CHAPTER

2

Definitions

Key Objectives

- Types of models
- Black box versus white box
- Definition of a test

Functional verification requires that several elements are in place. It relies on the ability to simulate the design under test (DUT) with a specific input stimulus, observing the results of that stimulus on the design, and deciding if the results are correct. Figure 2-1 illustrates this basic verification environment.

There are several components in Figure 2-1. The entire environment is run in a simulator, since the device under test has generally not been built yet. There are many different simulators and simulation languages available. Two of the most common languages are Verilog and VHDL. Both of these languages have a variety of commercial simulators available. The basic purpose of the simulation language is to allow the behavior of a design to be concisely described and efficiently simulated.

Abstraction Levels

The level of detail in the DUT description may vary based on the abstraction level that is used. A more abstract model will describe the overall behavior of the design, but may leave some internal details out.

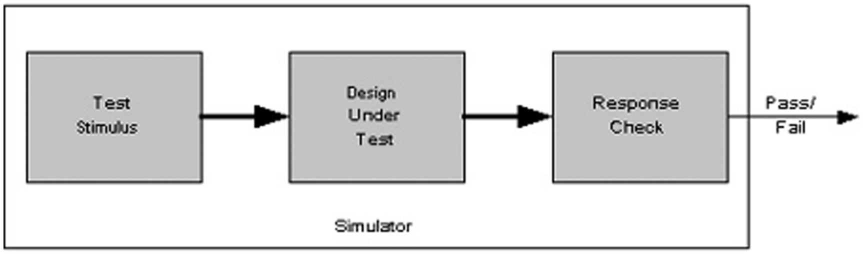

Figure 2-1: Simulation Environment

A less abstract model will provide more detail, and be closer to the actual implementation when the device is built. Usually, the more abstract models are faster to write and simulate, while a more detailed model is used to synthesize the actual device. It is not uncommon in complex designs to start with a more abstract model that focuses on the algorithmic and architectural issues of the system, then create a more detailed model that contains the information necessary to synthesize the actual device.

Behavioral Model

The highest abstraction level is usually referred to as a behavioral model. At this level, there may not be any timing information in the model. Some functions, such as error handling, may be missing. The goal at this level is usually to examine the basic operation of the design, and possibly the interactions among various components within the system. It is important that most of the tests and response checks are able to run with the behavioral model, since this permits both the model and the tests to be verified. As an example of a behavioral model, consider an integer multiplier. A real design may consist of a Wallace tree with multiple adders instantiated. A behavioral model can abstract all of that away. Listing 2-1 shows a behavioral multiplier in a pseudo language.

At the behavioral level, the abstraction may be very significant. The example in Listing 2-1 shows a multiplier that provides no

Listing 2-1: Behavioral Model

```
    input   [7:0]  a;
    input   [7:0]  b;
    output [15:0]  c;

  begin
    c = a * b;
  end
```

implementation detail at all. In this example, there is no timing information to indicate how many cycles it takes to perform a multiplication, nor any pipelining information. The algorithm is provided by the software language. There are many cases where the algorithm of an abstract block may be provided by existing software tools, even when another language is linked to the simulator to provide the algorithms.

An abstract model may provide cycle-level timing information. In the case of the multiplier example, a delay could be added to correspond to the estimated cycle delay of the model. In cases where a cycle-accurate model is required, then care must be taken that the cycle delay built into the behavioral model exactly matches the number of cycles needed for the implementation.

Register-Transfer-Level Model

A more detailed abstraction level is referred to as the register transfer level (RTL). In many systems, this is the level at which most design and verification effort is done. As the name implies, the RTL specifies where the storage elements are placed, providing accurate cycle-level timing information, but not subcycle timing, such as propagation delays. This level still shares many constructs with software, but requires several hardware constructs as well to simulate the parallel nature of hardware. As a result, hardware description languages such as Verilog and VHDL are commonly used for RTL modeling. At this level, all of the functional detail is usually available in the model.

This is because the RTL is often the bottom-most layer for creating a design.

At the RTL level, the placement of the registers is specified, but the asynchronous logic may still be abstract. Figure 2-2 shows an example of an RTL diagram.

The hardware description language (HDL) version of this code will still have software constructs. Listing 2-2 shows a segment of RTL code to implement the counter above. As always, the code is idealized. In this case, the reset conditions have been left out.

Note that while the registers have been defined for this logic, the model is still abstract, since the incrementer and the control logic are still expressed as software constructs. Nonetheless, with all the registers in place, this model will be a cycle-accurate model of the actual

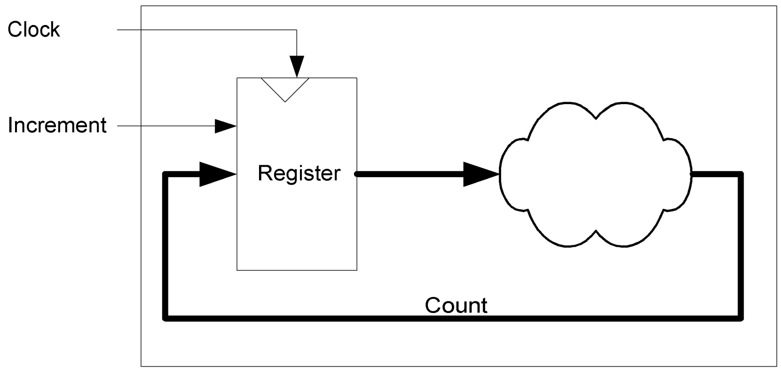

Figure 2-2: RTL Counter Diagram

Listing 2-2: RTL Counter Example

```
reg   [7:0]  count;

always @(posedge clock) begin
  if (increment == 1) count = count + 1;
  else                count = count;
end
```

device. From a simulation viewpoint, this model should behave almost exactly the same as the actual device will. This can be important when exact cycle behavior needs to be verified.

Gate-Level Model

Gate-level models tend to be the lowest abstraction layer used in the front-end design of a system. At the gate level, each individual logic element is specified, along with all the interconnections between the elements. For any reasonably complex design, this abstraction level is difficult to work with, since there is so much detail. Simulators tend to be quite slow at this level as well for the same reason. Despite the detail, there is still significant abstraction at this level, since the individual transistors are not described.

While most functional verification tends to be done at higher abstraction layers, there are times when a gate-level model may be important for verification. This will happen when the abstraction to RTL may hide some gate-level behavior that needs to be verified. This may occur in places where subcycle timing information becomes important, or where the RTL simulation may not be accurate. Some common examples where gate-level simulation may be of interest are for clock boundaries, where propagation delay, setup and hold times are important, or for reset conditions where "X" or "Z" logic states corrupt the behavior of the RTL abstraction.

Since these specific issues are usually a tiny fraction of the overall design, most verification is done at a higher abstraction layer. This not only improves simulation performance, but permits easier debug, since the more abstract models tend to be easier to read, understand, and debug.

Verification of a Design

In order to test the design model, the test must drive stimulus into the design under test, and examine the results of that stimulus.

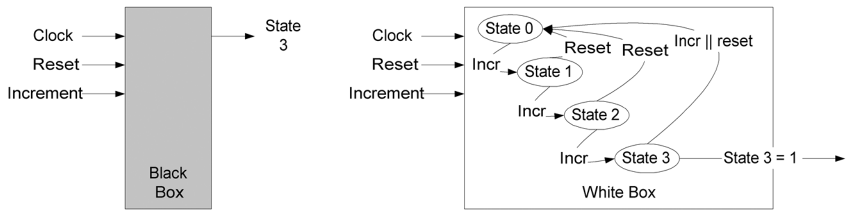

Figure 2-3: Black Box vs. White Box

The stimulus and checking can be done at the boundaries of a design, or it can examine the internals of design. This is referred to as black box versus white box testing.

To illustrate the difference between these approaches, consider a simple state machine that can increment or reset, as shown in Figure 2-3.

White Box Model

In a white box model, the test is aware of the inner workings of the device under test. This allows the test to directly monitor internal states to determine if the device is behaving correctly. For example, in Figure 2-3, the test may monitor the current state of the state machine. This allows the test to directly check that the state increments and resets correctly. The test obtains the immediate feedback about the state of the design when new stimulus is introduced, so errors may be reported earlier, and the test does not need to infer how the device under test is operating.

Even when a test is connected to only the periphery of a block, it may still be a white box test if it is operating with knowledge of the internals. If a test is written to cause a specific condition inside the block that is not specified in the design specification, then it is probably a white box test. As an example, a design may use a first-in, first-out queue (FIFO) to implement a function. If the test writer knows that the FIFO is four stages deep and writes a test for that, then the test is using knowledge of the implementation, rather than knowledge of the specification, and is thus a white box test.

Black Box Model

In the black box case, the test is limited to examining the input and output signals, and determining if the model is working correctly based only on information gathered from the outputs. In a black box model, the test environment either cannot or does not access the internals of the device under test. In some cases, the model may be encrypted, or otherwise be inaccessible. In other cases, the tests are written to look only at the inputs and outputs even when the internals are accessible.

The black-box-based test must infer internal operations without being directly able to observe them. In the example shown in Figure 2-3, the state can only be determined by asserting the increment signal until the State 3 output is asserted. There is no other measurement available.

As a result, a test may require more simulation cycles, and more logic to determine if the block is operating correctly. This is an additional burden on a black box test, but there are also some advantages.

A white box test is dependent on the internal structure of the logic. Any changes to the device may affect the test. One obvious example is synthesis. When an RTL model is synthesized to gates, the signals may change, preventing a white box model from running until it has been modified. Similarly, a white box test tends not to run on a design in multiple abstraction layers, since the internal workings of different abstraction layers may be significantly different.

A black box test, on the other hand, may run on a block that is designed as a behavioral, RTL, or gate-level model without modifications. Because it is looking only at the inputs and outputs of a component or design, it tends to be more re-usable as the scope of the design changes. This level of re-use may offset the additional complexity needed to infer the operation of the device.

There are other reasons for using black box models as well. First, a test is often focused on verifying the intent of a design, rather than the implementation. In this case, the details of how a design was constructed are irrelevant from a test viewpoint. The test is measuring only the final results, which would be found on the outputs. For this type of test, there is no reason to examine the internal state of the module.

A second reason that black box models are used is that they can separate the design from the verification. If a test is written by examining the inner workings of a design, it is possible that the test writer will write a test to verify a block based on reading the code from that block, rather than basing the test on a specification. This effectively corrupts the test so that little is gained from that test. For white box tests to be effective, they may use internal information, but care must be taken to ensure that the test is based on the specification, rather than on the RTL.

Both black box and white box tests have value. It is important to consider the goals and re-use requirements of the test to determine which is best suited for any particular situation. It is not uncommon for tests that are focused on the details of small blocks of logic to be white box tests, while those that look at broader multi-block issues are black box tests.

In some cases, a black box test will examine a few key points in a design, but for the most part use only the inputs and outputs. This is sometimes referred to as a gray box model, since it follows the basic concepts of the black box model, but with a few added outputs.

Definition of a Test

The simulation environment supports not only the device under test, but also the test stimulus generation and checking. The test itself may also reside in the simulator, written in the language that the simulator

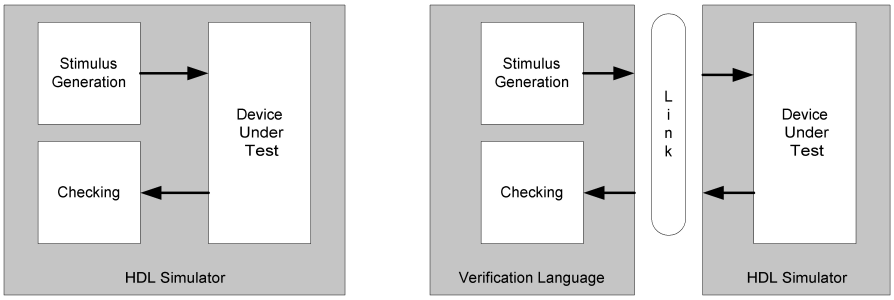

Figure 2-4: Tests Written in Simulator or in Verification Language

supports, or it may be written in a different language which is linked to the simulator.

A test that is written directly in the simulation language is easiest to understand since there is no additional software linkage. On the other hand, there are many verification constructs that may be useful when writing a test that are not supported by an HDL simulation language. Figure 2-4 illustrates what these two might look like.

There are several commercial companies that provide both an HDL simulator and a connected verification language, or a verification language that can connect to a simulator so that users do not need to deal with the link between the two languages.

Whichever language choice is used, a test is expected to drive stimulus into a device under test, and to observe the results in some fashion to determine that the DUT behaved as expected.