Testing Asynchronous Circuits: A Survey[1]

Henrik Hulgaard, Steven M. Burns, and Gaetano Borriello

Department of Computer Science and Engineering, FR–35
University of Washington
Seattle, WA 98195

Technical Report 94-03-06
March 6, 1994

[1]Submitted to INTEGRATION, THE VLSI JOURNAL.

Testing Asynchronous Circuits: A Survey

Henrik Hulgaard, Steven M. Burns, and Gaetano Borriello

Department of Computer Science and Engineering, FR–35
University of Washington
Seattle, WA 98195
E-mail: {henrik,burns,gaetano}@cs.washington.edu

March 6, 1994

Abstract

Asynchronous circuit design has been studied for decades, but it has only recently been feasible to construct large and efficient asynchronous systems. This paper surveys different techniques for checking whether an asynchronous circuit has fabrication defects. The inherent differences between asynchronous and synchronous circuits, primarily that asynchronous circuits do not have a global clock, necessitate a review of the testing techniques used for synchronous circuits and a re-evaluation of the trade-offs involved. New methods that efficiently utilize the structure of asynchronous circuits are possible, most notably self-checking asynchronous circuits can be implemented with little or no circuit overhead.

Keywords: Asynchronous circuits, stuck-at fault testing, path delay fault testing, self-checking circuits, test generation.

1 Introduction

Asynchronous circuits promise a number of advantages over synchronous systems. They have no problems with clock skew, can be designed for average case rather than worst case performance, have potentially lower power consumption, and have a higher degree of modularity. Designing asynchronous circuits is challenging because hazards and races must be carefully considered. Therefore, the focus of research in the area has been primarily directed to synthesis and verification techniques, while little attention has been paid to techniques to efficiently verify whether a fabricated asynchronous circuit has any physical faults. However, as asynchronous circuits become larger [24, 45] and start to be used in commercial products [40], testing concerns become critical. This paper reviews recently developed approaches for testing digital asynchronous circuits and systems for fabrication defects.

Several aspects of asynchronous circuits make them harder to test than synchronous circuits. Asynchronous circuits by definition have no global synchronization signals. This drastically reduces the amount of control over the circuit as it cannot easily be "single stepped" through a sequence of states—a common way to test synchronous circuits. Also, because asynchronous circuits tend to have more state holding elements than synchronous circuits, generating test vectors is harder and design techniques to ease testing will have a higher area overhead. Finally, an asynchronous circuit may have hazards or races when faulty, and these delay faults are notoriously difficult to detect.

However, other aspects of asynchronous circuits tend to make them easier to test. Because asynchronous circuits use local handshakes instead of global clock signals to synchronize operations, a stuck-at fault on the signals used for this handshake will cause communicating modules to wait indefinitely, an effect that is easily observable. These differences lead to new approaches for testing asynchronous circuits or a reevaluation of the trade-offs involved when applying techniques developed for testing synchronous circuits.

The structure of the paper is the following. Section 2 introduces some basic testing terminology. Section 3 discusses a class of asynchronous circuits where a faulty circuit will deadlock for all faults, the easiest way to test. The generation of test vectors is described in Section 4. Techniques for making a circuit easier to test are discussed in Section 5. Finally, Section 6 describes methods for testing the actual delays in an asynchronous circuit, a necessity for circuits designed using delay assumptions.

2 Testing Terminology

The outputs of the circuit under test are called *primary outputs* and we assume we can easily observe these. Similarly, the inputs to the circuit are called *primary inputs* and these are assumed to be easily controllable.

The *controllability* of a circuit is the ability to establish a specific signal value at each node in the circuit by proper setting of the circuit's primary inputs. *Observability* of a circuit is the ability to determine the value at any node in the circuit by observing the primary outputs while controlling the primary inputs. The *testability* of a circuit is a measure that attempts

to reflect the ease with which a circuit can be tested. A circuit with high testability generally has a higher degree of observability and controllability than one with low testability.

A *failure* in a circuit occurs when the circuit deviates from the specified behavior. A *fault* is a physical defect which may or may not result in a failure. *Fault detection* is the process of determining whether a given circuit contains one or more faults. This is done by applying a sequence of input values (called *test vectors*) to the circuit and observing the primary outputs. If the outputs differ from the specification, a failure has occurred and a fault is present in the circuit.

In a *test*, a set of test vectors are applied to the circuit in order to detect as many faults as possible. The effectiveness of a test is measured by the *fault coverage*, which is the ratio of faults potentially detected by the test to the total number of possible faults in the circuit. The *length* of a test is the number of test vectors in the test and the *test time* is the time it takes to apply the test vectors and observe the results. Key quality measures for a given test approach includes the time to generate the test vectors, the fault coverage, and the test time. Also, testing overhead such as increased area, decreased operating speed, and added I/O pins influences which test approach is most suitable for a given circuit. Often different approaches offer different trade-offs between these criteria. For example, there is generally a trade-off between the time to generate the test vectors, the length of a test, and the fault coverage.

A *fault model* is employed to test a circuit efficiently based on its structure rather than on its functionality[1]. The fault model is an abstraction of the physical faults we try to detect. The more detailed the fault model, the more actual (physical) faults can be modelled. But this higher precision is obtained at the expense of more complex test generation algorithms, longer test generation times, and longer test times.

Fault models can describe the faults at different abstraction levels. The most common abstraction level is the gate level, but fault models exist which describe faults at the transistor level as well as on higher levels. Most fault models assume that the circuit only contains a single fault, as the number of potential multiple fault combinations is so large that test generation becomes infeasible.

A widely used fault model for synchronous circuits is the *(input) stuck-at fault model*. In this fault model it is assumed that a physical fault can be modeled as a signal in the circuit being either *stuck-at-0* or *stuck-at-1*. A gate can have either an input stuck-at-0 or stuck-at-1, or the output stuck-at-0 or stuck-at-1. Thus, a gate with n inputs has $2(n+1)$ different possible stuck-at faults. A wire branching out to n gates also has $2(n+1)$ stuck-at faults: two at the "input" to the wire and two for each of the n end-points. The possible stuck-at faults are shown in Figure 1. This fault model has proven reasonable good at representing the most common faults in fabricated synchronous circuits, and is simple enough to be practical. However, it should be noted that although the applicability of this fault model is generally accepted for synchronous circuits, this has not been established for asynchronous circuits.

A simpler fault model only considers faults on the outputs of the gates. A wire branching

[1]Full (i.e., exhaustive) functional testing generally takes a long time. For example, consider testing a 16-bit adder functionally. This would take $2^{16} \cdot 2^{16} \approx 4$ billion test vectors.

Figure 1

Possible places of stuck-at faults in the input stuck-at fault model for a gate (left) and a wire branching out to multiple gates (right).

out to n gates has only two different faults, independently of n, corresponding to the entire net being either stuck-at-0 or stuck-at-1. Similarly for an n-input gate, see Figure 2. This fault model is called the *output stuck-at fault model* (the *input stuck-at fault model* refers to the stuck-at fault model described above).

Figure 2

Possible places of stuck-at faults in the output stuck-at fault model for a gate (left) and a wire branching out to multiple gates (right).

3 Self-Checking Circuits

Asynchronous circuits have no global clock to synchronize operations. Instead the synchronization is achieved using local handshaking signals. While a synchronous circuit can easily be single-stepped through different states by using the global clock, this is much harder (sometimes impossible) for asynchronous circuits. The lack of global synchronization in asynchronous circuits means that synchronization must be achieved by other means. Two general approaches have been taken. One approach, used in the design of classical asynchronous state machines [18, 43], is to make timing assumptions about the delays of the gates and wires. In order to avoid critical races and hazards it is often necessary to add extra (functionally redundant) circuitry and appropriate delays. This makes it very difficult to fully test this class of circuits. For example, under the stuck-at fault model full fault coverage is not possible for a circuit with redundant logic.

Here we will focus on the alternative, which is to use explicit handshake signals for local synchronization. Because no absolute timing assumptions are made on the handshake, circuits are robust and easily composable, a property that has made this design approach popular [5, 9, 22, 26, 27, 30]. While the lack of global synchronization decreases the controllability of the circuit and thus makes an asynchronous circuit harder to test, the local synchronization tends to increase the observability. Consider the popular four-phase handshake protocol, see Figure 3. A computation is started by the environment by issuing a request (req) to the circuit. The completion of the computation is indicated by the circuit raising an acknowledge signal (ack). To complete the protocol, the request is lowered, and the circuit lowers the acknowledge signal in response. That is, the environment (the active

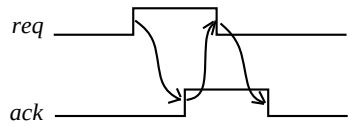

Figure 3

A four-phase handshake protocol.

part) will execute $req\uparrow;[ack];req\downarrow;[\neg ack]$ while the circuit will do $[req];ack\uparrow;[\neg req];ack\downarrow$[2]. In the presence of a stuck-at fault on either req or ack, either the environment or the circuit will wait forever. For example if the request signal is stuck-at-0, the passive end will wait on $req\uparrow$ and the active end will wait on $ack\uparrow$ after issuing $req\uparrow$. A transition that is supposed to occur but doesn't because of a stuck-at fault is called *inhibited* [25, 14]. A fault that causes an inhibited transition will always eventually cause the circuit to *halt*, a situation which is easily detected during the test. The circuit is tested by issuing the request and waiting a bounded amount of time, τ, for the circuit to raise the acknowledge signal. If the circuit does not, then it has halted and is thus faulty. The time bound τ can be determined given the fabrication technology and the circuit specification[3]. Circuits that have the property that they halt for all faults are called *self-checking* [4, 44] (or *self-diagnostic* [3, 8]). Thus, self-checking circuits are fully testable (i.e., 100% fault coverage). A test for a self-checking circuit attempts to toggle all nodes at least once, that is, during a test all nodes are driven both high and low. For example, a 4-phase handshake circuit is tested by performing a complete handshake.

The above notion of a self-checking circuit is different from the one conventionally used for synchronous systems. In synchronous systems special codes or state assignments are used so that the circuit produces an illegal output in the presence of a fault. A separate circuit (a *checker*) can then detect the illegal output code and raise an error signal. Faults are detected while running the circuit at its operation speed (called *on-line* testing). The overhead in terms of area is quite large, often as much as a factor of two. Similar self-checking approaches have been applied to classical asynchronous state machines by using a state assignment that brings the circuit into a special state when a fault exists [37, 29]. However, designing asynchronous circuits using the classical state machines approach [43, 18] and related approaches [15, 32] has turned out to be problematic for larger systems. The state machines have timing constraints that must be met to ensure correct operation (such as *fundamental mode* assumptions) and these constraints are hard to satisfy when composing multiple machines. We therefore here use the term "self-checking" to refer to asynchronous circuits that halts in the presence of a fault. The rest of this section discusses the classes of asynchronous circuits and fault models under which a faulty circuit always halts.

Delay-insensitive asynchronous circuits work correctly independently of the delays of both

[2]This is a *handshake expansion* [22]. $[e]$ indicates waiting on the Boolean expression e to become true, and $s\uparrow$ and $s\downarrow$ indicate driving the signal s high and low, respectively.

[3]A subtle point: if the circuit contains an arbiter, the arbiter may take an unbounded amount of time to leave a meta-stable state.

the gates and the wires in the circuit. Consequently, *every* transition in the circuit must be acknowledged by the receiver of the transition. This property is called the *acknowledgment property* in [25]. Specifically, if a wire fans out to multiple gates (a *fork*), each gate at the destinations of the fork must acknowledge the receipt of a signal transition before a new transition can occur on the input to the fork. Because each single transition in a delay-insensitive circuit is acknowledged, any stuck-at fault will cause the circuit to halt, making delay-insensitive circuits self-checking under the input stuck-at fault model.

As a very simple example, consider the circuit in Figure 4. Initially, z is 1 and x and y are 0. The C-element[4] will then change its output to 0, which changes x and y to 1, causing all the signals in the circuit to oscillate. Any stuck-at fault will cause this circuit to halt. For example, the fault $z1$-stuck-at-0 ($z1$ is the one end of the fork with z as input) will cause x to be 1 and thus the output of the C-element can never be 0. A complete test of the circuit simply needs to toggle all nodes in the circuit, which is done "automatically" in this example.

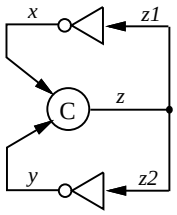

Figure 4

A simple delay-insensitive circuit where any input stuck-at fault will cause a halt.

The class of delay-insensitive circuits turns out to be very small [23]—a delay-insensitive circuit has to be constructed exclusively from C-elements and inverters (if single output gates are used exclusively). Thus most practical circuits will not be delay-insensitive. Some delay assumptions must be made in order to construct useful asynchronous circuits. One such assumption is that all wire delays are negligible. *Speed-independent* circuits make this assumption while maintaining that gate delays can be arbitrarily large. By assuming wire delays to be zero, a transition on the input to a fork needs only be acknowledged by one of the recipients, not all of them. The assumption is equivalent to assuming all forks are *isochronic* [23]. The isochronic fork assumption states that a transition on the input to a fork arrives at the ends of the fork at the *same* time. The implementation must satisfy this constraint.

In the output stuck-at fault model, a fork is considered a single node that can be either stuck-at-0 or stuck-at-1. Because a transition on the input to the fork still has to be acknowledged by at least one of the recipients, a speed-independent circuit is self-checking under this simple fault model [3, 4]. Unfortunately, it's questionable whether the output stuck-at fault

[4]A Muller C-element (or rendezvous element) is a stateholding element that waits for the inputs to be equal, and then changes the output to be the same as the inputs.

model reflects a reasonable number of physical faults. However, some speed-independent circuits turn out to be self-checking under the more general input stuck-at fault model.

The FIFO element in Figure 5 [41] is an example of a speed-independent circuit that is self-checking under the input stuck-at fault model. Data values in a FIFO queue built from these elements are coded using the dual-rail code that consists of three values: **true** (10), **false** (01), and **empty** (00). A dual-rail code for a variable x can be represented by two signal wires, named $x.t$ and $x.f$. Inputs to the FIFO queue must alternate between the

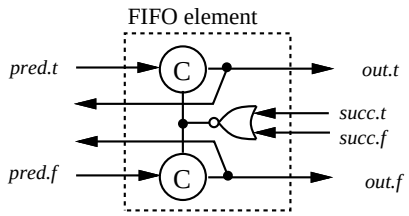

Figure 5

Implementation of FIFO element. The input *succ* is the output of the succeeding FIFO element.

empty value and a data value. The environment inserts a **true**-value by raising $pred.t$ or a **false**-value by raising $pred.f$. The signals $succ.t$ and $succ.f$ are the outputs of a succeeding FIFO element. The fork between the two C-elements is isochronic, but a stuck-at fault at one of the two ends of the fork will cause the circuit to halt because the C-element will not be able to propagate either the **true**-value or the **false**-value to the outputs. The queue is tested by propagating a **true**-value, an **empty**-value, and a **false**-value through the queue. If they all get through, the queue is fault-free under the input stuck-at fault model.

The class of quasi-delay-insensitive circuits [22] is "in between" delay-insensitive and speed-independent circuits; only some of the forks are assumed to be isochronic. This is a interesting class because it is possible to construct basic elements that have delay-insensitive interfaces, and only use isochronic forks within the elements. The elements are easy to compose (no timing constraints are imposed on the interconnections) and the isochronic forks are easier to implement because they are local. The existence of isochronic forks indicates that the circuits in this class are not self-checking under the input stuck-at fault model (but are under the output stuck-at fault model). We can introduce a fault model that models faults on forks differently depending on whether a fork is isochronic, called the *isochronic transition fault model* [34]. The fault model is a combination of the input and output stuck-at fault models. It considers input stuck-at faults for non-isochronic forks and output stuck-at faults for isochronic forks. Under this fault model, every quasi-delay-insensitive circuit is self-checking.

After having identified a number of classes of self-checking circuits and their corresponding fault models, it is natural to consider what happens if we consider circuits that are not self-checking. For example, quasi-delay-insensitive and speed-independent circuits are not generally self-checking under the input stuck-at fault model. A circuit that is not self-

checking will either contain redundant logic, in which case it may not be possible to test for all faults, or it may have *premature firings* [25] for some faults. A premature firing is a signal that changes too early according to the specification. To illustrate a premature firing in the presence of a stuck-at fault, consider the circuit in Figure 6 (from [25]). The circuit (called a D-element) sequences two four-phase handshakes. Ahandshake is started on li and lo, and before completing this handshake another full handshake is performed on ri and ro. The circuit specification is[5]

$$*[[li]; u\uparrow; [u]; lo\uparrow; [\neg li]; ro\uparrow; [ri]; u\downarrow; [\neg u]; ro\downarrow; [\neg ri]; lo\downarrow] \tag{1}$$

and the environment specification is

$$*[li\uparrow; [lo]; li\downarrow; [ro]; ri\uparrow; [\neg ro]; ri\downarrow; [\neg lo]]\,.$$

The two forks with li and ri as inputs are isochronic. Under the input stuck-at fault model, this circuit has two faults that cause a premature firing: $l1$-stuck-at-0 and $r2$-stuck-at-0. All other faults cause the circuit to halt. Consider $l1$-stuck-at-0. Initially all signals are low.

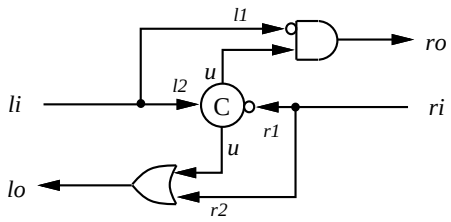

Figure 6

Implementation of a D-element.

The environment raises li which causes u to go up. Because $l1$ is stuck-at-0 both lo and ro go up, $ro\uparrow$ being a premature firing because it isn't supposed to happen until after the environment has lowered li. The next section will describe how to derive test sequences for premature firings.

4 Test Generation

The purpose of test generation is to determine input sequences that will cause a faulty circuit to behave differently from its specification.

If the circuit has redundant logic, the behavior of a faulty circuit may depend on the delays of the circuit elements, i.e., a fault may cause hazards or races that only occur for certain combinations of delays. To guarantee that these circuits will work under a large range of conditions, the values of the actual delay must be checked. How this delay test can be performed is described in Section 6. Here we will focus on stuck-at faults in circuits

[5]The operator "$*[s]$" denotes repetition of s.

without redundant logic. For non-redundant circuits, a fault is known to cause either the circuit to halt or a premature firing.

A test consist of two phases. First the fault must be *activated*, e.g., if the test is for a node stuck-at-0, a 1 must be assigned to the node in order to detect the fault. Secondly, the effect of the error must be made observable by propagating the error to a primary output.

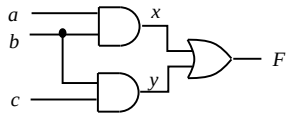

Figure 7

The fault x-stuck-at-0 is tested using the vector $(a, b, c) = (1, 1, 0)$.

Consider the simple combinational circuit in Figure 7 with the fault x-stuck-at-0. To find a test vector the fault is first activated by assigning 1 to x. This results in the partially specified vector $(a, b, c) = (1, 1, -)$. Then the effect of the fault must be propagated to F, which requires that y is 0. That is achieved by assigning 0 to c. Thus, the final test vector is $(1, 1, 0)$. If $F = 0$ for this input vector, the circuit is faulty. Efficient algorithms exists to generate test vectors for combinational circuits, e.g., the D-algorithm [35], PODEM [11] and FAN [10].

Test generation for sequential circuits is a much harder problem, and it does not have a general solution [28]. Because the output of a sequential circuit not only depends on the inputs but also on the present state, a test for a given fault must first put the circuit into a known state before applying a test pattern that will exercise the fault. A faulty circuit may not start in the specified initial state but instead in some other arbitrary state. *Self-initializing* sequences are used to put the circuit into a known state, although this state may be affected by the fault. This section describes approaches to test generation for asynchronous sequential circuits assuming that the initial states are known.

One approach is to conceptually (i.e., not physically) transform the sequential circuit into a combinational circuit and then apply combinational circuit test generation algorithms. This transformation is done by considering all state holding elements as combinational gates with an extra input (q), representing the present state, and an extra output (q^+), representing the next state of the element. A sequential circuit is transformed into a combinational one by making several copies of the circuit (called *time frames*), connecting the next-state output of a state holding element in one time frame to the present state input of the same element in the succeeding time frame, forming an *iterative array*. We can then apply any test generation algorithm for combinational circuits to the iterative array to generate a test vector for the sequential circuit. The number of time frames that are necessary to detect a given fault determines the number of test vectors needed to test for the fault. When using this technique for asynchronous circuits, special care must be taken that hazards and critical races are not introduced during the test. Hazard and race-free tests can be derived for asynchronous circuits using a 9-valued logic combined with the D-algorithm [13]. Figure 8

shows an iterative array of size three for the D-element. The C-element is changed into a combinational three-input majority gate, the third input being the present state input.

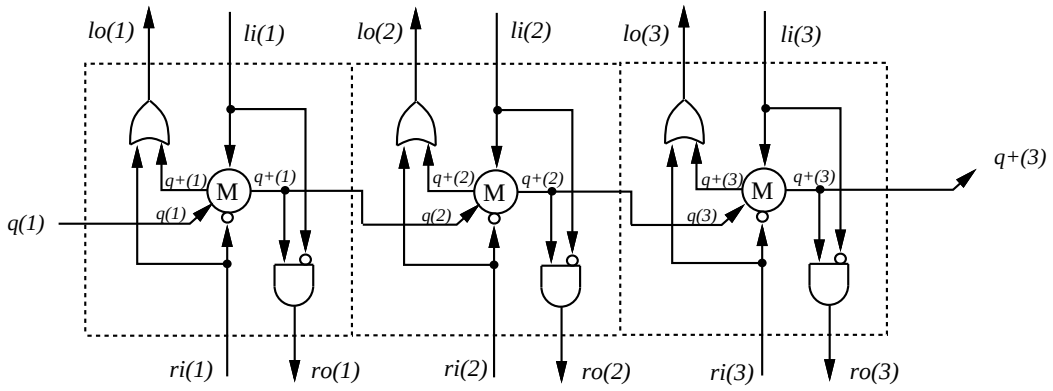

Figure 8

An iterative array of size three for the D-element in Figure 6.

The major problem for this test generation approach is that the number of time frames necessary in the worst case is exponential in the number of stateholding elements. In practice the maximum number of copies of the circuit is limited by a (small) user defined constant. This guarantees a reasonable execution time at the cost of lower fault coverage.

Another approach is to base the test generation on an analysis of the fault being considered. Stuck-at faults in non-redundant circuits will either cause the circuit to halt or generate a premature firing. We can derive the conditions for these two behaviors [14]. Consider the AND-gate in the D-element with $l1$ and u as inputs (Figure 6). The specification of the AND-gate can be written as[6]

$$\neg l1 \wedge u \rightarrow ro\uparrow \tag{2}$$

$$l1 \vee \neg u \rightarrow ro\downarrow \tag{3}$$

Consider the fault $l1$-stuck-at-0. This fault can inhibit $ro\downarrow$, and cause a premature firing of $ro\uparrow$. The $ro\downarrow$ transition is inhibited if there is a state in the execution of the circuit where ro is true, u is true (thus, u is not causing $ro\downarrow$), and $l1$ is true. In this state ro will go down in a fault-free circuit, but will remain high in the presence of the fault. The transition $ro\downarrow$ is inhibited in the states that satisfy the state predicate $INH_{ro\downarrow}$:

$$INH_{ro\downarrow} \equiv l1 \wedge u \wedge ro$$

A condition for the premature firing of ro is derived similarly:

$$PRE_{ro\uparrow} \equiv l1 \wedge u \wedge \neg ro$$

[6] This notation is called *production rules*. A production rule consists of a Boolean guard and an assignment: *guard* $\rightarrow$ *assignment*. When the guard is true the assignment can be performed.

To find a test for the fault we need to determine an input sequence that will put the circuit into a state where either *INH* or *PRE* is true. If we can find such a sequence that leads to satisfying *INH* (without *PRE* becoming true), we have found a test for the fault since this input sequence will eventually cause the faulty circuit to halt. The situation is more complex for faults that only cause a premature firing. Again, a sequence must be determined that will put the circuit into a state where *PRE* holds, and the effect of the premature firing must also be propagated to a primary output. Since this is not always possible, not all premature firings are testable.

Consider the fault $l1$-stuck-at-0. From the circuit specification (1) it is noted that $INH_{ro\downarrow}$ is false in all states and thus no test exists for this fault that will cause the circuit to halt. However, $PRE_{ro\uparrow}$ is true after $li\uparrow$ and $u\uparrow$, and before $ro\uparrow$. Because $ro\uparrow$ is a primary output the effect of the premature firing is directly observable. A test for the fault is $li\uparrow;[lo]$ and postponing $li\downarrow$. If $ro\uparrow$ occurs in this state, the fault is present. For the D-element it turns out that all faults can be tested using only one sequence, the execution of one cycle of the environment: $li\uparrow;[lo];li\downarrow;[ro];ri\uparrow;[\neg ro];ri\downarrow;[\neg lo]$.

The main problem with this approach is that if there is much concurrency in the circuit or the environment, the number of possible states explodes, and the time to find states that satisfies either *INH* or *PRE* becomes long. This approach is therefore mainly appropriate for circuits which are mostly sequential, i.e., control dominated circuits.

The two approaches just described generate test vectors for *one* specific fault. However, the goal is to find a (preferably small) set of vectors that test for most of the faults in the circuit. Clearly, we can apply the above techniques for each fault, but it is more efficient to use a *fault simulator* in the test generation [1]. A fault simulator determines the effect of a fault by simulation. It can be used to determine the fault coverage of a set of test vectors, and also to determine which faults in a circuit a given test vector detects. Given a circuit, a fault, and an input sequence, the simulator determines whether the output of the circuit differs from the specification. If it does, the vector detects the fault. However, a fault may cause a critical race in which case the output of the circuit may depend on the actual delays. The input vector is only a test if it always causes the circuit to deviate from the specification independent of the delays, i.e., a test always brings a faulty circuit to a final (stable) state that is different from the one specified. Thus the circuit must be analyzed under all possible delay assignments. If both gates and wires can take arbitrary (but bounded) delays, i.e., the delay-insensitive model, the analysis can be done by *ternary* simulation using a third logic value, X, denoting a unknown or changing signal [6]. For other delay models, this analysis becomes computationally much harder [39].

Assuming that a few test vectors can generally test for most faults, we can use the fault simulator to efficiently generate test vectors with a high fault coverage. The approach is illustrated in Figure 9. After the fault coverage is above some limit the test generation enters a second phase where a fault-oriented test generation is applied for each of the remaining faults. No fault simulation in necessary in this phase. The two phases can be merged by combining the fault simulator with an appropriate cost function [2]. The cost function incorporates testability measures and guides the search for a test vector. The input vector

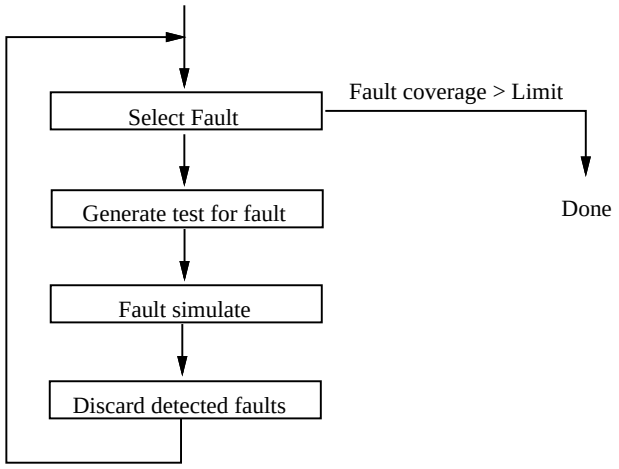

Figure 9

The use of fault simulation for test generation.

is changed until the cost drops below a given threshold, and at that point that vector is chosen. Depending on the cost function, the test generation can be guided to search for vectors that test for a group of faults or to search for specific faults. The cost function is changed dynamically, starting with a cost function that will put the circuit in a known initial state, then the cost function is changed to one that detects groups of faults, and finally to one that guides the search for the remaining faults.

5 Design for Testability

A level of controllability and observability may not be sufficient to test all possible faults. For example, to test for a premature firing, the circuit must be held in a state where the premature firing occurs and the faulty transition propagates to a primary output. This is not always possible. This section describes methods to increase the testability by adding test circuitry during the design phase, termed *design for testability*.

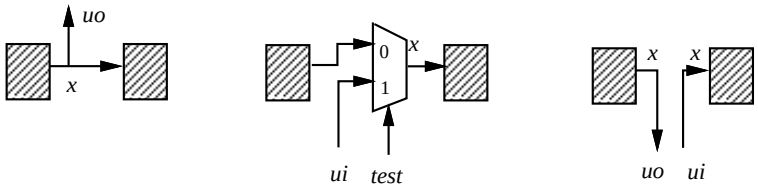

Figure 10

Introducing test points for signal x. An observation point (left), a control point (middle), and both (right).

The simplest way to increase testability is to introduce a *test point* into the circuit. Test points are of two types. An *observation point* is used to access an internal node by making

the node a primary output. A *control point* is used to set the value of an internal node from a primary input. A test point can also be both an observation and a control point, see Figure 10. Where to insert test points to minimize the total number is a difficult problem. Some heuristics are given in [14].

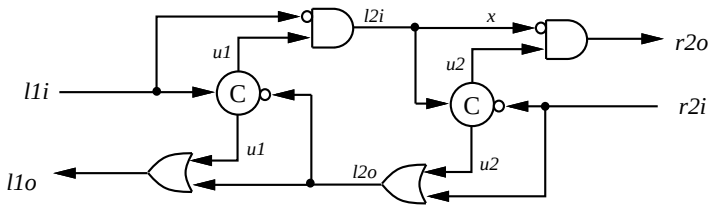

Figure 11

Two D-elements in series.

An example of the use of test points is shown in Figure 11. A premature firing that isn't testable exists when two D-elements are connected in series. As discussed above, a stuck-at-0 fault at the node x causes a premature firing of $r2o$ as $r2o$ then can fire before $l2i\downarrow$. To test for this fault, the circuit must be held in the state where the premature firing occurs. But $u2\uparrow$ causes $l2o\uparrow$ then $u1\downarrow$ and $l2i\downarrow$. The environment cannot prevent this sequence from happening. In order to test for this fault, a test point must be introduced in this loop. One possibility is to make $l2o$ a primary input and output, as shown to the right in Figure 10.

If the number of test points is small, the overhead is reasonable. But for more than a few test points, inserting test points is expensive in terms of I/O pins, which are normally a scarce resource. A way to reduce the number of I/O pins used for test points, at the cost of added test time, is to store the value of the test points in an internal register whose contents can be shifted in and out serially. This shift register can be implemented synchronously or asynchronously. A testable queue containing the values of the test points can be built from the FIFO element shown in Figure 5 [14]. This scheme requires only a few extra I/O pins independent of the number of test points.

Generalizing this idea leads to a popular method of simplifying test generation by introducing a *scan-path*. The registers in the circuit are extended to be *scan registers*, illustrated in Figure 12 with a conventional clocked register.

In normal operation (*test-mode* = 0), the scan registers work exactly as the original registers. In test mode, the scan registers form a shift register as the scan-output of one register is connected to the scan-input of the next. Their content can then be serially shifted out to the *scan-out* output, and new values can be shifted in on the *scan-in* input. Full observability and controllability is obtained for the values stored in the registers. By extending all registers in a circuit with scan capabilities, the circuit is divided into a scan-path with blocks of combination logic in between. The hard problem of generating test vectors for a sequential circuit is thus transformed into a much simpler problem of generating tests for blocks of combinational logic. The cost of simpler test generation is an increase in circuit area and potentially a longer test time because each test vector must be serially shifted in

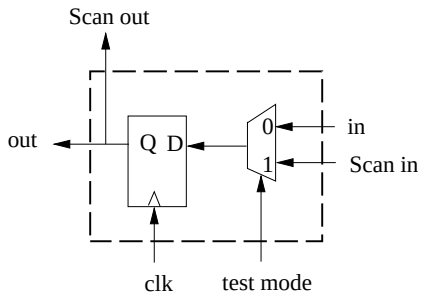

Figure 12

A scan-register.

and out. This cost can be reduced by having multiple scan-paths, trading off test time with I/O pins.

Counters are notoriously difficult to test because they contain many states and have a low controllability. Consider the n-bit counter shown in Figure 13. The toggle element (**tg**) steers input transitions to the outputs, X and Y, alternately, starting with the output marked with a "•". A two-phase (transition-) signaling protocol is used for request (*req*) and acknowledge (*ack*).

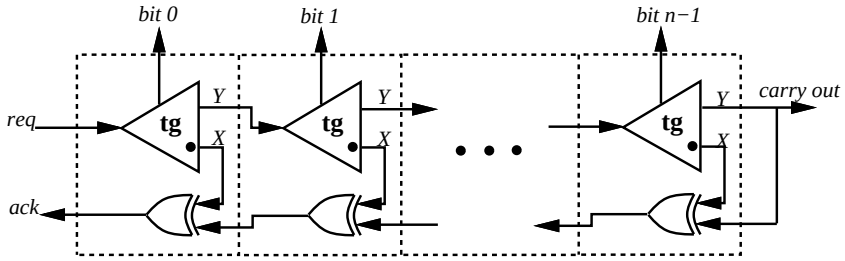

Figure 13

An n-bit asynchronous counter.

It is straightforward to come up with a test sequence for an n-bit counter: toggle the request signal 2^n times and observe the *carry out* signal. If the *carry* output does not change, the circuit has halted and a fault is present (assuming the counter is self-checking). One way to reduce the exponential test time is to partition the counter into m chunks of n/m bits and insert test points between the partitions. For example, a 16 bit counter can then be split into two 8-bit counters by adding a single test point, reducing the test time from 65536 cycles to 512 cycles (or even 256 if the two counters are tested concurrently). For large n many test points are needed. The number of additional I/O pins is reduced by introducing a scan-path through the toggle elements which are the only state holding elements in the counter. By doing so the state of the counter is made observable and controllable. The signal *req* is used as *scan-in* and *carry out* is used as scan out. Two extra signals are added: *test-mode* puts

the toggle element in scan-mode and *test-clock* is the clock used to scan data in and out of the scan register. In order to test the counter, only the signals *ack* and *carry out* have to be observable.

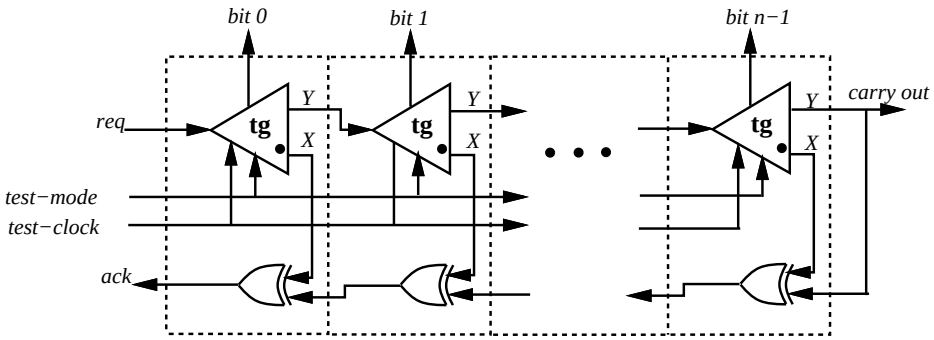

Figure 14

Extending an n-bit asynchronous counter with a scan-path.

Using the scan-approach, the counter can be tested in linear time. For a particular implementation [7], the extension of the toggle elements to scan elements increases the area of the counter by only 6% and the speed is reduced by about 7%. A comparable synchronous implementation of the counter has a 15% area increase to implement the scan-path and the test time is quadratic [16].

This example illustrates that even though generating the test sequences for a self-checking circuit is simple, it may take a long time to perform the test because the controllability is too low. A scan approach for cell-based designs that may also apply to asynchronous circuits is proposed in [36]. Each cell is extended with test circuitry that implements a scan-path through the cell. To reduce the test time, a test bus is introduced, which allows for multiple separate scan paths. The area increase is reported to be only approximately 6%. This approach seems appropriate if the cells are reasonable large. This is often the case for syntax-directed synthesis methods where each cell corresponds to a language construct in a high-level specification language (e.g., [5]).

The feasibility of the scan approach for asynchronous circuits has been demonstrated in [33] with a 144-bit scan-path in a systolic array. Asynchronous dual-rail combinational logic, implemented as PLAs, can be made fully testable by introducing a dual-rail scan path [12]. However, [14] shows that dual-rail combinational logic can be tested using standard test generation techniques, e.g., the D-algorithm.

An alternative approach to design for testability, which also addresses the problem of efficiently testing a circuit with low controllability, is described in [34]. Quasi-delay-insensitive circuits are synthesized from a CSP-like specification by translating the specification in a syntax-directed manner. Under the isochronic transition fault model, the synthesized circuits are self-checking by construction. Instead of adding a scan-path to the circuit to make it efficiently testable, a single test signal is added. In test mode, the operation of some of the

elements in the circuit is changed. Most notably, the element that sequentializes the communication on a particular channel is changed so that only a single communication is performed on the channel. This is sufficient to test the channel for stuck-at faults. The addition of the test circuitry is shown for the sequentializing element in Figure 15. By introducing the test

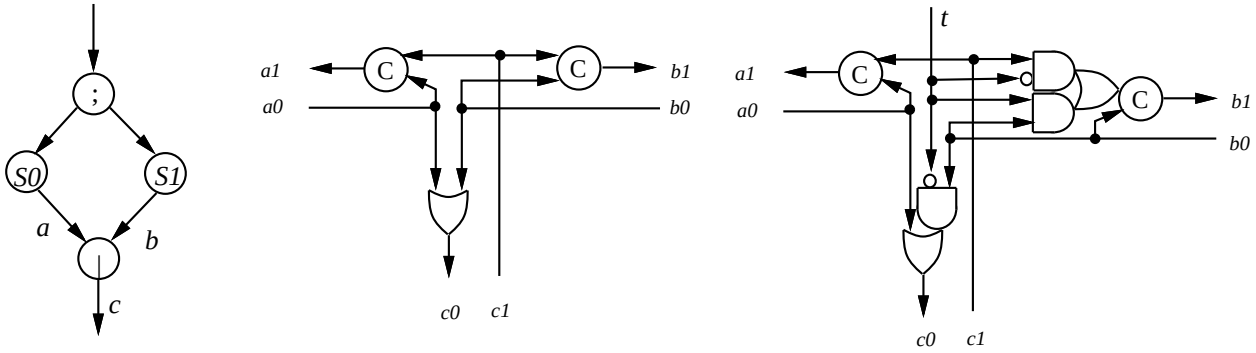

Figure 15

The sequential execution of two operations S_0 and S_1 that shares the channel c is translated into the structure shown on the left. The implementation of the operator that sequentializes the communication on a channel, "|", is shown in the middle. The addition of the test mode (the signal t) is shown on the right. Notice that when the circuit to the right is in test mode (t = true), the communication on the channel b is not propagated down to the c channel, but instead an acknowledge is generated immediately. This is the key that allows the circuit to be tested in linear time.

signal and modifying some of the basic building blocks, a synthesized circuit can always be tested in time linear in the size of the circuit. Because the circuit is self-checking (under the isochronic transition fault model) it is tested by executing a single computation while in test mode, and if a full handshake is performed on a particular channel the circuit is fault free. The cost is a considerable increase in area caused by the extra logic added to the basic circuit elements. However, one can trade off circuit area with test time by adding the test mode only to a subset of the sequencing elements.

Finally, [42] proposes a way to make an asynchronous state machine easily testable. The technique is a combination of the scan-technique with a clever state assignment. The flow table for an asynchronous state machine is extended with at most two extra state bits and a test input signal. By toggling the test input, a distinguishing sequence appears on an output, i.e., based on the output sequence both the present state and the number of stable states for the given input can be determined. This sequence is used to verify that the state machine has the correct number of stable states for a given input and can perform the transitions specified by the flow table. This process is called state machine identification [18]. Some care must be taken that a race-free state assignment is used. Different trade-offs between the length of the distinguishing sequences and the increase in number of states of the machine is exploited in [38]. The area overhead is minimal, but the state machine identification is a functional oriented test that takes time proportional to the number of states in the machine.

This limits the approach to small state machines. Also, the inputs must be controllable and the test-output must be observable. This can be achieved by adding a scan-path, at the expense of longer test time.

6 Path Delay Fault Testing

The classes of asynchronous circuits discussed in the previous sections have very few assumptions about the actual delays of the elements. In fact, the only timing assumption introduced is that some forks are isochronic. However, by designing circuits under absolute delay assumptions (for example that the delay of an inverter is between .5 and 2 ns), presumably the circuits can be made smaller and faster [31, 19]. In terms of testing the circuits, the consequence is that it must now be determined whether a fabricated circuit has the assumed delay properties. In order to do so a fault model that models delays must be used, therefore the stuck-at fault model is not appropriate.

This section describes a method for testing a circuit under a fault model that can model delay faults, called the *path delay fault model*. In this fault model, all paths between registers are considered. The circuit has a path delay fault if a given path in the fabricated circuit has a delay outside the specified interval. The path delay fault model is a more general fault model than the stuck-at fault models. Clearly, a path with a stuck-at fault has a delay outside the specified interval, as the delay will be infinite for either a rising or a falling transition.

To test a circuit under the path delay fault model, the delay of all paths in the circuit must be determined by a *path delay fault test*. The delay for a given path in the circuit, π, is tested by applying two vectors $\langle V_1, V_2 \rangle$ at times t_0 and t_1, respectively. The time between t_0 and t_1 is long enough to assure that all nodes in the circuit are stable at time t_1. The test vector pair has the property that when V_2 is applied after V_1, it causes a transition to occur at all nodes on π. The output of the circuit is latched at time t_2. If the latched value differs from the specification (which is the output corresponding to V_2), the delay on π is larger than $t_2 - t_1$, and a path delay fault is detected.

Consider the simple combinational circuit in Figure 16 implementing the function $F = a\overline{b} + \overline{b}c + b\overline{c}$. The delay (for a rising transition) of the path from a to F can be tested by the two vectors $V_1 = (a, b, c) = (0, 0, 0)$ and $V_2 = (1, 0, 0)$.

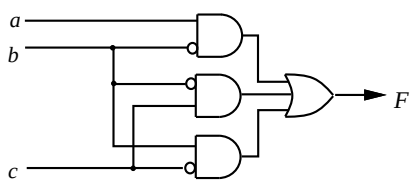

Figure 16

Implementation of the function $F = a\overline{b} + \overline{b}c + b\overline{c}$.

A *robust* test is a test for a path delay fault which is independent of delays in gates not on

the path under test. Test vector generation for robust path delay tests is described in [21].

A circuit can be made path delay fault testable by changing all state holding elements to scan elements. However, in contrast to the scan elements used in the scan-paths described in the previous section, the scan elements must be able to hold *two* values corresponding to a bit in the two vectors $\langle V_1, V_2 \rangle$.

Special care must be taken in the test generation to assure that the generated vectors do not generate hazards in the circuit independently of the gate delays, as a hazard could mask the presence of a delay fault. Hazards are not a problem in the regular scan approach, as the result is latched and scanned out when all signals are stable, i.e., at a time considerably larger than the longest path delay in the circuit. However, since the test for path delay faults is performed at the operation speed of the circuit, hazards are a problem when performing a delay test.

There may exist paths in a circuit that are not robust path delay fault testable. The circuit in Figure 16 is not robust path delay fault testable for the path from b through the topmost AND-gate (the term $a\overline{b}$). In order to test this path, the outputs of the two AND-gates not on the path must be kept low and without hazards, but this is not possible because one of these will be (a least momentarily) enabled when switching b.

In [17], a technique to make any Boolean (possibly redundant) function delay fault testable is presented. Let x be an input variable associated with a path that cannot be delay fault tested in the function F. F is decomposed into $F = xG + \overline{x}H + R$ such that x does not occur in G, H, and R, see Figure 17. Two test points, t_1 and t_2, are introduced

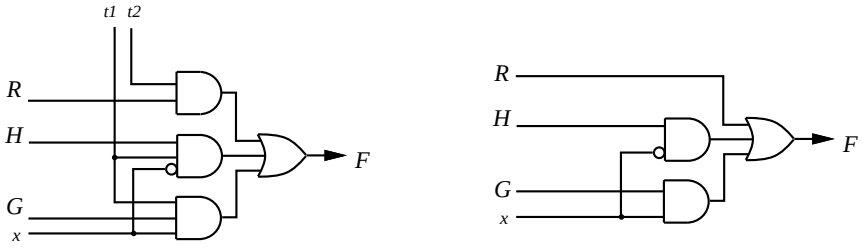

Figure 17

Making the path associated with x in the logic function F robust path delay fault testable. To the left is the general case and to the right is the case where the two functions $R + G$ and $R + H$ are both path delay fault testable.

(in normal operation these are set to **true**). These test points are used to select either the path from R to F or the path from G and H to F. The logic can be simplified if both the function $G + R$ and the function $H + R$ are path delay fault testable. Then the two added test points can be eliminated as shown to the right in Figure 17. This method is then applied recursively on the logic blocks G, H, and R. Clearly, this transformation increases the circuit area, decreases the operation speed, and increases the test time, but it turns out that most functions are robust path delay fault testable without any modifications (see [17, Table 1]).

Consider the example from Figure 16 where the path associated with b is not robust path delay fault testable. F can be rewritten as $F = b\overline{c} + \overline{b}(a + c)$. Thus, $x = b$, $G = \overline{c}$, $H = (a + c)$, and $R =$ false. The corresponding robust path delay fault testable circuit is shown at left in Figure 18. Eliminating the test points results in the circuit on the right in Figure 18.

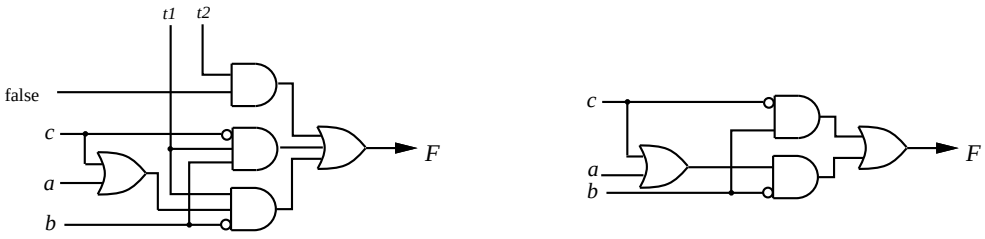

Figure 18

Path $\overline{b}$ from the function $F = a\overline{b} + \overline{b}c + b\overline{c}$ is made robust path delay fault testable.

An alternative approach to make all paths delay testable is called *variable phase splitting* [20]. The idea is to make both phases of the variables controllable. If both a variable, x, and its negated, $\overline{x}$, are used in a block of combinational logic, the variable is split into two independent variables, x' and x''. Under normal operation one will be the inverted version of the other, but under test they are individually controllable (for example both x' and x'' could be true simultaneously). A variable is split by storing it in a register which has outputs for both the true and negated version of the variable. The register is made into a scan register such that these two outputs are individually controllable in test mode. The scan register must still be able to hold two values for each of the outputs in order to perform the delay test. Thus, the register elements in this approach are even larger, but they eliminate the need for inserting test points and changing the logic to make a circuit fully path delay fault testable.

Testing a circuit under the path delay fault model is considerably more expensive than the other approaches described in this paper. The scan-registers must be able to hold two bits, extra signals are needed to control the test, and at least twice as many test vectors are needed. Furthermore, the skew on the clock to the scan-registers must be accurately controlled, as the worst case skew must be taken into account when determining the path delay. Thus, the overhead in terms of circuit area and test time is considerable, and it is unclear whether the benefits (in terms of speed and area efficiency) of circuits with absolute delay assumptions outweighs their testing overhead.

7 Conclusion

This paper has presented techniques for testing fabricated digital asynchronous circuits. For asynchronous circuits there is a strong relation between the timing assumptions made by the designer and the test approach necessary to test the fabricated circuit. The more assumptions the designer make about the delays in the circuit, the harder it becomes to test.

At one extreme is delay-insensitive circuits where *no* assumptions are made about the delays of the components. These are very simple to test because they always are self-checking. The only delay assumption in speed-independent and quasi-delay-insensitive circuits is that all or certain forks are isochronic. For these classes of circuits some faults (under the input stuck-at fault model) cause premature firings, and testing for these faults may require insertion of test points. For circuits where the actual delays must be within specified bounds for correct operation, the stuck-at fault model is not appropriate. Instead it must be verified that the actual delays are correct by performing a delay test. The path delay fault test requires that all registers are extended to scan-registers, making it expensive both in terms of circuit area and test time.

Many techniques traditionally used for synchronous circuits can be adapted to asynchronous circuits. Efficient test generation techniques for both combinational and sequential circuits can be applied with few modifications. Care must be taken that the test doesn't introduce hazards, a problem that doesn't exists for synchronous circuits. Also, a more detailed simulation is necessary when a fault simulator is used because faults can cause races (non-determinism) in an asynchronous circuit, and all possible behaviors must be considered. Design for testability techniques are also applicable to asynchronous circuits, though the trade-offs between circuit area, test time, and test coverage may be different.

Other techniques are unique for asynchronous circuits. Self-checking asynchronous circuits halt in the presence of faults, making this class of circuits easy to test. Whether a circuit is self-checking depends on the fault model and the timing assumptions under which the circuit is designed. The efficiency of testing self-checking circuits can be improved by introducing a test mode at the expense of increased circuit area.

The different test techniques represent different trade-offs between the quality and the cost of the test. Which one is appropriate depends on the circuit structure, the fault model used, and the delay assumptions made when designing the circuit. It may be beneficial to combine some of the approaches, for example identifying self-checking portions of the circuit and isolate them with scan-paths, thus achieving a small area overhead and easy test generation. An open question is what fault model is appropriate for asynchronous circuits. Most work has adopted the gate-level stuck-at fault model which has proven applicable for synchronous circuits, but it is not clear whether this model can be adapted to asynchronous circuits with the same level of coverage of physical faults.

References

[1] Miron Abromovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[2] V. D. Agrawal, K.-T. Cheng, and P. Agrawal. A directed search method for test generation using a concurrent simulator. *IEEE Transactions on Computer Aided Design*, 8(2):131–138, February 1989.

[3] Peter A. Beerel and Teresa H.-Y. Meng. Semi-modularity and self-diagnostic asynchronous control circuits. In *Advanced Research in VLSI, Proceedings of the 1991 University of California/Santa Cruz Conference*. The MIT Press, Cambridge, MA, 1991.

[4] Peter A. Beerel and Teresa H.-Y. Meng. Semi-modularity and testability of speed-independent circuits. *Integration, the VLSI journal*, 13(3):301–322, September 1992.

[5] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 262–265. IEEE Computer Society Press, November 1989.

[6] J. A. Brzozowski and C.-J. Seger. A unified framework for race analysis of asynchronous networks. *Journal of the ACM*, 36(1):20–45, January 1989.

[7] Gerald R. Carson and Gaetano Borriello. A testable CMOS asynchronous counter. *IEEE Journal of Solid-State Circuits*, 25(4), August 1990.

[8] Ilana David, Ran Ginosar, and Michael Yoeli. Self-timed is self-diagnostic. Technical report, Department of Computer Science, University of Utah, Salt Lake City, UT 84112, 1990.

[9] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*. Ph.D. thesis, Technische Universiteit Eindhoven, 1987.

[10] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, C-32(12):1137–1144, 1983.

[11] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, C-30(3):215–222, 1981.

[12] Dong S. Ha and Sudhaker M. Reddy. On testable self-timed logic circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 296–301. IEEE Computer Society Press, 1984.

[13] T. Hayashi, K. Hatayama, S. Ishiyama, and M. Takakura. Two test generation methods for sequential circuits. In *Proc. International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 1942–1945, 1989.

[14] Pieter J. Hazewindus. *Testing Delay-Insensitive Circuits*. Ph.D. thesis, California Institute of Technology, 1992.

[15] Lee A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.

[16] Mehdi Katoozi and Mani Soma. A testable CMOS synchronous counter. *IEEE Journal of Solid-State Circuits*, 5(23):1241–1248, October 1988.

[17] Kurt Keutzer, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. Synthesis for testability techniques for asynchronous circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 326–329. IEEE Computer Society Press, November 1991.

[18] Z. Kohavi. *Switching and Finite Automata Theory.* Computer Science Series. Tata McGraw-Hill Publishing, 1978.

[19] Luciano Lavagno, Kurt Keutzer, and Alberto L. Sangiovanni-Vincentelli. Synthesis of verifiably hazard-free asynchronous control circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI, Proceedings of the 1991 University of California/Santa Cruz Conference*, pages 87–102. The MIT Press, Cambridge, MA, 1991.

[20] Luciano Lavagno, Michael Kishinevsky, and Antonio Lioy. Testing redundant asynchronous circuits. Technical Report ID-TR:1993-124, Department of Computer Science, Technical University of Denmark, 1993.

[21] Chin J. Lin and Sudhakar M. Reddy. On delay fault testing in logic circuits. *IEEE Transactions on Computer Aided Design*, 6(5), September 1987.

[22] Alain J. Martin. From communicating processes to delay-insensitive circuits. Technical report, Department of Computer Science, California Institute of Technology, 1989. Caltech-CS-TR-89-1.

[23] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. The MIT Press, Cambridge, MA, 1990.

[24] Alain J. Martin et al. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Decennial Caltech Conference on VLSI*, pages 351–373. The MIT Press, Cambridge, MA, 1989.

[25] Alain J. Martin and Pieter J. Hazewindus. Testing delay-insensitive circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference*, pages 118–132. The MIT Press, Cambridge, MA, 1991.

[26] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 8(11):1185–1205, November 1989.

[27] Charles E. Molnar et al. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[28] S. Mourad. Sequential circuit testing. In *Proceedings COMPCON Spring*, pages 449–454. IEEE Computer Society Press, 1990.

[29] Yuzo Mukai and Yoshihiro Tohma. A method for the realization of fail-safe asynchronous sequential circuits. *IEEE Transactions on Computers*, C-23(7):736–739, July 1974.

[30] David E. Muller and W.S. Bartky. A theory of asynchronous circuits. In *The Annals of the Computation Laboratory of Harvard University. Volume XXIX: Proceedings of an International Symposium on the Theory of Switching, Part I.*, pages 204–243, 1959.

[31] C. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1992.

[32] Steven M. Nowick and David L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 318–321. IEEE Computer Society Press, November 1991.

[33] Deepak Rana, Steven P. Levitan, David A. Carlson, and Charles E. Hutchinson. A testable asynchronous systolic array implementation of an IIR filter. In *Proceedings of the IEEE 1986 Custom Integrated Circuits Conference*, pages 90–93. IEEE Computer Society Press, May 1986.

[34] Marly Roncken and Ronald Saeijs. Linear test times for delay-insensitive circuits: a compilation strategy. In S. Furber and M. Edwards, editors, *Proceedings of IFIP Working Conference on Asynchronous Design Methodologies*, pages 13–27, Manchester, UK, 31 March – 2 April, 1993. Elsevier Science Publishers.

[35] J.P. Roth, W.G. Bouricious, and P.R. Schneider. Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits. *IEEE Transactions on Electronic Computers*, EC-16(5):567–579, 1967.

[36] K. Sakashita, T. Hashizume, and T. Ohya. Cell-based test design method. In *Proc. International Test Conference*, pages 909–916. IEEE Computer Society Press, 1989.

[37] D. H. Sawin and G. K. Maki. Asynchronous sequential machines designed for fault detection. *IEEE Transactions on Computers*, C-32(3):239–249, March 1974.

[38] J. Saxena and D. K. Pradhan. Design for testability of asynchronous sequential circuits. In *Proceedings 1993 IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD '93)*, pages 518–522. IEEE Computer Society Press, October 1993.

[39] C.-J. Seger. The complexity of race detection in VLSI circuits. In Charles L. Seitz, editor, *Decennial Caltech Conference on VLSI*, pages 335–350. The MIT Press, Cambridge, MA, 1989.

[40] Charles L. Seitz and Wen-King Su. A family of routing and communication chips based on the Mosaic. In *Proceedings of the 1993 Symposium on Research on Integrated Systems*, pages 320–337. The MIT Press, Cambridge, MA, January 1993.

[41] Jørgen Staunstrup and Mark R. Greenstreet. Synchronized Transitions. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland/Elsevier, 1990.

[42] A. K. Susskind. A technique for making asynchronous sequential circuits readily testable. In *Proc. International Test Conference*, pages 842–846, 1984.

[43] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[44] V. I. Varshavsky, M. A. Kishinevsky, V. B. Marakhovsky, V. A. Peschansky, L. Y. Rosenblum, A. R. Taubin, and B. S. Tzirlin. *Self-timed Control of Concurrent Processes*. Kluwer Academic Publisher, 1990. (Russian edition: 1986).

[45] Ted E. Williams, M. Horowitz, R.L. Alverson, and T.S. Yang. A self-timed chip for division. In Paul Losleben, editor, *Proceedings of the Conference on Advanced Research in VLSI*, pages 75–95. The MIT Press, Cambridge, MA, March 1987.