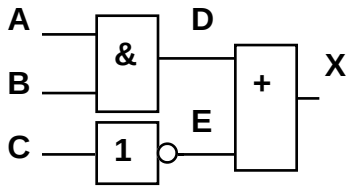


Figure 4.1-1 Circuit to Describe Formula Extraction.

the formula would be

$(X + \overline{D})(X + \overline{E})(\overline{X} + D + E)(\overline{D} + A)(\overline{D} + B)(D + \overline{A} + \overline{B})(\overline{C} + \overline{E})(C + E)$.

Now suppose we want the output of the circuit to be 0. We add the clause $\overline{X}$ to the formula. Thus our formula becomes

$(X + \overline{D})(X + \overline{E})(\overline{X} + D + E)(\overline{D} + A)(\overline{D} + B)(D + \overline{A} + \overline{B})(\overline{C} + \overline{E})(C + E)\overline{X}$

Our goal is to find values of A, B and C that would satisfy the formula (i.e., make it evaluate to 1). Each clause in the formula must be satisfied for the formula to be satisfied. Since $\overline{X}$ is a clause in the formula, then the formula can only be satisfied if $\overline{X}$ = 1 which implies X = 0. With X = 0, the first two clauses can only be satisfied with $\overline{D}$ = 1 and $\overline{E}$ = 1. This implies D = 0 and E = 0. With E = 0, the only way to satisfy the clause $(C + E)$ is with C = 1. All other clauses except $(D + \overline{A} + \overline{B})$ are now satisfied with values selected for X, D and E. Since D = 0, we need to have either $\overline{A}$ = 1 or $\overline{B}$ = 1 to satisfy $(D + \overline{A} + \overline{B})$. This implies A = 0 or B = 0. Therefore for X = 0, we need C = 1 and A or B = 0.

There is more to detecting stuck at faults using boolean satisfiability than has been discussed here. However, the information in the above discussion is enough for implementing our elementary operations. We now show how each of the elementary operations is implemented as an extension of the above discussion.

Single Cycle

The single cycle elementary operation follows directly from the above discussion, except that multiple lines in the circuit need to be set to desired values. For example, consider the circuit in Fig. 4.1-2. In this and all circuits in this section, the clock input to the bistable elements have been omitted. The combinational logic is the same as that of Fig. 4.1-1. Now suppose that we require the input of the bistable element (X) be set to 0 and the output (A) to be set to 1. We add the clause A to our last formula to get

$$(X + \overline{D})(X + \overline{E})(\overline{X} + D + E)(\overline{D} + A)(\overline{D} + B)(D + \overline{A} + \overline{B})(\overline{C} + \overline{E})(C + E)\overline{X}A$$

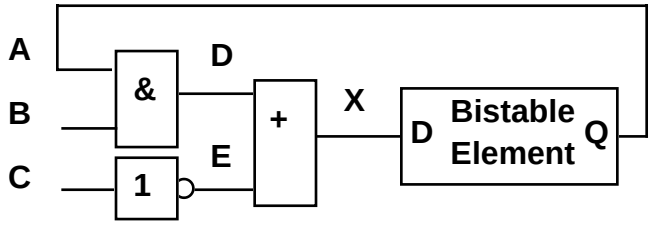


Figure 4.1-2 Circuit to Describe Pattern Generation for Single Cycle.

We satisfy the formula as we did before, except for the clause $(D + \overline{A} + \overline{B})$. Since A = 1 (from the last clause), and D = 0 (D = 0 was determined earlier) the only way to satisfy $(D + \overline{A} + \overline{B})$ is with $\overline{B}$ = 1. This implies B = 0, and our pattern is ABC = 101.

From the above discussion, the process of generating patterns for single cycle operations can be summarized in the following steps:

1. Extract a formula for the combinational logic.
2. Add clauses to set lines in the circuit to desired values.
3. Satisfy the formula of step 2.

Shift and Normal Operation

For shift and normal operation we have two cycles. To handle two cycles, we create two copies of the formula and label variables in one formula with subscript 0 (for time frame 0), and the other with 1 (for time frame 1). For example, consider the circuit in Fig. 4.1-3. The bistable elements shown in this figure have two data sources, S and D. The S input comes from the previous bistable element in the scan chain, and the D input comes from combinational logic. The two formulas for the combinational logic with X as output are

$$(X_0 + \overline{D_0})(X_0 + \overline{E_0})(\overline{X_0} + D_0 + E_0)(\overline{D_0} + A_0)(\overline{D_0} + B_0)(D_0 + \overline{A_0} + \overline{B_0})(\overline{C_0} + \overline{E_0})(C_0 + E_0)$$

and

$$(X_1 + \overline{D_1})(X_1 + \overline{E_1})(\overline{X_1} + D_1 + E_1)(\overline{D_1} + A_1)(\overline{D_1} + B_1)(D_1 + \overline{A_1} + \overline{B_1})(\overline{C_1} + \overline{E_1})(C_1 + E_1)$$

Now suppose we want X = 0, C = 1 and A = 0 in the first cycle, and we want X to remain 0, and C to remain to 1 in the second cycle. Our formulas become

$(X_0 + \overline{D_0})(X_0 + \overline{E_0})(\overline{X_0} + D_0 + E_0)(\overline{D_0} + A_0)(\overline{D_0} + B_0)(D_0 + \overline{A_0} + \overline{B_0})(\overline{C_0} + \overline{E_0})(C_0 + E_0)$

$\overline{X_0}\, C_0 \overline{A_0}$

and

$(X_1 + \overline{D_1})(X_1 + \overline{E_1})(\overline{X_1} + D_1 + E_1)(\overline{D_1} + A_1)(\overline{D_1} + B_1)(D_1 + \overline{A_1} + \overline{B_1})(\overline{C_1} + \overline{E_1})(C_1 + E_1)$

$\overline{X_1}\, C_1$

Extra clauses are added to define relationships between variables in the two time frames. The clauses added depend on the type of operation: shift or normal.

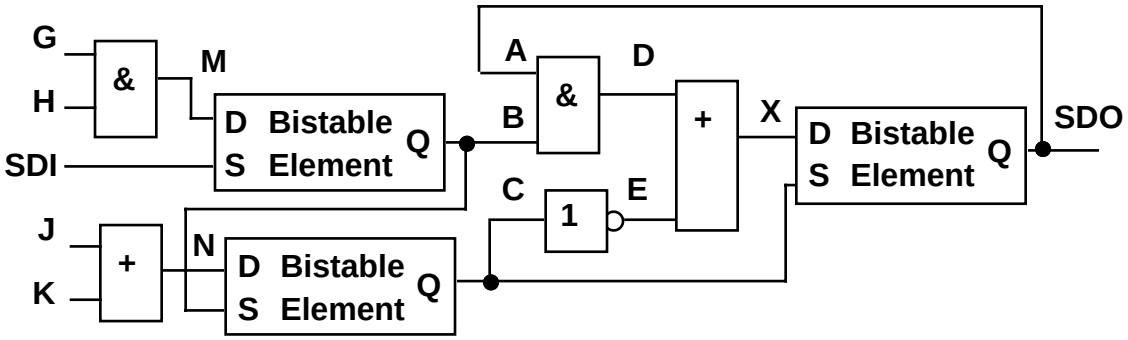


Figure 4.1-3 Circuit to Describe Pattern Generation for Shift and Normal Operation.

Shift Operation

In shift operation, the output of bistable element M_i (note that the subscript of a bistable element refers to the its order in scan chain and not the time frame) at time frame 0 should equal the output of M_{i+1} at time frame 1, because values are shifted in the scan chain between the two time frames. In our example, this means that we would need to add clauses to indicate

$SDI_0 = B_1$, $B_0 = C_1$ and $C_0 = A_1$.

The actual clauses are

$(SDI_0 + \overline{B_1})(\overline{SDI_0} + B_1)(B_0 + \overline{C_1})(\overline{B_0} + C_1)(C_0 + \overline{A_1})(\overline{C_0} + A_1)$

The complete formula to be satisfied is

$(X_0 + \overline{D_0})(X_0 + \overline{E_0})(\overline{X_0} + D_0 + E_0)(\overline{D_0} + A_0)(\overline{D_0} + B_0)(D_0 + \overline{A_0} + \overline{B_0})(\overline{C_0} + \overline{E_0})(C_0 + E_0)$

$\overline{X_0}\, C_0 \overline{A_0}$

$(X_1 + \overline{D_1})(X_1 + \overline{E_1})(\overline{X_1} + D_1 + E_1)(\overline{D_1} + A_1)(\overline{D_1} + B_1)(D_1 + \overline{A_1} + \overline{B_1})(\overline{C_1} + \overline{E_1})(C_1 + E_1)$

$\overline{X_1}\, C_1$

$(SDI_0 + \overline{B_1})(\overline{SDI_0} + B_1)(B_0 + \overline{C_1})(\overline{B_0} + C_1)(C_0 + \overline{A_1})(\overline{C_0} + A_1)$

Satisfying this formula, as we did earlier, will determine a test pattern that will fulfill our requirements. One solution is ABCGHJK(SDI) = 011----0. A"-" is an unspecified input, either 0 or 1 can be used.

From the above discussion, the pattern generation process for shift operations can

be summarized in the following steps:

1. Extract a formula for the combinational logic.
2. Make two copies of the formula, one labeled 0 and one labeled 1.
3. Add clauses between bistable element outputs to imply shift operation. The output of a bistable element in time frame 1 should be the same as the output of its preceding bistable element in time frame 0.
4. Satisfy the formula of step 3.

Normal Operation

For normal operation, the data input of the bistable element at time frame 0 becomes the output at time frame 1. The data input of the bistable element is the output of combinational logic and we need to add the clauses of this combinational logic, in time frame 0, to our formula if they are not already part of it. In our example, we will need

$X_0 = A_1$, $M_0 = B_1$, and $N_0 = C_1$.

For $X_0 = A_1$, we need to add clauses for the combinational logic driving X_0. For $M_0 = B_1$, we will need to add clauses for the AND gate driving M, and for $N_0 = C_1$, we will need to add clauses for the OR gate driving N. The final formula is

$(X_0 + \overline{D_0})(X_0 + \overline{E_0})(\overline{X_0} + D_0 + E_0)(\overline{D_0} + A_0)(\overline{D_0} + B_0)(D_0 + \overline{A_0} + \overline{B_0})(\overline{C_0} + \overline{E_0})(C_0 + E_0)$
$\overline{X_0}\, C_0 \overline{A_0}$
$(X_1 + \overline{D_1})(X_1 + \overline{E_1})(\overline{X_1} + D_1 + E_1)(\overline{D_1} + A_1)(\overline{D_1} + B_1)(D_1 + \overline{A_1} + \overline{B_1})(\overline{C_1} + \overline{E_1})(C_1 + E_1)$
$\overline{X_1}\, C_1$
$(A_1 + \overline{X_0})(\overline{A_1} + X_0)$
$(B_1 + \overline{M_0})(\overline{B_1} + M_0)(\overline{M_0} + G_0)(\overline{M_0} + H_0)(M_0 + \overline{G_0} + \overline{H_0})$
$(C_1 + \overline{N_0})(\overline{C_1} + N_0)(N_0 + \overline{J_0})(N_0 + \overline{K_0})(\overline{N_0} + J_0 + K_0)$

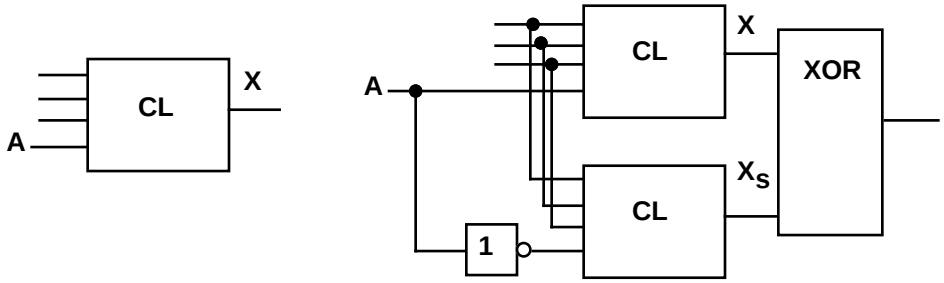
Satisfying this formula, as we did earlier, will determine a test pattern that will fulfill our requirements. One solution is ABCGHJK(SDI) = 0-10-1--.

From the above discussion, the pattern generation process for normal operations can be summarized in the following steps:

1. Extract a formula for the combinational logic.
2. Make two copies of the formula, one labeled 0 and one labeled 1.
3. Add clauses between bistable element inputs and outputs to imply normal operation. The output of a bistable element in time frame 1 should be the same value as its data input in time frame 0.
4. Add clauses in time frame 0 of combinational logic that drives inputs to flip-flops in step 3.
5. Satisfy the formula of step 4.

Path Sensitization

A path from an input A to an output X is *sensitized* if the other inputs of the combinational logic are set such that changing the value on A would change the value on X. From this definition, we can create the circuit in Fig. 4.1-4(b).



(a) Original Circuit (b) Expanded for Sensitization

Fig. 4.1-4 Circuit to Explain Sensitization.

The circuit is created by replicating the original circuit, Fig. 4.1-4(a), connecting all inputs to both circuits, except the one we are trying to sensitize, A. A is tied to the input of one of the circuits, and inverted to the input of the other circuit. The outputs of the two replicated circuits are exclusive-ored. A pattern that satisfies this new circuit (make the output 1), sensitizes a path from A to X. The value of A is not part of the pattern. In fact, changing the value of A would still give a 1 at the output. If there are multiple outputs, then A can be sensitized to any output. A similar approach is taken for using the formulas. The steps are as follows:

1. Extract a formula for the combinational logic.
2. Make two copies of the formula. One of them remains unlabeled, and label the other with s (for sensitize).
3. Add clauses to indicate that A should not be equal to A_S.
4. For each input J except A, include clauses to indicate J should be equal to Js.
5. Add clauses to indicate that X should not be equal to X_S. If there are multiple outputs, then add clauses to indicate that at least one of the outputs differ.
6. Satisfy the formula of step 5.

For example, consider the circuit we had in Fig. 4.1-1. The formula was

$$(X + \overline{D})(X + \overline{E})(\overline{X} + D + E)(\overline{D} + A)(\overline{D} + B)(D + \overline{A} + \overline{B})(\overline{C} + \overline{E})(C + E)$$

Now suppose we want to find a pattern that would sensitize A to the output. From step 2 in our procedure we will have

$$(X + \overline{D})(X + \overline{E})(\overline{X} + D + E)(\overline{D} + A)(\overline{D} + B)(D + \overline{A} + \overline{B})(\overline{C} + \overline{E})(C + E)$$
$$(X_s + \overline{D_s})(X_s + \overline{E_s})(\overline{X_s} + D_s + E_s)(\overline{D_s} + A_s)(\overline{D_s} + B_s)(D_s + \overline{A_s} + \overline{B_s})(\overline{C_s} + \overline{E_s})(C_s + E_s)$$

The clauses to indicate that A is not equal to A_S (step 3) are

$(A + A_s)(\overline{A} + \overline{A_s})$

and the clauses to indicate that B is equal to B_S and C is equal to C_S (step 4) are

$(B + \overline{B_s})(\overline{B} + B_s)(C + \overline{C_s})(\overline{C} + C_s)$

and the clauses to indicate that X is not equal to X_S (step 5) are

$(X + X_s)(\overline{X} + \overline{X_s})$

and the final formula is

$(X + \overline{D})(X + \overline{E})(\overline{X} + D + E)(\overline{D} + A)(\overline{D} + B)(D + \overline{A} + \overline{B})(\overline{C} + \overline{E})(C + E)$

$(X_s + \overline{D_s})(X_s + \overline{E_s})(\overline{X_s} + D_s + E_s)(\overline{D_s} + A_s)(\overline{D_s} + B_s)(D_s + \overline{A_s} + \overline{B_s})(\overline{C_s} + \overline{E_s})(C_s + E_s)$

$(A + A_s)(\overline{A} + \overline{A_s})$

$(B + \overline{B_s})(\overline{B} + B_s)(C + \overline{C_s})(\overline{C} + C_s)$

$(X + X_s)(\overline{X} + \overline{X_s})$

If we analyze this formula as we did before, we can show that this formula can only be satisfied if BC = 11. This means that we can sensitize A through the combinational logic if B is set to 1 and C is set to 1.

As mentioned in the example, the clauses to indicate that X is not be equal to X_S are $(X + X_s)(\overline{X} + \overline{X_s})$. If we had three outputs, X, Y and Z, then we want to add clauses to indicate X is not equal to X_S, or Y is not equal to Y_S, or Z is not equal to Z_S. This done with the following clauses

$(X + X_s + \overline{R_x})(\overline{X} + \overline{X_s} + \overline{R_x})(Y + Y_s + \overline{R_y})(\overline{Y} + \overline{Y_s} + \overline{R_y})(Z + Z_s + \overline{R_z})(\overline{Z} + \overline{Z_s} + \overline{R_z})$

$(R_x + R_y + R_y)$

Suppose X = 0 and X_S = 1. Then X_S = 1will satisfy the first term, and X = 0 will satisfy the second term. R_X can then be set to 1, which will satisfy the last term. R_Y and R_Z can then be set to 0, which will satisfy the rest of the clauses. Now suppose that X = X_S = 0, then the only way the first term can be satisfied is if R_X = 0. If Y = Y_S and Z = Z_S then R_Y and R_Z will also be 0, and the last term will not be satisfied.

Single Cycle Change Operation

In single cycle change operation we want to change the value on line in the circuit without cycling the clock. This means that we must keep the flip-flop values fixed, and change some of the primary inputs. The steps for this operation are:

1. Extract a formula for the combinational logic.
2. Make two copies of the formula. Label one copy with 0 (before transition), and the other with 1 (after transition).
3. For each input of the combinational logic that is the output of a flip-flop, add clauses to make the value in the 0 copy equal to the value in the 1 copy.
4. Add clauses for the desired output values before the transition in the 0 copy, and

after the transition in the 1 copy.

5. Satisfy the formula in step 4.

For example, consider the circuit in Fig. 4.1-5.

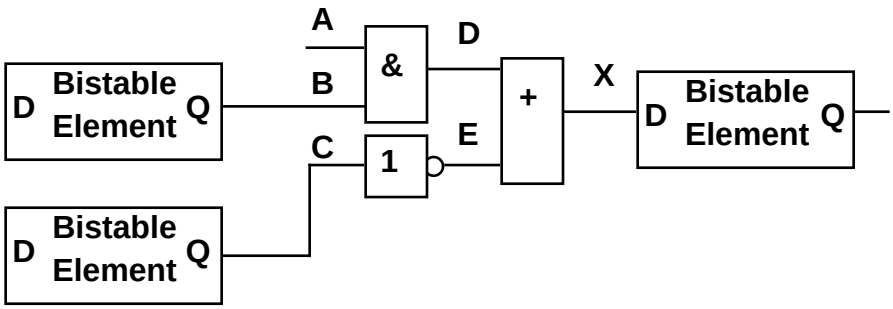


Figure 4.1-5 Circuit to Describe Single Cycle Changing.

The formula for the combinational logic is

$(X + \overline{D})(X + \overline{E})(\overline{X} + D + E)(\overline{D} + A)(\overline{D} + B)(D + \overline{A} + \overline{B})(\overline{C} + \overline{E})(C + E)$

The two copies (step 2 in the procedure) are

$(X_0 + \overline{D_0})(X_0 + \overline{E_0})(\overline{X_0} + D_0 + E_0)(\overline{D_0} + A_0)(\overline{D_0} + B_0)(D_0 + \overline{A_0} + \overline{B_0})(\overline{C_0} + \overline{E_0})(C_0 + E_0)$

and

$(X_1 + \overline{D_1})(X_1 + \overline{E_1})(\overline{X_1} + D_1 + E_1)(\overline{D_1} + A_1)(\overline{D_1} + B_1)(D_1 + \overline{A_1} + \overline{B_1})(\overline{C_1} + \overline{E_1})(C_1 + E_1)$

There are two inputs of the combinational logic that are outputs of flip-flops: B and C. From step 3, we add the following clauses to make $C_0 = C_1$ and $B_0 = B_1$.

$(\overline{B_1} + B_0)(\overline{B_0} + B_1)(\overline{C_1} + C_0)(\overline{C_0} + C_1)$

Now suppose that we want a rising transition on X. This means that we want X_0 to be 0 (i.e., we need the clause $\overline{X_0}$) and X_1 to be 1 (i.e., we need to add the clause X_1). The final formula is

$(X_0 + \overline{D_0})(X_0 + \overline{E_0})(\overline{X_0} + D_0 + E_0)(\overline{D_0} + A_0)(\overline{D_0} + B_0)(D_0 + \overline{A_0} + \overline{B_0})(\overline{C_0} + \overline{E_0})(C_0 + E_0)$

$(X_1 + \overline{D_1})(X_1 + \overline{E_1})(\overline{X_1} + D_1 + E_1)(\overline{D_1} + A_1)(\overline{D_1} + B_1)(D_1 + \overline{A_1} + \overline{B_1})(\overline{C_1} + \overline{E_1})(C_1 + E_1)$

$(\overline{B_1} + B_0)(\overline{B_0} + B_1)(\overline{C_1} + C_0)(\overline{C_0} + C_1)\ \overline{X_0}\ X_1$

This formula can only be satisfied if BC = 11, $A_0 = 0$ and $A_1 = 1$. This means that if a pattern is scanned in with B = 1 and C = 1, then changing A from 0 to 1 will change X from 0 to 1.

Pattern Compaction

The above operations generate patterns for the bistable elements in the design. The generated patterns are placed in pattern tables. Once patterns are generated for all the bistable elements in the design, the number of patterns can be reduced by combining patterns that are compatible. Two patterns are *compatible* if none of their corresponding

bits have conflicts, i.e., one has 0 and the other has 1. For example, the patterns 1-0 and 10- are compatible, and can be combined to 100. The patterns 1-0 and 0-- are not compatible, because the first bit of one pattern is 1 while the other is 0. These two patterns cannot be combined.

The patterns in a pattern table can be reduced by combining the compatible patterns in the pattern table. The resulting patterns may themselves be combined again. The process is repeated until no combining is possible.

4.2 Test Generation Using Elementary Operations

In the previous section, we defined some elementary operations, and showed how they can be implemented by modifying an existing combinational ATPG system. In our actual implementation, rather than extracting the formula and making copies, we make three copies of the netlists, and extract formulas from them as needed. The three netlists are called `net0`, `net1` and `nets`. The functions used to implement these operations and their arguments are:

`pat = SingleCycle(net,m,d,s,q,nets)` implementation of the single cycle operation. It has six inputs and one output.

The inputs are:

- `net`: netlist describing the circuit
- `m`: bistable element under test
- `d`: desired value on D input of bistable element under test
- `s`: desired value on S input of bistable element under test
- `q`: desired value on Q output of bistable element under test
- `nets`: netlist for sensitization (0 if no sensitization required)

The output is:

- `pat`: a test pattern that satisfies input requirements

`pat = ShiftOperation(net0,net1,m,d0,d1,s0,s1,q,nets)` implementation of the shift operation. It has nine inputs and one output.

The inputs are:

- `net0`: netlist at time frame 0
- `net1`: netlist at time frame 1
- `m`: bistable element under test
- `d0`: desired value on D input of bistable element under test at time 0
- `d1`: desired value on D input of bistable element under test at time 1
- `s0`: desired value on S input of bistable element under test at time 0
- `s1`: desired value on S input of bistable element under test at time 1
- `q`: desired value on Q output of bistable element under test at time 0

- nets: netlist for sensitization (0 if no sensitization required)

The output is:

- pat: a test pattern that satisfies input requirements

pat = NormalOperation(net0,net1,m,d0,d1,s0,s1,q,nets)

implementation of the normal operation. It has nine inputs and one output.

The inputs are:

- net0: netlist at time frame 0
- net1: netlist at time frame 1
- m: bistable element under test
- d0: desired value on D input of bistable element under test at time 0
- d1: desired value on D input of bistable element under test at time 1
- s0: desired value on S input of bistable element under test at time 0
- s1: desired value on S input of bistable element under test at time 1
- q: desired value on Q output of bistable element under test at time 0
- nets: netlist for sensitization (0 if no sensitization required)

The output is:

- pat: a test pattern that satisfies input requirements

CLSensitization(net,nets,n,formula) adds clauses to formula to sensitize line n to a primary output or the data input of a bistable element in the circuit.

The inputs are:

- net: netlist describing the circuit
- nets: netlist for sensitization
- n: line to be sensitized
- formula: formula to which new clauses are added. The formula is created by the calling function, and may already have clauses in it.

pat = SingleCycleChange(net0,net1,m,d0,d1,s0,s1,q,nets)

implementation of the normal operation. It has nine inputs and one output.

The inputs are:

- net0: netlist at time frame 0
- net1: netlist at time frame 1
- m: bistable element under test
- d0: desired value on D input of bistable element under test at time 0
- d1: desired value on D input of bistable element under test at time 1
- s0: desired value on S input of bistable element under test at time 0
- s1: desired value on S input of bistable element under test at time 1

- `q`: desired value on Q output of bistable element under test at time 0
- `nets`: netlist for sensitization (0 if no sensitization required)

The output is:

- `pat`: a test pattern that satisfies input requirements

The procedure `CompactPats()` takes one argument, `patTable`. Initially, `patTable`, contains the original patterns. The patterns are replaced by the compacted patterns in the same table.

In the rest of this section, we show how these functions are used to generate tests for the bistable elements of the four different architectures. Each architecture is handled in a separate section.

4.2.1 Test Generation for MD-Latches in MD-Latch Architecture

The MD-latch scan chain architecture is shown in Fig. 4.2.1-1. In Section 2.4.1 we showed that the triples for the MD-latch under test were divided in two groups: those with T = 0 and those with T = 1. The test requirements for triples with T = 1 match the shift operation, and the test requirements for triples with T = 0 match the normal operation. There are 8 triples in total, four with T = 1 and four with T = 0. The difference between the triples in each group is the value of the D input and the S input of the latch under test. Each of the four triples corresponds to one of the combinations of values on D and S. The procedure for generating patterns for an MD-latch, `GenPatsMDLatch()`, is shown in Fig. 4.2.1-2. The procedure has 5 inputs:

- `latch`: The latch under test
- `net0`: The netlist for time frame 0
- `net1`: The netlist for time frame 1
- `patTableT0`: A table to store patterns generated for triples with T = 0
- `patTableT1`: A table to store patterns generated for triples with T = 1

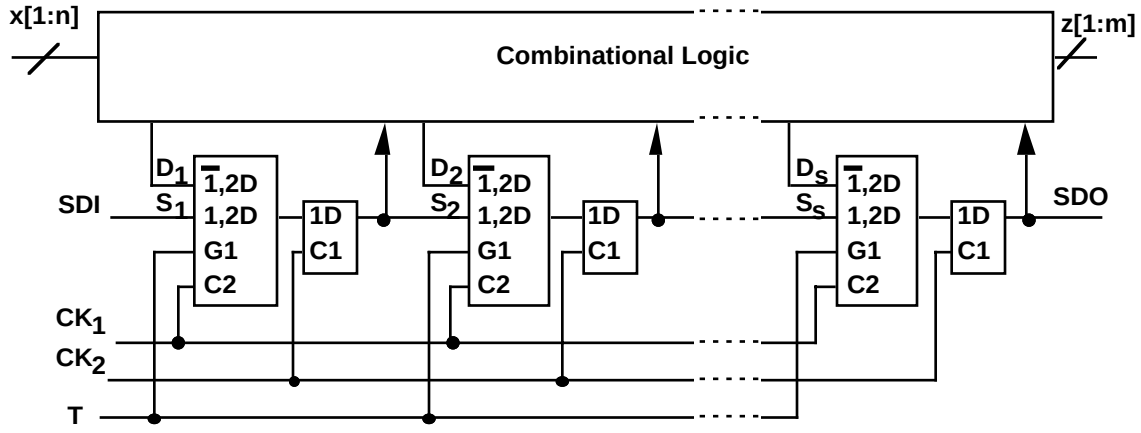


Figure 4.2.1-1 MD-Latch Scan Architecture.

The procedure begins with two nested loops that generate all the four combinations of d (d is the value applied on the D input of the latch) and s (s is the value applied on the S input of the latch). Since our state triples require constant values on d and s for the two cycles, we set the same values for d in both time frames, and the same value for s in both time frames. The initial value of the latch should be selected such that the output of the latch changes after applying a clock cycle. Since the final value of the latch output will be s, we need an initial value that is opposite of s (i.e., 1-s). The above discussion determines the arguments of `ShiftOperation` () shown in the procedure. The result of `ShiftOperation` () is a test pattern that satisfies the requirements on d and s of the latch under test. The four patterns generated by `ShiftOperation` () in the first loop are stored in `patTable1`.

The same steps are repeated for T = 0 triples with three differences:

1. `NormalOperation()` is used instead of `ShiftOperation ()`.
2. The `q` argument of `NormalOperation()` is 1-d instead of 1-s. In normal operation, the D input of the latch becomes the value at the output after a clock cycle. Therefore, since we want the output to change value, we need to start with the opposite value of `d` (i.e., 1-d).
3. `patTable0` is used instead of `patTable1. patTable0` stores all the patterns for triples with T = 0.

Procedure `GenPatsMDLatch()` needs to be called once for every MD-latch in the circuit. The procedure for doing this, TestMDL(), is shown in Fig. 4.2.1-3. The

```
GenPatsMDLatch(latch,net0,net1,patTable0,patTable1){
   /* Triples with T = 1 */
   for (d = 0 ; d < 2; d++){
      for (s = 0; s < 2; s++){
        pat = ShiftOperation(net0,net1,latch,d,d,s,s,1-s,0)
        AddPatToTable(patTable1,pat);
      }
   }
   /* Triples with T = 0 */
   for (d = 0 ; d < 2; d++){
      for (s = 0; s < 2; s++){
        pat = NormalOperation(net0,net1,latch,d,d,s,s,1-d,0)
        AddPatToTable(patTable0,pat);
      }
   }
}
```

Figure 4.2.1-2 Procedure GenPatsMDLatch() Generates Patterns for a Scan Chain MD-Latch.

procedure is divided into three parts:

1. Initialization: In these steps the network is duplicated for time frame 0 and time frame 1, and the pattern tables are set up.
2. Test Generation: `GenPatsMDLatch()` is called for each latch in the design, and the patterns generated are stored in `patTable0` and `patTable1`.
3. Test Compaction: Patterns in each table are compacted using procedure `CompactPats()`. After compaction, the patterns are written to two separate files T0 (patterns of triples with T = 0) and T1 (patterns of triples with T = 1).

The results of running procedure TestMDL() is two files T0 and T1 containing the desired test patterns. These test patterns are applied to the circuit by:

1. Setting T = 1 (scan mode).
2. Shifting the test pattern values into the MD-latches.
3. Setting the corresponding values for time frame 0 on the primary inputs.
4. Setting T = 0 (T = 1) for T0 (T1) patterns, and after sufficient time for the combinational logic to settle, check the Z outputs of the circuit.
5. Applying a clock cycle to CK_1 and CK_2. The corresponding values for time frame 1 on the primary inputs are set on the rising edge of CK_2.
6. Setting T = 1 and shifting out the MD-latch contents. The next pattern can be shifted in at the same time.

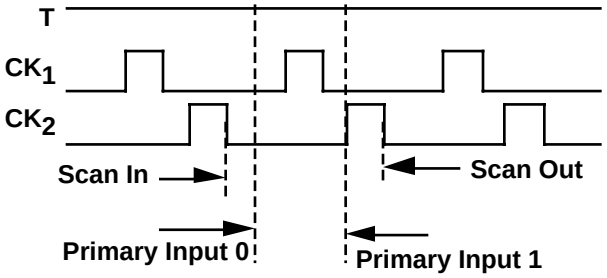
This procedure is similar to that for applying scan patterns for combinational logic

```
TestMDL(network){
   /* Initialization */
   net0 = dup(network); /* duplicate network for time 0 */
   net1 = dup(network); /* duplicate network for time 1 */
   patTable0 = newTable(); /* allocate mem for tables */
   patTable1 = newTable();
   /* Test Generation */
   foreachlatch(l,network) {
      GenPatsMDLatch(l,net0,net1,patTable0,patTable1);
   }
   /*Test Compaction */
   CompactPats(patTable0);
   CompactPats(patTable1);
   WritePats("T0",PatTable0);
   WritePats("T1",PatTable1);
}
```

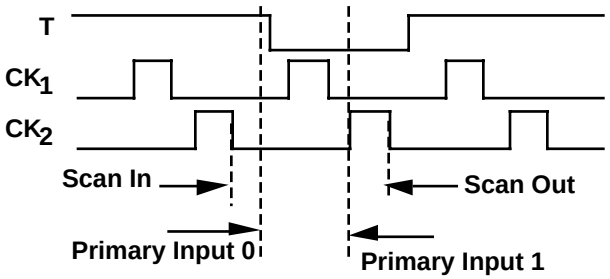
Figure 4.2.1-3 Procedure TestMDL() Generates Patterns for All MD-Latches.

[McCluskey 86]. The main difference is that in step 4, T remains 1 for some of the patterns. In step 4 we checked the outputs of the combinational logic. Even though this is not required for testing the latches, the patterns are likely to detect many faults in the combinational logic. The corresponding values for time frame 1 are applied on the rising transition of CK_2 because this is the point when the D-latch outputs, which are also combinational logic inputs, start to change. Changing the primary inputs at this time ensures keeping the same value on the D input of the latch under test until its output is captured on the falling edge of CK_2.

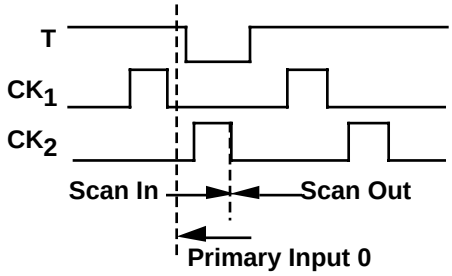
These procedures can be described using the waveforms shown in Fig. 4.2.1-4. Fig. 4.2.1-4a shows waveforms that correspond to the procedure for applying T1 patterns to the circuit. Figs. 4.2.1-4b and c show waveforms that correspond to the procedures for applying T0 patterns to the circuit.



(a) Timing for Patterns in T1



(b) Timing for T0 Patterns



(c) Timing for T0 Patterns for Setup States

Figure 4.2.1-4 Waveforms for Applying Test Patterns for MD-Latch Scan Chain.

As mentioned earlier, we will need to repeat the T0 patterns so that we can capture the outputs of the setup state of triples with T = 0. This is done by repeating the above procedure (for T0 only) except that:

1. T is set to 0 half a cycle before it is set in the previous procedure.
2. T is set to 1 after only half a cycle.

4.2.2 Test Generation for TP-Latches in LSSD Architecture

The LSSD scan chain architecture is shown in Fig. 4.2.2-1. In Section 2.4.2 we showed that the triples for the TP-latch under test are divided into two groups: those with CK_1 changing and those with TCK changing. The test requirements for triples with TCK changing match the shift operation, and the test requirements for triples with CK_1 changing match the normal operation. There are 8 triples in total, four with TCK changing and four with CK_1 changing. The difference between the triples in each group is the value of the D input and the S input of the latch under test. Each of the four triples corresponds to one of the combinations of values on D and S. The procedure for generating patterns for a TP-latch, `GenPatsTPLatch()`, is shown in Fig. 4.2.2-2. The procedure has 5 inputs:

- `latch`: The latch under test
- `net0`: The netlist for time frame 0
- `net1`: The netlist for time frame 1
- `patTableCK1`: A table to store patterns generated for triples with CK_1
- `patTableTCK`: A table to store patterns generated for triples with TCK

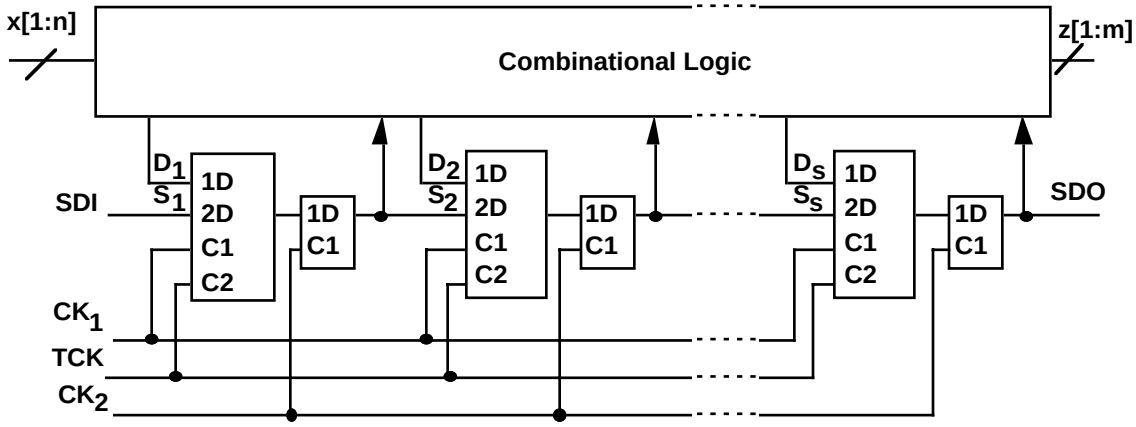


Figure 4.2.2-1 LSSD Architecture.

The procedure is very similar to `GenPatsMDLatch()` of the previous section. For each combination of s and d, the function `ShiftOperation` () is called to generate a pattern that will satisfy the requirements of s and d. The four patterns generated by `ShiftOperation` () in the first loop are stored in `patTableTCK1`.

The same steps are repeated for CK_1 triples with three differences:

1. `NormalOperation()` is used instead of `ShiftOperation()`.
2. The q argument of `NormalOperation()` is 1-d instead of 1-s. In normal operation the D input of the latch becomes the value at the output. Therefore, since we want the output to change value, we need to start with

the opposite value (i.e., 1-d).

3. `patTableCK1` is used instead of `patTableTCK`. `patTableCK1` stores all the patterns for CK_1 triples.

As with `GenPatsMDLatch()`, procedure `GenPatsTPLatch()` needs to be called once for every TP-latch in the circuit. The procedure for doing this, TestTPL(), is shown in Fig. 4.2.2-3.

```
GenPatsTPLatch(latch,net0,net1,patTableCK1,patTableTCK){
   /* Triples with TCK changing */
   for (d = 0 ; d < 2; d++){
      for (s = 0; s < 2; s++){
        pat = ShiftOperation(net0,net1,latch,d,d,s,s,1-s,0)
        AddPatToTable(patTableTCK,pat);
      }
   }
   /* Triples with CK1 changing */
   for (d = 0 ; d < 2; d++){
      for (s = 0; s < 2; s++){
        pat = NormalOperation(net0,net1,latch,d,d,s,s,1-d,0)
        AddPatToTable(patTableCK1,pat);
      }
   }
}
```

Figure 4.2.2-2 Procedure GenPatsTPLatch() Generates Patterns for a Scan Chain TP-Latch.

```
TestTPL(network){
   /* Initialization */
   net0 = dup(network); /* duplicate network for time 0 */
   net1 = dup(network); /* duplicate network for time 1 */
   patTableCK1 = newTable(); /* allocate mem for tables */
   patTableTCK = newTable();
   /* Test Generation */
   foreachlatch(l,network) {
      GenPatsMDLatch(l,net0,net1,patTableCK1,patTableTCK);
   }
   /*Test Compaction */
   CompactPats(patTableCK1);
   CompactPats(patTableTCK);
   WritePats("CK1",PatTableCK1);
   WritePats("TCK",PatTableTCK);
}
```

Figure 4.2.2-3 Procedure TestTPL() Generates Patterns for All TP-Latches in Scan Chain.

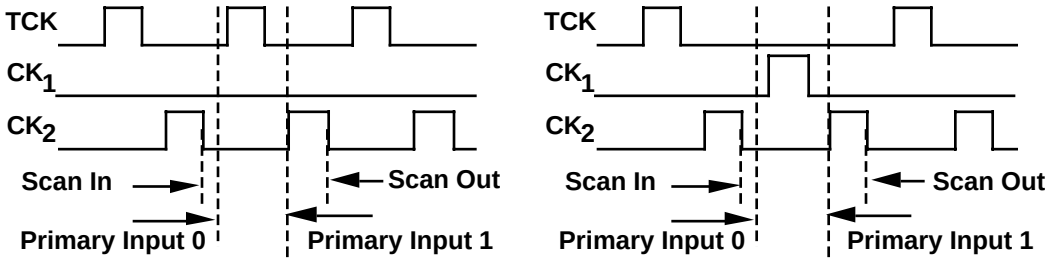
The procedure is divided into three parts:

1. Initialization: In these steps the network is duplicated for time frame 0 and time frame 1, and the pattern tables are set up.
2. Test Generation: `GenPatsTPLatch()` is called for each latch in the design, and the patterns generated are stored in `patTableCK1` and `patTableTCK`.
3. Test Compaction: Patterns in each table are compacted using procedure `CompactPats()`.

After compaction, the patterns are written to two separate files TCK (patterns of triples with TCK changing) and CK1 (patterns of triples with CK1 changing). The tests are applied to the circuit by:

1. Shifting the test pattern values into the TP-latches by using TCK and CK2.
2. Setting the corresponding values for time frame 0 on the primary inputs, and after sufficient time for the combinational logic to settle, check the Z outputs of the circuit..
3. Applying a clock cycle to CK_1(TCK) and CK_2 for CK1(TCK) patterns. The corresponding values for time frame 1 on the primary inputs are set on the rising edge of CK_2.
4. Shifting out the test pattern using TCK and CK_2. The next pattern can be shifted in at the same time.

This procedure is similar to the one for the MD-latches in the previous sub-section. As with the MD-latches, the corresponding values for time frame 1 are applied on the rising transition of CK_2 because this is the point when the D-latch outputs, which are also combinational logic inputs, start to change. Changing the primary inputs at this time ensures keeping the same value on the D input of the latch under test until its output is captured on the falling edge of CK_2. The above procedure is described using the



(a) Timing for Patterns in TCK **(b) Timing for Patterns in CK_1**

Figure 4.2.2-4 Waveforms for Applying Test Patterns for LSSD Scan Chain.

waveforms in Fig. 4.2.2-4. Unlike the MD-latch architecture, we need only apply each pattern once. As explained in Chapter 2, the setup states of CK_1 triples are identified by patterns of TCK triples.

4.2.3 Test Generation for MD Flip-Flops in MD Flip-Flop Architecture

The MD flip-flop scan chain architecture is shown in Fig. 4.2.3-1. Unlike the latch-based architectures, the flip-flop-based architectures do not have state triples. Instead, they have sequences that are required to identify unstable states. In Chapter 3, the unstable states were categorized into groups, and each group had waveforms that described the sequences of the group. Fig. 4.2.3-2 shows waveforms for the MD flip-flop.

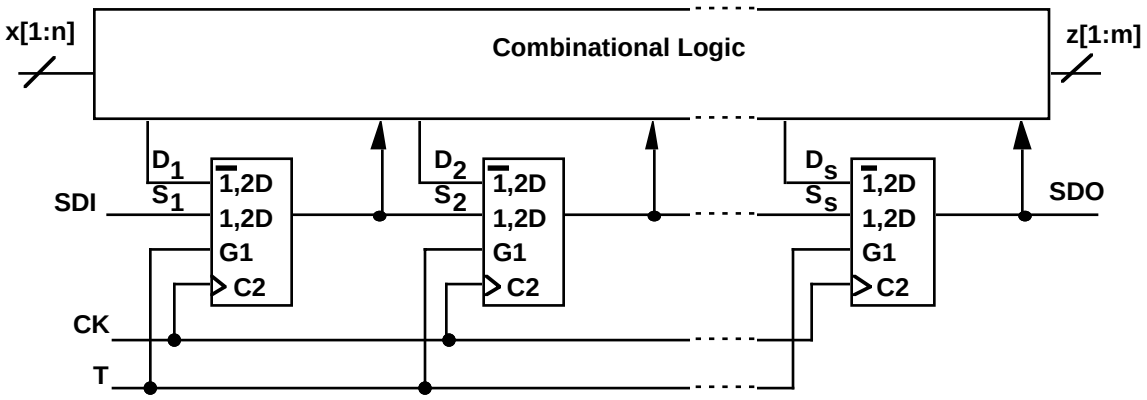
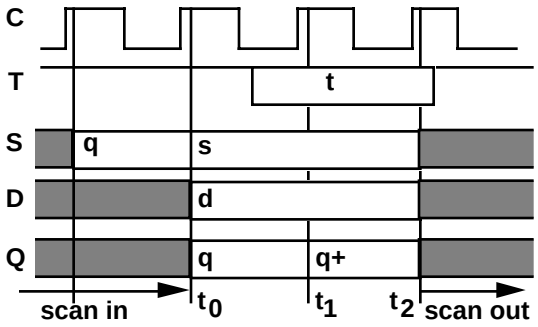
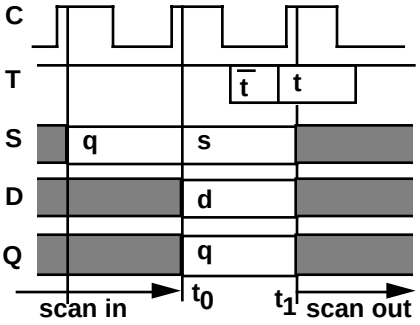


Figure 4.2.3-1 MD Flip-Flop Based Scan Architecture.

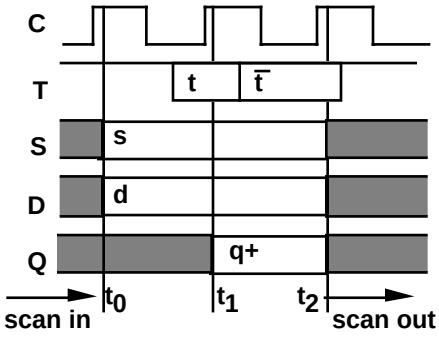
In the last two sections we had two pattern tables to store the patterns, since we only had two types of patterns. In the case of the MD flip-flop, each group has different waveforms so we will have more than two pattern tables. All groups except E1 and E2 have t in the T waveform. Having t in the T waveform implies that there are really two waveforms, one with t = 0 and one with t = 1, and the group can be split in two. For example, group A can be split into A with t = 0, and A with t = 1. Groups are split this way because patterns in a table must have the same waveform for T. Since we expect one pattern table for each split group, we would expect 14 tables for the MD flip-flop. However, some of the groups have the same waveforms for T, and can therefore share the same table. This is useful because fewer tables imply fewer patterns. There are a total of six tables. The six tables, and the groups they include are listed in Table 4.2.3-1.



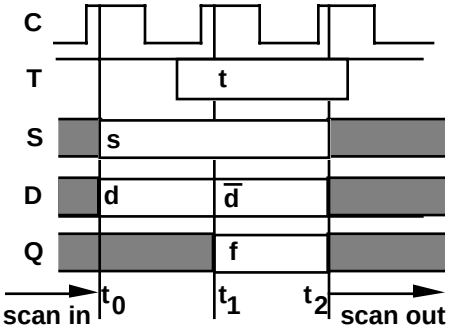
(a) Waveforms for Group A



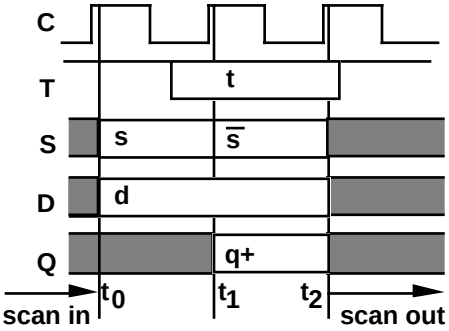
(b) Waveforms for Group B



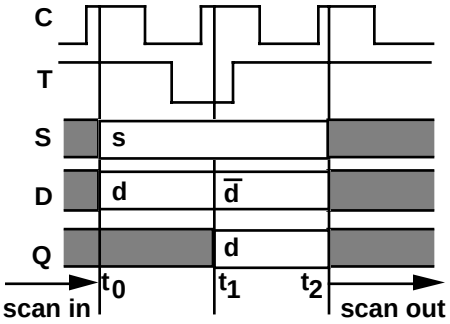
(c) Waveforms for Group C



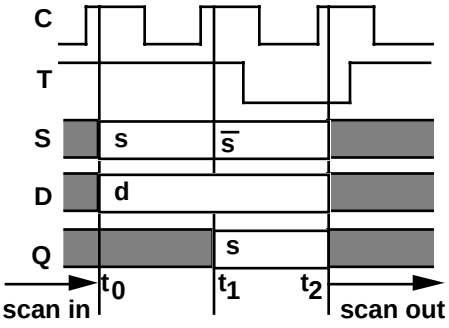
(d) Waveforms for Group D1



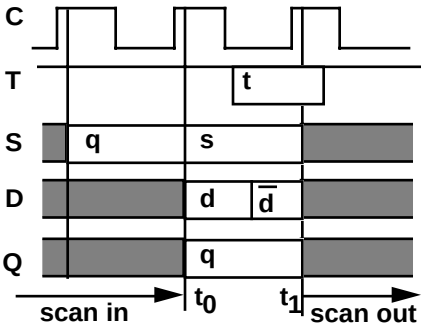
(e) Waveforms for Group D2



(f) Waveforms for Group E1



(g) Waveforms for Group E2



(h) Waveforms for Group F

Figure 4.2.3-2 Waveforms for Unstable State Groups for MD Flip-Flop. ($q+ = ts + \bar{t}d$) .

Table 4.2.3-1 Pattern Tables and Their Groups for MD Flip-Flops.

Pattern Table	Groups
A0	A, D1, D2 (all with t = 0)
A1	A, D1, D2, F (all with t = 1)
B0	B (with t = 0)
B1	B (with t = 1)
C0	C, F (with t = 0), E1, E2
C1	C (with t = 0)

The procedure for generating patterns for an MD flip-flop, `GenPatsMDFF()`, has 10 inputs:

- `ff`: The MD flip-flop under test
- `net0`: The netlist for time frame 0
- `net1`: The netlist for time frame 1
- `pA0`: Pattern Table A0 (see Table 4.2.3-1 for details)
- `pA1`: Pattern Table A1 (see Table 4.2.3-1 for details)
- `pB0`: Pattern Table B0 (see Table 4.2.3-1 for details)
- `pB1`: Pattern Table B1 (see Table 4.2.3-1 for details)
- `pC0`: Pattern Table C0 (see Table 4.2.3-1 for details)
- `pC1`: Pattern Table C1 (see Table 4.2.3-1 for details)
- `nets`: netlist for sensitization

The structure of the procedure is similar to that of the latches. Nested loops are used to generate different combinations of s d and q. Each combination is used to call one of the elementary operations. The resulting pattern is stored in the appropriate pattern table. Since there are many different groups and pattern tables, the handling of the groups is summarized in Table 4.2.3-2. This table shows for each group, how many cycles are needed for the test, the type of operation used to generate the test, the initial inputs or outputs that need to be set, and which input needs to change, the sensitization path (combinational logic or straight through the scan chain), and the pattern table the pattern is stored in. Fig. 4.2.3-3 shows part of `GenPatsMDFF()`. Procedure `GenPatsMDFF()` needs to be called once for every MD flip-flop in the circuit. The procedure for doing this, TestMDFF(), is shown in Fig. 4.2.3-4.

Table 4.2.3-2 Handling Groups in ATPG for MD Flip-Flop Scan Chain.

Group	t	Cycles	Operation	Initial Set	Change Input	Sense Path	Pattern Table
A	0	2	Normal	S,D,Q	–	CL	A0
A	1	2	Shift	S,D,Q	–	Scan	A1
B	0	1	Single	S,D,Q	–	CL	B0
B	1	1	Single	S,D,Q	–	Scan	B1
C	0	2	Normal	S,D	–	Scan	C0
C	1	2	Shift	S,D	–	CL	C1
D	0	2	Normal	S,D	D or S	CL	A0
D	1	2	Shift	S,D	D or S	Scan	A1
E_1	–	2	Normal	S,D	D	Scan	C0
E_2	–	2	Shift	S,D	S	CL	C1
F	0	1	SingleC	S,D	D	CL	C0
F	1	1	SingleC	S,D,Q	D	Scan	A1

```
GenPatsMDFF(ff,net0,net1,pA0,pA1,pB0,pB1,pC0,pC1,nets){
  /* Waveform A, t = 1 */
  for (d = 0 ; d < 2; d++){
    for (s = 0; s < 2; s++){
      for (q = 0; q < 2; q++){
        pat = ShiftOperation(net0,net1,ff,d,d,s,s,q,0);
        AddPatToTable(pA1,pat);
      }
    }
  }
  /* Waveform A, t = 0 */
  for (d = 0 ; d < 2; d++){
    for (s = 0; s < 2; s++){
      for (q = 0; q < 2; q++){
        pat = NormalOperation(net0,net1,ff,d,d,s,s,q,nets);
        AddPatToTable(pA0,pat);
      }
    }
  }
  :
```

Figure 4.2.3-3 Part of Procedure GenPatsMDFF() Generates Patterns for a Scan Chain MD Flip-Flop.

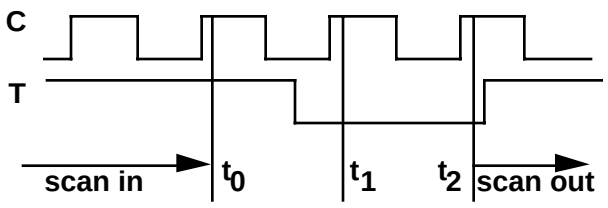
```
TestMDFF(network){
   /* Initialization */
   net0 = dup(network); /* duplicate network for time 0 */
   net1 = dup(network); /* duplicate network for time 1 */
   nets = dup(network); /* duplicate network for sensitize */
   pA0 = newTable(); /* allocate mem for tables */
   :
   pC1 = newTable();
   /* Test Generation */
   foreachff(f,network) {
      GenPatsMDFF(f,net0,net1,pA0,pA1,pB0,pB1,pC0,pC1,nets);
   }
   /*Test Compaction */
   CompactPats(pA0);
   :
   CompactPats(pB1);
   WritePats("A0",pA0);
   :
   WritePats("C1",pA1);
}
```

Figure 4.2.3-4 Procedure TestMDFF() Generates Patterns for All MD Flip-Flops in Scan Chain.

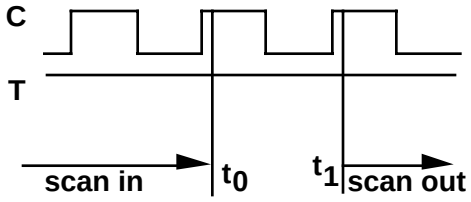
The procedure is divided into three parts:

1. Initialization: In these steps the network is duplicated for time frame 0, time frame 1 and path sensitization. Also, the pattern tables are set up.
2. Test Generation: `GenPatsMDFF()` is called for each flip-flop in the design, and the patterns generated are stored in `pA0`, `pA1`, `pB0`, `pB1`, `pC0` and `pC1`.
3. Test Compaction: Patterns in each table are compacted using procedure `CompactPats()`.

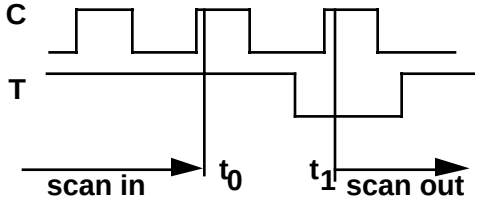
The procedure for applying these patterns to the circuit is similar to that of the latches in the last two sections, except that we have six waveforms to follow. These waveforms are shown in Fig. 4.2.3-5. These waveforms describe the procedure for applying the patterns to the chip. Primary inputs are applied at t_0 and t_1.



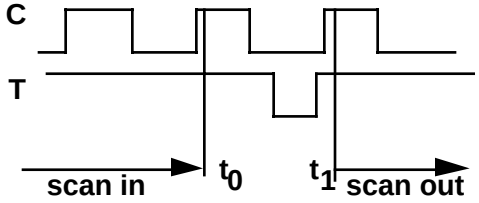
(a) Waveform A0



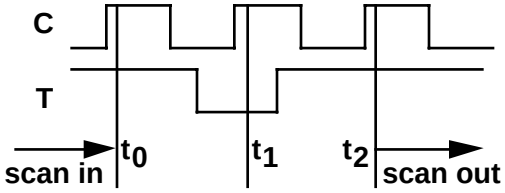
(b) Waveform A1



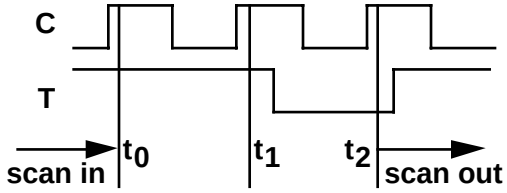
(c) Waveform B0



(d) Waveform B1



(e) Waveform C0



(f) Waveform C1

Figure 4.2.3-5 Waveforms for Applying Test Patterns for MD Flip-Flop Scan Chain.

4.2.4 Test Generation for TP Flip-Flops in TP Flip-Flop Architecture

The TP flip-flop scan chain architecture is shown in Fig. 4.2.4-1. Unlike the latch-based architectures, the flip-flop-based architectures do not have state triples. Instead, they have sequences that are required to identify unstable states. In Chapter 3 the unstable states were categorized into groups, and each group had waveforms that described the sequences of the group. The waveforms for the TP flip-flop are shown in Fig. 4.2.4-2.

Unlike the MD flip-flop waveforms, the waveforms for the TP flip-flop fall into only two types: those with a C_1 pulse and those without one. Therefore, we will use only two pattern tables. The two tables, and the groups they include are listed in Table 4.2.4-1.

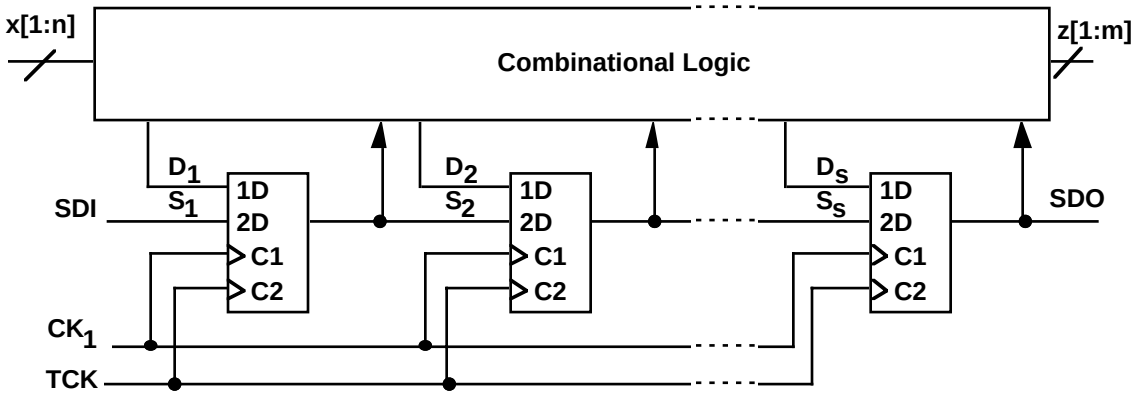


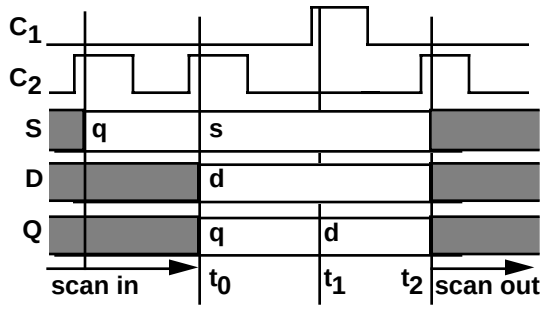
Figure 4.2.4-1 TP Flip-Flop Based Scan Architecture.

Table 4.2.4-1 Pattern Tables and Their Groups for TP Flip-Flops.

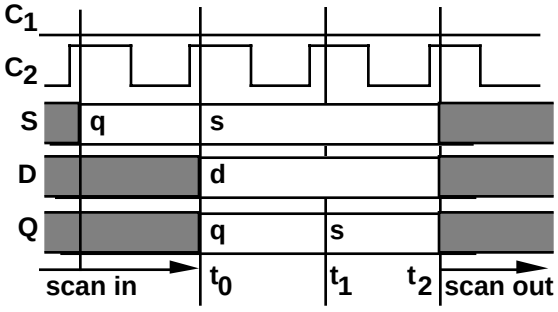
Pattern Table	Groups
A1	A1, B1, C1
A2	A2, B2, C2, D

The procedure for generating patterns for an TP flip-flop, `GenPatsTPFF()`, has 5 inputs:

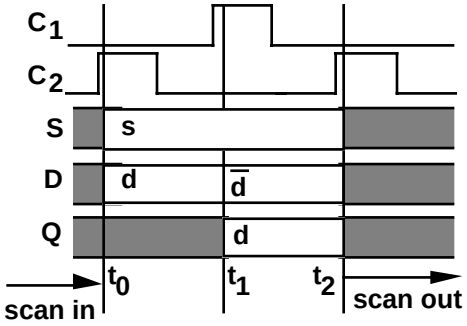
- `ff`: The MD flip-flop under test
- `net0`: The netlist for time frame 0
- `net1`: The netlist for time frame 1
- `pA1`: Pattern Table A1 (see Table 4.2.4-1 for details)
- `pA2`: Pattern Table A2 (see Table 4.2.4-1 for details)



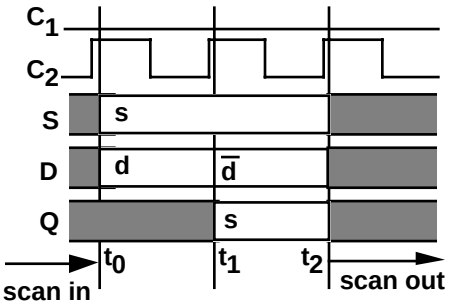
(a) Waveforms for Group A1



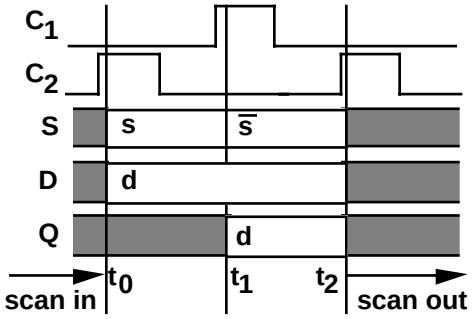
(b) Waveforms for Group A2



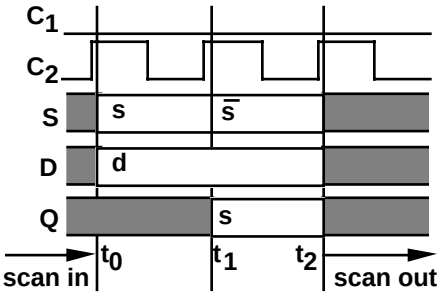
(c) Waveforms for Group B1



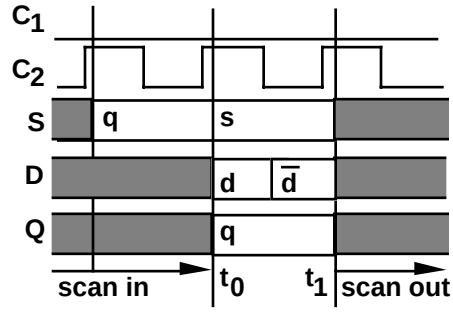
(d) Waveforms for Group B2



(e) Waveforms for Group C1



(f) Waveforms for Group C2



(g) Waveforms for Group D

Figure 4.2.4-2 Waveforms for Unstable State Groups for TP Flip-Flop.

The structure of GenPatsTPFF() is similar to `GenPatsMDFF()`. Nested loops are used to generate different combinations of s, d and q. Each combination is used to call one of the elementary operations. The resulting pattern is stored in the appropriate pattern table. Since there are many different groups, the handling of the groups is summarized in Table 4.2.4-2. This table shows for each group, how many cycles are needed for the test, the type of operation used to generate the test, the initial inputs or outputs that need to be set, and which input needs to change, and the pattern table the pattern is stored in. Fig. 4.2.4-3 shows part of `GenPatsTPFF()`.

```
GenPatsTPFF(ff,net0,net1,pA1,pA2){
   /* Waveform A1 */
   for (d = 0 ; d < 2; d++){
      for (s = 0; s < 2; s++){
        for (q = 0; q < 2; q++){
          pat = NormalOperation(net0,net1,ff,d,d,s,s,q,0);
          AddPatToTable(pA1,pat);
        }
      }
   }
   /* Waveform A2  */
   for (d = 0 ; d < 2; d++){
      for (s = 0; s < 2; s++){
        for (q = 0; q < 2; q++){
          pat = ShiftOperation(net0,net1,ff,d,d,s,s,q,0);
          AddPatToTable(pA2,pat);
        }
      }
   }
   :
```

Figure 4.2.3-3 Part of Procedure GenPatsTPFF() Generates Patterns for a Scan Chain TP Flip-Flop.

Table 4.2.4-2 Handling Groups in ATPG for TP Flip-Flop Scan Chain.

Group	Cycles	Operation	Initial Set	Change Input	Sense Path	Pattern Table
A1	2	Normal	S,D,Q	–	Scan	A1
A2	2	Shift	S,D,Q	–	Scan	A2
B1	2	Normal	S,D	D	Scan	A1
B2	2	Shift	S,D	D	Scan	A2
C1	2	Normal	S,D	S	Scan	A1
C2	2	Shift	S,D	S	Scan	A2
D	2	Shift	S,D	D	Scan	A1

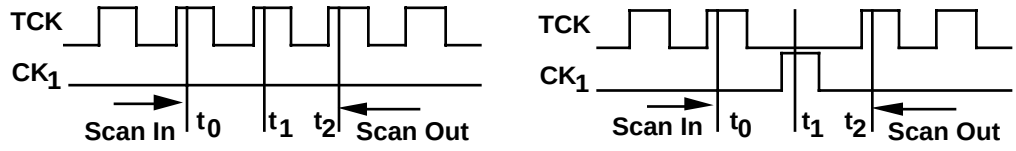
Procedure `GenPatsTPFF()` needs to be called once for every TP flip-flop in the circuit. The procedure for doing this, TestTPFF(), is shown in Fig. 4.2.3-4. The procedure is divided into three parts:

1. Initialization: In these steps the network is duplicated for time frame 0 and time frame 1, and the pattern tables are set up.
2. Test Generation: `GenPatsTPFF()` is called for each flip-flop in the design, and the patterns generated are stored in `pA1`, `pA2`.
3. Test Compaction: Patterns in each table are compacted using procedure `CompactPats()`.

```
TestTPFF(network){
   /* Initialization */
   net0 = dup(network); /* duplicate network for time 0 */
   net1 = dup(network); /* duplicate network for time 1 */
   pA1 = newTable(); /* allocate mem for tables */
   pA2 = newTable();
   /* Test Generation */
   foreachff(f,network) {
      GenPatsMDFF(f,net0,net1,pA0,pA1,pB0,pB1,pC0,pC1);
   }
   /*Test Compaction */
   CompactPats(pA1);
   CompactPats(pA2);
   WritePats("A1",pA1);
   WritePats("A2",pA2);
}
```

Figure 4.2.3-4 Procedure TestTPFF() Generates Patterns for All TP Flip-Flops in Scan Chain.

The procedure for applying these patterns to the circuit is similar to that of the MD flip-flops in the last section, except that we have two waveforms to follow. These waveforms are shown in Fig. 4.2.4-5. These waveforms describe the procedure for applying the patterns to the circuit. As with the MD flip-flop architecture, primary inputs are applied at t_0 and t_1.



(a) Waveform A1 **(b) Waveform A2**

Figure 4.2.3-5 Waveforms for Applying Test Patterns for TP Flip-Flop Scan Chain.

4.3 Summary

In this chapter, we used the results of Chapters 2 and 3 to modify a stuck-at fault automatic test generation program for stuck-at faults to generate patterns to test the bistable elements in the design. Our implementation is a modified version of SIS, with four new functions. Given a netlist, each function generates patterns for all the bistable elements of one of the four scan architectures. Chapter 6 has some results of running our implementation on some benchmark circuits.

Chapter 5. Fault Simulation

The effectiveness of a test can be measured by the number of defects it can detect. As mentioned in Chapter 1, fault models are often used to represent defects, and the effectiveness of tests is measured by fault simulation. Even though the stuck-at models are often used for fault simulation, we use the more accurate (for CMOS circuits) CrossCheck fault models, [Sucar 89; Chandra et. al. 93], for our simulation. The fault models comprise shorted interconnects (STI), open interconnects (OPI), short-to-power (STP), short-to-ground (STG), transistor stuck-on (SON), and transistor stuck-open (SOP). In the simulations, faults are injected by modifying a copy of the circuit description. The faulty circuits were simulated using HSpice [Kielkowski 94].

In CMOS, there are some faults whose presence does not change the functionality of the host circuit. Some of these cannot be detected (and thus are untestable or redundant). Others that cannot be detected by a Boolean voltage test (since the circuit functionality is correct) can, nevertheless, be discovered by a current test or a delay test [Ma and McCluskey 95]. The simulations reported here record whether tests caused excessive supply current (IDDQ) as well as incorrect outputs. The current limit for IDDQ testing is often determined experimentally, by plotting the values of many good and bad die, and selecting an appropriate threshold that would detect as many faulty circuits as possible without discarding many good ones [Hawkins 89] and [Perry 92]. For our simulations, the current limit is determined by plotting the maximum observed current for each fault, and selecting an appropriate threshold from the graph.

Four different bistable elements were simulated, first with traditional tests, and then with checking experiments. Each test was run twice, once with a cycle time (*cycle time* here is defined to be the time between the application of inputs) of 100 ns, and once with a cycle time of 10 ms. Outputs were measured just before applying the next input. The 100 ns cycle time is a typical test time for a boolean test, and the 10 ms cycle time is needed to allow IDD to settle to its quiescent value. For all four bistable elements, we selected a threshold 100,000 times the IDDQ current of the fault-free circuit based on a plot of the maximum currents of the faulty circuits. For each test several numbers are reported: the number of faults detected by either boolean or IDDQ testing, the number of faults detected by boolean testing at 100 ns or 10 ms, the number of faults detected by boolean testing at 100 ns alone and by testing at 10 ms alone, and number of faults detected when only an IDDQ test is done.

In all four bistable elements the checking experiments detected faults missed by the traditional test, showing that they are more effective than the traditional tests. The checking experiments do miss some faults, but after a detailed analysis, these faults are shown to be untestable. Part of the analysis is presented in this chapter, and the rest is presented in Appendix C.

Even though checking experiments were generated to verify the functionality of the flip-flops, the results show that, in addition to detecting functional faults, they are very useful in detecting faults that only cause excessive current.

5.1 Fault Simulation of D-Latch

We simulated four different tests for the D-latch. The first test is a pin fault test set, which targets stuck-at faults on the input and output of the D-latch. A D-latch can be viewed as a multiplexer that selects between D and Q, with C being the select signal. The second test is a multiplexer-based test. Patterns for testing multiplexers can be found in Makar and McCluskey [88]. The third test is a D-latch checking experiment (see Section 2.2.1) and the fourth test is a checking experiment for a D-latch in a shift register (see Section 2.3). The four tests are shown in Fig. 5.1-1. Dashed vertical lines show when the outputs are checked. The outputs of all except the last test are checked after every input

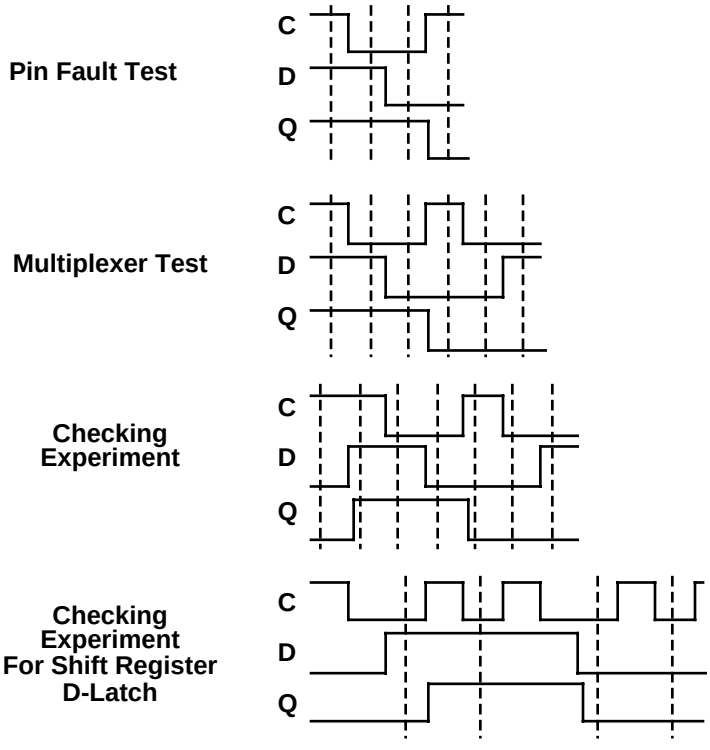


Figure 5.1-1 Tests Applied to D-Latch (Dashed Lines Indicate When Output Checked).

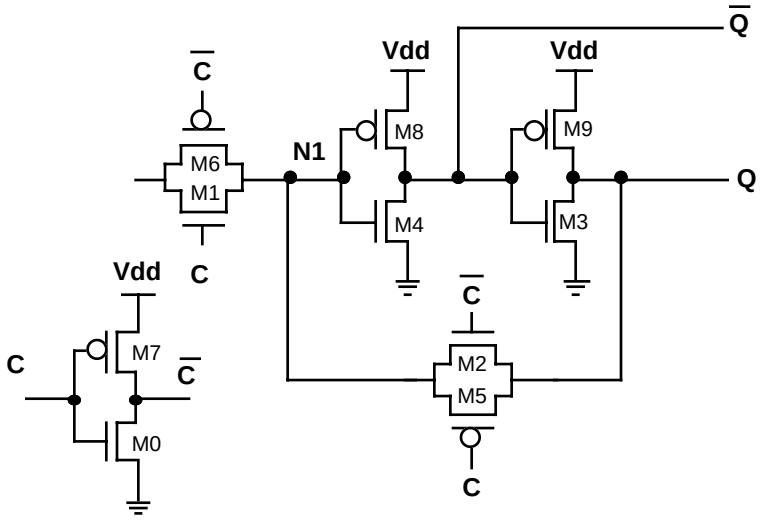


Figure 5.1-2 Transmission Gate D-Latch.

change. For the last test, the output is only checked before a positive transition of C, as this is the capture point for the shift register. Fig. 5.1-2 shows the simulated implementation. This implementation is chosen because it is commonly used in designs for its small size.

Fig. 5.1-3 shows the distribution of maximum IDDQ values for the faulty circuits. Here IDDQf refers to the IDDQ value of a faulty circuit, and IDDQg refers to the IDDQ value of the fault-free circuit (g for good). The graph plots the ratio of IDDQf / IDDQg (i.e., the ratio of IDDQ increase), versus the faults. The graph shows that all but 7 of the faults have an IDDQ value that is over 100,000 times that of the fault-free circuit IDDQ current. The graph also shows a sharp rise in IDDQf / IDDQg from 1 to 100,000. Such a sharp break in the graph makes for a good IDDQ threshold. Therefore, the IDDQ

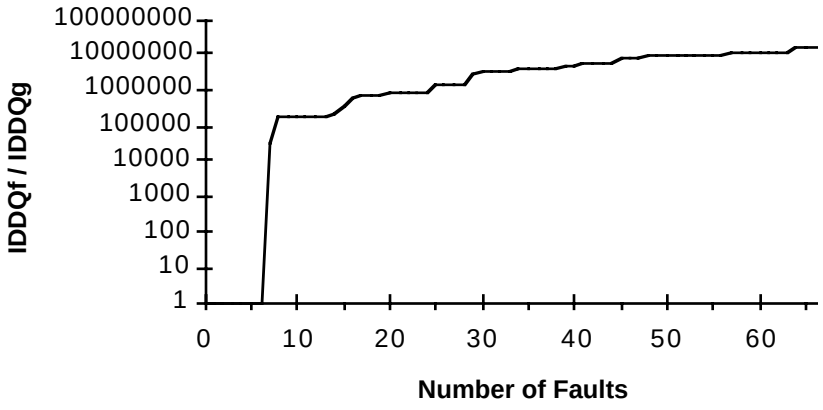


Figure 5.1-3 Current Distribution Graph for D-Latch (IDDQg = 320 pA).

threshold selected is 100,000 times the IDDQ current of the fault-free circuit. The maximum IDDQ value for the fault-free circuit was 320 pA, thus the threshold selected was 32 uA.

The results of the simulations are shown in Table 5.1-1. The table shows that the pin fault test and the multiplexer test miss several faults that are detected by the checking experiment. The faults missed by each of the tests are shown graphically in Figs. 5.1-4 through 5.1-7. In these figures white ovals indicate SOP or OPI faults, black ovals indicate SON faults, and thick black lines indicate STI faults. All STP and STG faults are detected by all three tests.

Table 5.1-1 Number of Faults Detected in D-Latch (Total Faults = 67).

	Boolean and IDDQ	Boolean Alone (100 ns and 10 ms)	Boolean Alone (100 ns, 10 ms)	IDDQ Alone
Pin Fault Test	54	35	(32,32)	34
Multiplexer Test	59	36	(33,36)	43
D-Latch Checking Exp.	64	47	(47,45)	60
Shift Reg D-Latch Checking Exp.	66	47	(47,43)	61

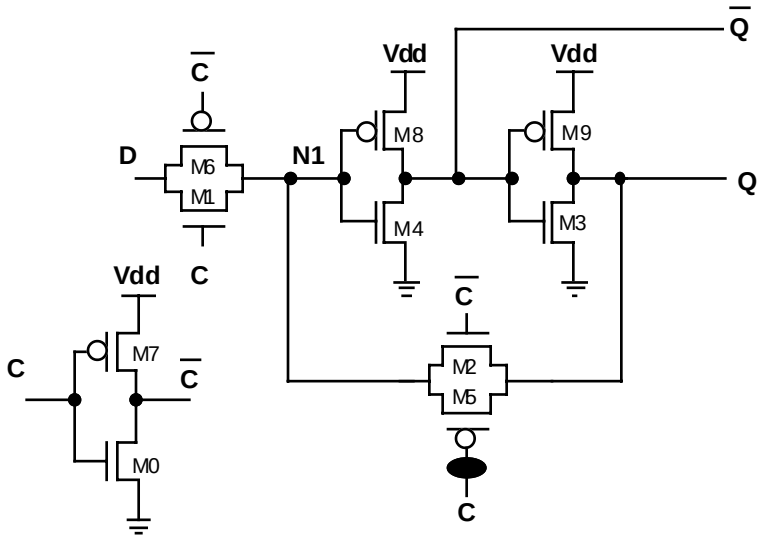


Figure 5.1-4 Faults Missed by Checking Experiment for D-Latch in Shift Register.

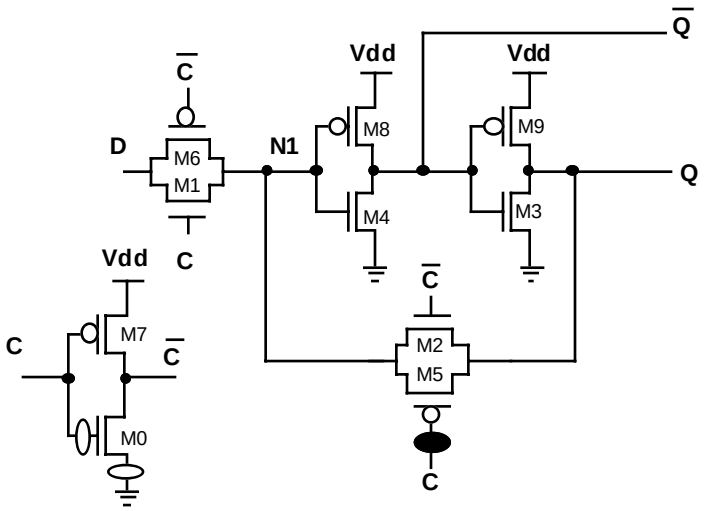


Figure 5.1-5 Faults Missed by Checking Experiment for D-Latch.

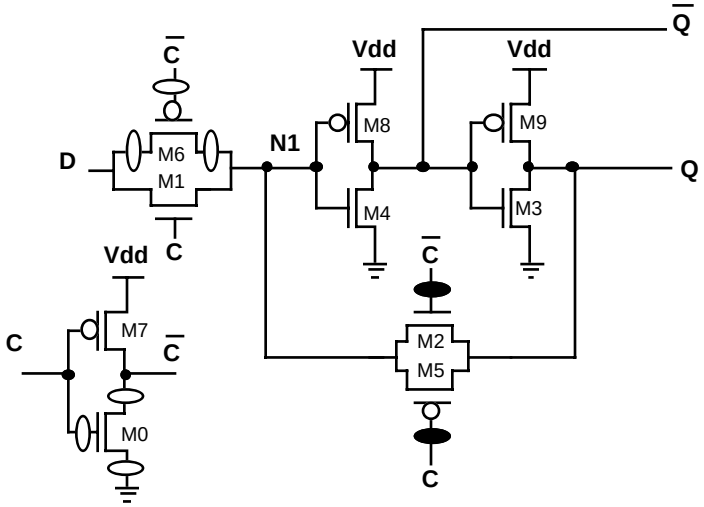


Figure 5.1-6 Faults Missed by Multiplexer Test for D-Latch.

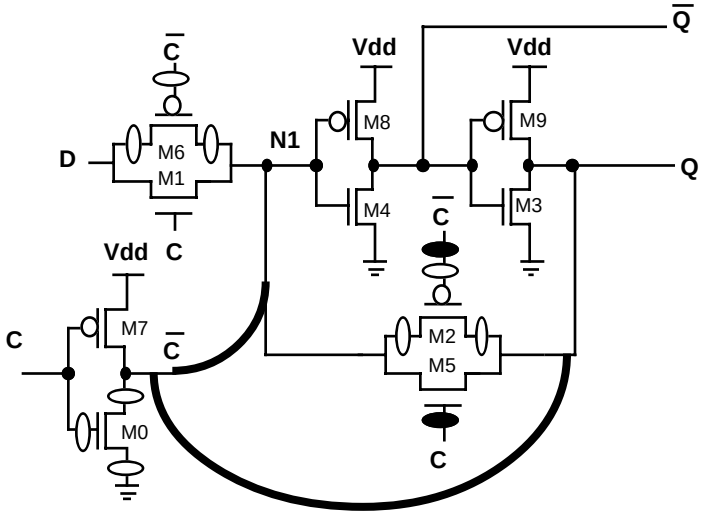


Figure 5.1-7 Faults Missed by Pin Fault Test for D-Latch.

The faults missed by the checking experiment fall into one of two categories. The faults on the M0 transistor cause the number of states to increase. With the presence of this fault, the application of CD = 10 in state 7 (see Table 5.1-2), produces an output of 0. However, when the same input is applied to state 3, the output is 1. Therefore, states 7 and 3 cannot be in the same internal state, and there must be a third internal state. The stuck-on fault on M5 is not detected because it is on the feedback path. Even though not indicated in the figures, the NMOS transistors of the transmission gates are stronger than their PMOS counterparts. This makes it possible to detect M2 stuck-on, but not detect M5 stuck-on. M5 stuck-on was the only fault missed by the MD-latch shift register checking experiment, implying that the checking experiment detected all the detectable faults.

Table 5.1-2 Flow Table for Two-State D-Latch.

	CD			Q
00	01	11	10	
(2), 0	(4), 0	7	(6), 0	0
[3], 1	[5], 1	[7], 1	6	1

The results in Table 5.2-1 also show that there many faults are detected only by IDDQ (see Fig. 5.1-8). These faults fall into three groups: stuck-on faults on the inverter PMOS transistors, stuck-open faults that cause degradation of internal signals, and shorted interconnects involving $\overline{Q}$. These faults are discussed in more detail in Appendix C. There are also some faults that are detected only by boolean test, and cannot be detected by IDDQ measurement. These are also discussed in Appendix C.

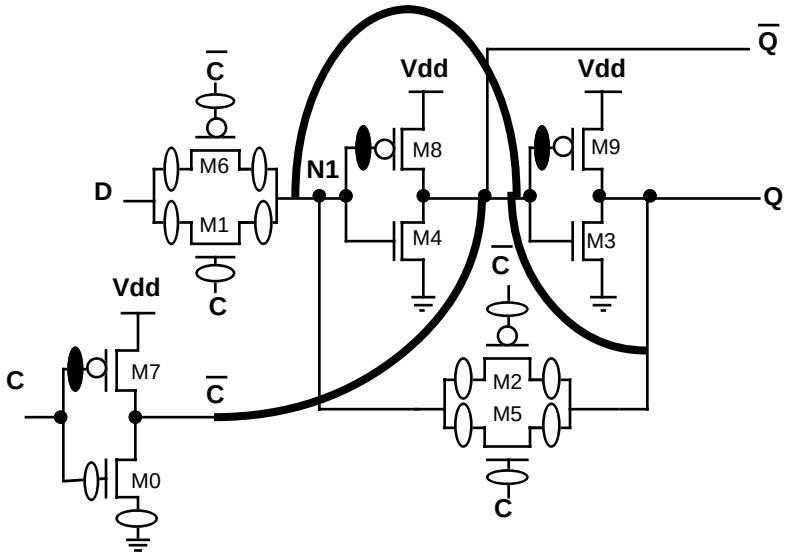


Figure 5.1-8 Faults Detected by IDDQ Test Missed by Boolean Test.

5.2 Fault Simulation of MD-Latch

We simulated three different tests for the MD-latch. The first test, a traditional test for a scan chain, involves scanning in zeroes and ones while ignoring the values on the D inputs of the MD-latches. Such a scan test is shown in Fig 5-2.1. For a fair comparison of the scan test with our test, the scan test is augmented with patterns to detect stuck-at faults on the D input of the MD-latch. Such patterns would be used in the full circuit to test the output of the combinational logic driving the D input of the latch. The other two tests are the MD-latch checking experiment (section 2.2.3) and the checking experiment for the MD-latch in a scan chain (section 2.4.1). The implementation used for simulation is shown in Fig. 5.2-2.

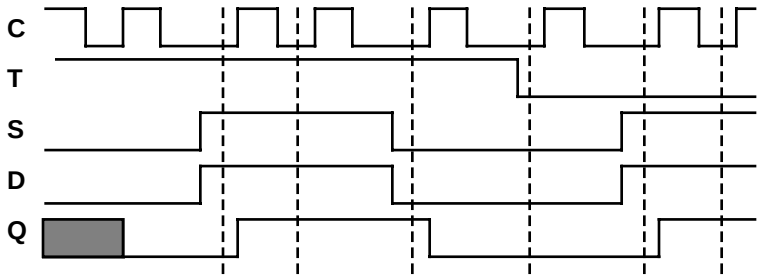


Figure 5.2-1 Traditional Test for MD-Latch in Scan Chain.

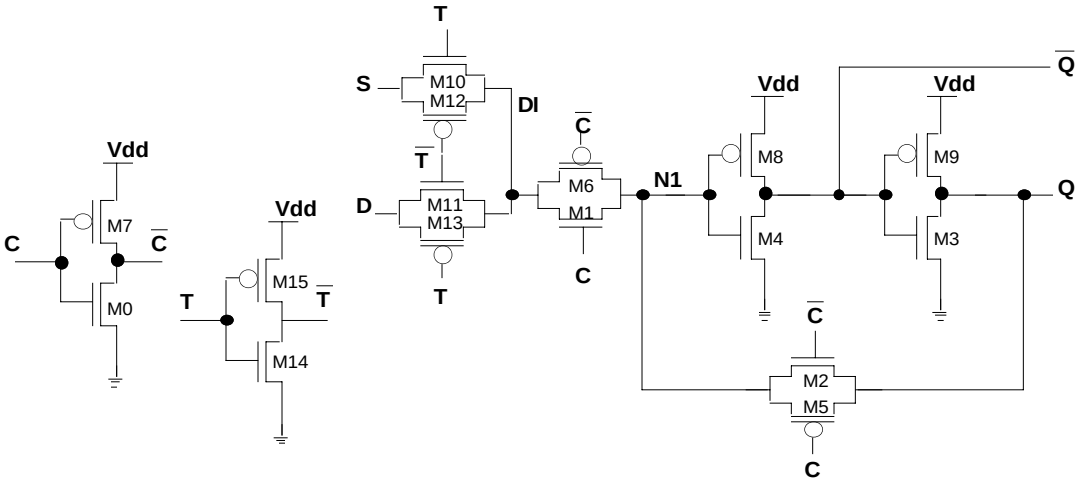


Figure 5.2-2 MD-Latch Implementation.

Fig. 5.2-3 shows the distribution of maximum IDDQ values for the faulty circuits. The graph also shows a sharp rise in IDDQf / IDDQg from 1 to 100,000. Such a sharp break in the graph makes for a good IDDQ threshold. Therefore, the IDDQ threshold selected is 100,000 times the IDDQ current of the fault-free circuit. The maximum IDDQ value for the fault-free circuit was 487 pA, thus the threshold selected was 48.7 uA.

The simulation results are compared in Table 5.2-1. The only fault missed by the checking experiment is M5 stuck-on. This is the same fault missed by the checking experiment for the D-latch. As with the D-latch, the fault is untestable because of the relative strengths of the PMOS and NMOS transistors. The results in Table 5.2-1 show that the scan test misses over 20 percent of the functional faults that were detected by the checking experiment.

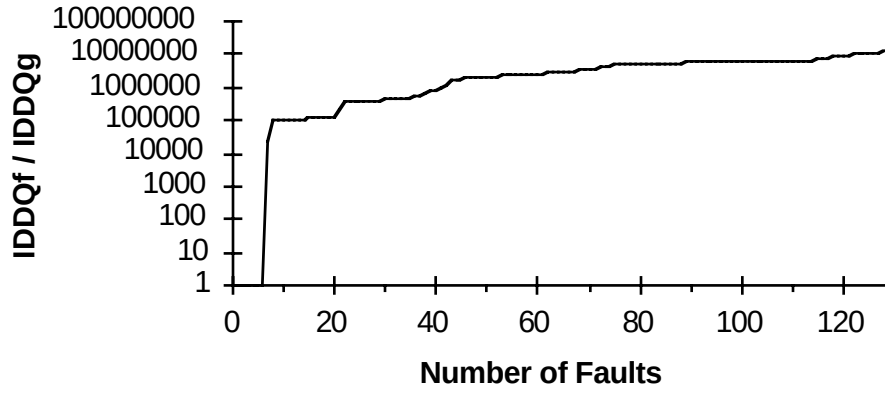


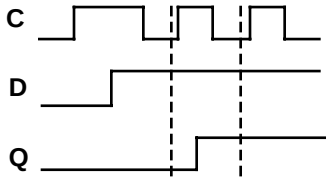
Figure 5.2-3 Current Distribution Graph for MD-Latch (IDDQg = 487 pA).

Table 5.2-1 Number of Faults Detected in MD-Latch (Total Faults = 129).

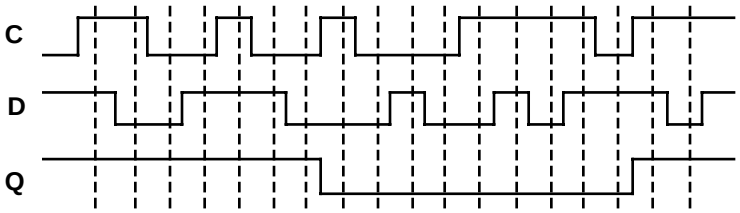
	Boolean and IDDQ	Boolean Alone (100 ns and 10 ms)	Boolean Alone (100 ns, 10 ms)	IDDQ Alone
Traditional Test	123	78	(78,74)	116
MD-Latch Checking Exp.	128	94	(92,87)	122
Scan MD-Latch Checking Exp.	128	92	(92,85)	122

5.3 Fault Simulation of D Flip-Flop

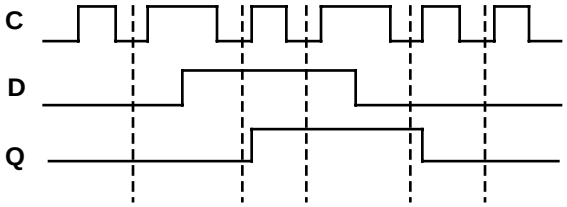
We simulated four different tests for the D flip-flop. The first test is a pin fault test set, which targets the faults on the input and output of the D flip-flop. The other three tests are: the checking experiment assuming no restrictions that was derived in Section 3.1.1, the data sequence transition tour test [Saxena 93], and the checking experiment for a shift register flip-flop (see section 3.2). The data sequence transition tour applies one of the four data transitions (0->0, 0->1, 1->1 and 1->0) in each cycle . The checking experiment for a shift register flip-flop (labeled as Shift Reg. D FF Checking Exp.) extends the data sequence transition tour by having three consecutive cycles with D = 0, and three consecutive cycles of D = 1. It includes the 1->0 and 0->1



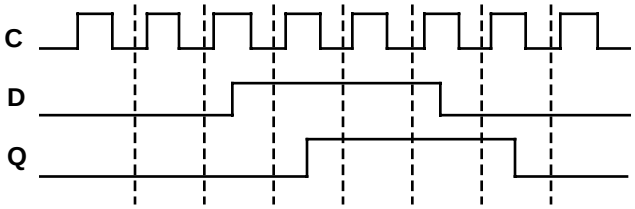
(a) Pin Fault Test for D Flip-Flop



(b) Checking Experiment for D Flip-Flop



(c) Data Sequence Transition Tour for D Flip-Flop



(d) Checking Experiment for Shift Register D Flip-Flop

Figure 5.3-1 Tests Applied to D Flip-Flop (Dashed Lines Indicate When Output is Checked).

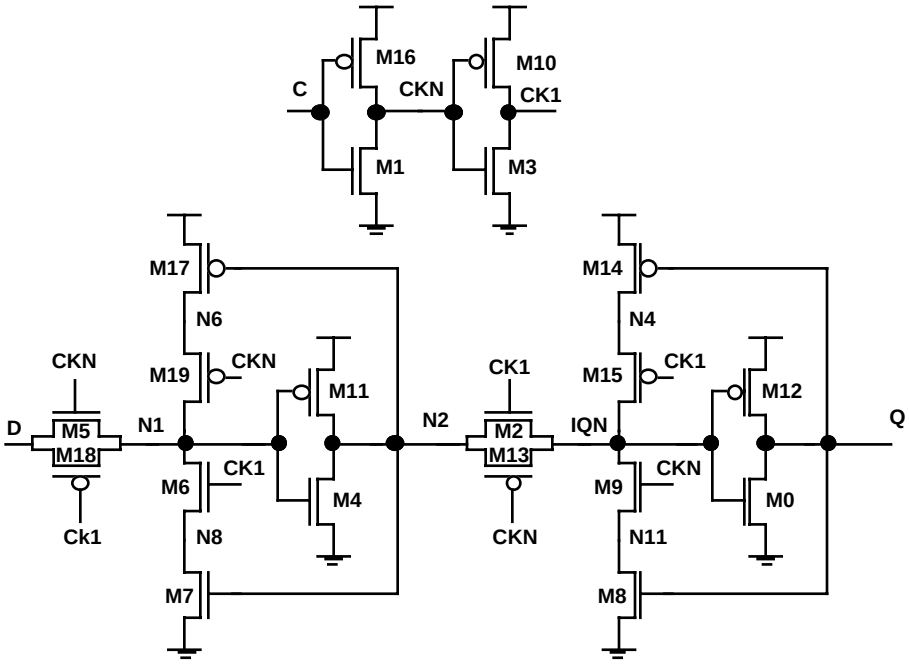


Figure 5.3-2 D Flip-Flop Implementation.

transitions. The four tests are shown in Fig. 5.3-1. In this figure, dashed lines indicate when the output is sampled. Except for the second test, the output is always sampled just before the rising transition of the C input. For the second test, the output is sampled after every input change. The flip-flop implementation used for the simulation is shown in Fig. 5.3-2.

Fig. 5.3-3 shows the distribution of maximum IDDQ values for the faulty circuits. The graph shows a sharp rise in IDDQf / IDDQg from 1 to 100,000. Such a sharp break in the graph makes for a good IDDQ threshold. Therefore, the IDDQ threshold selected

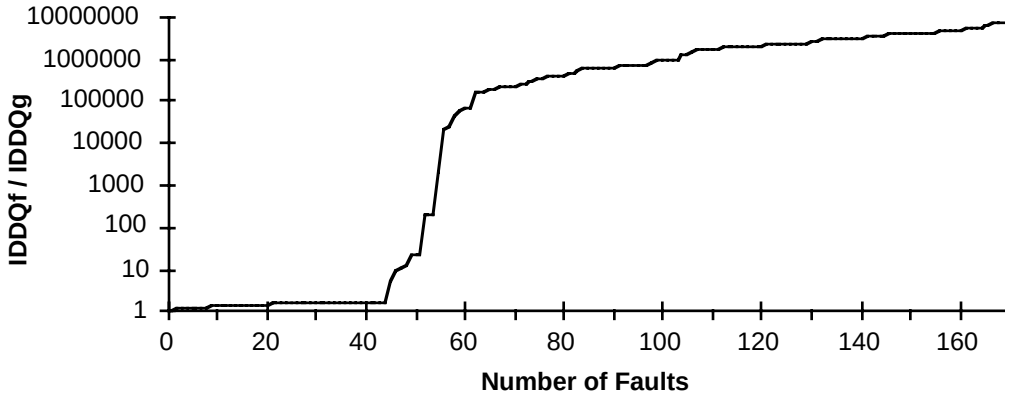


Figure 5.3-3 Current Distribution Graph for D Flip-Flop (IDDQg = 800 pA).

is 100,000 times the IDDQ current of the fault-free circuit. The maximum IDDQ value for the fault-free circuit was 800 pA, thus the threshold selected was 80 uA.

The results of the simulations are shown in Table 5.3-1. The table shows that the pin fault test misses several faults that are detected by the checking experiment. The functional faults missed by the pin fault test but detected by the checking experiment (10 of them) are shown in Fig. 5.3-4. If IDDQ measurement is also used, the faults missed (4 of them) are shown in Fig. 5.3-5. Faults that are missed by the checking experiment (15 of them) are shown in 5.3-6. In these figures white ovals indicate SOP or OPI faults, black ovals indicate SON faults, and thick black lines indicate STI faults. STP faults are shown as thick black lines connected to VDD. All STG faults are detected by all tests.

Table 5.3-1 Number of Faults Detected in D Flip-Flop (Total Faults = 170).

	Boolean and IDDQ	Boolean Alone (100ns and 10ms)	Boolean Alone (100 ns, 10 ms)	IDDQ Alone
Pin Fault Test	151	129	(109,127)	79
Checking Exp.	155	139	(118,138)	103
Data Sequence Transition Tour	155	139	(117,138)	100
Shift Reg. D FF Checking Exp.	155	139	(117,138)	100

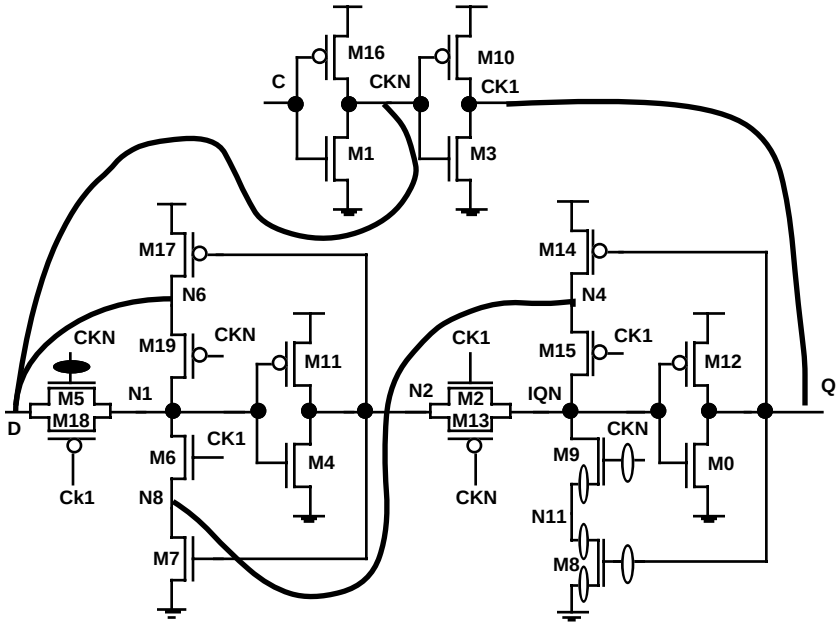


Figure 5.3-4 Faults Missed by Pin Fault (10 of them) Test but Detected by Checking Experiment for D Flip-Flop (Boolean Test Alone).

The faults missed by the checking experiment fall into two main groups. The first group of faults missed by the checking experiment is the stuck-open faults on the transmission gates. These faults, though undetectable, could add a delay to the circuit, and will thus behave as delay faults. A pattern that would detect a path delay fault to the input of the flip-flop may be able to detect these faults.

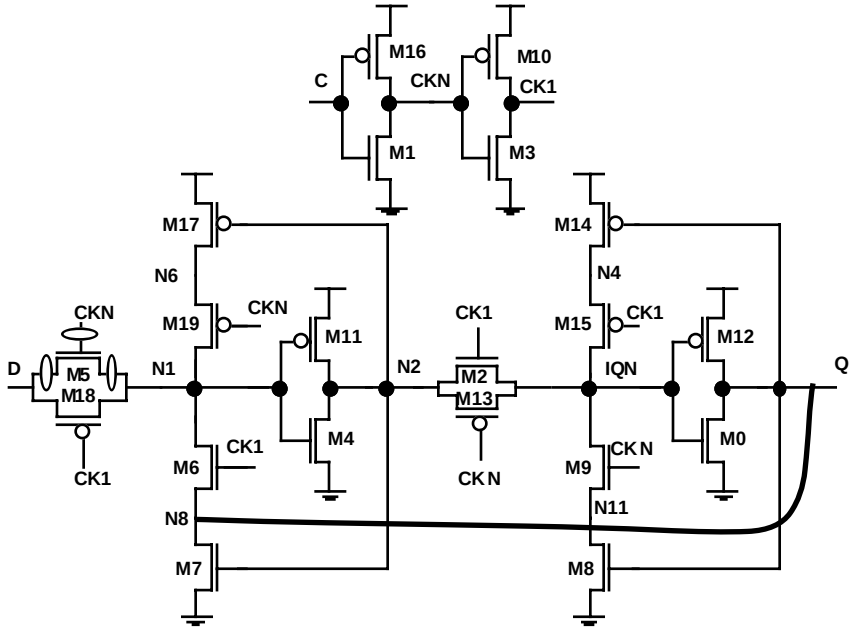


Figure 5.3-5 Faults Missed by Pin Fault (4 of them) Test Set but Detected by Checking Experiment for D Flip-Flop (Boolean and IDDQ Tests Used).

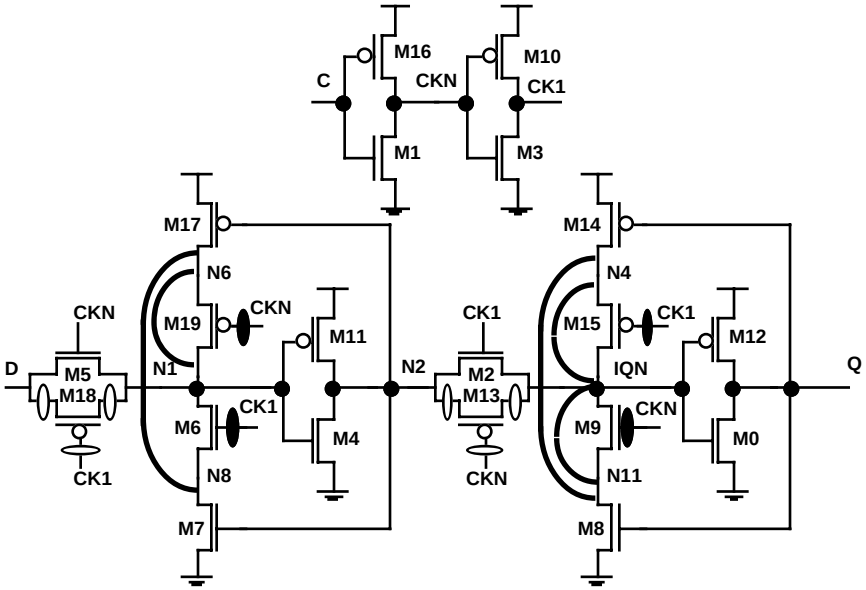


Figure 5.3-6 Faults Missed by Checking Experiment.

The other group of faults missed by the checking experiment, the stuck-ons and shorted-interconnects, will turn the master or slave latch into a dynamic latch. Since a dynamic latch cannot guarantee holding its value for a very long time, then loading a value and waiting a long time may change the value in the flip-flop and the fault would be detected. Thus a very slow test (data retention test) is needed for these faults. A more detailed analysis of these faults is given in Appendix C. This analysis shows that faults missed by the checking experiment are undetectable. In other words, the checking experiment detects all the detectable faults. The pin fault test misses many faults detected by the checking experiment.

5.4 Fault Simulation of MD Flip-Flop

Four different tests for the MD flip-flop were simulated using HSpice. The first test, a traditional test, is based on scanning in and out the 01100 pattern, and patterns that would detect stuck-at 0 and stuck-at 1 on the D input of the flip-flop. The second test is a pin fault test set, which targets stuck-at faults on the input and output of the MD flip-flop. The other two tests are the MD flip-flop checking experiment (section 3.1.2) and the checking experiment for the MD-latch in a scan chain (section 3.3.1). The flip-flop implementation used for the simulation is shown in Fig. 5.4-1.

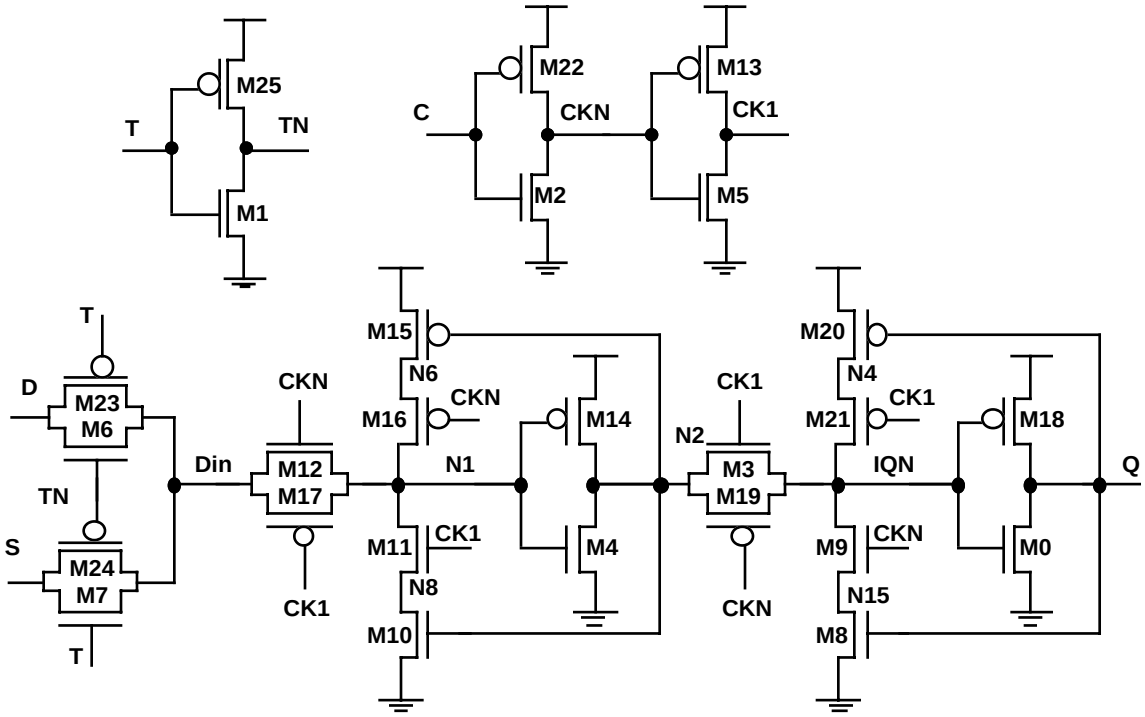


Figure 5.4-1 MD Flip-Flop Implementation Used in Simulation.

Fig. 5.4-2 shows the distribution of maximum IDDQ values for the faulty circuits. The graph shows a sharp rise in IDDQf / IDDQg from 1 to 100,000. Such a sharp break in the graph makes for a good IDDQ threshold. Therefore, the IDDQ threshold selected is 100,000 times the IDDQ current of the fault-free circuit. The maximum IDDQ value for the fault-free circuit was 1 nA, thus the threshold selected was 100 uA.

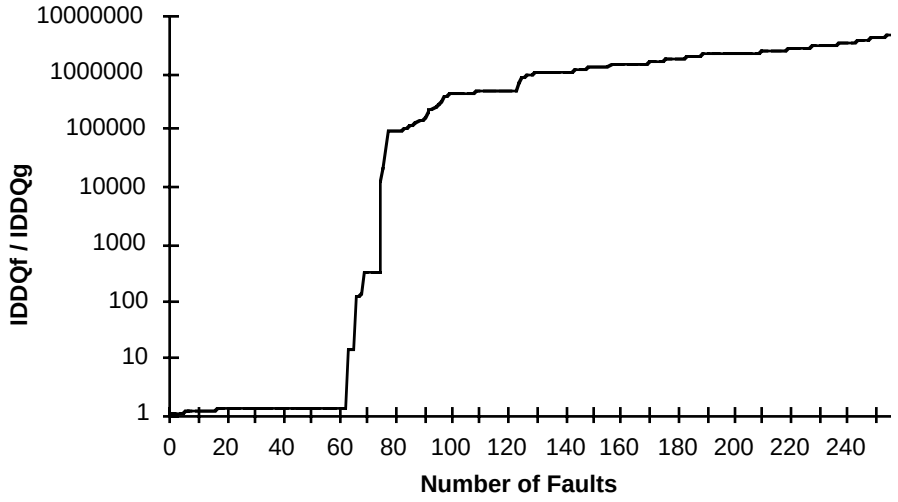


Figure 5.4-2 Current Distribution Graph for MD Flip-Flop(IDDQg = 1 nA).

The results of the simulations are shown in Table 5.4-1. From the table, there are 19 faults that were not detected by the checking experiment. These faults are shown graphically in Fig. 5.4-3. The table also shows that the pin fault test misses ten faults that are detected by the checking experiment. These faults are shown in Fig. 5.4-4. In these figures white ovals indicate stuck-open or open-interconnect faults, black ovals indicate SON faults, and thick black lines indicate shorted-interconnect faults. All short-to-power and short-to-ground faults are detected by all tests. The faults missed by the checking experiment fall into the same two groups as the faults missed by the checking experiment for the D flip-flop.

A detailed analysis of these faults is given in Appendix C. This analysis shows that faults missed by the checking experiment are undetectable. In other words, the checking experiment detects all the detectable faults. The traditional test and the pin fault tests miss many faults (about 10 %) detected by the checking experiment.

Table 5.4-1 Number of Faults Detected in MD Flip-Flop (Total Faults = 256).

	Boolean and IDDQ	Boolean Alone (100 ns and 10 ms)	Boolean Alone (100 ns, 10 ms)	IDDQ Alone
Traditional Test	212	167	(145,166)	155
Pin Fault Test	227	184	(162,183)	161
MD Flip-Flop Checking Exp.	237	207	(186,204)	182
Scan MD Flip-Flop Checking Exp.	237	206	(184,204)	181

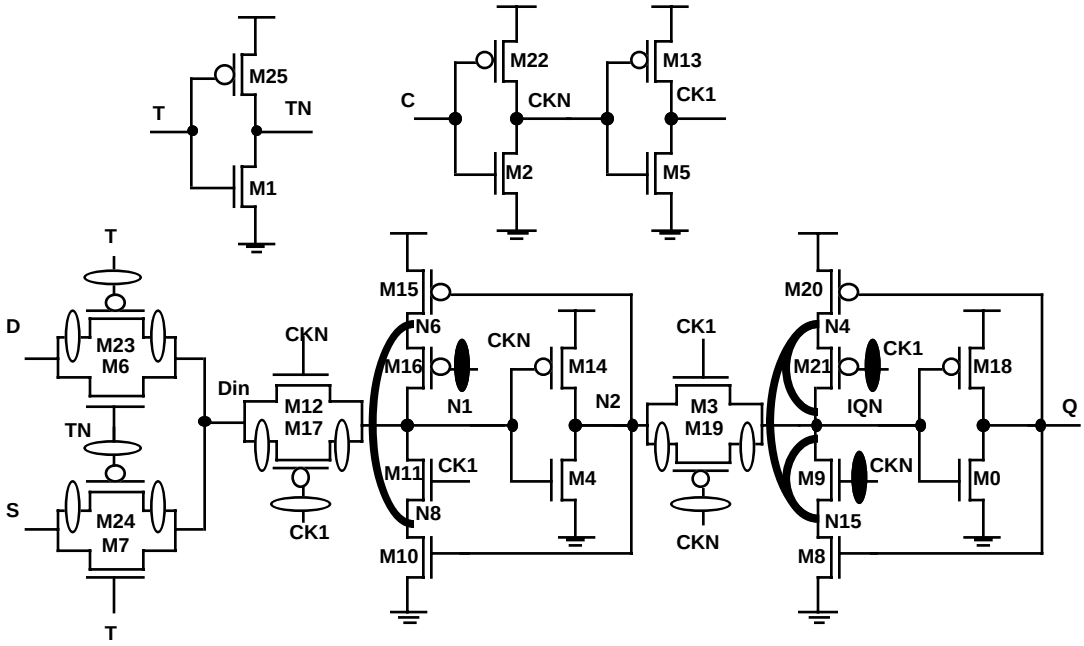


Figure 5.4-3 Faults Missed by Checking Experiment of MD Flip-Flop (19 of them).

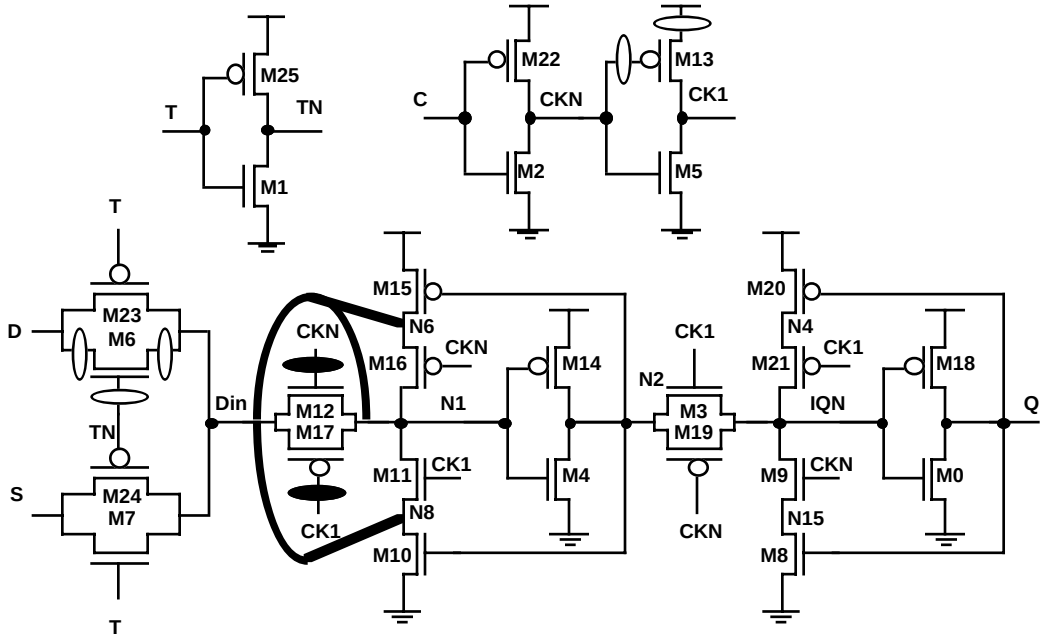


Figure 5.4-4 Faults Missed by Pin Fault Test Detected by Checking Experiment of MD Flip-Flop (10 of them).

5.5 Summary

In this chapter we presented fault simulation results for four different bistable elements, in which we compared checking experiments against traditional and pin fault tests. In all cases, we showed that our checking experiments detected all detectable faults in the circuits, while the traditional tests missed many faults. For example, the traditional test for the MD flip-flop missed about 10 percent of the faults detected by the checking experiment. The traditional test for the D-latch missed as many as 20 percent of the faults detected by the checking experiment. These results show the effectiveness of checking experiments in detecting defects in bistable elements.

Chapter 6. ATPG Results

In Chapter 4, we presented ATPG algorithms for generating patterns to test memory elements in scan designs. In that chapter, we described an implementation based on modifying SIS [Sentovich et. al. 92]. We present some results from running this program. First, we present results for a small circuit, a three-bit binary counter. A binary counter is chosen because it is a commonly used circuit. A three-bit version is picked so that the circuit is general enough, while at the same time small enough to allow for HSpice fault simulation. As in Chapter 5, HSpice is used to perform fault simulation using CrossCheck models, to compare the effectiveness of different tests. Even though it would be difficult to fault simulate the ISCAS-89 benchmark circuits [Brglez et. al. 89] using HSpice, we generate patterns for them, and compare their size with the size of stuck-at test patterns. Results indicate that the size of our test increases with circuit size at a similar rate as the stuck-at test.

Two tests were generated for the three-bit counter shown in Fig. 6-1. The first test, a traditional test, consists of five patterns: four to give 100 percent stuck-at coverage for the combinational logic, and the 01100 pattern for testing the flip-flops. The stuck-at patterns are shown in Table 6-1. The second test was generated by our ATPG program for MD flip-flops described in Chapter 4. The test consists of 82 patterns. Table 6-2 shows a summary of this test. Each MD flip-flop requires 80 sub-sequences to form a checking experiment. Since there are 3 flip-flops, we have 240 sub-sequences. Of these,

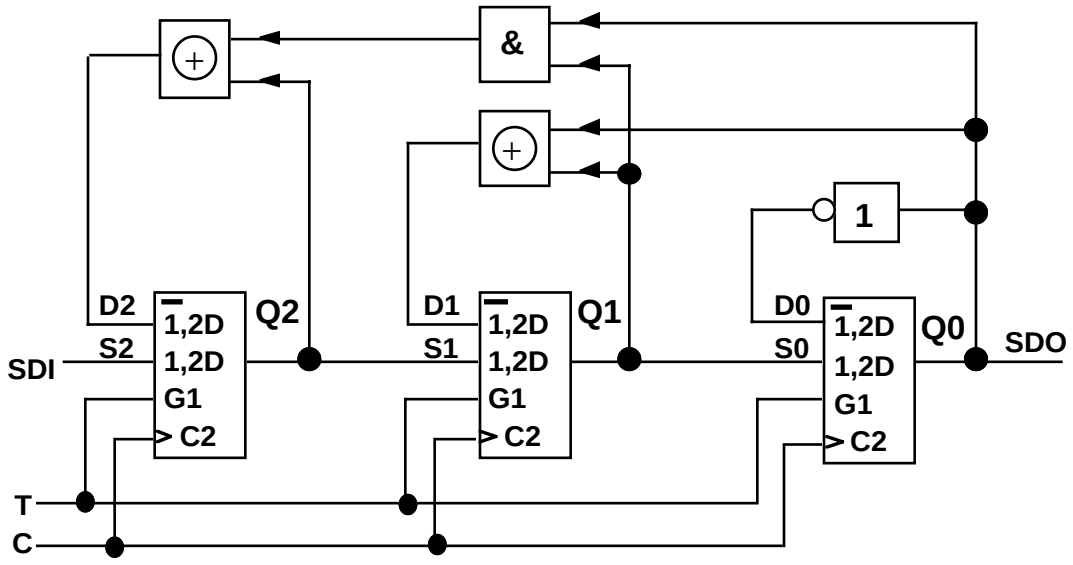


Figure 6-1 Circuit for a Three-Bit Binary Counter.

the program generated patterns for 133 of them. For the other 107, the program proved that no pattern could be generated, due to the circuit structure. For example, looking at the Q0 flip-flop, D0 will always be the opposite value of Q0, which implies that D0 cannot hold the same value for two cycles, a requirement for many of the sub-sequences needed to form a checking experiment for flip-flop Q0. The 133 patterns were compacted to the 82 patterns in Table 6-3. The size of our test is a lot larger than the stuck-at test. This is mainly due to the small size of combinational logic in the circuit. Later in this chapter, we show that the test size difference decreases with larger circuits.

As with the tester, patterns for HSpice must be scanned in before applying them to the internal logic. Each pattern for the combinational logic requires three cycles to be scanned in, one cycle to capture the output of the combinational logic, and three cycles to scan out the flip-flop values. Since a new pattern can be scanned in while the flip-flop values are scanned out, each pattern requires 4 cycles (three for scanning the pattern and one for capturing the output of the combinational logic). The waveforms for applying one of the patterns is shown in Fig. 6-2. On the other hand, patterns from our ATPG tool require four or five cycles. The 82 patterns of our test are split into 6 groups, as described in Section 4.2.3, in Table 6-3. Each group corresponds to one of the waveforms in Fig. 6-3. Three cycles are needed to scan in the pattern and then one or two more cycles are needed depending on the waveform used. For example, if a pattern corresponds to waveform A0, then we need to scan in the pattern (which requires 3 cycles), set T = 0, apply two cycles (first cycle is from t_0 to t_1 and the second is from t_1 to t_2), set T = 1 to scan out the values of the flip-flops. A waveform can be formed for each pattern in Table 6-1 and Table 6-3. These waveforms (in HSpice format) are used as the stimulus for the HSpice simulation.

Table 6-1 Stuck-At Patterns for Three-Bit Binary Counter.

Pattern No.	$Q_2Q_1Q_0$
1	1 1 1
2	0 1 1
3	1 0 1
4	1 1 0

Table 6-2 Results of ATPG for Three-Bit Binary Counter.

Number of Sub-sequences	240
Number of Sub-sequences With No Pattern	107
Number of Aborted Sub-sequences	0
Number of Sub-sequences With Pattern	133
Number of Compacted Patterns	83

Table 6-3 Patterns From MDFF ATPG for Three-Bit Binary Counter.

A0	A1	B0	B1	C0	C1
$Q_2Q_1Q_0S_0S_1$	$Q_2Q_1Q_0S_0S_1$	$Q_2Q_1Q_0S_0S_1$	$Q_2Q_1Q_0S_0S_1$	$Q_2Q_1Q_0S_0S_1$	$Q_2Q_1Q_0S_0S_1$
0 0 0 0 0	0 0 0 1 0	0 0 0 - 0	0 0 0 - 1	0 0 0 - -	0 0 0 1 0
0 0 0 1 1	0 0 0 - 1	0 0 1 - 1	0 0 1 - 0	0 0 1 - -	0 0 0 - 1
0 0 1 - -	0 0 1 0 0	0 1 0 - -	0 1 0 - -	0 1 0 0 0	0 0 - 0 0
0 1 0 0 0	0 0 1 1 1	0 1 1 - 0	0 1 1 - 0	0 1 0 1 1	0 1 0 - 0
0 1 0 1 1	0 1 0 - 0	0 1 1 - 1	0 1 1 - 1	0 1 1 0 0	0 1 0 - 1
0 1 1 0 0	0 1 0 - 1	1 0 0 - 0	1 0 0 - 0	0 1 1 1 1	0 1 1 0 1
0 1 1 0 1	0 1 1 0 0	1 0 1 - 1	1 0 1 - 1	1 0 0 - -	0 1 1 1 1
0 1 1 1 0	0 1 1 0 1	1 1 0 - -	1 1 0 - -	1 0 1 - -	1 0 0 - 1
0 1 1 1 1	0 1 1 1 1	1 1 1 - 0	1 1 1 - 0	1 1 0 0 0	1 0 1 - 0
1 0 0 0 0	1 0 0 1 1	1 1 1 - 1	1 1 1 - 1	1 1 0 1 1	1 0 1 - 1
1 0 0 1 1	1 0 1 - 0			1 1 1 0 0	1 1 0 0 0
1 0 1 - -	1 0 1 - 1			1 1 1 1 1	1 1 0 1 0
1 1 0 0 0	1 1 0 0 0				1 1 1 0 1
1 1 0 1 1	1 1 0 1 0				1 1 1 1 1
1 1 1 0 0	1 1 0 1 1				1 1 1 - 0
1 1 1 0 1	1 1 1 0 0				
1 1 1 1 0	1 1 1 0 1				
1 1 1 1 1	1 1 1 1 1				

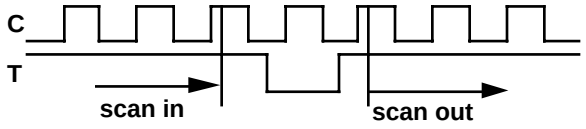
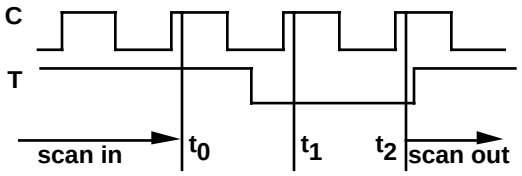
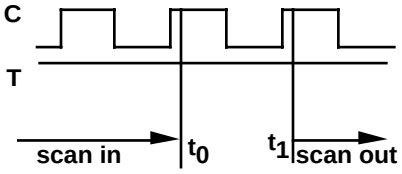


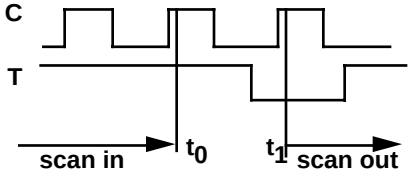
Figure 6-2 Waveforms for Applying Traditional Scan Patterns.



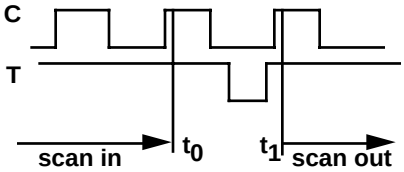
(a) Waveform A0



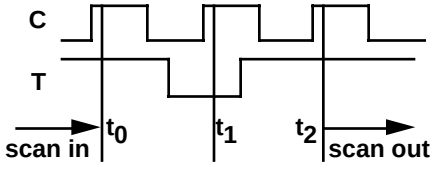
(b) Waveform A1



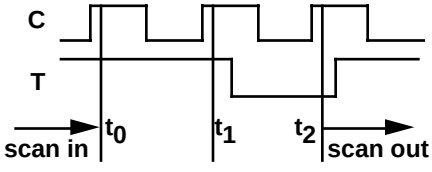
(c) Waveform B0



(d) Waveform B1



(e) Waveform C0



(f) Waveform C1

Figure 6-3 Waveforms for Applying Checking Experiment Test Patterns.

Both test sets were simulated in HSpice using the fault models described in Chapter 5. The results are summarized in Table 6-4. The checking experiment detected 45 faults that were not detected by the stuck-at test set. This small example shows that a 100 percent stuck-at test (i.e., a test generated to detect all stuck-at faults in the circuit) can leave many real flip-flop faults undetected.

Table 6-4 shows that the checking experiment missed a large number of faults. The large number of undetected faults is a direct result of the large number (107) of sub-sequences for which no pattern can be generated. Another reason for the large number of undetected faults is that we only simulated the 100 ns case because of the size of the circuit. Therefore, we could not make IDDQ measurements. As shown in Chapter 5, IDDQ measurement detects many faults not detected by boolean testing.

Table 6-4 Summary of Simulation Results for Three-Bit Binary Counter.

Faults Injected	651
Faults Not Detected by Traditional Method	393
Faults Not Detected by Checking Experiment	348
Faults Detected by Checking Exp. Missed by Traditional	45

One practical concern with testing chips is the size of the test being applied. To address this issue, we generated patterns for the ISCAS 89 benchmark circuits for all four architectures, and compared them to the stuck-at test lengths. Table 6-5 shows the number of vectors for all the ISCAS 89 circuits for each architecture, and for the stuck-at tests. The name of the ISCAS 89 circuits indicates the number of lines in the circuit. This is directly related to the size of the circuit. The number of patterns for the LSSD architecture is always the smallest of our tests, and the number of patterns for the MD flip-flop architecture is always the largest.

To compare our test size with the test size of the stuck-at test, we calculate the ratio of the size of our tests to the size of the stuck-at tests. These ratios are shown in Table 6-6. Since the ratios do not increase considerably with circuit size, we conclude that the size of our test will not be a problem with large circuits.

In this chapter we showed that traditional tests can miss faults inside MD flip-flops of a small circuit. These faults may affect normal operation of the circuit. Our test detects many of these faults. However, it did miss many faults, but that was due to the circuit constraints. We also generated patterns for larger circuits. This showed that our algorithm generates tests that can be practically used for testing real circuits.

Table 6-5 Number of Patterns for Different Tests.

Circuit	MD-Latch	LSSD	MD Flip-Flop	TP Flip-Flop	Stuck-At
S27	29	19	118	52	14
S298	71	47	356	162	66
S344	97	62	395	165	65
S349	91	61	375	158	66
S382	135	85	578	244	74
S386	49	31	282	114	88
S400	133	85	576	242	71
S444	97	65	487	212	80
S510	65	43	306	149	78
S526	106	68	572	257	139
S641	187	128	578	257	124
S713	123	84	572	255	133
S820	54	37	346	144	161
S832	54	37	349	148	171
S1196	126	80	440	233	201
S1423	341	216	1640	756	218
S1488	68	45	421	179	247
S1494	68	45	422	180	243
S5378	571	336	2139	1023	700

Table 6-6 Number of Patterns Divided by Stuck-At Test Length.

Circuit	MD-Latch	LSSD	MD Flip-Flop	TP Flip-Flop	Stuck-At
S27	2.07	1.36	8.43	3.71	1.00
S298	1.08	0.71	5.39	2.45	1.00
S344	1.49	0.95	6.08	2.54	1.00
S349	1.38	0.92	5.68	2.39	1.00
S382	1.82	1.15	7.81	3.30	1.00
S386	0.56	0.35	3.20	1.30	1.00
S400	1.87	1.20	8.11	3.41	1.00
S444	1.21	0.81	6.09	2.65	1.00
S510	0.83	0.55	3.92	1.91	1.00
S526	0.76	0.49	4.12	1.85	1.00
S641	1.51	1.03	4.66	2.07	1.00
S713	0.92	0.63	4.30	1.92	1.00
S820	0.34	0.23	2.15	0.89	1.00
S832	0.32	0.22	2.04	0.87	1.00
S1196	0.63	0.40	2.19	1.16	1.00
S1423	1.56	0.99	7.52	3.47	1.00
S1488	0.28	0.18	1.70	0.72	1.00
S1494	0.28	0.19	1.74	0.74	1.00
S5378	0.82	0.48	3.06	1.46	1.00

Chapter 7. Concluding Remarks

A new approach for testing memory elements in digital circuits has been described in this thesis. Traditional approaches for testing memory elements in a scan chain involve shifting in a sequence of zeroes and ones. We showed, in Chapters 5 and 6, that this approach misses many faults in the circuit. These faults may affect normal circuit operation. Our new approach is based on checking experiments for the memory elements. Checking experiments are used because they guarantee the detection of all faults that do not increase the number of states. Since a checking experiment makes no assumption about the circuit implementation, it is implementation independent. This is especially useful since designers often use different implementations of memory elements to optimize their circuits for area and performance. Analysis of faults inside memory elements [Al-Assadi 93] has shown that some faults inside the memory elements cannot be mapped to functional fault models. This implies that transistor level test generation is needed to target the specific faults. However, the checking experiment will detect these faults without special analysis since it detects all the faults inside the memory element.

Our approach "breaks up" the checking experiment into a set of small sub-sequences. For each of these sub-sequences a test pattern is generated. By automating the process of generating the patterns, a designer or test engineer need only provide a gate level description of the circuit to generate tests that together guarantee a checking experiment for each memory element in the design. These test patterns are scanned in like any normal pattern, appropriate changes to the control inputs of the memory elements are applied, and the results are scanned out.

Our implementation of the test generator was based on modifying an existing stuck-at test tool. This was done to illustrate that current tools can be easily adapted to include tests for memory elements. The implementation was done in a hierarchical fashion, where elementary functions were used to implement common features needed for the different architectures. This not only made it easier to debug problems, but also makes it easier to add tests for new memory element types in the future. The implementation currently includes test generators for the four different implementations discussed in this thesis.

Our test was compared with the traditional test by performing fault simulation of some of the memory elements. The results clearly indicate that there are faults that traditional tests miss that are detected by our new test. Though the size of our test was considerably larger than that of the stuck-at test for the small binary counter, we showed

that the test size increases with circuit size by about the same rate as the test for stuck-at faults. In conclusion, tests based on checking experiments for latches and flip-flops are a thorough economic technique for testing the memory elements of digital circuits.

Appendix A. Using Checking Experiments To Test Two-State Latches

This appendix contains revised text of Makar, S.R. and E.J. McCluskey, "Using Checking Experiments to Test Two-State Latches," CRC Technical Report 94-11.

Center for Reliable Computing

TECHNICAL REPORT

Using Checking Experiments To Test Two-State Latches

Samy R. Makar and Edward J. McCluskey

94-11R

(CSL TR # 94-641)

May 1996

Center for Reliable Computing
ERL 460
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract:

A general technique to determine conditions for exhaustive functional testing (checking experiment) of two-state latches is derived. This technique is used to derive conditions for checking experiments of various two-state latches. Minimum-length checking experiments are also presented. One of the checking experiments for the D-latch is simulated using an HSpice implementation of the transmission gate latch. All detectable shorted interconnects, open interconnects, short-to-power, short-to-ground, stuck-open, and stuck-on faults are detected. A pin fault test set and a multiplexer-based test set are also simulated. These tests miss some faults detected by the checking experiment.

Funding:

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant No. N00014-92-J-1782, in part by the Advanced Research Projects Agency under Contract No. DABT63-94-C-0045, and in part by the National Science Foundation under Grant No. MIP-9107760. It was also funded in part by Cirrus Logic.

Imprimatur: Erin Kan and Jonathan Chang

1. Introduction

Tests for stuck-at faults at latch inputs and outputs miss many internal faults [Reddy 86], [Lee 90] and [Al-Assadi 93]. Reddy, in [Reddy 86], derived tests for stuck-open faults in different latch implementations. Lee, in [Lee 90], analyzed bridging faults in scan registers, and combined the use of current and voltage tests. Al-Assadi, in [Al-Assadi 93], mapped many, but not all, of the internal faults to functional fault models. He also showed that some of the internal faults cannot be mapped to functional fault models. What is needed is a test set that detects all faults (both internal and I/O faults). This paper presents such test sets.

A *checking experiment* is an input-output sequence that distinguishes a given state machine from all other state machines with the same inputs and outputs, and the same number of or fewer states. Checking experiments were first defined by Hennie as follows, "Any circuit that responds to the *checking experiment* in accordance with a given state table and starting state either must be operating correctly or else must have suffered a malfunction not in the given class." [Hennie 64]. The important property of a checking experiment is that it contains enough information to derive the flow table.

Even though Hennie's work was for pulse-mode circuits (often called synchronous sequential circuits), checking experiments can be derived for fundamental-mode circuits (often called asynchronous circuits) by modifying the procedure. Friedman, in [Friedman 71], discusses the restrictions in fundamental-mode circuits. Since there is no inherent clock, the machine can change state after any input changes, and the same input cannot be repeated. Also, for deterministic behavior, only one input can be changed at a time. Some faults can cause critical races, making the behavior non-deterministic, and can thus not be guaranteed to be detected. We did not encounter such faults in our simulations.

In Section 2, we present various latches and minimum-length checking experiments for them. In Section 3, we present a general technique for deriving checking experiments for two-state latches. Separate sub-sections are devoted to the derivation of checking experiment requirements and minimum-length checking experiments for each latch type, with details given in the appendices. The checking experiment for the D-latch was simulated in HSpice using a transmission gate implementation. The results of this simulation are compared with those of a pin fault test set and a multiplexer-based test set in Section 4.

2. Latches and Their Minimum-Length Checking Experiments

Various latches are discussed in this paper (see Table 2-1). The simplest latch type is the SR-latch. An *SR-latch* is a sequential element that can be set or reset by activating the appropriate input. Even though the SR-latch is still occasionally used, the most commonly used latch today is the D-latch. A *D-latch* is a sequential element, in which the data input is propagated to the output when the clock is active, otherwise it holds the stored value. A *D-latch with Asynchronous Set/Reset* is a D-latch that can be set or reset when the clock is not active. Scan-paths require latches with two different data sources. These can be either Multiplexed-Data latches or Two-Port latches. A *Multiplexed-Data latch* (*MD-latch*) is a D-latch with multiplexed data inputs; a *Two-Port latch* has two control inputs with the data source determined by the active control input [McCluskey 86]. A *Load Enable latch* is a D-latch with a gated control input, and a *D-Enable latch* is a D-latch with gated data. An *XOR Input latch* performs an exclusive-or operation on its two data inputs. This latch is commonly used in an LFSR to generate pseudo-random vectors, and to compress results. Other latches commonly used for BIST are the *Built-In Logic Block Observer latch (BILBO latch),* and the *Concurrent Built-In Logic Block Observer latches (CBILBO latches).* The BILBO latch has two data inputs. It can be configured to load either of the two inputs (one a scan input, and the other for normal

Table 2-1 Latches and Their Excitation Functions.

Latch Type	Excitation Function	Assumptions	M*
SR-Latch	$Q = S + \overline{R}q$	SR = 0	6
D-Latch	$Q = CD + \overline{C}q$		7
D-Latch with Asynchronous Set/Reset	$Q = \overline{R}\,(S + CD + \overline{C}q)$	SR = 0	14
MD-Latch	$Q = C\,(TS + \overline{T}D) + \overline{C}\,q$		26
Two-Port Latch	$Q = C_1D_1 + C_2D_2 + \overline{C}_1\overline{C}_2q$	$C_1C_2 = 0$	23
Load Enable Latch	$Q = CLD + \overline{(LC)}q$		15
D-Enable Latch	$Q = CDE + \overline{C}q$		16
XOR Input Latch	$Q = C\,(D \oplus S) + \overline{C}q$		13
BILBO Latch	$Q = C\,(B_1D \oplus \overline{B}_2S) + \overline{C}\,q$		58
CBILBO Latch	$Q_1 = C\,(\overline{B}_1D \oplus S) + \overline{C}\,q_1$ $Q_2 = C\,(B_2S + \overline{B}_2D) + \overline{C}\,q_2$		25 26

*M - minimum length (number of test vectors) of checking experiment.

operation), load the exclusive-or of the two inputs (for signature analysis), or load 0. The CBILBO latches, an extension of the BILBO latch, are two latches that can operate simultaneously as a pseudo-random pattern generator and a signature analyzer. The two latches are treated separately, with outputs Q_1 and Q_2. Table 2-1 shows the excitation function of each of these latches and the minimum-length of a checking experiment. Minimum-length checking experiments for each of these latches are shown in Tables 2-2 through 2-11. Details for each latch type are presented in Section 3.

Table 2-2 A Minimum-Length (6) Checking Experiment for a SR-Latch.

S	0	0	1	0	0	0
R	1	0	0	0	1	0
Q	0	0	1	1	0	0

Table 2-3 A Minimum-Length (7) Checking Experiment for a D-Latch.

C	1	1	0	0	1	0	0
D	1	0	0	1	1	1	0
Q	1	0	0	0	1	1	1

Table 2-4 A Minimum-Length (14) Checking Experiment for an Asynchronous Set/Reset Latch.

C	1	1	0	0	0	0	0	0	0	0	0	0	1	0
D	1	0	0	0	0	0	0	1	1	1	1	1	1	1
R	0	0	0	0	0	1	0	0	0	0	1	0	0	0
S	0	0	0	1	0	0	0	0	1	0	0	0	0	0
Q	1	0	0	1	1	0	0	0	1	1	0	0	1	1

Table 2-5 A Minimum-Length (26) Checking Experiment for an MD-Latch.

D	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
S	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0
T	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
C	1	1	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0
Q	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	1	0	0

Table 2-6 A Minimum Length (23) Checking Experiment for a Two-Port Latch.

D_1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1
C_1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0
D_2	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
C_2	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
Q	1	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1

Table 2-7 A Minimum-Length (15) Checking Experiment for a Load Enable Latch.

L	1	1	0	0	1	1	0	0	1	0	0	1	1	0	0
D	1	0	0	0	0	1	1	1	1	1	1	1	0	0	0
C	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1
Q	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1

Table 2-8 A Minimum-Length (16) Checking Experiment for a D-Enable Latch.

D	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0
E	1	1	1	1	1	0	0	1	1	1	0	0	1	0	0	0
C	1	1	0	0	1	1	0	0	1	0	0	0	0	0	1	0
Q	1	0	0	0	1	0	0	0	1	1	1	1	1	1	0	0

Table 2-9 A Minimum-Length (13) Checking Experiment for an XOR Input Latch.

D	0	0	0	0	0	0	1	1	1	1	1	1	0
S	1	0	0	1	1	1	1	1	1	0	0	0	0
C	1	1	0	0	1	0	0	1	0	0	1	0	0
Q	1	0	0	0	1	1	1	0	0	0	1	1	1

Table 2-10 A Minimum-Length (58) Checking Experiment for a BILBO Latch.

D	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1
S	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
B_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
B_1	0	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
C	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0
Q	1	0	0	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1
D	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0
S	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1
B_2	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
B_1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
C	0	1	0	0	1	0	0	1	0	0	1	1	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0
Q	1	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	1	1	1	1	0	0

Table 2-11a A Minimum-Length (25) Checking Experiment for a CBILBO Latch 1.

D	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0
S	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0
B_1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
C	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0
Q_1	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	1	0	0

Table 2-11b A Minimum-Length (26) Checking Experiment for a CBILBO Latch 2.

D	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
S	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0
B_2	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
C	1	1	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0
Q_2	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	1	0	0

3. Deriving Checking Experiments for Two-State Latches

All latches described in Table 2-1 are typically implemented as two-state latches; the flow tables of these latches have only two rows. In this section, special properties of two-state flow tables are analyzed and a method for generating checking experiments for them is developed. This method was used to generate the checking experiments in Tables 2-2 through 2-11. The details are presented in sub-sections of this section.

Each cell in a flow table, a *total state*, corresponds to an assignment of values to the circuit inputs and internal states. A total state is an *unstable state* if it causes a change in internal state of the machine. A total state is a *stable state* if the next internal state is the same as the current internal state. The notation used in this paper for stable total states is shown in Table 3-1. A total state is *unspecified* if it is not adjacent to a stable total state. Such states cannot be reached because of the single-input change restriction on fundamental mode circuits [McCluskey 86]. Unspecified states are shown with "–" in the flow tables. A sequence *visits* a total state when the sequence applies the input of the total state while the machine is in the internal state of the total state. A total state is *identified* by a sequence if the sequence provides enough information to reconstruct the corresponding entry in the flow table.

Table 3-1 Notation: Stable Total States.

Notation	Definition
(2)	Output = 0
[3]	Output = 1

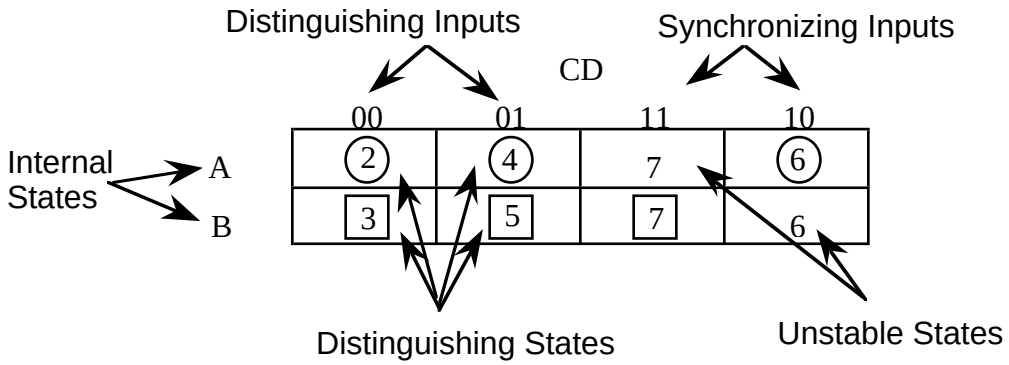


Figure 3-1 Definitions in Flow Table.

For a state machine to have sequential behavior, there must be at least one column in the flow table with different outputs. Otherwise, the machine acts as pure combinational logic. The total states in a column of a two-state flow table with differing outputs are called *distinguishing states*, and the input is called a *distinguishing input*. A state machine must also have two columns that change the internal state, so that both internal states are reachable. Inputs of such columns are called *synchronizing inputs* because they force the machine into a known state. These definitions are shown in Fig. 3-1.

As mentioned earlier, a checking experiment contains enough information to reconstruct the flow table. Therefore, it must identify all total states in the flow table. In this paper, we analyze flow tables that have distinguishing inputs and synchronizing inputs only (i.e. no

columns in the table have two stable states with the same output), because all latches studied here have flow tables that fall into this class of flow tables. The requirement for reconstructing the flow table can be refined into three simpler requirements (proof of this will come later): all total states must be visited, all unstable states must be identified, and all distinguishing states must be identified.

We start with the first requirement, all total states must be visited. If a total state is not visited by the sequence, then the effect of applying the input of the total state when the machine is in the internal state of the total state is not known. This requirement is proven in Lemma 1.

Lemma 1: A checking experiment for a two-state flow table with only distinguishing and synchronizing inputs must visit all total states in the flow table.

Proof: Suppose that a sequence does not visit one of the stable total states. Create a second flow table by copying the original flow table and changing the output of the state not visited by the sequence. The output response of the sequence when applied to the second flow table would be the same as that of the original one because the sequence never visits the only total state that differs in the two flow tables. Since the two flow tables give the same response to the same input sequence, the sequence cannot be a checking experiment. Now suppose that the sequence does not visit one of the unstable states. In this case, the sequence would have the same response for a flow table that had the unstable total state replaced by a stable total state (the output does not matter). Since the input sequence has the same response for two flow tables, it cannot be a checking experiment. Therefore, a checking experiment must visit all total states (stable and unstable) in the flow table.

Even though visiting all the total states is a necessary requirement for a checking experiment, it is not a sufficient one. Consider the flow tables in Table 3-2. The sequence in Table 3-2c visits all the states of the two flow tables, Tables 3-2a and 3-2b, but the output response is the same for the two flow tables. Since the two flow tables are different, the sequence is not a checking experiment. Assuming that the flow table in Table 3-2a is the desired flow table, the sequence did not identify the unstable states 8 and 9. An unstable state is identified, when the sequence shows that the input caused a change in internal state. To show that an input causes a change in internal state, we need to show that the total states before and after the application of the input have different internal states. The internal state after the application of the input can be identified by following the input with a distinguishing input. The total state would then be a distinguishing state. The total state before the application of the input does not have to be a distinguishing state, but its internal state must be known. This analysis

suggests the creation of state triples. A *state triple* is a set of three total states that contains the *setup state* (same internal state as unstable state and input unit distance from unstable state input), the total state associated with the unstable state, and a distinguishing state. The setup state and the distinguishing state must have different internal states. The last state of the triple is a distinguishing state. Visiting a distinguishing state does not change the internal state of the machine.

Table 3-2a Two-State Flow Table.

	CD							
	00	01	11	10	00	01	11	10
	A=0				A=1			
	(2),0	(4),0	7	(6),0	9	(8),0	–	–
	[3],1	[5],1	[7],1	6	[9],1	8	–	–

Table 3-2b Another Flow Table That Produces Same Output Sequence When Table 3-2c Sequence Is Applied.

	CD							
	00	01	11	10	00	01	11	10
	A=0				A=1			
	(2),0	(4),0	7	(6),0	[9],1	(8),0	–	–
	[3],1	[5],1	[7],1	6	[11],1	(10),0	–	–

Table 3-2c Sequence That Visits All Total States, But Is Not A Checking Experiment.

A	0	0	0	0	0	0	1	1
C	1	0	0	1	0	0	0	0
D	1	1	0	0	0	1	1	0
Q	1	1	1	0	0	0	0	1
State	7	5	3	6	2	4	8	9

Lemma 2: A checking experiment for a two-state flow table with only distinguishing and synchronizing inputs must visit at least one state triple of each unstable state to identify all unstable states. Triples can overlap, the distinguishing state of one triple can be the setup state of another triple.

Proof: We have already shown that a checking experiment must visit all the total states, including the unstable states. Visiting an unstable state requires visiting the setup state of a triple of the unstable state before visiting the unstable state itself. Therefore, a sequence that visits an unstable state u but does not identify it is not visiting a distinguishing state after visiting the unstable state (i.e., it only

visits the first two states of the triple). In this case, the input applied by the sequence after visiting the unstable state *u* is a synchronizing input. Create a second flow table by copying the original flow table and changing the unstable state *u* to a stable total state and give it the same output of the other total state in the same column. When the input corresponding to our unstable state is applied to either flow table, we get the same output. Since the next input is a synchronizing input, the next output and internal state will be the same for both flow tables. Therefore, a checking experiment must visit at least one state triple of each unstable state to identify all the unstable states.

Consider the flow tables in Table 3-3. Graphs for the state triples of Table 3-3a are shown in Fig. 3-2. The sequence shown in Table 3-3c visits all the total states, and identifies all the unstable states. However, both flow tables in Tables 3-3a and 3-3b, produce the same output response for the input sequence of Table 3-3c.

The sequence in Table 3-3c produces different outputs for both 00 and 01, indicating that they are distinguishing inputs. However, there are two possible permutations (barring isomorphism) for the distinguishing states in the flow table. These are shown in the first two columns of Tables 3-3a and 3-3b. To distinguish between the two flow tables, a sequence must have CD = 00,01 or CD = 01,00 as sub-sequences.

Table 3-3a Two-State Flow Table.

	CD						
00	01	11	10	00	01	11	10
A=0				A=1			
(2),0	(4),0	7	(6),0	9	(8),0	–	–
[3],1	[5],1	[7],1	6	[9],1	8	–	–

Table 3-3b Another Flow Table That Produces the Same Output Sequence When Table 3-3c Sequence Is Applied.

	CD						
00	01	11	10	00	01	11	10
A=0				A=1			
(2),0	[5],1	[7],1	(6),0	9	8	–	–
[3],1	(4),0	7	6	[9],1	(8),0	–	–

Table 3-3c Sequence That Visits All Total States, Identifies Unstable States, But Is Not A Checking Experiment.

A	0	0	1	0	0	0	0	0	0	1	0
C	1	0	0	0	1	0	1	1	0	0	0
D	1	1	1	1	1	1	1	0	0	0	0
Q	1	1	0	0	1	1	1	0	0	1	1
State	7	5	8	4	7	5	7	6	2	9	3
Triples		A							D		
				B			C				

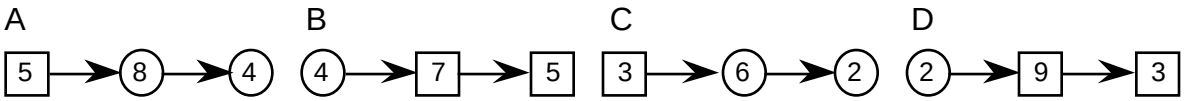


Figure 3-2 Graphs of State Triples for Flow Table in Table 3-3a.

This brings us to the third requirement: identifying the distinguishing states. As shown in this example, it is not enough to visit the distinguishing states. Two distinguishing inputs are said to be *linked* in a sequence, if the sequence provides enough information to determine the order of the distinguishing states in their two columns. If a distinguishing input follows another distinguishing input in the sequence, then the two are linked. However, there can be many distinguishing inputs. Every distinguishing input can have one of two possible permutations of distinguishing states. Therefore, for n distinguishing inputs there are 2^n possible column permutations. Half of these permutations are nothing more than other permutations with the rows exchanged. Thus they do not need to be considered, and there are 2^{n-1} possible unique permutations. For a single distinguishing input, there is one unique permutation. Therefore, any distinguishing input is linked to itself, making the link relation reflexive. From the definition of link, link is a symmetric property. If a is linked to b, then b is linked to a. Now suppose that there are three distinguishing inputs a, b and c. If a is linked to b, then there is only one unique permutation for the distinguishing states in columns of a and b. Similarly, if a is linked to c, then there is only one unique permutation for the distinguishing states in columns of a and c. Therefore, there is one unique permutation for the all three columns, and so the link relationship is transitive. Since the link relationship is reflexive, symmetric, and transitive, it must be an equivalence relationship.

Lemma 3: A checking experiment for a two-state flow table with only distinguishing and synchronizing inputs must link all distinguishing inputs to identify all distinguishing states.

Proof: We have already shown that a checking experiment must visit all the total states, including the distinguishing states. Suppose that two distinguishing inputs are not linked in a sequence. Since link is an equivalence relation between distinguishing inputs, the distinguishing inputs would fall into two equivalence

classes. Within each of the classes, there is only one unique permutation of distinguishing states. Create a second flow table by copying the original flow table, and swapping the rows in the distinguishing input columns of one of the equivalence classes. Also, swap the rows of any synchronizing inputs that are a unit distance from any of the distinguishing inputs in that class, if the synchronizing input is not a unit distance from a distinguishing input of the other class. Applying the sequence to the new flow table would give the same response as when applied to the original flow table. Therefore, if the distinguishing inputs are not linked in a sequence, the sequence is not a checking experiment.

Now that we have shown that the three conditions (visiting all states, identifying all unstable states, and identifying all distinguishing states) are necessary for a sequence to be a checking experiment, we show that if all three conditions are satisfied that the sequence is a checking experiment. In other words, given a two-state flow table with distinguishing and synchronizing inputs, any sequence that satisfies all three conditions is guaranteed to be a checking experiment. These conditions are necessary and sufficient. The proof is given in Theorem 1.

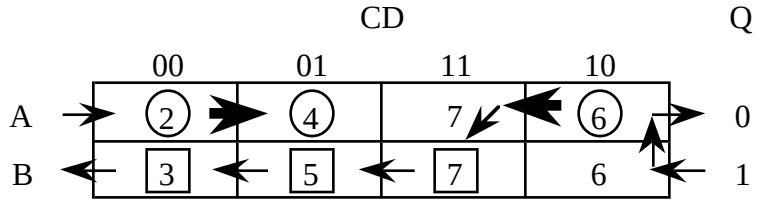
Theorem 1: A sequence for a two-state machine with distinguishing and synchronizing inputs is a checking experiment if and only if it satisfies the following properties:
1. Visits all the total states.
2. Visits at least one state triple for each unstable state.
3. Links all the distinguishing inputs.

Proof: From Lemmas 1, 2 and 3, a checking experiment must satisfy all the above conditions. Now, we need to show that if a sequence satisfies the three conditions, then it is a checking experiment. If a sequence links all the distinguishing inputs, then there can only be one permutation of distinguishing states in the flow table. If the sequence also visits at least one triple for each unstable state, then all entries in the flow table are identified. Therefore only one flow table can be constructed from the response of the sequence, making it a checking experiment.

An important consequence of Theorem 1 is that a checking experiment does not require all possible transitions in a two-state flow table. For example, consider the flow table in Table 3-4. The graphs of the triples are shown in Fig. 3-3. A sequence formed by combining the triples is 6,7,5,3,6,2. Since state 3 follows state 5, the distinguishing inputs are linked. State 4 can be added to the end of the sequence to satisfy the first requirement of Theorem 1. Therefore, the state sequence becomes: 6,7,5,3,6,2,4. The transitions through this sequence are shown graphically in Table 3-4. Thick arrows are used to indicate the beginning and end of the

sequence. The following six transitions are not included in this sequence: 4 -> 2, 2 -> 6, 4 -> 7, 3 -> 5, 5 -> 7, and 7 -> 6.

Table 3-4 Flow Table Marked With Checking Sequence.



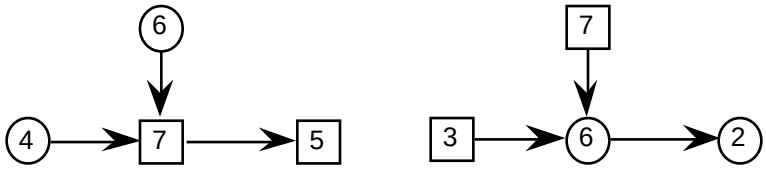


Figure 3-3 Graph of Triples for Flow Table in Table 3-4.

The above example suggests the following procedure to generate a checking experiment for a two-state flow table with distinguishing and synchronizing inputs. More examples for using this procedure are given in the following subsections.

Procedure for Deriving Checking Experiments from Two-State Flow Tables.

1. Determine all the state triples for the unstable states.
2. Select one triple for each unstable state.
3. Combine the triples of step 2. As in the example above, step 2 and 3 may be performed simultaneously (i.e., select the triples that best fit together), as long as a triple for each unstable state is used.
4. If the sequence resulting from step 3 does not form a link among all the distinguishing inputs then modify the sequence by adding extra states so that it does, without destroying the triples.
5. Add any missing total states to the sequence.
6. Convert the state sequence into an input sequence, adding a synchronizing input, if necessary (i.e., if the first input of the first triple is not a synchronizing input).

The first pattern in the sequence for a two-state flow table should force the machine into a known state. Therefore, in step 2 of the procedure, the setup state of the first triple should correspond to a synchronizing input whenever possible.

Combining triples would be most efficient if the distinguishing state of one triple is the setup state of another. Since triples cause a change in the internal state, the final state of one triple can be the setup state of another if the two triples cause internal state changes in the

opposite directions (i.e. if the first triple causes the machine to change from internal state A to internal state B, then the second triple should cause the machine to change from internal state B back to internal state A). If there are more triples that cause state changes in one direction than in the other, then some of the state changes will need to be repeated in order to get to the setup states of all the triples.

Lower Bound on Checking Experiment Length for Two-State Latches: The length of a checking experiment (L) is bounded by the following equation.

$$L \geq S + 1 + \sum_{i=0}^{S} \max(n_i - 1,0) \quad \text{if } v = 0$$

$$L \geq S + v + \sum_{i=0}^{S} \max(n_i - 1,0) \quad \text{if } v > 0$$

where S = number of total stable states

n_i = number of unstable states for which s_i is the only distinguishing state of all its triples

v = difference between the number of unstable states in the two rows

Proof: As seen from Lemma 1, a checking experiment must visit every total state. A machine can only be in one stable total state for every input pattern. Therefore, there must be at least as many patterns in the sequence as there are total stable states. There will always be at least one extra pattern for initialization. To be useful, the first pattern should force the machine into a known state (a synchronizing input). If there are n_i unstable states that require s_i as the only distinguishing state of all their triples, then s_i must appear at least n_i times. One of the occurrences of state s_i is accounted for in S (the total number of stable states). Therefore s_i must appear an additional n_i - 1 times. If n_i is zero, then nothing should be added. Hence the term max(n_i - 1,0) is added for each total state. If the number of unstable states in one row differs from the number of unstable states in the other row by *v*, then at least *v* -1 extra internal state changes have to be applied. Each extra state change requires at least one more pattern. Adding *v* -1 to the initialization pattern gives *v*.

In many of the latch flow tables, half the inputs are distinguishing inputs, and the other half are synchronizing inputs. A single input variable determines whether an input is a distinguishing input or a synchronizing input. This class of state machines will be referred to as *single-input control state machines*, and the variable that determines the input type will be called a *control input*. For example, in Table 2.2-5, all inputs are distinguishing inputs when C = 0, and

all inputs are synchronizing inputs when C = 1. Therefore, the flow table describes a single-input control state machine with C as the control input. An interesting property of such state machines is that the distinguishing state of one triple cannot be a setup state of another triple, because there are no unstable states adjacent to a distinguishing state of a triple. As before, if there are more triples that cause state changes in one direction than in the other, then some of the state changes will need to be repeated to get to the setup states of all the triples. In this case, since the distinguishing state of a triple is not a setup state of another triple, we will need two extra inputs instead of just one. Therefore, if there are *v* more unstable states in one row than in the other, then 2(*v* -1) extra patterns are needed. This is used to derive a tighter lower bound on the length of the checking experiment.

Lower Bound on Sequence Length of Single-Input Control State Machine: The length of a checking experiment (L) of a Single-Input Control State Machine is bound by

$$L \geq S + 1 \qquad \text{if } v = 0$$

$$L \geq S + 2v - 1 \qquad \text{if } v > 0$$

where S = number of stable total states

v = difference between the number of unstable states in the two rows

Proof: In a single-input control state machine each distinguishing input is a unit distance from only one synchronizing input. This implies that no two triples can have the same distinguishing state. Therefore, the summation term in the original bound will always be 0. If v = 0, then the arguments for the previous bound apply. If v > 1, then two patterns are needed for each additional transition because the distinguishing state of a triple cannot be used as a setup state for another triple. Therefore 2(v -1) extra patterns are needed. Combining this with the initialization pattern gives 2v -1.

Another property of single-input control state machines is that a sequence that uses distinguishing states as setup states for all except the first triple, will identify the distinguishing states as well as the unstable states. This property is proved in Theorem 2.

Theorem 2: If a sequence is applied to a single-input control state machine, and the setup states of all but the first triple are distinguishing states, then the sequence links all the distinguishing inputs and is a checking experiment.

Proof: In a single-input control state machine each distinguishing input has a distinguishing state that is a setup state of a triple, and another distinguishing state that is a distinguishing state of the same triple. Therefore, there is a one-to-one correspondence between distinguishing inputs and triples. Now, if triple B is applied after triple A, the distinguishing input corresponding to triple A is linked to the one corresponding to triple B. Suppose triple C is applied after triple B, then the distinguishing input corresponding to triple B is linked to the one corresponding to triple C. Since link is an equivalence relation, the distinguishing input corresponding to triple A is linked to the one corresponding to triple C. Using the same argument, it can be shown that the distinguishing input corresponding to triple A is linked to all the distinguishing inputs. Therefore, the sequence links all the distinguishing inputs, identifying all the stable states. Since the sequence identifies all the stable states and the unstable states, it is a checking experiment.

3.1 Two-State SR-Latch

The equation for the SR-latch is $Q = S + \overline{R}q$. The latch is set when S = 1 and reset when R = 1. S and R should not be 1 at the same time (an *set dominant SR-latch* would be set if SR = 1, and a *reset dominant SR-latch* would be reset if SR = 1). The flow table for the SR-latch is shown in Table 3.1-1. The graphs of the state triples are shown in Fig. 3.1-1.

The setup state of one triple is the distinguishing state of the other triple. Therefore, the two triples can be combined without adding any states between them. Two possible state sequences are 2,5,3,4,2 and 3,4,2,5,3. Since there is only one distinguishing input, the

Table 3.1-1 Flow Table for Two-State SR-Latch.

	SR				Q
	00	01	11	10	
	(2)	(4)	–	5	0
	[3]	4	–	[5]	1

Figure 3.1-1 Graphs of State Triples for Two-State SR-Latch.

distinguishing states are identified by simply visiting them. The two sequences contain all the total states in the flow table. The input sequence that would generate these state sequences is shown in Table 3.1-2. Since the sequences satisfy Theorem 1, they are checking experiments. Since the setup state in both triples is a distinguishing state, a synchronizing input needs to be added to the beginning of either sequence. The sequences in Table 3.1-2 are minimum length.

Table 3.1-2 Minimum-Length (6) Checking Experiments for Two-State SR-Latch.

S	0	0	1	0	0	0	S	1	0	0	0	1	0
R	1	0	0	0	1	0	R	0	0	1	0	0	0
Q	0	0	1	1	0	0	Q	1	1	0	0	1	1
State	4	2	5	3	4	2	State	5	3	4	2	5	3
Triples		B					Triples		A				
				A							B		

3.2 Two-State D-Latch

The equation for a D-latch is $Q = CD + \overline{C}\,q$, and the flow table for the D-latch is shown in Table 3.2-1. The graphs of the state triples are shown in Fig. 3.2-1. The two triples can be combined in any order. Suppose we start with the triple A. The first part of the state sequence is 6,7,5 (state 6 is picked because it is a synchronizing input). Now, looking at triple B, there are two choices: 7,6,2 or 3,6,2. State 7 has already been entered in the first sequence, thus entering it again has no benefit. Choosing state 3 as the setup state would make state 3 follow state 5, linking the two distinguishing inputs. Thus the sequence becomes 6,7,5,3,6,2. The sequence is missing state 4, which can be added to the end of the sequence. This makes the final sequence 6,7,5,3,6,2,4. The same approach can be used starting with triple B in Fig. 3.2-1. The sequence

Table 3.2-1 Flow Table for Two-State D-Latch.

	CD				Q
	00	01	11	10	
	(2)	(4)	7	(6)	0
	[3]	[5]	[7]	6	1

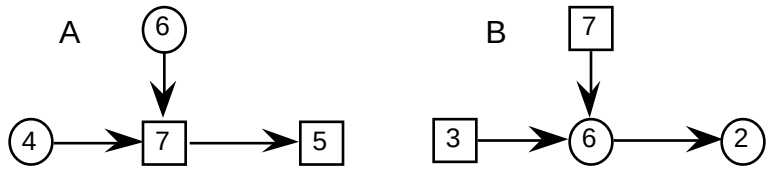


Figure 3.2-1 Graphs of State Triples for Two-State D-Latch.

in that case would be 7,6,2,4,7,5,3. These two state sequences, and the checking experiments that would generate them, are shown in Table 3.2-2. Since the lengths of these sequences meet the lower bound, these sequences are minimum length.

Table 3.2-2 Minimum-Length (7) Checking Experiments for Two-State D-Latch.

C	1	1	0	0	1	0	0	C	1	1	0	0	1	0	0
D	1	0	0	1	1	1	0	D	0	1	1	0	0	0	1
Q	1	0	0	0	1	1	1	Q	0	1	1	1	0	0	0
State	7	6	2	4	7	5	3	State	6	7	5	3	6	2	4
Triples	B			A				Triples	A			B			

3.3 Two-State D-Latch With Asynchronous Set/Reset

The equation for the D-latch with Asynchronous Set/Reset is $Q = \overline{R}(S + CD + \overline{C}q)$. The latch is set when S = 1 and reset when R = 1. S and R should not be 1 at the same time, and neither should be 1 when C = 1. When both R and S are 0, the latch behaves exactly like a D-latch. The flow table for this latch is shown in Table 3.3-1, and graphs of the state triples are shown in Fig. 3.3-1.

Table 3.3-1 Flow Table for Two-State Asynchronous Set/Reset Latch.

	CD																
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
	R = 0								R = 1								
	S = 0				S = 1				S = 0				S = 1				Q
	(2)	(4)	7	(6)	9	11	-	-	(8)	(10)	-	-	-	-	-	-	0
	[3]	[5]	[7]	6	[9]	[11]	-	-	8	10	-	-	-	-	-	-	1

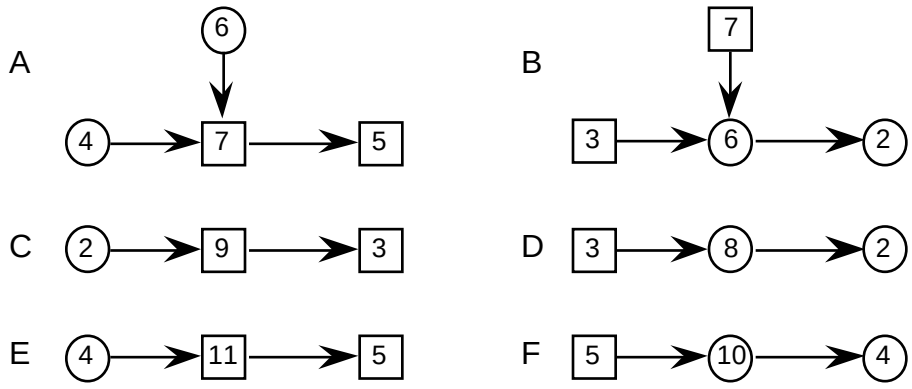


Figure 3.3-1 Graphs of State Triples for Two-State D-Latch With Asynchronous Set/Reset.

In Fig. 3.3-1 only the first two triples have synchronizing setup states. Therefore, a minimum-length sequence should start with one of these two triples. There are ten total states, and state 2 and state 5 appear twice as the only distinguishing inputs of state triples. From the first bound derived in Section 3, the minimum test length must be at least 13. State 4 and state 3 appear twice as setup states. However, in the first two triples there are alternate setup states. If these are used, then one of them could be the synchronizing input, but the other would appear twice in the sequence, raising the minimum length to 14. One such sequence is shown in Table 3.3-2. Since state 4 directly follows state 2, the distinguishing inputs are linked.

Table 3.3-2 A Minimum-Length (14) Checking Experiment for Asynchronous Set/Reset Latch.

C	1	1	0	0	0	0	0	0	0	0	0	0	1	0
D	1	0	0	0	0	0	0	1	1	1	1	1	1	1
R	0	0	0	0	0	1	0	0	0	0	1	0	0	0
S	0	0	0	1	0	0	0	0	1	0	0	0	0	0
Q	1	0	0	1	1	0	0	0	1	1	0	0	1	1
State	7	6	2	9	3	8	2	4	11	5	10	4	7	5
Triples	B				D					F				
			C				E					A		

3.4 Two-State MD-Latch

The equation for an MD-latch (Multiplexed-Data latch) is $Q = C(TS + \overline{T}D) + \overline{C}q$. When T = 0, the latch operates in normal mode (D is used as the input), and when T = 1 it uses S as input. The flow table for the MD-latch is given in Table 3.4-1. The graphs of the triples are shown in Fig. 3.4-1.

Table 3.4-1 Flow Table for Two-State MD-Latch.

DS

00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
C = 0								C = 1								
T = 0				T = 1				T = 0				T = 1				Q
(2)	(4)	(6)	(8)	(10)	(12)	(14)	(16)	(18)	(20)	19	21	(22)	23	25	(24)	0
[3]	[5]	[7]	[9]	[11]	[13]	[15]	[17]	18	20	[19]	[21]	22	[23]	[25]	24	1

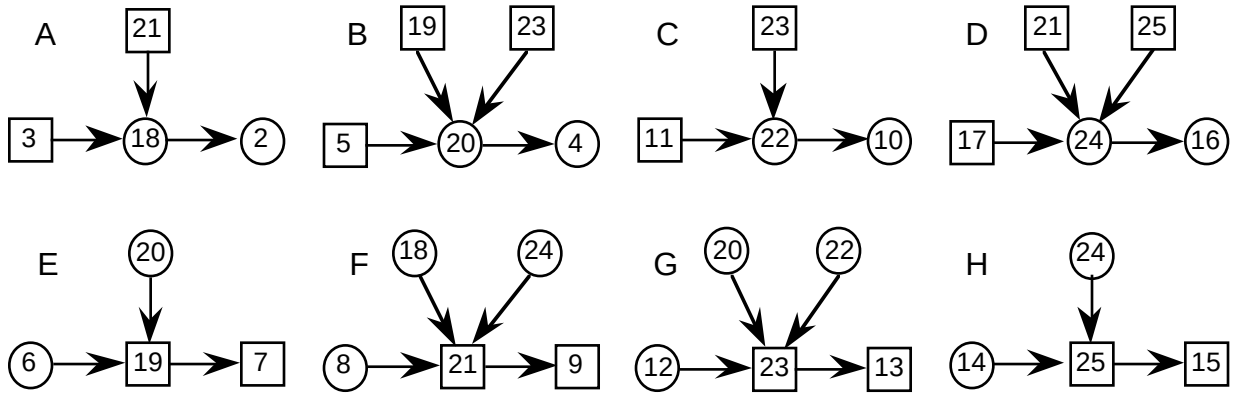


Figure 3.4-1 Graphs of State Triples for Two-State MD-Latch.

Looking at the flow table, all the distinguishing states occur when C = 0, and all the unstable states occur when C = 1. Therefore, the MD-latch is a single-input control state machine, and C is the control input. There are 24 total states, and the number of unstable states in each of the rows is 4. Therefore, the length of a checking experiment must be at least 25. Appendix A shows that the length must be at least 26. The details of combining the triples to derive minimum-length sequences also appear in Appendix A. One such sequence is shown in Table 3.4-2.

Table 3.4-2 A Minimum-Length (26) Checking Experiment for MD-Latch.

D	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
S	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0
T	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
C	1		0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0
Q	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	1	0	0
State	24	21	9	3	18	2	8	6	19	7	5	20	4	12	23	13	11	22	10	12	14	25	15	17	24	16
Triples	F			A				E			B			G			C				H			D		

3.5 Two-State Two-Port Latch

The equation for a Two-Port latch is $Q = C_1D_1 + C_2D_2 + \overline{C_1}\,\overline{C_2}q$. C_1 and C_2 should not be both active at the same time. The Two-Port latch loads the data input corresponding to the active control input. The flow table is shown in Table 3.5-1.

The last four columns are marked as don't cares because the operation of the latch is not defined when both control lines are active. Graphs of the triples are shown in Fig. 3.5-1. Since states 2 and 7 appear twice as distinguishing states of triples, and there are 16 total states, the lower bound on the test length is 19. Appendix B shows that the length must be at least 23. The

details of combining the triples to derive minimum-length sequences also appear in Appendix B. One such sequence is shown in Table 3.5-2.

Table 3.5-1 Flow Table for Two-State Two-Port Latch.

	D_1D_2																
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
	$C_2 = 0$								$C_2 = 1$								
	$C_1 = 0$				$C_1 = 1$				$C_1 = 0$				$C_1 = 1$				Q
	(2)	(4)	(6)	(8)	(10)	(12)	11	13	(14)	15	17	(16)	-	-	-	-	0
	[3]	[5]	[7]	[9]	10	12	[11]	[13]	14	[15]	[17]	16	-	-	-	-	1

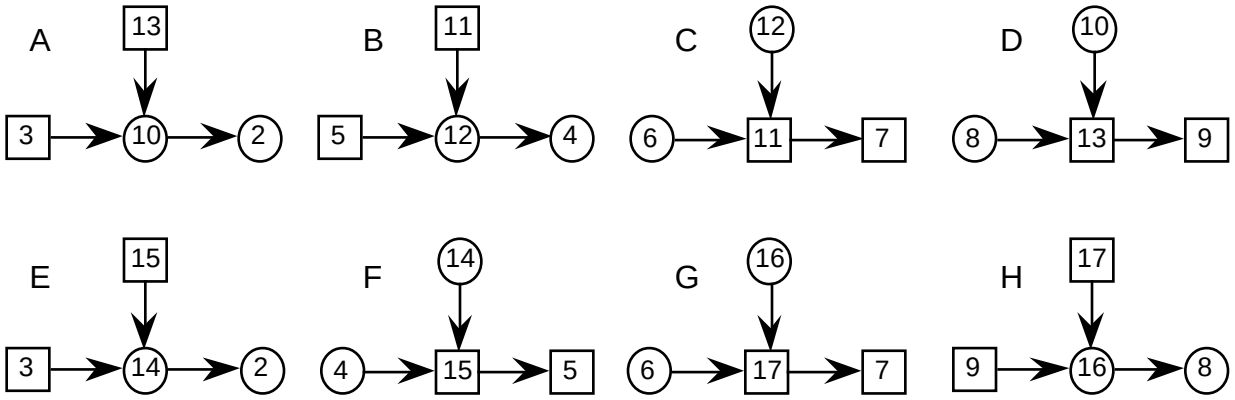


Figure 3.5-1 Graphs of State Triples for Two-State Two-Port Latch.

Table 3.5-2 A Minimum Length (23) Checking Experiment for Two-Port Latch.

D_1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1
C_1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0
D_2	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
C_2	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
Q	1	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1
State	13	10	2	4	15	5	3	14	2	8	13	9	16	8	6	17	7	5	12	4	6	11	7
Triples	A			F			E			D													
												H			G			B			C		

3.6 Two-State Load Enable Latch

The equation for the Load Enable latch is $Q = CLD + \overline{(LC)}q$. When L = 0, the clock has no effect on the latch, and the latch retains the stored value. When L = 1, it behaves like a D-latch. The flow table for the Load Enable latch is given in Table 3.6-1. Graphs of the triples are shown in Fig. 3.6-1. Since there are 14 stable total states, a checking experiment must have at least 15 patterns. One such sequence is shown in Table 3.6-2. The details are in Appendix C.

Table 3.6-1 Flow Table for Two-State Load Enable Latch.

LD

00	01	11	10	00	01	11	10	
C = 0				C = 1				Q
(2)	(4)	(6)	(8)	(10)	(12)	15	(14)	0
[3]	[5]	[7]	[9]	[11]	[13]	[15]	14	1

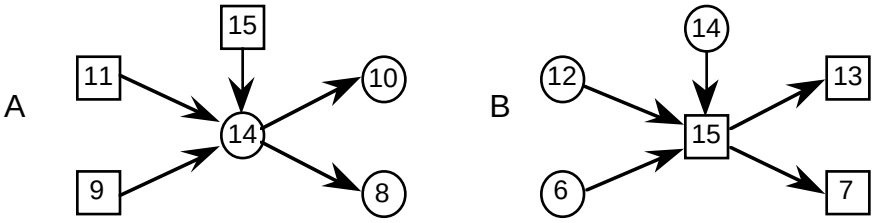


Figure 3.6-1 Graphs of State Triples of Two-State Load Enable Latch.

Table 3.6-2 A Minimum-Length (15) Checking Experiment for Load Enable Latch.

L	1	1	0	0	1	1	0	0	1	0	0	1	1	0	0
D	1	0	0	0	0	1	1	1	1	1	1	1	0	0	0
C	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1
Q	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1
State	15	14	10	2	8	6	4	12	15	13	5	7	9	3	11
Triples	A							B							

3.7 Two-State D-Enable Latch

The equation for the D-Enable latch is $Q = CDE + \overline{C}q$. A D-Enable latch operates as a D-latch when E = 1. When E = 0, the D-Enable latch will load 0. The flow table for the D-Enable latch is shown in Table 3.7-1. The graphs of the state triples are shown in Fig 3.7-1.

Looking at the flow table, all the distinguishing states occur when C = 0, and all the unstable states occur when C = 1. Therefore, the D-Enable latch is a single-input control state machine, and C is the control input. There are 12 total states, one unstable state in the first row,

Table 3.7-1 Flow Table for Two-State D-Enable Latch.

DE

00	01	11	10	00	01	11	10	
C = 0				C = 1				Q
(2)	(4)	(6)	(8)	(10)	(12)	11	(14)	0
[3]	[5]	[7]	[9]	10	12	[11]	14	1

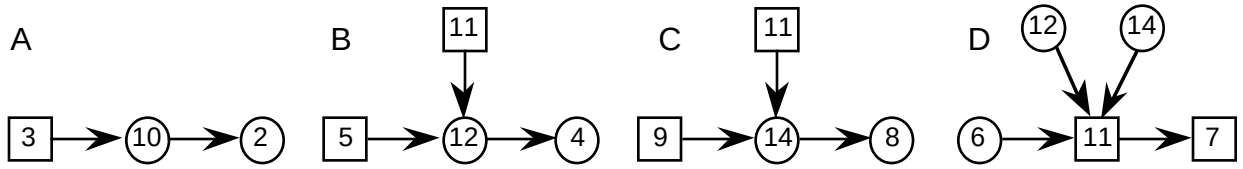


Figure 3.7-1 Graphs of State Triples for Two-State D-Enable Latch.

Table 3.7-2 A Minimum-Length (16) Checking Experiment for Two-State D-Enable Latch.

D	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0
E	1	1	1	1	1	0	0	1	1	1	0	0	1	0	0	0
C	1	1	0	0	1	1	0	0	1	0	0	0	0	0	1	0
Q	1	0	0	0	1	0	0	0	1	1	1	1	1	1	0	0
State	11	12	4	6	11	14	8	6	11	7	9	3	5	3	10	2
Triples	B				C			D						A		

and three in the second. Therefore, the length of a checking experiment must be at least 15. Appendix D shows that the length must be at least 16. One possible minimum-length sequence is shown in Table 3.7-2. Details of deriving a checking experiment are given in Appendix D.

3.8 Two-State XOR Input Latch

The equation for the XOR input latch is $Q = C(D \oplus S) + \overline{C}q$. The data loaded into the latch is $D \oplus S$. The flow table is given in Table 3.8-1. Since there are 12 stable states in the flow table, a checking experiment must have at least 13 patterns. Graphs of the state triples are shown in Fig. 3.8-1.

Table 3.8-1 Flow Table for Two-State XOR Latch.

	DS								
	00	01	11	10	00	01	11	10	
	C = 0				C = 1				Q
	(2)	(4)	(6)	(8)	(10)	11	(12)	13	0
	[3]	[5]	[7]	[9]	10	[11]	12	[13]	1

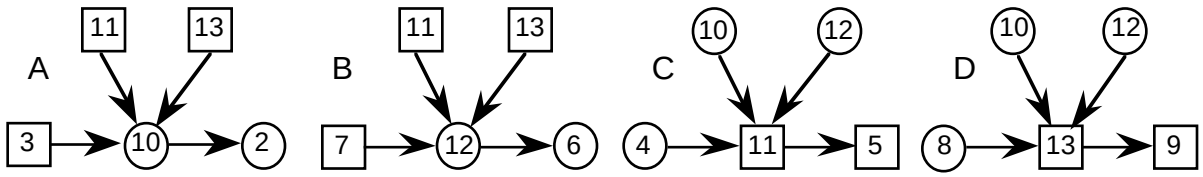


Figure 3.8-1 Graphs of State Triples for Two-State XOR latch.

Starting with any triple, there are two possible sequences. For example, starting with A, the two sequences are ACBD and ADBC. Since there are four triples, and any of them can be picked as the starting point, there are 8 sequences. Note that a sequence that connects the triples not only demonstrates all the unstable states, but also visits all the stable states. The transition from one triple to the other forms a link among all distinguishing inputs. The sequences formed by connecting the triples are all minimum-length checking experiments. One of the minimum-length checking experiments is given in Table 3.8-2.

Table 3.8-2 A Minimum-Length (13) Checking Experiment for XOR Input Latch.

D	0	0	0	0	0	0	1	1	1	1	1	1	0
S	1	0	0	1	1	1	1	1	1	0	0	0	0
C	1	1	0	0	1	0	0	1	0	0	1	0	0
Q	1	0	0	0	1	1	1	0	0	0	1	1	1
States	11	10	2	4	11	5	7	12	6	8	13	9	3
Triples	A			C			B			D			

3.9 Two-State BILBO Latch

The equation for a BILBO latch is $Q = C(B_1D \oplus \overline{B_2}S) + \overline{C}q$. Based on the setting of B_1 and B_2, the latch can be configured to load D (when B_1B_2 = 11), reset the latch (when B_1B_2 = 01), load S (when B_1B_2 = 00), and load $S \oplus D$ (when B_1B_2 = 10). The flow table for the BILBO latch is given in Table 3.9-1.

Table 3.9-1 Flow Table for Two-State BILBO Latch.

DS (C = 0)

00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
B_2 = 0								B_2 = 1								
B_1 = 0				B_1 = 1				B_1 = 0				B_1 = 1				Q
(2)	(4)	(6)	(8)	(10)	(12)	(14)	(16)	(18)	(20)	(22)	(24)	(26)	(28)	(30)	(32)	0
[3]	[5]	[7]	[9]	[11]	[13]	[15]	[17]	[19]	[21]	[23]	[25]	[27]	[29]	[31]	[33]	1

DS (C = 1)

00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
B_2 = 0								B_2 = 1								
B_1 = 0				B_1 = 1				B_1 = 0				B_1 = 1				Q
35	(34)	(36)	37	(38)	(40)	(42)	(44)	39	(46)	41	(48)	(50)	(52)	43	45	0
[35]	34	36	[37]	38	40	42	44	[39]	46	[41]	48	50	52	[43]	[45]	1

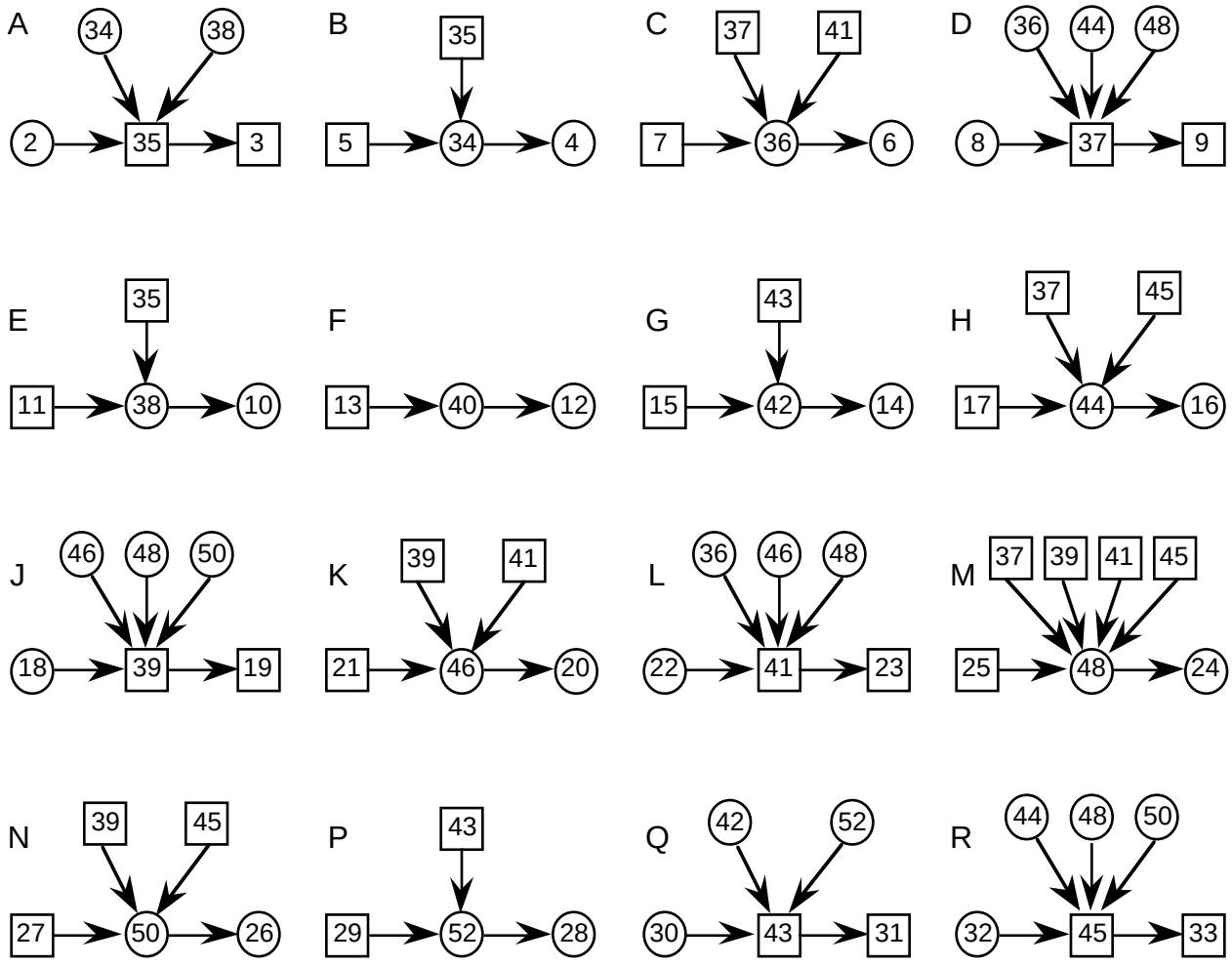


Figure 3.9-1 Graphs of State Triples for Two-State BILBO Latch.

Table 3.9-2 A Minimum-Length (58) Checking Experiments for Two-State BILBO Latch.

D	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1
S	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
B_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
B_1	0	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
C	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0
Q	1	0	0	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1
State	35	38	10	2	35	3	5	34	4	2	35	17	44	16	32	45	33	27	50	26	18	39	19	21	46	20	22	41	23
Triples	E			A			B				H			R			N			J			K			L		
D	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0
S	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1
B_2	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
B_1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
C	0	1	0	0	1	0	0	1	0	0	1	1	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0
Q	1	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	1	1	1	1	0	0
State	7	36	6	8	37	9	25	48	24	32	45	43	42	14	30	43	31	29	52	28	30	43	31	15	17	11	13	40	12
Triples	C			D			M				G			Q			P									F		

Looking at the flow table, all the distinguishing states occur when C = 0, and all the unstable states occur when C = 1. Therefore, the BILBO latch is a single-input control state machine, and C is the control input. There are 48 total states, and four more unstable states in the second row than in the first. Therefore, the length of a checking experiment must be at least 55. Appendix E shows that the length must be at least 58. One possible minimum-length sequence is shown in Table 3.9-2. Details of deriving a checking experiment are given in Appendix E.

3.10 Two-State CBILBO Latches

CBILBO latches are an extension of BILBO latch that can operate simultaneously as a pseudo-random pattern generator and a signature analyzer. Each of the two CBILBO latches has a different mode signal. The first latch loads S when $B_1 = 1$, and loads $\overline{S \oplus D}$ when $B_1 = 0$. The second latch is an MD-latch, with B_2 as the select input, S and D as the data inputs. Since the MD-latch has already been analyzed, only the first latch is considered in this section. The equation of this latch is $Q_1 = C(\overline{B}_1 D \oplus S) + \overline{C} q_1$, and the flow table is given in Table 3.10-1. The graphs of the state triples are shown in Fig. 3.10-1.

Looking at the flow table, all the distinguishing states occur when C = 0, and all the unstable states occur when C = 1. Therefore, the CBILBO latch is a single-input control state machine, and C is the control input. Since there are 24 total states, the length of a checking experiment must be at least 25. An example of a minimum-length sequence is shown in Table 3.10-2. Details of deriving a checking experiment are given in Appendix F.

Table 3.10-1 Flow Table for Two-State CBILBO.

	DS																
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
	C = 0								C = 1								
	$B_1 = 0$				$B_1 = 1$				$B_1 = 0$				$B_1 = 1$				Q_1
0	(2)	(4)	(6)	(8)	(10)	(12)	(14)	(16)	(18)	19	(20)	21	(22)	23	25	(24)	0
1	[3]	[5]	[7]	[9]	11	[13]	[15]	[17]	18	[19]	20	[21]	22	[23]	[25]	24	1

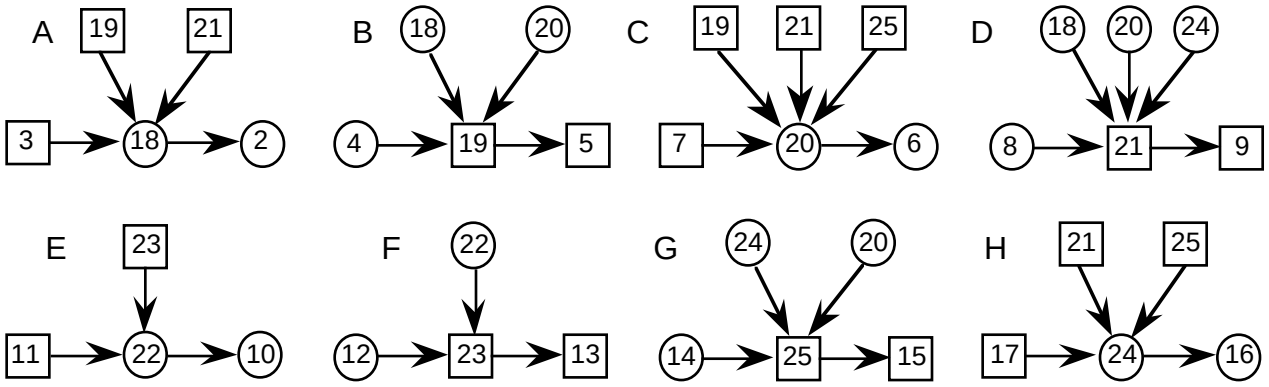


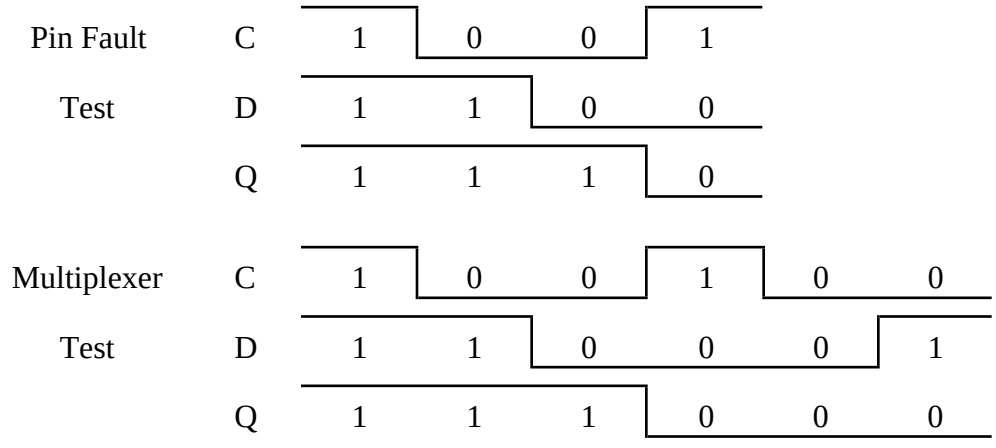
Figure 3.10-1 Graphs of State Triples for Two-State CBILBO Latch.

Table 3.10-2 A Minimum-Length (25) Checking Experiment for CBILBO Latch.

D	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0
S	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0
B_1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
C	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0
Q_1	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	1	0	0
State	24	25	15	7	20	6	4	19	5	3	18	2	8	21	9	17	24	16	14	12	23	13	11	22	10
Triples	G			C			B			A			D			H				F			E		

4. D-Latch Simulation

In this section, three different tests for the D-latch simulated using HSpice are compared. The first test is a pin fault test set, which targets the faults on the input and output of the D-latch. A D-latch can be viewed as a multiplexer that selects between D and Q, with C being the select signal. The second test is a multiplexer-based test. Patterns for testing multiplexers can be found in [Makar 88]. The third test is the checking experiment derived in Section 3.2. The three tests are shown in Fig. 4-1. The implementation used for the simulation is shown in Fig. 4-2.



Checking	C	1	1	0	0	1	0	0
Experiment	D	1	0	0	1	1	1	0
	Q	1	0	0	0	1	1	1

Figure 4-1 Test Sequences for Simulating D-Latch.

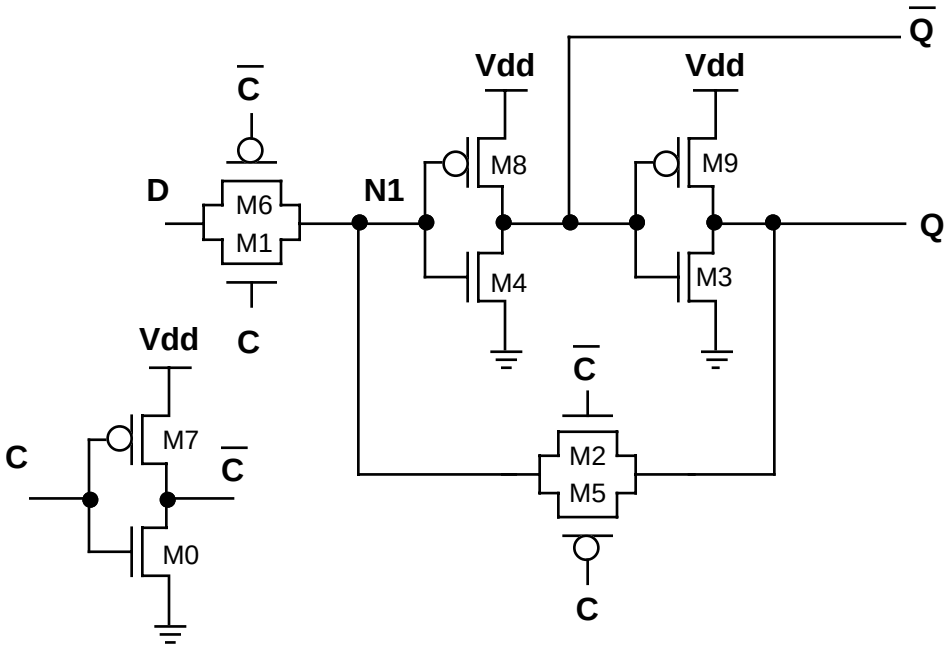


Figure 4-2 Transmission Gate D-Latch.

In the simulation, faults are injected by modifying the circuit description. The fault models used are based on the CrossCheck fault models [Sucar 89] and [Chandra 93]. The faults injected are shorted interconnects (STI), open interconnects (OPI), short-to-power (STP), short-to-ground (STG), transistor stuck-on (SON), and transistor stuck-open (SOP).

In CMOS, there are some faults whose presence does not change the functionality of the host circuit. Some of these cannot be detected (and thus are untestable or redundant). Others that cannot be detected by a Boolean voltage test (since the circuit functionality is correct) can, nevertheless, be discovered by a current test [Ma 95]. The simulations reported here record whether tests caused excessive supply current or incorrect outputs.

The current limit for IDDQ testing is often determined experimentally, by plotting the values of many good and bad die, and selecting an appropriate threshold that would detect as many faulty circuits as possible without discarding many good ones [Hawkins 89] and [Perry 92]. Fig. 4-3 shows the IDDQ distribution for circuits with the faults described above when the checking experiment was applied. Here IDDQf refers to the IDDQ value of a circuit with a fault present, and IDDQg refers to the IDDQ value of the fault-free circuit. The graph

plots the ratio of IDDQf / IDDQg (i.e. the ratio of IDDQ increase), versus the faults. The graph shows that all but 7 of the faults cause an IDDQ that is over 100 thousand times that of the normal current. Using this graph, a threshold of 32 uA was selected. Maximum IDDQ for the fault-free circuit was about 320 pA.

The results of the simulations are shown in the graph of Fig. 4-4. For each test three numbers are reported: the number of faults detected by either a voltage or current test, the number of faults detected if the voltage measurement is used alone, and the number of faults detected if the current measurement is used alone. In spite of the fact that the checking experiments were generated to verify the functionality of the latches, the graph shows that, in addition to detecting functional faults, they are very useful in detecting faults that only cause excessive current. One drawback to the current tests is the large amount of time they require.

The graph also shows that the pin fault test and the multiplexer test miss several faults that are detected by the checking experiment. The faults missed by each of the tests are shown graphically in Fig. 4-5 through 4-7. In these figures white ovals indicate SOP or OPI faults, black ovals indicate SON faults, and thick black lines indicate STI faults. All STP and STG faults are detected by all three tests.

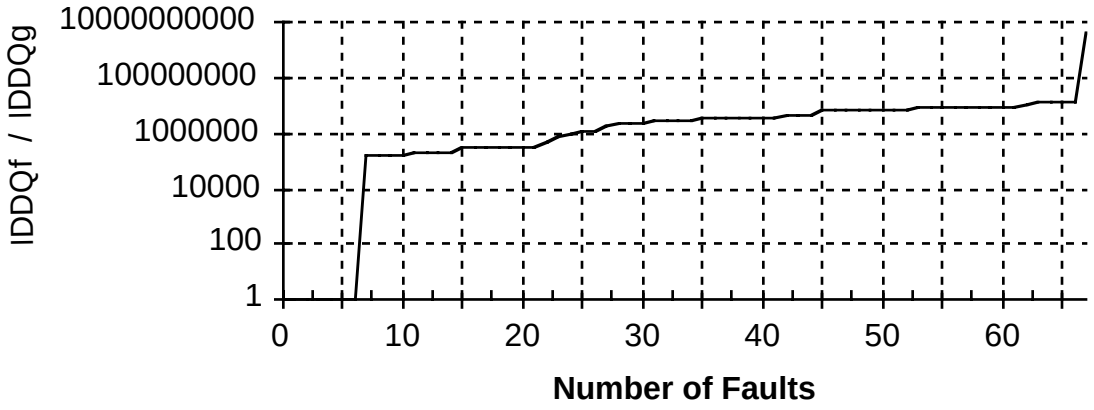


Figure 4-3 IDDQf / IDDQg Distribution.

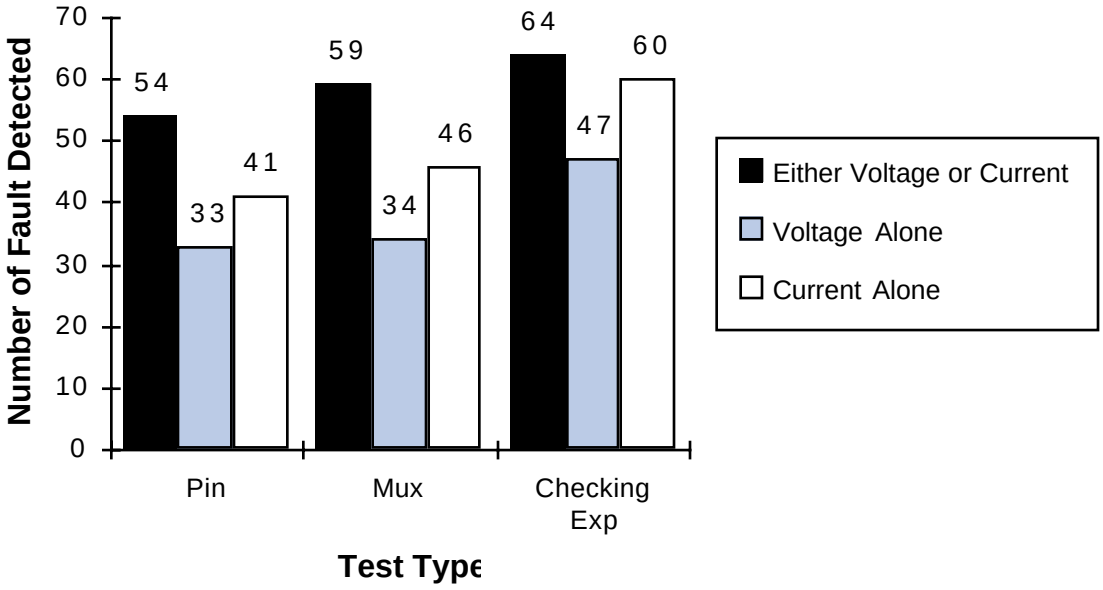


Figure 4-4 Fault Coverage of the Three Test Sets (Total Faults = 67).

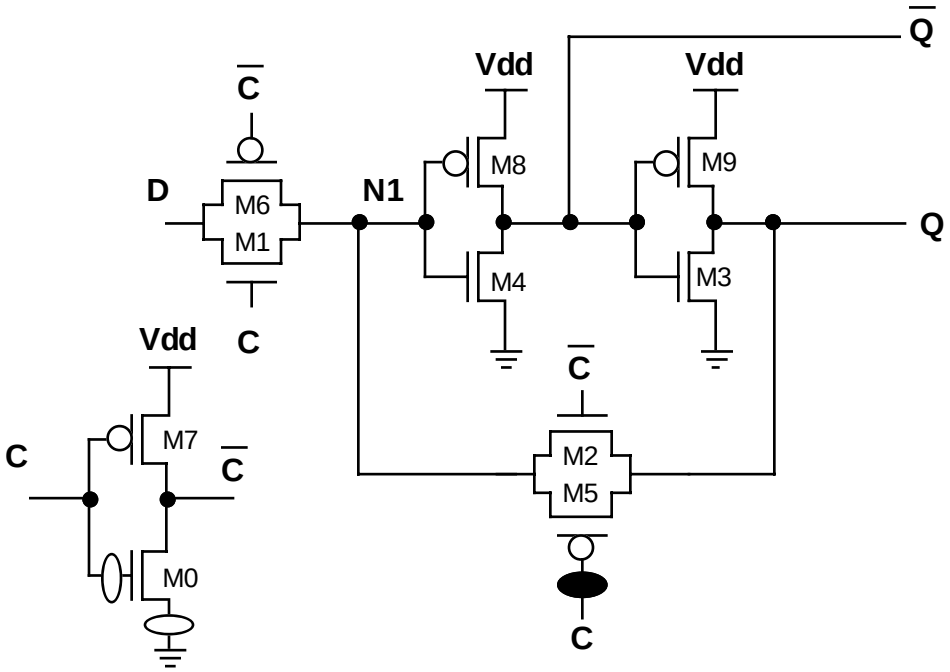


Figure 4-5 Faults Missed by Checking Experiment.

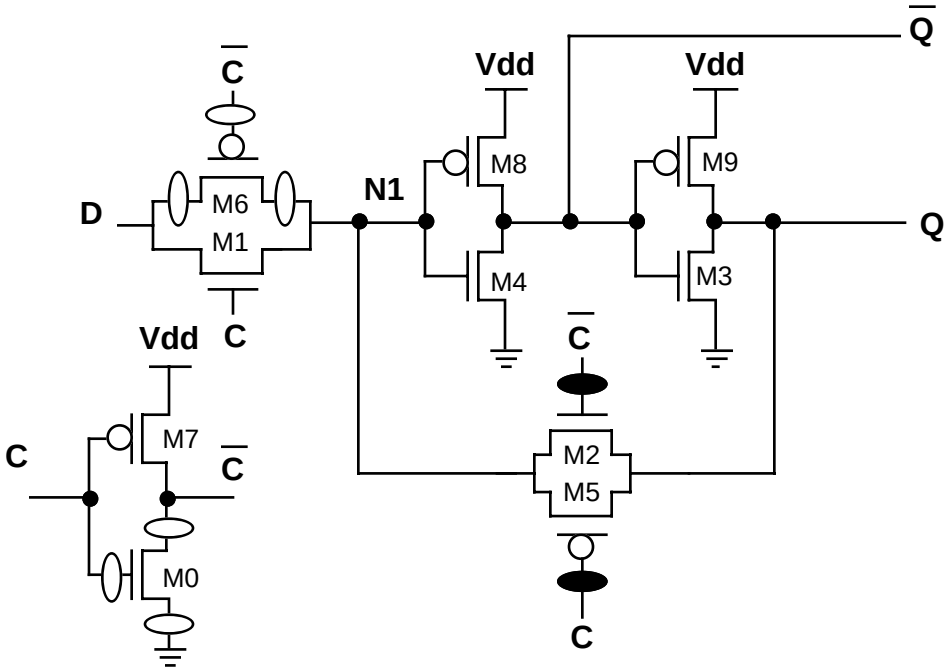


Figure 4-6 Faults Missed by Multiplexer Test.

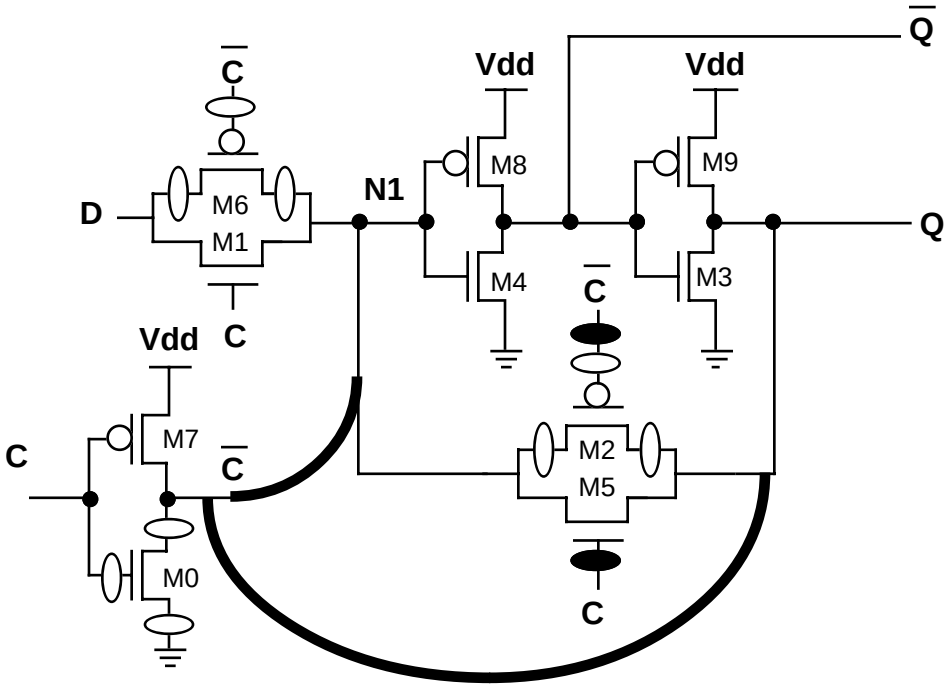


Figure 4-7 Faults Missed by Pin Fault Test.

The faults missed by the checking experiment fall into one of two categories. The faults on the M0 transistor cause the number of states to increase. With the presence of the fault, the application of CD = 10 when in state 7 (see Table 4-1), produces an output of 0. However, when the same input is applied to state 3, the output is 1. Therefore, states 7 and 3 cannot be in the

same internal state, and there must be a third internal state. The stuck-on fault on M5 is not detected because it is on the feedback path. Even though not indicated in the figures, the NMOS transistors of the transmission gates are stronger than their PMOS counterparts. This makes it possible to detect M2 stuck-on, but not detect M5 stuck-on.

Table 4-1 Flow Table for Two-State D-Latch.

	CD			Q
00	01	11	10	
(2)	(4)	7	(6)	0
[3]	[5]	[7]	6	1

As mentioned earlier, some faults are only detected by current measurement. In the D-latch, there are two kinds of faults that are detected by current and not by voltage. A stuck-on fault in M8 or M9 would cause a large IDDQ current when the corresponding NMOS transistor is turned on. However, due to the sizing ratio between the two transistors, the output voltage value would not be affected enough to be detected. In fact, the inverter would behave as an NMOS inverter rather than a CMOS inverter. A plot of the relevant voltages and currents in the circuit with M8 stuck-on fault are shown in Fig. 4-8. In these graphs, $\overline{Q}$ has a fault-free value of 0 volts between 10 and 40 ms. With the presence of the fault, this voltage is around 1 volt, which is not high enough to be detected by a voltage test. In the same time frame, the IDDQ current reaches 2.5 mA. The other kind of faults that are only detected by current measurements are the stuck-open faults on M6 and M1, and faults on transmission gate in the feedback path. These faults cause voltage degradation on N1 which in turn causes M8 or M4 to turn on when they should not, raising the IDDQ current. A plot of the relevant voltages and currents in the circuit with M1 stuck-open fault are shown in Fig. 4-9. For the circuit with M5 stuck-open fault, the voltages and currents are shown in Fig. 4-10. In this case, the current reaches around 50 uA. Even though this value is much lower than that of M8 stuck-on fault, it is still above the threshold of 32 uA.

Some faults can only be detected by voltage measurements. Stuck-open faults on M3, M4, M8, and M9 do not substantially increase the IDDQ current. Also, a STI fault between D and N1 does not increase the IDDQ current.

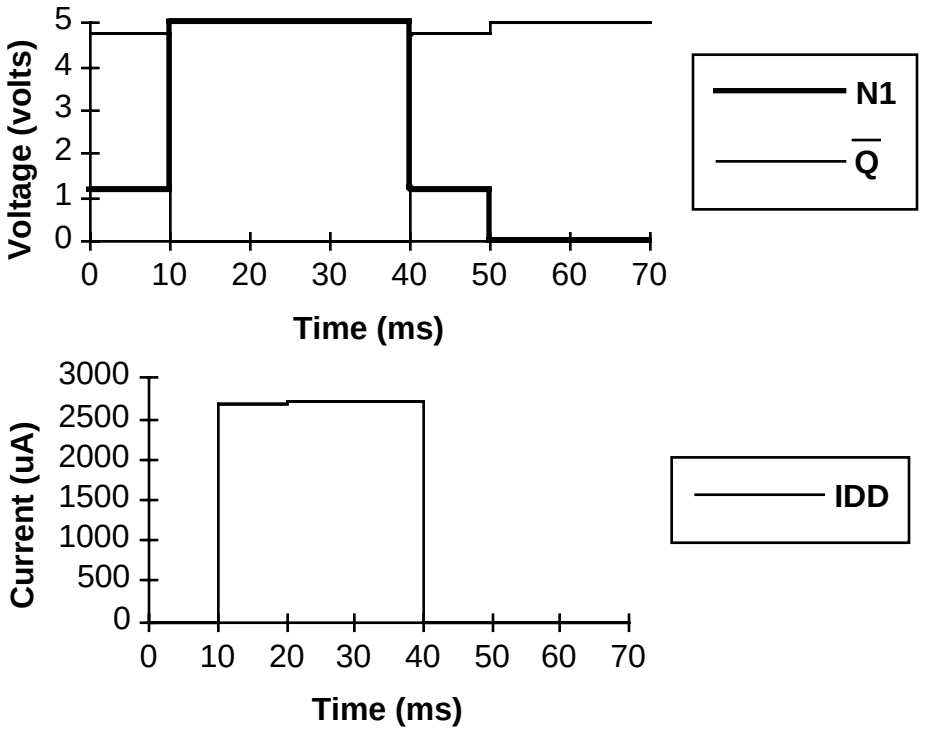


Figure 4-8 HSpice Output for M8 Stuck-On.

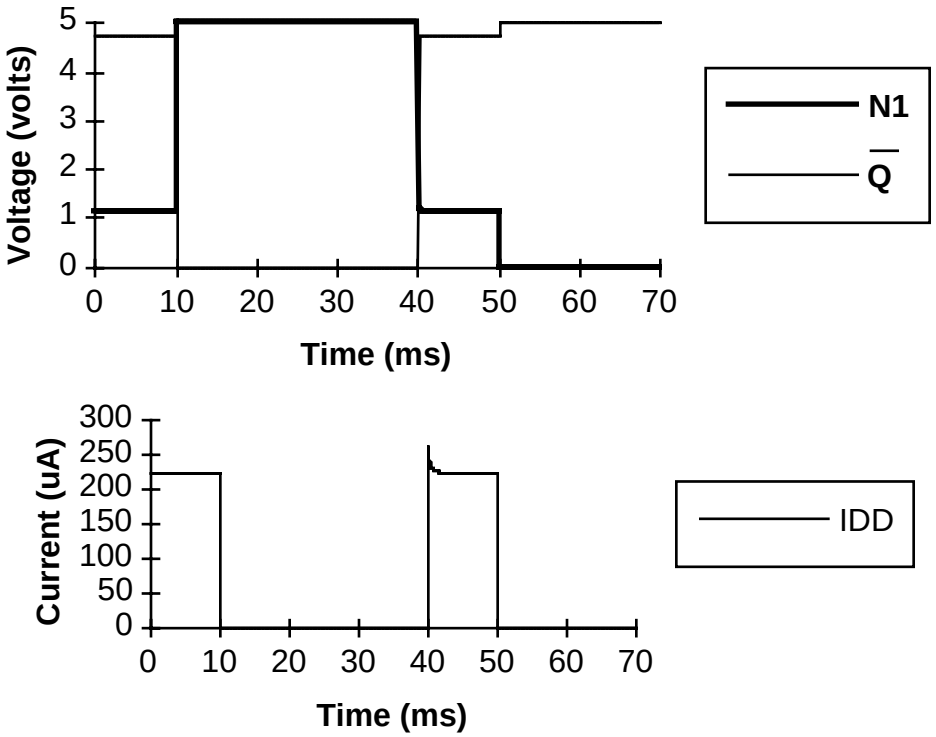


Figure 4-9 HSpice Output for M1 Stuck-Open.

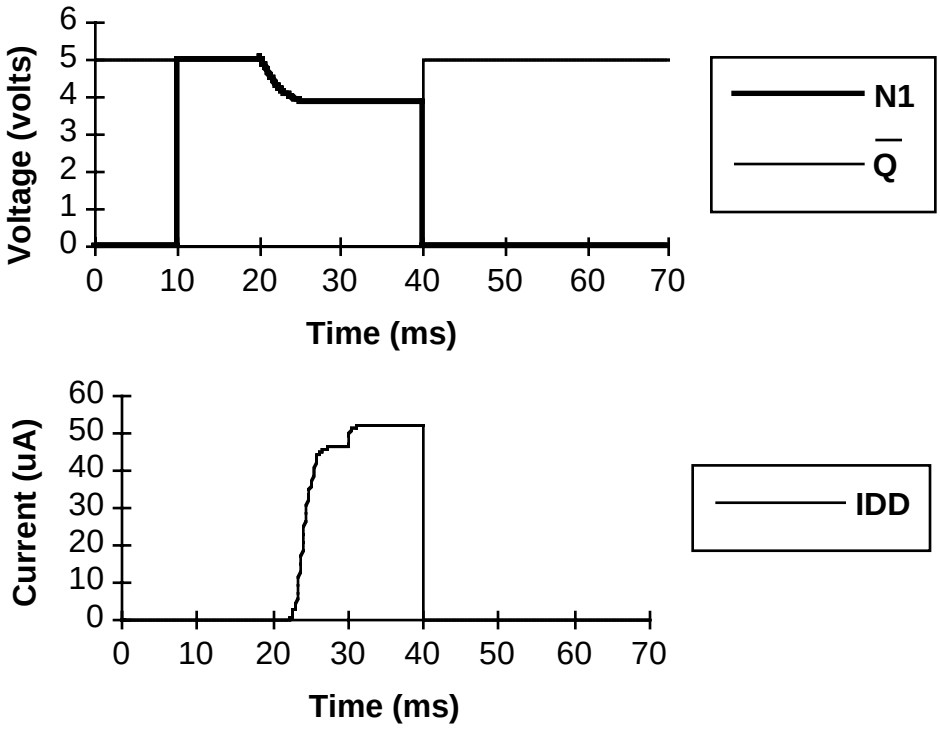


Figure 4-10 HSpice Output for M5 Stuck-Open.

5. Conclusions

Ten latch types were studied. For each, we derived requirements that can be used to verify whether or not a given sequence is a checking sequence. In addition, one example of a minimum-length checking sequence is derived for each latch type.

These checking sequences are guaranteed to detect any latch defects that do not increase the number of states and do not cause the latch operation to depend on other elements in the design (coupling or transition faults). To investigate whether the increase-in-state restriction could be a problem, one latch implementation was simulated using HSpice for several faults that could cause extra states. The checking sequence missed only two faults that cause an increase in state. A test set for the pin faults of the latch, and a test set for the multiplexer in the latch were also simulated. These two tests missed faults that were detected by the checking sequence.

These checking sequences are, in fact, shorter than some of the popular “rule of thumb” sequences such as “read 0 to 1, 1 to 0, 0 to 1, 1 to 1” and also shorter than sequences derived by ensuring that all possible transitions in the flow table are activated by the test sequence.

We feel that the checking sequences are a thorough, efficient technique for testing latches.

6. Acknowledgments

The authors would like to thank Jonathan Chang and Erin Kan for their valuable comments. This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant No. N00014-92-J-1782, in part by the Advanced Research Projects Agency under Contract No. DABT63-94-C-0045, and in part by the National Science Foundation under Grant No. MIP-9107760. It was also funded in part by Cirrus Logic.

7. References

[Al-Assadi 93] Al-Assadi, W.K. “Faulty Behavior of Storage Elements and Its Effects on Sequential Circuits”, *IEEE Transactions on VLSI,* Vol. 1, No. 4, December, 1993.

[Chandra 93] Chandra, S et. al. “CrossCheck: An Innovative Testability Solution,” *IEEE Design and Test,* pp. 56-68, June, 1993.

[Friedman 71] Friedman, A.D. and P.R. Menon, Fault Detection in Digital Circuits, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1971.

[Hawkins 89] Hawkins, CF. ”Quiescent Power Supply Current Measurement for CMOS IC Defect Detection,” *IEEE Trans. on Industrial Electronics*, pp. 211-218, May 89.

[Hennie 64] Hennie, F.C. “Fault Detecting Experiments for Sequential Circuits,” *Proc. of the Fifth Annual Switching Theory and Logical Design Symposium*, S-164, Princeton, N.J., pp. 95-110, 1964.

[Lee 90] Lee, K.J. and M.A. Breuer, “A Universal Test Sequence for CMOS Scan Registers,” *IEEE Custom Integrated Circuits Conference*, pp. 28.5.1-28.5.4, 1990.

[Makar 88] Makar, S.R., and E.J. McCluskey, "On the Testing of Multiplexers," *Proc. Int. Test Conf.*, pp. 669-679, 1988.

[Ma 95] Ma, S., P. Franco and E.J. McCluskey, “An Experimental Chip to Evaluated Test Techniques, Experimental Results,” submitted to *ITC* 1995.

[McCluskey 86] McCluskey, E.J., *Logic Design Principles*, Prentice-Hall, New Jersey, 1986.

[Perry 92] Perry, R.. “IDDQ Testing in CMOS Digital ASICs - Putting It All Together,” *Proc. Int. Test Conf.*, pp. 151-157, 1992.

[Reddy 86] Reddy, M.K.and, S.M. Reddy, “Detecting FED Stuck-Open Faults in CMOS Latches and Flip-Flops,” *IEEE Design and Test*, pp. 17-26,1986.

[Saxena 93] Saxena, N. R. et. al., “Algorithmic Synthesis of High Level Tests for Data Path Designs,” *FTCS*, pp. 360-369, 1993.

[Sucar 89] Sucar, H., “High Performance Test Generation for Accurate Defect Models in CMOS Gate Array Technology,” *ICCAD*, pp. 166-169.

Appendix A Details of the Two-State MD-Latch

The graphs of the state triples for the MD-latch are shown in Fig. A-1. The flow table is given in Table A-1. The lower bound on the length of the checking experiment is 25, since there are 24 total states. Fig. A-2 shows the relationship between the triples. In this graph, each node represents a triple from Fig. A-1. An arrow indicates that the distinguishing states of the first triple is adjacent (differ in one input variable) to the setup state of the second triple. The graph consists of two disjoint parts which implies that at least one extra pattern is needed to connect all the triples. This implies that at least one additional pattern is needed, raising the minimum length to 26.

Fig. A-3 shows how the total states corresponding to the extra patterns connect the sub-graphs. Starting with A, C, E or H would mean only one transition between the two sub-graphs. Since the first triple starts with a synchronizing input, extra patterns would be needed to apply the distinguishing setup state of the first triple. For example, suppose we start with triple E. This would make the sequence end with triple A in state 2. Then adding state 6 to the end of the sequence would require two additional inputs, raising the length to 27. If the triple used to start the sequence has a distinguishing setup state that connects the two sub-graphs, then no additional inputs are needed. The triples B, D, F and G satisfy this criteria. For example, the distinguishing setup state of triple B is state 5. This happens to be the one of the states than can connect E to A. However, starting in state B, D, F, or G requires an extra input to go back to the original sub-graph making the length 26, the minimum length. Since there are four triples that can be split, and in each case there are two choices for states between the sub-graphs (note that there aren't four, because the original starting state of the starting triple must be one of the two states between the two sub-graphs), then there are 8 minimum-length checking experiments. Since a distinguishing state is used as the setup state for all the triples, then, from Theorem 2, any sequence formed by the graph corresponds to a checking experiment. One such checking experiment was shown in Table 3.4-2 in Section 3.4.

Table A-1 Flow Table for Two-State MD-Latch.

DS

00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
C = 0								C = 1								
T = 0				T = 1				T = 0				T = 1				Q
(2)	(4)	(6)	(8)	(10)	(12)	(14)	(16)	(18)	(20)	19	21	(22)	23	25	(24)	0
[3]	[5]	[7]	[9]	[11]	[13]	[15]	[17]	18	20	[19]	[21]	22	[23]	[25]	24	1

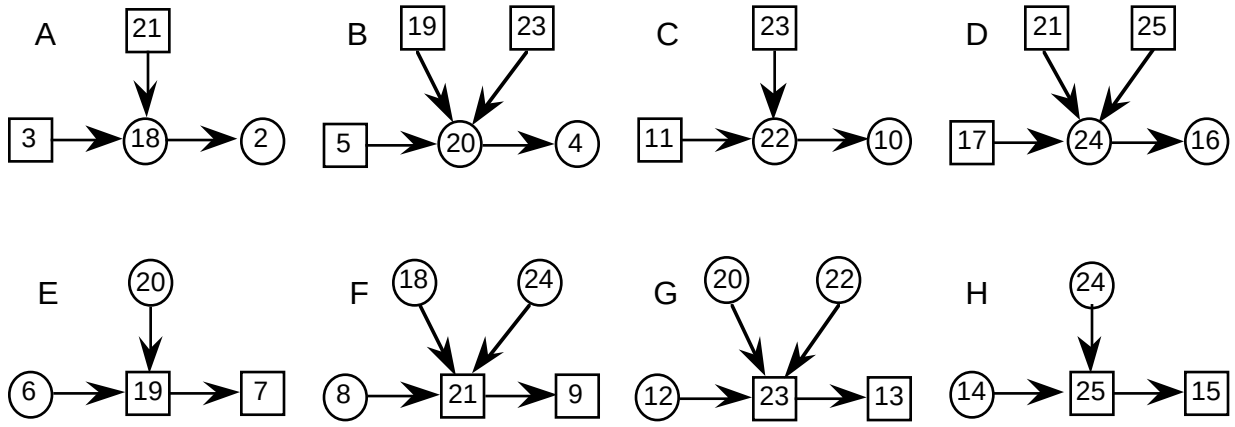


Figure A-1 Graphs of State Triples for Two-State MD-Latch.

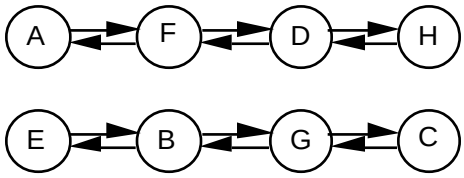


Figure A-2 Constraint Graph for State Triples.

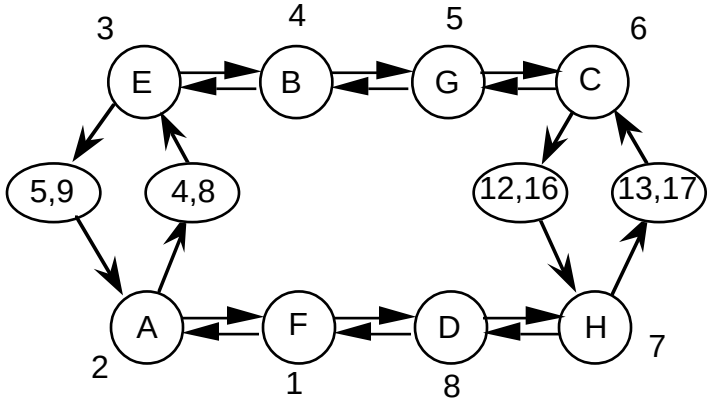


Figure A-3 Connecting the Disjoint Graph.

Appendix B Details of the Two-State Two-Port Latch

The graphs of the state triples for the Two-Port latch are shown in Fig. B-1. The flow table is shown in Table B-1. Fig. B-2 shows the relationship between the triples. A light arrow indicates that the distinguishing state of the first triple is adjacent to the setup state of the second triple. A dark arrow indicates that the distinguishing state of the first triple is the setup state of the second triple.

The lower bound on the length of the checking experiment is 19, since there are 16 total states, and states 2 and 7 appear twice as the only distinguishing states of triples. States 6 and 3

appear twice as starting states. Of course, an unstable state could be used instead as the starting state but that would still require an additional pattern. This raises the minimum length to 20. Looking at B and F, if B comes before F in the sequence, then state 5 must appear twice. If B comes after F in the sequence then state 4 must appear twice. The same argument can be applied to D and H. This raises the minimum length to 22. Now, looking at the graph in Fig. B-2, if B and F directly follow one another, and H and D directly follow one another, then it will not be possible to go through the whole graph without adding another pattern. But, if B and F do not directly follow each other, then both states 4 and 5 must appear twice. So in all cases, another pattern is needed, raising the minimum length to 23.

The connections between the nodes in Fig. B-2 form links between the distinguishing inputs. Only 3 links are needed to form a chain. The links are shown graphically in Fig. B-3. Any continuous traversal of this graph would form all links at least once. Therefore, the sequences generated are checking experiments. The order of one such sequence is marked on the graph of Fig. B-2. The actual checking experiment is shown in Table 3.5-2 in Section 3.5.

Table B-1 Flow Table for Two-State Two-Port Latch.

	D_1D_2																
	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
	$C_2 = 0$								$C_2 = 1$								
	$C_1 = 0$				$C_1 = 1$				$C_1 = 0$				$C_1 = 1$				Q
	(2)	(4)	(6)	(8)	(10)	(12)	11	13	(14)	15	17	(16)	-	-	-	-	0
	[3]	[5]	[7]	[9]	10	12	[11]	[13]	14	[15]	[17]	16	-	-	-	-	1

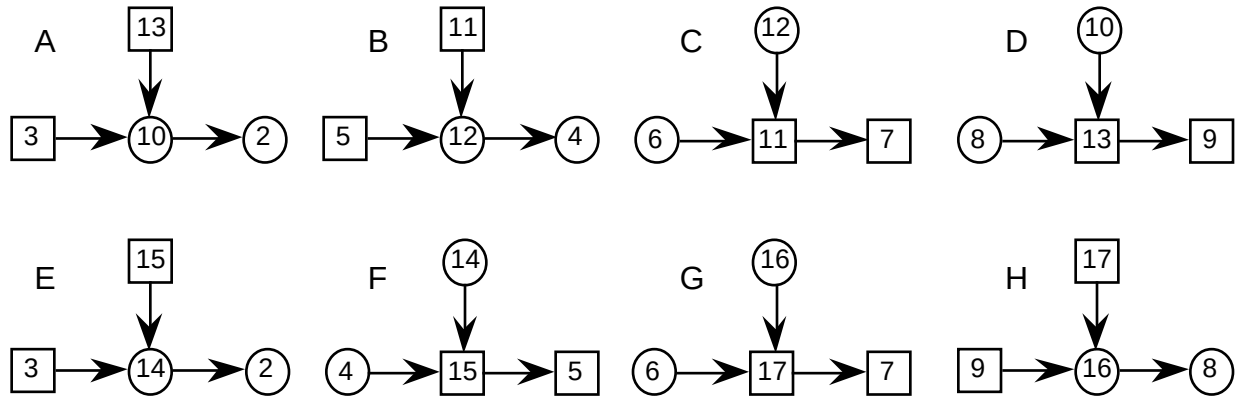


Figure B-1 Graphs of State Triples for Two-State Two-Port Latch.

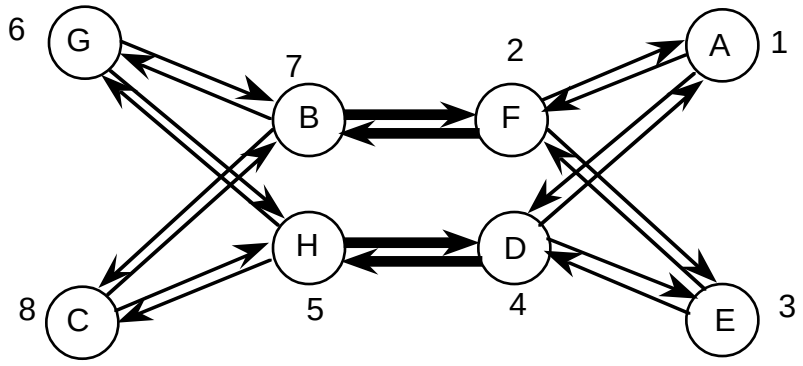


Figure B-2 Constraints on Order of Triples.

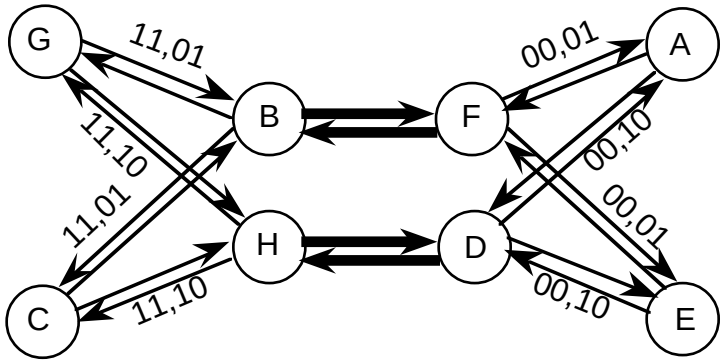


Figure B-3 Links in Triples.

Appendix C Details of the Two-State Load Enable Latch

The graphs of the state triples for the Load Enable latch are shown in Fig. C-1. The flow table is given in Table C-1. This latch is unique in that it has more distinguishing inputs than synchronizing inputs. The distinguishing states are shown graphically in Fig. C-2. In this graph, two states connected by an edge can follow each other in the sequence. The triples and the graphs of Fig. C-2 can be combined to produce minimum-length checking experiments. There are two ways to order the graphs to achieve a minimum-length checking experiment, ADBC and BCAD. Each of these orders contains two sequences. For example, either state 10 or state 8 could be used to go from Graph A to Graph D. If state 10 is selected then state 12 must be used to go from Graph D to Graph B. Similarly, there are two choices in going from Graph B to Graph C. This gives a total of four possible sequences for each graph order, and a total of eight minimum-length checking experiments. One of these checking experiments is shown in Table 3.6-2 in Section 3.6.

Since the states in Graph C are distinguishing states of all the distinguishing inputs, and since the sequences described go through all the states in Graph C before going into any other

state, the sequences described above form a chain for all the distinguishing inputs. Hence, all the sequences form a chain among the distinguishing inputs.

Table C-1 Flow Table for Two-State Load Enable Latch.

	LD							
	00	01	11	10	00	01	11	10
	C = 0				C = 1				Q
	(2)	(4)	(6)	(8)	(10)	(12)	15	(14)	0
	[3]	[5]	[7]	[9]	[11]	[13]	[15]	14	1

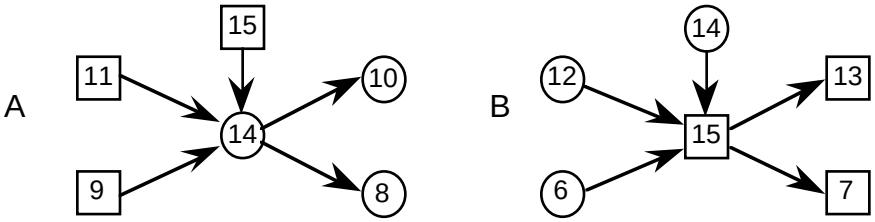


Figure C-1 Graphs of State Triples for Two-State Load Enable Latch.

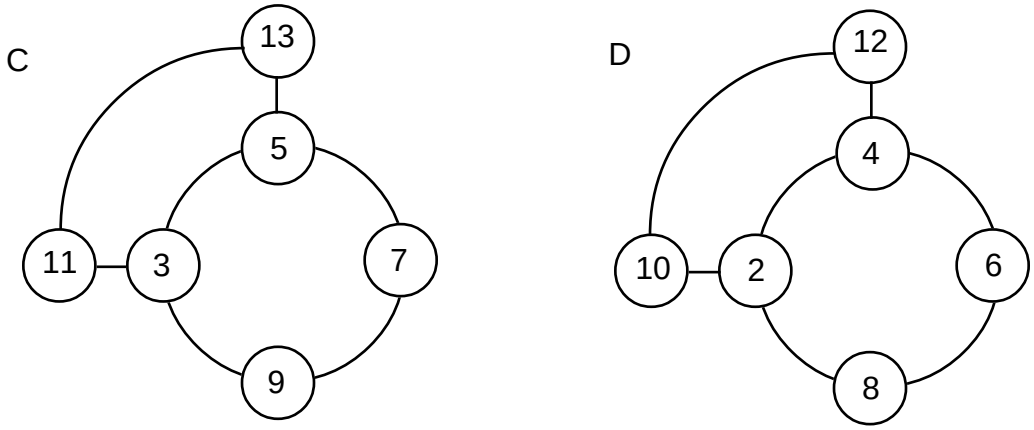


Figure C-2 Distinguishing States Adjacent States Can Follow Each Other in Sequence.

Appendix D Details of the Two-State D-Enable Latch

The graphs of the state triples for the D-Enable latch are shown in Fig. D-1. The flow table is given in Table D-1. The lower bound on the length of the checking experiment is 15, since there are 12 states, there are two more unstable states in one row than the other, and the latch is a single-input control state machine. The only way to state 3, the setup state of triple A,

is to go through state 7 followed by either state 5 or state 9. The situation is shown in Fig. D-2. Suppose state 5 is picked. To enter state 9, either state 7 must be entered once again, or state 9 is followed by state 3, making state 3 be entered again. In either case one more pattern is needed raising the minimum length to 16.

Fig. D-3 shows the constraint between the state triples. One possible sequence of triples is BCDA. This sequence is shown in Fig. 3.7-2 of Section 3.7, and is of minimum length since it has 16 patterns.

Table D-1 Flow Table for D-Enable Latch.

DE								
00	01	11	10	00	01	11	10	
C = 0				C = 1				Q
(2)	(4)	(6)	(8)	(10)	(12)	11	(14)	0
[3]	[5]	[7]	[9]	10	12	[11]	14	1

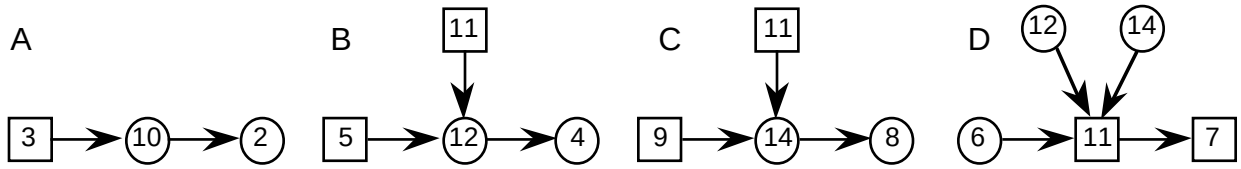


Figure D-1 Graphs of State Triples for D-Enable Latch.

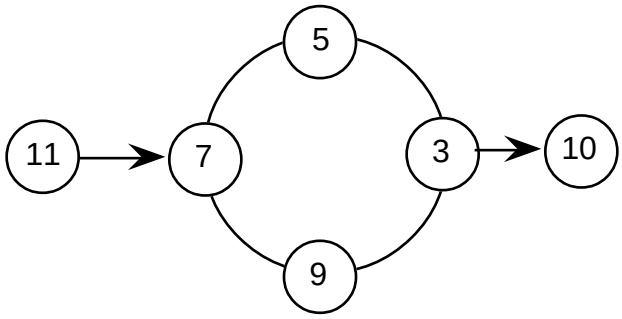


Figure D-2 Constraints on States.

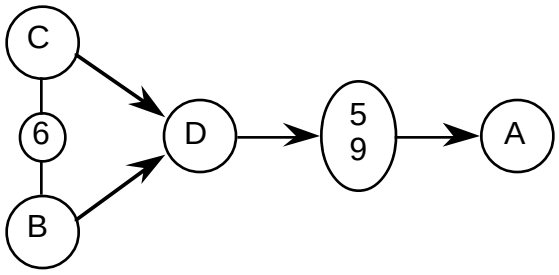


Figure D-3 Constraint Graph for State Triples.

Appendix E Details of the Two-State BILBO Latch

The graphs of the state triples for the BILBO latch are shown in Fig. E-1. The flow table is given in Table E-1. The lower bound on the length of the checking experiment is 55, since there are 48 total states and there are four more unstable states in the lower row of the flow table than in the upper one. Fig. E-2 show the relationship between the triples. The graph contains four disconnected subgraphs. The smallest subgraph contains only triple F. Fig. E-3 shows how the F subgraph can be joined to the other subgraphs. Since F connects to the middle of two other subgraphs, the subgraphs would need to be split. Each split would result in the addition of two patterns. To avoid splitting both subgraphs, which would result in 59 patterns, F should either be the first or last triple.

Table E-1 Flow Table for Two-State BILBO Latch.

DS (C = 0)

00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	Q
$B_2 = 0$								$B_2 = 1$								
$B_1 = 0$				$B_1 = 1$				$B_1 = 0$				$B_1 = 1$				
(2)	(4)	(6)	(8)	(10)	(12)	(14)	(16)	(18)	(20)	(22)	(24)	(26)	(28)	(30)	(32)	0
[3]	[5]	[7]	[9]	[11]	[13]	[15]	[17]	[19]	[21]	[23]	[25]	[27]	[29]	[31]	[33]	1

DS (C = 1)

00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	Q
$B_2 = 0$								$B_2 = 1$								
$B_1 = 0$				$B_1 = 1$				$B_1 = 0$				$B_1 = 1$				
35	(34)	(36)	37	(38)	(40)	(42)	(44)	39	(46)	41	(48)	(50)	(52)	43	45	0
[35]	34	36	[37]	38	40	42	44	[39]	46	[41]	48	50	52	[43]	[45]	1

If F were the first triple, state 13 would need to be preceded by one of the following sequences: 35,3,5; 35,3,11; 43,31,15; 43,31,29. The first pattern in each of these sequences is accounted for in the bound by the initialization factor. The second two patterns in the sequence don't account for any of the other triples, so they will add to the minimum length. So the minimum length becomes 57. The distinguishing state of triple F is state 12. This state is not adjacent to the setup state of any triple. Therefore, an additional pattern is needed raising the minimum to 58. Now, suppose that triple F is in the last triple. Again 58 patterns are needed. Fig. E-4 shows the setup of one possible sequence. The actual sequence is shown in Table 3.9-2 in Section 3.9.

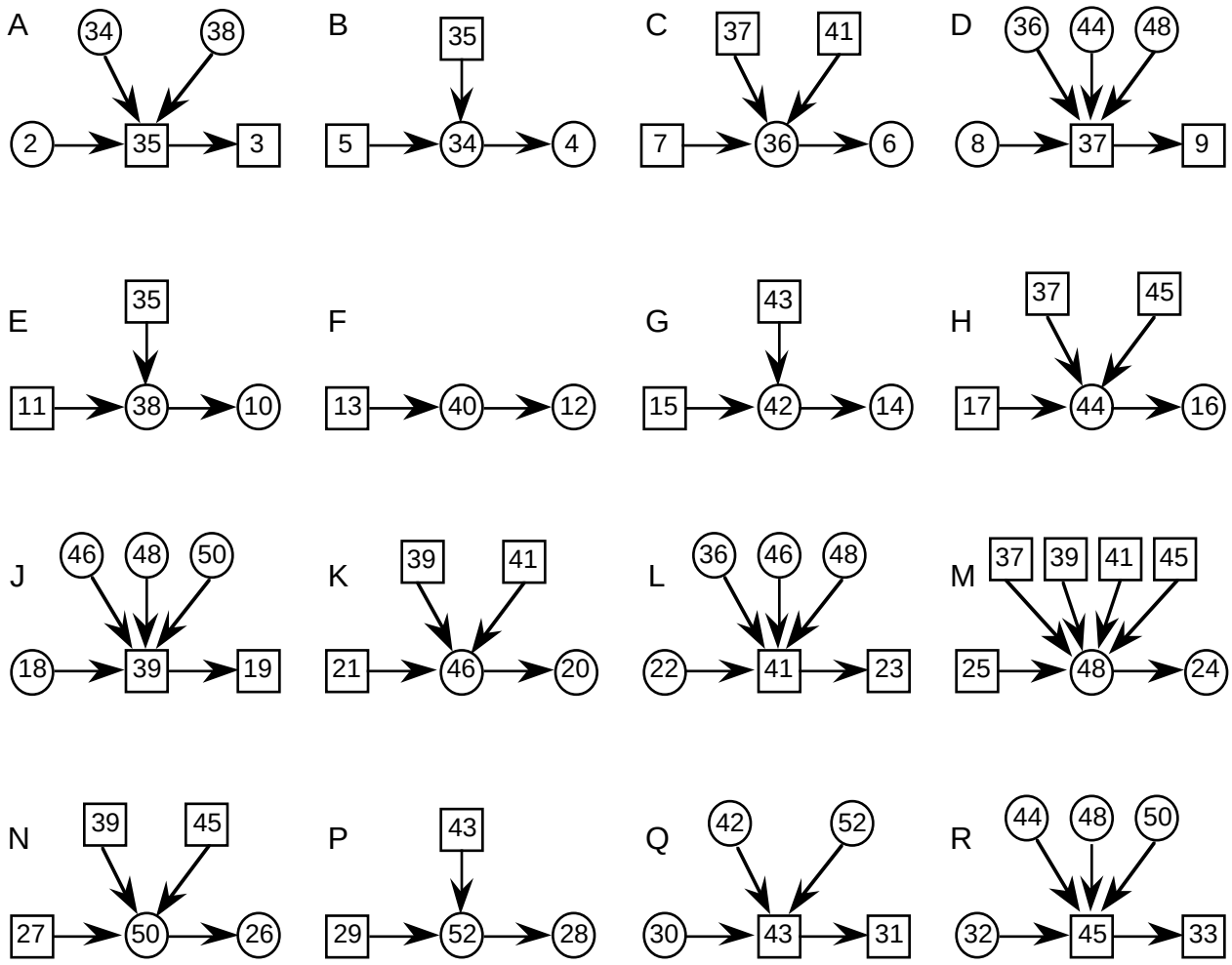


Figure E-1 Graphs of State Triples for Two-State BILBO Latch.

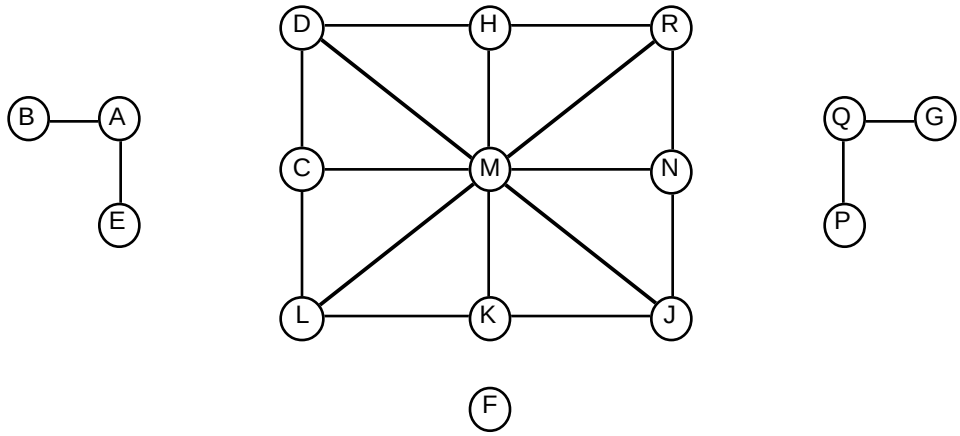


Figure E-2 Constraint Graph for State Triples.

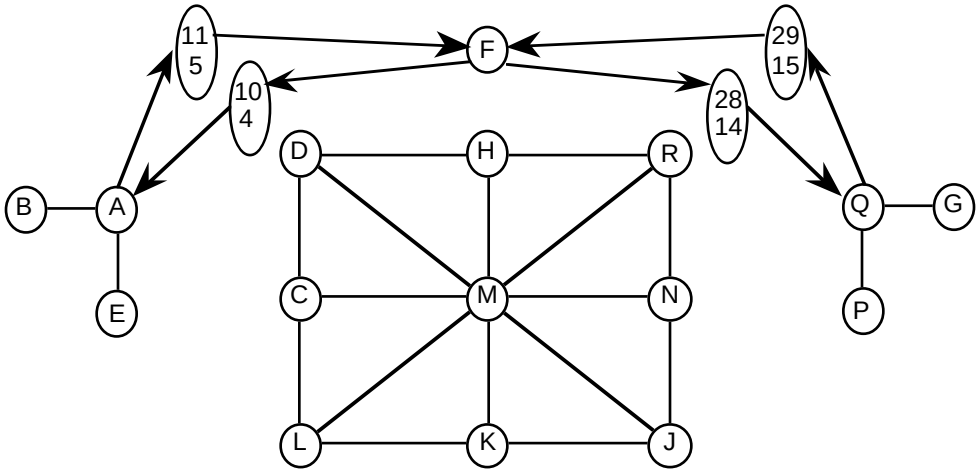


Figure E-3 Connection of Triple F to Other Triples.

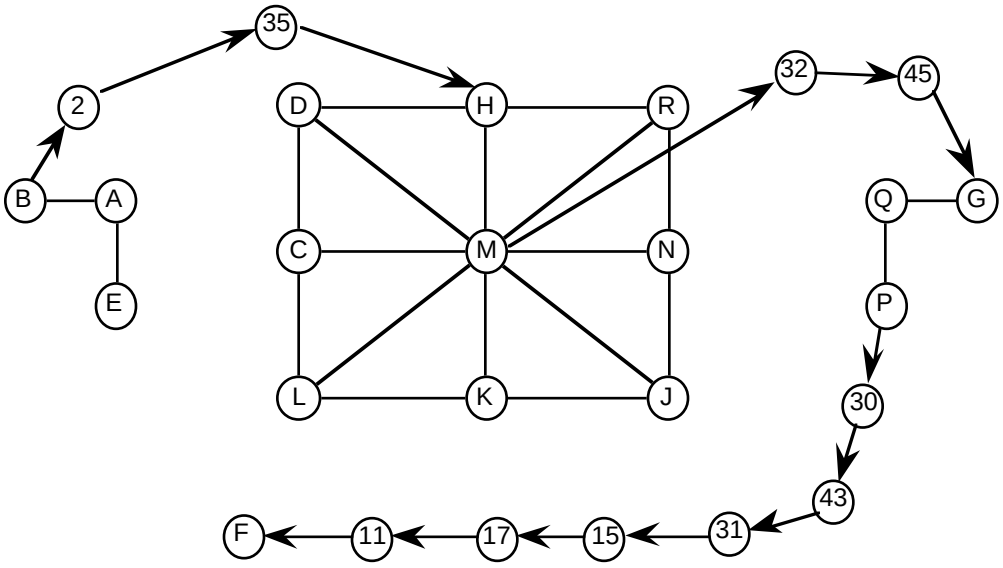


Figure E-4 Connecting the Disjoint Graph

Appendix F Details of the Two-State CBILBO Latch

The graphs of the state triples of the CBILBO latch are shown in Fig. F-1. The flow table is given in Table F-1. The lower bound on the length of the checking experiment is 25, since there are 24 states. Fig. F-2 shows the relationship between the triples. The graph is split into two disjoint sub-graphs. Fig. F-3 shows how the two sub-graphs can be connected with one additional state in the sequence. The minimum-length checking experiment can be achieved if the state used to connect the two disjoint graphs is a distinguishing setup state of the starting triple. This can be done if triple G (distinguishing setup state is 14), H (distinguishing setup state is 17), F (distinguishing setup state is 12) or E (distinguishing setup state is 11). The minimum-length checking experiment starting with triple G is shown in Table 3.10-2 of Section 3.10. From Theorem 2, a sequence derived from the graph in Fig. F-3 forms a chain among all the distinguishing inputs.

Table F-1 Flow Table for Two-State CBILBO Latch.

DS	00	01	11	10	00	01	11	10	00	01	11	10	00	01	11	10	
	C = 0								C = 1								
	B_1 = 0				B_1 = 1				B_1 = 0				B_1 = 1				Q
	(2)	(4)	(6)	(8)	(10)	(12)	(14)	(16)	(18)	19	(20)	21	(22)	23	25	(24)	0
	[3]	[5]	[7]	[9]	[11]	[13]	[15]	[17]	18	[19]	20	[21]	22	[23]	[25]	24	1

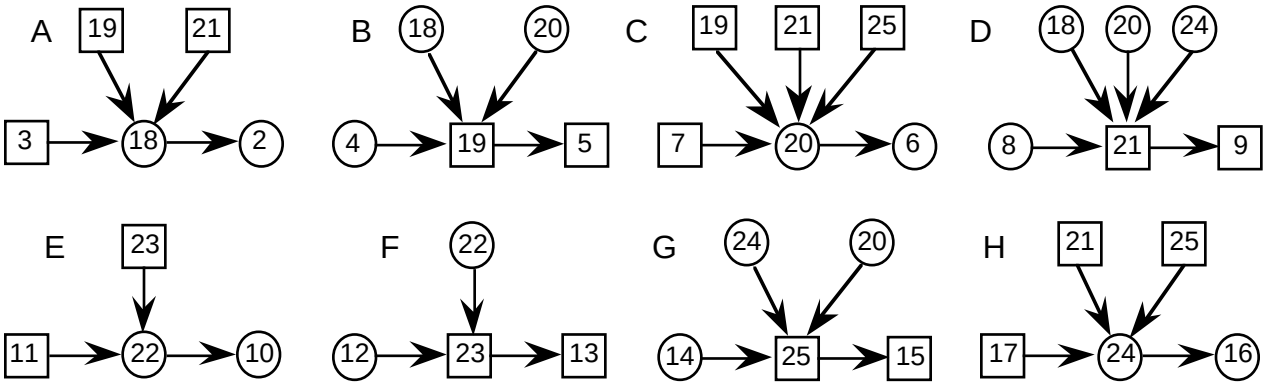


Figure F-1 Graphs of State Triples for Two-State CBILBO Latch.

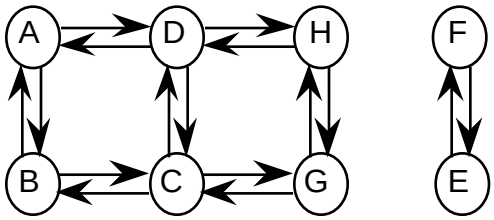


Figure F-2 Constraint Graph for State Triples.

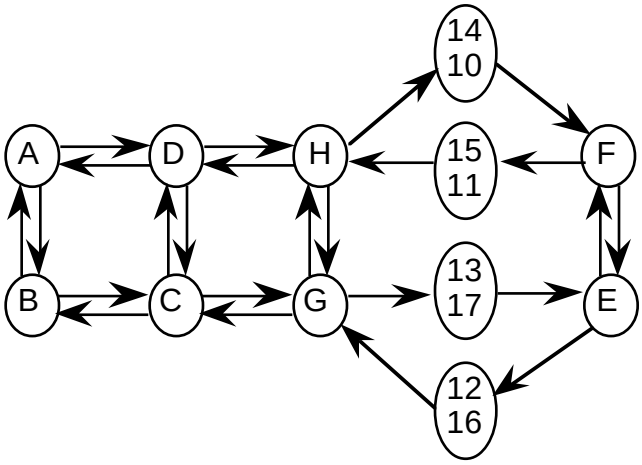


Figure F-3 Connecting the Disjoint Graph.

Appendix B. Deriving Checking Experiment for Flip-Flops

This Appendix includes detailed information on deriving checking experiments for an MD flip-flop and a TP flip-flop. We use the transition graphs from Chapter 3 to generate these checking experiments. Section B.1 covers the MD flip-flop, and Section B.2 covers the TP flip-flop.

B.1 Checking Experiment for an MD Flip-flop

In Section 3.1.2, we showed that 137 transitions (which implies 138 patterns) are needed for a checking experiment for an MD flip-flop. In this section we show how a checking experiment of this length can be derived. Figs. B.1-1 through B.1-4 show all the transitions in the primitive flow table. The graphs are not disjoint, since some states appear in multiple graphs. Our goal is to find a minimum-length sequence that includes all transitions in these graphs. The graphs of Fig. B.1-1 through B.1-3 are Euleran and a state sequence that would apply all transitions only once can be easily derived. The graphs in Fig. B.1-4 are not Euleran and some transitions will need to be repeated. For example, graph A in Fig. B.1-4 has state 18 with an incoming edge and no outgoing edge,

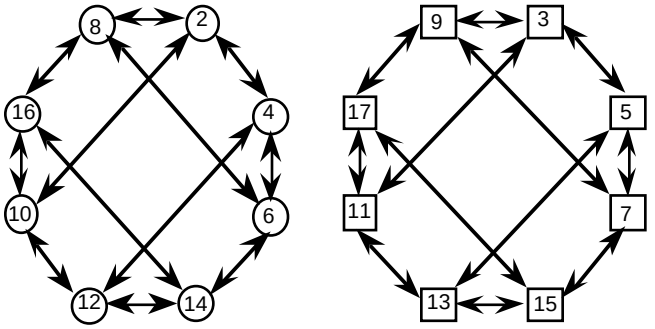


Figure B.1-1 Graphs of Transitions Within First Quadrant for MD Flip-Flop.

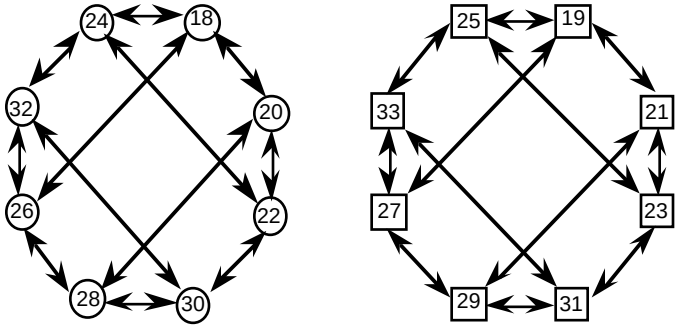


Figure B.1-2 Graphs of Transitions Within Fourth Quadrant for MD Flip-Flop.

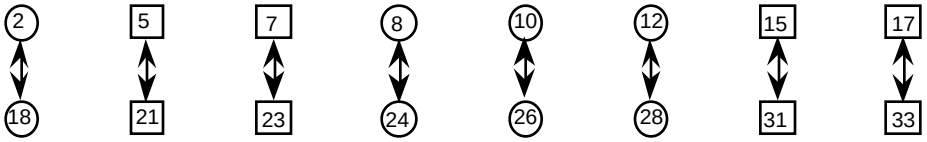


Figure B.1-3 Graphs of Balanced Transitions Between Quadrants.

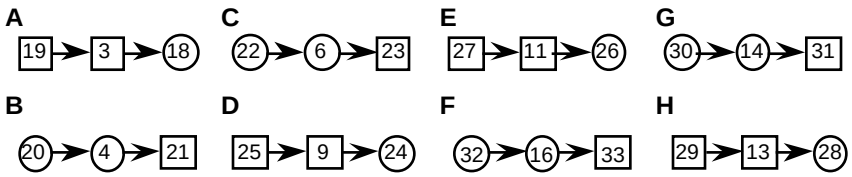


Figure B.1-4 Graphs of Unbalanced Transitions Between Quadrants.

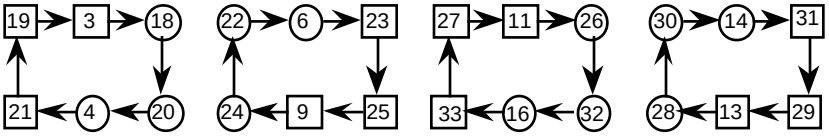


Figure B.1-5 Graphs of B.1-4 With Extra Transitions to Become Euleran.

and graph B has state 20 with an outgoing edge and no incoming edge. Since an edge from state 18 to state 20 is allowed (see the first graph in Fig. B.1-2), we can add a transition between the two states, and connect the two graphs. The same argument can be applied to state 21 and 19, and we can add a transition from state 21 to state 19. The new graphs are shown in Fig. B.1-5. Vertical edges are used to indicate the added transitions. These graphs are Euleran and we can derive a sequence that would apply all transitions only once. The graphs from Fig. B.1-1, B.1-2, B.1-3 and B.1-4 can be combined to form one big Euleran graph, and a sequence can be derived from it. A simpler approach is to derive a sequence for each graph and combine the sequence. The graphs suggest three types of sequences:

A - sequence of states within a quadrant (Fig. B.1-1, Fig B.1-2) (length 24)

B - sequences of states in Fig. B.1-5 (length 6)

C - balanced sequences in Fig. B.1-3 (length 2)

Starting with State 18, Fig. B.1-6 shows how the sequences can be combined to form a complete minimum-length checking experiment. States are picked horizontally until an "interrupter", shown as |, is reached. The sequence at the interrupter is then picked, and then we go back to the original sequence. For example, we start with 18, follow that with 20. Then we hit the first interrupter, so we pick the sequence at the bottom of the figure, 22,30....20. Note that this sequence ends with the same state as the

state just before the interrupter, so we can continue with the original sequence. In this case the next state will be 4. The checking experiment derived from Fig. B.1-6 is shown in Table B.1-1.

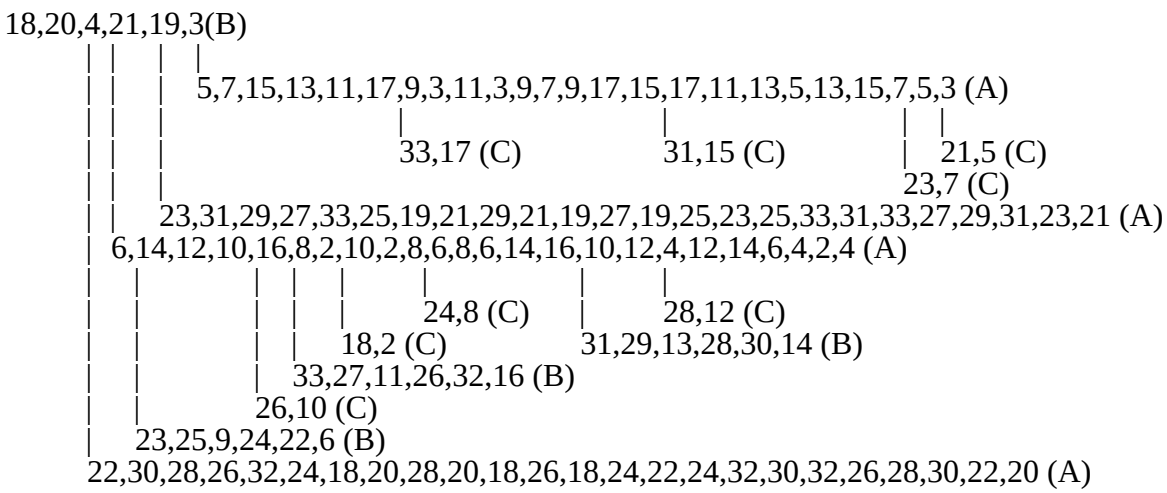


Figure B.1-6 Combining Sub-Sequences of Graphs.

Table B.1-1 Example of Minimum-Length (138) Checking Experiment for MD Flip-Flop.

C	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		1	1	1	1	1	1	1	1	1	1	0	0
T	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0
S	0	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	1	0	0	1
D	0	0	1	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1	0	0	1	0	0	1	1	1	1	1	1
Q		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State	-	18	20	22	30	28	26	32	24	18	20	28	20	18	26	18	24	22	24	32	30	32	26	28	30	22	20	4	6
C	1	1	0	1	1	0	0	0	0	1	0	0	1	1	0	1	1	0	0	0	1	0	0	0	0	1	0	0	0
T	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0
S	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1
D	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Q	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State	23	25	9	24	22	6	14	12	10	26	10	16	33	27	11	26	32	16	8	2	18	2	10	2	8	24	8	6	8
C	0	0	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
T	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0	0	0	0	1	1	1	1	0	0	0	0
S	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	1	1	0	0	0	0	1	1	0	0	1	1	0	0
D	0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	0	1
Q	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
State	16	14	31	29	13	28	30	14	16	10	12	4	12	28	12	14	6	4	2	4	21	23	31	29	27	33	25	19	21
C	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0
T	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1
S	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	0	0	0	1	1	0	0	1	1	1	1	0	0
D	1	1	0	0	0	0	1	0	0	1	0	0	1	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0
Q	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
State	29	21	19	27	19	25	23	25	33	31	33	27	29	31	23	21	19	3	5	7	15	13	11	17	33	17	9	3	11
C	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1							
T	0	0	0	0	1	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0							
S	0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0							
D	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0							
Q	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0							
State	3	9	7	9	17	15	31	15	17	11	13	5	13	15	7	23	7	5	21	5	3	18							

B.2 Checking Experiment for a TP Flip-flop

In Section 3.1.2, we showed that 89 transitions (which implies 90 patterns) are needed for a checking experiment for a TP flip-flop. In this section we show how a checking experiment of this length can be derived. Figs. B.2-1 through B.2-3 show all the transitions in the primitive flow table. These graphs are not disjoint, since some states appear in multiple graphs. As with the MD flip-flop in Section B.1, our goal is to find a minimum-length sequence that includes all transitions in these graphs. The graphs of Fig. B.2-1 and B.2-2 are Euleran and a state sequence that would apply all transitions only once can be easily derived. The graphs in Fig. B.2-3 are not Euleran and some transitions will need to be repeated. For example, graph A in Fig. B.2-3 has state 18 with an incoming edge and no outgoing edge, and graph B has state 20 with an outgoing edge and

no incoming edge. Since an edge from state 18 to state 20 is allowed (see Fig. B.2-1), we can add a transition between the two states, and connect the two graphs. The same argument can be applied to state 21 and 19, and we can add a transition from state 21 to state 19. The new graphs are shown in Fig. B.2-4. Vertical edges are used to indicate the added transitions. These graphs are Euleran and we can derive a sequence that would apply all transitions only once. The graphs from Fig. B.2-1, and B.2-4 can be combined to form one big Euleran graph, and a sequence can be derived from it. A simpler approach is to derive a sequence for each graph and then combine the sequence. The graphs suggest three types of sequences:

A - sequence of states within a quadrant (Fig. B.2-1) (length 8)

B - sequences of states in Fig. B.2-4 (length 6)

C - balanced sequences in Fig. B.2-2 (length 2)

Starting with State 18, Fig. B.2-4 shows how the sub-sequences can be combined to form a complete minimum-length checking experiment. States are picked horizontally until an "interrupter", shown as |, is reached. The sequence at the interrupter is then picked, and then we go back to the original sequence. For example, we start with 18, follow that with 20. Then we hit the first interrupter, so we pick the sequence at the bottom of the figure, 22,24....20. Note that this sequence ends with the same state as the state just before the interrupter, so we can continue back with the sequence. In this case the next state will be 4. The actual checking experiment is shown in Table B.2-1.

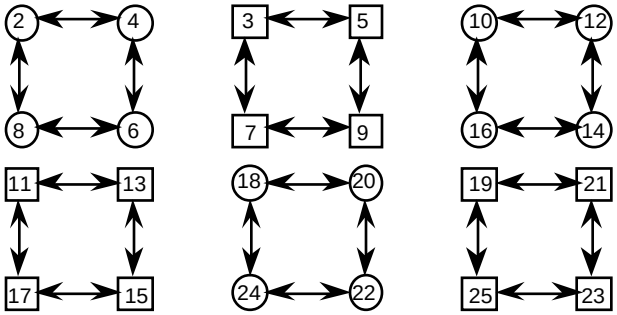


Figure B.2-1 Graphs of Transitions Within Quadrants for TP Flip-Flop.

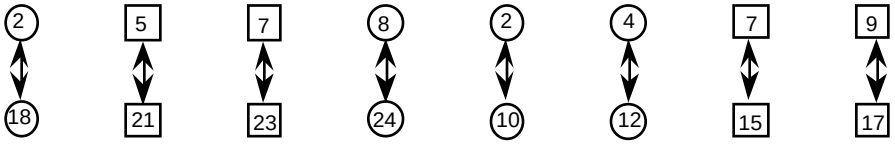


Figure B.2-2 Graphs of Balance Transitions Between Quadrants for TP Flip-Flop.

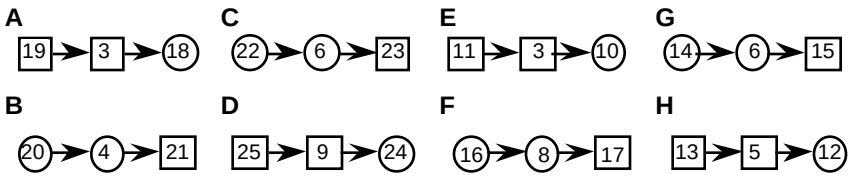


Figure B.2-3 Graphs of Unbalanced Transitions Between Quadrants for TP Flip-Flop.

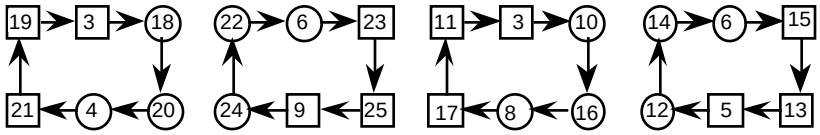


Figure B.2-4 Graphs of B.2-3 With Extra Transitions to Become Euleran.

```
18,20,4,21,19,3 (B)
      | |     |     |
      | |     |     5,7,9,3,9,7,5,3 (A)
      | |     |       |  |      |     |
      | |     |       |  |      |     19,3 (C)
      | |     |       |  |      23,7 (C) |
      | |     |       |  25,9 (C)  |     11,3 (C)
      | |     |       |         |      15,7 (C)
      | |     |       |        17,9 (C)
      | |     |       21,5 (C)
      | |     |            |
      | |     |           13,5(C)
      | |     21,23,25,19,25,23,21,19 (A)
      | |          |
      | |          25,9,24,22,6,23 (B)
      | |                        |
      | |                        15,13,5,12,14,6 (B)
      | 6,8,2,4,2,8,6,4 (A)
      |     |
      |     17,11,3,10,16,8 (B)
      |        |         |
      |        |         12,14,16,10,16,14,12,10 (A)
      |        11,13,15,17,11,17,15,13 (A)
      22,24,18,20,18,24,22,20 (A)
```

Figure B.2-5 Combining Sub-Sequences of Graphs.

Table B.2-1 Example of Minimum-Length (90) Checking Experiment for TP Flip-Flop.

C_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
D_1	0	0	0	1	1	0	0	0	1	1	0	0	1	1	1	0	0	1	1	0	1	1	0	0	0	0	0	1	1	0
C_2	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D_2	0	0	1	1	0	0	1	0	0	1	1	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0
Q		0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
State	-	18	20	22	24	18	20	18	24	22	20	4	6	8	17	11	13	15	17	11	17	15	13	11	3	10	12	14	16	10
C_1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0
D_1	1	1	0	0	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0
C_2	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	1	1	1
D_2	0	1	1	0	0	0	0	1	0	0	1	1	1	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	0	0
Q	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
State	16	14	12	10	16	8	2	4	2	8	6	4	21	19	21	23	25	9	24	22	6	15	13	5	12	14	6	23	25	19
C_1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
D_1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0
C_2	1	1	1	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1
D_2	0	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0
Q	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
State	25	23	21	19	3	5	21	5	13	5	7	9	25	9	17	9	3	9	7	23	7	15	7	5	3	19	3	11	3	18

Appendix C. Fault Analysis

In this appendix, we present a detailed analysis of some of the fault simulation results presented in Chapter 5.

C.1 Analysis of Faults in D-Latch

As mentioned in Chapter 5, some faults are only detected by current measurement. Fig. C.1-1 shows the faults in the D-latch that are detected by current, and not detected by boolean test. In the D-latch, there are three kinds of faults that are detected by current and not by voltage. A stuck-on fault in M8 or M9 would cause a large IDDQ current when the corresponding NMOS transistor is turned on. However, due to the sizing ratio between the two transistors, the output voltage value would not be affected enough to be detected. In fact, the inverter would behave as an NMOS inverter rather than a CMOS inverter. A plot of the relevant voltages and currents in the circuit with a stuck-on fault on M8 are shown in Fig. C.1-2. In these graphs, $\overline{Q}$ has a fault-free value of 0 volts between 10 and 40 ms. With the presence of the fault, this voltage is around 0.1 volt, which is not high enough to be detected by a voltage test. In the same time frame, the IDDQ current reaches 2.5 mA. The second kind of faults that are only detected by current measurements are the stuck-open faults on M6 and M1, and stuck-open faults on the transmission gate in the feedback path. These faults cause voltage degradation on N1 which in turn causes M8 or M4 to turn on when they should not, raising the IDDQ current. A plot of the relevant voltages and currents in the circuit with stuck-open fault on M1 are shown in Fig. C.1-3. For the circuit with a stuck-open fault on M5 , the voltages and currents are shown in Fig. C.1-4. In this case, the current reaches around 50 uA. Even though this value is much lower than that of M8 stuck-on fault, it is still above the threshold of 32 uA. The third kind of faults are shorted interconnects. These faults cause the voltage output to change, but not far enough to be detected as a change in boolean value. For example, a short between Q and $\overline{Q}$ makes the output become about 2.5 volts. A very high current is observed with these faults

Some faults can only be detected by voltage measurements (see Fig. C.1-5). Stuck-open faults on M3, M4, M8, and M9 do not substantially increase the IDDQ current. Also, a STI fault between D and N1 does not increase the IDDQ current.

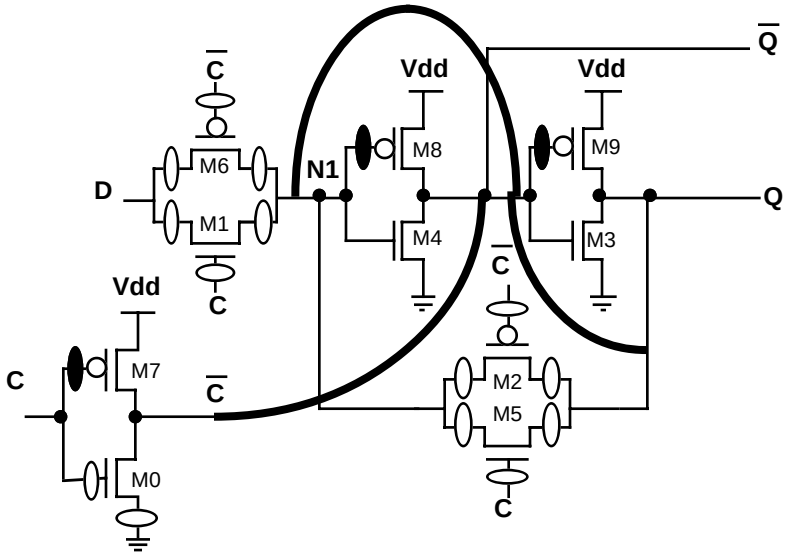
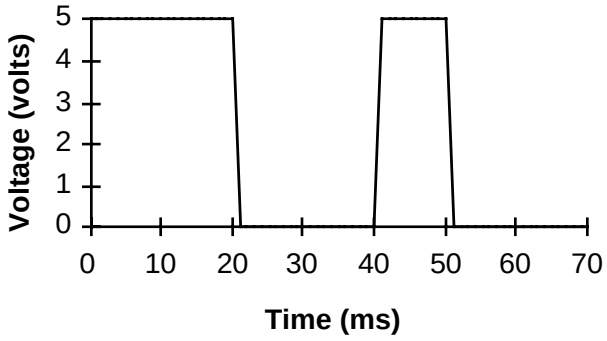
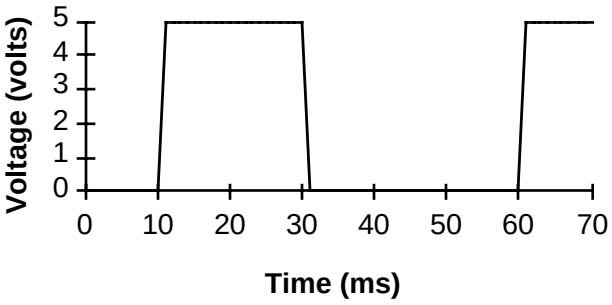


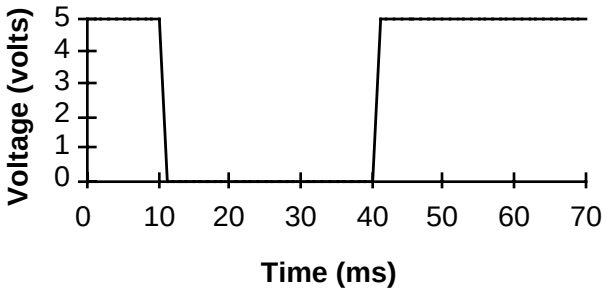
Figure C.1-1 Faults Detected by IDDQ Test Missed by Boolean Test.



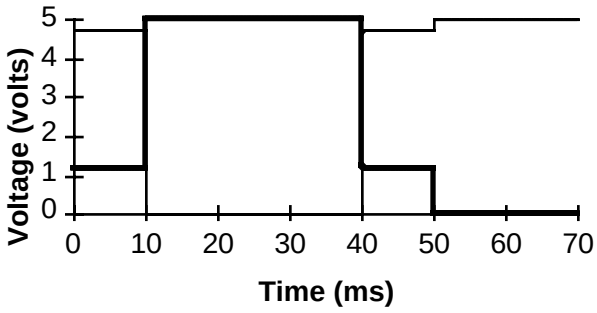
(a) C Input



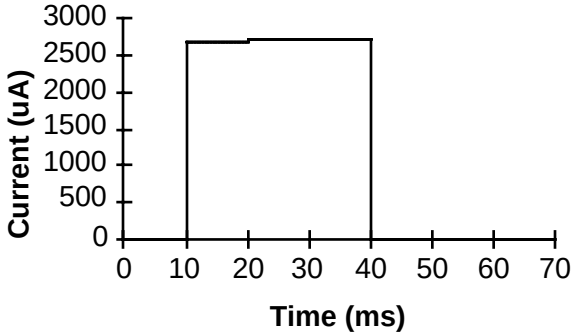
(b) D Input



(c) $\overline{Q}$ Fault-Free Output

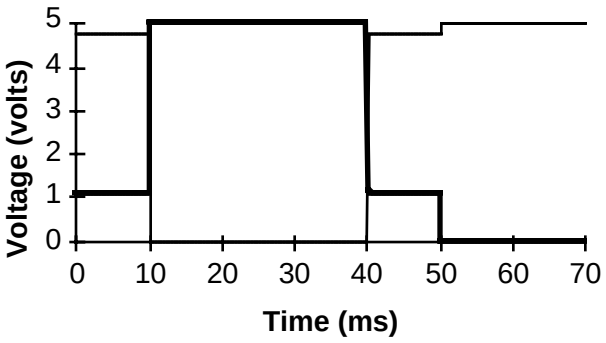


(d) N1 (Thick Line) and $\overline{Q}$ in Faulty Circuit

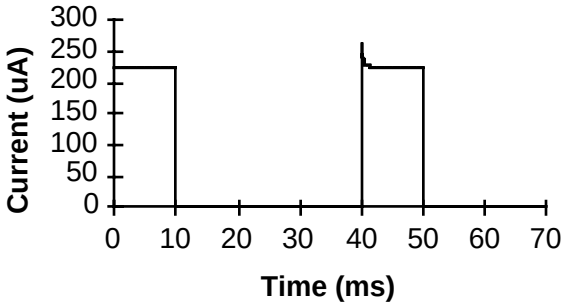


(e) IDD in Faulty Circuit

Figure C.1-2 HSpice Output for M8 Stuck-On.

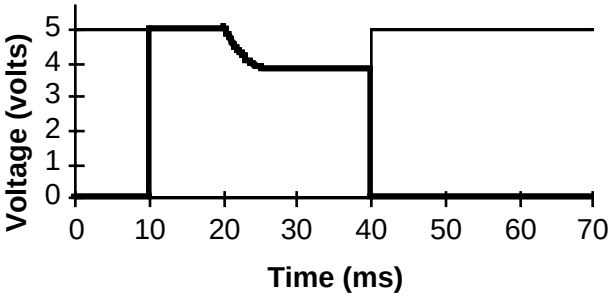


(a) N1 (Thick Line) and $\overline{Q}$ in Faulty Circuit

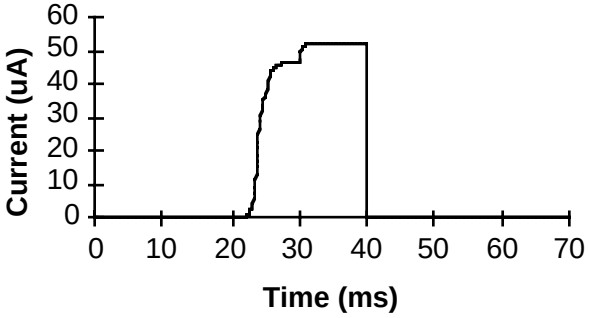


(b) IDD in Faulty Circuit

Figure C.1-3 HSpice Output for M1 Stuck-Open.



(a) N1 (Thick Line) and $\overline{Q}$ in Faulty Circuit



(b) IDD in Faulty Circuit

Figure C.1-4 HSpice Output for M5 Stuck-Open.

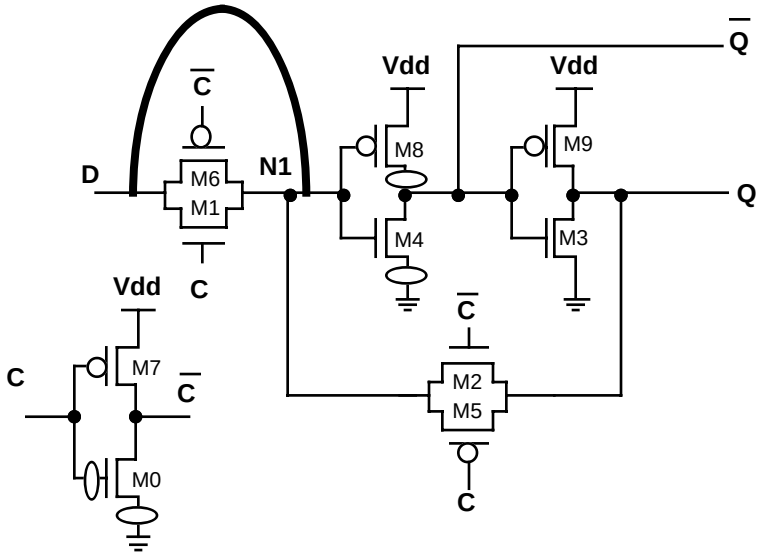


Fig. C.1-5 Faults Detected by Boolean Test Missed by IDDQ Test.

C.2 Analysis of Faults in D Flip-Flop

The analysis in this section is divided into four parts. In the first part, we analyze the faults that were missed by the checking experiment. We show that these faults would have been detected if they were detectable. In the second part, we look at the faults missed by boolean measurement, and detected by IDDQ measurements. Here, we show that these faults could not have been detected by any boolean test, and thus can only be detected by IDDQ measurement. It may be important to detect such faults because they are likely to affect the circuit reliability. In the third part, we analyze a fault that is not detected by a 10 ms cycle test, but detected by a 100 ns cycle test, and in the last part, we analyze faults missed by a 100 ns cycle test, but detected by a 10 ms cycle test.

1. Faults Missed by Checking Experiment

The checking experiment for the D Flip-Flop missed 15 faults. These faults are shown in Fig. C.2-1. As before, white ovals indicate SOP or OPI faults, black ovals indicate SON faults, and thick black lines indicate STI faults. These faults can be categorized into four groups based on the circuit behavior in the presence of the fault.

STI's N4-N11, N6-N8

Looking at N4-N11, when CK1 = 1, neither node involved is sensitized to the output. When CK1 = 0, CKN = 1, N4 and N11 will have the same voltage. Therefore, we cannot have a difference in the voltage output. Now, depending on the value of Q, either M14 or M8 will be off. Therefore, the fault cannot

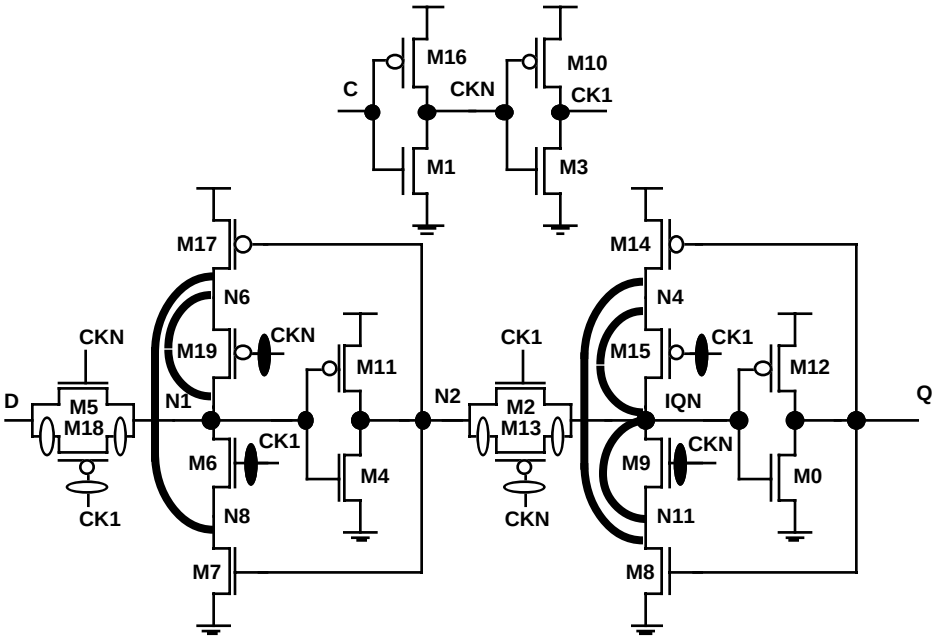


Figure C.2-1 Faults Missed by Checking Experiment.

cause a direct path from VDD to ground, making detection by IDDQ testing impossible. Therefore, the fault is undetectable. A similar argument can be made for N6-N8.

All the SON faults

Consider M15 SON. In the fault-free circuit, if IQN = 1, then Q = 0, turning on M14 and making N4 = 1. Since IQN and N4 have the same voltage in the fault-free circuit, then the fault cannot be detected when IQN = 1. Now if IQN = 0, Q = 1, and M14 is turned off, leaving N4 with no drive at all. The fault would cause N4 to discharge to 0 in a short time. Thus the boolean output would not be affected, and IDDQ would remain low. The other SON faults are similarly undetectable.

Rest of STI's (N4-IQN, N11-IQN, N1-N6)

The same analysis used for the SON faults can be used for these STI faults. However, N8-N1 is actually detected, while M6 SON is not. The reason for this is that when a transistor is stuck-on there is some small resistance between the source and the drain. However, the STI fault contains no resistance (actually it has a resistance of 0.001 ohms for HSpice to work). This fault gets detected when N1 changes from 0 to 1. At that point, N2 changes from 1 to 0. Initially, when N2 was 1, M7 was on, making N8 = 0. After N1 changes to 1,

we end up with a conflict in values between N1 and N8. This voltage difference resolves to about 1.78 volts. This voltage is low enough to keep N2 from falling, preventing M7 from going off and thus N1 remains low. So, to detect faults in this category we need to have a transition from 0 to 1 or from 1 to 0 depending on the location of the fault. Since a checking experiment guarantees this, then any checking experiment would detect these faults if they are detectable.

All the SOP faults

These faults actually do affect the IDDQ current (13X for faults at M13 and 126X for faults at M18). However, they fall well below the threshold that we have set. Lowering the threshold would detect them. However, a lower threshold could cause a loss of yield. The reason these faults increase the current is that they cause voltage degradation of N1 or IQN, partly turning on both transistors of the downstream inverters.

2. Faults Missed by Boolean Testing Detected by IDDQ Testing

Boolean testing missed 16 faults that were detected by IDDQ testing. These faults are shown in Fig. C.2-2. As before, white ovals indicate SOP or OPI faults, black ovals indicate SON faults, and thick black lines indicate STI faults. STP faults are shown as thick black lines connected to VDD. These faults can be categorized into six groups

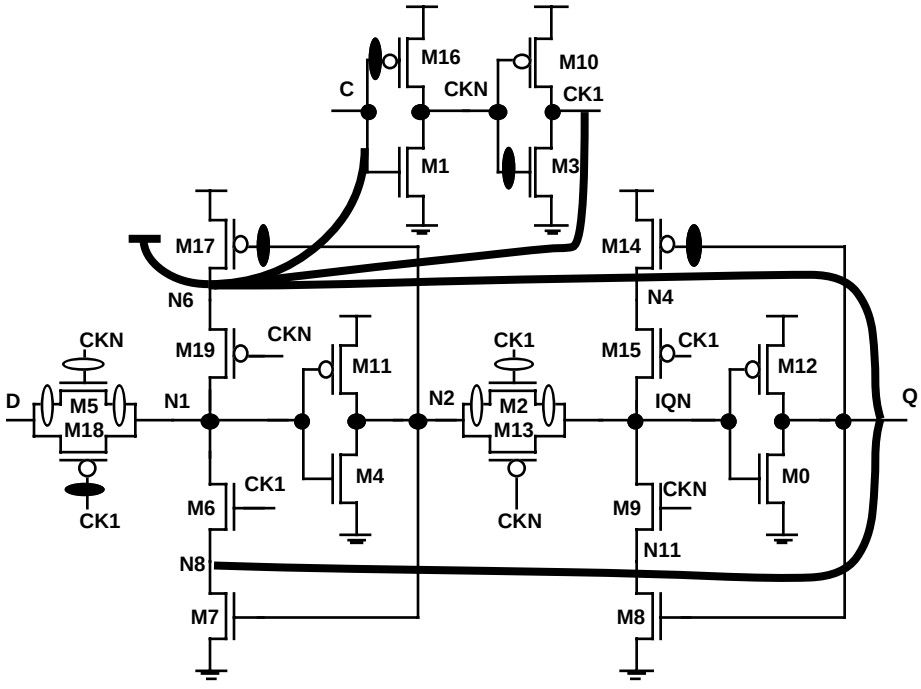
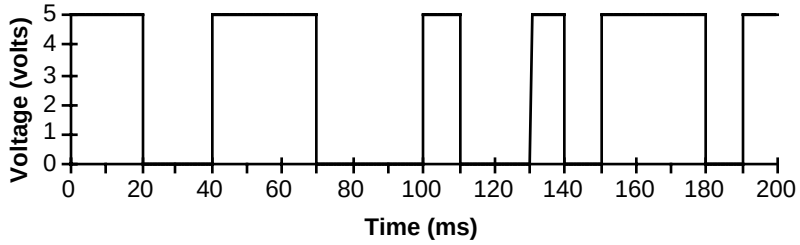


Figure C.2-2 Faults Missed by Boolean Testing Detected by IDDQ Testing.

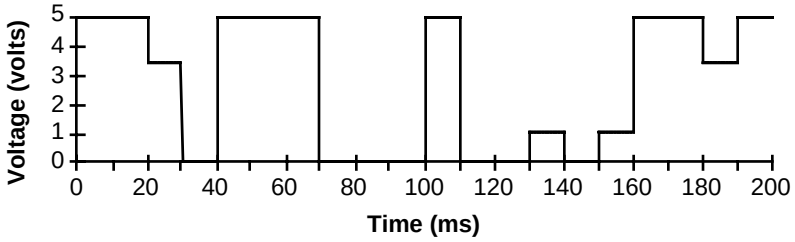
based on the circuit behavior in presence of the fault.

M18 SON

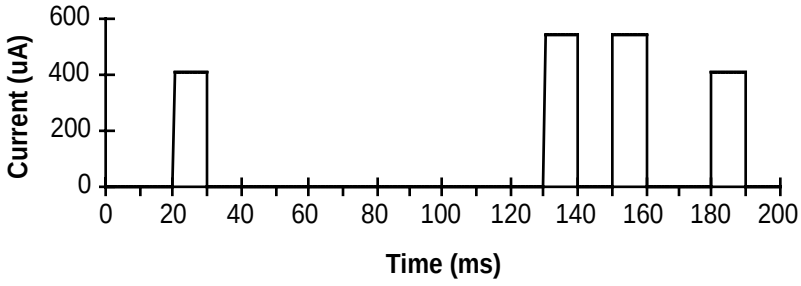
In the fault-free circuit, when C = 1 and D changes value, N1 should not change. However, in the faulty circuit, N1 becomes 3.4 volts when it should be 5 volts. This happens between 20 and 30 ms and between 180 and 190 ms (see Fig. C.2-3). The voltage degradation causes M11 and M24 to partially turn on, leading to an IDDQ current of 400 uA. Similarly N1 becomes 1.1 volts between 130 and 140 ms and between 150 and 160 ms (see Fig. C.2-3) when it should be holding 0 volts. In this case the IDDQ current becomes 540 uA. Even though the voltage at N1 is degraded, the voltage is adjusted at N2, and thus the fault cannot be detected by boolean testing.



(a) D Input



(b) N1 in Faulty Circuit

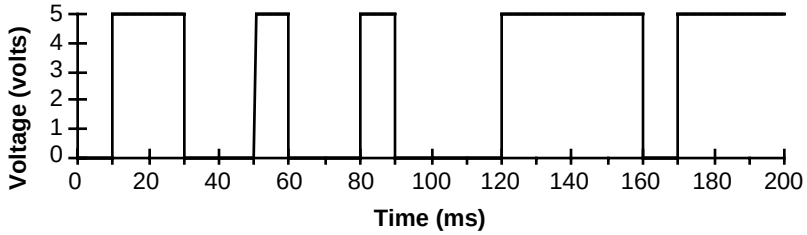


(d) IDD in Faulty Circuit

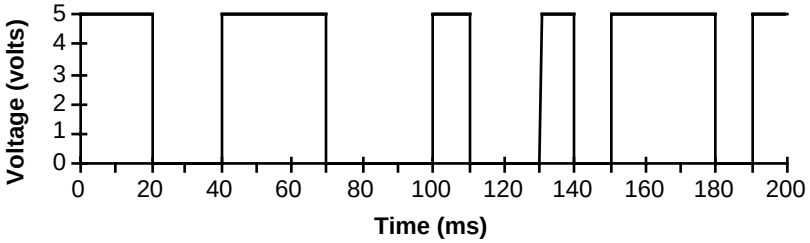
Figure C.2-3 Waveforms for M18 SON.

All the SOP's

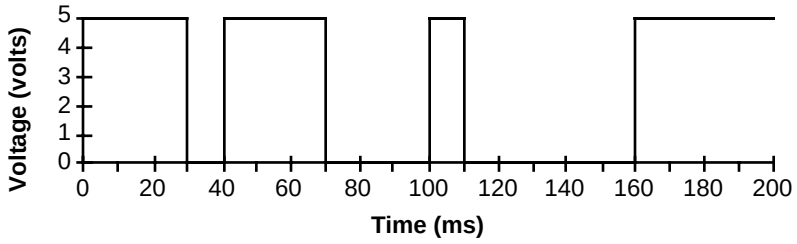
Just like M18 SON fault, these faults cause a degradation at either N1 or IQN. For example, M2 SOP causes N1 to become 960 mVolts when it should be 0 volts. This causes the IDDQ current to be around 110 uA. The affect on the voltage value is not sufficient to be detected by boolean testing. Fig. C.2-4 shows the waveforms for M2 SOP.



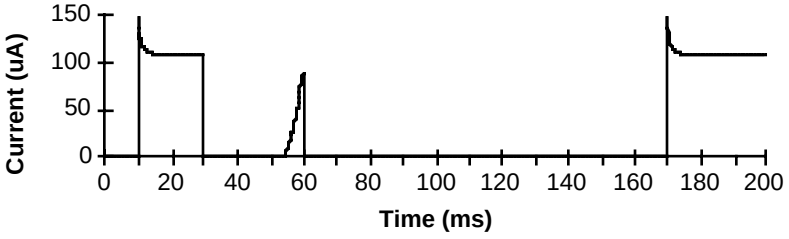
(a) C Input



(b) D Input



(c) N1 in Faulty Circuit

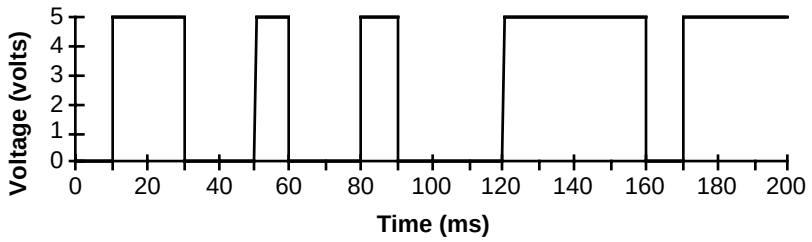


(d) IDD in Faulty Circuit

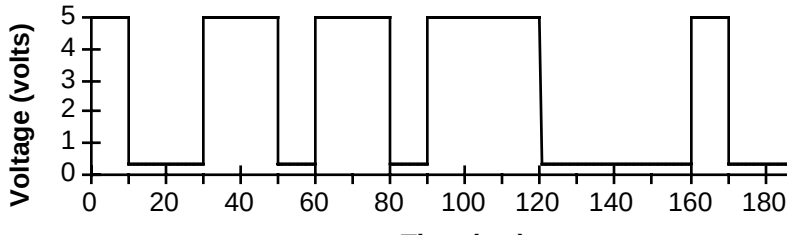
Figure C.2-4 Waveforms for M2 SOP.

Other SON's

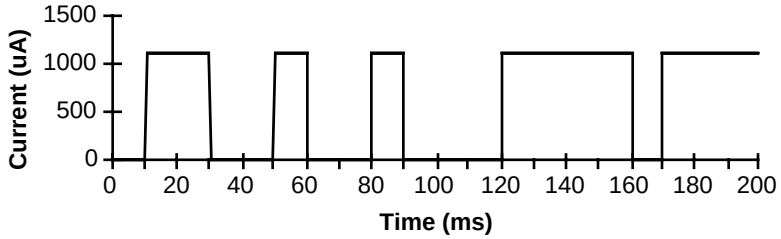
These faults create a straight path from VDD to ground. The output voltage is the result of the voltage divider created by having the faulty transistor and other transistors on at the same time. For these faults the voltage is not affected enough to be detected, but the current increases dramatically. As an example, the waveforms for M16 SON are shown in Fig. C.2-5. The waveforms show an IDDQ value of about 1.1 mA.



(a) C Input



(b) CKN in Faulty Circuit

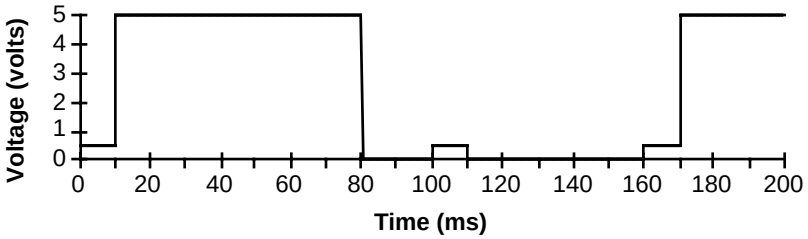


(c) IDD in Faulty Circuit

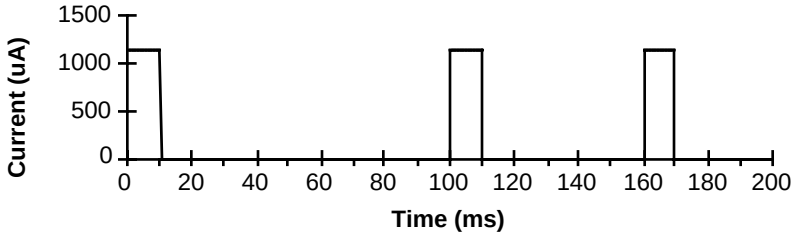
Figure C.2-5 Waveforms for M16 SON.

STI's N6-Q, N8-Q

N6-Q prevents a clean 0 volt on Q, giving about 0.5 volts. This is enough to turn on both transistors of down stream buffers increasing the IDDQ current to about 1 mA. Again 0.5 volts is not high enough to be differentiated from 0. The waveforms for N6-Q are shown in Fig. C.2-6. Similar arguments can be made for N8-Q.



(a) Q in Faulty Circuit

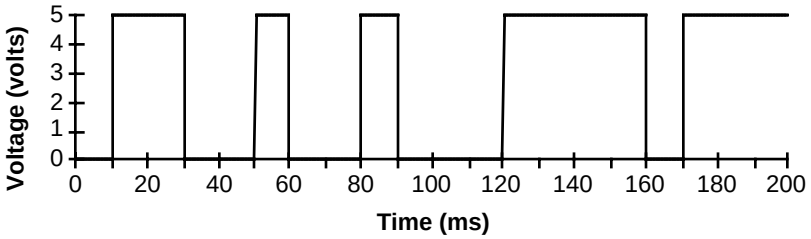


(b) IDD in Faulty Circuit

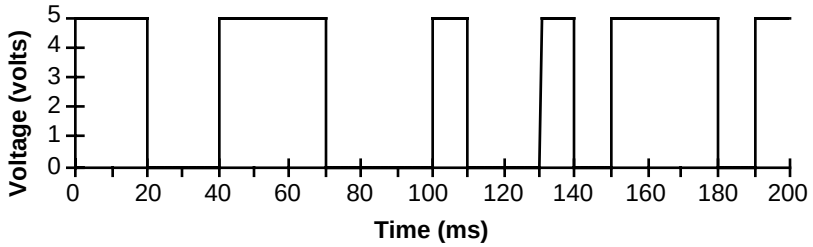
Figure C.2-6 Waveforms for STI N6-Q.

STI's N6-CK1, N6-C

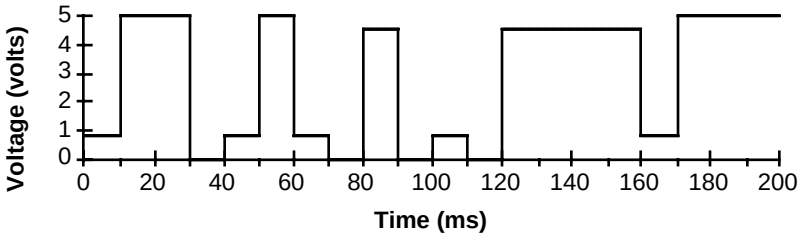
These faults are activated when C = 0 and D = 1. This would make N1 = 1, N2 = 0, turning on M17, and setting N6 to 1. At this point, M19 is turned off, so any effect at N6 cannot be observed. N6 = 1 and C = 0 causes a current of about 1.23 mA. The value of the connected node follows C. The waveforms are shown in Fig. C.2-7.



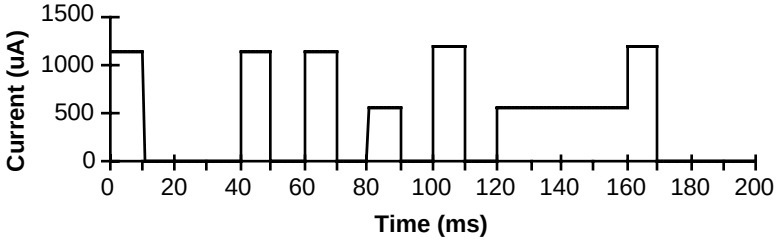
(a) C Input



(b) D Input



(c) N6 in Faulty Circuit

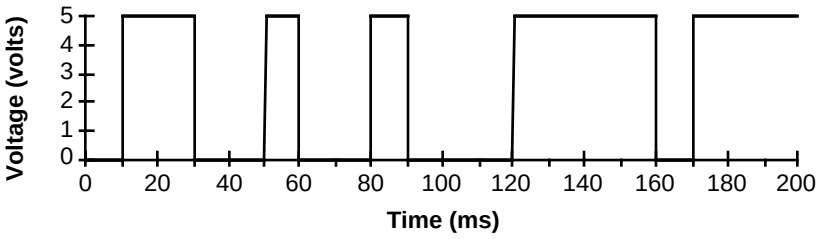


(d) IDD in Faulty Circuit

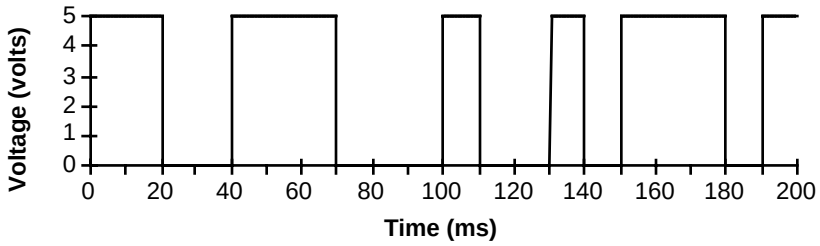
Figure C.2-7 Waveforms for STI N6-CK1.

N6 STP

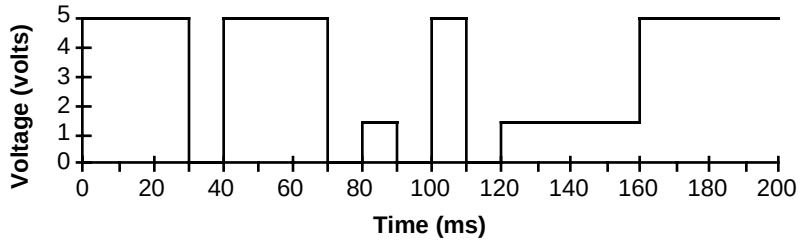
This fault can only have a boolean affect when CKN = 0 and N1 is storing a 0, as this would turn on M19. In this case, N1 is degraded to 1.36 volts and the current reaches 700 uA. The waveforms are shown in Fig. C.2-8.



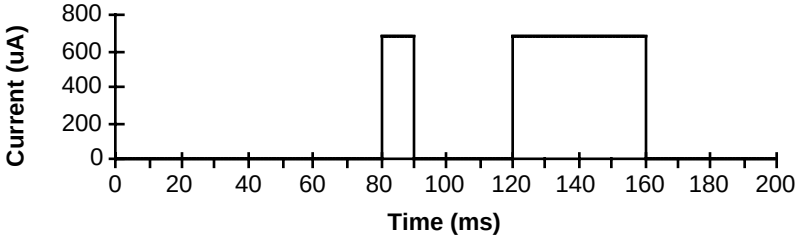
(a) C Input



(b) D Input



(c) N1 in Faulty Circuit



(d) IDD in Faulty Circuit

Figure C.2-8 Waveforms for N6 STP.

3. Faults Missed by 10 ms Cycle Time, Detected by 100 ns Cycle Time

Faults that are detected at a slow speed but are missed at high speed are called stationary faults [Ma 95]. One fault, M1 SOP (see Fig. C.2-9), affects the output in the 100 ns, but is missed by the 10 ms cycle time. M1 SOP causes CKN to float when C = 1. The value on the node begins to discharge slowly. In the case of 100 ns cycle time, CKN

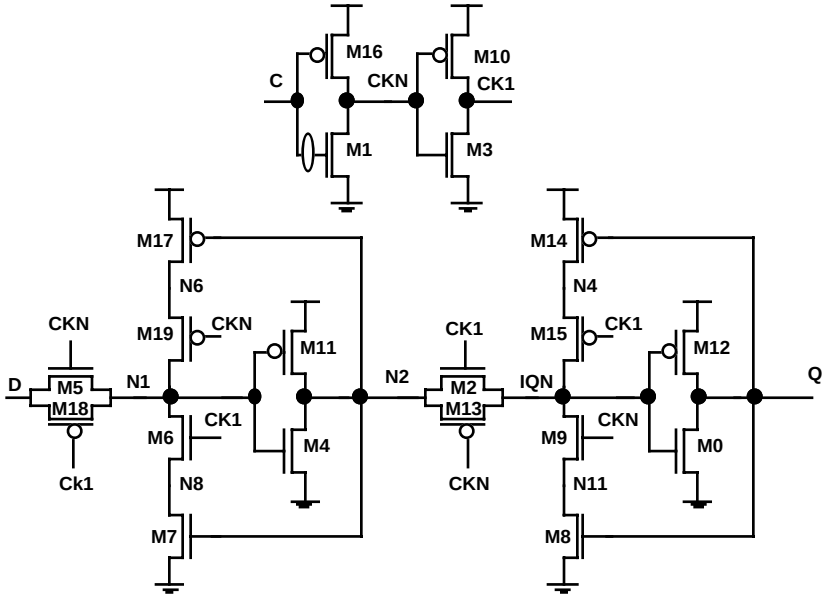
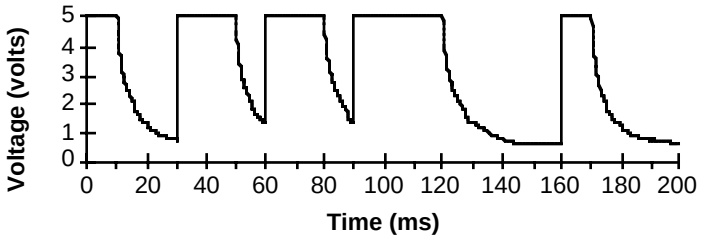
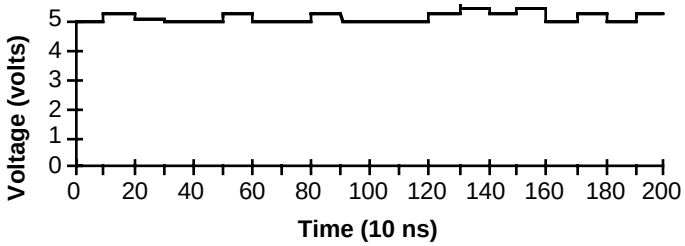


Figure C.2-9 Fault Detected by 100 ns Cycle Missed by 10 ms Cycle.



(a) CKN in Faulty Circuit With 10 ms Cycle



(b) CKN in Faulty Circuit With 100 ns Cycle

Figure C.2-10 Waveforms for M1 SOP.

has not discharged too much and the value still appears as 1. Thus, CKN will always have 1 on it, and the fault behaves as a STP fault. In the case of 10 ms cycle time, it gets enough time to discharge, and thus the circuit behaves normally. The waveforms showing these signals are shown in Fig. C.2-10.

4. Faults Missed by 100 ns Cycle Time, detected by 10 ms Cycle Time

There were 21 faults that were missed by the 100 ns cycle time that were detected by the 10 ms cycle time. They were all SOP faults (see Fig. C.2-11). The faults are divided into two groups: those in the master, and those in the slave.

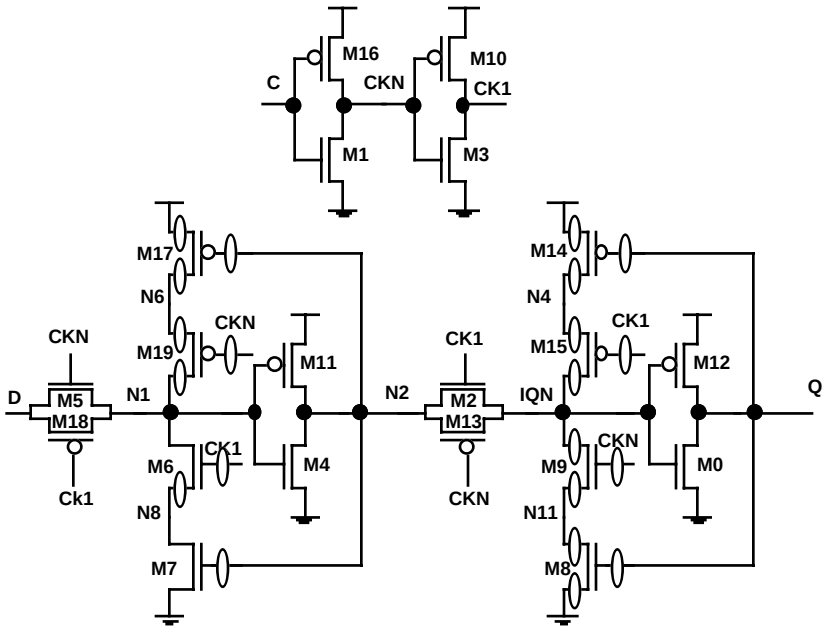
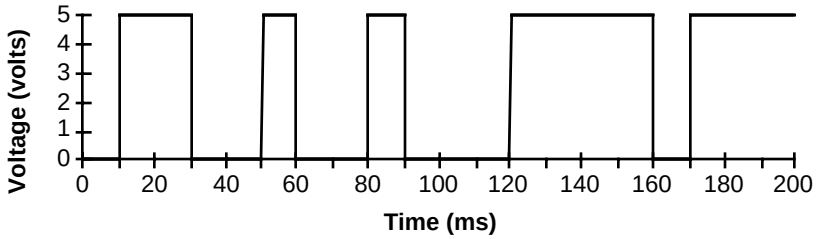


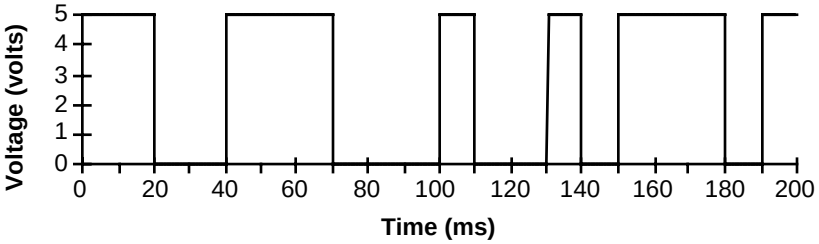
Figure C.2-11 Faults Detected by 10 ms Cycle Missed by 100 ns Cycle.

SOP's in the slave

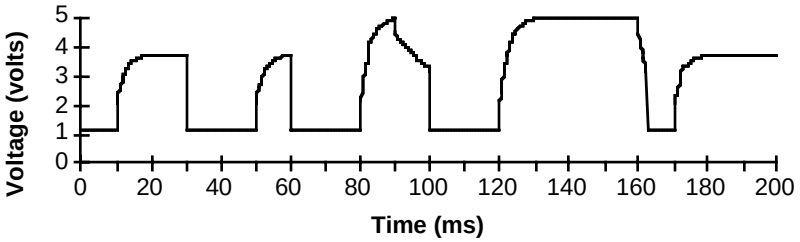
These faults cause N4, N11, or IQN to float. Given enough time, the node will discharge. For example, consider M14 SOP. After loading a 0, M14 is expected to be turned on, but because of the fault, M14 remains off, and N4 discharges slowly. In the case of 100 ns timing, N4 does not discharge enough to affect the output. However, with the 10 ms timing, N4 discharges completely. This results in the extra pulse between 100 and 120 ms in the output as seen in Fig. C.2-12.



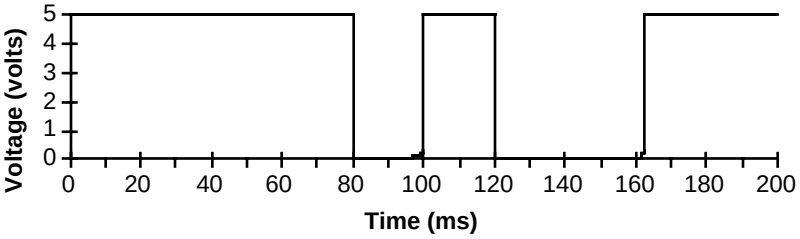
(a) C Input



(b) D Input



(c) N4 in Faulty Circuit

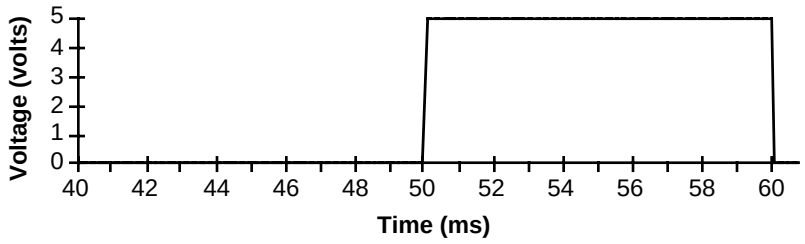


(d) Q in Faulty Circuit

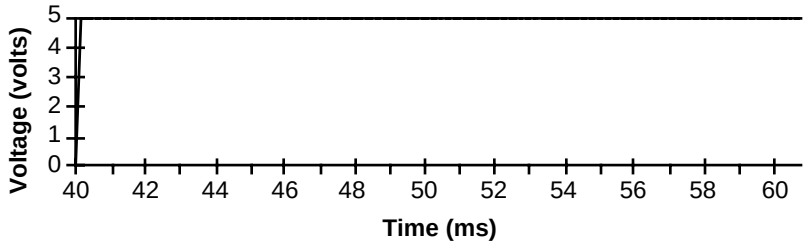
Figure C.2-12 Waveforms for M14 SOP.

SOP's in the master

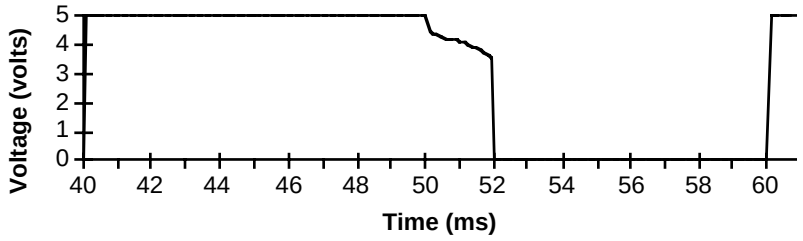
The faults in the master cause the stored value at N1 to discharge slowly. The value decays completely in 2 ms. The effect at the primary output is a narrow 2 ms pulse that appears after the rising edge of C. This is shown in Fig. C.2-13.



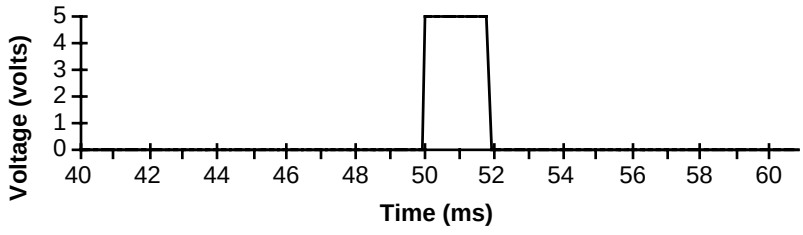
(a) C Input



(b) D Input



(c) N1 in Faulty Circuit



(d) Q in Faulty Circuit

Figure C.2-13 Waveforms for M17 SOP.

C.3 Faults Missed by Checking Experiment of MD Flip-Flop

The checking experiment missed 19 faults. These faults are shown in Fig. C.3-1. As before, white ovals indicate SOP or OPI faults, black ovals indicate SON faults, and thick black lines indicate STI faults. These faults can be categorized into five groups based on the circuit behavior in the presence of the fault.

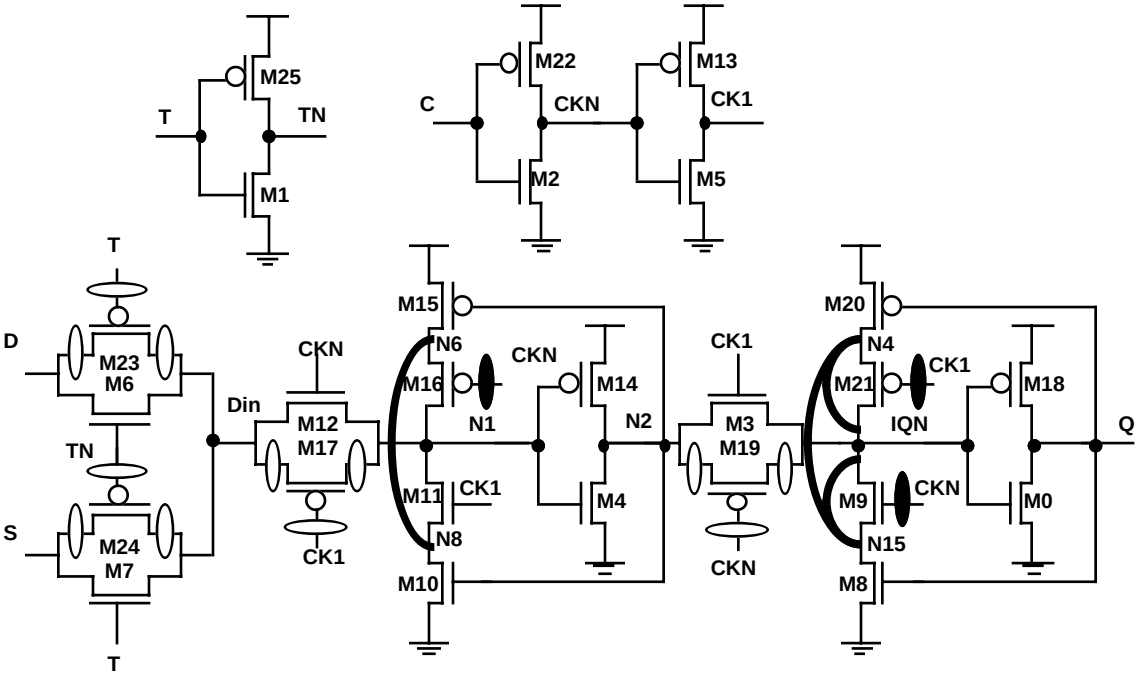


Figure C.3-1 Faults Missed by Checking Experiment (19 of them).

STI's N4-N15, N6-N8

Looking at N4-N15, when CK1 = 1, neither node involved is sensitized to the output. When CK1 = 0, CKN = 1, N4 and N15 will have the same voltage. Therefore, we cannot have a difference in the voltage output. Now, depending on the value of Q, either M20 or M8 will be off. Therefore, the fault cannot cause a direct path from VDD to ground, making detection by IDDQ testing impossible. Therefore, the fault is undetectable. A similar argument can be made for N6-N8.

SON faults in Slave

Consider M21 SON. In the fault-free circuit, if IQN = 1, then Q = 0, turning on M20 and making N4 = 1. Since IQN and N4 have the same voltage in the fault-free circuit, then the fault cannot be detected when IQN = 1. Now if IQN = 0, Q = 1, and M20 is turned off, leaving N4 with no drive at all. The fault would cause N4 to discharge to 0 in a short time. Thus the boolean

output would not be affected, and IDDQ would remain low. The other SON faults are similarly undetectable.

Rest of STI's (N4-IQN, N15-IQN)

The same analysis used for the SON faults can be used for these STI faults.

SON in Master (M16 SON)

The arguments presented for the SON faults for the slave did not apply to the faults in the master. The reason for that is that weaker drive due to the multiple transmission gates in series. N6-N1 prevents the flip-flop from loading a 0 after a 1 has been loaded, while N8-N1 prevents the loading of a 1. M11 SON causes a high current. M16 SON is not detected due to ratios of the transistor strengths.

All the SOP faults

These faults actually do affect the IDDQ current (13X for faults at M19 (slave transmission gate), and 128X for faults at M17 (master transmission gate), and 350 for M28 and M24 (multiplexer transmission gates)). However, they fall well below the threshold that we have set. Lowering the threshold would detect them. However, a lower threshold could cause a loss of yield. The reason these faults increase the current is that they cause voltage degradation of N1 or IQN, partly turning on both transistors of the downstream inverters.

In this section we have shown that all the faults that are not detected by the checking experiment are not detectable. Therefore, the limitation that checking experiment does not guarantee detection of some faults, did not affect its quality.

References

[Abramovici 90] Abramovici, M., *Digital Systems Testing and Testable Design*, Computer Science Press, New York, 1990.

[Al-Assadi 93] Al-Assadi, W.K., "Faulty Behavior of Storage Elements and Its Effects on Sequential Circuits," *IEEE Transactions on VLSI*, Vol. 1, No. 4, December, 1993.

[Brglez et. al. 89] Brglez, F., D. Bryan and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuit," *IEEE International Symposium on Circuits and Systems*, pp. 1929-1934, 1989.

[Chandra et. al. 93] Chandra, S., K. Pierce, G. Srinath, H. Sucar and V. Kulkarni, "CrossCheck: An Innovative Testability Solution," *IEEE Design and Test of Computers*, Vol. 10, No. 2, pp. 56-67, June, 1993.

[Chin and McCluskey 87] Chin, C.K. and E.J. McCluskey, "Test Length for Pseudorandom Testing," *IEEE Trans. on Computers*, Vol. C-36, No. 2, pp. 252-256, February, 1987.

[Eichelberger and Williams 77] Eichelberger, E.B., and T.W. Williams, "A Logic Design Structure for LSI Testability," *Proc. 14th Des. Autom. Conf.*, New Orleans, LA. pp. 462-468, June 20-22, 1977.

[Friedman and Menon 71] Friedman, A.D. and P.R. Menon, *Fault Detection in Digital Circuits*, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

[Fujiwara and Shimono 83] Fujiwara, H. and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Trans. on Computers*, Vol. C-32, No. 12, pp. 1137-1144, December, 1983.

[Fujiwara 85] Fujiwara, H., *Logic Testing and Design for Testability*, MIT Press, Massachusetts, 1985.

[Goel 81] Goel, P., "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. on Computers*, Vol. C-30, No. 3, pp. 215-222, March, 1981.

[Hawkins 89] Hawkins, CF., "Quiescent Power Supply Current Measurement for CMOS IC Defect Detection," *IEEE Trans. on Industrial Electronics*, pp. 211-218, May, 1989.

[Hennie 64] Hennie, F.C., "Fault Detecting Experiments for Sequential Circuits," *Proc. of the Fifth Annual Switching Theory and Logical Design Symposium*, S-164, Princeton, New Jersey, pp. 95-110, 1964.

[Kielkowski 94] Kielkowski, R., *Inside SPICE Overcoming the Obstacles of Circuit Simulation*, McGraw Hill, USA, 1994.

[Kubuo 68] Kubo, H., "A Procedure for Generating Test Sequences to Detect Sequential Circuit Failures," NEC Res. and Dev., No. 12, October, 1968.

[Larrabee 89] Larrabee, T., "Test Pattern Generation Using Boolean Satisfiability," *IEEE Trans. on CAD*, pp. 4-15, January, 1989.

[Lee and Breuer 90] Lee, K.J. and M.A. Breuer, "A Universal Test Sequence for CMOS Scan Registers," *IEEE Custom Integrated Circuits Conference*, pp. 28.5.1-28.5.4, 1990.

[Ma and McCluskey 95] Ma, S., P. Franco and E.J. McCluskey, "An Experimental Chip to Evaluated Test Techniques, Experimental Results," *Proceedings ITC*, pp. 663-672, 1995.

[Makar and McCluskey 88] Makar, S.R. and E.J. McCluskey, "On the Testing of Multiplexers," *Proceedings ITC*, pp. 669-679, 1988.

[Makar and McCluskey 94] Makar, S.R. and E.J. McCluskey, "Using Checking Experiments to Test Two-State Latches," CRC Technical Report 94-11, 1994.

[Makar and McCluskey 95a] Makar, S.R. and E.J. McCluskey, "Checking Experiments to Test Latches," *Proceedings VTS*, pp. 196-201, 1995.

[Makar and McCluskey 95b] Makar, S.R. and E.J. McCluskey, "Functional Tests for Scan Chain Latches," *Proceedings ITC*, pp. 606-615, 1995.

[McCluskey 86] McCluskey, E.J., *Logic Design Principles*, Prentice-Hall, New Jersey, 1986.

[McCluskey et. al. 88] McCluskey, E.J., S. Makar, S. Mourad and K. Wagner, "Probability Models for Pseudorandom Test Sequences," *IEEE Trans. on CAD*, Vol. 7, No. 1, pp. 68-74, January, 1988.

[Miczo 86] Miczo, A., *Digital Logic Testing and Simulation*, Harper and Row, New York, 1986.

[Mourad 90] Mourad, S., "Sequential Circuit Testing," *COMPCON*, p 449-454, 1990.

[Muth 76] Muth, P., "A Nine-Valued Circuit Model for Test Generation," *IEEE Trans. on Computers*, Vol. C-25, No. 6, pp. 630-636, 1976.

[Perry 92] Perry, R., "IDDQ Testing in CMOS Digital ASICs - Putting It All Together," *Proceedings ITC*, pp. 151-157, 1992.

[Putzolu and Roth 71] Putzolu, G.R. and J.P. Roth, "A Heuristic Algorithm for the Testing of Asynchronous Circuits," *IEEE Trans. on Computers*, Vol. C-20, No. 6, pp. 639-647, June, 1971.

[Reddy and Reddy 86] Reddy, M.K. and S.M. Reddy, "Detecting FET Stuck-Open Faults in CMOS Latches and Flip-Flops," *IEEE Design and Test of Computers*, Vol. 3, No. 5, pp. 17-26, October, 1986.

[Roth 66] J.P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," IBM *Journal of Research and Development*, Vol. 10, No. 4, pp. 278-291, July, 1966.

[Savir and Bardell 84] Savir, J. and P.H. Bardell, "On Random Pattern Test Length," *IEEE Trans. on Computers*, Vol. C-33, No. 6, pp. 467-474, June, 1984.

[Saxena et. al. 93] Saxena, N. R., R. Tangirala and A. Srivastava, "Algorithmic Synthesis of High Level Tests for Data Path Designs," *FTCS*, pp. 360-369, 1993.

[Sellers et. al. 68] Sellers, E.F., M.Y. Hsiao and L.W. Bearnsin, "Analyzing Errors with the Boolean Difference," *IEEE Trans. on Computers*, C-17, No. 7, pp. 676-683, July, 1968.

[Sentovich et. al. 92] Sentovich, E.M., K.J. Singh, L. Lavagno, C. Moon, R. Mrgai, A. Saldanha, H. Savoj, P. R. Stephan, R.K. Brayton, A.S. Vincentelli, "SIS A system for Sequential Circuit Synthesis," *Electronics Research Lab* Memorandum, No. UCB/ERL M92/41, 1992.

[Sucar 89] Sucar, H., "High Performance Test Generation for Accurate Defect Models in CMOS Gate Array Technology," *ICCAD*, pp. 166-169, 1989.

[Williams and Angel 73] Williams, M.J. and J.B. Angel, “Enhancing Testability of Large Scale Integrated Circuits via Test Points and Additional Logic,” *IEEE Trans. on Computers*, C-22, No. 1, pp. 46-60, January, 1973.

[Williams 85] Williams, T.W., “Test Length in a Self-Testing Environment,” *IEEE Design and Test of Computers*, Vol. 2, No. 2, pp. 59-63, April , 1985.

[Wilson 72] Wilson, R., *Introduction to Graph Theory*, Academic Press, New York, 1972.