CHAPTER

5

FUNCTIONAL COVERAGE

As the complexity of today's ASIC designs continues to increase, the challenge of verifying these designs intensifies at an even greater rate. Advances in this discipline have resulted in many sophisticated tools and approaches that aid engineers in verifying complex ASIC designs. However, the age-old question of *when is the verification job done,* remains one of the most difficult questions to answer. Consider random test generators, which are heavily used to generate simulation stimulus on-the-fly. At issue is knowing which portions of a design are repeatedly exercised and which portions are not touched at all. Or more fundamentally, exactly *what* functionality has been exercised using these techniques. Historically, answering these questions has been problematic. This has led to the development of various coverage metrics ranging from code coverage (used to identify unexercised lines of code) to functional coverage (used to identify key functionality that has not been explored) [Piziali 2004].

When specifying functional coverage, there is actually a spectrum ranging from higher-level architectural functional coverage models down to lower-level functional coverage models for RTL implementation. For example, specifying that a certain sequence related to a bus transaction must be encountered during the course of verification would be a higher form of functional coverage—and we provide examples of transaction forms of specification that can be used for functional coverage in Section 5.7 "AHB example" on page 158. Conversely, specifying that a specific FIFO counter must reach its maximum value at some point during simulation is an example of lower-level functional coverage for RTL implementation. There are many excellent commercial products available focusing on *higher levels of functional coverage.* However, the verification team building higher-level functional coverage models often neglects many interesting user-defined *functional coverage points for RTL implementation.* In fact, verification engineers generally lack the depth of knowledge of the implementation that would allow them to specify important corner case conditions (like FIFO pointer overflow or underflow). Hence, a good functional coverage methodology combines higher-level forms of functional coverage related to the specification with lower-level functional coverage for corner cases in the RTL implementation.

This chapter explores verification approaches that provide feedback about what *has* been tested and what *has not* been tested. We begin by building a knowledge base for understanding testing approaches, and from there, we lead into coverage techniques and build a case for user-defined functional coverage. After laying this foundation, we concentrate on our central topic, which is adopting an effective functional coverage methodology that provides pertinent feedback for RTL implementation. Finally, we explore available RTL implementation functional coverage technologies and close this chapter with actual functional coverage examples.

The reader might wonder, *why are you talking about functional coverage in a book focused on assertions*? In fact, the two topics are related. For example, a property language, such as PSL, can specify assertions (which monitor and report *undesirable* behavior) as well as functional coverage (which monitors and reports *desirable* behavior that must occur for the verification process to be complete). Please note that our emphasis in this book is on RTL *implementation-level* functional coverage, as opposed to system or *specification-level* functional coverage. There are numerous commercial solutions available that address specification-level functional coverage as well a literature written on this topic [Bergeron 2003]. And while all levels of functional coverage are valuable, the significance of implementation-level functional coverage specified by the designer is often overlooked. Hence, we have decided to expend a little more effort discussing techniques and methodologies around implementation-level functional coverage.

5.1 Verification approaches

black-box verification

The typical approach in design verification is through *black-box verification.* As discussed in Chapter 1, “Introduction”, with this approach, a team creates a model of a design written in a hardware description language that is instantiated in a system-level model (testbench) that drives stimulus to the design under verification (DUV) and provides a mechanism for observing and validating the output responses. Recall that this approach has limited observability and controllability. And these aspects become worse as the size of the design grows,

white-box verification

White-box testing can effectively complement traditional black-box testing by adding intimate knowledge of the internal implementation of the design. Recall from our discussion in Chapter 1 that instead of simply stimulating the external ports of a black-box using knowledge of the expected external stimulus and response, white-box testing adds a dimension of understanding that looks into how the external stimulus is used and how the external response is generated. With this insight, an engineer can also ensure that internal features operate correctly.

As discussed earlier in this book, we add assertions to monitor internal behavior. For example, an assertion can monitor a state machine to ensure that it is always one-hot. In the presence of undesirable behavior, an assertion identifies the error at its source. Additionally, assertions can validate designer assumptions and areas of concern.

Another white-box testing method adds internal or implementation level *functional coverage* to the RTL. This also extends the verification environment to more areas of the design and does not confine observation to external ports of the DUV. *Functional coverage* monitors desirable (and expected) behavior in the same way that *assertions* monitor undesirable (and unexpected) behavior. Thus white-box testing; with assertions, functional coverage, or a combination of both; provides an effective verification method, which is well documented by Kantrowitz and Noack [1996], Taylor et al. [1998], Bentley [2001], Bergeron [2003], Lacish et al. [2002], Abarbanel et al. [2000], Ziv [2002a]. Ziv [2002b], Betts et al. [2002] and Bening and Foster [2001], However, using only white-box testing misses system-level failures that the testbench, which is used in black-box testing, can detect. Consequently, an ideal methodology combines white-box and black-box testing approaches.

gray-box verification

The term *gray-box testing* is often used when we blend black-box verification methods with elements of white-box verification. By using this approach, the verification coverage can be greatly increased.

5.2 Understanding coverage

Functional verification, in general, is a process that demonstrates that the RTL implementation satisfies all requirements established in the *specification* and *design/architect* phases of the design process (see Section 1.4 "Phases of the design process" on page

14), while not exhibiting any unexpected behavior. Ultimately, the only thing that matters in functional verification is high coverage; that is, ideally we would like to explore all combinations of input values with respect to all possible sequences of internal state. Without high coverage, corner cases go unexplored, which can result in functional bugs within the silicon. Traditionally, vector-based verification techniques (such as simulation, acceleration, and emulation) have been the primary processes used for design validation, coupled with coverage techniques to expose unverified portions of the design.

To ensure that a design is correct when using traditional simulation techniques, the design must be exercised with all possible sequences of input and register states. While this is possible on smaller designs, today's large, complex designs make this method impractical. Additionally, formal techniques such as model checking can be used to exhaustively verify correct functional behavior; however, this technique does not scale to large designs. In view of the coverage limitations of available techniques, we must devise other methods that enable us to gather coverage data and answer the question: *When is the verification job done?* To address the question adequately, we must know what we *have not* verified (holes) and what functionality we *have* verified (related to the specification).

5.2.1 Controllability versus observability

Fundamental to the discussion of coverage is understanding the concepts of controllability and a observability. *Controllability* refers to the ability to stimulate a specific line of code or structure within the design. Note that, while in theory a testbench has high controllability of the input bus of its device under verification, it can have low controllability of an internal point. *Observability,* in contrast, refers to the ability to observe the effects of a specific internal, stimulated line of code or structure. Thus, a testbench generally offers limited observability, if it only observes what is on the external ports of the device or model. And all the internal signals and structures are often hidden from the testbench.

5.2.2 Types of traditional coverage metrics

A number of coverage metrics have been developed to determine the effectiveness and quality of the verification process. We summarize several coverage techniques in the following sections.

Ad-hoc metrics. Ad-hoc metrics include items such as bug rate, length of simulation after last bug found, and total simulation cycles. These metrics can provide interesting *quantitative* data, but when used for coverage metrics, they provide little *qualitative* data on how well the design has been verified or how much of the design has been left untested. The pressing question is: *If the bug rate reduces to zero while unverified features still exist, is the verification effort really complete?*

Programming code metrics. Most commercial coverage tools are based on a set of metrics originally developed for software program testing [Beizer 1990][Horgan et al. 1994]. These programming code metrics measure syntactical characteristics of the code due to execution stimuli. In other words, it is a measure of *controllability.* Examples are as follows:

- *Line coverage* measures the number of times a particular line of code was executed (or not) during a simulation.
- *Branch coverage* measures the number of times a section of code diverges into a unique flow.
- *Path coverage* measures the number of times a unique path through the code (including both statements and branches) is executed during a simulation.
- *Expression coverage* measures controllability of the individual variables, which contribute to the expression's output value.

For more on programming code coverage, see Drako and Cohen [1998] and Tasiran and Keutzer [2001].

observability and controllability related to programming code coverage

The value of code coverage is that it identifies *holes* (that is, something that has never been exercised during the course of verification). However, a shortcoming of programming code metrics is that they are limited to measuring the *controllability* aspect of our test stimuli applied to the RTL code. Activating an erroneous statement does not mean that the design bug would manifest itself at an observable point during the course of simulation.

Techniques have been proposed to measure the *observability* aspect of test stimuli by Devadas et al. [1996] and Fallah et al. [1998]. What is particularly interesting are the results presented by Fallah et al., which compare traditional line coverage and their observability coverage using both directed and random simulation. They found instances where the verification test stimuli achieved 100% line coverage, yet achieved only 77% observability coverage. Other instances achieved 90% line coverage, and achieved only 54% observability coverage.

functional correctness

Another drawback with programming code metrics is that they provide no qualitative insight into our testing for *functional correctness.* Kantrowitz and Noack [1996] propose a technique for functional coverage analysis that combines correctness checkers with coverage analysis techniques. In this chapter, we describe a similar technique that combines event monitors, assertion checkers, and coverage techniques into a methodology for validating *functional correctness* and measuring desirable events (that is, observable points of interest) during simulation.

In spite of these limitations, programming code metrics still provide a valuable, albeit crude, indication of which portions of the design have not been exercised. Keating and Bricaud [1999] recommend targeting 100% programming code coverage during block level verification. It is important to recognize, however, that achieving 100% programming code coverage does not translate into 100% observability (detection) of errors or 100% functional coverage. The cost and effort of achieving 100% programming code coverage must be weighed against the option of switching our focus to an alternative coverage metric (for example, measuring functional behavior using functional coverage specification).

State machine and arc coverage metrics . *State machine* and *arc coverage* is another measurement of controllability. These metrics address the number of visits to a unique state or arc transition as a result of the test stimuli. The value these metrics provide is in their ability to uncover unexercised arc transitions, which enables us to tune our verification strategy. Like programming code metrics, however, state machine and arc coverage metrics provide no measurement of observability (for example, an error resulting from arc transitions might not be detected), nor does it provide a measurement of the state machine's functional correctness (for example, valid sequences of state transitions).

Functional coverage metrics. User-defined functional coverage allows the designer, who possesses the greatest knowledge of the low-level design details and implementation assumptions, to specify functional coverage for points in the design that are known to be significant. The remainder of this chapter focuses on functional coverage.

5.2.3 What is functional coverage?

functional coverage versus programming code metrics

Functional coverage and programming code coverage tools offer different perspectives to coverage metrics. Code coverage tools take a blind approach to coverage by monitoring the design as a whole without specific knowledge of its operation. Conversely, since someone familiar with the design adds functional coverage, application domain knowledge is inherent.

functional coverage versus assertions

While RTL-implementation functional coverage and assertions can be implemented with the same form of specification, they focus on two different areas. To eliminate confusion between the two cases, this book refers to the detection and reporting of illegal behavior as *assertions*—and detection and reporting of expected (or desired) behavior as *functional coverage.* Essentially, assertions and functional coverage are both properties of the design. However, functional coverage provides indications of when a specific functionality of the design has been exercised. Nevertheless, both provide coverage feedback.

Definitions

Use the following definitions to guide your understanding of the terms we use in this book.

- *functional coverage* refers to the entire functional coverage methodology
- *functional coverage point* is a specific feature or event in the design that we want to monitor and include coverage information for in our functional coverage reports
- *functional coverage model* is a collection of functional coverage points that will be applied to a specific design
- *cross functional coverage* is an analysis of functional coverage over time

Grinwald et al. [1998] describe a coverage methodology that separates the coverage model definition from the coverage analysis tools. This enables the user to define unique coverage metrics for significant points within the design. They cite examples of user-defined coverage that targets the proper handling of interrupts and a branch unit pipe model of coverage. In general, user-defined functional coverage provides an excellent means for focusing and directing the verification effort on areas of specific concern.

internal monitors

For RTL implementation functional coverage, *internal monitors* are often used to capture functional coverage events. This provides an automated approach to obtaining coverage data. Since an engineer familiar with the design inserts the functional coverage monitor into the RTL, these monitors provide coverage feedback on areas of the design that the engineer feels are important.

5.2.4 Building functional coverage models

questions you should ask yourself related to functional coverage

Bergeron [2003] provides an interesting view of functional coverage. In his book, he describes the questions we must answer to build an effective functional coverage model.

- *What* should be covered? Questions such as *did the FIFO become full?* And *did I see a READ instruction?* are examples of what to cover.

- *Where* is the best place to monitor for the covered event? For example, when we have a READ instruction in a FIFO, we must ask ourselves if we should look for the READ instruction entering the FIFO or leaving the FIFO. Since there is a chance the instruction can enter the FIFO but never leave due to a reset condition, in some cases, it makes sense to look for the instruction leaving the FIFO. Monitoring the instruction entering the FIFO would give a false indication that the READ instruction was actually used.

- *When* should I look for the condition that I am covering? Is it on every clock edge or only on clock edges when the instruction valid signal is active?

- *Why* should we cover the event? Is the event interesting enough to monitor it, log it, include it in reports, and analyze it? Bergeron's example in this case is a 32-bit address decoder. Why would we want to monitor every 32-bit address when the decoder only analyzes the upper 4 bits to decode the address into 16 pages?

Asking these questions will aid you in understanding an effective process for applying functional coverage to your design.

shared ownership of functional coverage

The task of building functional coverage models is owned by both the verification and design engineers. The verification engineer specifies the high level aspects of the functional coverage model with a black-box view of the design. This occurs during the specification phase (refer to Section 1.4 "Phases of the design process" on page 14 for a description of the design process phases) and uses the specification as the primary input for specifying this portion of the functional coverage model. The verification engineer also describes coverage points down into the design at major interfaces between major blocks within the design during the architect/design phase. This approach takes a big picture view of the system. *(Are all processor transactions covered on the processor bus?)* Also during the architect/design phase, the design engineer specifies coverage points on the higher level design decisions, particularly at interfaces between the major blocks of the design. (*What is the protocol of the interface between these two major blocks?*) Finally, during the implementation phase, the design engineer takes a detailed picture view of the design. *(How is my FSM implemented?)* Additionally, the verification engineer adds coverage points to the testbench as it is implemented. Using this shared approach to functional coverage, a complete functional coverage model is created.

5.2.5 Sources of functional coverage

With this foundation understanding of functional coverage, we are ready to move forward and gain an understanding of the importance of how to build a functional coverage model. This section explores multiple sources that form the functional coverage model.

specification knowledge

Specification. The design specification is the initial source for specifying the functional coverage model. By building the functional coverage model utilizing the design specification, there is a direct mapping from the feedback generated by the functional coverage to the design specification. If this process of equating the functional coverage model to the design specification is done thoroughly, the test plan description becomes as simple as *reach 100% functional coverage.* This approach also follows the reconvergent model described by Bergeron [2003], because the functional coverage model written by the verification engineer is driven by the specification, not the implementation, which is driven by the design engineer.

verification knowledge

Testplan. One way to collect functional coverage is by creating and executing a test plan. Verification engineers create a test plan that details a list of required testing that must be accomplished to validate the design specification. Traditionally, the test plan is derived from the design specification. We then execute test plan items in a testing environment and verify the desired result. This verification step can be a manual or automated check, depending on the sophistication of the testing environment.

Writing specific directed tests to achieve functional coverage specified in the test plan is labor-intensive. Random test generation tools will often automatically cover many of the test plan item cases during random simulation. Functional coverage helps identify test plan items that are covered during random simulation.

design knowledge

Implementation. Specific design implementation knowledge is also captured by the functional coverage model. The engineer creating the design identifies interesting points within the design.

Functional coverage embedded in the RTL design produces an automated method of collecting functional coverage. Since the coverage points are associated with the model description, they are evaluated whenever the model is exercised, whether it be in block- or chip-level simulation environments. Functional coverage points can be as simple as *all input cases of a decoder have been seen* or as complicated as a multi-cycle handshake requirement of a bus specification.

Implementation-level functional coverage can be linked with test plans to more easily describe the individual test plan items and to provide a feedback path for when an individual test plan item has been completed. When functional coverage is combined with the detailed test plan, it answers the question: *Am I done verifying this design?*

assertion knowledge

Assertions. Assertions provide a form of functional coverage. While their objective is to detect and report illegal behavior, they also provide functional coverage when the input stimulus of the assertion has been activated but the condition being checked does not fail. For example, lets examine the following implication assertion:

```
assert always (A -> B);
```

This states that whenever A evaluates *true,* B must evaluate *true.* Notice that if A never occurs during the course of verification, we might get a false sense of security that the assertion is valid. Thus, an assertion is not very useful if the stimulus it is checking is never activated. Hence, triggering events associated with assertions are excellent functional coverage points. For example, the following PSL specification covers the triggering event in the assertion described above:

```
cover {A};
```

5.3 Does functional coverage really work?

The previous section explained what functional coverage is and how it can be used. This section provides concrete data to support the effectiveness of including functional coverage as part of your verification strategy.

5.3.1 Benefits of functional coverage

A summary of the major benefits of functional coverage is listed below. Many of these items are discussed in further detail elsewhere in this chapter.

- Functional coverage answers the question: *Am I done verifying the design?*
- Functional coverage provides feedback on areas of a design that the user designates as significant. This feedback gives an indication of how effective the project's current set of tests are in exercising the features of a design.
- Through analysis of the functional coverage results, verification teams optimize the tests that comprise the regression suites by removing tests that do not provide additional coverage. This optimization reduces the time and computer resources required for the project's regressions.
- Functional coverage provides specific feedback that can direct future verification efforts. For example, if the functional coverage shows certain features are not being tested, the design team can modify test generation algorithms to target those areas. Refer to Section 5.4.4, "Coverage analysis" and Section 5.4.6, "Coverage-driven test generation" for further details.
- Functional coverage provides the feedback needed to help determine the effectiveness of random environments and to help steer the configurations of these environments.
- Testplans can be written directly using property specification languages that generate concise and unambiguous functional coverage. Refer to Section 5.3.2, "Success stories" for more details.
- Functional coverage increases observability. Refer to Section 5.2.2, "Types of traditional coverage metrics" for more details on controllability and observability.
- A functional coverage methodology provides an automated means for collecting and reporting functional coverage.
- The impact of changes in the testing strategy can be seen through functional coverage over time.

5.3.2 Success stories

The following are examples of verification success achieved through functional coverage methodologies.

- Grinwald et al. [1998] describes how functional coverage allowed his team to trim the number of tests within their regression suite without a reduction in functional coverage. As a result, they significantly reduced the time and computer resources required to execute the regression suite.

- Ziv [2002b] describes two verification efforts that utilized functional coverage.
  - First, a PowerPC processor execution unit coverage model contained over 4400 functional coverage points. After 25,000 tests, about 64 percent of the functional coverage points were hit and additional tests were not providing a substantial increase in coverage. By analyzing the functional coverage, it was determined that two major areas were not well covered. By using this observation, adjustments to the test generators provided an immediate and substantial increase in coverage. Continued testing provided close to complete coverage of all functional coverage points.
  - Second, a branch unit for an S/390 processor coverage model contained about 1400 functional coverage points. By going through the process of adding the coverage points, the team was able to gain a better understanding of the design, even before beginning to collect coverage. In addition, the functional coverage provided information that identified several performance bugs.
- From the authors' own experience on the SX1000, a super scalar processor chipset project at Hewlett-Packard, the coverage model was comprised of over 14,000 functional coverage points. By analyzing functional coverage for each verification environment, especially the random environments, the team found that several key test generation features they believed were enabled, were actually disabled. Additionally, functional coverage across all verification environments was merged and tracked across model releases. Analysis of the functional coverage results helped identify specific features that were not exercised in any verification environment. As a result, additional targeted tests were written to cover these features. Before tape out, 100% functional coverage was achieved. About 90% was easily attained through normal verification efforts. The last 10% required a substantial effort to reach.
- Bentley [2001] described the use of functional coverage on the Intel Pentium 4 microprocessor project. His team used almost 2.5 million unit-level coverage points combined with over 250,000 inter-unit coverage points to describe their coverage model. By the end of the project, the team was successful in covering 90% of the unit-level points and 75% of the inter-unit points, ultimately delivering high quality silicon.

5.3.3 Why is functional coverage not used

As with any technology, incorrect use produces useless results. And generally, people who encounter useless results avoid the technology. Functional coverage is no different. This section discusses some of the primary reasons given for avoiding functional coverage as part of an overall verification methodology. Following each point is a discussion of how a complete functional coverage methodology can eliminate the concern that is presented.

using functional coverage the "right" way is more an art than a science. [Ziv 2002b]

Too difficult. Some engineers consider the additional effort required to specify the functional coverage points too costly. It is true that adopting a new methodology involves a learning curve. And initially, the full benefits of the methodology are not necessarily realized. However, it is our experience that when a sound functional coverage methodology is adopted by the entire team, the rewards are far greater than the cost of implementation.

provides too much data

Too much data. Others suggest that functional coverage produces too much information to analyze. Engineers might confess they ignore the functional coverage reports because the data is not meaningful. If you are experiencing this scenario, make adjustments to your functional coverage methodology. Identifying functional coverage points is an important aspect of the process. If care is not taken to identify interesting coverage points, the functional coverage produced will be blurred by data that provides no significant feedback. However, by following a solid methodology that directs the team through the process of creating the functional coverage model, you can easily avoid this problem. Additionally, organizing the data in meaningful ways, as described later in this chapter, also eliminates problems with analyzing too much data.

provides too few results

Limited results. Some will argue that functional coverage provides a limited quantity of results. It is true that the quality and amount of results generated by functional coverage is directly related to the amount of effort spent identifying functional coverage points. If a limited effort is put forth to build a functional coverage model, the results will be limited. However, an effective functional coverage methodology (as discussed in Section 5.4) ensures that this does not happen.

5.4 Functional coverage methodology

Just as with assertions, functional coverage is most effective when we institute a good functional coverage methodology. The following sections describe the basic practices that form a sound methodology. We take you from creating a process through visualizing the organization and devising the analysis involved in implementing your methodology.

5.4.1 Steps to functional coverage

how do I start?

One of the first obstacles to overcome is answering the question, *How do I start?* This section explores the basic steps involved in getting your functional coverage running.

choose the form of specification

The first step is to identify a form of functional coverage specification that you will use as the basis for your project (for example, PSL or SystemVerilog cover properties—or some other appropriate tool-specific form). Refer to Section 5.5, "Specifying functional coverage" for details.

start with the functional specification

With the functional coverage specification form in place, the next step is to begin the process of identifying and creating functional coverage points. The easiest place to start is with the functional specification for your system. Also refer to Section 5.4.2, "Correct coverage density" as a guide for further functional coverage points.

convert points to monitors

Once you have identified functional coverage points, convert them into monitors using the selected functional coverage form. The specifics will vary depending on the chosen form of specification. In some cases, the testbench tool or simulator is able to parse the functional coverage specification directly (for instance, PSL or SystemVerilog coverage constructs). In other cases, such as with a custom coverage technology or when working with tools that do not recognize languages such as PSL, additional work is required to translate the functional coverage specification into simulation monitors (see [Abarbanel et al. 2000]).

collect the coverage data

A defined process must be in place to collect and merge the functional coverage results. Functional coverage is collected as soon as basic model stability is achieved. Even though only basic features are available at this point, functional coverage still provides effective feedback.

On super scalar processor chipset sx1000 project at Hewlett-Packard, an updated version of the design was released approximately once a week. Functional coverage was collected and posted on the project web site for each model release. It was the system test coordinator's responsibility to collect the results from all the different verification environments, merge them, and analyze how the results changed from release to release.

analyze the functional coverage

Once you collect functional coverage, you must analyze it. This analysis includes looking for improved coverage and noticing functional coverage points that were reached in previous releases but are no longer being hit. This data is used to make appropriate adjustments to the test generators and to write additional tests to increase the overall functional coverage. The practice of using the functional coverage to adjust the test generation can be a manual or automatic process. The concept of a test generator that automatically reacts to functional coverage is explored in Section 5.4.6, "Coverage-driven test generation".

5.4.2 Correct coverage density

To obtain the best information from functional coverage, the methodology must foster uniformity by including directions on where to add functional coverage points in the design. Without consistent placement, functional coverage has less impact. Assume that your analysis indicates that 90% of the functional coverage has been exercised. This data is not terribly meaningful if only 10% of the design is covered by functional coverage points. A well-defined guide to adding functional coverage points provides a measure of how much of the design is covered by functional coverage points.

Additionally, functional coverage is driven by an assessment of the features you want to validate. Achieving 100% functional coverage is an indication that the testing you want to do is complete. Ensuring the functional coverage model is a complete representation of what you wish to validate across the entire system is an important part of a successful functional coverage methodology.

This section explores some of the more common locations to apply functional coverage.

concise forms of specification

Table coverage. Teams often use tables to concisely describe the requirements for sections of a design. An error table may document the different detectable errors within a block and the different paths from which each error can be produced. Mapping functional coverage points to each entry in the table provides excellent feedback on how well the features in the table are being exercised.

corner cases

Interesting corner cases. Cover any interesting corner case that may be hit in a random environment. These cases are generally specific to each portion of the design. An engineer familiar with the design has knowledge of these corner cases. A corner case is any portion of the design that includes a complex algorithm or is an area of concern for the designer. One example would be to check for a specific sequence of requests to an arbiter that concerns the designer.

full and empty conditions

Queue levels. Add functional coverage points to capture how full a queue is (such as: full, half_full, full_minus_one, or full_minus_two). This provides better feedback on how much of a queue's depth is being utilized through simulations. Although it is difficult (and sometimes impossible) to actually get to a full state, feedback from functional coverage indicates if tweaks to test generation parameters or tests are moving the test in the right direction. Any resource that can become full falls into this category.

bypass and stall

Bypass and Stall cases. Pipeline logic associated with registers often has a bypass or stall mode. Functional coverage monitors when these modes are used and these alternate paths are exercised. For instance, after a stall occurs, the next stage may need to avoid taking new data. In this case, functional coverage is used to capture the case when a stall occurred and the following data was available but held off.

Note the following caution: When the logic is idle, the bypass control could default to a specific case, which could cause the functional coverage condition to be active for a high percentage of the simulation if not properly qualified. This gives extraneous information. Always consider *when* the functional coverage point is of interest.

hw detectable errors

Hardware error detection cases. Often a design will include logic to record that certain hardware errors were detected. Place functional coverage points on hardware error log fields to capture an indication of how the error was reached. Since a hardware error can be detected in a variety of ways, ensure that all the error detection paths are identified in the coverage model. For instance, a particular detected error may have a feature that disables hardware error logging. Create functional coverage points to capture the cases in which the hardware error is enabled, detected, and logged as well as the cases in which the hardware error is disabled, detected, but not logged.

debug logic

Post-silicon debug logic. When a design includes specific logic to aid in post-silicon debug, this logic requires verification as well. Functional coverage can give feedback on the debug logic to ensure it has been tested.

internal sequences

Internal activity cases. Functional coverage can track the internal activities of the design in the same way it is used to track activities on external interfaces. Example of internal activities include scenarios such as *a packet was received with type=x* or *a recall response action was launched.* In these cases, use a different functional coverage point for each packet type or action. Go one step further and insert a different functional coverage point for each circumstance that results in a given action being launched.

traffic patterns

Traffic patterns. In addition to monitoring specific packets, it is often interesting to watch interesting traffic patterns. A traffic pattern is a sequence of events or transactions that are expected to occur within a system.

states

FSM states and state transitions. Most commercial coverage tools provide FSM state and arc (that is, transition) coverage. However, for multiple, interacting state machines spanning multiple hierarchies, these tools are not effective. High-level state machine activity is a great place to add functional coverage for important states or transitions.

use the spec

Specification-driven cases. Use the functional specification to identify useful functional coverage points. Include a mapping of these points back to the functional specification. For instance, include the specification heading number in the functional coverage point name. Shimizu and Dill [2002] extend this concept even further by automatically deriving testbench stimulus, checking properties, and functional coverage from the interface protocol specification.

what do I need to test

Other cases. Thinking about questions like: *What do I need to test in this section of the design*? Or *would you want to know if X occurred?* This is a good ways to come up with the functional coverage that should be added.

5.4.3 Incorrect coverage density

Use this section with great care to avoid over generalizing these concepts and misusing the guidance we offer. Although it is generally a good practice to use functional coverage in all areas of the design, there are some situations in which functional coverage does not add value to the coverage results. On the contrary, they produce a performance penalty and provide extraneous data that hinders coverage analysis.

too much

Too much data. As mentioned previously, one of the problems associated with any methodology is that you can generate more data than you can analyze in a reasonable amount of time. The goal is to obtain meaningful feedback from the functional coverage. If a functional coverage point does not provide meaningful feedback, it should not be a part of the coverage model. When you ensure that all the functional coverage provides meaningful information, you eliminate one of the reasons engineers avoid functional coverage.

always active

Functional coverage points that fire every cycle. In general, this type of functional coverage does not add enough information to the functional coverage to warrant the increased log file sizes. An alternative to eliminating this point is to constrain it to only activate at significant times.

too basic

Basic functionality. If the coverage point is too basic, the feedback is not useful, making the functional coverage reports so large that they become unmanageable (and eventually discarded). Associate functional coverage points with *interesting* cases that provide *real* feedback on how well the design is being exercised. When considering a coverage point, ask yourself what you will gain by knowing that the point was reached.

code coverage

Code coverage duplication. Functional coverage is superfluous if it provides feedback that is easily monitored by code coverage tools. If you need to trim back the number of functional coverage points, consider this area.

too many

Too many functional coverage points to add. If the set of functional coverage is too large to enumerate, use an alternative method to track coverage. An engineer should not define so many functional coverage points that it is impossible to write *and* track the functional coverage. This step prevents the coverage reports from becoming unwieldy. In certain cases, it may be possible to create a subset of the functional coverage points and use cross functional coverage to analyze the temporal relationship between those events.

5.4.4 Coverage analysis

An essential part of an effective functional coverage methodology is defining the process for analyzing and acting on data. This analysis occurs periodically throughout the project.

Coverage data organization

Functional coverage models produce a large volume of data. It is imperative that you have methods to sort and organize this data. This section explores methods that aid organization.

Taylor et al. [1998] describe one way to organize coverage data so that it can be used to effectively direct verification efforts. They divided the coverage analysis into the following four categories.

State transition. State transition analysis concentrates on complex state machines to ensure that all possible states and state transitions are exercised.

Sequence. Sequence analysis focuses on sequences of functional coverage over time. For instance, Taylor et al. [1998] used sequence analysis to ensure that every type of command leaving the CPU was followed by every type of command entering the CPU. This is equivalent to cross functional coverage.

Occurrence. The value of some functional coverage is that it was active at least once. The fact that it was activated many times gives no more coverage data than knowing that it was activated once. This type of functional coverage was analyzed with occurrence analysis. For instance, functional coverage can be used to ensure that all bits of an adder block have produced a carry-out.

Case. Case analysis deals with collecting statistics on the entire set of simulations. Statistics such as system configuration, instruction types issued, and bypass modes enabled are just a few examples of items from this category.

functional coverage groups

Another method for managing functional coverage better is *functional coverage grouping*. This provides a sorting parameter that allows functional coverage to be classified into functional categories, which is useful during data analysis. A functional coverage instance is associated with a unique coverage group. Some suggested coverage groups are listed below. However, groups are customized for each project's specific needs and include any number of categories that effectively organize a design's functional coverage.

Normal. Most designs include a *Normal* functional coverage group that consists of functional coverage points that don't have special functional characteristics. This is the default group for functional coverage points.

Not Reachable. A *Not Reachable* group includes functional coverage points that cannot be hit. While this group may seem superfluous, not reachable functional coverage points can appear in a design in cases where there are multiple instantiation of subblocks. For instance, a 1 to 4 decoder could have a functional coverage point associated with each decoded state. However, a particular instantiation of the decoder may only use half of the states. As a result, half of the functional coverage points associated with this instantiation will never be exercised due to the design implementation. If your functional coverage methodology does not allow for identifying these cases, the functional coverage reports will be difficult to interpret.

Error. Almost all designs will include logic for detecting, and in many cases correcting, runtime errors (for example, ECC logic and parity). An error group is useful because some tests in the regression suite will not target error cases. Since they are always identified as uncovered, when analyzing coverage of this subset of tests, the coverage data is skewed if the error monitors are included in the results. By including an error functional coverage group, you separate the error coverage data from other functional coverage or eliminate them from coverage reports for tests that don’t target them.

Debug. Special logic is often added to debug features of a design, but (like error logic) they are not always targeted in normal testing. As with the *error* group, a *debug* group allows you to ignore any functional coverage associated with this logic when necessary.

Not Supported. Often during the design cycle, teams make trade-offs that remove a feature from the design. Any work completed on these portions of the design, including functional coverage points, is often left in the design for a future revision. However, these portions of the design will not be targeted in the verification efforts and any related functional coverage should be ignored. By placing these points in a *not supported* group*,* they are automatically excluded from functional coverage reports.

associating groups with functional coverage points

There are several methods to associate functional coverage points with a coverage group. A simple method is to prefix the name of each coverage point with the group name. By sorting the functional coverage points by name, the points are segregated into the individual groups. Another method, which requires additional infrastructure, is to use a mapping method. For instance, a separate file can list all the functional coverage points and their associated group. Example 5-1 shows a possible format for this file, which is parsed prior to generating reports to provide the group information.

Example 5-1 Associating functional coverage points with groups

```
FIFO_FULL                   QUEUE
FIFO_HALF_FULL              QUEUE
BAD_TRANSACTION             ERROR
```

Tracking functional coverage

Track functional coverage throughout the design and verification process, starting when the model is at an initial level of stability. When used early in the process, functional coverage ensures that design and tool features are accurately enabled. Later in the process, functional coverage ensures that all portions of the design are being exercised. You should track functional coverage along with each release of the model.

coverage across environments

It is important to track functional coverage across all the verification environments. Teams often use multiple verification environments to target specific portions of a design. By analyzing functional coverage data across all verification efforts, you avoid duplication. This also identifies areas of the design that have yet to be exercised by any environment.

tracking coverage over time

By tracking functional coverage across release models and over time, it is easy to watch progress in functional coverage. In addition, it is an accurate way to monitor the settings of the verification tools. For example, *feature X* is enabled, but one of the model releases included a bug that required the team to temporarily disable testing *of feature X.* After fixing the bug, it is important to re-enable the feature in the verification tools. However, this step is easily overlooked. By monitoring functional coverage, the team is alerted to the oversight, because events that occurred in previous model releases do not occur in the current model releases.

Actions to take

direct future verification efforts

At some point, the collected coverage will show a significant slow down in increasing coverage. This should prompt the team to make a change in test generators that results in an increase in the functional coverage. Also, use functional coverage to help direct the random verification environments, which are effective with corner cases of the design.

5.4.5 Coverage best practices

This section describes some practices that enhance a functional coverage methodology.

when to add functional coverage

Add during design creation. As described in Chapter 2, "Assertion Methodology" on page 21, for assertions, our experience shows that the best time to specify functional coverage is prior to and during RTL implementation. During this phase, the designer is most intimately familiar with the inner workings of the design and is aware of features that are the most interesting in coverage analysis. In addition, in the process of considering where to add functional coverage, designers often find design flaws (that is, before running verification tools). This methodology is also described by Foster et al. [2002] and Foster and Coelho [2001]

failed simulations

Exclude failed simulations. Since a failed simulation can be the result of the design stepping into illegal states, you can falsely reach functional coverage points. For this reason, do not collect functional coverage from simulations that failed.

names **Assign a name to every functional coverage point instantiation.** Assigning a specific name makes it easier to track and sort the functional coverage. It is also useful to define a consistent naming convention. The following are two commonly used sorting conventions.

- Use the sub-block name as a prefix
- Use functional group names as prefixes or suffixes.

maximum reporting **Control the maximum number of times a functional coverage point can fire.** As discussed with assertions, it is also useful to provide a mechanism to cap the maximum number of times a functional coverage point will fire. This is especially important if the coverage data is used only to see if certain portions of a design have been exercised (occurrence coverage). This practice reduces the size of functional coverage logs, which saves disk space and improves simulation performance. Note: Since all functional coverage data is not logged, if you use this feature, results from cross functional coverage are incomplete.

when did I reach it **Watch for specific functional coverage points.** In volume simulations, capturing functional coverage is an automated process. However, some functional coverage points could be very difficult to hit. For these functional coverage points, it is important to capture the simulation parameters, such as test name or random seeds, that were used in the simulation that hit them. To address this need, implement a method to query the functional coverage log from each simulation looking for any *hard to reach* functional coverage points. Another option is to use the severity level of functional coverage to force the simulation to have the appearance of a failure when the points are reached. If they fire, the process archives the simulation parameters along with the functional coverage. With this information, you can recreate and analyze the simulation in further detail.

common logfile format **Use a common logfile format.** An important part of functional coverage is to define a common log file format that you will use across the entire project. With a common format, whether it is a database or a text file, you can merge and compare the coverage data in many ways, including between model releases and across verification environments.

what to log **Record the right information.** The important pieces of information to record include the following:

- Functional coverage point name
- Indication of instantiation (this could be the hardware path or other identifying information)

- Time (time is only important if a cross functional coverage analysis is performed)
- Coverage group for sorting the data (if needed).

multiple instances

A typical design has multiple instantiations of a module, so it follows that you have multiple instances of a functional coverage point with a single name. To effectively identify each instance, the functional coverage technology must log the hardware path to each functional coverage point.

merge data

Merge data across simulations. For volume progression simulations, a method is required to merge functional coverage from all simulations. This provides access to all the data that would be needed for any type of functional coverage analysis, including cross functional coverage. However, saving all this data requires a large amount of storage. Alternatively, if you do not intend to use cross functional coverage, configure the reporting mechanism such that it does not log the time information.

A variety of statistics can be generated, such as, average number of functional coverage point firings per simulation as well as maximum and minimum number of functional coverage point firings across all simulations. Examples of functional coverage reports are described by Kantrowitz and Noack [1996] and in Example 5-2.

Example 5-2 Functional coverage report

```
Group       Id                Total   TestsHit   Avg    Max    Min
-------------------------------------------------------------------------------
Normal;BLK  QMU_XX4_FROM_1    119528  1787       67     270    0
Normal;BLK  QMU_XX4_FROM_2    162409  1946       83     463    0
Normal;BLK  QMU_XX4_BYPASS    0       0          0      0      0
<...>
NoReach;BLK QMU_PR4_FROM_XIN  0       0          0      0      0
<...>

There were 462 out of 474 total Normal points hit in BLK (97.5%)
There were 0 out of 25 total NoReach points hit in BLK (0.0%)
```

regression testing

Regression and progression testing. The early stages of the verification process focus on *regression testing.* Regression testing involves running a set of known, working tests (called a regression suite) on each successive model release to ensure that new features do not create new bugs in previously verified portions of the design. In simpler terms, all tests that had been working continue to work. Regression testing is backward-looking. It does not concentrate on new features. It concentrates on previously verified features. A good regression suite evolves over the life of the project. As more tests are created and validated, they are added to the regression suite.

Functional coverage is a critical part of regression testing. By capturing and analyzing functional coverage from regression testing, you assure your team that the functional coverage is, at the very least, staying constant. In other words, the regression suite should continue to produce the same level of functional coverage for each model release.

progression testing

Progression testing is forward-looking. It explores new regions of the design and tries to exercise existing areas of the design in different ways than in the past. The purpose of *progression testing* is to increase the total number of cycles simulated in hopes of finding hidden corner case bugs. The exact role of progression testing is somewhat dependent on where you are in the verification life cycle.

Progression testing is used early in the verification life cycle, especially in the random environments, to gain additional coverage and to exercise hard-to-reach corner cases. Functional coverage plays a valuable role by reporting how much of the design is being exercised. In this phase, capture and analyze functional coverage to ensure that the effort you expend is really increasing coverage.

Later in the verification life cycle, when the error rate is minimal and functional coverage has reached targeted levels, the emphasis of progression testing shifts to focus on increasing the total number of simulation cycles in hopes of finding those remaining corner case bugs. In this phase of progression testing, functional coverage data has a reduced value and the simulation performance is the most important factor. In this progression testing phase, it is acceptable to turn off functional coverage to improve simulation performance.

correctness

Functional coverage correctness. If functional coverage is to provide the intended coverage feedback, teams must correctly specify the coverage points. Several possibilities exist for ensuring the correctness of the functional coverage points implementations.

- **Directed tests.** One method to ensure that functional coverage is specified correctly is to write a simple directed test that stimulates the model in such a way that the functional coverage point in question fires. Unfortunately, this method is time consuming. Thus, you cannot use this method exclusively if your design has a large number of functional coverage points.

- **Linting**. Use lint checkers not only for the synthesizable portion of a design, but also for the functional coverage points. Simple errors in the coverage specification are easily caught using this method.

- **Peer reviews**. Peer reviews are just as effective for finding problems with functional coverage specification as they are for finding errors in the actual design implementation. Review the coverage specification as part of normally scheduled design reviews or at reviews that specifically focus on functional coverage.

- **Validation checks using simulations**. An effective way to verify the correctness of functional coverage specification is to compare the coverage data obtained from simulations with the stimulus for that simulation. Using deductive reasoning skills, you can determine whether the scenarios required to generate the functional coverage were, in fact, present. If they were not, analyze the coverage points for errors. Likewise, investigate coverage points that do not fire even though it is reasonable to expect that they would.

5.4.6 Coverage-driven test generation

reactive testbench

Random test generators are giving verification engineers a new approach for achieving their objectives. As the use of random generators increases, the next step to increased simulation productivity is to combine the observability provided by functional coverage with the controllability provided by random test generators. By including feedback from functional coverage into the random test generators, the focus of the generators can be shifted based on functional coverage levels or as the result of reaching a specific functional coverage point. For example, a random generator may include a method to inject a full bandwidth stream of transactions in hopes of filling queues. By providing feedback that the queues of interest have been filled, the test generator can immediately switch to another focus, instead of involving the entire simulator in keeping the queue full. This process of feeding back functional coverage to test generators is often called a *reactive testbench.*

Another tool available to the verification engineer is Hardware Verification Languages (HVLs). HVLs provide a language specifically designed for testbench generation. In addition, most modern HVLs provide mechanisms for creating reactive testbenches.

Reactive testbench generation requires application-specific knowledge of your design. Methods for easily generating reactive testbenches have recently been investigate, for example [Adir et al., 2002a], [Adir et al., 2002b], [Benjamin et al., 1999], [Geist et al.., 1996], [Ziv et al., 2001]. In some cases [Ur and Yadin, 1999], a separate description language was developed to describe a parallel model of the system by which coverage feedback was provided. They found that the cost associated with developing a reactive testbench was less than the effort required to manually provide the feedback. With emerging hardware verification languages (HVL), this feedback path can be captured along with the testbench.

5.5 Specifying functional coverage

There are many excellent tool-dependant and hardware verification language (HVL) solutions available for measuring functional coverage (for example, *e* [e Language Reference Manual] or *OpenVera* [OpenVera Language Reference Manual 2003])—and we recommend you take advantage of these offerings whenever possible to reduce project resources in developing a functional coverage model. As previously stated, often these tool-dependant solutions focus on higher-level forms of functional coverage, such as transactions. Hence, methodologies and convenience around RTL implementation functional coverage is rarely addressed. When creating your own *RTL implementation* functional coverage methodology, we recommend a tool-independent solution that can be delivered with IP or re-usable blocks. This allows you to leverage RTL functional coverage across all verification environments and tools.

specify once

Functional coverage points should only have to be *specified once* in the design and then supported by all verification environments and tools. Verilog-based assertion solutions such as OVL and SystemVerilog generally enable this by default, as the functional coverage points are evaluated by the Verilog simulator. For property languages such as PSL, ensure that the tool suite your project uses supports the language.

5.5.1 Embedded in the RTL

Bening and Foster [2001] describe a simple method to implement functional coverage by embedding the functional coverage points directly into the Verilog RTL. Example 5-1 illustrates functional coverage that detects when a queue reaches its full mark.

Example 5-3 RTL functional coverage implementation

```
`ifdef COVERAGE_ON
// look for a queue-full condition
always @ (posedge clk) begin
   if (reset_n == 1'bl & & q _full) begin
        $display("COV_Q _FULL @  %0d:%t:%m",  $time);
   end // end if (...q_full...)
end
`endif
```

While this approach provides an easy method to add functional coverage, you must type a substantial amount of text for each functional coverage point that you add. This may be a deterrent for some designers. Also, since the output is through a system function, there is limited control of the messaging. Ensure that you use a common message format for all instantiations using this method. Since output is to standard output in this example, a method is required to extract the functional coverage messages from messages produced by other sources. In this example, a prefix of "COV_" identifies functional coverage messages.

5.5.2 Functional coverage libraries

In Chapter 3, "Specifying RTL Properties" on page 61, we discussed the specification for a library of assertions through the OVL. Although the OVL does not currently provide a direct way to specify functional coverage, use this concept to create your own set of reusable templates that specify functional coverage points and encapsulate the functional coverage point's detection and reporting methods in an RTL module. This simplifies what an engineer must type when instantiating functional coverage points. In addition, it provides a centralized location for controlling, maintaining, and optimizing the detection and reporting mechanisms. This approach also adds an abstraction layer between the designer and the assertion language. The language used within the template library can be moved to the latest technology without the need to change the instantiated coverage points. Example 5-4 and Example 5-5 illustrate this method for a *queuefull* functional coverage point

Example 5-4 Coverage module in template library (Verilog)

```
module cover_monitor (clk, reset_n, test);
input clk, reset_n, test;
parameter event_id="COV_ ";
  `ifdef COVERAGE_ON
  // look for test condition
  always @ (posedge clk) begin
    if (reset_n == 1'bl && test) begin
      $display ("%s @ %0d:%t:%m", event_id, $time);
    end // end if (...test...)
  end
  `endif
endmodule
```

Example 5-5 Coverage module instantiation (Verilog)

```
// detect a queue full condition
cover_monitor # ("COV_Q1_FULL") dv_q_full(clk, reset_n, ql_full);
```

5.5.3 Assertion-based methods

You can use any assertion solution that allows for severity levels as the basis for functional coverage. By setting the severity for functional coverage instantiations to a non-error level, assertions can become functional coverage points.

Open Verification Library (OVL)

The Accellera OVL (www.openveriflib.org) provides a library of assertions that include severity levels and reporting parameters. These allow you to convert the assertions to functional coverage points and customize reporting. With this mechanism, the severity level changes the reporting information and the action required when the functional coverage point is activated.

Example 5-6 shows how the OVL **ovl_error** task can be modified to incorporate a separate severity level so that the OVL modules can be used for functional coverage. Example 5-7 shows how you can then use the OVL to define a functional coverage point. Please note when using this method, the event condition must be specified in a negative sense because assertions are activated when the condition fails.

Example 5-6 Modifying ovl_error for functional coverage

```
#define COVERAGE 2
task ovl_error;
    input [8*63:0]  err_msg;
  begin
    if (severity_level != `COVERAGE) begin
      error_count = error_count + 1;
       `ifdef ASSERT_MAX_REPORT ERROR
    if (error count <= `ASSERT_MAX_REPORT_ERROR)
     `endif
        $display("OVL_ERROR :%s:%s:%0s: severity %0d : time %0t:%m",
                    assert_name, msg, err msg,severity_level, $time);
      if (severity_level == 0) ovl_finish;
    endif
    else
      $display("OVL_COV :%s:%s:%0s : severity %0d : time %0t:%m",
                    assert_name, msg, err_msg,severity_level, $time);
  end
endtask
```

Example 5-7 Using OVL for functional coverage

```
assert_always # ( 'COVERAGE, 0, "Q_FULL") myQfull (clk, reset_n, !q_full);
```

SystemVerilog

The current SystemVerilog specification from Accellera (www.accellera.org) includes a language extension for adding assertions to the Verilog language. While often used for assertions, this new language extension is also effective for functional coverage points when you use appropriate severity levels. For example, use a severity level of *Error* for assertions and a severity level of *Info* for functional coverage points. At the conclusion of each simulation, use a post processing step to filter out messages that were printed with severity level *Info* and log or analyze them as you would any other style of functional coverage. Additionally, SystemVerilog is currently defining a `cover` feature that directly supports functional coverage.

As SystemVerilog is finalized, adopted as a new standard, and implemented in the various vendor simulators, it is possible that some of the differentiating features of the various tools will be to provide automatic logging of SystemVerilog assertions. Furthermore, it is hoped that vendors will include analysis tools to aid in reviewing functional coverage.

Example 5-8 shows how to use SystemVerilog to define a functional coverage point. Please note that the exact syntax may change when SystemVerilog specification is finalized.

Example 5-8 Using SystemVerilog for functional coverage

```
always @ (posedge clk) begin
  if (reset_n)
    myQfull: cover (q_full) $info("queue was full");
end
```

PSL The current formal property language specification from Accellera is named Property Specification Language (PSL). While PSL has only recently been finalized by Accellera, it is already beginning to be supported by simulators. Example 5-9 shows how to use PSL to define a functional coverage point.

Example 5-9 Using PSL for functional coverage

```
default clock = (posedge clk);
sequence qFullCondition = {reset_n & q_full};
cover qFullCondition;
```

5.5.4 Post processing

Another implementation of functional coverage includes a post-processing mechanism. This can be used exclusively to generate functional coverage by processing signal logs. Alternatively, you can use it to perform cross functional coverage analysis. In either case, the only simulation-time logging required is to capture the signal states you will need to evaluate during the post-processing steps. Once the simulation is complete, a custom post-processing script can parse the signal logs looking for functional coverage cases. This process was used by Kantrowitz and Noack [1996].

5.5.5 PLI logging and reporting

Standard Verilog-based functional coverage reporting techniques discussed earlier in this section are limited by the reporting capabilities of the language itself—since they are built around the system task `$display`. Since `$display` reports to `standard output`, this normally requires some sort of post processing step to parse a log of output. An alternative to this approach is to develop custom report libraries through the PLI interface. Since this ties the coverage points into a custom program, the possibilities are limitless for the types of processing that can be done.

5.5.6 Simulation control

The concepts of controlling functional coverage are similar to those of controlling assertions (discussed in Section 2.4 "Assertions and simulation" on page 42). Bening and Foster [2001] also describe several elements of controlling functional coverage. The exact method you use will vary depending on your functional coverage technology.

Enabling functional coverage. Since it includes additional checks and reporting, functional coverage reduces the performance of a simulation. For this reason, it is important to use some mechanism to enables and disable functional coverage. Refer to Section 5.4.5, "Coverage best practices" for more details on the need to control functional coverage.

‘ifdef

A simple mechanism such as `'ifdef COVERAGE_ON` provides a coarse process for controlling functional coverage. This method requires that *all* functional coverage be enabled or disabled.

global enable signal

A second method includes a *global enable signal* within the functional coverage point. The global enable signal ensures the coverage monitoring does not begin until after the system is out of reset and possibly initialization. This method is used within the OVL template libraries as shown in Example 5-10. The testbench drives the signal `'Top.dv_coverage_enable` shown in this example.

Example 5-10 Using global enable to control functional coverage (Verilog)

```
// check to ensure reset & init is done
if ( 'TOP.dv_coverage_enable) begin
   always @ (posedge clk) begin
      if (reset_n == 1'b1 && q_full) begin
         $display("cov_Q_FULL @ %0d:%t:%m", $time);
     end // end if (...q_full...)
   end // end always (...)
end // end if ('TOP...)
```

individual enable signal

Finally, use an additional *enable signal* to the port list of the functional coverage module itself. This method, while similar to the previous method, adds the capability of grouping functional coverage into categories. Each group of functional coverage within a single category uses a separate enable signal. This method gives finer control over enabling and disabling functional coverage.

Reset and initialization. In general, most functional coverage data is only useful when the design is out of reset and has been initialized. For this reason, it is important to provide reset and initialization state as inputs to the functional coverage points.

5.6 Functional coverage examples

Consider the FIFO model described in Example 3-6 (see page 70). The following examples show relevant functional coverage points we could implement for this FIFO.

Example 5-11 *OVL* FIFO coverage model

```
// when checking functional coverage with an OVL, the expression
// must be expressed in the negative.

`ifdef `COVERAGE_ON
//  FIFO full
assert_always  #( `COVERAGE,0,"FIFO_FULL") myFIFOFull  (clk,  reset_n,
            !({push,pop}==2'b10 && cnt==FIFO_depth-2));

// FIFO full-1
assert_always #(`COVERAGE, 0,"FIFO_FULL M1") myFIFOFullM1 (clk,
            reset_n, !({push,pop}==2'b10 & & cnt==FIFO_depth-3));

// FIFO full-2
assert_always  #(`COVERAGE, 0,"FIFO_FULL  M2")  myFIFOFullM2  (clk,
            reset_n,({push,pop}==2'b10 && cnt==FIFO_depth-4));

//  FIFO empty
assert_always #( `COVERAGE, 0,  "FIFO_EMPTY") myFIFOEmpty (clk,
                  reset_n, !( {push,pop}==2'b01 && cnt==1));

// unneccessary coverage point
assert_always #( `COVERAGE, 0,  "FIFO_EMPTY") myFIFOPush (clk,
                  reset_n,! (push==1'b1));
`endif  // COVERAGE_ON
```

Example 5-11 shows the coverage model for the FIFO. The obvious points of the FIFO being full and empty are covered. Notice in these points how the event condition provided to the monitor includes both the `cnt` level and the strobe indicating a push or a pop. If we consider only the `cnt` value, the coverage point would fire every clock cycle that the FIFO was full or empty. In many cases, the FIFO may become full and stay full for five cycles before a value is popped out of the FIFO. We want the functional coverage to indicate we filled the FIFO one time in this case, not five times. Next, we include two additional coverage points that indicate when the FIFO is one entry short of full and two entries short of full. (When an engineer is working on a test to fill the FIFO, it is helpful to have some feedback on whether the test is getting close to the target of filling the FIFO.)

We discussed how important it is to carefully consider whether to add a specific functional coverage point. In Example 5-11, an additional functional coverage point monitors the number of times a value was pushed onto the FIFO. While this is a valid coverage point, we should exclude it because it does not provide valuable information. Example 5-12 and Example 5-13 show the same functional coverage model implemented in PSL and SystemVerilog, respectively. In these two examples, we do not include the unnecessary functional coverage point.

Example 5-12 *PSL* FIFO coverage model

```
default clock = (posedge clk);

sequence COVER_FIFO_FULL ={reset_n && rose(cnt==(FIFO_depth-1))};
cover COVER_FIFO_FULL;

sequence COVER_FIFO_FULL_M1 = {reset_n && rose (cnt==(FIFO_depth-2))};
cover COVER_FIFO_FULL_M1;

sequence COVER_FIFO_FULL_M2 = {reset_n && rose (cnt==(FIFO_depth-3))};
cover COVER_FIFO_FULL_M2;

sequence COVER FIFO_EMPTY = {reset_n && rose(cnt == 0)};
cover COVER_FIFO_EMPTY;
```

Example 5-13 also shows an alternative method to capture only the initial entry into a full or empty condition.

Example 5-13 *System Verilog* FIFO coverage model

```
`ifdef 'COVERAGE_ON
always @ (posedge clk) begin
  if  (reset_n)  begin
    //  FIFO  full
    myQfull: cover property ($rose(cnt ==(FIFO_depth-1)));

    //  FIFO  full-1
    myQfullm1: cover property ($rose(cnt ==(FIFO_depth-2)));

    //  FIFO full-2
    myQfullm2: cover property ($rose(cnt ==(FIFO_depth-3)));

    //  FIFO  empty
    myQempty: cover property ($rose(cnt == 0));
  end
end
`endif // COVERAGE_ON
```

Continue to consider the FIFO example. If the design uses this FIFO to store command packets prior to a decode block, it is interesting to know when the different command packets are popped from the FIFO by the decode block. Example 5-14 shows examples of functional coverage points that monitor for the following command packets: `` `READ``, `` `WRITE``, `` `IO_READ``, `` `IO_WRITE``.

Example 5-14 *OVL* FIFO command packet coverage points

```
// when checking functional coverage with an OVL, the expression
// must be expressed in the negative.

 `ifdef `COVERAGE_ON

assert_always #(`COVERAGE,0,"CMD_READ") read (clk, reset_n,
             !(pop==1'b1  &&  data_out==`READ));

assert_always #(`COVERAGE,0,"CMD_IO_READ") io_read (clk, reset_n,
             !(pop==1'b1  &&  data_out==`IO_READ));

assert_always #(`COVERAGE, 0, "CMD_WRITE") write (clk, reset_n,
             !(pop==1'b1  && data_out== `WRITE));

assert_always #(`COVERAGE, 0, "CMD_IO_WRITE") io_write (clk, reset_n,
             !(pop==1'b1  &&  data_out==`IO_WRITE));
 `endif // COVERAGE_ON
```

5.7 AHB example

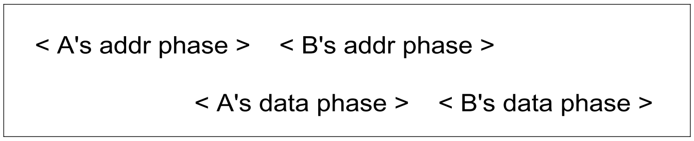
In this section, we discuss a transaction modeling technique proposed by Marschner et al. [2002] for the Advanced High-Performance Bus (AHB) protocol—supported by the ARM Advanced Microcontroller Bus Architecture (AMBA). The technique they proposed is based on specifying a set of PSL sequences, which represent various AHB transactions. The set of sequences are then combined into an assertion that requires only valid transactions to occur following the completion of any previous transaction (indicated by `hready` going high), as demonstrated in Example 5-15.

Example 5-15 *PSL* AHB valid transactions following the completion of any previous transaction

```
assert always ({hready} |=> { SERE_AHB_BURST_MODE_READ
                            | SERE_AHB_BURST_MODE_WRITE
                            | SERE_AHB_SINGLE_READ
                            | SERE_AHB_SINGLE_WRITE
                            | SERE_AHB_INACTIVE
                            | SERE_AHB_RESET
                            });
```

AHB is a pipelined bus with all transfers taking at least two cycles to complete. For example, consider two transfers A and B demonstrated in Figure 5-1:

Figure 5-1 **AHB pipeline bus**



The Slave's response to the address (and control) phase occurs one cycle later in the data phase. The Slave can either set `hready` high, to acknowledge the data, or set it low inserting a wait cycle in the data phase (and consequently, the next address phase).

The address phase also includes control. Hence, an `htrans` transfer occurs in the address phase and the slave response (with `hready` & `hresp`) occurs one cycle later.

Marschner et al. [2002] demonstrated how to specify the AHB burst-mode read transaction assertion as a set of sequences. We now extend their transaction specification discussion by demonstrating how to create a functional coverage model for an AHB burst-mode read.

A burst-mode read transaction can be specified as a sequence of a *first read* operations, followed by one or more sequences of *next read* operations, for which the address and data are pipelined. Hence, a simplified description for the burst-mode read transaction can be described as follows. The transaction initially begins when the slave is operating on the previous data transfer. At this point, the master will request a first data transfer. Then, for all subsequent burst-mode read transfers (until the transaction completes), the slave will operate on the previous data transfer while the master is requesting the next data transfer or it signals that it is busy (that is, `hburst== 'BUSY)`.

Example 5-16 demonstrates how to specify the transaction for a burst-mode read as a sequence in PSL. This sequence can then be used to build a functional coverage model (using the PSL cover directive). During the course of simulation, any occurrence of a burst-mode read transaction is reported. Conversely, if burst-mode read transaction is never detected across our entire suite of regression tests, then we know that some fundamental aspect of the design has gone untested. Note that this specification, although useful for measuring functional coverage, would have to be made a little more precise when used as an assertion (for example, an assertion would need to account for a BUSY transfer).

Example 5-16 ***PSL* AHB read burst mode transaction**

```
`define AHB_WAIT (!hready && (hresp==`OKAY))
`define AHB_OKAY (hready && (hresp==`OKAY))
`define AHB_ERROR (hresp==`ERROR)
`define AHB_SPLIT (hresp==`SPLIT)
`define AHB_RETRY (hresp==`RETRY)

sequence SERE_AHB_SLAVE_RESPONSE = {
  `AHB_WAIT[*];
  {
    { `AHB_OKAY}
  | { { !hready; hready} && {`AHB_ERROR [*2]} }
  | { { !hready; hready} && {`AHB_SPLIT [*2]} }
  | { { !hready; hready} && {`AHB_RETRY [*2]} }
  }
};

`define AHB_FIRST_TRANS (htrans==`NONSEQ)
`define AHB_NEXT_TRANS (htrans==`SEQ)
`define AHB_MASTER_BUSY (htrans==`BUSY)
`define AHB_READ_INCR(!hwrite  &&  (hburst==`INCR)

// slave response to the previous data in parallel with the master's
// assertion of the control signals for the next address

sequence SERE_AHB_READ_FIRST = {
  {SERE_AHB_SLAVE_RESPONSE} &&
  {(`AHB_FIRST_TRANS && `AHB_READ_INCR)[*]}
};

sequence SERE_AHB_READ_NEXT = {
  {SERE_AHB_SLAVE_RESPONSE} &&
  {
    {(AHB_NEXT_TRANS && `AHB_READ_INCR)  [*]} |
    {`AHB_MASTER_BUSY[*] }
  }
};

sequence SERE_AHB_BURST_MODE_READ = {
  {SERE_AHB_READ_FIRST}; {SERE_AHB_READ_NEXT}[*]
};

cover {SERE_AHB_BURST_MODE_READ};
```

Note that in addition to specifying the large transaction as a functional coverage point, we recommend that you also specify the various sequence segments to provide finer granularity in identifying exactly what behaviors have been covered.

5.8 Summary

This chapter introduced the concept of functional coverage and discussed its role in the verification process. We provided specific details to allow the reader to construct an effective functional coverage methodology. Finally, we discussed sources of functional coverage technology.