Chapter 3

FINDING BUGS AND REPAIRING CIRCUITS

In the previous chapter we described the current design and verification methodologies at different design stages. In this chapter we take a closer look at the verification techniques used in these methodologies. Among the techniques available for functional verification, simulation-based verification is prevalent in the industry because of its linear and predictable complexity as well as its flexibility in being applied, in some form, to any design. However, simulation can only find bugs that can be exposed by the given stimuli. Therefore, unless all possible input scenarios can be covered by the stimuli, the correctness of the design cannot be guaranteed. To address this problem, formal methods have been developed to prove the correctness of the design under certain pre-defined properties. Nonetheless, their scalability is often limited because the proving process can be very complicated. In addition, developing properties may be as difficult as developing the design itself. Therefore, formal techniques are applied to only a small portion of the current designs. To overcome this problem, hybrid techniques that utilize both simulation and formal methods have been proposed. In this chapter, we first review simulation-based verification techniques. Next, we describe commonly used formal methods. Finally, we introduce the scan chain Design-for-Debugging (DFD) construct and the metal fix technique that facilitate post-silicon debugging.

3.1 Simulation-Based Verification

Simulation is the most commonly used technique for verifying the correctness of a design. In its simplest form, called direct test, engineers manually develop the test vectors that are applied to the design and then inspect their output responses. Developing the test suites, however, can be costly and time-consuming. In addition, scenarios not considered by the designers may be overlooked by the test developers as well. Therefore, techniques that automate
testbench generation have been proposed to avoid the bias from human engineers. A common methodology to this context is *constrained-random simulation*. It involves connecting a logic simulator with stimuli coming from a constraint-based random generator, i.e., an engine that can automatically produce random legal inputs for the design at a very high rate based on a set of rules (or constraints) derived from the specification.

A fast simulator is the core of simulation-based verification methodologies; therefore, in this section we review two commonly used simulation algorithms. Since the quality of test vectors generated in constrained-random simulation greatly affects the thoroughness of verification, we also review several solutions that improve this quality.

### 3.1.1 Logic Simulation Algorithms

Logic simulation mimics the operation of a digital circuit by calculating the outputs of the circuit using given input stimuli. For example, if a 0 is applied to the input of an inverter, logic simulation will produce a 1 on its output. Algorithms that perform simulation can be categorized into two major types: oblivious and event-driven [11]. In the oblivious algorithm, all gates are simulated at each time point. In event-driven simulation, value changes in the netlist are recorded, and only the gates that might cause further changes are simulated. Event-driven algorithms are potentially more efficient than oblivious algorithms because they only simulate the part of the netlist that had their values changed; however, the overhead to keep track of the gates that should be simulated is also a concern.

A typical oblivious simulation algorithm works as follows:

1. A linear list of gates is produced by levelizing the netlist. Gates closer to the primary inputs (i.e., those at lower levels of logic) are placed on the front of the list.

2. At each time point, all the gates in the list are simulated. Since gates with smaller levels of logic are simulated first, the simulation values at the inputs of the gate currently being simulated are always valid. As a result, the simulation value at the gate’s output is also correct.

Event-driven algorithms are more complicated than oblivious ones because the algorithms must keep track of the gates that need to be resimulated. One such algorithm, proposed by Lewis [90], is shown in Figure 3.1. Two phases are used in Lewis’ algorithm, including the node phase (also called the event phase) and the gate phase (also called the evaluation phase).

The node phase corresponds to the code labeled “fanout:”, and the gate phase corresponds to the code labeled “simulate:”. There are two lists that represent the state of the netlist: the first one is for the active nodes, while the