Coverage Cookbook

COOKBOOK

Mentor®
A Siemens Business

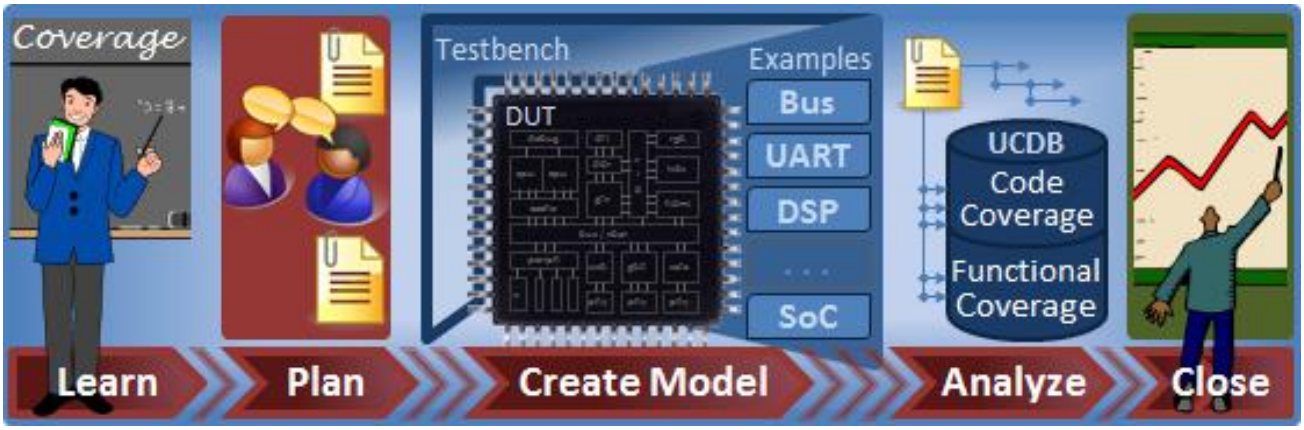
FUNCTIONAL VERIFICATION

www.verificationacademy.com

Table of Contents

Coverage

The Coverage Cookbook describes the different types of coverage that are available to keep track of the progress of the verification process, how to create a functional coverage model from a specification, and provides examples of how to implement functional coverage for different types of designs.



Coverage Cookbook contents:

- BiQuad IIR Filter test plan - A test plan for the BiQuad IIRFilter
- BiQuad IIR Filter example covergroups - Code fragments to illustrate the implementation of the BiQuad IIR functional coverage model

SoC coverage example - Explains the process for creating a SoC functional coverage model based on use cases

Appendices

- Requirements Writing Guidelines - Guidelines for thinking about and writing requirements
- Please note that it may not always be possible or appropriate to supply source code for all of the examples in the Coverage Cookbook

Introduction

Coverage Cookbook

As the saying goes, "What doesn't get measured likely won't get done." And that is certainly true when trying to determine a design project's verification progress or trying to answer the important question, "Are we done?" Whether your simulation methodology is based on a directed testing approach or constrained-random verification, to understand your verification progress, you need to answer the following questions:

- Were all the design features and requirements identified in the testplan verified?
- Were some lines of code or structures in the design model never exercised?

Coverage is the metric we use during simulation to help us answer these questions. Yet, once coverage metrics become an integral part of our verification process, it opens up the possibility for more accurate project schedule predictions, as well as providing a means for optimizing our overall verification process. At this stage of maturity, we can ask questions such as:

- When we tested feature X, did we ever test feature Y at the exact same time?
- Has our verification progress stalled for some unexpected reason?
- Are there tests that we could eliminate to speed up our regression suite and still achieve our coverage goals?

The book you are holding contains excerpts from the online Coverage Cookbook resource, which is evolving to address all aspects of a coverage-driven verification methodology, such as: coverage planning, coverage modeling, coverage implementation, coverage analysis, and coverage closure. Check out the Coverage Cookbook website for a set of downloadable examples contained in this book—and join a community of engineers interested in learning how to leverage coverage on their projects.

Find us online at https://verificationacademy.com/cookbook

Coverage Cookbook Authors:

- Gordon Allan
- Gabriel Chidolue
- Thomas Ellis
- Harry Foster
- Michael Horn
- Peet James
- Mark Peryer

Coverage Metrics and process (Theory)

What is coverage?

As the saying goes, "What doesn't get measured might not get done." And that is certainly true when trying to determine a design project's verification progress, or trying to answer the question "Are we done?" Whether your simulation methodology is based on a directed testing approach or constrained-random verification, to understand your verification progress you need to answer the following questions:

- Were all the design features and requirements identified in the testplan verified?
- Were there lines of code or structures in the design model that were never exercised?

Coverage is the metric we use during simulation to help us answer these questions. Yet, once coverage metrics become an integral part of our verification process, it opens up the possibility for more accurate project schedule predictions, as well as providing a means for optimizing our overall verification process. At this stage of maturity we can ask questions such as:

- When we tested feature *X*, did we ever test feature *Y* at the exact same time?
- Has our verification progress stalled for some unexpected reason?
- Are there tests that we could eliminate to speed up our regression suite and still achieve our coverage goals?

Hence, coverage is a simulation metric we use to measure verification progress and completeness.

Observability and Controllability

Fundamental to the discussion of coverage is understanding the concepts of *controllability* and *observability*. Informally, controllability refers to the ability to influence or activate an embedded finite state machine, structure, specific line of code, or behavior within the design by stimulating various input ports. Note that, while in theory a simulation testbench has high controllability of the design model's input ports during verification, it can have very low controllability of an internal structure within the model. In contrast, observability refers to the ability to observe the effects of a specific internal finite state machine, structure, or stimulated line of code. Thus, a testbench generally has limited observability if it only observes the external ports of the design model (because the internal signals and structures are often indirectly hidden from the testbench).

To identify a design error using a simulation testbench approach, the following conditions must hold:

1. The testbench must generate proper input stimulus to activate a design error.
2. The testbench must generate proper input stimulus to propagate all effects resulting from the design error to an output port.
3. The testbench must contain a monitor that can detect the design error that was first activated then propagated to a point for detection.

It is possible to set up a condition where the input stimulus activates a design error that does not propagate to an observable output port. In these cases, the first condition cited above applies; however, the second condition is absent, as illustrated in Figure 1.

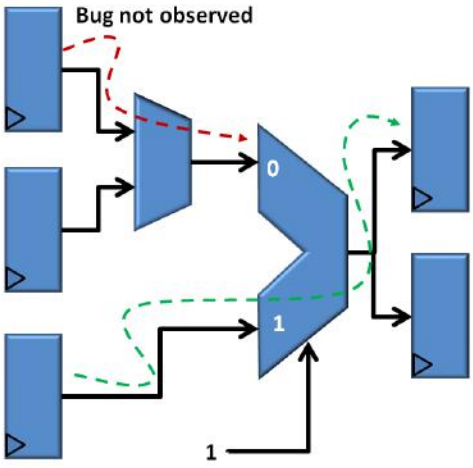


Figure 1. Poor observability & controllability misses bugs

In general, coverage is a metric we use to meaure the controllability quality of a testbench. For example, code coverage can directly identify lines of code that were never activated due to poor controllability issues with the simulation input stimulus. Similarly, functional coverage can identify expected behaviors that were never activated during a simulation run due to poor controllability.

Although our discussion in this section is focused on coverage, it's important to note that we can address observability concerns by embedding assertions in the design model to facilitate low-level observability, and creating monitors within and on the output ports of our testbench to facilitate high-level observability.

Summary

So what is coverage? Simply put, coverage is a metric we use to measure verification progress and completeness. Coverage metrics tells us what portion of the design has been activated during simulation (that is, the controllability quality of a testbench). Or more importantly, coverage metrics identify portions of the design that were never activated during simulation, which allows us to adjust our input stimulus to improve verification.

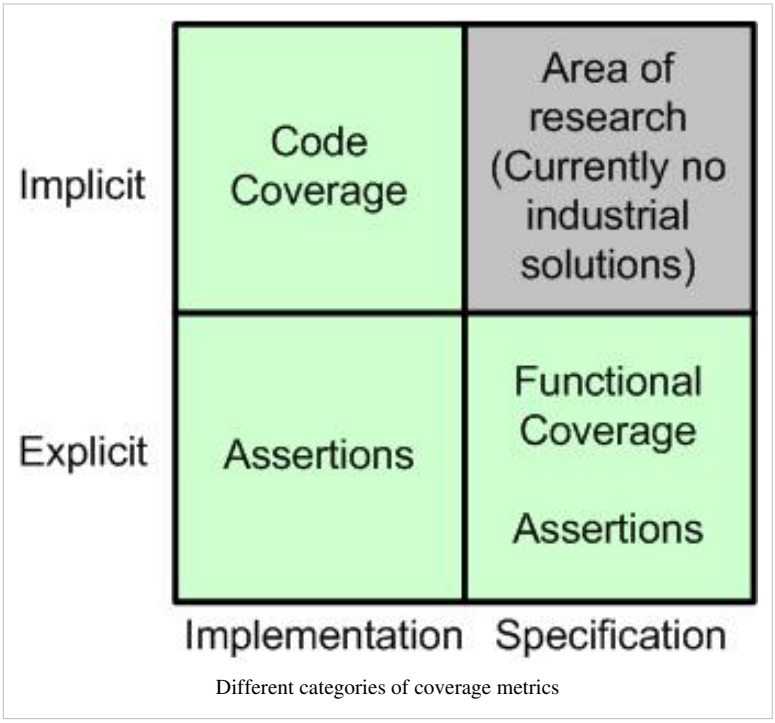
There are different kinds of coverage metrics available to you, and the process of how to use them is discussed in the Coverage Cookbook examples.

Kinds of coverage

No single metric is sufficient at completely characterizing the verification process. For example, we might achieve 100% code coverage during our simulation regressions. However, this would not mean that 100% of the functionality was verified. The reason for this is that code coverage does not measure the concurrent interaction of behavior within, or between multiple design blocks, nor does it measure the temporal sequences of functional events that occur within a design. Similarly, we might achieve 100% functional coverage, yet only achieve 90% code coverage. This might indicate that there is either a problem with the fidelity in our functional coverage model (that is, an important behavior of the design was missing from the coverage model), or possibly some functionality was implemented that was never initially specified (for example, perhaps the specification and testplan needs to be updated with some late stage change in the requirements). Hence, to get a complete picture of a project's verification progress we often need multiple metrics.

Coverage Classification

To begin our discussion on the kinds of coverage metrics, it is helpful to first identify various classifications of coverage. In general, there are multiple ways in which we might classify coverage, but the two most common ways are to classify them by either their **method of creation** (such as, *explicit* versus *implicit*), or by their **origin of source** (such as, *specification* versus *implementation*).



Different categories of coverage metrics

For instance, functional coverage is one example of an explicit coverage metric, which has been manually defined and then implemented by the engineer. In contrast, line coverage and expression coverage are two examples of an implicit coverage metric since its definition and implementation is automatically derived and extracted from the RTL representation.

Coverage Space Classification

Coverage associated with the two categories we just described can be combined to form a coverage space, which is often referred to as a coverage model. [1] For instance, an explicit specification coverage space consists of coverage metrics that are manually created by an engineer, derived from a design's requirements document or specification. Another kind of explicit coverage is the instrumentation created by an engineer that is based on the behavior encapsulated by the design implemention, such as the filling or emptying events associated with a particular FIFO in an RTL model.

Similarly, an implicit implementation coverage space consists of coverage metrics that are automatically extracted by a tool (such as a simulator), and derived from a design implementation (such as an RTL model). Another part of the implicit specification coverage space consists of coverage metrics that are automatically extracted by a tool, and are derived from the design specification. This part of the coverage space is currently an area of academic research, although there have been a few EDA tools recently emerge that attempt to automatically extract higher-level coverage properties by observing the effects of simulation patterns on an implementation (such as an RTL model). Note that these higher-level functional behaviors cannot be automatically extracted from the implementation alone, which is why they fall into the coverage metrics associated with the implicit specification coverage space.

Coverage Metrics

There are two primary forms of coverage metrics in production use in industry today and these are:

- Code Coverage Metrics (Implicit coverage)
- Functional Coverage/Assertion Coverage Metrics (Explicit coverage)

References

[1] A. Piziali, *Functional Verification Coverage Measurement and Analysis*, Kluwer Academic Publishers, 2004.

Code Coverage

In this section, we introduce various coverage metrics associated with a design model's implicit implementation coverage space. In general, these metrics are referred to as code coverage or structural coverage metrics.

Benefits:

Code coverage, whose origins can be traced back to the 1960's, is one of the first methods invented for systematic software testing.[1] One of the advantages of code coverage is that it automatically describes the degree to which the source code of a program has been activated during testing-thus, identifying structures in the source code that have not been activated during testing. One of the key benefits of code coverage, unlike functional coverage, is that creating the structural coverage model is an automatic process. Hence, integrating code coverage into your existing simulation flow is easy and does not require a change to either your current design or verification approach.

Limitations:

In our section titled What is coverage, we discussed three important conditions that must occur during simulation to achieve successful testing. They were:

1. The testbench must generate proper input stimulus to activate a design error.
2. The testbench must generate proper input stimulus to propagate all effects resulting from the design error to an output port.
3. The testbench must contain a monitor that can detect the design error that was first activated then propagated to a point for detection.

Code coverage is a measurement of structures within the source code that have been activated during simulation. One limitation with code coverage metrics are that you might achieve 100% code coverage during your regression run, which means that your testbench provided stimulus that activated all structures within your RTL source code, yet there are still bugs in your design. For example, the input stimulus might have activated a line of code that contained a bug, yet the testbench did not generate the additional required stimulus that propagates the effects of the bug to some point in the testbench where it could be detected. In fact, researchers have studied this problem and found cases where a testbench achieved 90% code coverage-yet, only 54% of the code was covered would be observable during a simulation run.[2] That means that a bug could exist on a line of code that had been marked as covered—yet the bug was never detected due to insufficient input stimulus to propagate the bug to an observability point.

Another limitation of code coverage is that it does not provide an indication on exactly what functionality defined in the specification was actually tested. For example, you could run into a situation where you achieved 100% code coverage, and then assume you are done. Yet, there could be functionality defined in the specification that was never tested—or even functionality that had never been implemented! Code coverage metrics will not help you find these situations.

Even with these limitations, the automatic aspect of code coverage makes it a relatively simple way to identify input stimulus deficiencies in your testbench. And is a great first choice for coverage metrics as you start to evolve your advanced verification process capabilities.

Types of Code Coverage Metrics

Toggle Coverage

Toggle coverage is a code coverage metric used to measure the number of times each bit of a register or wire has toggled its value. Although this is a realitively basic metric, many projects have a testing requirement that all ports and registers, at a minimum, must have experienced a zero-to-one and one-to-zero transition.

In general, reviewing a toggle coverage analysis report can be overwhelming and of little value if not carefully focused. For example, toggle coverage is often used for basic connectivity checks between IP blocks. In addition, it can be useful to know that many control structures, such as a one-hot select bus, have been fully exercised.

Line Coverage

Line coverage is a code coverage metric we use to identify which lines of our souce code have been executed during simulation. A line coverage metric report will have a count associated with each line of source code indicating the total number of times the line has executed. The line execution count value is not only useful for identifying lines of source code that have never executed, but also useful when the engineer feels that a minimum line execution threshold is required to achieve sufficient testing.

Line coverage analysis will often reveal that a rare condition required to activate a line of code has not occurred due to missing input stimulus. Alternatively, line coverage analysis might reveal that the data and control flow of the source code prevented it either due to a bug in the code, or dead code that is not currently needed under certain IP configurations. For unused or dead code, you might choose to exclude or filter this code during the coverage recording and reporting steps, which allows you to focus only on the relavent code.

Statement Coverage

Statement coverage is a code coverage metric we use to identify which statements within our souce code have been executed during simulation. In general, most engineers find that statement coverage analysis is more useful than line coverage since a statement often spans multiple lines of source code-or multiple statements can occur on a single line of source code.

A metrics report used for statement coverage analysis will have a count associated with each line of source code indicating the total number of times the statement has executed. This statement execution count value is not only useful for identifying lines of source code that have never executed, but also useful when the engineer feels that a minimum statement execution threshold is required to achieve sufficient testing.

Block Coverage

Block coverage is a variant on the statement coverage metric which identifies whether a block of code has been executed or not. A block is defined as a set of statements between conditional statements or within a procedural definition, the key point being that if the block is reached, all the lines within the block will be executed. This metric is used to avoid unscrupulous engineers from achieving a higher statement coverage by simply adding more statements to their code.

Branch Coverage

Branch coverage (also referred to as decision coverage) is a code coverage metric that reports whether Boolean expressions tested in control structures (such as the *if*, *case*, *while*, *repeat*, *forever*, *for* and *loop* statements) evaluated to both true and false. The entire Boolean expression is considered one true-or-false predicate regardless of whether it contains logical-and or logical-or operators.

Expression Coverage

Expression coverage (sometimes referred to as condition coverage) is a code coverage metric used to determine if each condition evaluated both to true and false. A condition is an Boolean operand that does not contain logical operators. Hence, expression coverage measures the Boolean conditions independently of each other.

Focused Expression Coverage

Focused Expression Coverage (FEC), which is also referred to as Modified Condition/Decision Coverage (MC/DC), is a code coverage metric often used used by the DO-178B safety critical software certification standard, as well as the DO-254 formal airborne electronic hardware certification standard. This metric is stronger than condition and decision coverage. The formal definition of MC/DC as defined by DO-178B is:

Every point of entry and exit in the program has been invoked at least once, every condition in a decision has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decisions outcome. A condition is shown to independently affect a decisions outcome by varying just that condition while holding fixed all other possible conditions. [3]

It is worth noting that completely closing Focused Expressing Coverage can be non-trivial.

Finite-State Machine Coverage

Today's code coverage tools are able to identify finite state machines within the RTL source code. Hence, this makes it possible to automatically extract FSM code coverage metrics to measure conditions. For example, the number of times each state of the state machine was entered, the number of times the FSM transitioned from one state to each of its neighboring states, and even sequential arc coverage to identify state visitation transitions.

Typical Code Coverage Flow

The objective of gathering and analyzing code coverage metrics is to identify portions of the source code that have not been exercised by the current verification environment. From a project perspective, it is generally best to wait until the implementation of the RTL is close to complete before seriously starting to gather and analyze code coverage results. Otherwise, you can waste a lot of cycles trying to make sense of a moving target from the changing RTL code. With that said, we recommend that you at least run a few simulations that capture coverage metrics early in the project cycle (that is, prior to seriously gathering coverage metrics) to work out any potential issues in your coverage flow.

From a high-level perspective, there are generally three main steps involved in a code coverage flow, which include:

1. Instrument the RTL code to gather coverage
2. Run simulation to capture and record coverage metrics
3. Report and analyze the coverage results

Part of the analysis step is to identify coverage holes, and determine if the coverage hole is due to one of three conditions:

1. Missing input stimulus required to activate the uncovered code
2. A bug in the design (or testbench) that is preventing the input stimulus from activating the uncovered code
3. Unused code for certain IP configuations or expected unreachable code related during normal operating conditions

The first condition requires you to either write additional directed stimulus or adjust random constraints to generate the required input stimulus that targets the uncovered code. The second condition obviously requires the engineer to fix the bug that is preventing the uncovered code from being exercised. The third condition can be addressed by directing the

coverage tool to exclude the unused or unreachable code during the coverage recording and reporting steps. Formal tools can be used to automate the identification of unreachable code, and then automatically generate the exclusion files.

References

[1] J. Miller, C. Maloney, "Systematic mistake analysis of digital computer programs." *Communications of the ACM* 6 (2): 58-63, February 1963.

[2] F. Fallah, S. Devadas, K. Keutzer: "OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification." *Proceedings of the Design Automation Conference*, 1998: 152-157

[3] DO-178B, "Software Considerations in Airborne Systems and Equipment Certification", RCTA, December 1992, pp.31, 74.

[4] M. Stuart, D. Dempster: Verification Methodology Manual for Code Coverage in HDL Designs - TransEDA, August 2000

Functional Coverage

The objective of functional verification is to determine if the design requirements, as defined in our specification, are functioning as intended. But how do you know if all the specified functionality was actually implemented? Furthermore, how do we know if all the specified functionality was really tested? Code coverage metrics will not help us answer these questions.

In this section, we introduce an *explicit* coverage metric referred to as *functional coverage*, which can be associated with either the design's specification or implementation coverage space. The objective of measuring functional coverage is to measure verification progress with respect to the functional requirements of the design. That is, functional coverage helps us answer the question: Have all specified functional requirements been implemented, and then exercised during simulation? The details on how to create a functional coverage model are discussed separately in the Testplan to functional coverage chapter.

Benefits:

The origin of functional coverage can be traced back to the 1990's with the emergence of constrained-random simulation. Obviously, one of the value propositions of constrained-random stimulus generation is that the simulation environment can automatically generate thousands of tests that would have normally required a significant amount of manual effort to create as directed tests. However, one of the problems with constrained-random stimulus generation is that you never know exactly what functionality has been tested without the tedious effort of examining waveforms after a simulation run. Hence, functional coverage was invented as a measurement to help determine exactly what functionality a simulation regression tested without the need for visual inspection of waveforms.

Today, the adoption of functional coverage is not limited to constrained-random simulation environments. In fact, functional coverage provides an automatic means for performing *requirements tracing* during simulation, which is often a critical step required for DO-254 compliance checking. For example, functional coverage can be implemented with a mechanism that links to specific requirements defined in a specification. Then, after a simulation run, it is possible to automatically measure which requirements were checked by a specific directed or constrained-random test—as well as automatically determine which requirements were never tested.

Limitations:

Since functional coverage is not an implicit coverage metric, it cannot be automatically extracted. Hence, this requires the user to manually create the coverage model. From a high-level, there are two different steps involved in creating a functional coverage model that need to be considered:

1. Identify the functionality or design intent that you want to measure
2. Implementing the machinery to measure the functionality or design intent

The first step is addressed through verification planning, and the details are addressed in the section on getting from a testplan to functional coverage.

The second step involves coding the machinery for each of the coverage items identified in the verification planning step (for example, coding a set of SystemVerilog covergroups for each verification objective identified in the verification plan). During the coverage model implementation phase, there are also many details that need to be considered, such as: identifying the appropriate point to trigger a measurement and defining controllability (disable/enable) aspects for the measurement. These and many other details are addressed in the detailed coverage examples.

Since the functional coverage must be manually created, there is always a risk that some functionality that was specified is missing in the coverage model.

Types of Functional Coverage Metrics

The functional behavior of any design, at least as observed from any interface within the verification environment, consists of both data and temporal components. Hence, from a high-level, there are two main types of functional coverage measurement we need to consider: *Cover Groups'* **and Cover Properties.**

Cover Group Modeling

With respect to functional coverage, the samping of state values within a design model or on an interface is probably the easiest to understand. We refer to this form of funcational coverage as *cover group modeling*. It consists of state values observed on buses, grouping of interface control signals, as well as register. The point is that the values that are being measured occur at a single explicitly or implicitly sampled point in time. SystemVerilog covergroups are part of the machinery we typically use to build the functional data coverage models, and the details are discussed in the block level design example and the discussion of the corresponding example covergroup implementations.

Cover Property Modeling

With respect to functional coverage, temporal relationships between sequences of events are probably the hardest to reason about. However, ensuring that these sequences of events are properly tested is important. We use *cover property modeling* to measure temporal relationships between sequences of events. Probably the most popular example of cover properties involves the handshaking sequence between control signals on a bus protocol. Other examples include power-state transition coverage associated with verifying a low-power design. Assertions and coverage properties are part of the machinery that we use to build temporal coverage models, and are addressed in the bus protocol monitor example.

Assertion Coverage

The term assertion coverage has many meanings in the industry today. For example, some people define assertion coverage as the ratio of number of assertions to RTL lines of code. However, assertion density is a more accurate term that is often used for this metric. For our discussion, we use the term assertion coverage to describe an implementation of of coverage properties using assertions.

Difference Between Cover Groups and Cover Properties

To help illustrate the difference between coverage modeled with covergroups and cover properties, lets look at a simple nonpipelined bus example, as illustrated in Figure 1.

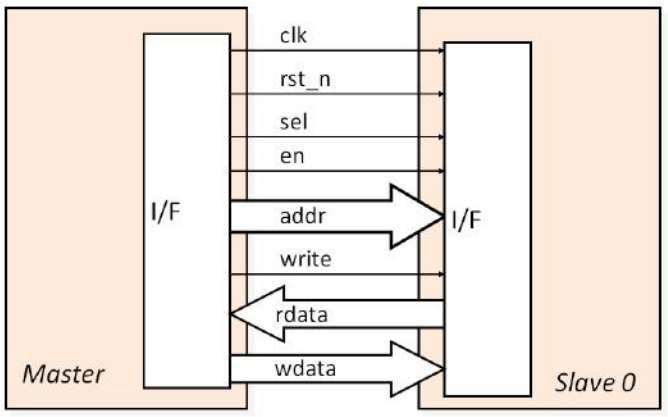


Figure 1. Simple nonpipelined bus interface

A single write and read bus sequence for our non-pipelined bus protocol are illustrated in Figure 2.

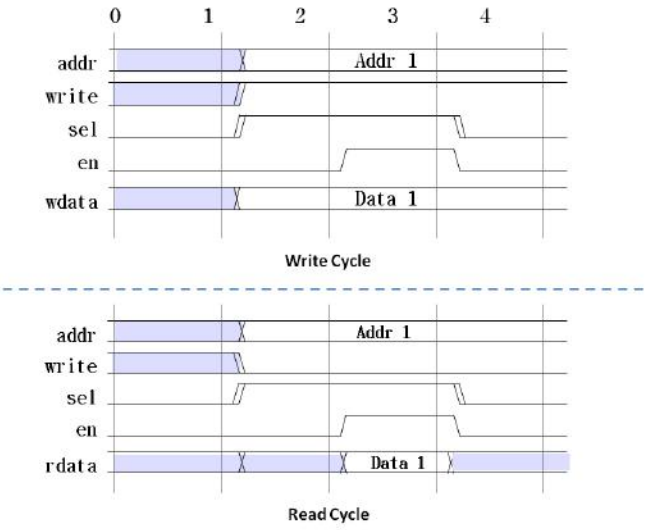


Figure 2. Write and read cycles for a simple nonpipelined bus protocol

To verify our bus example, it's important to test the boundary conditions for the address bus for both the write sequence and read sequence (that is, the bits within *addr* at some point contained all zeros and all ones). In addition, it's also important that we have covered a sufficient number of non-boundary conditions on the address bus during our regression. We are only interested in sampling the address bus when the slave is selected and the enable strobe is active (that is, sel==1'b1 && en==1'b1). Finally, we will want to keep track of separate write and read events for these coverage items to ensure that we have tested both these operations sufficiently.

This is one example of using *cover groups* to model functional coverage (e.g., the SystemVerilog *covergroup* construct). In addition, we could apply the same data coverage approach to measuring the read and write data busses.

Now, let's look at *cover properties* with respect to this example. There is a standard sequence that is followed for both the write and read cycle. For example, let's examine a write cycle. At clock one, since both the slave select (*sel*) and bus enable (*en*) signals are de-asserted, our bus is in an INACTIVE state. The first clock of the write sequence is called the bus START state, which the master initiates by asserting one of the slave select line (*sel==1'b1*). During the START state, the master places a valid address and valid data on the bus. The data transfer (referred to as the bus ACTIVE state) actually occurs when the master asserts the bus enable strobe signal (*en*). In our case, it is detected on the rising edge of clock three. The address, data, and control signals all remain valid throughout the ACTIVE state.

When the ACTIVE state completes, the bus enable strobe signal (*en*) is de-asserted by the bus master, and thus completes the current single write operation. If the master has finished transferring all data to the slave (that is, there are no more write operations), then the master de-asserts the slave select signal (*sel*). Otherwise, the slave select signal remains asserted, and the bus returns to the bus START state to initiate a new write operation. Multiple back-to-back write operations (without returning to the bus INACTIVE state) are known as burst write.

From a temporal coverage perspective, a set of assertions could be written to ensure proper sequencing of states on the bus. For example, the only legal bus state transitions are illustrated in Figure 3. Furthermore, it's important to test a single write and read cycle, as well as the burst read in write operation. In fact, we might want to measure the various burst write and read cycles.

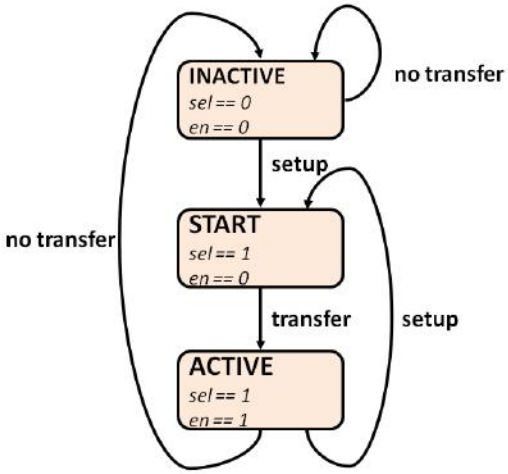


Figure 3. Legal non-pipelined bus state sequence

By combining *cover groups* and *cover properties*, we are able to achieve a higher fidelity coverage model that more accurately allows us to measure key features of the design.

Details on how to code temporal coverage are covered in the **APB3 Bus protocol monitor example.**

Typical Functional Coverage Flow

In many respects, the typical functional coverage flow is very similar to the typical code coverage flow. The objective of gathering and analyzing functional coverage metrics is to identify features and requirements from the specification that have not been exercised by the current verification environment and the test cases run on it. From a project perspective, it is generally best to wait until the implementation of the test cases associated with coverage are close to complete before seriously starting to gather and analyze functional coverage results. Otherwise, you can waste a lot of cycles trying to make sense of a moving target from the changing stimlus. With that said, we recommend that you at least run a few

simulations that capture coverage metrics early in the project cycle (that is, prior to seriously gathering coverage metrics) to work out any potential issues in your coverage flow.

From a high-level perspective, there are generally four main steps involved in a functional coverage flow, which include:

1. Create a functional coverage model
2. If using assertions, instrument the RTL model to gather coverage
3. Run simulation to capture and record coverage metrics
4. Report and analyze the coverage results

Part of the analysis step is to identify coverage holes, and determine if the coverage hole is due to one of three conditions:

1. Missing input stimulus required to activate the uncovered functionality
2. A bug in the design (or testbench) that is preventing the input stimulus from activating the uncovered functionality
3. Unused functionality for certain IP configuations or expected unreachable functionality related during normal operating conditions

The first condition requires you to either write additional directed tests or adjust random constraints to generate the required input stimulus that targets the uncovered functionality. The second condition obviously requires the engineer to fix the bug that is preventing the uncovered functionality from being exercised. The third condition can be addressed by directing the coverage tool to exclude the unused or unreachable functionality during the coverage recording and reporting steps.

Specification to testplan

Testplan Creation Approaches

The goal in creating a coverage model spreadsheet or testplan is to capture a subset of the design intent and behavior that is targeted for functional coverage. It is a time consuming, manual process that involves combing over various design specification documents and extracting the necessary requirements one at a time. It is best if this is done by a cross functional team staffed by architects, designers, firmware and verification engineers to get multiple points of view and different inputs. Without a cross functional aspect, various subsets of the design intent are easily missed. Creating the testplan is best done by holding multiple meetings, each of which targets a particular design area (the xyz block), and lasts for a fixed length of time (1 hour, every morning next week at 9am) and with a goal (50 requirements). Generally, there are two approaches that can be taken:

1. Bottom Up: Go over block by block or interface by interface
2. Top Down: Follow the use model(s) or data flow of the chip.

Two Approaches

	Bottom Up	Top Down
Definition	Extract requirements from available low level, detailed design and implementation specifications. This approach is more design oriented.	Extract requirements from high level architecture and use model specifications. This approach is customer/verification/user oriented.
Pros	• Low hanging fruit: Easiest to find, extract and prioritize. • Easier to link to coverage. • Easier to close on coverage goals. • Because you comb over every block and interface, key, highly specific and important coverage is picked up that might be glossed over by the top down method.	• Can give more useful, high level, interesting coverage information, such as utilization, to explore tradeoffs. • Can be done before design specs are completed, without implementation details. • Goes towards intelligent testbench automation (ITA - Infact) using flow chart graphs. • Forces a customer centric look at the design.
Cons	• Need well developed specs with implementation details. • Can lead to an explosion of requirements. Too many to implement in a reasonable amount of time. Needs prioritization. • Tend to be low level, uninteresting coverage. Lots of data, little useful information to explore tradeoffs.	• Needs access to high level specifications or architects with clear use model definitions. • Use model(s) can sometimes grow exponentially and result in a huge coverage space with too many iterations. • Coverage tends to be more upstream, generation oriented coverage, not downstream DUT or Scoreboard oriented. This can be misleading.
Approach	Have a series of meetings each focused on a subset of the design, such as a block or interface, and gather the appropriate specifications and engineering personnel to extract out the requirements, refine them, prioritize them, and link them to some coverage group, coverage point or cross in a spreadsheet.	Have a series of meetings with the architect and come up with a single high level use model first, then create a use model(s) document the goes into further details using lots of diagrams (tables, graphs, etc.) and minimal words. Then take this document and rework it into spreadsheet format.

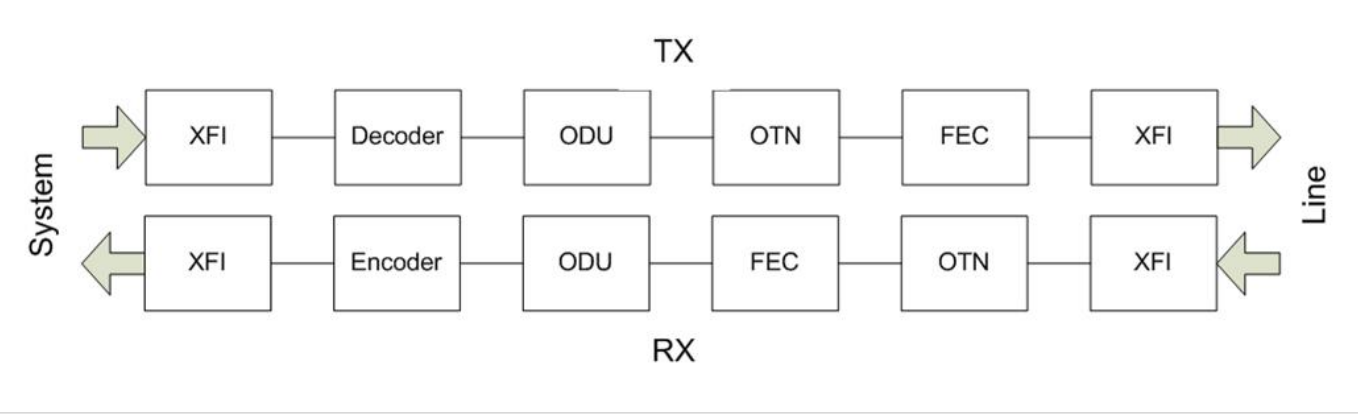
Choosing Between The Bottom Up And Top Down Approaches

Bottom Up	Top Down
Small Designs	Large Designs
Good Design Specifications	Good Architecture Specifications
Good Implementation Specification	Access to Use Model(s) information
Control Designs	Data Movement Designs
General (multiple) application, used by many customers	Single application, used specifically by one or a few customers

Often a combination of top down and bottom up can be used. You can start with a top down and map out the main flow which naturally brings out categories and then do bottom up on each of the categories. It is wise to do this at the beginning of the project; as soon as some form of design specifications are ready. Get started by extracting a few hundred requirements, put them into a spreadsheet and then add more later as the project progresses. Some teams link each requirement to a coverage element right away as each requirement is extracted and refined. Others, enter in all the requirements into the spreadsheet, and then take a second pass to add the coverage linking later on. Neither way is better than the other, the important thing is to get the coverage linking done while the particular requirement(s) details are still fresh in your mind. To leave the links till later in the project will mean that you have to revisit each requirement and its associated documentation all over again, which will take longer.

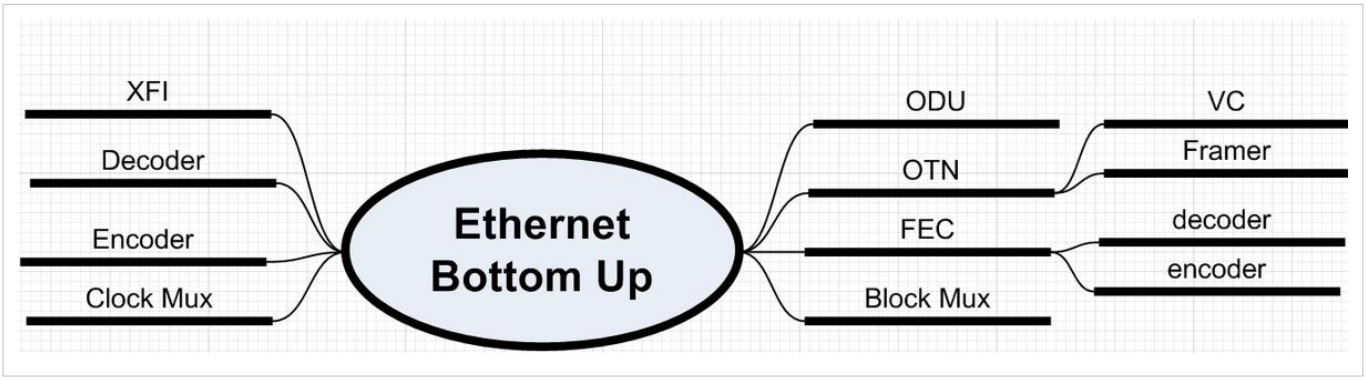
Bottom Up Example

Below is a block diagram of a Ethernet Chip with an TX and RX path. Each path has a pipeline of blocks that the Ethernet frames pass through. Some of these blocks can be muxed in or muxed out for various configurations. Also there are various clocking configurations and each block has its own configuration setup details. With a bottom up approach we would go through each block's design specification and extract out the requirements for that block. We would also go through the global block and clock mux settings and extract out the requirements for each of those. The key is to divide up the work into small, digestible blocks or sub-blocks, so that the detailed requirements and behaviors can be easily extracted in a reasonable amount of time.



The first thing you need to do to start the bottom up approach is to gather as many people who know the design as possible, architects, designers, verification team, experts on various interfaces, etc. Next, a team of people need to sub-divide up the work into some logical, manageable size. This can be done my making a brainstorming diagram, also called a mindmaps. Microsoft Visio and similar software enable easy capture of these types of diagrams on-the-fly, as the

team brainstorms together. Each topic or sub-block can be broken down further and further as needed and they all are correlated in the brainstorming diagram. A simple example for the Ethernet chip is shown in the brainstorming diagram below. For more complicated designs, the brainstorming diagram would have many more sub categories branching off of each block to divide up the requirement extraction work into manageable amounts. Each branch in the brainstorming diagram might end up being a corresponding a category or subcategory in the Ethernet testplan, or if large, might be its own hierarchical spreadsheet. Some of the mindmapping software can take these brainstorming diagrams and export the information into a spreadsheet with section numbers for each category and subcategory. This gives a great starting point and a ready framework for your testplan.



The brainstorming diagram is a great first start. Each grouping or branch can then be broken out and a testplan creation meeting(s) held to flesh out the requirements for that particular topic. At each meeting gather all available design and implementation specifications, as well as any industry specification for that block or topic so they can be consulted.

Once you have a topic you can use the yellow sticky method [1], where you give post-it notes to a team who take 20 min to extract out requirements onto yellow stickies and then stick them all up on a white board for grouping into further categories. Rules and features are extracted out into detailed requirements and then each entered as a row into a spreadsheet with a title, and a brief description that describes the essence of that requirement. See the section on the do's and don'ts of requirements writing below.

Adding some sort of unique, alpha numeric requirements tag number to each requirement is a good idea, especially if you do have requirements written at multiple levels. The tags can then be used to link higher level requirements to lower level requirements and vice versa. Requirements tracing tools, like ReqTracer, can be used to further regiment the requirement tag naming and help by automating the tracking of all your requirements. Another good idea is to to add other useful information that would be helpful to guide further work with each requirement. This extra useful information might be the location in the spec that the requirement came from, the author, notes, priority, estimated effort, questions to answer later, etc. Finally, each requirement needs to be linked to some specific closure element, like a covergroup, coverpoint, cross, assertion, test, etc. A second pass on each requirement where each is refined, and prioritized is a good idea. See the testplan format page for a description and example of the recommended format.

The apb monitor, uart and datapath examples in the coverage cookbook use a bottom up planning approach.

[1] The Yellow Sticky Method is described in more detail in the book - Verification Plans: The Five-Day Verification Strategy for Modern Hardware Verification Languages by Peet James, Springer 2003.

Guidelines for writing requirements are available in the Requirements Writing Guidelines article. It is a good idea for the verification team to compile a list such as this before starting the planning process and to divide them up into rules (must be followed) and suggestions (good ideas). In effect, this is defining the requirements for writing requirements.

Top Down Example

A top down approach could be used on the same generic Ethernet design that was discussed in the bottom down approach. Instead of going block by block, we develop and follow the flow of how the design will actually be used a real application. We follow the use model(s) of the chip, or what is sometimes called the "Day in the life" of the chip. The power is applied. What happens first? Next? etc. For this Ethernet chip the high level use model has this flow:

1. Setup/Configuration

- Block mux configuration

- Clock mux configuration

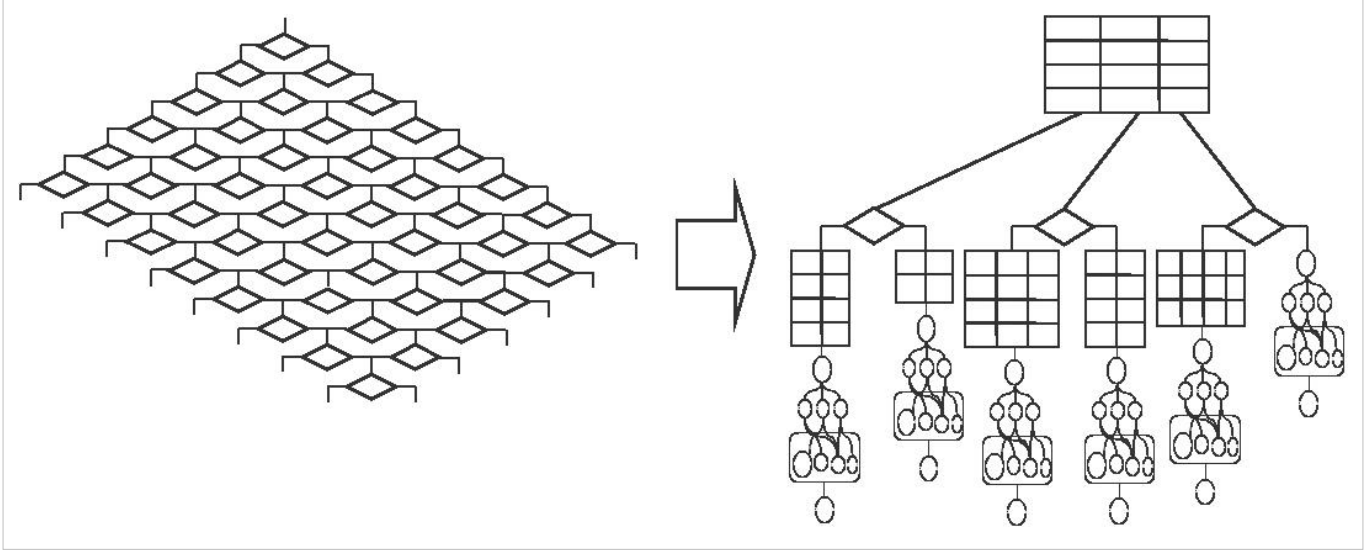
- Each block is then configured

2. Traffic

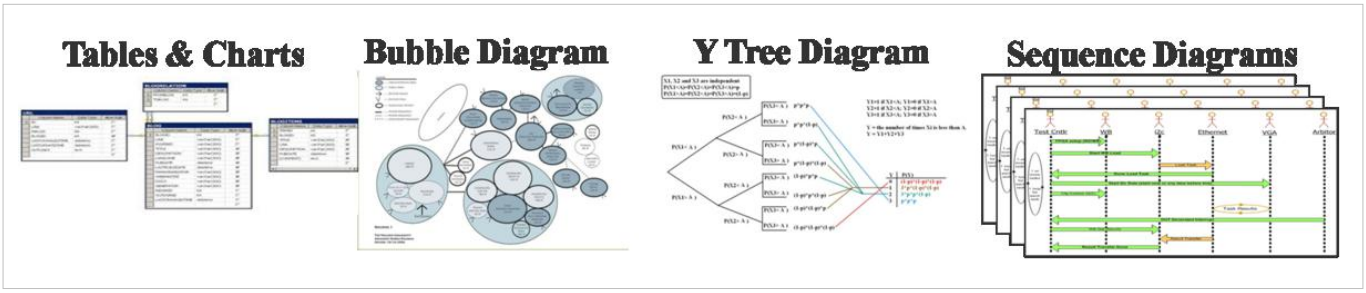
3. Unexpected event (LOS, error, etc)

Next, you expand on this basic high level flow, expanding out more details. You can do this line by line, or by path or mode. For instance, maybe certain block and clock combinations are named modes, so you can go into more detail by mode. You could also just follow a path, like system to line and line to system.

A common problem of this approach is that even with a design like this Ethernet pipeline, with its simple flow, the requirements can easily explode exponentially into what seems like too many combinations. This is common, so the explosion needs to be reworked into some logical breakdown as shown in this diagram:



When you look at the two parts of the above diagram the left exponential one looks like one huge uncloseable covergroup, while the one on the right you can see covergroups and coverpoints naturally fallout from each table or diagram. So you take each part of the high level use model flow and you expand out each one using whatever table or diagram that is useful to contain that particular sections exponential nature. For instance, in the above block muxing section of the Setup/Configuration you might develop a table of the potential useful setups and name each one. In other cases a Y-tree, Sequence, Bubble diagram or some other chart would be more useful. Often it is a good idea to gather the high level use model flow and all these diagrams into a new use model document, intermixed with minimal words.



Use a table, chart or diagram that best holds the exponential nature of each area of the use model:

- Tables are good for small space, like a few bits of a register field, or a list of behaviors.
- Bubble diagrams are good to show relationships between tasks or items, like the power areas and their settings.
- Y tree diagrams are good for showing choices and decisions, ANDs & ORs, priorities.
- Sequence diagrams show progression, cause & effect, handshaking
- You can always combine diagrams together, like the group of tables above, connected by lines.

See the WB SOC design example for use models of how these diagrams are used in a coverage context.

Once you have broken out your use model(s) into a progressive collection of useful diagrams and tables, it is a good idea to put them all in one document for easy viewing and dissemination. Some teams combine them into one big diagram; others put them together in a presentation with descriptive informational slides between the diagrams. Other formats for these diagrams include documents (separate or added as a chapter in the design architecture or implementation specifications) or as html files for a project website. The presentation format is the most common, and most useful. The collection document can go by many names for example:

- UMD: Use model document
- DITL: Day in the life document
- CAD: Coverage Architecture Document

Whatever you call the document, this document typically is very useful for introducing a new team member to the design to give them a clear overview. The team often will revert back to this document and these diagrams to flesh out more details as the verification project progresses.

Once you have a UMD, your verification team can take it and use it as a guide to write a testplan. They can comb through it and extract out the requirements and put them in the testplan. They can take each diagram, chart, and table and make it a section or sub-section in the spreadsheet, or if large, break it out into its own hierachical spreadsheet. The key is to divide up the categories and sub-categories so that each spreadsheet row is for a single requirement and can be usefully to be linked to some coverage element. Another key is to write each requirement at about the same level. Each bubble in a bubble diagram might be a single requirement or an entire subsection of requirements. Each choice on a Y-tree diagram might be a single requirement or more. Each table can be a coverage group, each row or column, a coverpoint.

The extraction of the requirements from the UMD often follows the same bottom up extraction process of described above. The UMD usually makes it easier, because of the inherent flow of the UMD and its diagrams. With practice, the verification team will start to visualize cover groups and coverpoints more readily, simply by looking at all the diagrams in their UMD. Just like with the bottom up approach adding the link and type to a coverage group, coverpoint, cross, assertion or test is best done as you write the requirement.

See the Wishbone SOC example section for more details on how to take the UMD content and create a testplan spreadsheet.

Testplan Review

The verification process has many important aspects that request time and effort of the verification team. The building of the testbench, the running of tests, the schedule, etc., all too often take precedence over the coverage model testplan spreadsheet and its development is deferred. Often, a preliminary testplan is created but the links to actual functional coverage elements are left out. The results are poor coverage implementation and minimal coverage results. The team ends up verifying in the dark, letting random generation occur, but not using coverage as a feedback to guide the testing to any conclusion or closure. They tape out with a "good enough" approach to coverage that is not based on any real coverage metric data. Having a good testplan with well defined requirements that are each linked to real coverage elements links is key. Taking the time to make this testplan will pay off in the long run. Adding the links as the requirements are written is the best approach. It also ensures that the team does not have to revisit all the documentation that inspired each requirement. To avoid this problem, mature verification teams implement a testplan review process modeled after good document or code review processes. A three stage process generally works well:

1. PRELIMINARY REVIEW: A testplan is made early on and the first review is done early as well. It is a quick review, to make sure the testplan was created, has coverage linking and type, and is on the right track. It does not need to be perfect, but be the best that can be done at the time. It will evolve over the course of the project.
2. MAIN REVIEW: About two-thirds way through a project, the real review occurs. The testplan is the coverage model which defines a prioritized subset of design behavior and intent. The goal here is to make sure the priorities and the chosen subset is correct. You can't cover everything. You can't verify everything. The team must choose their subset and do the most verification and coverage in the allotted time. This review will take some time, often 2-5 days. The testplan is reviewed in detail, making sure each row's requirement is clear and is being met with the coverage linking. All issues are addressed, and entered into a bug tracking tool. Often some form of reorganization of requirements is needed to bring the testplan up to date. It might need additions to accommodate missing content or design changes, but often it must be reduced so it can be realistically accomplished in the remaining time scheduled. Often reprioritizations occur, and some work is moved to a future tape out. The goal of the review is to find and fix any major problems or missing parts in the coverage model testplan spreadsheet.
3. FINAL REVIEW: This review is done in the final weeks of the project and if the other two reviews were done well, it is a final confirmation that the plan is valid. All big issues should have already been found and dealt with. In the final review exception details are added and any final concerns addressed before the testplan is closed.

This testplan review process is often combined with a similar three step code review process in which the rtl and testbench code are reviewed.

Executable Testplan Format

A testplan is a document which captures the important features of a design and how they will be verified.

Motivation for Creating a Testplan

Functional Verification has been described as a major challenge in SoC Designs with many citing inadequate visibility into the verification process as a major factor contributing to this challenge.
This lack of visibility impacts design quality, schedule predictability and cost (resource/tools/infrastructure).
Typical questions that verification managers need answers to are: When are we done with verification? Are all critical requirements of the system verified? Are we using all verification resources adequately to meet project schedules?

By putting an executable plan in place that captures a prioritized list of our verification intent, we are able to do the following:

- Organize ourselves better. It is not possible to completely verify all states of a design however we can identify all critical, important and nice to have features,
  then put a plan in place where we ensure that all critical and some important features are fully verified within schedule. If schedule permits, other features can be verified as well
- During Day to Day execution of the Verification project, an executable plan can help give you insight into current status. Progress can now be tracked
  instantly because you can see exactly where you are relative to what you intended to verify. You can visualize this progress against Milestones, priority of feature, or even resource
- Helps manage function coverage closure as the project is being executed. Verification Engineers can focus their coverage closure efforts on the critical features
  that have been identified as holes, followed by important features and then by the nice to have features.

Creating a Testplan

In many cases, the features will be verified in simulation and recorded as verified using either code coverage or functional coverage. The testplan can also include information about lab validation and firmware/hardware integration testing. For testplans which include code coverage and functional coverage, the connection between the testplan and simulations can be automated. To make the testplan executable a certain document format must be followed. The format which Questa's Verification Management solution uses is described below.

Capturing the Testplan

Flexibility is key when it comes to connecting a testplan to a simulator. In Questa Verification Management's case, several different source document formats are valid. The only requirement is that the source document must be able to be exported to an XML document. For example, Microsoft Word, Microsoft Excel, OpenOffice Writer, OpenOffice Calc and Adobe FrameMaker can all easily output an XML document which can be processed and made executable. In fact, it is perfectly acceptable to capture one section of your testplan in Word and another in Excel. These different sections can then be combined together as described below in the Re-Using Existing Testplans section.

Of the different input options, spreadsheets are the most commonly used and lend themselves very nicely to the type of data which is captured in a testplan. The spreadsheet format provides a clean and concise means of visualizing

verification intent. The rest of this article will describe both required information needed in a spreadsheet and how to flexibly add additional information for usage throughout the testplan's life cycle.

Contents of the Testplan

Capturing the requirements for testing into the plan is a process. That process is defined in the Specification to Testplan article. The process involves rigorous analysis of specifications, collaboration between architecture, design and verification teams and reviews to ensure nothing is missed. All of the requirements which come out of this process need to be captured in the testplan with some required structure. The required structure allows the testplan to become executable and to become part of the verification cycle.

Required Plan Structure

Questa's Verification Management solution requires four distinct pieces of information for each requirement captured. They are Section, Title, Link and Type. Additionally, other information is understood out of the box. The additional information for each requirement includes a Description, a Weight and a Goal. While these additional fields are not required, they are used the majority of the time. Questa's Verification Management solution also has the flexibilty to allow user defined fields.

If we look at the typically used fields in a spreadsheet format, each field is represented by a column in the spreadsheet.

Section	Title	Description	Link	Type	Weight	Goal
1	Parent_1					
1.1	Child_1					
2	Parent_2					

Plan Structure

Each row in the spreadsheet corresponds to a requirement captured during the testplan creation process. Each column has specific meaning in Questa's Verification Management solution.

Section and Title

The **Section** and **Title** columns work in conjunction to create the naming and hierarchy within the testplan.

Section

The Section column, usually a number, is used to create hierarchy within the testplan and group related testplan items together in a parent / child relationship. In spreadsheets, the user is responsible for entering this information. Typically you will start numbering sections with the number '1' and continuing sequentially. A sub-section beneath that section would then be numbered "1.1" and so on, where each additional level of hierarchy would be represented by the addition of a "." between section numbers.

Title

The title column captures the name of the requirement or design feature to be verified. This is the name of the testplan section that will appear within Questa. The name chosen here should have meaning as it will be visible through the tool flow.

When the testplan is extracted by Questa, it uses the information in the **Section** and **Title** columns to create a hierarchical name and unique tag for each row of the testplan. For example, given the simple testplan example above, **Section 1.1** would have a hierarchical name **/testplan/Parent_1/Child_1**.

Description

The description column allows for more detail to be added to the spreadsheet. This could include references to other documentation to allow engineers to gather more information or it could be a simple explanation as to why the requirement exists. Any text can be captured in this column. It is technically optional, but in practice a requirement captured in a testplan should have an entry in the description column.

Link and Type

The **Link** and **Type** columns are used to specify the code or functional coverage items that will be linked to the requirement. A requirement can be linked to multiple different coverage metrics, including metrics of different types. Questa supports linking to Covergroups, Coverpoints, Crosses, Assertions, Cover Directives, Directed tests and code coverage metric types. These columns also allow other testplans to be imported as described in the Re-Using Existing Testplans section.

Link

This column is where you specify the name of the actual coverage object(s) in the coverage database that is linked to this respective testplan item. This could include a specific covergroup instance, an assertion, etc.

Type

This column is where you specify the type of the coverage object you specified in the **Link** column.

Together, the **Link** and **Type** column information is used to find the corresponding coverage objects in the coverage database efficiently and create the links between the testplan and the specified coverage objects. These columns enable the testplan to become executable.

Weight

The weight column captures an integer number that reflects the relative importance of the current testplan item amongst its siblings, to its parent testplan section. The default is 1 if not specified. When coverage for the testplan is being calculated by Questa, which uses a "weighted-average" calculation algorithm, these weights are taken into account. For more information about how Questa calculates tesplan coverage please see Questa documentation on Verification Management.

Additionally, the weight column can be used to exclude portions of a testplan by specifying a value of 0 for the testplan section / item row that need to be excluded.

Goal

This column specifies the verification objective for a particular testplan section. Legal values range from 1 to 100, with the default being 100 if not specified. Questa uses this information to determine the point at which a testplan section / item is deemed to be covered. It does not alter how coverage is calculated.

Other Columns Understood By Questa

There are other special purpose columns which can be used in a coverage testplan. These columns are all optional however when they are present Questa leverages their content.

Path

When linking to a specific instance of a covergroup, two options exist. Optionally, the full hierarchical path to the covergroup could be added into the **Link** column. This works very well if only one coverge item is being accessed at that level of design hierarchy. If however, multiple coverage items are being linked to in the same level of design

hierarchy, the **Path** column allows for the specification of the design path which will be prepended to the entry in the link column to create a fully qualified reference.

Unimplemented

As testplans are being defined, it is common for requirements to be captured where corresponding coverage items don't yet exist in a testbench or design. To handle this situation, a requirement can be maked as unimplemented by either adding a value of 'yes' or a number greater than zero to the **Unimplmented** column. This will cause testplan coverage calculations to accurately reflect that a requirement exists which is not yet covered by showing zero coverage for that requirement. By default, it is assumed that coverage for a requirement is implemented unless this column is specified.

User Defined Columns

To ensure users have the flexibility to record other vital information related to a requirement, Questa's Verification Management solution allows user defined columns to be created as well. This allows for the specification of anything that needs to be tracked. It can also give you better visibility into the verification process. Some columns which are routinely added include which engineer is responsible for testing a requirement and what the priority of the requirement or design feature is.

These added columns help to answer questions such as:

- How am I doing on my high priority testplan items?
- Where am I with regards to meeting my current project milestones?
- How are my doing in terms of my resources?

The spreadsheet used in the System Level Functional Coverage Example adds **Owner** and **Priority** user defined columns.

Re-Using Existing Testplans

Often it is useful to be able to make use of existing testplan's in other contexts. A few common situations where this becomes useful are:

- Testplans were developed at the block level and need to be incorporated into the chip or system level testplan.
- A testplan is delivered with a piece of IP or verification IP. For example, protocol specific Verification IP should contain a complete protocol testplan to ensure a protocol is completely verified.
- Incorporate auto generated testplans into our sub-system or SoC level plan. For example, for power aware simulations simulators should be able to create a testplan to ensure the power scenarios are completely
- Combine testplans captured by different editing tools into one master testplan.

In these situations, existing testplans can be hierarchically imported into the top-level testplan being created, allowing for visualization of the full depth of the verification effort.

When creating such a testplan, the **Type** column will be given a value of *"XML"* to inform Questa that this section does not reference a coverage object, but rather it refers to another testplan document. The **Link** column will be used to specify the actual testplan file that contains the testplan being imported (via either a relative or full-path to the file).

Testplan to functional coverage

Arriving at functional coverage closure is a process that starts with the functional specification for the design, which is analyzed to determine:

- What features need to be tested
- Under what conditions the features need to be tested
- What testbench infrastructure is required to drive and monitor the design's interfaces
- How the testbench will check that the features work

Deriving a functional coverage model is not an automatic process, it requires interpretation of the available specifications and the implementation of the model requires careful thought.

The Process

The process that results in a functional coverage model is usually iterative and the model is built up over time as each part of the testbench and stimulus is constructed. Each iteration starts with the relevant and available functional specification documents which are analyzed in order to identify features that need to be checked by some combination of configuration and stimulus generation within the testbench.

In general terms, a testbench has two sides to it, a control path used to stimulate the design under test to get it into different states to allow its features to be checked; and an analysis side which is used to observe what the design does in response to the stimulus. A self-checking mechanism should be implemented in the testbench to ensure that design is behaving correctly, this is usually referred to as the scoreboard. The role of the functional coverage model is to ensure that the tests that the DUT passes have checked the design features for all of the relevant conditions. The functional coverage model should be based on observations of how the design behaves rather than how it has been asked to behave and should therefore be implemented in the analysis path. The easiest way to think about this is that with a testbench, the stimulus that runs on it and the scoreboard(s) have to be designed to test all the features of a design, and that the functional coverage model is used to ensure that all the desired variations of those tests have been seen to complete successfully.

Verification is an incomplete process, even for "simple" designs it can be difficult to verify everything in time available. For reasonable sized designs there is a trade-off between what could be verified and the time available to implement, run, and debug test cases, this leads to prioritization based on the technical and commercial background to the project. A wise verification strategy is to start with the highest priority items and work down the priority order, whilst being prepared to re-prioritize the list as the project progresses. The functional coverage model should evolve as each design feature is tested, and each additional part of the functional coverage model should be put in place before the stimulus.

Process Guidelines

The functional coverage model is based on functional requirements

The testbench is designed to test the features of the design. The role of the functional coverage model is to check that the different variants of those features have been observed to work correctly. Features may also be referred to as requirements or in some situations as stories.

For instance - a DUT generates a data packet with a CRC field. The CRC is based on the contents of the packet which has, say 10 variants. The testbench generates stimulus that makes the DUT produce the data packets and the scoreboard checks the CRC field to make sure that the DUT has calculated it correctly. The role of the functional coverage monitor in this case is to ensure that all 10 packet variants are checked out.

Choose the most appropriate implementation

When deciding how to implement a functional coverage model, there is a choice as to how the functional coverage will be implemented. The SystemVerilog language supports two main types of functional coverage:

Coverage Type	Implemented As	Used for
Covergroup Modeling	SystemVerilog Covergroups	Checking permutations of condition and state when a known result is achieved
Cover Property Modeling	SystemVerilog Assertions - sequences and properties	Checking that a set of state transitions has been observed

Covergroup functional coverage relies on sampling the value of one or more data fields to count how many times different permutations of those values occur.

Cover Property or temporal based coverage is based on counting how many times a particular sequence of states and/or conditions occurred during a test. Temporal coverage is usually used to get coverage on control paths or protocols where timing relationships may vary. Examples include:

- Whether a FIFO has been driven into an overflow or underflow condition
- Whether a particular type of bus cycle has been observed to complete

The first step in developing a functional coverage model is deciding which of these two approaches should be taken for each of the areas of concern.

Functional Coverage should be based on observation

The stimulus side of the testbench should be used to drive the DUT, the stimulus side should not be used for coverage because the DUT or the stimulus may not work correctly resulting in false coverage. Instead the functional coverage should be based on what is observed to happen at the DUT outputs in the testbench. In a UVM testbench, the functional coverage would be based on the content of analysis transactions. This has implications on the design of the testbench and the analysis transactions.

Functional Coverage Validity

Functional coverage data is only valid when a check passes. If you are not using automatic tools to merge coverage from different simulations, then you should ensure that the coverage model is only sampled when a test passes.

However, if you are using verification management tools to collect coverage from multiple simulations, then there is no need to do this since the coverage results from tests that fail would not be merged with the rest of the functional coverage data.

Collect coverage on negative testing separately from positive checking

If the verification plan calls for positive and negative testing of a feature, design and collect the functional coverage separately for both types of testing.

Design the coverage model for analysis

A common mistake is to develop a thorough functional coverage model only to realize later that it is difficult to understand the results. Use the available language constructs to make it easier to understand where the missing coverage is. In other words, it is worth investing some time when creating the functional coverage model to save time later when you need to understand what the results are telling you. See the Coverage Design For Analysis page for more discussion about this topic.

Determining the appropriate level of fidelity

How close a match to the detail of the design implementation is required? How much abstraction can be applied?

In simple cases where there are relatively few values to check creating a coverage model which is very closely related to the design detail is a manageable problem. However, when there is a wide range of values, careful decisions need to be made concerning the level of detail. For instance, a device configuration with about 20 possible values can easily be modeled directly, whereas something like a 32 bit address range needs to be split in to a set of interesting values or value ranges which requires some abstraction.

Use a consistent coding style

Using a consistent coding style for implementing covergroups and assertion code is important since it makes them easier to understand and interpret. It also makes integration with verification planning and tracking tools easier.

See the SystemVerilog coding guidelines for an example covergroup coding style. Note that the actual style used is unimportant, but using a consistent style is important.

Covergroup based coverage considerations

When designing covergroup based coverage there are a number of key points to consider based on the functionality and characteristics of the design

Which values are important?

It is easy to collect information in an uninformed way by simply specifying that a variable be sampled by a coverpoint, and this will result in the automatic generation of 2**n coverage bins where n is the number of bits in the variable. For narrow width fields this may be OK, but for wider fields this is unlikely to be very useful. In most cases there will be a number of values or ranges of values that will be important to cover, these should be derived from the specification and

coded into the coverpoint as a set of bins.

What are the dependencies between data variables?

When analyzing the different ways in which a design might be configured or a packet might be constructed, are there relationships between different variables? If such a relationship does exist, then a cross product between the variables should be specified in the covergroup.

Are there boundary conditions that should be checked?

Are there particular values or combinations of values that should be checked because they are at the limits of operation or are at a known inflection point in the design? This will invariably require some "reading between the lines" of the specification and some design or implementation knowledge. Any boundary conditions identified should be added to the coverage model to ensure that they are tested.

Are there illegal conditions?

If there are conditions which should not occur, then the covergroup can have a term to trap those conditions. The term does not contribute to the functional coverage but it can help detect either a design or a testbench error.

Are there conditions that are not important?

Even the simplest of designs may have more ways of configuring it than are ever realistically going to be used. If there is a way to determine which modes are most likely to be used then it also likely that there will be some that are known to be either useless or very unlikely to used, these can be omitted from the permutations of coverage values collected. There may also be a degree of prioritization here, with certain configurations that have to be tested, then later on, if there is time, lower priority configurations can be checked.

When is the right time to sample the coverage?

The data coverage collection code needs to sample the data values it is referencing. The sample point needs to:

- Only occur when the associated check has passed
- Occur when the data values are valid
- Occur when the data values are stable

If the sampling is based on receiving a UVM analysis transaction, then if it comes directly from a monitor it may need to have a means to discriminate between valid and invalid analysis traffic. If the functional coverage collector is fed analysis transactions from a scoreboard then the scoreboard should qualify that a check has passed before sending the analysis transaction.

Are there times when the data coverage sample is not valid?

If there are, then guards will have to be coded into the functional coverage implementation code.

What information is required in the analysis transaction?

For testbenches based on a TLM methodology, such as OVM or UVM, the information required by the functional coverage needs to come from the analysis transaction. This implies that the analysis transaction has to have all the information that is going to be sampled, and this may well affect the transaction and the design of the monitor or scoreboard that is generating the transaction.

Summary

When considering how a design feature is to be tested, and what the covergroup based functional coverage model for that feature should be, remember to answer these questions:

Coverage Criterion	Feature on which data coverage is to be collected
Which values are important?	Identify the important values to hit.
What are the dependencies between the values?	Identify the important cross products between data values
Are there illegal conditions?	Identify values, or combinations of values that should not occur
When is the right time to sample?	Specify a valid sampling point
When is the data invalid?	Identify conditions when the data should not be sampled

Temporal coverage considerations

Control paths and protocols rely on timing relationships between signals to implement handshakes and transfers. These types of relationships are easiest to verify using temporal sequences and properties. Functional coverage information can be derived from these checks in several ways:

- If a property is asserted and never fails, but can be shown to complete then it can be assumed that the functionality described in the property has been covered
- If a sequence completes, then the functional behavior encoded in that sequence has been observed and functional coverage can be assumed
- If a sequence completes, then a covergroup can be sampled to check under what conditions it completed (Hybrid coverage)
- If a property assertion passes, then the pass statement can be used to trigger sampling of a covergroup to check under what conditions the pass occurred (Hybrid coverage)

Checks should be specific

A property or a sequence should only check for one condition, having multiple conditions checked in a property or a sequence leads to aliasing in functional coverage.

If a sequence *and* or *or* fusion operator is used to define a property, then to collect specific functional coverage on each potential path through the property the property will have to be un-rolled and implemented as a sequence specific to each of the paths.

Hybrid Coverage

There may be times when a hybrid of data coverage and temporal coverage techniques is required to collect specific types of functional coverage. For example, checking that all modes of protocol transfer have occured is best done by writing a property or sequence that identifies when the transfer has completed successfully and then sampling a covergroup based on the interesting signal fields of the protocol to check that all relevant conditions are seen to have occurred.

The APB Bus protocol monitor contains an example implementation of using hybrid functional coverage.

Functional Coverage Examples

Different styles of design require alternative approaches to building the functional coverage model. Here are some worked examples that illustrate some common design categories:

Design Type	Covergroups	Assertions	Functional Coverage modeling strategy	Link to example	Example Tarball
Control based designs	Maybe	Yes	In this style of design there are timing relationships between different signals which need to be checked and seen to work	APB Bus Protocol Example	**(download source code examples online at** [1] **).**
Peripheral style design, programmed via registers	Yes	Maybe	Most of the functional coverage can be derived from content of the registers which are used to control and monitor the behaviour of the device. The register interface may also serve the data path. There may be scope for using assertions on signal interfaces.	UART Coverage Example	**(download source code examples online at** [1] **).**
DSP datapath style design	Yes	No	In this class of design, the stimulus pumps data through the design datapath and compares the output against a reference model. The functional coverage is primarily about ensuring that the algorithm 'knobs' have been tested sufficiently.	Biquad Filter Example	**(download source code examples online at** [1] **).**
Aggregator/Controller style, e.g. Memory Controller	Yes	Yes	Coverage of combinations of abstract stimulus on multiple ports, coverage of Config registers, coverage of features of target DDR specification	To be released	
SoC with vertical reuse of UVM analysis components	Yes	Maybe	At the SoC level not functional coverage is use case driven, and only some interface or block level coverage can be reused	SoC Coverage Example	Not applicable

Coding for analysis

Taking care with the implementation of covergroups is an investment in time that can pay back when you or someone else need to understand where the missing functional coverage is.

Covergroup Labeling

The way in which you use labeling when coding a covergroup can have a huge impact on understanding the coverage results. A covergroup can be assigned a option.name string which helps with identification of which particular part of a testbench the coverage is associated with. In side a covergroup, coverpoints can be labelled and bins can be named. Using all of these techniques makes it much easier to understand the coverage results during analysis.

Covergroup naming

If multiple instances of the same covergroup are used within a testbench, then the option.name parameter can be used to assign an identity string to each instance. The name string can be passed in as an argument when the covergroup is constructed. In a UVM environment, the name could be passed in using get_full_name() method.See the following code example.

```
// Class containing a covergroup
class my_cov_mon;

  covergroup my_cg(string instance_name);
    option.per_instance = 1;
    option.name = instance_name;
  // ...
  endgroup: my_cg

  function new(string cg_inst_nmae);
    my_cg = new(cg_inst_name);
  // ...
  endfunction

endclass: my_cov_mon

// UVM Component containing a covergroup
class my_cov_mon extends uvm_subscriber #(my_txn);

  covergroup my_cg(string instance_name);
    option.per_instance = 1;
    option.name = instance_name;
  // ...
  endgroup: my_cg

  function new(string name = "my_cov_mon", uvm_component parent = null);
```

```
    super.new(name, parent);
    my_cg = new(this.get_full_name());  // Gets the UVM hierarchy for the component
  endfunction

endclass: my_cov_mon
```

A covergroup can also be named programatically using the covergroup set_inst_name() built-in method.

```
// UVM Covergroup based component
class my_cov_mon extends uvm_subscriber #(my_txn);

  covergroup my_cg;
  //...
  endgroup: my_cg

  function new(string name = "my_cov_mon", uvm_component parent = null);
    super.new(name, parent);
    my_cg = new();
  endfunction

  function void build_phase(uvm_phase phase);
    my_cg.set_inst_name("TLB_coverage"); // Sets the instance name
    //...
  endfunction: build_phase
endclass: my_cov_mon
```

Coverpoint and bin labeling

As an example consider the following functional coverage problem from the UART example. In this example the UART word format is determined by the contents of the Line Control Register (LCR):

LCR Bit	Value	Description
[1:0]	2'b00	5 bit data character
	2'b01	6 bit data character
	2'b10	7 bit data character
	2'b11	8 bit data character
2	1'b0	1 Stop bit
	1'b1	2 Stop bits
[5:3]	3'b??0	No Parity
	3'b001	Odd parity
	3'b011	Even parity
	3'b101	Stick 0 parity
	3'b111	Stick 1 parity

Lower Analysis Potential

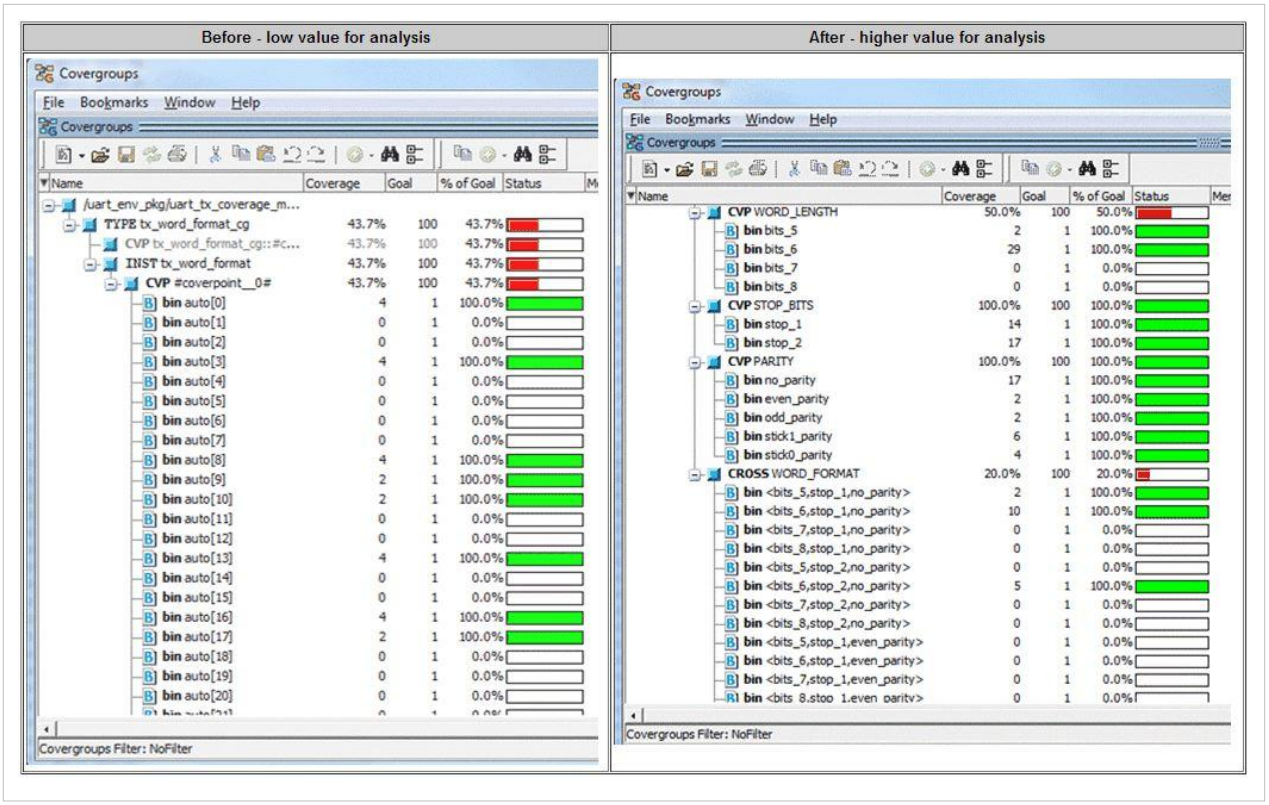
```
covergroup tx_word_format_cg()
  with function sample(bit[5:0] lcr);

  option.name = "tx_word_format";
  option.per_instance = 1;

  coverpoint LCR[5:0];

endgroup: tx_word_format_cg
```

Higher Analysis Potential

```
covergroup tx_word_format_cg
  with function sample(bit[5:0] lcr);

  option.name = "tx_word_format";
  option.per_instance = 1;

  WORD_LENGTH: coverpoint lcr[1:0] {
    bins bits_5 = {0};
    bins bits_6 = {1};
    bins bits_7 = {2};
    bins bits_8 = {3};
  }

  STOP_BITS: coverpoint lcr[2] {
    bins stop_1 = {0};
    bins stop_2 = {1};
  }

  PARITY: coverpoint lcr[5:3] {
    bins no_parity = {3'b000, 3'b010, 3'b100, 3'b110};
    bins even_parity = {3'b011};
    bins odd_parity = {3'b001};
    bins stick1_parity = {3'b101};
    bins stick0_parity = {3'b111};
  }

  WORD_FORMAT: cross WORD_LENGTH, STOP_BITS, PARITY;

endgroup: tx_word_format_cg
```

In order to check that all possible word formats have been transmitted we could implement a covergroup by creating a coverpoint for LCR[5:0] and not specifying any bins. This would create a set of default bins, one for each possible value of the register, as shown in the left hand code example. If the functional coverage collected samples these bits at least once, then there is no problem, but if not then it is reasonably difficult to figure out which bin corresponds to which condition - see the 'before' screen shot from the Questa covergroup browser. Here, not using labels has caused the simulator to use auto-bins, which means that the missing bin values need to be converted to binary and then mapped to the register fields to identify the missing configurations.

A better way to implement the covergroup is to use a labeled coverpoint for each register field and then using the bins syntax for each of the values in the register truth table. When this is simulated, the cross products created reflect the different bin labels, which makes it much easier to determine which functional coverage conditions have not been sampled. It also makes it easier to see whether there are any gross coverage conditions that have been missed. See the 'after' screen shot from the Questa covergroup GUI for the refactored covergroup.



How Covergroup options affect the reporting and computation of coverage

Implementation Options

The analysis of functional coverage information is affected by the way in which the coverage results are reported. There are three covergroup options which impact coverage reporting and can cause considerable confusion, and these are:

- option.per_instance
- option.get_inst_coverage
- type_option.merge_instances

If these options are not specified in the code that implements a covergroup, then they are not enabled by default. In other words, they are set to 0.

These three options should be explicitly declared in covergroup to ensure that the coverage computation and reporting is consistent and as required.

Covergroup types and instances

When a covergroup is declared it becomes a type that may be instantiated several times in the testbench - for instance the same type of interface is used for several ports in a design and therefore the same covergroup is used to measure protocol coverage. The default coverage reporting method is to report the coverage for the covergroup type as a weighted average of the coverage from all of the covergroups of that type. What this means is that if one of the ports has been exercised to achieve 100% coverage, but others have not, then the coverage reported will not be less than 100% and it will not be possible to analyse which interfaces have not been exercised.

per_instance option

If the covergroup option.per_instance is set to 1, then the covergroup reporting is broken out per instance, but the overall coverage reported is still the weighted average. In the example quoted, this would enable the coverage for each port to be examined, possibly leading to a detection of a design bug or a short-coming in the stimulus generation.

merge_instances option

If the covergroup type_option.merge_instances is set to 1, then the overall coverage reported for all the instances of the covergroup is a merge, or logical OR, of all the coverage rather than a weighted average. This is potentially useful if you have multiple instances of the same design IP and and it is being exercised in different ways by different parts of the testbench. One outcome from using the merge_instances option is that one covergroup instance achieves 100% coverage masking another instance that achieves 0% coverage, since the overall coverage will be reported as 100%.

get_inst_coverage option

To help with the scenario where the merge_instances option has been enabled, the option.get_inst_coverage variable can be set to 1 to enable the SystemVerilog $get_inst_coverage() system call to return the coverage for an instance of a covergroup, therefore allowing the coverage for all individual instances to be checked. If the merge_instances option is set to 0, then the get_inst_coverage variable has no effect.

Summary

Interaction between per_instance and merge_instances settings:

option.per_instance	type_option.merge_instances	Coverage reporting behaviour
0	0	Overall coverage reported as a weighted average of the coverage for all instances of the covergroup
1	0	Overall coverage reported as a weighted average of the coverage for all instances of the covergroup, and broken out for each instance of the covergroup.
0	1	Overall coverage reported as a merge of the coverage for the individual instances of the covergroup
1	1	Overall coverage reported as a merge of individual coverage results, get_inst_coverage() enabled, coverage reporting broken out for each instance of the covergroup

Interaction between merge_instances and get_inst_coverage

type_option.merge_instances	option.get_inst_coverage	Data returned by $get_inst_coverage()
1	0	Type based coverage - i.e. merge of all instance coverage
1	1	Instance specific coverage
0	1|0	Instance specific coverage

Simulator Specific Run Time Options

The options described are covergroup options defined in the SystemVerilog LRM and can therefore be added to covergroup code. Simulators also add command line options which allow users to change the way in which coverage is reported, although these options tend to be global affecting all covergroups within the testbench being simulated.

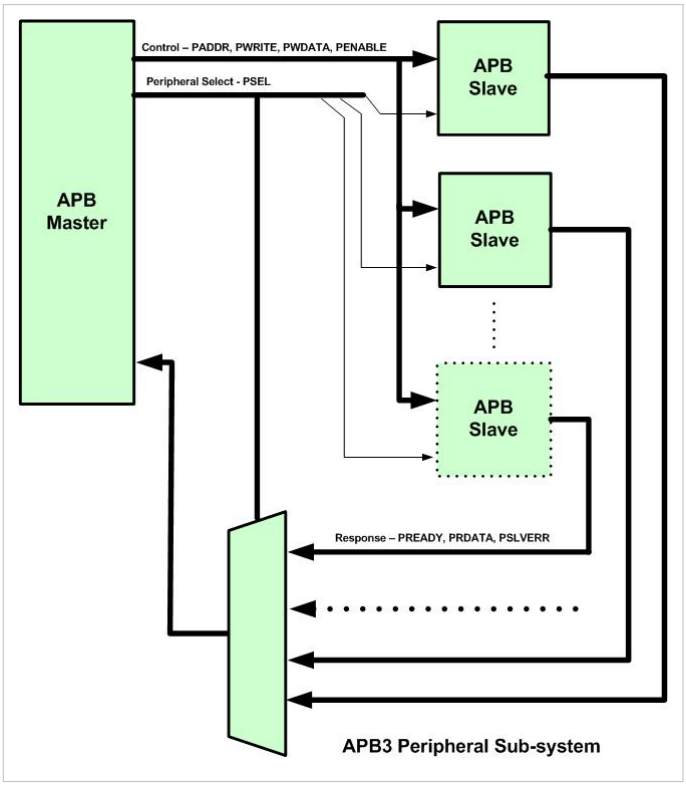
Coverage Examples (Practice)

Bus protocol coverage

A bus protocol defines how data is transfered between master and slave devices on the bus. The protocol specifies which signals are used to qualify when the master is making a request and when the slave is ready to respond to that request. One way to verify a bus protocol is to implement a protocol monitor as a SystemVerilog interface that uses a mixture of SVA concurrent assertions and covergroups to check that the bus traffic it observes complies to the protocol and to collect functional coverage information on what types of transfers have occured over the bus. A protocol monitor is a passive verification component which monitors the protocol signals, it can be attached to external signals in the top level of a testbench or it could be bound to internal bus signals inside a design. A bus protocol monitor is a good example of a reuseable verification component.

APB3 Protocol Overview

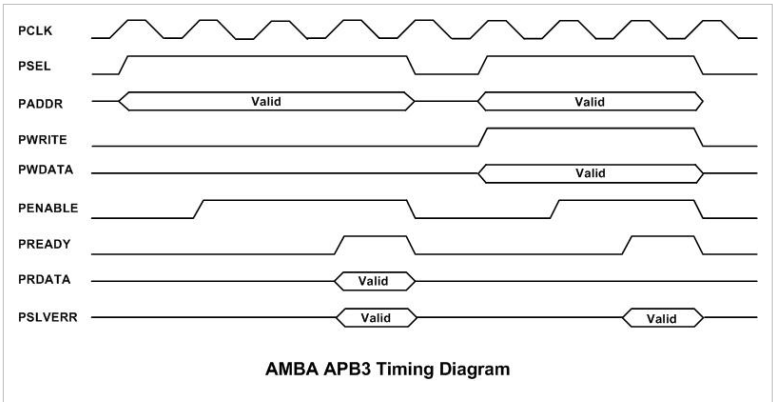
The ARM AMBA APB3 protocol is an example of a simple peripheral bus protocol which can be used to illustrate the main points associated with implementing a protocol monitor and showing how it can be used to collect functional coverage information.



With the APB3 protocol, a single master can interface to several slave peripheral devices. The master generates a set of control fields for address, write, and write data which are common to all the slaves. Each slave is selected by an

individual peripheral select line (PSEL) and then enabled by a common PENABLE signal. Each slave generates response signals, ready, read data and status which are multiplexed back to the master. The block diagram shows a typical APB3 peripheral block.

The timing relationship between the APB3 signals is shown in the timing diagram below.



AMBA APB3 Timing Diagram

Deriving The Protocol Properties

The easiest way to start defining properties is to write simple unambiguous descriptions in English for each of the protocol rules. Once these are defined and agreed upon then it is usually relatively straight-forward to implement the assertion code.

The properties outlined below have also been summarised as a test plan.

For the APB protocol we can define the following:

Unknown (X, Z) Signals

If a protocol signal is in an undefined state then it will cause a problem under certain conditions. Properties that check for unknown signal states are useful in the early stages of debugging a design, but are less useful at later stages in verification. However, they do illustrate the first type of functional coverage collection - a property that succeeds and never fails proves functional coverage for that property.

For the APB3 protocol the following unknown signal rules are true:

- PSEL shall always be in a known state
- If at least one PSEL line is at a logic 1, then the following control field signals shall also be in a known state:
  - PADDR
  - PENABLE
  - PWRITE
- If at least one PSEL line is at a logic 1 and PWRITE is at logic 1, then PWDATA shall be in a known state
- If PSEL and PENABLE are both at logic 1, then PREADY shall be in a known state
- If PREADY is at logic 1, then PSLVERR shall be in a known state
- If PWRITE is at logic 0 and PREADY is at logic 1, then PRDATA shall be in a known state

Another way to specify this is to require that all of the bus signals must be in a known state all of the time, however this may not always be practical and the properties defined are the minimum for the protocol to work correctly in all conditions.

See the unknown signal properties section of the example for an example implementation.

Timing Relationships

The timing relationships between the signals in the protocol can be described using sequences and properties. If a covered sequence completes or an asserted and covered property passes then functional coverage can be assumed for the function in question. For the APB3 protocol, the following temporal relationships can be defined:

- Once PREADY is sampled at logic 1, PENABLE shall go low by the next clock edge
- When a PSEL line goes to a logic 1, then the following signals shall be stable until the end of the cycle when PREADY is sampled at a logic 1
  - PSEL
  - PWRITE
  - PADDR
  - PWDATA (iff PWRITE is at logic 1)
- There shall be at least one clock cycle where PENABLE is at logic 0, between bus transfers
- When a PSEL line goes to a logic 1, then PENABLE shall go to a logic 1 on the following clock edge

See the Timing Relationships section on the example page for an implementation of these properties.

Other Properties

There may be other protocol rules which are not strictly temporal in nature. For the APB3 protocol the following property is true:

- Only one PSEL line shall be active at a logic 1 at any time

See the Other Properties section of the examples page for an implementation.

Functional Coverage

In addition to the functional coverage represented by the protocol assertions which check for valid transfers, we need to check that all possible types of transfer have occurred. This is best done by using data coverage for the various bus fields to check that we have seen transfers complete for each of the valid values. The fields that are relevant to bus protocol functional coverage are:

- PSEL - That all PSEL lines on the bus have been seen to be active - i.e. transfers occurred to all peripherals on the bus
- PWRITE - That we have seen reads and writes take place
- PSLVERR - That we have seen normal and error responses occur

Creating a cross product between these fields checks that all types of transfer have occurred between the master and each slave on the APB3 bus. See the Functional Coverage section on the examples page for an implementation.

Other types of functional coverage that could be collected would be:

- Peripheral delay - checking that a range of peripheral delays have been observed
- Peripheral address ranges - Checking that specific address ranges have been accessed

However, these are likley to be design specific and should be collected using a separate monitor.

Implementing The Protocol Monitor

Once these relationships and properties have been defined, they need to be implemented using a mixture of SVA and covergroup syntax in a SystemVerilog interface. An example protocol monitor is described here.

The source code for the protocol monitor can be downloaded here:

(download source code examples online at https://verificationacademy.com/cookbook/code-examples).

APB3 Protocol test plan

During the verification planning process a test plan should be created that lists the features of a design, how they are to be verified and how test closure will be achieved. The APB3 protocol monitor is a reusable verification component (VIP) and it can be reused when verifying any design which has an APB3 bus interface. The test plan that is listed below would be included as part of the test plan for such a design, effectively imported as a leaf in the test plan hierarchy.

APB3 Protocol Test Plan

Section	Description	Coverage Type	Priority
Unknown Signal Value Checks			
PRESETn	PRESETn is always in a known state	Assertion	1
PSEL	PSEL is always in a known state	Assertion	1
PADDR	PADDR is valid when PSEL is active	Assertion	1
PWRITE	PWRITE is valid when PSEL is active	Assertion	1
PENABLE	PENABLE is valid when PSEL is active	Assertion	1
PWDATA	PWDATA is valid when PWRITE is active	Assertion	1
PREADY	PREADY is valid when PENABLE is active	Assertion	1
PSLVERR	PSLVERR is valid when PENABLE is active	Assertion	1
PRDATA	PRDATA is valid when PREADY is asserted	Assertion	1
Timing Relationship checks			
PENABLE de-assertion	PENABLE is de-asserted once PREADY becomes active	Assertion, Cover directive	1
PSEL to PENABLE	There is only one clock delay between PSEL and PENABLE	Assertion, Cover directive	1
Signal Stability	When PSEL becomes active, the PWRITE, PADDR, and PWDATA signals should be stable to the end of the cycle	Assertion, Cover directive	1
Other Checks			
PSEL Unique	Only one PSEL line is active at a time	Assertion	1
Functional Coverage			
APB3 Protocol	All types of APB3 protocol transfers have taken place with all types of response for all active PSEL lines	Covergroup	2

APB3 Protocol Monitor

The APB3 Protocol Monitor is passive and intended to be a reuseable verification component. Therefore, it is parameterised to allow it to be used with different bus widths and all of the signals on the port interface are inputs. The monitor has been implemented as an interface with all of the signals declared as input pins, this allows it to be potentially used with a virtual interface handle in a UVM testbench, allowing the covergroup to be accessed programatically. Declaring the signals as inputs allows them to be explicitly connected to any set of APB3 bus signals without having to have an exact match in the signal naming.

```
//
// Interface declaration for the APB Protocol Monitor
//
interface apb_monitor #(int no_slaves = 16, int addr_width = 32, int data_width = 32)
  (
  input PCLK,
  input PRESETn,
  input[addr_width-1:0] PADDR,
  input[data_width-1:0] PWDATA,
  input[data_width-1:0] PRDATA,
  input[no_slaves-1:0] PSEL,
  input PWRITE,
  input PENABLE,
  input PREADY,
  input PSLVERR);
```

Unknown Signal Properties

The monitor implements and asserts the properties for unknown protocol signal conditions that were defined in English on the previous page. Since these properties are valid for a clock period there is no need to collect explicit coverage using a cover directive; if there are no assertion failures then coverage of these properties is implied.

Where possible, the approach used is to define a general purpose property which can be asserted for different signals.

```
// Checks for unknown signal values:

// Reuseable property to check that a signal is in a known state
property SIGNAL_VALID(signal);
  @(posedge PCLK)
  !$isunknown(signal);
endproperty: SIGNAL_VALID

RESET_VALID: assert property(SIGNAL_VALID(PRESETn));
PSEL_VALID: assert property(SIGNAL_VALID(PSEL));

// Reuseable property to check that if a PSEL is active, then
// the signal is in a known state
```

```
property CONTROL_SIGNAL_VALID(signal);
  @(posedge PCLK)
  $onehot(PSEL) |-> !$isunknown(signal);
endproperty: CONTROL_SIGNAL_VALID

PADDR_VALID: assert property(CONTROL_SIGNAL_VALID(PADDR));
PWRITE_VALID: assert property(CONTROL_SIGNAL_VALID(PWRITE));
PENABLE_VALID: assert property(CONTROL_SIGNAL_VALID(PENABLE));

// Check that write data is in a known state if a write
property PWDATA_SIGNAL_VALID;
  @(posedge PCLK)
  ($onehot(PSEL) && PWRITE) |-> !$isunknown(PWDATA);
endproperty: PWDATA_SIGNAL_VALID

PWDATA_VALID: assert property(PWDATA_SIGNAL_VALID);

// Check that if PENABLE is active, then the signal is in a known state
property PENABLE_SIGNAL_VALID(signal);
  @(posedge PCLK)
  $rose(PENABLE) |-> !$isunknown(signal)[*1:$] ##1 $fell(PENABLE);
endproperty: PENABLE_SIGNAL_VALID

PREADY_VALID: assert property(PENABLE_SIGNAL_VALID(PREADY));
PSLVERR_VALID: assert property(PENABLE_SIGNAL_VALID(PSLVERR));

// Check that read data is in a known state if a read
property PRDATA_SIGNAL_VALID;
  @(posedge PCLK)
  ($rose(PENABLE && !PWRITE && PREADY)) |-> !$isunknown(PRDATA)[*1:$] ##1 $fell(PENABLE);
endproperty: PRDATA_SIGNAL_VALID

PRDATA_VALID: assert property(PRDATA_SIGNAL_VALID);
```

Timing Relationships

The monitor implements the timing relationships described in English on the previous page. The functional coverage strategy is to assume that if these assertions do not fail but are seen to complete with a cover directive then they add valid functional coverage:

```
// PENABLE goes low once PREADY is sampled
property PENABLE_DEASSERTED;
  @(posedge PCLK)
  $rose(PENABLE && PREADY) |=> !PENABLE;
endproperty: PENABLE_DEASSERTED
```

```
PENABLE_DEASSERT: assert property(PENABLE_DEASSERTED);
COV_PENABLE_DEASSERT: cover property(PENABLE_DEASSERTED);

// From PSEL active to PENABLE active is 1 cycle
property PSEL_TO_PENABLE_ACTIVE;
  @(posedge PCLK)
  (!$stable(PSEL) && $onehot(PSEL)) |=> PENABLE;
endproperty: PSEL_TO_PENABLE_ACTIVE

PSEL_2_PENABLE: assert property(PSEL_TO_PENABLE_ACTIVE);
COV_PSEL_2_PENABLE: cover property(PSEL_TO_PENABLE_ACTIVE);



// FROM PSEL being active, then signal must be stable until end of cycle
property PSEL_ASSERT_SIGNAL_STABLE(signal);
  @(posedge PCLK)
  (!$stable(PSEL) && $onehot(PSEL)) |-> $stable(signal)[*1:$] ##1 $fell(PENABLE);
endproperty: PSEL_ASSERT_SIGNAL_STABLE

PSEL_STABLE: assert property(PSEL_ASSERT_SIGNAL_STABLE(PSEL));
PWRITE_STABLE: assert property(PSEL_ASSERT_SIGNAL_STABLE(PWRITE));
PADDR_STABLE: assert property(PSEL_ASSERT_SIGNAL_STABLE(PADDR));
PWDATA_STABLE: assert property(PSEL_ASSERT_SIGNAL_STABLE(PWDATA & PWRITE));
COV_PSEL_STABLE: cover property(PSEL_ASSERT_SIGNAL_STABLE(PSEL));
COV_PWRITE_STABLE: cover property(PSEL_ASSERT_SIGNAL_STABLE(PWRITE));
COV_PADDR_STABLE: cover property(PSEL_ASSERT_SIGNAL_STABLE(PADDR));
COV_PWDATA_STABLE: cover property(PSEL_ASSERT_SIGNAL_STABLE(PWDATA & PWRITE));
```

Other Properties

The monitor checks that only one PSEL line is active at a logic 1 at any point in time. Since this property is checked on every clock cycle, if there are no failures then it implies functional coverage.

```
// Check that only one PSEL line is valid at a time:
property PSEL_ONEHOT0;
  @(posedge PCLK)
  $onehot0(PSEL);
endproperty: PSEL_ONEHOT0

PSEL_ONLY_ONE: assert property(PSEL_ONEHOT0);
```

Functional Coverage

To check that we have seen transfers complete correctly for each of the possible protocol conditions for each of the peripherals on the bus, we implement an array of covergroups, one for each peripheral, which collects the protocol coverage specific to that peripheral. The covergroups are sampled when a simple sequence holds. Note that to improve performance each covergroup is only sampled when the relevant PSEL line is true.

```
// Functional Coverage for the APB transfers:
//
// Have we seen all possible PSELS activated?
// Have we seen reads/writes to all slaves?
// Have we seen good and bad PSLVERR results from all slaves?
covergroup APB_accesses_cg();

// Since we check this for each PSEL we need to set the
// per_instance flag
option.per_instance = 1;

RW: coverpoint PWRITE {
  bins read = {0};
  bins write = {1};
}
ERR: coverpoint PSLVERR {
  bins err = {1};
  bins ok = {0};
}

APB_CVR: cross RW, ERR;

endgroup: APB_accesses_cg

// Array of these covergroups
APB_accesses_cg APB_protocol_cg[no_slaves];

// Creation the covergroups
initial begin
  foreach(APB_protocol_cg[i]) begin
    APB_protocol_cg[i] = new(i);
  end
end

// Sampling of the covergroups
sequence END_OF_APB_TRANSFER;
  @(posedge PCLK)
  $rose(PENABLE & PREADY);
```

```
endsequence: END_OF_APB_TRANSFER


cover property(END_OF_APB_TRANSFER) begin
  foreach(PSEL[i]) begin
    if(PSEL[i] == 1) begin
      APB_protocol_cg[i].sample();
    end
  end
end
```

Complete Example Code

The source code for the monitor and a testbench to exercise it can be downloaded here:

(download source code examples online at https://verificationacademy.com/cookbook/code-examples).

Block Level coverage

Block level functional verification can take full advantage of the the fact that all the block interfaces are exposed and can be stimulated seperately with complete freedom. The desired outcome of the verification process is that the block has been thoroughly tested so that it can reused at a higher level of integration with complete confidence. The functional coverage model is part of the instrumentation used to determine whether the design has been verified to the desired level of quality.

The block level design example is a UART, which contains contains registers which allow the DUT to be configured for its different modes of operation, and can also be used to get status information. The UART is verified using a UVM block level testbench, which contains a UVM register model.

UART Overview

The function of an Universal Aysynchronous Receiver Transmitter (UART) is to transmit and receive characters of differing formats over a pair of serial lines asynchronously. With an asynchronous serial link, there is no shared sampling clock, instead the receive channel samples the incoming serial data stream with a clock that is 16x the data rate. When there is no data to transmit the data lines are held high, and transmission of a data character commences by taking the data line low for one bit period to transmit the start bit. The receiving end detects the start bit and then samples and unpacks the serial data stream that can consist of between 5 and 8 bits of data, parity and then a stop bit which is always a 1.

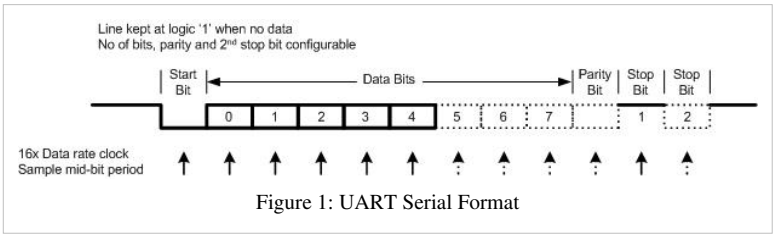


Figure 1: UART Serial Format

Register Map

The UART design in this example is based on the industry standard 16550a UART. It has 10 registers which control its operation and in a system these are used by software to control the device and to send and receive characters. The transmit and receive paths are buffered with 16 word deep FIFOs.

The register map is summarised here:

Register	Address Offset	Width	Access	Description
Receive Buffer	0x0	8	R	Receive data FIFO output
Transmit Buffer	0x0	8	W	Transmit data FIFO input
Interrupt Enable (IER)	0x4	8	R/W	Enables for UART interrupts
Interrupt Identification (IIR)	0x8	8	R	Interrupt status
FIFO Control (FCR)	0x8	8	W	Set receive data FIFO thresholds
Line Control (LCR)	0xC	8	R/W	Sets the format of the UART data word
Modem Control (MCR)	0x10	8	R/W	Used to control the modem interface outputs
Line Status (LSR)	0x14	8	R	Transmit and receive channel status
Modem Status (MSR)	0x18	8	R	Used to monitor the modem interface inputs
Divisor 1	0x1C	8	R/W	LSB of the 16 bit divider
Divisor 2	0x20	8	R/W	MSB of the 16 bit divider

For the UVM testbench, a UVM register model will be written to abstract stimulus for configuring and controlling the operation of the UART. One benefit of using this register model is that we can reference it for the functional coverage model. For more details on the UART functionality and the detailed register map, please refer to the datasheet.

External Interfaces

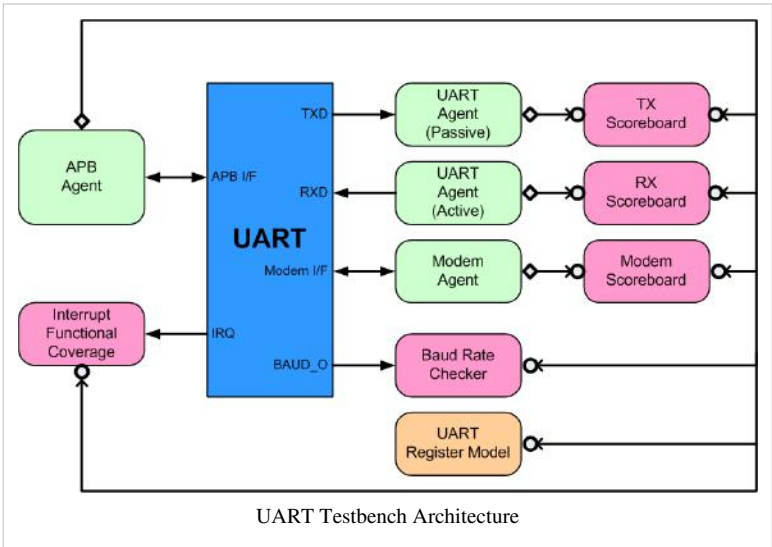
The UART block has a number of discrete interfaces which need to be driven or monitored. The UART example testbench is implemented using UVM, therefore the driving and monitoring of these interfaces will be done by Universal Verification Components (UVCs) or agents. If the testbench was implemented using another methodology, then BFM or BFM-like models would be used. However, the principles of how you model and collect coverage are essentially the same.

The UART has the following external interfaces which will need to be driven and monitored in the testbench.

- APB Host interface – Requires an APB agent
- TX Serial line – Requires a passive UART agent
- RX Serial line – Requires an active UART agent
- Modem interface – Requires a simple parallel I/O agent
- Interrupt line – Requires a monitor

Testbench Architecture

The UVM testbench architecture used for this example is shown in the block diagram.



UART Testbench Architecture

An outline functional test plan for the UART has been created as part of the process of mapping its features to test cases and functional coverage.

Designing The Functional Coverage Model

The purpose of the functional coverage model is to ensure that all the functional test cases simulated with the testbench put the design under test into all of the conditions that are considered to be interesting. The functional coverage model is based on the functionality of the design and is closely related to the way in which the testbench works since it will be relying on data coming from the testbench components and it will be relying on testbench events to sample.

The design of the functional coverage model for the UART will be discussed in terms of each area of concern within the device. In each case, the necessary tests are considered first, then the functional coverage model required in order to ensure that all conditions have been seen to be tested.

Example implementations of the covergroups for each part of the functional coverage model are described separately and are available in the example UART testbench.

The Transmit Channel

Testing the transmit channel

The transmit channel needs to be verified by checking that it transmits the correct serially formatted data for each of the possible word formats, this requires the stimulus to configure the design and then to send data by filling the transmit buffer. The UART agent will interpret the transmitted serial stream according to the configuration of the device and send out analysis transactions on reception of a character. The monitor in the UART agent will perform low level format and parity error checks on the serial data and flag errors if they are detected since it has access to the clock and data signals. If the UART is working correctly, the transmit data path should be error free, there is no scope for the testbench to introduce errors.

In order to check that the transmit channel is working correctly we can compare the content of the analysis transaction written by the passive UART monitor when a character is received with the character originally written to the transmit buffer of the UART. This implies scoreboard analysis connections to the UART agent and the APB agent. The UART transmit buffer writes will have to be buffered in a FIFO structure in the scoreboard so that they can be compared with the characters received by the UART.

The transmit channel has two buffer status bits (TX empty and TX FIFO empty) which are read back from the Line Status Register, these need to be tested by the stimulus generation path. There is also a TX FIFO empty status interrupt which will be discussed in the section on interrupts.

Functional coverage design for the transmit channel

As the testbench checks that the UART transmits serial data correctly, the role of the functional coverage model is to check that each of the word formats has been configured and seen to work correctly. The word format is determined by the contents of the Line Control Register (LCR), therefore the functional coverage model needs to sample the LCR values every time a character is transmitted. There are 6 bits in the LCR register which determine the word format and these are:

LCR Bit Field	Function
[1:0]	Selects 5,6,7 or 8 bit data
2	Selects 1 or 2 stop bits
[5:3]	Enables parity and type of parity

We need to see all possible permutations of these configuration settings in order to say that we have achieved functional coverage for the transmit channel. An example implementation of the SystemVerilog covergroup used to collect this functional coverage is implemented in the example UART testbench.

Transmit channel coverage summary

Coverage Criterion	Transmit Channel Coverage
Which values are important?	LCR[5:0] - defining all permutations of UART serial word format
What are the dependencies between the values?	No dependencies
Are there illegal conditions?	No, all permutations are valid
When is the right time to sample?	When a character has been transmitted
When is the data invalid?	N/A

The Receive Channel

Testing the receive channel

The testbench drives a serial data stream into the UART using an active UART agent. An active agent is a verification component that drives the receive serial data line based on stimulus transactions and monitors the state of the line, converting pin level activity back to transactions. The receive channel has to be able to cope with errors that might be present in the format of the incoming serial data stream, this calls for positive and negative checking of the receive channel and the UART agents driver is able to insert bit level errors and its monitor is able to detect them.

The checking mechanism used by the receive scoreboard is to compare the data sent by the UART agent with the data read from the receive buffer of the UART device. Any errors inserted by the UART agent need to be seen to be detected by the design either as bits set in the Line Status Register (LSR) or by the generation of a line status interrupt. The checks that need to be made by the testbench for the receive channel include:

- That a start bit is detected correctly
- That parity has been received correctly - if not a parity error is generated
- That at least one stop bit has been received - if not a framing error is generated
- That a data overrun condition is detected correctly
- That the data received flag works correctly
- That a break condition is detected correctly

There are a number of receive channel interrupt conditions that are considered in the section on interrupts.

Functional coverage design for the receive channel

The functional coverage model for the receive channel needs to collect functional coverage for error free conditions and then for each of the possible error or special conditions. The word format for the receive channel is configured via the LCR and so for error free character formats the same covergroup design as was developed for the transmit channel can be used, except that it would be sampled when a data character is read from the receive buffer of the UART.

For the various error conditions, a separate covergroup is used and this would cross the parity error, framing error, overrun error and break condition bits from the Line Status Register (LSR) with the word format bits in the LCR. This ensures that all possible error or special conditions have been detected with all word formats.

Example implementations of these covergroups are present in the example UART testbench.

Receive channel coverage summary

Coverage Criterion	Receive Channel Coverage
Which values are important?	LCR[5:0] - defining all permutations of the UART serial word format LSR[4:0] - Status bits for Break Interrupt (BI), Framing Error (FE), Parity Error (PE), Overrun Error (OE) and Data Received (DR)
What are the dependencies between the values?	For error free RX conditions DR and all word formats For injected error RX, cross product of LCR & LSR bits
Are there illegal conditions?	Cannot have OE with no DR valid
When is the right time to sample?	When a RX character has been received and DR is valid
When is the data invalid?	N/A

The Modem Interface

Testing the modem interface

The modem interface is a group of 4 input and 4 output signals which are used to implement hardware handshaking between two UARTs. In some UART designs these lines have a hardware interlock with the UART state machines to enable/disable transmission and reception of data, but this is not the case with this UART design. The modem outputs are driven as a result of a write to the Modem Control Register (MCR) and changes to the modem inputs are read back from

the Modem Status Register (MSR).

There is a loop-back mode which also needs to be checked. In this mode writes to the MCR are reflected back in the MSR and none of the external signals change or cause any changes. The modem scoreboard checks this functionality separately from the normal mode of operation.

The testbench contains a modem agent that has a driver for the modem inputs and a monitor which sends transactions to the modem scoreboard when any of the modem signals (inputs or outputs) change. The scoreboard also receives transactions from the APB agent's monitor so that it can keep track of UART register accesses, this allows it to know when a modem output should have changed due to a write to the MCR and when a modem input change should be reflected in a read from the MSR.

Functional coverage design for the modem interface

The modem scoreboard checks that the modem interface works correctly, and in overall terms this includes the path between the modem registers and the modem signals.

For the modem outputs, all possible output signal combinations need to be observed with or without loopback, this means that the contents of the MCR can be sampled when they are written by the testbench. The functional coverage required consists of all permutations of the signal conditions.

The modem inputs are read back from the MSR, there are four signal sample values and four signal change values which need to be seen to have been read back in all possible states. The loopback bit affects whether the MSR values are derived from the modem interface signals or the MCR signals, so this needs to be part of the coverage model. The modem input coverage model is sampled when the MSR is read, and the coverage required is a cross product of all possible values for the MSR plus the value of the MCR loopback bit.

Example implementations of these coverage models are present in the uart_modem_coverage_monitor in the example UART testbench.

Modem interface coverage summary

Coverage Criterion	Modem Interface Coverage
Which values are important?	MCR[4:0] - Controlling outputs and loopback mode MSR[7:0] - input status and changes to input values
What are the dependencies between the values?	Each of the modem signals are orthogonal, but the loopback mode creates a dependency between the MCR bits and the MSR bits. For coverage all permutations are relevant.
Are there illegal conditions?	No
When is the right time to sample?	When a change occurs on the Modem interface, or there is a write to the MCR, determined by the modem scoreboard.
When is the data invalid?	Immediately after a change in the loopback mode, handled by the scoreboard

UART Interrupts

Testing UART interrupts

The testbench contains a monitor for the UART interrupt line and some of the test cases have stimulus which enables the various interrupts and then handles the interrupt conditions as they occur. The scoreboarding within the testbench checks the validity of the interrupt conditions dependent on its source.

Interrupts can be generated by the UART for the following conditions:

- Transmit FIFO empty
- Receive data FIFO threshold reached (1, 4, 8, 14 characters)
- Receiver line status - Parity error, Framing error or Break condition
- Receiver timeout - At least one character in the FIFO, but no receive channel activity for at least 4 character times
- Modem status change

Functional coverage design for the UART interrupts

There are several areas where functional coverage needs to be collected. The first concerns checking that all possible combinations of interrupt enables and interrupt sources have been tested.

The Interrupt Enable Register (IER) is a four bit register which is used to enable the different interrupt types, when an interrupt occurs this needs to be sampled for all possible active states - i.e. at least one interrupt is enabled. If an interrupt occurs with no active enables, then this is an error condition.

The Interrupt Identification Register (IIR) prioritizes active interrupt status. The value of this register needs to be sampled when an interrupt occurs followed by a read to the IIR. All IIR values need to be observed, but which ones are observable at any point in time is determined by interrupt priority and the content of the IER. Therefore, the IIR status codes need to be crossed with the IER values, but filtered for conditions that cannot occur - for instance a receive status interrupt will not occur unless the receive status interrupt is enabled.

The receiver FIFO threshold level interrupt needs to have been seen to occur for the four possible values of the receive FIFO threshold, crossed with all the possible word formats. This covergroup needs to be sampled when a receive FIFO threshold interrupt occurs.

The receive line status interrupt has several potential sources, the functional coverage for this needs to be sampled when a line status interrupt occurs. The data that needs to be sampled is the Line Status for Parity errors, Framing errors, Overrun errors or break condition, together with the FIFO error bit, these bits need to be crossed, but with some illegal conditions filtered - for instance a line status interrupt with no active line status bits repesents an error, or a break condition will invalidate the parity and framing errors.

The transmitter empty interrupt should be checked for all possible word formats.

The modem status interrupt can come from one of the four signal change detected bits of the MSR. These bits are crossed for all permutations, and also with the MCR loopback bit. The modem status interrupt source covergroup is sampled when a modem status interrupt occurs.

Example implementations of covergroups used to capture functional coverage for the UART Interrupts can be found here.

UART interrupt coverage summary

Interrupt handling	
Coverage Criterion	**UART interrupt coverage summary**
Which values are important?	IER[3:0] - Enables for the four sources of interrupts IIR[[3:0] - Identifying the interrupt source
What are the dependencies between the values?	Interrupts should only occur if they are enabled Need to see all valid permutations of interrupt enables and interrupt sources
Are there illegal conditions?	Invalid conditions are interrupt sources reported when an interrupt type is not enabled
When is the right time to sample?	For the interrupt enables, when an interrupt occurs. For interrupt ids, when an interrupt occurs, followed by a read from the IIR register
When is the data invalid?	N/A

Receiver FIFO Threshold Interrupt	
Coverage Criterion	**UART interrupt coverage summary**
Which values are important?	LCR[5:0] - Defininig the different word formats FC[7:6] - Defining the different FIFO threshold values
What are the dependencies between the values?	Need a cross between the LCR and FCR bits to ensure that FIFO threshold interrupts have occurred for all possible permutations.
Are there illegal conditions?	None
When is the right time to sample?	When an RX FIFO threshold interrupt occurs
When is the data invalid?	N/A

Receiver Line Status Interrupt	
Coverage Criterion	**UART interrupt coverage summary**
Which values are important?	LSR[4:1] - Defining the different types of RX line status
What are the dependencies between the values?	None, each status bit has a distinct source
Are there illegal conditions?	When the break condition occurs, PE and FE are not valid
When is the right time to sample?	When a line status interrupt occurs, followed by a read from the LSR
When is the data invalid?	N/A

Transmitter Empty Interrupt	
Coverage Criterion	**UART interrupt coverage summary**
Which values are important?	LCR[5:0] - Defining the UART serial format
What are the dependencies between the values?	Cross product defining all permutations of the word format
Are there illegal conditions?	None
When is the right time to sample?	When a TX empty interrupt occurs, followed by a read from the LSR
When is the data invalid?	N/A

Modem Status Interrupt	
Coverage Criterion	**UART interrupt coverage summary**
Which values are important?	MSR[3:0] - The modem i/p signal change flags
What are the dependencies between the values?	None, each signal is orthogonal
Are there illegal conditions?	None
When is the right time to sample?	When a modem status interrupt occurs, followed by a read from the MSR
When is the data invalid?	The MSR flags are reset on read, so a second read will return invalid status

Baud Rate Divisor

Testing the baud rate divisor

The UART serial stream is sampled at 16x the data rate. Inside the UART there is a 16 bit divider which divides down the APB PCLK to generate the 16x data rate clock. It is important to check that data can be received and transmitted correctly at different data rates, but it would take too long to check all functionality at all the possible divider values (i.e. 2**16 (65536)). The larger the divider value, the slower the data rate, so most of the UART functionality is tested at a high data rate in order to save simulation time.

The UART has a data rate divider output – BAUD_O. This can be checked at different rates separately from checking the rest of the UART functionality using a specialised checker component that looks at this signal and the APB PCLK counting the interval between BAUD_O transitions. The component also receives APB bus analysis transactions so that it can determine when the divider values are changed. Functional coverage for the divider can be collected inside this component.

Functional coverage design for the baud rate divisor

The baud rate divisor can have 65536 possible values, it is unnecessary to check that all of them work. However, a number of strategic values should be chosen to ensure that specific corner cases have been seen to work. Exactly which values are important is dependent on the implementation of the divider, but what we know is that the divider value is programmed via two 8 bit registers. The covergroup is sampled on each BAUD_O transition.
If we knew the frequency of the APB PCLK in the target application we could also target the divider ratios for the standard Baud rates such as 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600 and 115200 bits per second.

Baud rate divisor coverage summary

Coverage Criterion	Receive Channel Coverage
Which values are important?	DIV1 and DIV2 register contents, potentially all possible values
What are the dependencies between the values?	DIV1 & DIV are concatonated, otherwise no dependencies
Are there illegal conditions?	The divider cannot have a value of 0
When is the right time to sample?	On the rising edge of the BAUD_O signal
When is the data invalid?	If the divider registers are being programmed, or have just been programmed in which case the divide ratio will not match the register content (this is not an error)

Register Interface

Testing the register interface

The register interface is implicitly tested by the functional stimulus for each of the various test cases. There is a specific test case to check that the register reset values are correct.

Functional coverage design for the register interface

In order to check that all possible register accesses have taken place, a cross of the register addresses and the value of the read/write bit is used. The register access covergroup is sampled at the end of each register transfer. Registers which are read only or write only are excluded from the cross for write and read accesses.

This type of register interface coverage is often generated automatically by register generators.

Register interface coverage summary

Coverage Criterion	Receive Channel Coverage
Which values are important?	Address bits [7:0] and read/write bit Only interested in valid register addresses
What are the dependencies between the values?	Need to cross the valid addresses with the read/write bit to get the register access space
Are there illegal conditions?	The MSR and LSR registers are read only, so writes to these registers are invalid
When is the right time to sample?	When an APB bus transaction completes
When is the data invalid?	N/A

UART test plan

As part of the verification planning process, a test plan should be drawn up to list all the design features to be tested and to help identify the type of functional coverage required to check that all the tests have been run for all conditions. The test plan may an input to an automated tracking system where functional coverage results are collected and compared against entries in the plan to allow the verification team to measure progress and to help identify what to work on next. Often, there will be a way of linking the covergroups and assertions in the testbench with the items in the plan document.

Writing a test plan is an iterative process, and what is shown here is a 'snapshot' of the test plan for the UART example.

UART Test Plan

Section	Description	Coverage Type	Priority
Registers			
Reset Values	All registers return the specified reset values	Test result	1
Register Accesses	All registers have been accesses for all possible access modes	Covergroup, cross	1
Bit level register accesses	All read-write bits in the registers toggle correctly	Test result	1
APB Protocol	APB Protocol has been tested in all modes	APB Monitor	1
Transmitter			
Character formats	All possible character formats are transmitted correctly	Covergroup, cross	1
TX FIFO Empty flag	The FIFO empty flag is set when the FIFO is empty and is read back correctly	Design Assertion, Covergroup	1
TX empty flag	The transmit empty flag is set correctly and is read back correctly	Design Assertion, Covergroup	1
Receiver			
Character formats	All possible character formats are received correctly	Covergroup, cross	1
Data Received Flag	The data received flag is set when data is available and is read back correctly	Design Assertion, Covergroup	1
RX Line Status			
Framing Error	Framing errors are detected for one or two stop bits	Design Assertion, Covergroup	2
Parity Error	Parity errors are detected for all types of parity	Design Assertion, Covergroup	1
Break Indication	A break condition is detected correctly for all character formats	Covergroup, cross	2
Overrun Error	RX overrun is detected for all character formats	Covergroup, cross	2
FIFOE	That the FIFO error condition is valid for all error/indication types	Covergroup, cross	2
Status	Any valid combination of error/indicator has been observed	Covergroup, cross	2
Modem Interface			
Modem Outputs	All combinations of modem output values have been seen	Covergroup, Cross	3

Modem Inputs	All combinations of modem input values have been seen	Covergroup, Cross	3
	The modem input status change signals work correctly	Design Assertion, Covergroup	3
Loopback mode	Modem output bits are routed to the right modem status bits	Covergroup	2
Interrupts			
Interrupt Enable	All combinations of the interrupt enable bits have been used	Covergroup, cross	1
Interrupt ID	All valid interrupt IDs have been detected	Covergroup, cross	1
Receive FIFO Interrupt	Seen for all possible character formats	Covergroup, cross	1
	All possible RX FIFO Threshold values checked	Covergroup, cross	2
Receive Line Status Interrupt	Interrupts generated for all possible combinations of errors and indicators for all character formats	Covergroup, cross	1
Transmit empty interrupt	Generated for all character formats	Covergroup, cross	1
Modem Status interrupt	Generated for all combinations of the signal change bits	Covergroup, cross	3
Receive timeout interrupt	Has been checked for the shortest and longest character format and 4 other formats	Covergroup	4
Baud Rate			
Divider values	Check UART operation for a range of baud rate divider values	Covergroup	1
	Check baud rate divider ratio for a selection of values via baud rate divider output	Covergroup	2
Code Coverage			
Statement coverage	Check each executable line of the RTL has been covered		1
Branch coverage	Check each branch in the RTL has been taken		1
FSM coverage	Each arc in the RTL FSMs has been taken		1

Notes:

1. The priority column indicates the relative importance of each feature. Items marked priority 1 will be verified first, followed by prioirity 2 items, down to priority 4.
2. The APB interface behaviour is checked by inserting the APB protocol monitor in the testbench, connected to the APB port on the UART, its functional coverage will be merged with the other UART functional coverage
3. There are several checks that are performed using assertions which the designer has implemented in the design, these are included in the table as Design Assertions
4. Code coverage is included as a category in the test plan so that it can be tracked

UART example covergroups

The covergroup examples on this page are implementations of the functional coverage descriptions on the Block Level Functional Coverage Example page. They illustrate different aspects of how to code covergroups and are described here to allow you to see how the high level functional descriptions map to concrete implementations. They are presented in the same order as they are described on the higher level page.

UART Transmit Channel Coverage

The functional coverage group for the transmit channel is based on the content of the UART LCR register. Its purpose is to determine that all possible word formats have been transmitted. The covergroup is sampled every time the TX data scoreboard determines that a character has been transmitted., the covergroup is implemented inside a UVM subscriber component.

```
class uart_tx_coverage_monitor extends uvm_subscriber #(lcr_item);

`uvm_component_utils(uart_tx_coverage_monitor)

covergroup tx_word_format_cg with function sample(bit[5:0] lcr);

  option.name = "tx_word_format";
  option.per_instance = 1;

  WORD_LENGTH: coverpoint lcr[1:0] {
    bins bits_5 = {0};
    bins bits_6 = {1};
    bins bits_7 = {2};
    bins bits_8 = {3};
  }

  STOP_BITS: coverpoint lcr[2] {
    bins stop_1 = {0};
    bins stop_2 = {1};
  }

  PARITY: coverpoint lcr[5:3] {
    bins no_parity = {3'b000, 3'b010, 3'b100, 3'b110};
    bins even_parity = {3'b011};
    bins odd_parity = {3'b001};
    bins stick1_parity = {3'b101};
    bins stick0_parity = {3'b111};
  }

  WORD_FORMAT: cross WORD_LENGTH, STOP_BITS, PARITY;
```

```
endgroup: tx_word_format_cg



function new(string name = "uart_tx_coverage_monitor", uvm_component parent = null);
  super.new(name, parent);
  tx_word_format_cg = new();
endfunction

function void write(T t);
  tx_word_format_cg.sample(t.lcr[5:0]);
endfunction: write

endclass: uart_tx_coverage_monitor
```

UART Receive Channel Coverage

The receive channel coverage is required for error-free reception and reception with errors or special conditions present. The covergroup used for error free reception is the same as for the transmit channel coverage, except that it is sampled when the scoreboard determines that an error-free received character is read from the UART RX buffer. For the non-error free coverage, the relevant bits of the LSR are crossed with the word format bits from the LCR register.

UART Modem Interface Coverage

The bits in the MCR register are crossed to check that all possible permutations are programmed. The bits in the MSR register are also crossed with themselves and the loopback bit in the MCR to ensure that all possible states are observed. Both covergroups are implemented inside a UVM subscriber that receives register access transactions and determines which covergroup to sample based on the address and read-write direction - see the write() method.

```
class uart_modem_coverage_monitor extends uvm_subscriber #(apb_seq_item);

`uvm_component_utils(uart_modem_coverage_monitor)



covergroup mcr_settings_cg() with function sample(bit[4:0] mcr);

  option.name = "mcr_settings_cg";
  option.per_instance = 1;

  DTR: coverpoint mcr[0];
  RTS: coverpoint mcr[1];
  OUT1: coverpoint mcr[2];
  OUT2: coverpoint mcr[3];
  LOOPBACK: coverpoint mcr[4];

  MCR_SETTINGS: cross DTR, RTS, OUT1, OUT2, LOOPBACK;
```

```
endgroup: mcr_settings_cg

covergroup msr_inputs_cg() with function sample(bit[7:0] msr, bit loopback);

  option.name = "msr_inputs_cg";
  option.per_instance = 1;

  DCTS: coverpoint msr[0];
  DDSR: coverpoint msr[1];
  TERI: coverpoint msr[2];
  DDCD: coverpoint msr[3];
  CTS: coverpoint msr[4];
  DSR: coverpoint msr[5];
  RI: coverpoint msr[6];
  DCD: coverpoint msr[7];
  LOOPBACK: coverpoint loopback;

  MSR_INPUTS: cross DCTS, DDSR, TERI, DDCD, CTS, DSR, RI, DCD, LOOPBACK;

endgroup: msr_inputs_cg

uart_reg_block rm;

function new(string name = "uart_modem_coverage_monitor", uvm_component parent = null);
  super.new(name, parent);
  mcr_settings_cg = new();
  msr_inputs_cg = new();
endfunction

function void write(T t);
  uvm_reg_data_t data;

  if((t.addr[7:0] == 8'h18) && (t.we == 0)) begin
    data = rm.MCR.get_mirrored_value();
    msr_inputs_cg.sample(t.data[7:0], data[4]);
  end
  else if((t.addr[7:0] == 8'h10) && (t.we == 1)) begin
    mcr_settings_cg.sample(t.data[4:0]);
  end

endfunction: write

endclass: uart_modem_coverage_monitor
```

UART Interrupt Coverage

There are a number of covergroups required to check the UART interrupt functional coverage.

Interrupt enable coverage

This covergroup is sampled every time an interrupt occurs. It checks the state of the IER register, decoding the bit patterns in order to determine which interrupt combinations have not been enabled.

```
covergroup int_enable_cg() with function sample(bit[3:0] en);

  option.name = "interrupt_enable";
  option.per_instance = 1;

  INT_SOURCE: coverpoint en {
    bins rx_data_only = {4'b0001};
    bins tx_data_only = {4'b0010};
    bins rx_status_only = {4'b0100};
    bins modem_status_only = {4'b1000};
    bins rx_tx_data = {4'b0011};
    bins rx_status_rx_data = {4'b0101};
    bins rx_status_tx_data = {4'b0110};
    bins rx_status_rx_tx_data = {4'b0111};
    bins modem_status_rx_data = {4'b1001};
    bins modem_status_tx_data = {4'b1010};
    bins modem_status_rx_tx_data = {4'b1011};
    bins modem_status_rx_status = {4'b1100};
    bins modem_status_rx_status_rx_data = {4'b1101};
    bins modem_status_rx_status_tx_data = {4'b1110};
    bins modem_status_rx_status_rx_tx_data = {4'b1111};
    illegal_bins no_enables = {0}; // If we get an interrupt with no enables it's an error
  }

endgroup: int_enable_cg
```

Interrupt source coverage

This covergroup checks that all possible interrupt status conditions have been sampled. It crosses the content of the IER with the IIR, and also filters conditions which cannot occur. The ignore_bins in the cross ensure that if an interrupt is disabled then interrupt ids of that type are ignored in the cross.. It is sampled when there is an interrupt followed by a read from the IIR register.

```
covergroup int_enable_src_cg() with function sample(bit[3:0] en, bit[3:0] src);

  option.name = "interrupt_enable_and_source";
  option.per_instance = 1;

  IIR: coverpoint src {
```

```
    bins rx_status = {6};
    bins rx_data = {4};
    bins timeout = {4'hc};
    bins tx_data = {2};
    bins modem_status = {0};
    ignore_bins no_ints = {1};
    illegal_bins invalid_src = default;
  }

  IEN: coverpoint en {
    bins rx_data_only = {4'b0001};
    bins tx_data_only = {4'b0010};
    bins rx_status_only = {4'b0100};
    bins modem_status_only = {4'b1000};
    bins rx_tx_data = {4'b0011};
    bins rx_status_rx_data = {4'b0101};
    bins rx_status_tx_data = {4'b0110};
    bins rx_status_rx_tx_data = {4'b0111};
    bins modem_status_rx_data = {4'b1001};
    bins modem_status_tx_data = {4'b1010};
    bins modem_status_rx_tx_data = {4'b1011};
    bins modem_status_rx_status = {4'b1100};
    bins modem_status_rx_status_rx_data = {4'b1101};
    bins modem_status_rx_status_tx_data = {4'b1110};
    bins modem_status_rx_status_rx_tx_data = {4'b1111};
    illegal_bins no_enables = {0}; // If we get an interrupt with no enables its an error
  }

  ID_IEN: cross IIR, IEN {
    ignore_bins rx_not_enabled = binsof(IEN) intersect{4'b0010, 4'b0100, 4'b0110,
4'b1000, 4'b1010, 4'b1100, 4'b1110} && binsof(IIR) intersect{4};
    ignore_bins tx_not_enabled = binsof(IEN) intersect{4'b0001, 4'b0100, 4'b0101,
4'b1000, 4'b1001, 4'b1100, 4'b1101} && binsof(IIR) intersect{2};
    ignore_bins rx_line_status_not_enabled = binsof(IEN) intersect{4'b0001, 4'b0010,
4'b0011, 4'b1000, 4'b1001, 4'b1010, 4'b1011} && binsof(IIR) intersect{4'hc, 6};
    ignore_bins modem_status_not_enabled = binsof(IEN) intersect{4'b0001, 4'b0010,
4'b0011, 4'b0100, 4'b0101, 4'b0110, 4'b0111} && binsof(IIR) intersect{0};
  }

endgroup: int_enable_src_cg
```

Receive FIFO threshold interrupt coverage

The receive FIFO threshold level is determined by bits [7:6] of the FCR register. When a receive threshold interrupt occurs, word format is crossed with the FCR bits to ensure that all possible combinations occur. It is sampled when an interrupt occurs followed by a read from the IIR register that indicates a FIFO threshold interrupt.

```
covergroup rx_word_format_int_cg() with function sample(bit[5:0] lcr, bit[1:0] fcr);

  option.name = "rx_word_format_interrupt";
  option.per_instance = 1;

  WORD_LENGTH: coverpoint lcr[1:0] {
    bins bits_5 = {0};
    bins bits_6 = {1};
    bins bits_7 = {2};
    bins bits_8 = {3};
  }

  STOP_BITS: coverpoint lcr[2] {
    bins stop_1 = {0};
    bins stop_2 = {1};
  }

  PARITY: coverpoint lcr[5:3] {
    bins no_parity = {3'b000, 3'b010, 3'b100, 3'b110};
    bins even_parity = {3'b011};
    bins odd_parity = {3'b001};
    bins stick1_parity = {3'b101};
    bins stick0_parity = {3'b111};
  }

  FCR: coverpoint fcr {
    bins one = {0};
    bins four = {1};
    bins eight = {2};
    bins fourteen = {3};
  }

  WORD_FORMAT: cross WORD_LENGTH, STOP_BITS, PARITY, FCR;

endgroup: rx_word_format_int_cg
```

Receive Line Status interrupt coverage

This covergroup is sampled when a line status interrupt occurs followed by a read from the LSR register.

```
covergroup lsr_int_src_cg() with function sample(bit[7:0] lsr);

  option.name = "lsr_int_src_cg";
  option.per_instance = 1;

  LINE_STATUS_SRC: coverpoint lsr[4:1] {
    bins oe_only = {4'b0001};
    bins pe_only = {4'b0010};
    bins fe_only = {4'b0100};
    bins bi_only = {4'b1000, 4'b1100, 4'b1010, 4'b1100}; // BI active discounts pe & fe
    bins bi_oe = {4'b1001, 4'b1101, 4'b1011, 4'b1101}; // BI active discounts pe & fe
    bins oe_pe = {4'b0011};
    bins oe_fe = {4'b0101};
    bins fe_pe = {4'b0110};
    bins no_ints = {0};
  }

endgroup: lsr_int_src_cg
```

There are a few things to note about the bins in this covergroup:

- If a Break occurs, then it is also likely to create framing and parity errors
- The receive line status interrupt enable also enables the RX timeout, this will not be detected by this covergroup which is why there is a no_ints bin

Modem Status interrupt coverage

The modem status interrupt can be caused by one of four status bits becoming true, this covergroup checks for all four bits being active and also the error condition where none are active but a modem status interrupt has occurred. The MSR conditions are crossed with the MCR loopback bit since they can be generated in normal and loopback mode. This covergroup is sampled when a modem status interrupt occurs followed by a read from the MSR.

```
covergroup modem_int_src_cg() with function sample(bit[4:0] src);

  option.name = "modem_int_src_cg";
  option.per_instance = 1;

  MODEM_INT_SRC: coverpoint src[3:0] {
    wildcard bins dcts = {4'b???1};
    wildcard bins ddsr = {4'b??1?};
    wildcard bins teri = {4'b?1??};
    wildcard bins ddcd = {4'b1???};
    illegal_bins error = {0};
  }
```

```
  LOOPBACK: coverpoint src[4] {
    bins no_loopback = {0};
    bins loopback = {1};
  }

  MODEM_INT_CAUSE: cross MODEM_INT_SRC, LOOPBACK;

endgroup: modem_int_src_cg
```

Note that the fidelity of this covergroup is reduced since wildcard bins are used to check that each of the MSR interrupt source bits is seen to be active true, rather than all combinations. The reasoning behind this is that each bit is orthogonal to the other, and that therefore there is no functional relationship between them.

Baud Rate Divisor coverage

The baud rate divisor registers are sampled every time the baud rate checker determines that the baud rate has settled. The values sampled have been selected on the basis that they represent values likely to cause problems to the implementation of the divider.

```
covergroup baud_rate_cg() with function sample(bit[15:0] div);

  coverpoint div {
    bins div_ratio[] = {16'h1, 16'h2, 16'h4, 16'h8,
                        16'h10, 16'h20, 16'h40, 16'h80,
                        16'h100, 16'h200, 16'h400, 16'h800,
                        16'h1000, 16'h2000, 16'h4000, 16'h8000,
                        16'hfffe, 16'hfffd, 16'hfffb, 16'hfff7,
                        16'hffef, 16'hffdf, 16'hffbf, 16'hff7f,
                        16'hfeff, 16'hfdff, 16'hfbff, 16'hf7ff,
                        16'hefff, 16'hdfff, 16'hbfff, 16'h7fff,
                        16'h00ff, 16'hff00, 16'hffff};
  }
endgroup: baud_rate_cg
```

Register Access Coverage

The purpose of this covergroup is to check that all valid register accesses have occured. It is sampled every time there is a register access. It has named bins for each of the valid register addresses and for each state of the read/write bit, these are crossed with an ignore_bins to ensure that the read only LSR and MSR registers are handled correctly.

```
covergroup reg_access_cg();

  option.name = "reg_access_cg";
  option.per_instance = 1;

  RW: coverpoint we {
```

```
    bins read = {0};
    bins write = {1};
  }

  ADDR: coverpoint addr {
    bins data = {0};
    bins ier = {8'h4};
    bins iir_fcr = {8'h8};
    bins lcr = {8'hC};
    bins mcr = {8'h10};
    bins lsr = {8'h14};
    bins msr = {8'h18};
    bins div1 = {8'h1c};
    bins div2 = {8'h20};
  }

  REG_ACCESS: cross RW, ADDR {
    ignore_bins read_only = binsof(ADDR) intersect {8'h14, 8'h18} && binsof(RW) intersect {1};
  }

endgroup: reg_access_cg
```

UART Example Source Code

The source code for the UART covergroups, the UART RTL and a UVM testbench that implements several tests can be downloaded from here:

(download source code examples online at https://verificationacademy.com/cookbook/code-examples).

Note that the RTL for the UART is example code, supplied as part of the example and there is no warranty on its correctness. On no account should it be used in a real design.

Datapath Coverage

What is a datapath block?

A datapath block takes an input data stream and implements a transform function that generates the output data. The transfer function may have settings which change its characteristics, or it may be a fixed implementation. In its path from the input to the output, the data does not interact with other blocks, hence the term datapath. Examples of datapath blocks include custom DSP functions, modems, encoders and decoders and error correction hardware.

A datapath block is generally tested with meaningful, rather than random, data and the output is related to the input by the transform function and is therefore meaningful as well. The input to a datapath block is most likely generated from a software (c) based model of the system in which the function was originally modelled, and the output of the block is usually compared against the output from a golden reference model.

In some cases the output data may require subjective testing. For instance, a video encoding block would require a video format signal as its input and the encoded output would have to be checked visually to check that the result of the encoding was of an acceptable quality.

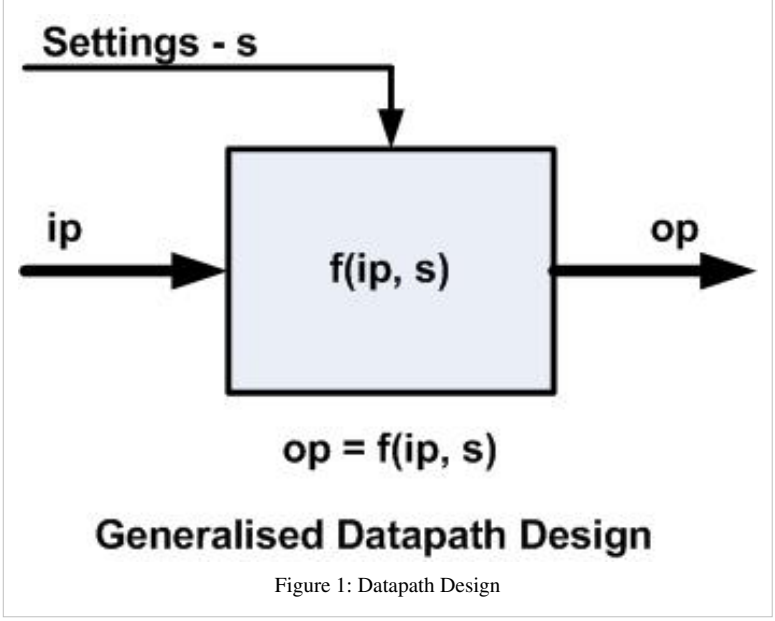


Figure 1: Datapath Design

Functional coverage for a datapath block is usually focussed on its settings (sometimes referred to as the "knobs"), or the parameters which affect its transform function. The role of the functional coverage model is to check that the block has been tested with all desired combinations of parameter settings. The value of the data that is fed into the datapath block may also be relevant to the coverage since it could be used to prove that a combination of input values has been processed against each valid set of parameters.

The BiQuad IIR Filter Example

In order to illustrate how the functional coverage model would be designed for a datapath block, a relatively simple BiQuad IIR filter will be used as an example. This filter takes in a digital stream of data that represents an analog value and the values of the co-efficients of the different taps in the filter allow the input data to be processed to generate the output data. Using different co-efficient values the filter can be configured to either be a Low Pass (LP), a High Pass (HP) or a Band Pass (BP) filter, with different roll-off points. The maths for calculating the co-efficient values is not presented here, but can be calculated using a spreadsheet.

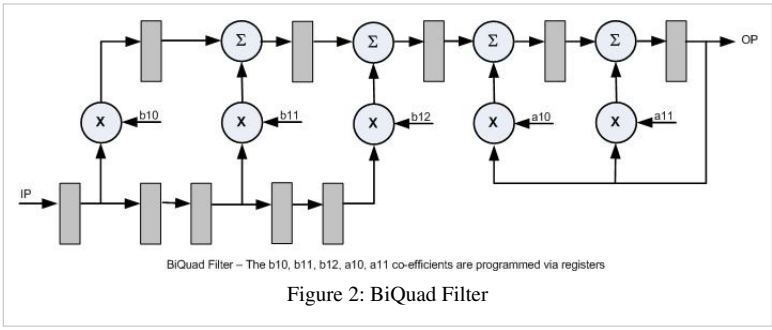


Figure 2: BiQuad Filter

In theory, the BiQuad filter design can handle a continuous, or infinite, range of possible input and co-efficient values, so the the verification problem needs to be constrained to something practical. In this case, the IIR filter is going to be used as a programmable filter for audio data with frequencies between 50 Hz and 20 KHz, and it will be tested for correct operation as a Low Pass, High Pass and Band Pass filter over the frequency range, varying the co-efficient values to set the corner frequencies. The co-efficients are stored in registers which can be programmed using an APB interface. The input data will be a frequency swept sine wave and the resultant output sine wave will be checked to make sure that the right level of attenuation has been achieved according to the intended characteristics of the filter. The diagram below illustrates the filter testbench architecture.

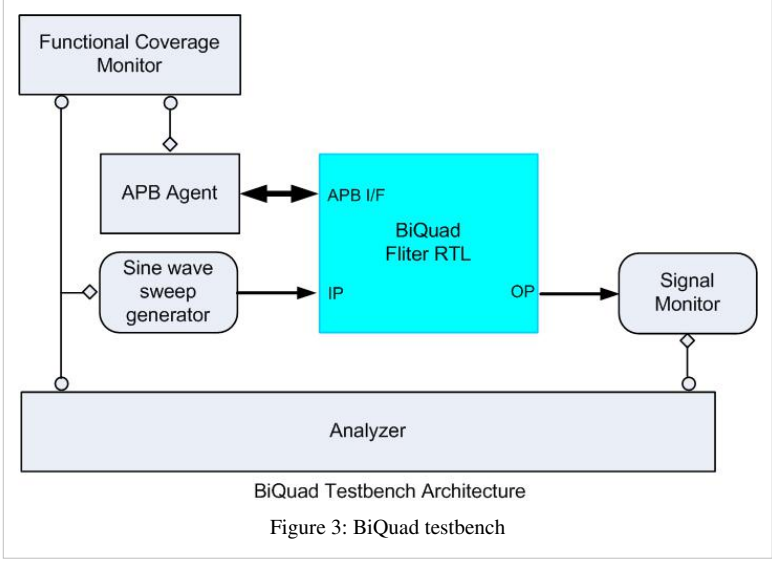


Figure 3: BiQuad testbench

For each filter type, the filter parameters for corner frequencies will be tested at 200 Hz intervals in the 0 - 4 KHz range, and then at 1 KHz intervals in the 4-20 KHz range - this equates to 36 possible sets of co-efficient values, each of which are valid for a particular corner frequency.

The input frequency sweep waveform will be sampled to ensure that it covers all the frequencies of interest and this information should be crossed with the set of co-efficient values to ensure that all possible combinations have been observed. This strategy is summarised in the BiQuad IIR Filter Test Plan.

In terms of sampling, the covergroups for a particular filter type should only be sampled when the filter has been configured in that mode, and they should be sampled when the input frequency crosses a frequency increment boundary.

Coverage Criterion	BiQuad Filter Coverage
Which values are important?	The calculated discrete values for the filter co-efficients, ordered by filter type. Several discrete frequencies.
What are the dependencies between the values?	The co-efficients should be crossed with the input frequency to check all options have been tested.
Are there illegal conditions?	No since we are representing a sub-set of a continuous range of values, but some filter/frequency values are out of range.
When is the right time to sample?	When the frequency sweep waveform is sampled at one of the frequencies of interest.
When is the data invalid?	The right covergroup needs to be sampled for the right type of filter (LP, HP, BP)

See the example implementation of the BiQuad functional coverage model for more details.

BiQuad IIR Filter test plan

As part of the verification planning process a functional coverage/test plan should be created. In the case of the BiQuad IIR filter, the primary concern is to check that it can be configured to act as a low, high and band pass filter in the audio frequency range and that it functions correctly in each of these modes.

BiQuad IIR Filter Test Plan

Section	Description	Coverage Type	Priority
Low Pass Filter Operation			
	The filter can be configured and operates correctly as a low pass filter in the audio frequency range (0-20Khz)	Covergroup	1
High Pass Filter Operation			
	The filter can be configured and operates correctly as a high pass filter in the audio frequency range (0-20Khz)	Covergroup	2
Band Pass Filter Operation			
	The filter can be configured and operates correctly as a band pass filter in the audio frequency range (0-20Khz)	Covergroup	3

BiQuad IIR Filter example covergroups

The functional coverage model for the BiQuad IIR filter is based on ensuring that all variations of 3 sets of variables have been checked by the stimulus:

- The filter type - Low Pass, High Pass and Band Pass
- The filter specification, in terms of the co-efficients - b10, b11, b12, a10, a11
- The input frequency, measured in hertz

To create the coverage model, we need to create a set of covergroups and a coverage monitor.

Covergroup Design

Each filter configuration is represented by a set of co-efficient values. These are effectively unique and can be separated out into groups of values that apply to each of the three filter types, these values then need to be crossed with the filter input frequency to check that coverage has been obtained for all possible combinations.

One way to do this would be to create a single covergroup with separate coverpoints for each filter type, with bins for each combination of filter co-efficient values. However, at any particular time the BiQuad filter can only be configured to operate in one mode, so there is a covergroup for each of the filter types, and only one of each of the covergroups will be sampled at each particular frequency change. The example shown in the code example is for the Low Pass filter type, but the other covergroups only differ in terms of the co-efficient values.

```
class LP_FILTER_cg_wrapper extends uvm_object;

`uvm_object_utils(LP_FILTER_cg_wrapper)

// Co-Efficient values
bit[23:0] b10;
bit[23:0] b11;
bit[23:0] b12;
bit[23:0] a10;
bit[23:0] a11;

// LP_FILTER Covergroup:
covergroup LP_FILTER_cg() with function sample(int frequency);

  option.name = "LP_FILTER_cg";
  option.per_instance = 1;

  // Bins for frequency intervals:
  IP_FREQ: coverpoint frequency {
    bins HZ_100 = { 100 };
    bins HZ_200 = { 200 };
    bins HZ_400 = { 400 };
    bins HZ_800 = { 800 };
    bins HZ_1k = { 1000 };
```

```
    bins HZ_1k5 = { 1500 };
    bins HZ_2k = { 2000 };
    bins HZ_2k5 = { 2500 };
    bins HZ_3k = { 3000 };
    //  .......
    bins HZ_17k = { 17000 };
    bins HZ_18k = { 18000 };
    bins HZ_19k = { 19000 };
    bins HZ_20k = { 20000 };
  }

  // Co-efficient bins for different low pass knee frequencies:
  CO_EFFICIENTS: coverpoint {b10, b11, b12, a10, a11} {
    bins CE_200 = {120'h0002C10005830002C1FDA1603DAC66};
    bins CE_400 = {120'h000AD30015A6000AD3FB43263B6E74};
    bins CE_800 = {120'h0029C90053930029C9F68940373067};
    bins CE_1000 = {120'h004029008052004029F42E2B352ED0};
    bins CE_1200 = {120'h005ACF00B59E005ACFF1D4AA333FE8};
    bins CE_1400 = {120'h00798300F307007983EF7CF4316303};
    bins CE_1600 = {120'h009C11013822009C11ED27392F977E};
    bins CE_1800 = {120'h00C24501848B00C245EAD3A32DDCBA};
    // ....
    bins CE_16k = {120'h1DC4A13B89431DC4A127B0D70F61AE};
    bins CE_17k = {120'h20FA4441F48920FA4431EA5811FEBA};
    bins CE_18k = {120'h246A9C48D538246A9C3C563415543B};
    bins CE_19k = {120'h281DC4503B88281DC446FCC7197A49};
    bins CE_20k = {120'h2C1D25583A4B2C1D2551E4AB1E8FEA};
  }

  LP_X: cross CO_EFFICIENTS, IP_FREQ;
endgroup: LP_FILTER_cg

function new(string name = "LP_FILTER_cg_wrapper");
  super.new(name);
  LP_FILTER_cg = new();
endfunction

function void sample(int frequency);
  LP_FILTER_cg.sample(frequencey);
endfunction

endclass: LP_FILTER_cg_wrapper
```

A SystemVerilog covergroup is intantiated inside a class, it has to be constructed in the class constructor method (new()). The Low Pass filter covergroup is instantiated inside a wrapper class, this allows it to be created when required by constructing the wrapper object. The covergroups sample() method is then chained into the wrapper object's sample() method. This is the recommended way to implement covergroups in a class environment.

Inside the covergroup itself there is a coverpoint for the frequency which has a set of bins which correspond to each of the input frequencies of interest. The coverpoint for the co-efficients is based on the concatonated value of all of the co-efficients (an 80 bit value), and the bins correspond to the co-efficient values for different configurations from a 200 Hz knee frequency up to 20 Khz. The cross product of the two coverpoints is LP_X.

Functional Coverage Monitor Design

The functional coverage monitor component that is used to sample the different covergroups is an extension of a uvm_subscriber component class. The monitor has a write() method which is called each time the frequency monitor in the testbench detects a change in the input frequency. The monitor contains a covergroup wrapper for each of the filter type covergroups. The write method checks which mode the filter is being tested in and then samples the corresponding covergroup. Since the co-efficients for the BiQuad filter are stored in registers, the co-efficient values are stored in a UVM register model and then assigned to the appropriate covergroup wrapper object before its sample() method is called. The write() method is effectively ensuring that the right covergroup is sampled when there is a change in the input frequency.

```
class biquad_functional_coverage extends uvm_subscriber #(freq_sample);

`uvm_component_utils(biquad_functional_coverage)

// Filter mode is defined in the env configuration object
// together with the register model:
biquad_env_config cfg;

// Cover groups - one for each type of filter:
LP_FILTER_cg_wrapper lp_cg;
HP_FILTER_cg_wrapper hp_cg;
BP_FILTER_cg_wrapper bp_cg;

extern function new(string name = "biquad_functional_coverage", uvm_component parent = null);
extern function void build_phase(uvm_phase phase);
extern function void write(T t);

endclass: biquad_functional_coverage

function biquad_functional_coverage::new(string name = "biquad_functional_coverage", uvm_component parent = null);
  super.new(name, parent);
endfunction

function void biquad_functional_coverage::build_phase(uvm_phase phase);
  lp_cg = LP_FILTER_cg_wrapper::type_id::create("Low_Pass_cg");
```

```
  hp_cg = HP_FILTER_cg_wrapper::type_id::create("High_Pass_cg");
  bp_cg = BP_FILTER_cg_wrapper::type_id::create("Band_Pass_cg");
endfunction: build_phase


function void biquad_functional_coverage::write(T t);
  // Update the filter co-efficients and then sample
  // according to the filter mode:
  case(cfg.mode)
    LP: begin
          lp_cg.b10 = cfg.RM.B10.f.value[15:0];
          lp_cg.b11 = cfg.RM.B11.f.value[15:0];
          lp_cg.b12 = cfg.RM.B12.f.value[15:0];
          lp_cg.a10 = cfg.RM.A10.f.value[15:0];
          lp_cg.a11 = cfg.RM.A11.f.value[15:0];
          lp_cg.sample(t);
        end
    HP: begin
          hp_cg.b10 = cfg.RM.B10.f.value[15:0];
          hp_cg.b11 = cfg.RM.B11.f.value[15:0];
          hp_cg.b12 = cfg.RM.B12.f.value[15:0];
          hp_cg.a10 = cfg.RM.A10.f.value[15:0];
          hp_cg.a11 = cfg.RM.A11.f.value[15:0];
          hp_cg.sample(t);
        end
    BP: begin
          bp_cg.b10 = cfg.RM.B10.f.value[15:0];
          bp_cg.b11 = cfg.RM.B11.f.value[15:0];
          bp_cg.b12 = cfg.RM.B12.f.value[15:0];
          bp_cg.a10 = cfg.RM.A10.f.value[15:0];
          bp_cg.a11 = cfg.RM.A11.f.value[15:0];
          bp_cg.sample(t);
        end
  endcase
endfunction: write
```

Although, the functional coverage model has been implemented as a UVM class, the same principles could be applied to a module or interface based implementation.

Example Source Code

The source code for the Biquad example can be downloaded from here:

(download source code examples online at https://verificationacademy.com/cookbook/code-examples).

SoC coverage example

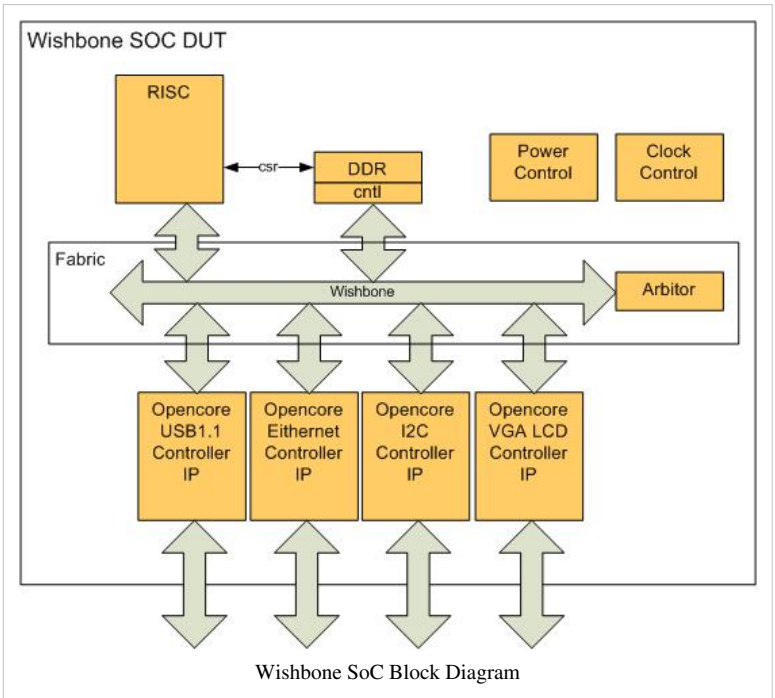
System level functional verification can take full advantage of the fact that the entire design is a self contained unit that will be used by customers, and thus has some logical use model that the customer will follow. Also, being a system, often it is made up of trusted IP, and the verification focus is aimed more at the block interconnect and any new functionality. Working at a system level, leads naturally to the creation of a coherent coverage model spreadsheet/testplan using the top down method. The desired outcome of the verification process is that the system has been thoroughly verified by exercising a viable subset of the system's design intent and behavior. The functional coverage model is part of the instrumentation used to determine whether the design has been verified to the desired level of quality.

In general the verification strategy at the SoC level is quite different from the block level. At the SoC level typically you are reusing blocks and are really just interested in verifying that the reused blocks are interconnected together correctly, that the different parts of the system interact with each other correctly, and then that you are getting acceptable performance. In most SoCs, if you include all the full use model(s) of all blocks, then there will always be more use case scenarios than you will have time to verify, therefore the most important ones need to be prioritized. Generally your focus can be prioritized by what areas are new (untested IP, brand new RTL) or problematic. It is a good idea to review each IP block for any coverage model spreadsheets and determine whether they are viable for reuse at the SoC level. Some engineers will use the existence of a good coverage model as one of the key criteria for selecting IP.

The system level design example is a Wishbone system on a chip (SoC), which contains a RISC based processor assembly (CPU with memory), an arbitrated Wishbone fabric, and four interface blocks (USB, Ethernet, I2C & VGA cores).

Wishbone SoC Overview

A block diagram of the Wishbone SoC is below:



Wishbone SoC Block Diagram

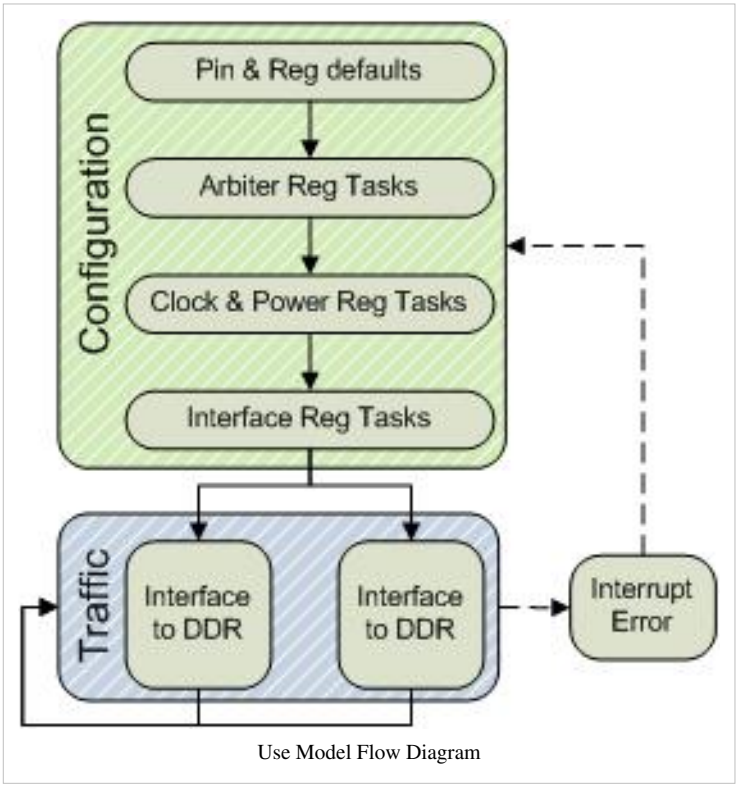
Notice that the blocks are interconnected using an arbitrated Wishbone bus fabric. A RISC processor assembly with DDR memory, used for both firmware & traffic storage is attached to the fabric, as well as for interface cores. There is also side logic that takes care of power and clock control.

Here are some facts about this design (These facts are just arbitrary for the purpose of the example):

- Trusted, reused IP:
  - Ethernet, USB, I2C, VGA
- New IP:
  - RISC processor & memory: one rev back, so should be stable, but new to us
- New project specific design:
  - Wishbone Fabric, plus Clock & Power control
- The testbench will use an available wishbone agent in place of the RISC processor to drive the testbench. So in reality the Wishbone SoC, from a verification stimulus perspective, is a series of wishbone single or block operations (reads, writes, read-modify-writes) going across the fabric.
- The DDR located firmware has late availability and will be folded in when ready, but not available for most of the verification effort.
- The interconnect, configuration and throughput are the main concerns, especially the power and clock control to minimize power consumption.
- There is a Wishbone SoC architecture document with some basic register, power and clock implementation information.
- There are IP block level documents for the 5 IP cores (Processor, USB, Ethernet, I2C, VGA), but they are register and interface oriented, with minimal design detail. The I2C core however has a testplan spreadsheet that we can fold into the SoC testplan hierarchy

Functional Coverage Model Creation: Use Model Diagrams

Like most SoC architectural documentation, the Wishbone SoC design architecture doc does not have a use model description. The architects did make a two page "data sheet" with the block diagram (above) and a list of features that have some use model information. So a high level use model had to be generated with the help of the chief design architects.



Use Model Flow Diagram

The result was this flow diagram and steps:

1. Setup/Configuration
   - Pin and default register values control some initialization
   - Firmware sets registers via single write Wishbone operations
   - Arbiter set up
   - Clock and Power set up
   - Interface set up (USB, Ethernet, I2C, VGA cores)
2. Traffic via memory wishbone operations (single or block moves)
   - DDR to Interface
   - Interface to DDR
3. Unexpected event
   - Interrupt, error, etc

This flow diagram is useful to show the overall high level use of the chip and even at this level starts to show coverage both in its flow and its diverging choices. To go further, each block in the flow needs to be expanded out using whatever table or diagram that can best represent the setup or data flow within that area. The diagrams typically used for this process are described in the top down section of the coverage model testplan creation. The following sections illustrate the use of the various diagram types in the context of the WB SoC example.

Configuration Part 1: Table Example

Pin & Register Default upon Power Up				
Firm ware Pins (1:0)	00 - ROM	01- DDR	10 – I2C	11 - USB
Pwr Reg	All off	All on	Proc Only	Top Only
Clk Reg	6 regions: all can be off or on (on have 1 or more speeds)			

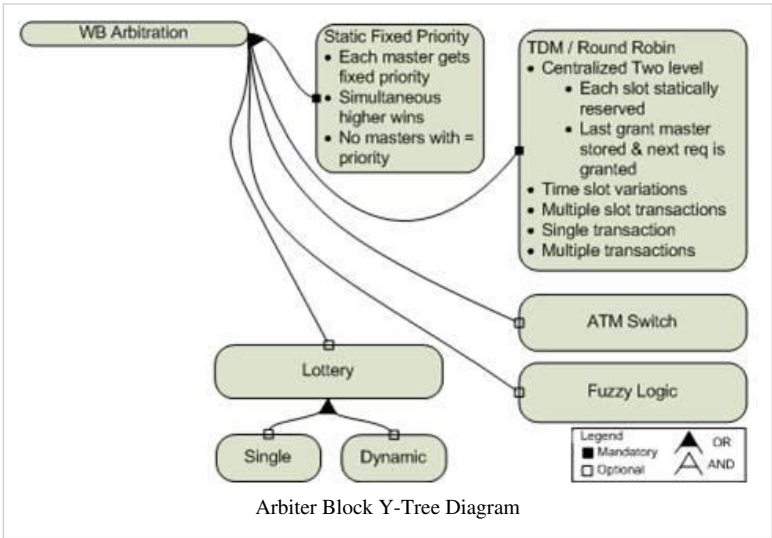
Pin and Reg Configurations

Take each of the sub blocks in the high level use model flow and expand it out using a table, diagram or chart that is best suited to describe that sub block's information. For instance the first "Pin and Register defaults" sub block describes how the Wishbone SoC is initialized upon power on and is best expanded into a table. A simple table is used here because of the small space of this simple power up and default configuration. There are just two pins that select where the firmware will come from: inside the ROM of the processor, preloaded in the DDR, or read into the DDR from the I2C or the USB. The startup power state is hard coded as defaults in the power register with only 4 choices. Likewise the start up clock register has 6 bits, one for each region (4 interfaces, fabric and the processor subsystem), which can be on or off, and each with a separate default speed register.

This table then leads to a section in the testplan and individual requirements. See sections 1.1, 1.2 and 1.3 in the testplan spreadsheet picture below.

Configuration Part 2: Y Tree Example

The Arbiter sub block is best expanded into a Y-Tree diagram:

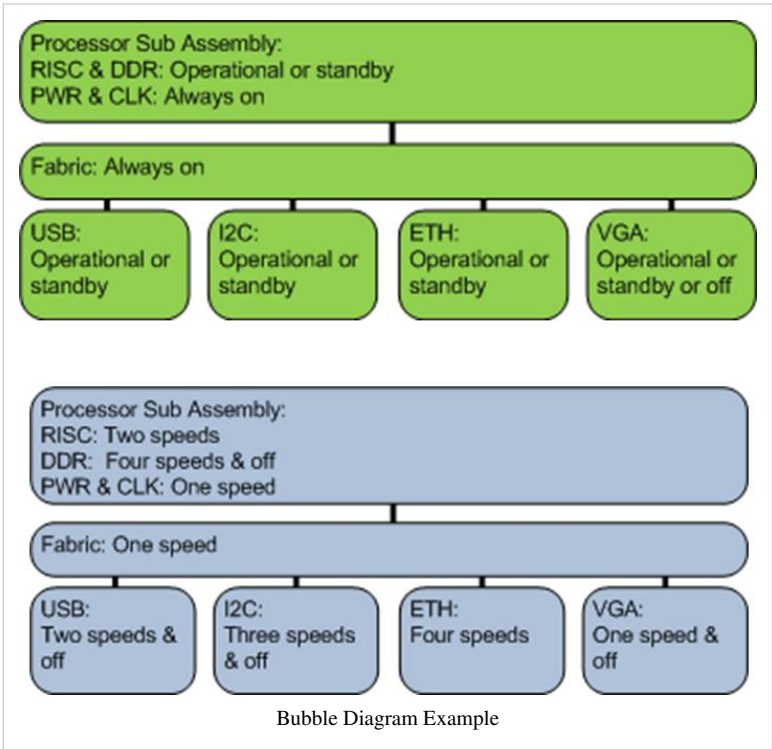


Arbiter Block Y-Tree Diagram

Here the Arbiter configuration choices are shown in a Y tree diagram. Notice the legend for mandatory and optional nodes and OR or AND choices. In practice, the Wishbone is arbitrated with a round robin, but the diagram shows more possibilities with 5 classic arbitration schemes. All 5 arbitration schemes could be further Y'd into many other sub choices. The Y tree diagram is good for showing choices and priority.

This Y-Tree diagram then leads to a section in the testplan and to individual requirements. See sections 2.1 to 2.2.5 in the testplan spreadsheet picture below.

Configuration Part 3: Bubble Diagram Examples

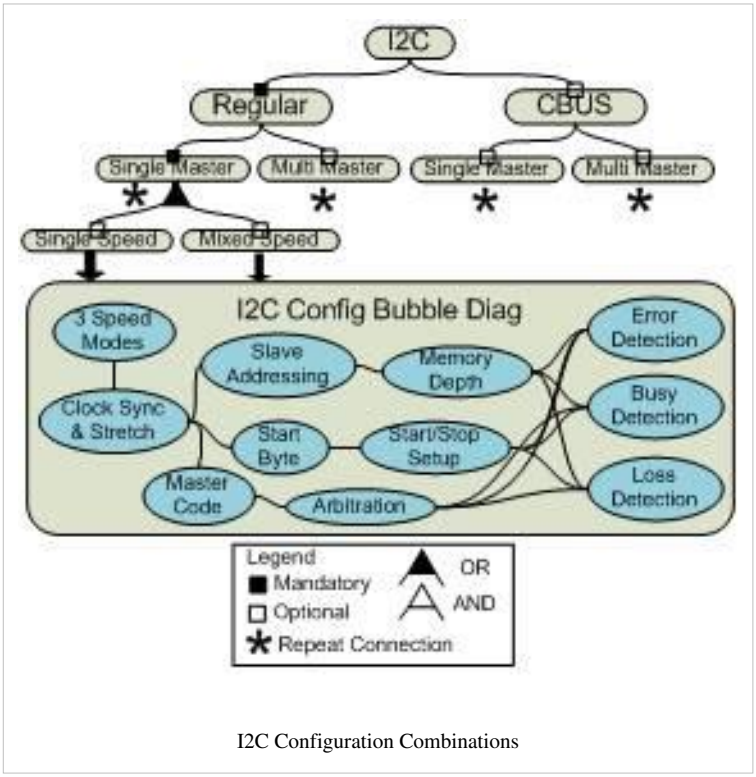


Bubble Diagram Example

Most SoCs address various clocking domains and power consumption, especially static power issues. The power and clock management logic for SoCs is growing in complexity and thus needs a prioritized place in the overall verification strategy and the testplan. The power and clock configurations are best described using bubble diagrams that mimic the block diagram of the chip. Each of the six regions of the WB SoC is shown with its corresponding power and clock configuration information. Bubble diagrams work well in this situation, showing the relationship of each area to one another, as well as each bubble showing that particular region's power and clock settings.

These bubble diagrams then lead to sections in the testplan and individual requirements. See sections 3.1-2 for clocking and 4.1-2 for power in the testplan spreadsheet picture below.

Configuration Part 4: I2C Config Combination Example

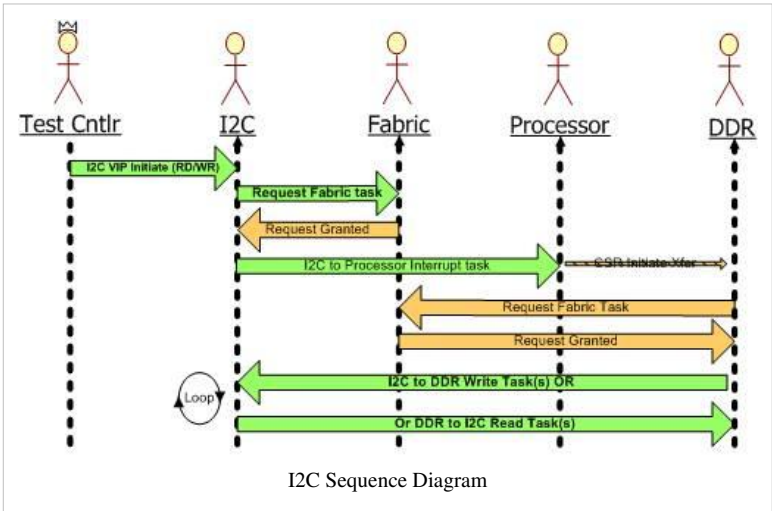


I2C Configuration Combinations

The I2C blocks configuration is described using a combination of Y-tree and bubble diagrams. The Y tree at the top shows the choices between the regular and special CBUS modes of the I2C. It also shows the choices between the number of allowed masters and their speed choices. Then a bubble diagram is used to show other various configuration areas and their relationship (with the lines). Because the information in the bubble diagram is the same for all four choices a * is used to indicate that this info is repeated for the other 3 choices. This illustrates how you can mix and match various tables and diagram styles to best convey the necessary information.

This combination diagram then leads to a section in the testplan and individual requirements. See section 5.3 in the testplan spreadsheet picture below. It is hierarchically referenced to the testplan that came with the i2c IP.

Traffic: I2C Sequence Diagram Example



I2C Sequence Diagram

After all the configurations are done, then traffic is initiated. This is an example sequence diagram for the I2C read or write. The diagram shows the handshake between the testbench, starting with the testbenches test controller:

1. The test controller initiates by telling the I2C vip to start a transaction sequence, read or write.
2. The I2C IP then requests the bus from the fabric and the arbiter grants the request when ready.
3. Next the I2C IP sends an single write operation to the processor, declaring the direction (read or write) and size, etc.
4. The processor runs some firmware and tells the DDR via the CSR to initiate the transfer.
5. The DDR requests the bus from the fabric and the arbiter grants the request when ready.
6. The I2C then sends or receives the data via single or block operations (depending on the size, and read or write) and releases the bus when done.

The sequence diagram (borrowed from UML) is a great way to show data movement and handshaking. If there are hundreds and hundreds of data sequences, you do not need a sequence diagram for each one, but can instead divide them up into categories of similar sequences and make a "family" sequence diagram for each one. You can also often show both directions (Read and Write) on the same diagram as we did above for the WB SoC. In the I2C example above we might have other sequence diagrams with throttled data speeds, stalls, retries, errors, etc. The firmware might also direct other types of operations, and each could have its own sequence diagram.

SoC Firmware

The Wishbone SoC, like all SoCs, will ultimately be driven by software running on the processor. This firmware is not available until late in the project, and has its own development and testing process. There are several sound approaches for integrating firmware into the verification process:

- If the processor is not trusted, a processor and memory subsystem testbench can be created where firmware can be brought in in stages as it is made available. Firmware can be divided out by low level and high level functionality, prioritized into what will be done on the subset testbench and what will be done using other means (C model testing, prototypes, first pass chips, etc.). This layering of the firmware testing can be represented as diagrams and included in the coverage model testplan spreadsheet.
- If the processor is trusted, it can be left out of the main SoC testbench. This is the approach that was used for the Wishbone SoC. A Wishbone VIP agent is placed where the processor would be and a Control Status Register (CSR) agent was created to drive the CSR interface to the DDR. The two agents work in concert as directed by the top level

test controller/virtual sequence to mimic the processor firmware activity on both the Wishbone bus and the DDR memory. In this way the goal of focusing on the overall interconnect; configuration and throughput traffic across fabric is addressed.

- Another approach is to have the actual processor RTL in place, and to put pseudo firmware into the DDR. This preliminary pseudo firmware code is made up of necessary low level functions to do basic firmware operations, like doing register read and writes across the fabric to do configuration, or to do basic data moves between the DDR and one of the four interfaces. The testbench then controls the running of these functions via a back door access. Questa's infact has a software driven verification package for addressing this type of problem.

Whichever method is used, solely or in various combination, it is important that these strategies are fleshed out early and incorporated into both the overall verification architecture and implementation documentation and the coverage model testplan spreadsheet. Several columns can be added to any testplan spreadsheet that spell out how and where various firmware features will be used, tested and covered.

Functional Coverage Model Creation: Spreadsheets

Once the use model of the Wishbone SoC has been fleshed out into a progressive collection of useful diagrams & tables, they are typically entered into a single document for easy viewing and dissemination. See the end of the top down example on the Specification to testplan creation page for an explanation. This single use case description, or Day in the Life (DITL), document can then be combed over like any other specification as described in the bottom up example of the same link, extracting the various requirements to enter in as rows into a Wishbone coverage model testplan spreadsheet. Each diagram can serve as a section or subsection in the spreadsheet, and the various table content and choices in the diagrams naturally point to coverage groups. An example of what this might look follows: Click here for a easier to read pdf version of the WB testplan. You can also download the xml file below.

WB SoC Top Level Coverage Model Testplan

Section	Title	Description	Link	Type	Weight	Goal	Owner	Priority
1	Initialization	Power on bring up settings			1	100		
1.1	Firmware Source	FW_SRC pins (defined in WBsocDAD section 5.3.3) define the initilization location of the base firmware. Four possible sources [ROM, DDR, I2C, USB]	cov_init_fw_src_cp	CoverPoint	1	100	Fred	2
1.2	Power default Cfg	The power register (defined in WBsocDAD section 5.3.1) defines the hard coded default power settings upon initialization. Four settings [All off, All on, Processor Only, Top Only]	cov_init_pwr_cp	CoverPoint	1	100	Sam	2
1.3	Clock default Cfg	The clock register (defined in WBsocDAD section 5.3.2) defines the hard coded default clock settings upon initialization. All 6 zones have their own settings [On/Off, speed setting]	cov_init_clk_zone_cp cov_init_clk_speed_cp cov_init_clk_cross	CoverPoint CoverPoint Cross	1	100	Sam	2
2	Arbitraion Cfg	Only 2 of the 5 arbitration schemes used			1	100		
2.1	Static Fixed Priority				1	100	Joan	1
2.1.1	Priority Setting	Setting each source/destination as master or slave and its priority (defined in WBsocDAD section 3.2.12).	cov_sfp_priority_cp	CoverPoint	1	100	Joan	1
2.1.2	Fixed Priority	All source/destination priority unique	cov_sfp_fixed_cp	CoverPoint	1	100	Joan	1
2.1.3	Higher Wins	Simultaneous access with higher priority winning	cov_sfp_sametime_cp	CoverPoint	1	100	Joan	2
2.1.4	Equal Priority	Some source/destinations with same priority, first wins and sumultaneous	dt_sfp_equal cov_sfp_equal_cp	Test CoverPoint	1	100	Joan	3
2.2	TDM/RoundRobin			CoverPoint	1	100	Joan	1
2.2.1	Slot Reservation		cov_tdm_slotset_cp	CoverPoint	1	100	Joan	1
2.2.2	Last Master Granted	Multiple requests with queued up requests to test last master granted FSM (defined in WBsocDID section 6.5.2)	cov_tdm_lmg_cp	CoverPoint	1	100	Joan	3

Wishbone SoC Testplan

WB SoC Top Level Coverage Model Testplan (continued)

Section	Title	Description	Link	Type	Weight	Goal	Owner	Priority
2.2.3	Time Slot Variation	Total number of slots and slot duration (defined in WBsocDAD section 4.1.7).	cov_tdm_slotnum_cp cov_tdm_slotlength_cp cov_tdm_slot_cross	CoverPoint CoverPoint Cross	1	100	Joan	3
2.2.4	Multi Slot Transactions				1	100	Joan	3
2.2.5	Arbitration Metrics	Total number of req/ack pairs, conflicts & no acks. Req to Ack times	cov_tdm_metric_cg	CoverGroup	1	100	Joan	4
3	Clock Cfg				1	100		
3.1	Block Clk Configuration	See clock bubble diagram (defined in WBsocDITL figure 5.1 & priority paragraph following) and capture all the prioritized configurations into coverpoints and crosses. Expand out here later.	cov_clk_cfg_cg	CoverGroup	1	100	Sam	1
3.2	Block Clk timing checks	See clock definitinons tables (defined in WBsocDID Tables 4.1-6) and apply assertions for the clock periods and clock skews. Expand out here later.	assert_clk_xyz_prd assert_clk_abc_skew	Assertion Assertion	1	100	Sam	2
4	Power Cfg				1	100		
4.1	Block Clk Configuration	See clock bubble diagram (defined in WBsocDITL figure 5.1 & priority paragraph following) and capture all the prioritized configurations into coverpoints and crosses. Expand out here later.	cov_clk_cfg_cg	CoverGroup	1	100	Sam	1
4.2	Block Clk timing checks	See clock definitinons tables (defined in WBsocDID Tables 4.1-6) and apply assertions for the clock periods and clock skews. Expand out here later.	assert_clk_xyz_prd assert_clk_abc_skew	Assertion Assertion	1	100	Sam	2
5	Interface Cfg				1	100		
5.1	usb cfg	Separate hierarchical spreadsheet	-root 5.1 usb.xml	XML	1	100	Ralph	see xls
5.2	eth cfg	Separate hierarchical spreadsheet	-root 5.2 eth.xml	XML	1	100	July	see xls
5.3	i2c cfg	Separate hierarchical spreadsheet from vip	-root 5.3 i2c.xml	XML	1	100	George	see xls
5.4	vga cfg	Separate hierarchical spreadsheet	-root 5.4 vga.xml	XML	1	100	Ben	see xls
6	Traffic	TBD - Filled out in next planning session			1	100		
6.1		TBD - Filled out in next planning session			1	100		
7	Interupt	TBD - Filled out in next planning session			1	100		
7.1		TBD - Filled out in next planning session			1	100		

Wishbone SoC Testplan (continued)

This spreadsheet shows the basic necessary content for a testplan. A real testplan for a large SoC would be larger (at least 500 rows), but this example is reduced to fit here. See the Coverage Plan Format article for a general description of what the various columns are used for and what are legal entries.

Things to notice in the WB SoC testplan spreadsheet:

- Notice that other documents are referenced in many of the descriptions. There is no need to re-enter redundant information again here, just reference the document and section.
- Notice the descriptions are short and not formal. Some verification teams have a prioritized language with specific definitions of specific words for their definitions. A description has to start with one of these key words, for example "The WB SoC shall...." or "The i2c interface will only use ....".
- Notice (section 2.2.5, 3.1, 3.2, etc.) are not as detailed for now and are left for future expansion. These will probably be broken out into more rows (2.2.5.1, etc.) where the specific coverpoints are defined. They have been started here, so that they are not forgotten.
- Notice the naming conventions of the links, dt for directed test, assert_ for an assertion, and cov_ to start a functional coverage group or point with _cg or _cp at the end. Distinctive acronyms like "sfp" for static fixed priority are used for clarity. These nomenclatures make it easier to write scripts to manipulate coverage information. These conventions should be decided upon at that start of a project, written down, and used throughout a project uniformly.
- Notice on rows 5.1-4 that separate, lower level spreadsheets are linked in here hierarchically. The i2c one came with its VIP, the other 3 will be new, but will each be in their own separate testplan spreadsheet. The link ensures that they will be folded in as if they were in this top level testplan spreadsheet, and the section numbers will correlate.

- Notice on some links (1.3, 2.14) that there are more than one link/type per particular requirement. This is because many requirements might take a combination of a directed test, coverpoints and assertions to fully cover all of that requirement's details.
- Not shown: It is possible that a single type item, such as a coverage point, might cover several requirements. In this case the same link name and type will be used in each of the requirements rows.
- Notice the last two columns (Owner, Priority). These are added for clarity and to record useful information associated with each requirement. They can be read into a tool, such as Questa's Verification Management tools. There they can be sorted on and viewed, and are stored in a UCIS compliant database, but they are not used inside the simulator. Here each requirement is given an owner so that they can sort by their name and see just the requirements that they are responsible for. The priority can then guide the order that they work on their requirements.
- Notice the last four rows are TBD (To be determined) as there was not enough time to flesh out these rows during this weeks meeting, so it was left for next week. This is common, but it is important to fill in your spreadsheet as you go along.

Click here for a copy of the WB SoC testplan and then click on file-save as in your browser to save this WBsoc.xml file. You should be able to open the downloaded file in Microsoft Excel.

Spreadsheet Automation & Reuse

Block Level And VIP Testplan Reuse

A common misconception is that functional coverage testbench components developed for a block level testbench can automatically be reused at higher levels of integration. In practice, only a sub-set of a block's functionality or an interface protocol is used once it has been integrated, this means that the functional coverage components have to be carefully examined if they are reused and coverage exclusions will have to be worked out. Any block level testplan will also have to be carefully examined to determine whether it is worth importing into the SoC testplan, or whether it is better to include an relevant entries individually.

Modern VIP often comes with its own testplan which is usually based on protocol coverage monitors or assertions implemented within the VIP. This sort of testplan can be reused as a hierarchical testplan element as described, however, it is very likely that only a sub-set of the protocol will actually be used by the SoC. This may mean adapting the coverage plan to ensure that the relevant modes of operation have been checked at the SoC level.

Automatic Testplan Creation

The power bubble diagrams and resulting section in the testplan spreadsheet shown above often lead to information being captured in IEEE 1801 UPF format. The system power states and power state transitions need to be verified. It helps to have a platform that allows the automatic creation of testplan from the UPF information. This power architecture testplan can be combined hierarchically with the rest of the System level testplan. Failing this the user will just have to manually create a testplan that contains all their power configurations. Typically it is a little of both, manual entries combined with automated entries.

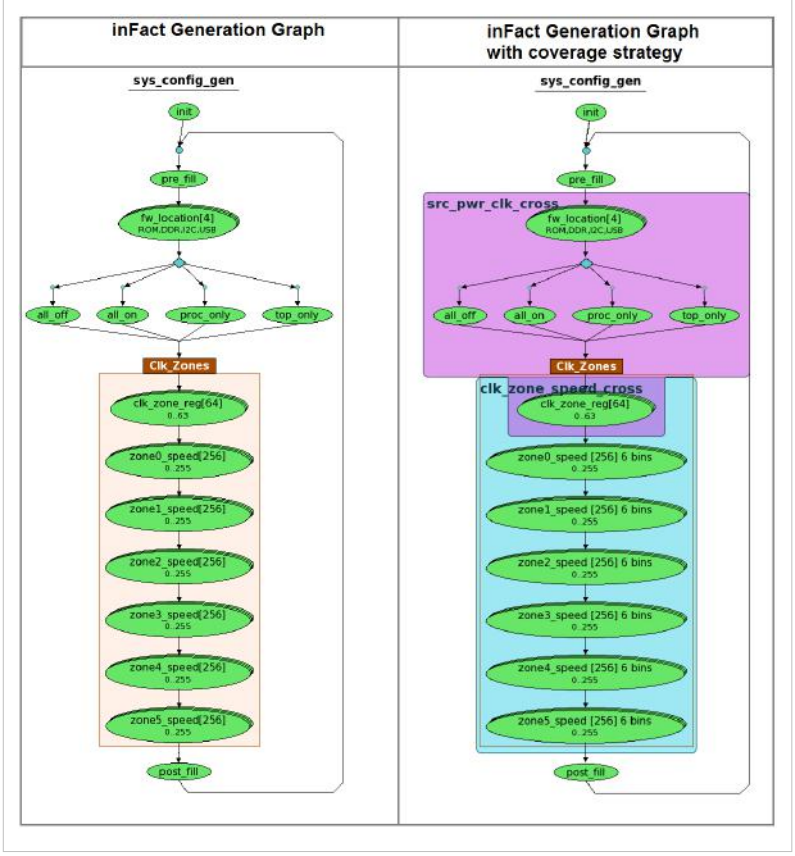
It is not uncommon for EDA tools to provide complete verification solutions including auto generated coverage models and assertions tuned to the needs of the SoC design. Usually this is for well understood and bounded issues such as power aware checkers, power state coverage models and clock domain crossing. Power and clocking verification tools such as Questa's Power Aware and Clock-Domain Crossing (CDC), can output a simulation readable testplan spreadsheet that can be hierarchically linked to other testplans and read into Questa. These auto generated coverage models testplans and their associated assertions can be linked to the higher level SoC testplan to get a measure of

completion.

Intelligent Testbench Automation

Another significant benefit of the top down approach to coverage model creation is the reuse of the use model diagrams. These flow-oriented diagrams can actually be generated by (or easily translated to input into) intelligent testbench automation tools such as Questa inFact. These tools capture and use a graph-based representation of your use model flows. This can be done at various levels. For example, a graph may represent the interface level (such as an i2c interface), a block level configuration (i2c register options), or even the top level configuration or traffic flow scenarios (coordinating the configuration and control of data from one peripheral or processor to others). Flow graphs can be connected hierarchically, thus allowing a full use model to be implemented in the testbench. This greatly decreases the development time needed to implement the stimulus generation parts of a testbench. Furthermore, these tools allow for coverage to be very easily defined by selecting various paths through the graph flow, and can then systematically run simulations to reach the coverage goal with the fewest iterations possible. This results in reaching coverage closure with 10-100x fewer simulation cycles than with random selection for typical coverage goals.

When this process is used, the testplan spreadsheet is still necessary. A simpler version with just categories and subcategories can be made up front, the graphs created, and then the spreadsheet can be updated as the coverage is added to the graphs. An example of the WB SoC top level graph is given below. On the left is the simple view of the graph which captures the legal configuration space options for generation. The right image is an example of one set of coverage goals overlayed on the graph (essentially two cross coverages denoted by the purple and blue regions).



Appendices

Requirements Writing Guidelines

Requirements Writing Guidelines

When creating a testplan, requirements to have a successful chip need to recorded in a useful, easy to digest manner. The following rules and guidelines will help to ensure this happens. It is a good idea for the verification team to compile a list such as this before starting the planning process and to divide them up into rules (must be followed) and suggestions (good ideas). In effect, this is defining the requirements for writing requirements.

- Don't rewrite anything that is already detailed in the source specifications, just reference the original document.
- Divide up the categories and subcategories so that each row is a single requirement. Don't write five requirements on one row. Write one requirement per row.
- Each requirement should be unique. Do not use ten requirements, when one will do. The gauging criteria is often whether or not it will easily link to a coverage element.
- Each requirement must be linked to some coverage element (test, covergroup, coverpoint, cross. assertion, code coverage, etc).
- Write each requirement at about the same level. Don't write one at a subsystem level, and the next at the AND-gate level. If you do have multi-level requirements, come up with a natural three to five level scale and define them clearly. Maybe use alpha numeric tags to distinguish levels, or maybe put each level in its own hierarchical testplan spreadsheet.
- A requirement is typically written in the positive, a description of what the design shall do. However, some requirements which place bounds on behavior are easier to write in the negative; in other words, a description of what the design shall not do.
- It is alright to add a requirement that is not going to be addressed by the verification process. It might be addressed by C-modeling, or by FPGA validation in a lab, or by some other means. Include it anyway, and add a column that states which process is being used on that requirement.
- Identify each requirement with both unique name and a unique number system.
- Requirements might be sub-divided into major categories like design requirements (about the design), verification requirements (about the run management), testbench requirements (about the testbench), software requirements (about the firmware), tool requirements (about Questa), library requirement (about which part of UVM that you will use), etc.
- Requirements should be ranked or prioritized. This may be a scale of 1-3, or could be a complex risk equation that takes in other parameters.
- Requirements should be ordered. Have categories and sub-categories, do not just have them entered sporadically, have some logical order.
- Each design requirement needs to be thought from all three verification perspectives, generation, checking and coverage. How will a situation be generated to exercise this requirement? What will check that it is right; an assertion, a scoreboard, or both? What sort of permutations will need to be covered, how many permutations? Some testplans will have three columns with a brief description of each of these.

- If a requirement is connected to some reused verification entity, it should be specified. A column for current or future reusability can be added and filled in.
- It is alright to have a requirement that is earmarked for a special directed test, but these should be not be widespread
- Testbenches often have levels of abstractions, often labeled with some layering (L1-3) or naming (configuration layer, traffic layer, etc.). A column that specifies each requirements abstraction layer can be added.
- Normal function and error handling function requirements should be separated, but do not leave out the error requirements.
- Some requirements might need to be ported across several environments, block, sub-system, system, lab, etc. This should be noted. A designated column can delineate this.
- Some requirements might be constraints in disguise. This is fine. Just note it.
- Some requirements are assertions in disguise; they have a cause and effect nature such as "after this, this will always happen". This is fine, just note it. It is wise to categorize assertions in some logical fashion, such as interface, internal, etc.
- Some requirements are configuration oriented. You may not need to specify each and every configuration, just point to where they are described in other documents, or describe each unique family of sequences. Divide them by how covergroups and coverpoints will capture them.
- Some requirements are sequence oriented, meaning they are configurations or traffic that need to be generated to stimulate the design. When you define sequence requirements it is best to start by defining each unique family of sequences by categories and sub categories, such as higher categories like configurations, traffic, interrupts, errors, etc., and then break those down into sub categories as needed. You do not need to specify each and every sequence, especially if they are already described in other documents, but make categorizes of them, each of which will lead to an interesting covergoups
- Some requirements might just be assumptions made, or required, that lead to easier implementation. This is fine.
- Scoreboard or assertion checking limitations should be included. Often the transfer function of a scoreboard or assertion might be too complex to be fully addressed. Specify what will be addressed, and what will not be addressed. For scoreboard, what actual transaction level elements will be checked?
- Another more advanced approach is to think covergroups and coverpoints up front and then to work backwards, reverse engineering and writing the requirements.

For the latest product information, call us or visit www.verificationacademy.com

Corporate Headquarters
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777
Phone: 503.685.7000
Fax: 503.685.1204

Sales and Product Information
Phone: 800.547.3000
sales_info@mentor.com

Silicon Valley
Mentor Graphics Corporation
46871 Bayside Parkway
Fremont, CA 94538 USA
Phone: 510.354.7400
Fax: 510.354.7467

North American Support Center
Phone: 800.547.4303

Europe
Mentor Graphics
Deutschland GmbH
Arnulfstrasse 201
80634 Munich
Germany
Phone: +49.89.57096.0
Fax: +49.89.57096.400

Pacific Rim
Mentor Graphics (Taiwan)
11F, No. 120, Section 2,
Gongdao 5th Road
HsinChu City 300,
Taiwan, ROC
Phone: 886.3.513.1000
Fax: 886.3.573.4734

Japan
Mentor Graphics Japan Co., Ltd.
Gotenyama Trust Tower
7-35, Kita-Shinagawa 4-chome
Shinagawa-Ku, Tokyo 140-0001
Japan
Phone: +81.3.5488.3033
Fax: +81.3.5488.3004

Mentor®
A Siemens Business