CHAPTER 2 VERIFICATION PLANNING

As stated in the previous chapter—and in several other published works—more effort is required to verify a design than to write the RTL code for it. As early as 1974, Brian Kernighan, creator of the C language, stated that "*Everyone knows debugging is twice as hard as writing a program in the first place*." A lot of effort goes into specifying the requirements of the design. Given that verification is a larger task, even more effort should go into specifying how to make sure the design is correct.

Every design team signs up for first-time success. No one plans for failures and multiple design iterations. But how is *first-time success* defined? How can resources be appropriately allocated to ensure critical functions of the design are not jeopardized without a definition of what functionality is critical? The *verification plan* is that specification. And that plan must be based on the intent of the design, not its implementation. Of course, corner cases created by the implementation, which are not apparent from the initial intent, have to be verified but that should be done once the initial intent-based verification plan has been completed.

This chapter will be of interest to verification lead engineers and project managers. It will help them define the project requirements, allocate resources, create a work schedule and track progress of the project of time. Examples in this chapter are based on OpenCore *Ethernet IP Core Specification*, Revision 1.19, November 27, 2002. This document can be found in the Examples section of the companion Web site:

```
http://vmm-sv.org
```

PLANNING PROCESS

The traditional approach of writing verification plans should be revised to take advantage of new verification technologies and methodologies. The new verification constructs in SystemVerilog offer new promises of productivity but only if the verification process is designed to take advantage of them. Individual computers, like individual language features, can improve the productivity of the person sitting in front of it. But taking advantage of the network, like the synergies that exist among those language features, can be achieved only by redesigning the way an entire business processes information; and it can dramatically improve overall efficiency.

The traditional verification planning process—when there is one—involves identifying testcases targeting a specific function of the design and describing a specific set of stimulus to apply to the design. Sometimes, the testcase may be self-checking and looks for specific symptoms of failures in the output stream observed from the design. Each testcase is then allocated to individual engineers for implementation. Other than low-level bus-functional models directly tied to the design's interface, little is reused between testcases. This approach to verification is similar to specifying a complete design block by block and hoping that, once put together, it will meet all requirements.

A design is specified in stages: first requirements, then architecture and finally detailed implementation. The verification planning process should follow similar steps. Each step may be implemented as separate cross-referenced documents, or by successive refinement of a single document.

Functional Verification Requirements

The purpose of defining functional verification requirements is to identify the verification requirements necessary for the design to fulfill the intended function. These requirements will form the basis from which the rest of the verification planning process will proceed. These requirements should be identified as early as possible in the project life cycle, ideally while the architectural design is being carried out. It should be part of a project's technical assessment reviews.

It is recommended that the requirements be identified and reviewed by a variety of stakeholders from both inside and outside the project team. The contributors should include experienced design, verification and software engineers so that the requirements are defined from a hardware and a software perspective. The reviews are designed to ensure that the identified functional verification requirements are complete.

Rule 2-1 — *A definition of what the design does shall be specified.*

Defining what the design does—what type of input patterns it can handle, what errors it can sustain—is part of the verification requirements. These requirements ensure that the design implements the intended functionality. These requirements are based on a functional specification document of the design agreed upon by the design and verification teams.

These requirements are outlined, separate from the functional specification document.

Example 2-1. Ethernet IP Core Verification Requirements

```
R3.1/14/0 Packets are limited to MAXFL bytes
R3.1/13/0 Does not append a CRC
R3.1/13/1 Appends a valid CRC
R4.2.3/1  Frames are transmitted
```

Rule 2-2 — *A definition of what the design must not do shall be specified.*

Defining what makes the behavior of the design incorrect is also part of the verification requirements. These requirements ensure that functional errors in the design will not go unnoticed. The section titled "Response Checking" on page 31 specifies guidelines on how to look for errors.

A functional specification document is concerned with specifying the intended behavior. That is captured by Rule 2-1. Verification is concerned with detecting errors. But it can only detect errors that it is looking for. The verification requirements must outline which errors to look for. There is an infinite number of ways something can go wrong. The verification requirements enumerate only those errors that are relevant and probable, given the functionality and architecture of the design.

Rule 2-3 — *Any functionality not covered by the verification process shall be defined.*

It is not possible to verify the behavior of the design under conditions it is not expected to experience in real life. The conditions considered to be outside the usage space of the design must be outlined to clearly delineate what the design is and is not expected to handle.

For example, a datacom design may be expected to automatically recover from a parity error, a complete failure of the input protocol or a loss of power. But a

processor may not be expected to recover from executing invalid instruction codes or a loss of program memory.

Example 2-2. Ethernet IP Core Verification Requirements

```
R3.1/9/0  Frames are lost only if attempt limit is
          reached
R4.2.3/2  Frames are transmitted in BD order
```

Rule 2-4 — *Requirements shall be uniquely identified.*

Each verification requirement must have a unique identifier. That identifier can then be used in cross-referencing the functional specification document, testcase implementation and functional coverage points.

Rule 2-5 — *Requirement identifiers shall never be reused within the same project.*

As the project progresses and the design and verification specifications are modified, design requirements will be added, modified or removed. Corresponding verification requirements will have to be added, modified or removed. When adding new requirements, never reuse the identifier of previously removed verification requirements to avoid confusion between the obsolete and new requirements.

Rule 2-6 — *Requirements shall refer to the design requirement or specification documents.*

The completeness of the functional verification requirements is a critical aspect of the verification process. Any verification requirement that is missing may cause a functional failure in the design to go unnoticed. Cross-referencing the functional verification requirements with the design specification will help ensure that all functional aspects of the design are included in the verification plan.

Furthermore, verification is complex enough without verifying something that is not ultimately relevant to the final design. If something is not specified, don’t verify it. Do not confuse this idea with an incomplete specification. The former is a "don’t care." The latter is a problem that must be fixed.

Recommendation 2-7 — *Requirements should be ranked.*

Not all requirements are created equal. Some are critical to the correct operation of the design, others may be worked around if they fail to operate. Yet others are optional and included for speculative functionality of the end product.

Ranking the requirements lets them be prioritized. Resources should be allocated to the most important requirements first. The decision to tape-out the design should similarly be taken when the most important functional verification requirements have been met.

Example 2-3. Ethernet IP Core Verification Requirements Ranking

```
R3.1/14/0 Packets are limited to MAXFL bytes SHOULD
R3.1/14/1 Packets can be up to 64kB          SHOULD
R3.1/14/2 Packets can be up to 1500 bytes    MUST
```

Recommendation 2-8 — *Requirements should be ordered.*

Many requirements depend on the correct operation of other requirements. The latter requirements must be verified first. Dependencies between requirements should be documented.

For example, verifying that all configuration registers can be correctly written to must be completed before verifying the different configurations.

Example 2-4. Ethernet IP Core Verification Requirements Order

```
R4.2.3/1  Frames are transmitted
R3.1/13/0 Does not append a CRC
R3.1/14/2 Packets can be up to 1500 bytes
R3.1/13/1 Appends a valid CRC
```

Recommendation 2-9 — *The requirements should be translated into a functional coverage model.*

A functional coverage model allows the automatic tracking of the progress of the verification implementation. This model also enables a coverage-driven verification strategy that can leverage automation in the verification environment to minimize the amount of code necessary to implement all of the requirements of the functional verification. More details on functional coverage modeling and coverage-driven verification can be found in Chapter 6. For example, Example 6-8 shows how the requirements shown in Example 2-5 can be translated into a functional coverage model.

Example 2-5. Ethernet IP Core Verification Requirements Coverage Model

```
R3.1/14/0 Packets are limited to MAXFL bytes
          At least one packet transmitted with length:
                  MAXFL-4
                  MAXFL-1
                  MAXFL
                  MAXFL+1
                  MAXFL+4
                  65535
          MAXFL set to
                  Default (1536)
                  1518
                  1500
          HUGEN set to
                  0
                  1
          Cross coverage of
                  frame length x MAXFL x HUGEN value
```

Recommendation 2-10 —*Implementation-specific requirements should be specified using coverage properties.*

Some functional verification requirements are dictated by the implementation chosen by the designer and are not apparent in the functional specification of the design. These functional verification requirements create corner cases that only the designer is aware of. These requirements should be specified in the RTL code itself through coverage properties.

For example, the nature of the chosen implementation may have introduced a FIFO (first-in, first-out). Even though the presence of this FIFO is not apparent in the design specification, it still must be verified. The designer should specify coverage properties to ensure that the design was verified to operate properly when the FIFO was filled and emptied.

Note that if the FIFO was never supposed to be completely filled, as assertion should be used on the *FIFO is Full* state instead of a coverage property.

Verification Environment Requirements

The primary aim of this step is to define the requirements of the verification infrastructure necessary to produce a design with a high degree of probability of being bug-free. Based on the requirements identified in the previous step, this step identifies the resources required to implement the verification requirements.

Rule 2-11 — *Design partitions to be verified independently shall be identified.*

Not all design partitions are created equal. Some implement critical functionality with little system-level controllability or observability. Others implement day-to-day transformations that are immediately observable on the output streams. The physical hierarchy of a design is architected to make the specification of the individual components more natural and to ease their integration, not to balance the relative complexity of their functional verification.

Some functional verification requirements will be easier to meet by verifying a portion of the design on its own. Greater controllability and observability can be achieved on smaller designs. But smaller partitions also increases their number, which increases the number of verification environments that must be created and increases the verification requirements of their integration.

The functional controllability and observability needs for each verification requirement must be weighed against the cost brought about by creating additional partitions.

Recommendation 2-12 —*Reusable verification components should be identified.*

Every independently verified design presents a set of interfaces that must be driven or monitored by the verification environment. A subset of those interfaces will also be presented by system-level verification of combinations of the independently-verified designs.

Interfaces that are shared across multiple designs—whether industry-standard or custom-designed—should share the same transactors to drive or monitor them. This sharing will reduce the number of unique verification components that will need to be developed to build all of the required verification environments. To that end, it will often be beneficial for the designs themselves to use common interfaces to facilitate the implementation of the functional verification task.

Different designs that share the same physical interfaces may have different verification requirements. The verification components for those interfaces must be able to meet all of those requirements to be reusable across these environments.

The opportunity for reusable verification components may reside at higher layers of abstraction. Even if physical interfaces are different, they may transport the same protocol information. The protocol layer that is common to those interfaces should be captured in a separate, reusable verification component. For example, MII (media-independent interface) and RMII (reduced media-independent interface) are two

physical interfaces for transporting Ethernet media access controller (MAC) frames. Although they have different signals, they transport the same data formats and obey the same MAC-layer protocol rules. Thus, they should share the same MAC frame descriptor and MAC-layer transactor.

Recommendation 2-13 —*Models of the design at various levels of abstraction should be identified.*

Many functional verification requirements do not need a model of the detailed implementation—such as a RTL or gate-level model—to be met. Furthermore, requirements that can be met with a model at a higher level of abstraction can do so with much greater performance. Some requirements, such as software validation, can be met only with difficulty if only an implementation-level model is available. Having a choice of models of the design at various levels of abstraction can greatly improve the efficiency the verification process.

Rule 2-14 — *The supported design configurations shall be identified.*

Designs often support multiple configurations. The verification may focus on only a subset of the configurable parameters first and expand later on. Similarly, the design may implement some configurable aspect before others with the verification process designed to follow a similar evolution.

The configuration selection mechanism should also be identified. Are only a handful of predetermined configurations going to be verified? Or, is the configuration going to be randomly selected? If the configuration is randomly selected, what are the relevant combinations of configurable parameters that should be covered? Constraints can be used to limit a random configuration to a supported configuration. Functional coverage should be used to record which configurations were verified.

Some configurable aspects require compile-time modification of the RTL code. For example, setting a top-level parameter to define the number of ports on the device or the number of bits in a physical interface is performed at compile-time. The model of the design is then elaborated to match. In these conditions, it is not possible to randomize the configuration because, by the time it is possible to invoke the `randomize()` method on a configuration descriptor, the model of the design has already been elaborated—and configured.

A two-pass randomization of the design configuration may have to be used. On the first pass, the configuration descriptor is randomized and an RTL parameter-setting configuration file is written. On the second pass, the RTL parameter-setting configuration file is loaded with the model of the design, and the same seed as in the

first pass, is reused to ensure the same configuration descriptor—used by the verification environment—is randomly generated.

Example 2-6. Supported Ethernet IP Core Configurations

```
- Variable number of TxBD (TX_BD_NUM)
    - If TX_BD_NUM == 0x00: TX disabled
    - If TX_BD_NUM == 0x80: Rx disabled
- MAXFL (PACKETLEN.15-0, MODER.14)
- Optional CRC auto-append (MODER.13)
```

Rule 2-15 — *The response-checking mechanisms shall be identified.*

The functional verification requirements describe what the design is supposed to do when operating correctly. It should also specify what kind of failures the design should not exhibit. The self-checking structure can easily determine if the right thing was performed. But it is much more difficult to determine that no wrong things were done in the process. For example, matching an observed packet with one of the expected packets is simple: If none match, the observed packet is obviously wrong. But if one matches, the question remains: Was it the packet that should have come out next?

The self-checking mechanisms can report only failures against expectations. The more obvious the symptoms of failures are, the easier it is to verify the response of the design. The self-checking mechanisms should be selected based on the anticipated failures they are designed to catch and the symptoms they present.

Some failures may be obvious at the signal level. This type of response is most efficiently verified using assertions. Some failures may be obvious as soon as an output transaction is observed. This type of response checking can be efficiently implemented using a scoreboarding mechanism. Other failures, such as performance measurements or fairness of accesses to shared resources, require statistical analysis of an output trace. This type of response checking is more efficiently implemented using an offline checking mechanism.

The available self-checking mechanisms are often dictated by the available means of predicting the expected response of the design. The existence of a reference or mathematical model may make an offline checking mechanism more cost effective than using a scoreboard.

Rule 2-16 — *Stimulus requirements shall be identified.*

It will be necessary to apply certain stimulus sequences or put the design into certain states to meet many of the functional verification requirements. Putting the design into a certain state is performed by applying a certain stimulus sequence. Thus, the problem is reduced to the ability of creating specific stimulus sequences to meet the functional verification requirements.

Traditionally, a directed sequence of stimulus was used to meet those requirements. However, the methodology presented in this book recommends the use of random generators to automatically generate stimulus to avoid writing a large number of directed testcases. Should the random stimulus fail to generate the required stimulus sequences, they are to be constrained to increase the probability that they will generate them in subsequent simulations.

The ability to constrain a random generator to create the required stimulus sequences does not happen by accident. Generators must be designed based on the stimulus sequences required by the verification requirements. They must offer the mechanisms to express constraints that will ultimately force the designed stimulus patterns to be generated as part of a longer random stimulus stream. If it remains unlikely that the required stimulus sequence will be randomly generated, then a directed stimulus sequence has to be used.

Example 2-7. Ethernet IP Core Stimulus Requirements

```
- Random configuration
  - Maximum packet length (PACKETLEN.MAXFL, MODER.HUGEN)
  - Appending of CRC (MODER.CRCEN)
  - Number of transmit buffer descriptors (TX_BD_NUM)
- Transmitted Packets
  - With and without CRC (TxBD.CRC)
  - Good & bad CRC
    - Bad CRC only if MODER.CRCEN or TxBD.CRC == 0
  - Various length (tied to maximum packet length)
```

Rule 2-17 — *Trivial tests shall be identified.*

Trivial tests are those that are run on the design first. Their objective is not to meet functional verification requirements, but to ascertain that the basic functionality of the design operates correctly before performing more exhaustive tests. Performing a write cycle followed by a read cycle, injecting a single packet or executing a series of null opcodes are examples of trivial tests.

Trivial tests need not be directed. Just as they can be used to determine the basic liveliness of the design, they can be used to determine the basic correctness of the verification environment. A trivial test may be a simple constrained-random test constrained to run for only a very short stimulus sequence. For example, a trivial test could be constrained to last for only two cycles: the first one being a write cycle, the second one a read cycle and both cycles constrained to use the same address.

The verification environment must be designed to support the creation of the trivial tests.

Example 2-8. Trivial Tests for Ethernet IP Core

```
- Rx disabled, transmit 1 packet
- Tx disabled, receive 1 packet
```

Recommendation 2-18 —*Error injection mechanisms should be defined.*

Designs may be able to sustain certain types of errors or stimulus exceptions. These exceptions must be identified as well as the mechanism for injecting them. Then, it is necessary to specify how correctness of the response to the exception will be determined.

Based on the verification requirements, it is necessary to identify the error injection mechanisms required in the verification environment. For low-level designs, it may be possible to inject errors at the same time as stimulus is being generated. For more complex systems with layered protocols, low-level errors are often impossible to accurately describe from the top of the protocol stack. Furthermore, there may be errors that have to be injected independent of the presence of high-level stimulus.

Exceptions may need to be synchronized with others stimulus—such as interrupt requests synchronized with various stages in the decode pipeline. Synchronizing an exception stream with a data stream may require using a multi-stream generator (see “Multi-Stream Generation” on page 236).

Example 2-9. Ethernet IP Core Error Injections

```
- Collisions at various symbol offsets (early, latest
early, earliest late, late)
- Collide on all transmit attempts
- Bad CRC on Rx frame
- Bad DA on Rx frame when MODER.PRO == 0
```

Recommendation 2-19 —*Data sampling interfaces for functional coverage should be identified.*

A functional coverage model should be used to track the progress toward the fulfilment of the functional verification requirements. The functional coverage model will monitor the verification environment and the design to ensure that each requirement has been met.

This monitoring requires that a *signature* is used in the design or verification environment to indicate that a particular verification requirement has been met. By observing the signature, the functional coverage model can record that the requirement corresponding to the signature's coverage point has been met.

For the coverage model to be able to observe those signatures, there must be set data sampling mechanisms. These mechanisms let the relevant data be observed by the functional coverage model. Data that is in different sampling domains or at the wrong level of abstraction will require a significant amount of processing before it can be considered suitable for the functional coverage model. Planning for a suitable data sampling interface up front will simplify the implementation of the functional coverage model.

Example 2-10. Coverage Sampling Interfaces for Ethernet IP Core

```
- DUT configuration descriptor
- Tx Frame after writing to TxBD
- TxBD configuration descriptor after Tx frame written
```

Recommendation 2-20 —*Tests to be ported across environments should be identified.*

To verify the correctness of the design integration or its integrity at different implementation stages, it may be necessary to port some tests to different environments. For example, some block-level tests may need to be ported to the system-level environment. Another example is the ability to take a simulation test and reuse it in a lab set up on the actual device. Yet another example is the ability to reproduce a problem identified in the lab in the simulation environment.

Tests that must be portable will likely have to be restricted to common features available in the different environments they will execute in. It is unlikely that these tests will be able to arbitrarily stress the capability of the design as much as a particular environment allows them to. Due to the different functional verification requirements met by the different verification environments, it is not realistic to expect to be able to port all tests from one environment to another.

Verification Implementation Plan

The primary aim of implementing the functional verification plan is to ensure that the implementation culminates in exhaustive coverage of the design and its functionality within the project time scales. The implementation is based on the requirements of the verification environments as outlined above.

This implementation plan should be started as early as possible in the project life cycle. Ideally, it should be completed before the start of the RTL-coding phase of the project and before any verification testbench code is written. This step is necessary to produce a design with a high degree of probability of being bug-free.

Recommendation 2-21 —*Functional coverage groups and coverage properties should be identified.*

The functional verification requirements should be translated into a functional coverage model to automatically track the progress of the verification project. A functional coverage model is implemented using a combination of `covergroup` or `cover property`. Which one is used depends on the nature of the available data sampling interface and the complexity of the coverage points.

Coverage properties are better at sampling signal-level data in the design based on a clock signal. But they can implement only a single coverage point. Coverage groups are better at sampling high-level data in the verification environment and can implement multiple coverage points that use the same sampling interface

Chapter 6 provides more guidelines for implementing functional coverage models.

Recommendation 2-22 —*Configuration and simulation management mechanisms should be defined.*

It must be easy—not just possible—to reproduce a simulation. It is necessary that there be a simple mechanism for ensuring that the exact model configuration used in a simulation be known. Which version of what source files, tools and libraries were used? Similarly, it must be simple to record and reissue the exact simulation command that was previously used —especially using the same random seed. Command-line options cannot be source-controlled. Therefore a script should be used to set detailed command-line options based on broad, high-level simulation options.

Recommendation 2-23 —*Constrainable dimensions in random generators should be defined.*

The random generators must be able to be constrained to generate the required stimulus sequences. Constraining the generators may involve defining sequences of data. But it also may involve coordinating multiple independent data streams onto a single physical channel or parallel channels, each stream itself made up of data sequence patterns. Constraints, state variables and synchronization events may need to be shared by multiple generator instances.

Controlability of the randomization process require the careful design of the data and transaction descriptor structures that are randomized and the generators that randomize them. The ability to constrain the generated data to create detailed stimulus scenarios tends to require more complex randomization processes. It may be more efficient to leave a few complex stimulus sequences as directed stimulus, and leave the bulk of the data generation to a simple randomization process.

Recommendation 2-24 —*Stimulus sequences unlikely to be randomly generated should be identified.*

There are some stimulus requirements that will remain unlikely to be automatically generated. Rather than complicate the random generators to create them or have to specify an overly complicated set of constraints to coerce the generators, it may be easier to specify them as directed stimulus sequences.

Directed stimulus sequences need not be for the entire duration of a simulation. They may be randomly injected as part of a random stimulus stream.

Recommendation 2-25 —*End-of-test conditions should be identified.*

When a test is considered done is an apparently simple but important question. Running for a constant amount of time or data may hide a problem located in a deeper state or by data being constantly pushed out by the forward pressure created be subsequent stimulus. Additional termination conditions could be defined: once a certain number of error messages have been reported, once a certain level of coverage has been hit, a watchdog timer has expired or the design going idle—whatever *idle* is must also be defined. The end-of-test condition could be created by only one condition or require a combination of the termination conditions.

Even when the end-of-test condition has been identified, how the simulation ends gracefully should be specified. There may be data that must be drained from the

design or statistics registers to be read. The contents of memories may have to be dumped. The testbench may have to wait until the design becomes idle.

Example 2-11. End of Test Conditions for Ethernet IP Core

```
- After N frames have been transmitted
- After M frames have been received
- If TxEN and interrupt not asserted for more than X
cycles
```

RESPONSE CHECKING

Rule 2-2 requires the enumeration of all errors that must be detected by the verification environment. These detection mechanisms require a strategy for predicting the expected response and to compare the observed response against those expectations. This section focuses on these strategies. Guidelines for the implementation of the self-checking structure can be found in section titled "Self-Checking Structures" on page 246.

It is difficult to describe a methodology for checking the response of a design because that response is unique to that design. Response checking can be described only in general terms. A broad outline of various self-checking structures can be specified. The availability in the SystemVerilog language of high-level data structures greatly facilitates the implementation of response checking. But it is not possible to describe the details of its overall implementation without targeting it to a specific design.

With traditional directed testcases, because the stimulus and functionality of the design are known, the expected response may be intellectually derived up front and hard-coded as part of the directed test. With random stimulus, although the functionality is known, the applied stimulus is not. The expected response must by computed based on the configuration and functionality of the design. The observed response is then compared against the computed response for correctness.

It is important to realize that the response-checking structure in a verification environment can only identify problems. Correctness is inferred from the failure to find inconsistencies. If the response-checking structure does not explicitly check for a particular symptom of failure, it will remain undetected. The functional verification requirements must include a definition of all possible symptoms of failure.

Recommendation 2-26 —*Response checking should be separate from stimulus.*

In directed tests, the response can be hardcoded in parallel with the stimulus. Thus, it can be implemented in a more ad-hoc or distributed fashion, in the same `program` that implements the stimulus. However, it is better to treat the response checking as an independent function.

Response checking that is hardcoded with the stimulus tends to focus on the symptoms of failure of the feature targeted by the directed test. This coding style causes functionality to be repeatedly checked in tests that focus on the same feature. But a test targeting a specific feature may happen to exercise an unrelated fault. If the response checking is concerned only with the feature being verified, then the failure will not be detected. This style of response checking may allow errors to go unnoticed if they occur in another functional aspect of the design.

By separating the checking from the stimulus, all symptoms of failures can be verified at all times.

Embedded Monitors

Response is generally understood as being observed on the external outputs of the design under verification. However, limiting response to external interfaces only may make it difficult to identify some symptoms of failure. If the verification environment does not have a sufficient degree of observability over the design, much effort may be spent trying to determine the correctness to an internal design structure because it is too far removed from the external interfaces. This problem is particularly evident in systems where internal buses or functional units may not be directly observable from the outside.

Suggestion 2-27 — *Design components can be replaced by transactors.*

Transactors need not be limited to interfacing with external interfaces. Like embedded generators described in section titled "Embedded Stimulus" on page 226, monitors can mirror or even replace an internal design unit and provide observability over that unit’s interfaces. The transaction-level interface of the embedded monitor remains externally accessible, making the mirrored or replaced unit interfaces logically external.

For example, an embedded RAM block could be replaced with a reactive transactor (slave), as illustrated in Figure 2-1. Correctness could be determined, not by dumping

or replicating the entire content of the memory, but by observing and fulfilling—potentially injecting errors—each memory access in real time.

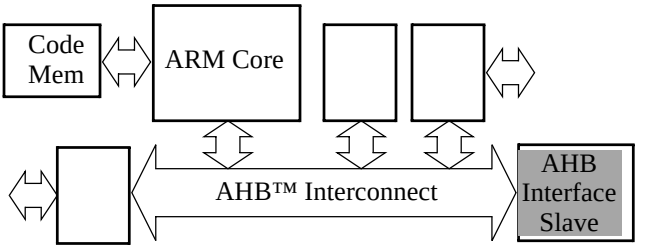

Figure 2-1. Replacing a Slave Unit with a Reactive Transactor

This approach is implementation-dependent. As recommended by Recommendation 2-32, assertions should be used to verify the response based on internal signals. However, assertions may not provide the high-level capabilities required to check the response. Assertions are also unable to provide stimulus, and thus cannot be used to replace a reactive transactor.

Assertions

The term *assertion* means a statement that is true. From a verification perspective, an assertion is a statement of the expected behavior. Any detected discrepancy in the observed behavior results in an error. Based on that definition, the entire testbench is just one big assertion: It is a statement of the expected behavior of the entire design. But in design verification—and in this book—assertion refers to a *property* expressed using a *temporal expression*.

Using assertions to detect and debug functional errors has proven to be very effective as the errors are reported near—both in space and time—the ultimate cause of the functional defect. But despite their effectiveness, assertions are limited to the types of properties that can be expressed using clocked temporal expressions. Some statements about the expected behavior of the design must still be expressed—or are easier to express—using behavioral code. Assertions and behavioral checks can be combined to work cooperatively. For example, a protocol checker can use assertions to describe the lower-level signaling protocol and use behavioral code to describe the higher-level, transaction-oriented properties.

Assertions work well for verifying local signal relationships. They can efficiently detect errors in handshaking, state transitions and physical-level protocol rules. They can also easily identify unexpected or spurious conditions. On the other hand, assertions are not well suited for detecting data transformation, computation and ordering errors. For example, assertions have difficulties verifying that all valid

packets are routed to the appropriate output ports according to their respective priorities.

The following guidelines will help identify which response-checking requirements should be implemented using assertions or behaviorally in the verification environment. More details on using assertions can be found in Chapters 3 and 7. Typical response-checking structures in verification environments are described in “Scoreboarding” on page 38, “Reference Model” on page 39 and “Offline Checking” on page 40.

Recommendation 2-28 —*Assertions should be limited to verifying physical-level assumptions and responses.*

Temporal expressions are best suited to cycle-based physical-level relationships. Although temporal expressions can be stated in terms of events representing high-level protocol events, the absence of a clock reference makes them more difficult to state correctly. Furthermore, the information may already be readily available in a transactor, making the implementation of a behavioral check often simpler.

Rule 2-29 — *A response-checking requirement that must be met on different levels of abstraction of the design shall be implemented using procedural code in the verification environment.*

Assertions are highly dependent on RTL or gate-level models. They cannot be easily ported to transaction-level models. Any response checking that must be performed at various abstraction levels of the design is better implemented in the testbench.

Recommendation 2-30 —*Response checking involving data storage, computations, transformations or ordering should be implemented in the verification environment.*

Temporal expressions are not very good at expressing large data storage requirements (such as ordering checks) and complex computations (such as a cyclic redundancy checks). Data transformation usually involves complex computations and some form of data storage. Data ordering checks involve multiple dimension queuing models. These types of checks are usually better implemented in the verification environment.

Rule 2-31 — *Responses to be checked using formal analysis shall be implemented using assertions.*

Formal tools cannot reason on arbitrary procedural code. They usually understand RTL coding style and temporal expressions. Some design structures are best verified using formal tools. Their expected response must be specified using assertions.

Recommendation 2-32 —*Response checking involving signals internal to the design should be implemented using assertions.*

Some symptoms of failures are not obvious at the boundary of the design. The failure is better detected on the internal structure implementing the desired functionality. The expressiveness of the temporal expressions usually makes such *white-box* verification easier to implement using assertions. Furthermore, being functionality internal to the design, it is unlikely that the equivalent check already exists in—or could be leveraged from—procedural code in a transactor. The response may also be interesting to verify using formal tools, as described in Chapter 7.

Recommendation 2-33 —*Assumptions made or required by the implementation on input signals should be checked using assertions.*

Often, the implementation depends on some assumed or required behavior of its input signals. Any violation of these assumptions or requirements will likely cause the implementation to misbehave. Tracing the functional failure of a design, which is observed on its outputs, to invalid assumptions or behavior on its input signals is time-consuming. These assertions capture the designer's assumptions and knowledge. They can be reused whenever the design is reused and detect errors should it be reused in a context where an assumption or requirement no longer hold. The assumptions and requirements may also be required to successfully verify the implementation using formal tools, as described in Chapter 7.

Recommendation 2-34 —*Implementation-based assertions should be specified in-line with the design by implementation engineers.*

Assertions that are implied or assumed by a particular implementation of the design are specific to that implementation. Different implementations may have different implications or assumptions. Therefore, they should be embedded with the RTL code. They can be captured only by the designer as they require specific knowledge about the implementation that only the designer has.

Rule 2-35 — *Assertions shall be based on a requirement of the design or the implementation.*

Assertions must be used to verify the intent of the design, not the language used to implement it. They must be considered similar to comments and not simply capture the obvious behavior of the language, as illustrated in the following example:

Example 2-12. Trivial and Obvious Assertion

```
always @ (posedge clk)
   i <= i + 1;
...
a: assert property (
   @(posedge clk) i == $past(i) + 1
);
```

Accuracy

The simplest comparison function compares the observed output of the design with the predicted output on a cycle-by-cycle basis. But this approach requires that the response be accurately predicted down to the cycle level, a complex task. If the design specification does not specify a particular end-to-end latency, why verify at a more accurate level of precision?

The layered verification environment (see section titled "Testbench Architecture" on page 104) allows the separation of verifying the timing from the content. The verification of the content of the design output can easily be performed with complete accuracy: Either the content of the output matches the expected content or it does not. The verification of the timing of the design output can easily sustain irrelevant variations. It may occur at different times, but as long as the output eventually comes within acceptable time boundaries, no error is reported.

The timing of physical interfaces can also be verified separately from the data being transported. Transactors can verify that the relative placement of signal transitions fall within acceptable bounds, as specified by the protocol. But they do not verify that these transitions occur at specific points in absolute time.

Ordering and sequencing are other aspects of accuracy. In some classes of designs, it may be difficult to predict the exact order in which the output transactions will be observed. Similarly, it may be difficult to determine in advance which particular transactions will be dropped to maintain some higher priority functions in the design. Rather than trying to predict the exact sequence of the output, it may be sufficient to predict the relative order of independent streams of transactions or simply assume that any transaction not observed on the output was dropped. Of course, any assumption

that could mask a functional defect should be independently confirmed through other means during the verification process.

Rule 2-36 — *Response checking shall not be more accurate than necessary.*

If it is not specified, don't check for it. Suggestion 2-41 and Suggestion 2-42 describe types of behavior that may be checked with varying degrees of accuracy.

Recommendation 2-37 —*Response checking should be transaction accurate.*

The response should be verified based on the correctness of the transaction data. The timing of transactions should only be verified with respect to the occurrence of other transactions i.e., sequencing, ordering and maximum latency.

Recommendation 2-38 —*Only interfaces should be checked for timing accuracy.*

Transactors monitoring an interface should check that the timing of the signals on that interface is internally consistent and timing accurate. The relative position of signal transitions should fall within acceptable bounds but not verified against an absolute time reference.

Interfaces should not be checked cycle by cycle to allow for nonfunctional variations. For example, whether a read cycle introduces zero or several wait states is not functionally relevant—unless the function being verified is the performance of the interface.

Recommendation 2-39 —*The relative timing of different interfaces should not be verified.*

The relative timing of signal transitions on different interfaces should not be verified, unless some specified relationship exists between the interfaces.

Recommendation 2-40 —*Cycle-level accuracy should be checked only when the specification is stated at the cycle level.*

If the functional verification requirement includes a cycle-level check of the response or throughput of a design, then these requirements trump all previously stated recommendations in this section. If it is specified, it must be verified.

Suggestion 2-41 — *It may not be necessary to predict the exact transaction execution order.*

This suggestion is a special case of Rule 2-36. It may be sufficient to verify that some relative order is maintained. For example, check that independent streams multiplexed onto a single output stream are in order, but do not attempt to predict the exact inter-stream ordering. Another example would be out-of-order processor instructions: As long as instructions are executed in order of data dependencies, the exact execution order may not need to be predicted.

Suggestion 2-42 — *It may not be necessary to predict exactly which transaction will be dropped.*

This suggestion is a special case of Rule 2-36. In some applications—e.g., network routers, transactions can be dropped as part of normal operations of the designs. Is it important to predict which transaction will be dropped? Or that, if transactions are observed to have been dropped, that the minimum number of transactions were dropped and for the right reasons, regardless of which ones were dropped? For example, would it be important to predict which packets were dropped to meet quality-of-service requirements? Or would it be sufficient to check that those packets that were dropped belong to the lowest quality-of-service class?

Instead of predicting which transaction will be dropped, it may be sufficient to identify that transactions were dropped and that it occurred if and only if a valid condition was present. Assertions can be used to detect the occurrence and duration of drop conditions, isolating the verification environment from implementation details.

Scoreboarding

A scoreboard is used to dynamically predict the response of the design. As illustrated in Figure 2-2, the stimulus applied to the design is concurrently provided to a transfer function. The transfer function performs all transformation operations on the stimulus to produce the form of the final response then inserts it in a data structure. Observed response from the stimulus is forwarded to the compare function to verify that it is an expected response.

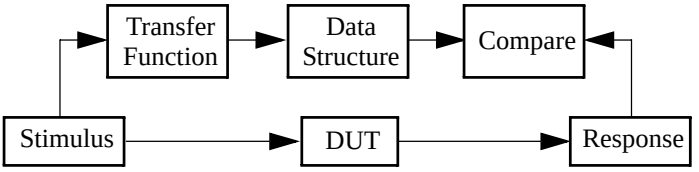

Figure 2-2. Scoreboarding

The transfer function is a transaction-level reference model that usually operates in zero time. It may also be implemented using a reference or golden model. The data

structure stores the expected response until it can be compared against the observed output. The compare function looks up the expected response in the data structure to identify if the observed response matches expectations. The data structure and compare function handle any acceptable discrepancy between the observed response and the expected output, such as ordering or latency.

The transfer function and data structure are usually configurable to match the configuration of the DUT: Different configurations may yield different responses. Transfer functions may be implemented in C. The *Direct Programming Interface* may be used to integrate them in the SystemVerilog environment. A directed test may implement its expected response using a test-specific transfer function that models only the necessary subset of the functionality that is exercised.

The term “scoreboard” is not well-defined in the industry. It sometimes refers to the storage data structure only, sometimes it includes the transfer function as well, and sometimes it includes the comparison function. In this book, the term *scoreboard* is used to refer to the entire dynamic response-checking structure.

Scoreboarding works well for verifying the end-to-end response of a design and the integrity of the output data. It can efficiently detect errors in data computations, transformation and ordering. It can also easily identify missing or spurious data. On the other hand, it is not well suited for detecting errors whose symptoms of failures are not obvious at the granularity of a single response. For example, scoreboarding has difficulty verifying the fairness of internal resource allocations and quality-of-service arbitrations. It may also be difficult to use a scoreboard to measure overall performance of the design under verification.

Reference Model

A reference model, like a scoreboard, is used to dynamically predict the response of the design. As illustrated in Figure 2-3, the stimulus applied to the design is concurrently provided the reference model. The output of the reference model is compared against the observed response.

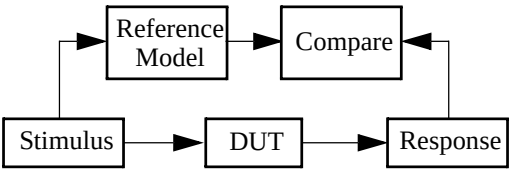

Figure 2-3. Reference Model

Reference models have the same capabilities and challenges as scoreboards. Unlike a scoreboard, the comparison function works directly from the output of the reference model. The reference model must thus produce output in the same order as the design itself. However, there is no need to produce the output with the same latency or cycle accuracy: The comparison function can handle latency and cycle discrepancies between the expected and observed response. A reference model need not be pin-accurate with the design. A reference model can be at the transaction level, with a high-level transaction interface: The comparison of the observed response with the response of the reference model is performed at the transaction level, not at the cycle-by-cycle level.

Using reference models depends heavily on their availability. If they are available, they should be used. If they are not available, scoreboarding techniques will be more efficient to implement. More often than transfer functions, reference models are implemented in C. The *Direct Programming Interface* may be used to integrate them in the SystemVerilog environment.

Offline Checking

Offline checking is used to predict the response of the design before or after the simulation of the design. As illustrated in Figure 2-4, in a pre-simulation prediction, the offline checker produces a description of the expected response, which is dynamically verified against the observed response during simulation. The compare function can dynamically compare the predicted response to the observed response or a utility can perform the comparison post-simulation. As illustrated in Figure 2-5, in a post-simulation prediction, the recorded stimulus and response of the design is compared against the predicted result by the offline response checker. In both cases, the response can be checked at varying degrees of details and accuracy, from cycle-by-cycle to transaction-level with reordering.

Using pre-simulation response prediction with dynamic response checking lets a simulation report any discrepancy while the design is in or near the state where the error occurs. It also avoids needlessly running long simulations when a fatal error occurs early in the run. Pre-simulation checking cannot generate stimulus based on the dynamic state of the design—such as the insertion of wait states—and may not exercise the design under all possible conditions.

Offline checking works well for verifying the end-to-end response of a design and the integrity of the output data based on executable system-level specifications or mathematical models. It can efficiently detect errors in data computations, transformation and ordering. Offline checking can also easily identify missing or spurious data. Post-simulation offline checking is also well suited for detecting errors

whose symptoms of failures are not obvious at the granularity of a single response. For example, it can verify the fairness of internal resource allocations and quality-of-service arbitrations by performing statistical analysis over the entire recorded response.

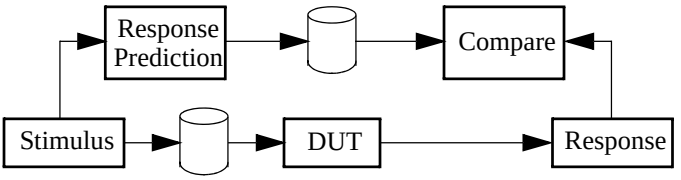

Figure 2-4. Pre-Simulation Offline Checking

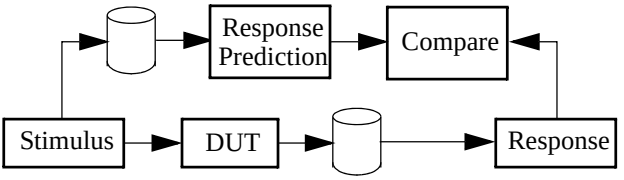

Figure 2-5. Post-Simulation Offline Checking

Offline checking need not be implemented separately from the runtime simulation environment. The invocation of external programs necessary to generate the input, predict the response and compare it with the observed response can be done by the simulator at the start or the end of the simulations. Although offline checking is usually used with a reference model, it can be used with scoreboarding techniques implemented as a separate offline program.

SUMMARY

This chapter described the necessary steps required to plan a verification project. First, the requirements that must be met by the verification projects are defined. These requirements create specifications for the stimulus, response-checking and functional coverage aspects of the verification environment.

Next, various strategies for computing or specifying the expected response of a design under verification were presented. The different strategies have different advantages and limitations when comparing the observed response against expected results. A particular strategy may have to be used to identify certain classes of failures, which may not be as easily identifiable in another approach.

Assertions are best for verifying implementation-specific and physical-level relationships; whereas testbenches are best for verifying transaction-level responses. Unlike testbenches, assertions are not limited to the primary DUT outputs to check its response. The complete response of a design will be verified using a combination of assertions and one or more verification environments.

http://www.springer.com/978-0-387-25538-5

Verification Methodology Manual for SystemVerilog
Bergeron, J.; Cerny, E.; Hunter, A.; Nightingale, A.
2006, XVII, 503 p., Hardcover
ISBN: 978-0-387-25538-5