SYSTEMVERILOG FOR VERIFICATION
A Guide to Learning the Testbench Language Features

SYSTEMVERILOG FOR VERIFICATION
A Guide to Learning the Testbench Language Features

CHRIS SPEAR
Synopsys, Inc.

Springer

Chris Spear
Synopsys, Inc.
377 Simarano Drive
Marlboro, MA 01752

SystemVerilog for Verification:
A Guide to Learning the Testbench Language Features

Printed on acid-free paper.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

This book is dedicated to my wonderful wife Laura,
whose patience during this project was invaluable,
and my children, Allie and Tyler, who kept me laughing.

Contents

List of Examples

List of Figures

List of Tables

Foreword

When Verilog was first developed in the mid-1980s the mainstream level of design abstraction was on the move from the widely popular switch and gate levels up to the synthesizable RTL. By the late 1980s, RTL synthesis and simulation had revolutionized the front-end of the EDA industry.

The 1990s saw a tremendous expansion in the verification problem space and a corresponding growth of EDA tools to fill that space. The dominant languages that grew in this space were proprietary and specific to verification such as OpenVera and e, although some of the more advanced users did make the freely available C++ language their solution. Judging which of these languages was the best is very difficult, but one thing was clear, not only they were disjointed from Verilog but verification engineers were expected to learn multiple complex languages. Although some users of Verilog were using the language for writing testbenches (sometimes going across the PLI into the C language) it should be no surprise to anybody if I say that using Verilog for testbenches ran out of steam even before the 1990s started. Unfortunately, during the 1990s, Verilog stagnated as a language in its struggle to become an industry standard, and so made the problem worse.

Towards the end of the 1990s, a startup company called Co-Design broke through this stagnation and started the process of designing and implementing the language we now know as the SystemVerilog industry standard. The vision of SystemVerilog was to first expand on the abstract capabilities of synthesizable code, and then to significantly add all the features known to be necessary for verification, while keeping the new standard a strict superset of the previous Verilog standards. The benefits of having a single language and a single coherent run-time environment cannot be expressed enough. For instance, the user benefits greatly from ease of use, and the vendor can take

many significant new opportunities to achieve much higher levels of simulation performance.

There is no doubt that the powerful enhancements put into SystemVerilog have also made the overall language quite complex. If you have a working knowledge of Verilog, and are overwhelmed by the complex verification constructs now in SystemVerilog and the books that teach you the advanced lessons, this is the book for you. The author has spent a large amount of time making language mistakes that you need not repeat. Through the process of correcting his mistakes with his vast verification experience, the author has compiled over three hundred examples showing you the correct ways of coding and solving problems, so that you can learn by example and be led gently into the productive world of SystemVerilog.

PHIL MOORBY
New England, 2006

Preface

What is this book about?

This book is the first one you should read to learn the SystemVerilog verification language constructs. It describes how the language works and includes many examples on how to build a basic coverage-driven, constrained-random layered testbench using Object Oriented Programming (OOP). The book has many guidelines on building testbenches, which help show why you want to use classes, randomization, and functional coverage. Once you have learned the language, pick up some of the methodology books listed in the References section for more information on building a testbench.

Who should read this book?

If you create testbenches, you need this book. If you have only written tests using Verilog or VHDL and want to learn SystemVerilog, this book shows you how to move up to the new language features. Vera and Specman users can learn how one language can be used for both design and verification. You may have tried to read the SystemVerilog Language Reference Manual (LRM) but found it loaded with syntax but no guidelines on which construct to choose.

I wrote this book because, like many of my customers, I spent much of my career using procedural languages such as C and Verilog to write tests, and had to relearn everything when OOP verification languages came along. I made all the typical mistakes, and wrote this book so you won't have to repeat them.

Before reading this book, you should be comfortable with Verilog-1995. Knowledge of Verilog-2001, SystemVerilog design constructs, or SystemVerilog Assertions is not required.

Why was SystemVerilog created?

In the late 1990s, the Verilog Hardware Description Language (HDL) became the most widely used language for describing hardware for simulation and synthesis. However, the first two versions standardized by the IEEE (1364-1995 and 1364-2001) had only simple constructs for creating tests. As design sizes outgrew the verification capabilities of the language, commercial Hardware Verification Languages (HVL) such as OpenVera and *e* were created. Companies that did not want to pay for these tools instead spent hundreds of man-years creating their own custom tools.

This productivity crisis (along with a similar one on the design side) led to the creation of Accellera, a consortium of EDA companies and users who wanted to create the next generation of Verilog. The donation of the Open-Vera language formed the basis for the HVL features of SystemVerilog. Accellera's goal was met in November 2005 with the adoption of the IEEE standard P1800-2005 for SystemVerilog, IEEE (2005).

Importance of a unified language

Verification is generally viewed as a fundamentally different activity from design. This split has led to the development of narrowly focused language for verification and to the bifurcation of engineers into two largely independent disciplines. This specialization has created substantial bottlenecks in terms of communication between the two groups. SystemVerilog addresses this issue with its capabilities for both camps. Neither team has to give up any capabilities it needs to be successful, but the unification of both syntax and semantics of design and verification tools improves communication. For example, while a design engineer may not be able to write an object-oriented testbench environment, it is fairly straightforward to read such a test and understand what is happening, enabling both the design and verification engineers to work together to identify and fix problems. Likewise, a designer understands the inner workings of his or her block, and is the best person to write assertions about it, but a verification engineer may have a broader view needed to create assertions between blocks.

Another advantage of including the design, testbench, and assertion constructs in a single language is that the testbench has easy access to all parts of the environment without requiring specialized APIs. The value of an HVL is its ability to create high-level, flexible tests, not its loop constructs or declaration style. SystemVerilog is based on the Verilog constructs that engineers have used for decades.

Importance of methodology

There is a difference between learning the syntax of a language and learning how to use a tool. This book focuses on techniques for verification using constrained-random tests that use functional coverage to measure progress and direct the verification. As the chapters unfold, language and methodology features are shown side by side. For more on methodology, see Bergeron et al. (2006).

The most valuable benefit of SystemVerilog is that it allows the user to construct reliable, repeatable verification environments, in a consistent syntax, that can be used across multiple projects.

Comparing SystemVerilog and SystemC for high-level design

Now that SystemVerilog incorporates Object Oriented Programming, dynamic threads, and interprocess communication, it can be used for system design. When talking about the applications for SystemVerilog, the IEEE standard mentions architectural modeling before design, assertions, and test. SystemC can also be used for architectural modeling. There are several major differences between SystemC and SystemVerilog:

- SystemVerilog provides one modeling language. You do not have to learn C++ and the Standard Template Library to create your models
- SystemVerilog simplifies top-down design. You can create your system models in SystemVerilog and then refine each block to the next lower level. The original system-level models can be reused as reference models.
- Software developers want a free or low-cost hardware simulator that is fast. You can create high-performance transaction-level models in both SystemC and SystemVerilog. SystemVerilog simulators require a license that a software developer may not want to pay for. SystemC can be free, but only if all your models are available in SystemC.

Overview of the book

The SystemVerilog language includes features for design, verification, assertions, and more. This book focuses on the constructs used to verify a design. There are many ways to solve a problem using SystemVerilog. This book explains the trade-offs between alternative solutions.

Chapter 1, **Verification Guidelines**, presents verification techniques to serve as a foundation for learning and using the SystemVerilog language.

These guidelines emphasize coverage-driven random testing in a layered testbench environment.

Chapter 2, **Data Types**, covers the new SystemVerilog data types such as arrays, structures, enumerated types, and packed variables.

Chapter 3, **Procedural Statements and Routines**, shows the new procedural statements and improvements for tasks and functions.

Chapter 4, **Basic OOP**, is an introduction to Object Oriented Programming, explaining how to build classes, construct objects, and use handles.

Chapter 5, **Connecting the Testbench and Design**, shows the new SystemVerilog verification constructs, such as program blocks, interfaces, and clocking blocks, and how they are used to build your testbench and connect it to the design under test.

Chapter 6, **Randomization**, shows you how to use SystemVerilog's constrained-random stimulus generation, including many techniques and examples.

Chapter 7, **Threads and Interprocess Communication**, shows how to create multiple threads in your testbench, use interprocess communication to exchange data between these threads and synchronize them.

Chapter 8, **Advanced OOP and Guidelines**, shows how to build a layered testbench with OOP's inheritance so that the components can be shared by all tests.

Chapter 9, **Functional Coverage**, explains the different types of coverage and how you can use functional coverage to measure your progress as you follow a verification plan.

Chapter 10, **Advanced Interfaces**, shows how to use virtual interfaces to simplify your testbench code, connect to multiple design configurations, and create interfaces with procedural code so your testbench and design can work at a higher level of abstraction.

Icons used in this book

Table 1.   Book icons

	Shows verification methodology to guide your usage of SystemVerilog testbench features
	Shows common coding mistakes

Final comments

If you would like more information on SystemVerilog and Verification, you can find many resources at

```
http://chris.spear.net/systemverilog
```

This site has the source code for the examples in this book. All of the examples have been verified with Synopsys' Chronologic VCS 2005.06 and 2006.06. The SystemVerilog Language Reference Manual covers hundreds of new features. I have concentrated on constructs useful for verification and implemented in VCS. It is better to have verified examples than to show all language features and thus risk having incorrect code. Speaking of mistakes, if you think you have found a mistake, please check my web site for the Errata page. If you are the first to find any mistake in a chapter, I will send you a free book.

CHRIS SPEAR
Synopsys, Inc.

Acknowledgments

United Airlines always had a quiet place to work and plenty of snacks. "Chicken or pasta?"

Lastly, a big thanks to Jay Mcinerney for his brash pronoun usage.

Chapter 1

Verification Guidelines

"Some believed we lacked the programming language to describe your perfect world..."
(The Matrix, 1999)

1.1 Introduction

Imagine that you are given the job of building a house for someone. Where should you begin? Do you start by choosing doors and windows, picking out paint and carpet colors, or selecting bathroom fixtures? Of course not! First you must consider how the owners will use the space, and their budget, so you can decide what type of house to build. Questions you should consider are; do they enjoy cooking and want a high-end kitchen, or will they prefer watching movies in the home theater room and eating takeout pizza? Do they want a home office or extra bedrooms? Or does their budget limit them to a basic house?

Before you start to learn details of the SystemVerilog language, you need to understand how you plan to verify your particular design and how this influences the testbench structure. Just as all houses have kitchens, bedrooms, and bathrooms, all testbenches share some common structure of stimulus generation and response checking. This chapter introduces a set of guidelines and coding styles for designing and constructing a testbench that meets your particular needs. These techniques use some of the same concepts as shown in the *Verification Methodology Manual for SystemVerilog* (VMM), Bergeron et al. (2006), but without the base classes.

The most important principle you can learn as a verification engineer is: "Bugs are good." Don't shy away from finding the next bug, do not hesitate to ring a bell each time you uncover one, and furthermore, always keep track of each bug found. The entire project team assumes there are bugs in the design, so each bug found before tape-out is one fewer that ends up in the customer's hands. You need to be as devious as possible, twisting and torturing the design to extract all possible bugs now, while they are still easy to fix. Don't let the designers steal all the glory — without your craft and cunning, the design might never work!

This book assumes you already know the Verilog language and want to learn the SystemVerilog Hardware Verification Language (HVL). Some of

the typical features of an HVL that distinguish it from a Hardware Description Language such as Verilog or VHDL are

- Constrained-random stimulus generation
- Functional coverage
- Higher-level structures, especially Object Oriented Programming
- Multi-threading and interprocess communication
- Support for HDL types such as Verilog's 4-state values
- Tight integration with event-simulator for control of the design

There are many other useful features, but these allow you to create test-benches at a higher level of abstraction than you are able to achieve with an HDL or a programming language such as C.

1.2 The Verification Process

What is the goal of verification? If you answered, "Finding bugs," you are only partly correct. The goal of hardware design is to create a device that performs a particular task, such as a DVD player, network router, or radar signal processor, based on a design specification. Your purpose as a verification engineer is to make sure the device can accomplish that task successfully — that is, the design is an accurate representation of the specification. Bugs are what you get when there is a discrepancy. The behavior of the device when used outside of its original purpose is not your responsibility although you want to know where those boundaries lie.

The process of verification parallels the design creation process. A designer reads the hardware specification for a block, interprets the human language description, and creates the corresponding logic in a machine-readable form, usually RTL code. To do this, he or she needs to understand the input format, the transformation function, and the format of the output. There is always ambiguity in this interpretation, perhaps because of ambiguities in the original document, missing details, or conflicting descriptions. As a verification engineer, you must also read the hardware specification, create the verification plan, and then follow it to build tests showing the RTL code correctly implements the features.

By having more than one person perform the same interpretation, you have added redundancy to the design process. As the verification engineer, your job is to read the same hardware specifications and make an independent assessment of what they mean. Your tests then exercise the RTL to show that it matches your interpretation.

What types of bugs are lurking in the design? The easiest ones to detect are at the block level, in modules created by a single person. Did the ALU correctly add two numbers? Did every bus transaction successfully complete? Did all the packets make it through a portion of a network switch? It is almost trivial to write directed tests to find these bugs as they are contained entirely within one block of the design.

After the block level, the next place to look for discrepancies is at boundaries between blocks. Interesting problems arise when two or more designers read the same description yet have different interpretations. For a given protocol, what signals change and when? The first designer builds a bus driver with one view of the specification, while a second builds a receiver with a slightly different view. Your job is to find the disputed areas of logic and maybe even help reconcile these two different views.

To simulate a single design block, you need to create tests that generate stimuli from all the surrounding blocks — a difficult chore. The benefit is that these low-level simulations run very fast. However, you may find bugs in both the design and testbench as the latter will have a great deal of code to provide stimuli from the missing blocks. As you start to integrate design blocks, they can stimulate each other, reducing your workload. These multiple block simulations may uncover more bugs, but they also run slower.

At the highest level of the DUT, the entire system is tested, but the simulation performance is greatly reduced. Your tests should strive to have all blocks performing interesting activities concurrently. All I/O ports are active, processors are crunching data, and caches are being refilled. With all this action, data alignment and timing bugs are sure to occur.

At this level you are able to run sophisticated tests that have the DUT executing multiple operations concurrently so that as many blocks as possible are active. What happens if an MP3 player is playing music and the user tries to download new music from the host computer? Then, during the download, the user presses several of the buttons on the player? You know that when the real device is being used, someone is going to do all this, so why not try it out before it is built? This testing makes the difference between a product that is seen as easy to use and one that locks up over and over.

Once you have verified that the DUT performs its designated functions correctly, you need to see how it operates when there are errors. Can the design handle a partial transaction, or one with corrupted data or control fields? Just trying to enumerate all the possible problems is difficult, not to mention how the design should recover from them. Error injection and handling can be the most challenging part of verification.

As the design abstraction gets higher, so does the verification challenge. You can show that individual cells flow through the blocks of an ATM router

correctly, but what if there are streams of different priority? Which cell should be chosen next is not always obvious at the highest level. You may have to analyze the statistics from thousands of cells to see if the aggregate behavior is correct.

One last point: you can never prove there are no bugs left, so you need to constantly come up with new verification tactics.

1.3 The Verification Plan

The verification plan is closely tied to the hardware specification and contains a description of what features need to be exercised and the techniques to be used. These steps may include directed or random testing, assertions, HW/SW co-verification, emulation, formal proofs, and use of verification IP. For a more complete discussion on verification see Bergeron (2006).

1.4 The Verification Methodology Manual

This book in your hands draws heavily upon the VMM that has its roots in a methodology developed by Janick Bergeron and others at Qualis Design. They started with industry standard practices and refined them based on experience on many projects. VMM's techniques were originally developed for use with the OpenVera language and were extended in 2005 for SystemVerilog. VMM and its predecessor, the Reference Verification Methodology for Vera, have been used successfully to verify a wide range of hardware designs, from networking devices to processors. This book uses many of the same concepts.

So why doesn't this book teach you VMM? Like any advanced tool, VMM was designed for use by an expert user, and excels on difficult problems. Are you in charge of verifying a 10 million gate design with many communication protocols, complex error handling, and a library of IP? VMM is the right tool for the job. But if you are working on smaller modules, with a single protocol, you may not need such a robust methodology. Just remember that your block is part of a larger system; VMM is still useful to promote reuse. The cost of verification goes beyond your immediate project.

If you are new to verification, have little experience with Object Oriented Programming, or are unfamiliar with constrained-random tests, the techniques in this book might be the right path to choose. Once you are familiar with them, you will find the VMM to be an easy step up.

The biggest thing missing from this book, when compared with the VMM, is the set of base classes for data, environment, and utilities for managing log

files and interprocess communication. These are all very useful, but are outside the scope of a book on the SystemVerilog language.

1.5 Basic Testbench Functionality

The purpose of a testbench is to determine the correctness of the design under test (DUT). This is accomplished by the following steps.

- Generate stimulus
- Apply stimulus to the DUT
- Capture the response
- Check for correctness
- Measure progress against the overall verification goals

Some steps are accomplished automatically by the testbench, while others are manually determined by you. The methodology you choose determines how the above steps are carried out.
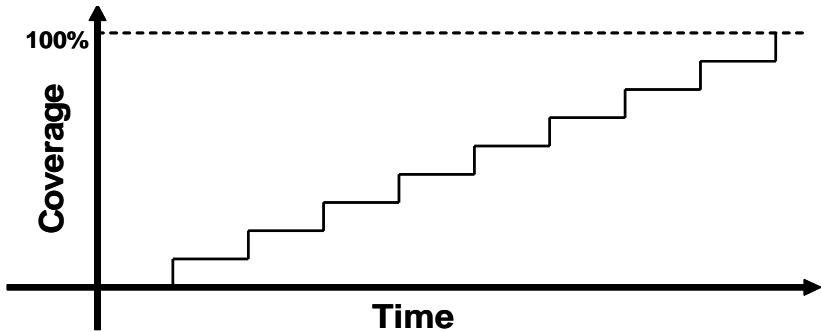
1.6 Directed Testing

Traditionally, when faced with the task of verifying the correctness of a design, you may have used directed tests. Using this approach, you look at the hardware specification and write a verification plan with a list of tests, each of which concentrated on a set of related features. Armed with this plan, you write stimulus vectors that exercise these features in the DUT. You then simulate the DUT with these vectors and manually review the resulting log files and waveforms to make sure the design does what you expect. Once the test works correctly, you check off the test in the verification plan and move to the next.

This incremental approach makes steady progress, always popular with managers who want to see a project making headway. It also produces almost immediate results, since little infrastructure is needed when you are guiding the creation of every stimulus vector. Given enough time and staffing, directed testing is sufficient to verify many designs.

Figure 1-1 shows how directed tests incrementally cover the features in the verification plan. Each test is targeted at a very specific set of design elements. Given enough time, you can write all the tests need for 100% coverage of the entire verification plan.

Figure 1-1 Directed test progress

What if you do not have the necessary time or resources to carry out the directed testing approach? As you can see, while you may always be making forward progress, the slope remains the same. When the design complexity doubles, it takes twice as long to complete or requires twice as many people. Neither of these situations is desirable. You need a methodology that finds bugs faster in order to reach the goal of 100% coverage.

Figure 1-2 Directed test coverage
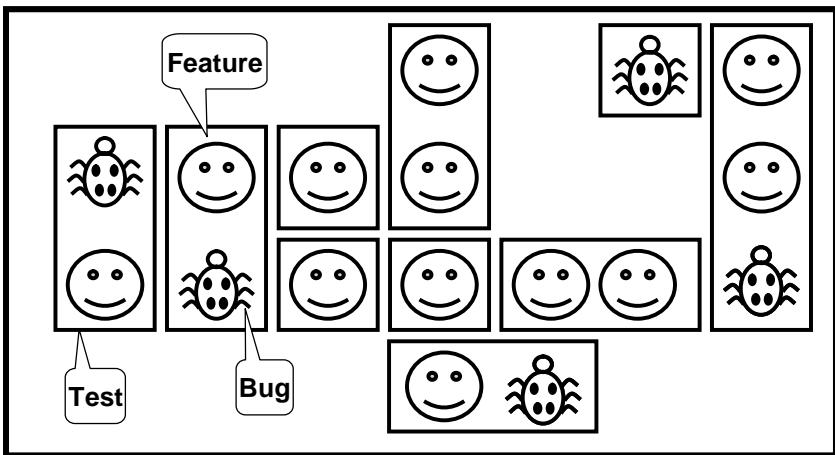
Figure 1-2 shows the total design space and the features that get covered by directed testcases. In this space are many features, some of which have bugs. You need to write tests that cover all the features and find the bugs.

1.7 Methodology Basics
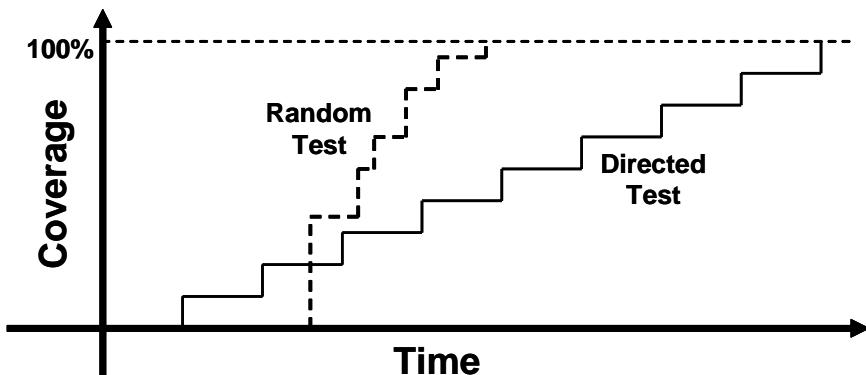
This book uses the following principles.

- Constrained-random stimulus
- Functional coverage
- Layered testbench using transactors
- Common testbench for all tests
- Test-specific code kept separate from testbench

All these principles are related. Random stimulus is crucial for exercising complex designs. A directed test finds the bugs you expect to be in the design, while a random test can find bugs you never anticipated. When using random stimulus, you need functional coverage to measure verification progress. Furthermore, once you start using automatically generated stimulus, you need an automated way to predict the results, generally a scoreboard or reference model. Building the testbench infrastructure, including self-prediction, takes a significant amount of work. A layered testbench helps you control the complexity by breaking the problem into manageable pieces. Transactors provide a useful pattern for building these pieces. With appropriate planning, you can build a testbench infrastructure that can be shared by all tests and does not have to be continually modified. You just need to leave "hooks" where the tests can perform certain actions such as shaping the stimulus and injecting disturbances. Conversely, code specific to a single test must be kept separate from the testbench so it does not complicate the infrastructure.

Building this style of testbench takes longer than a traditional directed testbench, especially the self-checking portions, causing a delay before the first test can be run. This gap can cause a manager to panic, so make this effort part of your schedule. In Figure 1-3, you can see the initial delay before the first random test runs.

Figure 1-3 Constrained-random test progress

While this up-front work may seem daunting, the payback is high. Every test you create shares this common testbench, as opposed to directed tests where each is written from scratch. Each random test contains a few dozen lines of code to constrain the stimulus in a certain direction and cause any desired exceptions, such as creating a protocol violation. The result is that your single constrained-random testbench is now finding bugs faster than the many directed ones.

As the rate of discovery begins to drop off, you can create new random constraints to explore new areas. The last few bugs may only be found with directed tests, but the vast majority of bugs will be found with random tests.

1.8 Constrained-Random Stimulus

While you want the simulator to generate the stimulus, you don't want totally random values. You use the SystemVerilog language to describe the format of the stimulus ("address is 32-bits, opcode is X, Y, or Z, length < 32 bytes"), and the simulator picks values that meet the constraints. Constraining the random values to become relevant stimuli is covered in Chapter 6. These values are sent into the design, and also into a high-level model that predicts what the result should be. The design's actual output is compared with the predicted output.

Figure 1-4 shows the coverage for constrained-random tests over the total design space. First, notice that a random test often covers a wider space than a directed one. This extra coverage may overlap other tests, or may explore new areas that you did not anticipate. If these new areas find a bug, you are in luck! If the new area is not legal, you need to write more constraints to keep away. Lastly, you may still have to write a few directed tests to find cases not covered by any other constrained-random tests.
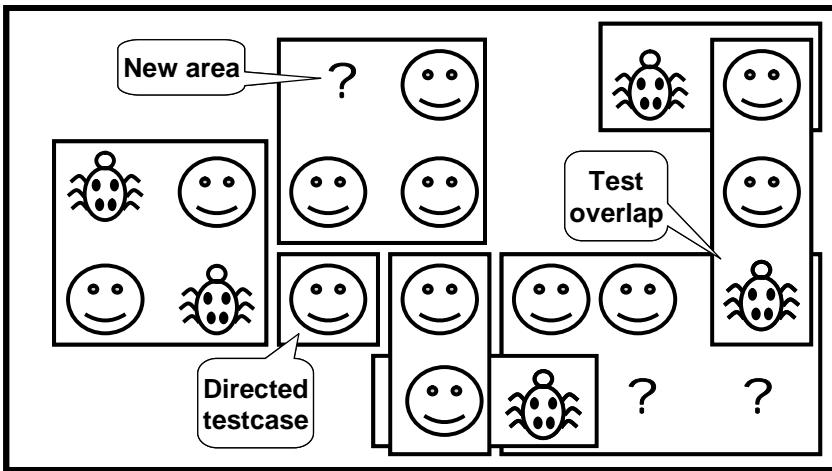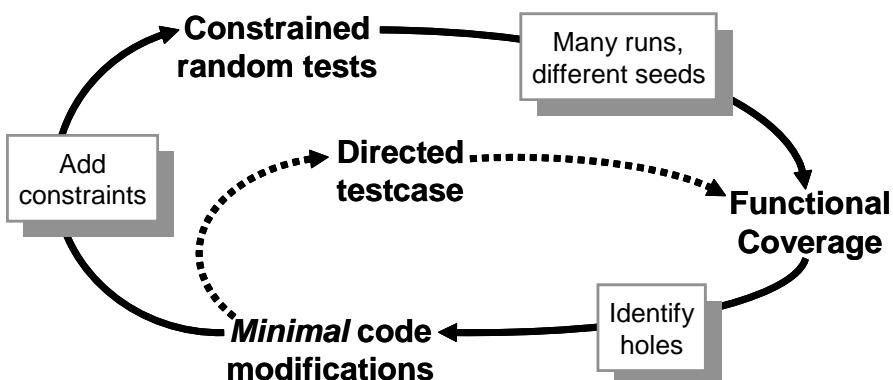
Figure 1-4 Constrained-random test coverage

Figure 1-5 shows the paths to achieve complete coverage. Start at the upper left with basic constrained-random tests. Run them with many different seeds. When you look at the functional coverage reports, find the holes, where there are gaps. Now you make minimal code changes, perhaps with new constraints, or injecting errors or delays into the DUT. Spend most of your time in this outer loop, only writing directed tests for the few features that are very unlikely to be reached by random tests.

Figure 1-5 Coverage convergence

1.9 What Should You Randomize?

When you think of randomizing the stimulus to a design, the first thing that you might think of is the data fields. This stimulus is the easiest to create – just call `$random`. The problem is that this gives a very low payback in terms of bugs found. The primary types of bugs found with random data are data path errors, perhaps with bit-level mistakes. You need to find bugs in the control logic.

You need to think broadly about all design input, such as the following.

- Device configuration
- Environment configuration
- Input data
- Protocol exceptions
- Delays
- Errors and violations

These are discussed in the following sections.

1.9.1 Device and environment configuration

What is the most common reason why bugs are missed during testing of the RTL design? Not enough different configurations are tried. Most tests just use the design as it comes out of reset, or apply a fixed set of initialization vectors to put it into a known state. This is like testing a PC's operating system right after it has been installed, without any applications installed. Of course the performance is fine, and there aren't any crashes.

In a real world environment, the DUT's configuration becomes more random the longer it is in use. For example, I helped a company verify a time-division multiplexor switch that had 2000 input channels and 12 output channels. The verification engineer said, "These channels could be mapped to various configurations on the other side. Each input could be used as a single channel, or further divided into multiple channels. The tricky part is that although a few standard ways of breaking it down are used most of the time, any combination of breakdowns is legal, leaving a huge set of possible customer configurations."

To test this device, the engineer had to write several dozen lines of directed testbench code to configure each channel. As a result, she was never able to try configurations with more than a handful of channels. Together, we wrote a testbench that randomized the parameters for a single channel, and

then put this in a loop to configure all the switch's channels. Now she had confidence that her tests would uncover configuration-related bugs that would have been missed before.

In the real world, your device operates in an environment containing other components. When you are verifying the DUT, it is connected to a testbench that mimics this environment. You should randomize the entire environment configuration, including the length of the simulation, number of devices, and how they are configured. Of course you need to create constraints to make sure the configuration is legal.

In another Synopsys customer example, a company was creating an I/O switch chip that connected multiple PCI buses to an internal memory bus. At the start of simulation they randomly chose the number of PCI buses (1–4), the number of devices on each bus (1–8) and the parameters for each device (master or slave, CSR addresses, etc.). They kept track of the tested combinations using functional coverage so that they could be sure that they had covered almost every possible one.

Other environment parameters include test length, error injection rates, delay modes, etc. See Bergeron (2006) for more examples.

1.9.2 Input data

When you read about random stimulus, you probably thought of taking a transaction such as a bus write or ATM cell and filling the data fields with random values. Actually this approach is fairly straightforward as long as you carefully prepare your transaction classes as shown in Chapters 4 and 8. You need to anticipate any layered protocols and error injection, plus scoreboarding and functional coverage.

1.9.3 Protocol exceptions, errors, and violations

There are few things more frustrating than when a device such as a PC or cell phone locks up. Many times, the only cure is to shut it down and restart. Chances are that deep inside the product there is a piece of logic that experienced some sort of error condition and could not recover, and thus stopped the device from working correctly.

How can you prevent this from happening to the hardware you are building? If something can go wrong in the real hardware, you should try to simulate it. Look at all the errors that can occur. What happens if a bus transaction does not complete? If an invalid operation is encountered? Does the design specification state that two signals are mutually exclusive? Drive them both and make sure the device continues to operate.

Just as you are trying to provoke the hardware with ill-formed commands, you should also try to catch these occurrences. For example, recall those mutually exclusive signals. You should add checker code to look for these violations. Your code should at least print a warning message when this occurs, and preferably generate an error and wind down the test. It is frustrating to spend hours tracking back through code trying to find the root of a malfunction, especially when you could have caught it close to the source with a simple assertion. See Vijayaraghavan (2005) for more guidelines on writing assertions in your testbench and design code. Just make sure that you can disable the code that stops simulation on error so that you can easily test error handling.

1.9.4 Delays and synchronization

How fast should your testbench send in stimulus? Always use constrained-random delays to help catch protocol bugs. A test that uses the shortest delays runs the fastest, but it won't create all possible stimulus. You can create a testbench that talks to another block at the fastest rate, but subtle bugs are often revealed when intermittent delays are introduced.

A block may function correctly for all possible permutations of stimulus from a single interface, but subtle errors may occur when data is flowing into multiple inputs. Try to coordinate the various drivers so they can communicate at different relative timing. What if the inputs arrive at the fastest possible rate, but the output is being throttled back to a slower rate? What if stimulus arrives at multiple inputs concurrently? What if it is staggered with different delays? Use functional coverage as discussed in Chapter 9 to measure what combinations have been randomly generated.

1.9.5 Parallel random testing

How should you run the tests? A directed test has a testbench that produces a unique set of stimulus and response vectors. To change the stimulus, you need to change the test. A random test consists of the testbench code plus a random seed. If you run the same test 50 times, each with a unique seed, you will get 50 different sets of stimuli. Running with multiple seeds broadens the coverage of your test and leverages your work.

You need to choose a unique seed for each simulation. Some people use the time of day, but that can still cause duplicates. What if you are using a batch queuing system across a CPU farm and tell it to start 10 jobs at midnight? Multiple jobs could start at the same time but on different computers, and will thus get the same random seed, and run the same stimulus. You

should blend in the processor name to the seed. If your CPU farm includes multiprocessor machines, you could have two jobs start running at midnight with the same seed, so you should also throw in the process ID. Now all jobs get unique seeds.

You need to plan how to organize your files to handle multiple simulations. Each job creates a set of output files such as log files and functional coverage data. You can run each job in a different directory, or you can try to give a unique name to each file.

1.10 Functional Coverage
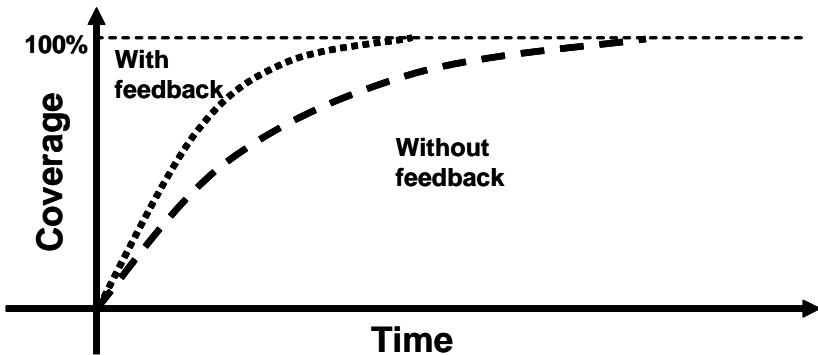
The previous sections have shown how to create stimuli that can randomly walk through the entire space of possible inputs. With this approach, your testbench visits some areas often, but takes too long to reach all possible states. Unreachable states will never be visited, even given unlimited simulation time. You need to measure what has been verified in order to check off items in your verification plan.

The process of measuring and using functional coverage consists of several steps. First, you add code to the testbench to monitor the stimulus going into the device, and its reaction and response, to determine what functionality has been exercised. Next, the data from one or more simulations is combined into a report. Lastly, you need to analyze the results and determine how to create new stimulus to reach untested conditions and logic. Chapter 9 describes functional coverage in SystemVerilog.

1.10.1 Feedback from functional coverage to stimulus

A random test evolves using feedback. The initial test can be run with many different seeds, creating many unique input sequences. Eventually the test, even with a new seed, is less likely to generate stimulus that reaches areas of the design space. As the functional coverage asymptotically approaches its limit, you need to change the test to find new approaches to reach uncovered areas of the design. This is known as "coverage-driven verification."

Figure 1-6 Test progress with and without feedback

What if your testbench were smart enough to do this for you? In a previous job, I wrote a test that generated every bus transaction for a processor, and additionally fired every bus terminator (Success, Parity error, Retry) in every cycle. This was before HVLs, so I wrote a long set of directed tests and spent days lining up the terminator code to fire at just the right cycles. After much hand analysis I declared success – 100% coverage. Then the processor's timing changed slightly! Now I had to reanalyze the test and change the stimuli.

A more productive testing strategy uses random transactions and terminators. The longer you run it, the higher the coverage. As a bonus, the test could be made flexible enough to create valid stimuli even if the design's timing changed. You could add a feedback loop that would look at the stimulus created so far (generated all write cycles yet?) and change the constraint weights (drop write weight to zero). This improvement would greatly reduce the time needed to get to full coverage, with little manual intervention.
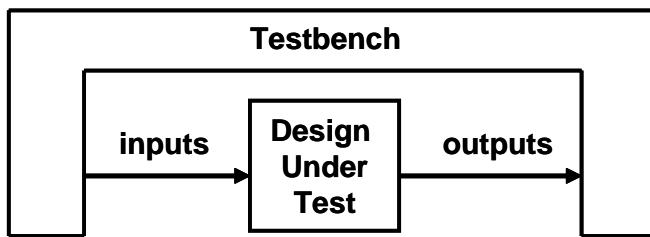
However, this is not a typical situation because of the trivial feedback from functional coverage to the stimulus. In a real design, how should you change the stimulus to reach a desired design state? There are no easy answers, so dynamic feedback is rarely used for constrained-random stimulus. Manual feedback is used in coverage-driven verification.

Feedback is used in formal analysis tools such as Magellan (Synopsys 2003). It analyzes a design to find all the unique, reachable states. Then it runs a short simulation to see how many states are visited. Lastly, it searches from the state machine to the design inputs to calculate the stimulus needed to reach any remaining states and then Magellan applies this to the DUT.
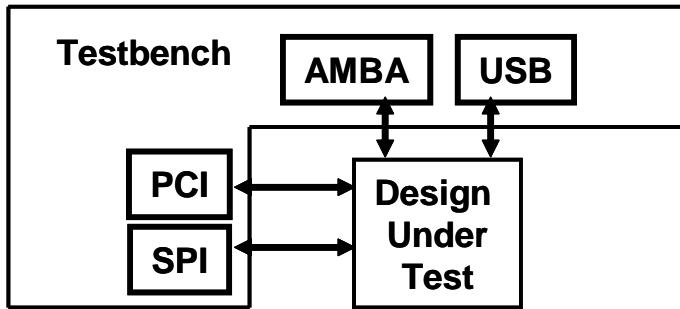
1.11 Testbench Components

In simulation, the testbench wraps around the DUT, just as a hardware tester connects to a physical chip. Both the testbench and tester provide stimulus and capture responses. The difference between them is that your testbench needs to work over a wide range of levels of abstraction, creating transactions and sequences, which are eventually transformed into bit vectors. A tester just works at the bit level.

Figure 1-7 The testbench — design environment

What goes into that testbench block? It is made of many bus functional models (BFM), that you can think of as testbench components — to the DUT they look like real components, but are part of the testbench, not RTL. If the real device connects to AMBA, USB, PCI, and SPI buses, you have to build equivalent components in your testbench that can generate stimulus and check the response. These are not detailed, synthesizable models but instead, high-level transactors that obey the protocol, but execute more quickly. If you are prototyping using FPGAs or emulation, the BFMs do need to be synthesizable.

Figure 1-8 Testbench components

1.12 Layered Testbench

A key concept for any modern verification methodology is the layered testbench. While this process may seem to make the testbench more complex, it actually helps to make your task easier by dividing the code into smaller pieces that can be developed separately. Don't try to write a single routine that can randomly generate all types of stimulus, both legal and illegal, plus inject errors with a multi-layer protocol. The routine quickly becomes complex and unmaintainable.

1.12.1 Flat testbench

When you first learned Verilog and started writing tests, they probably looked like the following low-level code that does a simplified APB (AMBA Peripheral Bus) Write. (VHDL users may have written similar code.)

Example 1-1 Driving the APB pins

```
module test(PAddr, PWrite, PSel, PRData, Rst, clk);
// Port declarations omitted...

initial begin
  // Drive reset
  Rst <= 0;
  #100 Rst <= 1;

  // Drive Control bus
  @(posedge clk)
  PAddr  <= 16'h50;
  PWData <= 32'h50;
  PWrite <= 1'b1;
  PSel   <= 1'b1;

  // Toggle PEnable
  @(posedge clk)
    PEnable <= 1'b1;
  @(posedge clk)
    PEnable <= 1'b0;

  // Check the result
  if (top.mem.memory[16'h50] == 32'h50)
    $display("Success");
  else
    $display("Error, wrong value in memory");
  $finish;
  end
endmodule
```

After a few days of writing code like this, you probably realized that it is very repetitive, so you created tasks for common operations such as a bus write, as shown in Example 1-2.

Example 1-2 A task to drive the APB pins

```
task write(reg [15:0] addr, reg [31:0] data);
  // Drive Control bus
  @(posedge clk)
  PAddr  <= addr;
  PWData <= data;
  PWrite <= 1'b1;
  PSel   <= 1'b1;

  // Toggle Penable
  @(posedge clk)
    PEnable <= 1'b1;
  @(posedge clk)
    PEnable <= 1'b0;
endtask
```

Now your testbench became simpler.

Example 1-3 Low-level Verilog test
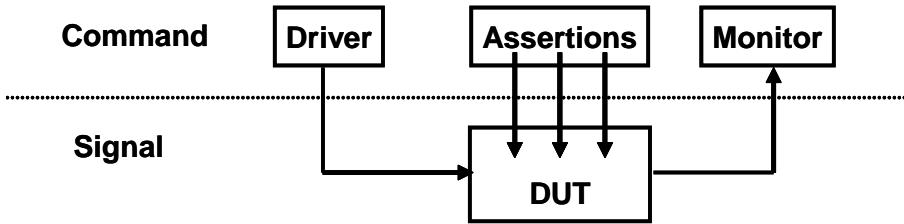
```
module test(PAddr, PWrite, PSel, PRData, Rst, clk);
  // Port declarations omitted...

  initial begin
    reset();                    // Reset the device
    write(16'h50, 32'h50);   // Write data into memory

    // Check the result
    if (top.mem.memory[16'h50] == 32'h50)
      $display("Success");
    else
      $display("Error, wrong value in memory");
    $finish;
  end
endmodule
```

By taking the common actions, such as reset, bus reads and writes, and putting them in a routine, you became more efficient and made fewer mistakes. This creation of the physical and command layers is the first step to a layered testbench.

1.12.2 The signal and command layers

Figure 1-9 shows the lower layers of a testbench.
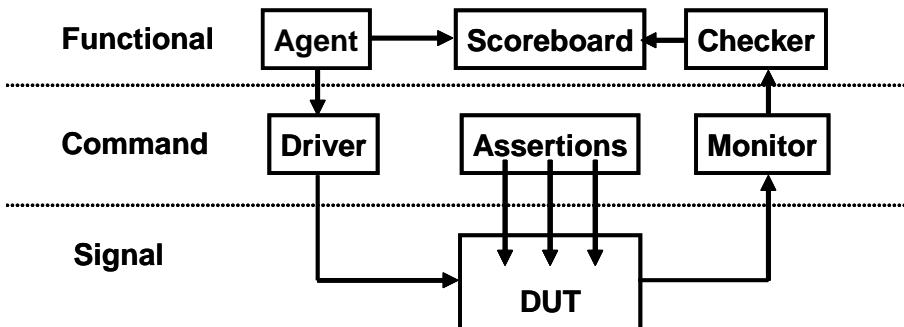
Figure 1-9 Signal and command layers

At the bottom is the signal layer that contains the design under test and the signals that connect it to the testbench.

The next level up is the command layer. The DUT's inputs are driven by the driver that runs single commands such as bus read or write. The DUT's output drives the monitor that takes signal transitions and groups them together into commands. Assertions also cross the command/signal layer, as they look at individual signals but look for changes across an entire command.

1.12.3 The functional layer
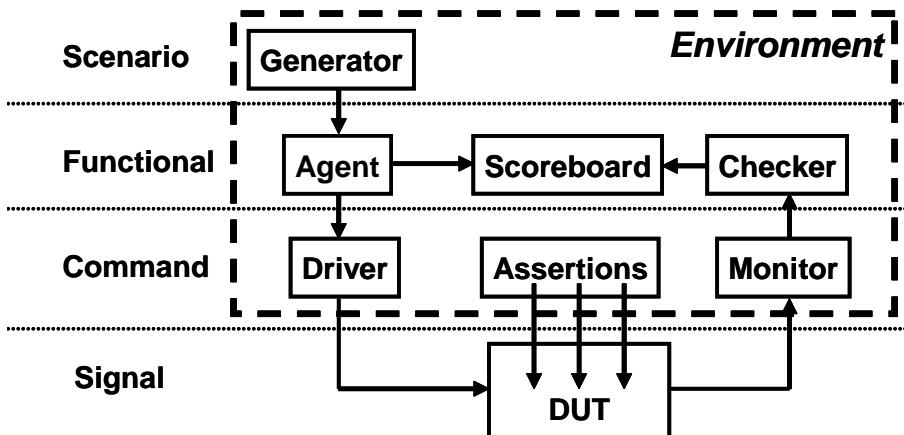
The functional layer feeds the command layer. The agent block (called the transactor in the VMM) receives higher-level transactions such as DMA read or write and breaks them into individual commands. These commands are also sent to the scoreboard that predicts the results of the transaction. The checker compares the commands from the monitor with those in the scoreboard.

Figure 1-10 Testbench with functional layer

1.12.4 The scenario layer

The functional layer is driven by the generator in the scenario layer. What is a scenario? Remember that your job as a verification engineer is to make sure that this device accomplishes its intended task. An example device is an MP3 player that can concurrently play music from its storage, download new music from a host, and respond to input from the user, such as volume and track controls. Each of these operations is a scenario. Downloading a music file takes several steps, such as control register reads and writes to set up the operation, multiple DMA writes to transfer the song, and then another group of reads and writes. The scenario layer of your testbench orchestrates all these steps with constrained-random values for parameters such as track size and memory location.

Figure 1-11 Testbench with scenario layer

The blocks in the testbench environment (inside the dashed line) are written at the start of development. During the project they may evolve and you may add functionality, but these blocks should not change for individual tests. This is done by leaving "hooks" in the code so that a test can change the behavior of these blocks without having to rewrite them. You create these hooks with callbacks (section 8.7) and factory patterns (section 8.3).

1.12.5 The test layer and functional coverage
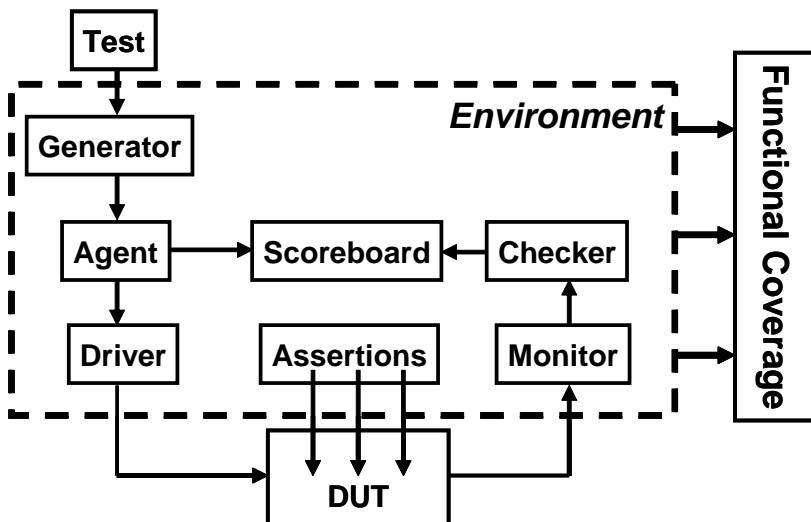
You are now at the top of the testbench, the test layer, as shown in Figure 1-12. Design bugs that occur between DUT blocks are harder to find as they involve multiple people reading and interpreting multiple specifications.

This top-level test is a conductor who does not play any musical instrument, but instead guides the efforts of others. The test contains the constraints to create the stimulus.

Functional coverage measures the progress of all tests in fullfilling the requirements in the verification plan. The functional coverage code changes through the project as the various criteria complete. Because it is constantly being modified, it is not part of the environment.

You can create a "directed test" in a constrained-random environment. Simply insert a section of directed test case into the middle of or in parallel with a random sequence. The directed code performs the work you want, but the random "background noise" may cause a bug to become visible, perhaps in an unanticipated block.

Figure 1-12 Full testbench with all layers

Do you need all these layers in your testbench? The answer depends on what your DUT looks like. A complicated design requires a sophisticated testbench. You always need the test layer. For a simple design, the scenario layer may be so simple that you can merge it with the agent. When estimating the effort to test a design, don't count the number of gates; count the number of designers. Every time you add another person to the team, you increase the chance of different interpretations of the specifications.

You may need more layers. If your DUT has several protocol layers, each should get its own layer in the testbench environment. For example, if you have TCP traffic that is wrapped in IP and sent in Ethernet packets, consider

using three separate layers for generation and checking. Better yet, use existing verification components.

One last note about the above diagram: It shows some of the possible connections between blocks, but your testbench may have a different set. The test may need to reach down to the driver layer to force physical errors. These are just guidelines – let your needs guide what you create.
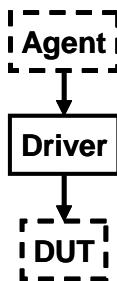
1.13 Building a Layered Testbench

Now it is time to take the previous diagrams and learn how to map the components into SystemVerilog constructs.

1.13.1 Creating a simple driver

First, take a closer look at one of the blocks, the driver.

Figure 1-13 Connections for the driver

The driver receives commands from the agent, may inject errors or add delays, and then breaks the command down into individual signal changes such as bus requests, handshakes, etc. The general term for such a testbench block is a "transactor," which, at its core, is a loop:

Example 1-4 Basic transactor code

```
task run();
  done = 0;
  while (!done) begin
    // Get the next transaction
    // Make transformations
    // Send out transactions
  end
endtask
```

Chapter 4 presents basic OOP and how to create an object that includes the routines and data for a transactor. Another example of a transactor is the agent. It might break apart a complex transaction such as a DMA read into multiple bus commands. Also in Chapter 4, you will see how to build an object that contains the data and routines that make up a command. These objects are sent between transactors using SystemVerilog mailboxes. In Chapter 7, you will learn about many ways to exchange data between the different layers and to synchronize the transactors.

1.14 Simulation Environment Phases

Up until now you have been learning what parts make up the environment. When do these parts execute? You want to clearly define the phases to coordinate the testbench so all the code for a project works together. The three primary phases are Build, Run, and Wrap-up. Each is divided into smaller steps.

The Build phase is divided into the following steps:

■ Generate Configuration: randomize the configuration of the DUT and the surrounding environment.

■ Build Environment: Allocate and connect the testbench components based on the configuration. A testbench component is one that only exists in the testbench, as opposed to physical components in the design that are built with RTL code. For example, if the configuration chose three bus drivers, the testbench would allocate and initialize them in this step.

■ Reset the DUT.

■ Configure the DUT based on generated configuration from the first step.

The Run phase is where the test actually runs. It has the following steps:

■ Start environment: run the testbench components such as BFMs and stimulus generators.

■ Run the test: start the test and then wait for it to complete. It is easy to tell when a directed test has completed, but doing so can be complex for a random test. You can use the testbench layers as a guide. Starting from the top, wait for a layer to drain all the inputs from the previous layer (if any), wait for the current layer to become idle, and then

wait for the next lower layer. You should also use time-out checkers to ensure that the DUT or testbench does not lock up.

The Wrap-up phase has two steps:

- Sweep: After the lowest layer completes, you need to wait for the final transactions to drain out of the DUT.
- Report: Once the DUT is idle, sweep the testbench for lost data. Sometimes the scoreboard holds transactions that never came out, perhaps because they were dropped by the DUT. With this information you can create the final report on whether the test passed or failed. If it failed, be sure to delete any functional coverage data, as it may not be correct.

As shown in the layer diagram, Figure 1-12, the test starts the environment. This runs each of the steps. More details can be found in Chapter 8.

1.15 Maximum Code Reuse

To verify a complex device with hundreds of features, you have to write hundreds of directed tests. If you use constrained-random stimulus, you will write far fewer tests. Instead, the real work is put into constructing the testbench, which contains all the lower testbench layers, scenario, functional, and command. This testbench code is used by all the tests, so it should remain generic.

These guidelines may seem to recommend a sophisticated testbench, but remember that every line that you put into it can eliminate a line in every single test. If you create a few dozen tests, there is a high payback. Keep this in mind when you read Chapter 8.

1.16 Testbench Performance

If this is the first time you have seen this methodology, you probably have some qualms about how it works compared to directed testing. A common objection is testbench performance. A directed test often simulates in less than a second, while constrained-random tests will wander around through the state space for minutes or hours. The problem with this argument is that it ignores a real verification bottleneck – the time required by you to create a test. You may be able to hand-craft a directed test in a day, and debug it and manually verify the results by hand in another day or two. The actual simulation run-time is dwarfed by the amount of your time that you invested.

There are several steps to creating a constrained-random test. The most significant is building the layered testbench, including the self-checking portion. The benefit of this work is shared by all tests, so it is well worth the effort. The next step is creating the stimulus specific to a goal in the verification plan. You may be crafting random constraints, or devious ways of injecting errors or protocol violations. Building one of these may take more time than making several directed tests, but the payoff will be much higher. A constrained-random test that tries thousands of different protocol variations is worth more than the handful of directed tests that could have been created in the same amount of time.

The third step in constrained-random testing is functional coverage. This task starts with the creation of a strong verification plan with clear goals that can be easily measured. Next you need to create the SystemVerilog code that instruments the environment and gathers the data. Lastly, and most importantly, you need to analyze the results to determine if you have met the goals, and if not, how you should modify the tests.

1.17 Conclusion

The continuous growth in complexity of electronic designs requires a modern, systematic, and automated approach to creating testbenches. The cost of fixing a bug grows by 10x as a project moves from each step of specification to RTL coding, gate synthesis, fabrication, and finally into the user's hands. Directed tests only test one feature at a time and cannot create the complex stimulus and configurations that the device would be subjected to in the real world. To produce robust designs, you must use constrained-random stimulus combined with functional coverage to create the widest possible range of stimulus.

Chapter 2

Data Types

2.1 Introduction

SystemVerilog offers many improved data structures compared with Verilog. Some of these were created for designers but are also useful for testbenches. In this chapter you will learn about the data structures most useful for verification.

SystemVerilog introduces new data types with the following benefits.

- ■ Two-state: better performance, reduced memory usage
- ■ Queues, dynamic and associative arrays and automatic storage: reduced memory usage, built-in support for searching and sorting
- ■ Unions and packed structures: allows multiple views of the same data
- ■ Classes and structures: support for abstract data structures
- ■ Strings: built-in string support
- ■ Enumerated types: code is easier to write and understand

2.2 Built-in Data Types

Verilog-1995 has two basic data types: variables (`reg`) and nets, that hold four-state values: 0, 1, Z, and X. RTL code uses variables to store combinational and sequential values. Variables can be unsigned single or multi-bit (`reg [7:0] m`), signed 32-bit variables (`integer`), unsigned 64-bit variables (`time`), and floating point numbers (`real`). Variables can be grouped together into arrays that have a fixed size. All storage is static, meaning that all variables are alive for the entire simulation and routines cannot use a stack to hold arguments and local values. A net is used to connect parts of a design such as gate primitives and module instances. Nets come in many flavors, but most designers use scalar and vector wires to connect together the ports of design blocks.

SystemVerilog adds many new data types to help both hardware designers and verification engineers.

2.2.1 The logic type

The one thing in Verilog that always leaves new users scratching their heads is the difference between a `reg` and a `wire`. When driving a port,

which should you use? How about when you are connecting blocks? System-Verilog improves the classic **reg** data type so that it can be driven by continuous assignments, gates and modules, in addition to being a variable. It is given the new name **logic** so that it does not look like a register declaration. The one limitation is that a **logic** variable cannot be driven by multiple drivers such as when you are modeling a bidirectional bus. In this case, the variable needs to be a net-type such as **wire**.

Example 2-1 shows the SystemVerilog **logic** type.

Example 2-1 Using the logic type

```
module logic_data_type(input logic rst_h);
  parameter CYCLE = 20;
  logic q, q_l, d, clk, rst_l;
  initial begin
    clk <= 0;                        // Procedural assignment
    forever #(CYCLE/2) clk = ~clk;
  end

  assign rst_l = ~rst_h;       // Continuous assignment
  not n1(q_l, q);              // q_l is driven by gate
  my_dff d1(q, d, clk, rst_l); // d is driven by module

endmodule
```

2.2.2   Two-state types

SystemVerilog introduces several two-state data types to improve simulator performance and reduce memory usage, over four-state types. The simplest type is the **bit**, which is always unsigned. There are four signed types: **byte**, **shortint**, **int**, and **longint**.

Example 2-2 Signed data types

```
bit b;          // 2-state, single-bit
bit [31:0] b32; // 2-state, 32-bit unsigned integer
int i;          // 2-state, 32-bit signed integer
byte b8;        // 2-state, 8-bit signed integer
shortint s;     // 2-state, 16-bit signed integer
longint l;      // 2-state, 64-bit signed integer
```

You might be tempted to use types such as **byte** to replace more verbose declarations such as **logic [7:0]**. Hardware designers should be careful as these new types are signed variables, so a **byte** variable can only count up to 127, not

the 255 you may expect. (It has the range -128 to +127.) You could use **byte unsigned**, but that is more verbose than just **bit [7:0]**. Signed variables may cause unexpected results with randomization, as discussed in Chapter 6.

Be careful connecting two-state variables to the design under test, especially its outputs. If the hardware tries to drive an X or Z, these values are converted to a two-state value, and your testbench code may never know. Don't try to remember if they are converted to 0 or 1; instead, always check for propagation of unknown values. Use the **$isunknown** operator that returns 1 if any bit of the expression is X or Z.

Example 2-3 Checking for four-state values

```
if ($isunknown(iport)
  $display("@%0d: 4-state value detected on input port",
           $time, iport);
```

2.3   Fixed-Size Arrays

SystemVerilog offers several flavors of arrays beyond the single-dimension, fixed-size Verilog-1995 arrays. Many enhancements have been made to these classic arrays.

2.3.1   Declaring and initializing fixed-size array

Verilog requires that the low and high array limits must be given in the declaration. Almost all arrays use a low index of 0, so SystemVerilog lets you use the shortcut of just giving the array size, similar to C:

Example 2-4 Declaring fixed-size arrays

```
int lo_hi[0:15];        // 16 ints [0]..[15]
int c_style[16];        // 16 ints [0]..[15]
```

You can create multidimensional fixed-size arrays by specifying the dimensions after the variable name. This is an unpacked array; packed arrays are shown later. The following creates several two-dimensional arrays of integers, 8 entries by 4, and sets the last entry to 1. Multidimensional arrays were introduced in Verilog-2001, but the compact declarations are new.

Example 2-5 Declaring and using multidimensional arrays

```
int array2 [0:7][0:3];  // Verbose declaration
int array3 [8][4];      // Compact declaration
array2[7][3] = 1;       // Set last array element
```

SystemVerilog stores each element on a longword (32-bit) boundary. So a **byte**, **shortint**, and **int** are all stored in a single longword, while a **long-int** is stored in two longwords. (Simulators frequently store four-state types such as **logic** and **integer** in two or more longwords.)

Example 2-6 Unpacked array declarations

```
bit [7:0] b_unpacked[3];    // Unpacked
```

The unpacked array of bytes, **b_unpacked,** is stored in three longwords.

Figure 2-1 Unpacked array storage

2.3.2   The array literal

You can initialize an array using an array literal that is an apostrophe and curly braces.[1] You can set some or all elements at once. You can replicate values by putting a count before the curly braces.

Example 2-7 Initializing an array

```
int ascend[4] = '{0,1,2,3}; // Initialize 4 elements
int decend[5];
int md[2][3] = '{'{0,1,2}, '{3,4,5}};

descend = '{4,3,2,1,0};      // Set 5 elements
descend[0:2] = '{5,6,7};     // Set first 3 elements
ascend = '{4{8}};            // Four values of 8
```

2.3.3   Basic array operations — **for** and **foreach**

The most common way to manipulate an array is with a **for** or **foreach** loop. In Example 2-8, the variable **i** is declared local to the **for** loop. The SystemVerilog function **$size** returns the size of the array. In the **foreach** statement, you specify the array name and an index in square brackets, and SystemVerilog automatically steps through all the elements of the array. The index variable is local to the loop.

[1.] VCS X-2005.06 follows the original Accellera standard for array literals and uses just the curly braces with no leading apostrophe. VCS will change to the IEEE standard in an upcoming release.

Example 2-8 Using arrays with `for` and `foreach` loops

```
initial begin
  bit [31:0] src[5], dst[5];
  for (int i=0; i<$size(src); i++)
    src[i] = i;
  foreach (dst[j])
    dst[j] = src[j] * 2;  // dst doubles src values
end
```

Note that the syntax of the **foreach** statement for multidimensional arrays may not be what you expected! Instead of listing each subscript in separate square brackets – `[i][j]` – they are combined with a comma – `[i,j]`.

Example 2-9 Initialize and step through a multidimensional array

```
initial begin
  $display("Initial value:");
  foreach (md[i,j])    // Yes, this is the right syntax
    $display("md[%0d][%0d] = %0d", i, j, md[i][j]);

  $display("New value:");
  md = '{{9, 8, 7}, 3{5}};  // Replicate last 3 values
  foreach (md[i,j])    // Yes, this is the right syntax
  $display("md[%0d][%0d] = %0d", i, j, md[i][j]);
end
```

Example 2-9 produces the following output:

Example 2-10 Output from printing multidimensional array values

```
Initial value:
md[0][0] = 0
md[0][1] = 1
md[0][2] = 2
md[1][0] = 3
md[1][1] = 4
md[1][2] = 5

New value:
md[0][0] = 9
md[0][1] = 8
md[0][2] = 7
md[1][0] = 5
md[1][1] = 5
md[1][2] = 5
```

2.3.4   Basic array operations – copy and compare

You can perform aggregate compare and copy of arrays without loops. (An aggregate operation works on the entire array as opposed to working on just an individual element.) Comparisons are limited to just equality and inequality. Example 2-11 shows several examples of compares. The **?   :** operator is a mini **if**-statement. Here it is choosing between two strings.

Example 2-11 Array copy and compare operations

```
initial begin
  bit [31:0] src[5] = '{0,1,2,3,4},
             dst[5] = '{5,4,3,2,1};

  // Aggregate compare the two arrays
  if (src==dst)
    $display("src == dst");
  else
    $display("src != dst");

  // Aggregate copy all src values to dst
  dst = src;

  // Change just one element
  src[0] = 5;

  // Are all values equal (no!)
  $display("src %s dst", (src == dst) ? "==" : "!=");

  // Are last elements 1-4 equal (yes!)
  $display("src[1:4] %s dst[1:4]",
          (src[1:4] == dst[1:4]) ? "==" : "!=");
end
```

You cannot perform aggregate arithmetic operations such as addition on arrays. Instead, you can use loops. For logical operations such as **xor**, you have to either use a loop or use packed arrays as described below.

2.3.5   Bit and word subscripts, together at last

A common annoyance in Verilog-1995 is that you cannot use word and bit subscripts together. Verilog-2001 removes this restriction for fixed-size arrays. Example 2-12 prints the first array element (binary 101), its lowest bit (1), and the next two higher bits (binary 10).

Example 2-12 Using word and bit subscripts together

```
initial begin
  bit [31:0] src[5] = `{5{5}};
  $displayb(src[0],,          // 'b101 or 'd5
           src[0][0],,        // 'b1
           src[0][2:1]);      // 'b10
end
```

While this change is not new to SystemVerilog, many users may not know about this useful improvement in Verilog-2001.

2.3.6   Packed arrays

For some data types, you may want both to access the entire value and also divide it into smaller elements. For example, you may have a 32-bit register that sometimes you want to treat as four 8-bit values and at other times as a single, unsigned value. A SystemVerilog packed array is treated as both an array and a single value. It is stored as a contiguous set of bits with no unused space, unlike an unpacked array.

2.3.7   Packed array examples
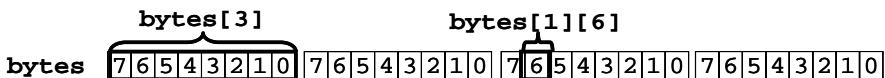
The packed bit and word dimensions are specified as part of the type, before the variable name. These dimensions must be specified in the **[lo:hi]** format. The variable **bytes** is a packed array of four bytes, which are stored in a single longword.

Example 2-13 Packed array declaration and usage

```
bit [3:0] [7:0] bytes;   // 4 bytes packed into 32-bits
bytes = 32'hdead_beef;
$displayh(bytes,,          // Show all 32-bits
          bytes[3],        // most significant byte "de"
          bytes[3][7]);    // most significant bit  "1"
```

Figure 2-2 Packed array layout

You can mix packed and unpacked dimensions. You may want to make an array that represents a memory that can be accessed as bits, bytes, or long-

words. In Example 2-14, **barray** is an unpacked array of three packed elements.
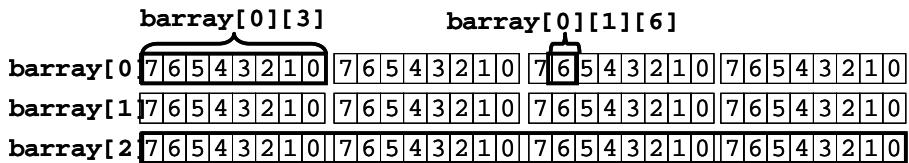
Example 2-14 Declaration for mixed packed/unpacked array

```
bit [3:0] [7:0] barray [3];    // Packed: 3x32-bit
barray[0] = 32'h0123_4567;
barray[0][3] = 8'h01;
barray[0][1][6] = 1'b1;
```

The variable **bytes** is a packed array of four bytes, which are stored in a single longword. **barray** is an array of three of these elements.

Figure 2-3 Packed arrays

With a single subscript, you get a longword of data, **barray[2]**. With two subscripts, you get a byte of data, **barray[0][3]**. With three subscripts, you can access a single bit, **barray[0][1][6]**. Note that because one dimension is specified after the name, **barray[3]**, that dimension is unpacked, so you always need to use at least one subscript.

2.3.8   Choosing between packed and unpacked arrays

Which should you choose — packed or unpacked array? A packed array is handy if you need to convert to and from scalars. For example, you might need to reference a memory as a byte or as a longword. The above array **barray** can handle this requirement. Only fixed-size arrays can be packed, not dynamic arrays, associative arrays, or queues (as shown below).

If you need to wait for a change in an array, you have to use a packed array. Perhaps your testbench might need to wake up when a memory changes value, so you want to use the **@** operator. But this is only legal with scalar values and packed arrays. Using the earlier examples, you can block on the variable **lw**, and **barray[0]**, but not the entire array **barray** unless you expand it: **@(barray[0] or barray[1] or barray[2])**.

2.4   Dynamic Arrays

The basic Verilog array type shown so far is known as a fixed-size array, as its size is set at compile time. But what if you do not know the size of the

array until run-time? You may choose the number of transactions randomly between 1000 and 100,000, but you do not want to use a fixed-size array that would be half empty. SystemVerilog provides a dynamic array that can be allocated and resized during simulation.

A dynamic array is declared with empty word subscripts `[]`. This means that you do not want to give an array size at compile time; instead, you specify it at run-time. The array is initially empty, so you must call the `new[]` opera- tor to allocate space, passing in the number of entries in the square brackets. If you pass the name of an array to the `new[]` operator, the values are copied into the new elements.

Example 2-15 Using dynamic arrays

```
int dyn[], d2[];            // Empty dynamic arrays

initial begin
  dyn = new[5];             // Allocate 5 elements
  foreach (dyn[j])
    dyn[j] = j;             // Initialize the elements
  d2 = dyn;                 // Copy a dynamic array
  d2[0] = 5;                // Modify the copy
  $display(dyn[0],d2[0]);   // See both values (0 & 5)
  dyn = new[20](dyn);       // Expand and copy
  dyn = new[100];           // Allocate 100 new integers
                            // Old values are lost
  dyn.delete;               // Delete all elements
end
```

The `$size` function returns the size of a fixed-size or dynamic array. Dynamic arrays have several specialized routines, such as `delete` and `size`. The latter function returns the size, but does not work with fixed-size arrays.

If you want to declare a constant array of values but do not want to bother counting the number of elements, use a dynamic array with an array literal. In Example 2-16 there are 9 masks for 8 bits, but you should let SystemVerilog count them, rather than making a fixed-size array and accidently choosing the wrong size of 8.

Example 2-16 Using a dynamic array for an uncounted list

```
bit [7:0] mask[] = '{8'b0000_0000, 8'b0000_0001,
                     8'b0000_0011, 8'b0000_0111,
                     8'b0000_1111, 8'b0001_1111,
                     8'b0011_1111, 8'b0111_1111,
                     8'b1111_1111};
```

You can make assignments between fixed-size and dynamic arrays as long as they have the same base type such as **int**. You can assign a dynamic array to a fixed array as long as they have the same number of elements.

When you copy a fixed-size array to a dynamic array, SystemVerilog calls **new[]** constructor to allocate space, and then copies the values.

2.5   Queues

SystemVerilog introduces a new data type, the queue, which provides easy searching and sorting in a structure that is as fast as a fixed-size array but as versatile as a linked list.

Like a dynamic array, queues can grow and shrink, but with a queue you can easily add and remove elements anywhere. Example 2-17 adds and removes values from a queue.

Example 2-17 Queue operations

```
int j = 1,
    b[$] = {3,4},
    q[$] = {0,2,5};  // {0,2,5}        Initial queue

initial begin
  q.insert(1, j);    // {0,1,2,5}      Insert 1 before 2
  q.insert(3, b);    // {0,1,2,3,4,5} Insert whole q.
  q.delete(1);       // {0,2,3,4,5}    Delete elem. #1

  // The rest of these are fast
  q.push_front(6);   // {6,0,2,3,4,5} Insert at front
  j = q.pop_back;    // {6,0,2,3,4}   j = 5
  q.push_back(8);    // {6,0,2,3,4,8} Insert at back
  j = q.pop_front;   // {0,2,3,4,8}   j = 6
  foreach (q[i])
    $display(q[i]);
end
```

When you create a queue, SystemVerilog actually allocates extra space so you can quickly add extra elements. Note that you do not need to call the **new[]** operator for a queue. If you add enough elements so that the queue runs out of space, SystemVerilog automatically allocates additional space. As a result, you can grow and shrink a queue without the performance penalty of a dynamic array.

It is very efficient to push and pop elements from the front and back of a queue, taking a fixed amount of time no matter how large the queue. Adding

and deleting elements in the middle is slower, especially for larger queues, as SystemVerilog has to shift up to half of the elements.
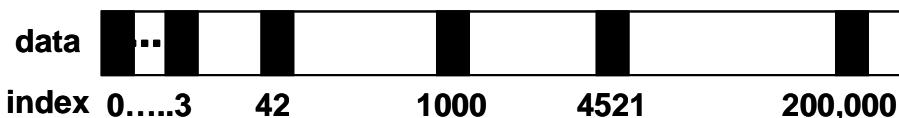
You can copy the contents of a fixed or dynamic array into a queue.

2.6   Associative Arrays

Dynamic arrays are good if you want to occasionally create a large array, but what if you want something really huge? Perhaps you are modeling a processor that has a multi-gigabyte address range. During a typical test, the processor may only touch a few hundred or thousand memory locations containing executable code and data, so allocating and initializing gigabytes of storage is wasteful.

SystemVerilog offers associative arrays that store entries in a sparse matrix. This means that while you can address a very large address space, SystemVerilog only allocates memory for an element when you write to it. In the following picture, the associative array holds the values 0:3, 42, 1000, 4521, and 200,000. The memory used to store these is far less than would be needed to store a fixed or dynamic array with 200,000 entries.

Figure 2-4 Associative array

Example 2-18 shows declaring, initializing, and stepping through an associative array. These arrays are declared with wildcard syntax `[*]`. You can remember the syntax by thinking that the array can be indexed with almost any integer.

Example 2-18 Declaring, initializing, and using associative arrays

```
initial begin
  logic [63:0] assoc[*], idx = 1;

  // Initialize widely scattered values
  repeat (64) begin
    assoc[idx] = idx;
    idx = idx << 1;
  end

  // Step through all index values with foreach
  foreach (assoc[i])
    $display("assoc[%h] = %h", i, assoc[i]);

  // Step through all index values with functions
  if (assoc.first(idx))
    begin                            // Get first index
    do
      $display("assoc[%h]=%h", idx, assoc[idx]);
    while (assoc.next(idx));  // Get next index
    end

  // Find and delete the first element
  assoc.first(idx);
  assoc.delete(idx);
end
```

Example 2-18 has the associative array, **assoc**, with very scattered elements: 1, 2, 4, 8, 16, etc. A simple **for** loop cannot step through them; you need to use a **foreach** loop, or, if you wanted finer control, you could use the **first** and **next** functions in a **do**...**while** loop. These functions modify the index argument, and return 0 or 1 depending on whether any elements are left in the array.

Associative arrays can also be addressed with a string index, similar to Perl's hash arrays. Example 2-19 reads name/value pairs from a file into an associative array. If you try to read from an element that has not been allocated yet, SystemVerilog returns a 0 for two-state types or X for 4-state types. You can use the function **exists** to check if an element exists, as shown below. Strings are explained in section 2.14.

Example 2-19 Using an associative array with a string index

```
/*
Input file looks like:
  42   min_address
  1492 max_address
*/

int switch[string], min_address, max_address;
initial begin
  int i, r, file;
  string s;
  file = $fopen("switch.txt", "r");
  while (! $feof(file)) begin
    r = $fscanf(file, "%d %s", i, s);
    switch[s] = i;
  end
  $fclose(file);

  // Get the min address, default is 0
  mid_address = switch["min_address"];

  // Get the max address, default = 1000
  if (switch.exists("max_address"))
    max_address = switch["max_address"];
  else
    max_address = 1000;
end
```

An associative array can be stored by the simulator as a tree. This additional overhead is acceptable when you need to store arrays with widely separated index values, such as packets indexed with 32-bit addresses or 64-bit data values.

2.7   Linked Lists

SystemVerilog provides a linked list data-structure that is analogous to the STL (Standard Template Library) List container. The container is defined as a parameterized class, meaning that it can be customized to hold data of any type.

Now that you know there is a linked list in SystemVerilog, avoid using it. C++ programmers might be familiar with the STL version, but SystemVerilog's queues are more efficient and easier to use.

2.8   Array Methods

There are many array methods that you can use on any unpacked array types: fixed, dynamic, queue, and associative. These routines can be as simple as giving the current array size to sorting the elements.

2.8.1   Array reduction methods

A basic array reduction method takes an array and reduces it to a scalar. The most common reduction method is **sum**, which adds together all the values in an array. Be careful of SystemVerilog's rules for handling the width of operations. By default, if you add the values of a single-bit array, the result is a single bit. But if you store the result in a 32-bit variable or compare it to a 32-bit variable, SystemVerilog uses 32-bits when adding up the values.

Example 2-20 Creating the sum of an array

```
bit on[10];  // Array of single bits
int summ;

initial begin
  foreach (on[i])
    on[i] = i;   // on[i] gets 0 or 1

  // Print the single-bit sum
  $display("on.sum = %0d", on.sum); // on.sum = 1

  // Sum the values using 32-bits as summ is 32-bits
  summ = on.sum;
  $display("summ = %0d", summ);      // summ = 5

  // Compare the sum to a 32-bit value
  if (on.sum >=32'd5)                // True
    $display("sum has 5 or more 1's");
  end
```

Other array reduction methods are **product**, **and**, **or**, and **xor**.

2.8.2   Array locator methods

What is the largest value in an array? Does an array contain a certain value? The array locator methods find data in an unpacked array. These methods always return a queue.

Example 2-22 uses a fixed-size array, **f[6]**, a dynamic array, **d[ ]**, and a queue, **q[$]**.The **min** and **max** functions find the smallest and largest elements in an array. Note that they return a queue, not a scalar as you might expect. These methods also work for associative arrays. The **unique** method returns a queue of the unique values from the array — duplicate values are not included.

Example 2-21 Array locator methods: min, max, unique

```
int f[6] = `{1,6,2,6,8,6};
int q[$] = `{1,3,5,7}, tq[$];

tq = q.min;                              // {1}
tq = q.max;                              // {7}
tq = f.unique;                           // {1,6,2,8}
```

You could search through an array using a **foreach** loop, but SystemVerilog can do this in one operation with a locator method. The **with** expression tells SystemVerilog how to perform the search.

Example 2-22 Array locator methods: find

```
int d[] = `{9,1,8,3,4,4}, tq[$];

// Find all elements greater than 3
tq = d.find with (item > 3);        // {9,8,4,4}
// Equivalent code
tq.delete;
foreach (d[i])
  if (d[i] > 3)
    tq.push_back(d[i]);

tq = d.find_index with (item > 3);   // {0,2,4}
tq = d.find_first with (item > 99); // {} - none found
tq = d.find_first_index with (item==8); // {2} d[2]=8
tq = d.find_last with (item==4);        // {4}
tq = d.find_last_index with (item==4);  // {6} d[6]=4
```

When you combine an array reduction such as **sum** using the **with** clause, the results may surprise you. In Example 2-23, the **sum** operator is adding up the number of times that the expression is true. For the first statement in Example 2-23, there are two array elements that are greater than 7 (9 and 8) so **count** is set to 2. Note that **sum-with** is a statement, not an expression, so you need to store the result in a temporary variable, and cannot use it directly, as in a **$display** statement.

<u>**Example 2-23**</u> Array locator methods

```
int count, d[] = '{9,1,8,3,4,4};

count = d.sum with (item >  7);  // 2: {9, 8}
count = d.sum with (item <  8);  // 4: {1, 3, 4, 4}
count = d.sum with (item == 4);  // 2: {4, 4}
```

2.9   Choosing a Storage Type

Here are some guidelines for choosing the right storage type based on flex-ibility, memory usage, speed, and sorting. These are just rules of thumb, and results may vary between simulators.

2.9.1   Flexibility

Use a fixed-size or dynamic array if it is accessed with consecutive posi-tive integer indices: 0, 1, 2, 3… Choose a fixed-size array if the array size is known at compile time, or choose a dynamic array if the size is not known until run-time. For example, variable-size packets can easily be stored in a dynamic array. If you are writing routines to manipulate arrays, consider using just dynamic arrays, as one routine will works for any size dynamic array as long as the element type (**int**, **string**, etc.) matches. Likewise, you can pass a queue of any size into a routine as long as the element type matches the queue argument. Associative arrays can also be passed regardless of size. However, a routine with a fixed-size array argument only accepts arrays of the specified length.

Choose associative arrays for nonstandard indices such as widely sepa-rated values because of random data values or addresses. Associative arrays can also be used to model content-addressable memories.

Queues are a good way to store data where the number of elements grows and shrinks a lot during simulation, such as a scoreboard that holds expected values. Lastly, queues are great for searching and sorting.

2.9.2   Memory usage

If you want to reduce the simulation memory usage, use two-state ele-ments. You should chose data sizes that are multiples of 32 bits to avoid wasted space. Simulators usually store anything smaller in a 32-bit word. For example, an array of 1024 bytes wastes ¾ of the memory if the simulator puts each element in a 32-bit word. Packed arrays can also help conserve memory.

For arrays that hold up to a thousand elements, the type of array that you choose does not make a big difference in memory usage (unless there are

many instances of these arrays). For arrays with a thousand to a million active elements, fixed-size and dynamic arrays are the most memory efficient. You may want to reconsider your algorithms if you need arrays with more than a million active elements.

Queues are slightly less efficient to access than fixed-size or dynamic arrays because of additional pointers. However, if your data set grows and shrinks often, and you store it in a dynamic memory, you will have to manually call **new[]** to allocate memory and copy. This is an expensive operation and would wipe out any gains from using a dynamic memory.

Modeling memories larger than a few megabytes should be done with an associative array. Note that each element in an associative array can take several times more memory than a fixed-size or dynamic memory because of pointer overhead.

2.9.3   Speed

Choose your array type based on how many times it is accessed per clock cycle. For only a few reads and writes, you could use any type, as the overhead is minor compared with the DUT. As you use an array more often, its size and type matters.

Fixed-size and dynamic arrays are stored in contiguous memory, so any element can be found in the same amount of time, regardless of array size.

Queues have almost the same access time as a fixed-size or dynamic array for reads and writes. The first and last elements can be pushed and popped with almost no overhead. Inserting or removing elements in the middle requires many elements to be shifted up or down to make room. If you need to insert new elements into a large queue, your testbench may slow down, so consider changing how you store new elements.

When reading and writing associative arrays, the simulator must search for the element in memory. The LRM does not specify how this is done, but popular ways are hash tables and trees. These requires more computation than other arrays, and therefore associative arrays are the slowest.

2.9.4   Sorting

Since SystemVerilog can sort any single-dimension array (fixed-size, dynamic, and associative arrays plus queues), you should pick based on how often the data is added to the array. If the data is received all at once, chose a fixed-size or dynamic array so that you only have to allocate the array once. If the data slowly dribbles in, chose a queue, as adding new elements to the head or tail is very efficient.

If you have values that are noncontiguous such as '{1, 10, 11, 50}, and are also unique, you can store them in an associative array by using them as an index. Using the routines **first**, **next**, and **prev**, you can search an associative array for a value and find successive values. Lists are doubly linked, so you can find values both larger and smaller than the current value. Both of these support removing a value. However, the associative array is much faster in accessing any given element given an index.

For example, you can use an associative array of bits to hold expected 32-bit values. When the value is created, write to that location. When you need to see if a given value has been written, use the **exists** function. When done with an element, use **delete** to remove it from the associative array.

2.9.5   Choosing the best data structure

Here are some suggestions on choosing a data structure.

- Network packets. Properties: fixed size, accessed sequentially. Use a fixed-size or dynamic array for fixed- or variable-size packets.

- Scoreboard of expected values. Properties: variable size, accessed by value. In general, use a queue, as you are adding and deleting elements constantly during simulation. If you can give every transaction a fixed id, such as 1, 2, 3 …, you could use this as an index into the queue. If your transaction is filled with random values, you can just push them into a queue and search for unique values. If the scoreboard may have hundreds of elements, and you are often inserting and deleting them from the middle, an associative array may be faster.

- Sorted structures. Use a queue if the data comes out in a predictable order or an associative array if the order is unspecified. If the scoreboard never needs to be searched, just store the expected values in a mailbox, as shown in section 7.6.

- Modeling very large memories, greater than a million entries. If you do not need every location, use an associative array as a sparse memory. If you do need every location, try a different approach where you do not need so much live data. Still stuck? Be sure to use 2-state values packed into 32-bits.

- Command names and values from a file. Property: lookup by string. Read the strings from the file, and then look up the commands in an associative array using the command as a string index.

You can create an array of handles that point to objects, as shown in Chapter 4 on Basic OOP.

2.10   Creating New Types with **typedef**

You can create new types using the **typedef** statement. For example, you may have an ALU that can be configured at compile-time to use on 8, 16, 24, or 32-bit operands. In Verilog you would define a macro for the operand width and another for the type.

Example 2-24 User-defined type-macro in Verilog

```
// Old Verilog style
`define OPSIZE 8
`define OPREG reg [`OPSIZE-1:0]


`OPREG op_a, op_b;
```

You are not really creating a new type; you are just performing text substitution. In SystemVerilog you create a new type with the following code. This book uses the convention that user-defined types use the suffix "**_t**."

Example 2-25 User-defined type in SystemVerilog

```
// New SystemVerilog style
parameter OPSIZE = 8;
typedef reg [OPSIZE-1:0] opreg_t;


opreg_t op_a, op_b;
```

In general, SystemVerilog lets you copy between these basic types with no warning, either extending or truncating values if there is a width mismatch.

Note that the **parameter** and **typedef** statements can be made global by putting them in **$root**, as shown in section 5.7.

One of the most useful types you can create is an unsigned, 2-state, 32-bit integer. Most values in a testbench are positive integers such as field length or number of transactions received. Put the following definition of **uint** in **$root** so it can be used anywhere in your simulation.

Example 2-26 Definition of uint

```
typedef bit [31:0] uint;    // 32-bit unsigned 2-state
typedef int unsigned uint;  // Equivalent definition
```

2.11   Creating User-Defined Structures

One of the biggest limitations of Verilog is the lack of data structures. In SystemVerilog you can create a structure using the **struct** statement, similar to what is available in C. But a **struct** is a degenerate class, so use a class instead, as shown in Chapter 4. Just as a Verilog module combines both data (signals) and code (always/initial blocks plus routines), a class combines data and routines to make an entity that can be easily debugged and reused. A **typedef** just groups data fields together. Without the code that manipulates the data, you are only creating half of the solution.

There are several places where a **typedef** is useful: creating simple user-defined types, unions, and enumerated types and virtual interfaces.

2.11.1   Creating a **struct** and a new type

You can combine several variables into a structure. Example 2-27 creates a structure called **pixel** that has three unsigned bytes for red, green, and blue.

Example 2-27 Creating a single pixel type

```
struct {bit [7:0] r, g, b;} pixel;
```

The problem with the above declaration is that it creates a single pixel of this type. To be able to share pixels using ports and routines, you should create a new type instead.

Example 2-28 The pixel struct

```
typedef struct {bit [7:0] r, g, b;} pixel_s;
pixel_s my_pixel;
```

Use the suffix "**_s**" when declaring a **struct**. This makes it easier for you to share and reuse code.

2.11.2   Making a union of several types

In hardware, the interpretation of a set of bits in a register may depend on the value of other bits. For example, a processor instruction may have many layouts based on the opcode. Immediate-mode operands might store a literal value in the operand field. This value may be decoded differently for integer instructions than for floating point instructions. Example 2-29 stores both the integer **i** and the real number **f** in the same location.

Example 2-29 Using `typedef` to create a union

```
typedef union { int i; real f; } num_u;
num_u un;
un.f = 0.0; // set n in floating point format
```

Use the suffix "**_u**" when declaring a union.

Unions are useful when you frequently need to read and write a register in several different formats. However, don't go overboard, especially just to save memory. Unions may help squeeze a few bytes out of a structure, but at the expense of having to create and maintain a more complicated data structure. Instead, make a flat class with a discriminant variable, as shown in section 8.5.4. This "kind" variable indicates which type of transaction you have, and thus which fields to read, write, and randomize. If you just need an array of values, plus all the bits, used a packed array as shown in 2.3.6

2.11.3   Packed structures

SystemVerilog allows you more control in how data is laid out in memory by using packed structures. A packed structure is stored as a contiguous set of bits with no unused space. The **struct** for a pixel, shown above, used three data values, so it is stored in three longwords, even though it only needs three bytes. You can specify that it should be packed into the smallest possible space.

Example 2-30 Packed structure

```
typedef struct packed {bit [7:0] r, g, b;} pixel_p_s;
pixel_p_s my_pixel;
```

Packed structures are used when the underlying bits represent a numerical value, or when you are trying to reduce memory usage. For example, you could pack together several bit-fields to make a single register. Or you might pack together the opcode and operand fields to make a value that contains an entire processor instruction.

2.12   Enumerated Types

An enumeration creates a strong variable type that is limited to a set of specified names such as the instruction opcodes or state machine values. Using these names, such as ADD, MOVE, or ROTW, makes your code easier to write and maintain than using literals such as 8'h01.

The simplest enumerated type declaration contains a list of constant names and one or more variables. This creates an anonymous enumerated type.

Example 2-31 A simple enumerated type

```
enum {RED, BLUE, GREEN} color;
```

You usually want to create a named enumerated type to easily declare multiple variables, especially if these are used as routine arguments or module ports. You first create the enumerated type, and then the variables of this type. You can get the string representation of an enumerated variable with the function **name**.

Example 2-32 Enumerated types

```
// Create data type for values 0, 1, 2
typedef enum {INIT, DECODE, IDLE} fsmstate_e;
fsmstate_e pstate, nstate;   // declare typed variables

initial begin
  case (pstate)
    IDLE:    nstate = INIT;    // data assignment
    INIT:    nstate = DECODE;
    default: nstate = IDLE;
  endcase
  $display("Next state is %0s",
           nstate.name);        // Use name function
  end
```

Use the suffix "**_e**" when declaring an enumerated type.

2.12.1   Defining enumerated values

The actual values default to integers starting at 0 and then increase. You can choose your own enumerated values. The following line uses the default value of 0 for **INIT**, then 2 for **DECODE**, and 3 for **IDLE**.

Example 2-33 Specifying enumerated values

```
typedef enum {INIT, DECODE=2, IDLE} fsmtype_e;
```

Enumerated constants, such as **INIT** above, follow the same scoping rules as variables. Consequently, if you use the same name in several enumerated types (such as **INIT** in different state machines), they have to be declared in different scopes such as modules, program blocks, routines, or classes.

Enumerated types are stored as **int** unless you specify otherwise. Be careful when assigning values to enumerated constants, as the default value of an **int** is 0. In Example 2-34, **position** is initialized to 0, which is not a legal **ordinal_e** variable. This behavior is <u>not</u> a tool bug – it is how the language is specified. So always specify an enumerated constant with the value of 0, just to catch this error.

Example 2-34 Incorrectly specifying enumerated values

```
typedef enum {FIRST=1, SECOND, THIRD} ordinal_e;
ordinal_e position;
```

Example 2-35 Correctly specifying enumerated values

```
typedef enum {ERR_O=0, FIRST=1, SECOND, THIRD} ordinal_e;
ordinal_e position;
```

2.12.2 Routines for enumerated types

SystemVerilog provides several functions for stepping through enumerated types.

- **first** returns first member of the enumeration.
- **last** returns last member of the enumeration.
- **next** returns the next element of the enumeration.
- **next(N)** returns the Nth next element.
- **prev** returns the previous element of the enumeration.
- **prev(N)** returns the Nth previous element.

The functions **next** and **prev** wrap around when they reach the beginning or end of the enumeration.

Note that there is no easy way to write a **for** loop that steps through all members of an enumerated type if you use an enumerated loop variable. You get the starting member with **first** and the next member with **next**. The problem is creating a comparison for the final iteration through the loop. If you use the test **current!=current.last**, the loop ends before using the last value. If you use **current<=current.last**, you get an infinite loop, as **next** never gives you a value that is greater than the final value.

You can use a **do**...**while** loop to step through all the values.

Example 2-36 Stepping through all enumerated members

```
enum {RED, BLUE, GREEN} color;
color = color.first;
do
  begin
  $display("Color = %0d/%0s", color, color.name);
  color = color.next;
  end
while (color != color.first);  // Done at wrap-around
```

2.12.3   Converting to and from enumerated types

The default type for an enumerated type is **int** (2-state). You can take the value of an enumerated variable and put it in an **integer** or **int** with a simple assignment. But SystemVerilog does not let you store a 4-state **integer** in an **enum** without explicitly changing the type. SystemVerilog requires you to explicitly cast the value to make you realize that you could be writing an out-of-bounds value.

Example 2-37 Assignments between integers and enumerated types

```
typedef enum {RED, BLUE, GREEN} COLOR_E;
COLOR_E color, c2;
integer c;

initial begin
  c = color;              // Convert from enum to integer
  c++;                    // Increment integer
  if (!$cast(color, c))   // Cast integer back to enum
    $display("Cast failed for c=%0d", c);
  $display("Color is %0d / %0s", color, color.name);
  c2 = COLOR_E'(c);       // No type checking done
  end
```

When called as a function as shown in Example 2-37, **$cast** tried to assign from the right value to the left. If the assignment succeeds, **$cast** returns 1. If the assignment fails because of an out-of-bounds value, no assignment is made and the function returns 0. If you use **$cast** as a task and the operation fails, SystemVerilog prints an error.

You can also cast the value using the **type'(val)** as shown above, but this does not do any type checking, so the result may be out of bounds. You should not use this style.

2.13   Constants

There are several types of constants in SystemVerilog. The classic Verilog way to create a constant is with a text macro. On the plus side, macros have global scope and can be used for bit field definitions and type definitions. On the negative side, macros are global, so they can cause conflicts if you just need a local constant. Lastly, a macro requires the ` character so that it will be recognized and expanded.

In SystemVerilog, parameters can be declared at the **$root** level so they can be global. This approach can replace many Verilog macros that were just being used as constants. You can use a **typedef** to replace those clunky macros. The next choice is a **parameter**. A Verilog **parameter** was loosely typed and was limited in scope to a single module. Verilog-2001 added typed parameters, but the limited scope kept parameters from being widely used.

SystemVerilog also supports the **const** modifier that allows you to make a variable that can be initialized in the declaration but not written by procedural code.

Example 2-38 Declaring a `const` variable

```
initial begin
  const byte colon = ":";
  ...
end
```

In Example 2-38, the value of **colon** is initialized when the **initial** block is entered. Example 3-10 shows a **const** routine argument.

2.14   Strings

If you have ever tried to use a Verilog reg variable to hold a string of characters, your suffering is over. The SystemVerilog **string** type holds variable-length strings. An individual character is of type **byte**. The elements of a string of length N are numbered 0 to N-1. Note that, unlike C, there is no null character at the end of a string, and any attempt to use the character "\0" is ignored. Strings use dynamic memory allocation, so you do not have to worry about running out of space to store the string.

Example 2-39 shows various string operations. The function **getc(N)** returns the byte at location N, while **toupper** returns an upper-case copy of the string and **tolower** returns a lowercase copy. The curly braces {} are used for concatenation. The task **putc(M)** writes a byte into a string at location **M**, that must be between 0 and the length as given by **len**. The **substr(start,end)** function extracts characters from location **start** to **end**.

Example 2-39 String methods

```
string s;

initial begin
  s = "SystemVerilog";
  $display(s.getc(0));        // Display: 83 ('S')
  $display(s.toupper());      // Display: SYSTEMVERILOG

  s = {s, "3.1b"};            // "SystemVerilog3.1b"
  s.putc(s.len()-1, "a");     // change b-> a

  $display(s.substr(2, 5));   // Display: stem

  // Create temporary string, note format
  my_log($psprintf("%s %5d", s, 42));
end

task my_log(string message);
  // Print a message to a log
  $display("@%0d: %s", $time, message);
endtask
```

Note how useful dynamic strings can be. In other languages such as C you have to keep making temporary strings to hold the result from a function. In Example 2-39, the **$psprintf**[2] function is used instead of **$sformat,** from Verilog-2001. This new function returns a formatted temporary string that, as shown above, can be passed directly to another routine. This saves you from having to declare a temporary string and passing it between the formatting statement and the routine call.

2.15   Expression Width

A prime source for unexpected behavior in Verilog has been the width of expressions. Example 2-40 adds $1 + 1$ using four different styles. Addition **A** uses two 1-bit variables, so with this precision 1+1=0. Addition **B** uses 8-bit precision because there is an 8-bit variable on the right side of an assignment. In this case, 1+1=2. Addition **C** uses a dummy constant to force SystemVerilog to use 2-bit precision. Lastly, in addition **D**, the first value is cast to be a 2-bit value with the cast operator, so 1+1=2.

[2.] The function **$psprintf** is implemented in Synopsys VCS and submitted for the next version of SystemVerilog. Other simulators may have already implemented it.

Example 2-40 Expression width depends on context

```
bit [7:0] b8;
bit one = 1'b1;                 // Single bit
$displayb(one + one);           // A: 1+1 = 0

b8 = one + one;                 // B: 1+1 = 2
$displayb(b8);

$displayb(one + one + 2'b0);  // C: 1+1 = 2 with constant

$displayb(2'(one) + one);      // D: 1+1 = 2 with cast
```

There are several tricks you can use to avoid this problem. First, avoid situations where the overflow is lost, as in addition **A**. Use a temporary, such as **b8**, with the desired width. Or, you can add another value to force the minimum precision, such as **2'b0**. Lastly, in SystemVerilog, you can cast one of the variables to the desired precision.

2.16   Net Types

Verilog allows you to use nets without defining them, a feature called implicit nets. This shortcut helps net-listing tools and lazy designers, but is guaranteed to cause problems if you ever misspell a net name. The solution is to disable this language feature with the Verilog-2001 compile directive: **`default_nettype none**. Put this (without a period) before the first module in your Verilog code. Any implicit net will cause a compilation error.

Example 2-41 Disabling implicit nets with 'default_nettype none

```
`default_nettype none

module first;
...
```

2.17   Conclusion

SystemVerilog provides many new data types and structures so that you can create high-level testbenches without having to worry about the bit-level representation. Queues work well for creating scoreboards where you constantly need to add and remove data. Dynamic arrays allow you to choose the array size at run-time for maximum testbench flexibility. Associative arrays are used for sparse memories and some scoreboards with a single index. Enu-

merated types make your code easier to read and write by creating groups of named constants.

But don't go off and create a procedural testbench with just these constructs. Explore the OOP capabilities of SystemVerilog in Chapter 4 to learn how to design code at an even higher level of abstraction, thus creating robust and reusable code.

Chapter 3

Procedural Statements and Routines

3.1   Introduction

As you verify your design, you need to write a great deal of code, most of which is in tasks and functions. SystemVerilog introduces many incremental improvements to make this easier by making the language look more like C, especially around argument passing. If you have a background in software engineering, these additions should be very familiar.

3.2   Procedural Statements

SystemVerilog adopts many operators and statements from C and C++. You can declare a loop variable inside a **for** loop that then restricts the scope of the loop variable and can prevent some coding bugs. The increment **++** and decrement **--** operators are available in both pre- and post- form. If you have a label on a **begin** or **fork** statement, you can put the same label on the matching **end** or **join** statement. This makes it easier to match the start and finish of a block. You can also put a label on other SystemVerilog end statements such as **endmodule**, **endtask**, **endfunction**, and others that you will learn in this book. Example 3-1 demonstrates some of the new constructs.

Example 3-1 New procedural statements and operators

```
initial
  begin : example
  integer array[10], sum, j;

  // Declare i in for statement
  for (int i=0; i<10; i++)         // Increment i
    array[i] = i;

  // Add up values in the array
  sum = array[9];
  j=8;
  do                              // do...while loop
    sum += array[j];              // Accumulate
  while (j--);                    // Test if j=0
  $display("Sum=%4d", sum);       // %4d - specify width
end : example                     // End label
```

Two new statements help with loops. First, if you are in a loop, but want to skip over rest of the statements and do the next iteration, use **continue**. If you want to leave the loop immediately, use **break**.

The following loop reads commands from a file using the amazing file I/O code that is part of Verilog-2001. If the command is just a blank line, the code just does a **continue** and skips any further processing of the command. If the command is "done," the code does a **break** to terminate the loop.

Example 3-2 Using break and continue while reading a file

```
initial begin
  logic [127:0] cmd;
  integer file, c;

  file = $fopen("commands.txt", "r");
  while (!$feof(file)) begin
    c = $fscanf(file, "%s", cmd);
    case (cmd)
      "": continue;     // Blank line - skip to loop end
      "done": break;    // Done - leave loop
      // Process other commands here
      ...
    endcase // case(cmd)
    end
  $fclose(file);
end
```

3.3   Tasks, Functions, and Void Functions

Verilog makes a very clear differentiation between tasks and functions. The most important difference is that a task can consume time while a function cannot. A function cannot have a delay, **#100**, a blocking statement such as **@(posedge clock)** or **wait(ready)**, or call a task. Additionally, a Verilog function must return a value, and the value must be used, as in an assignment statement.

In SystemVerilog, if you want to call a function and ignore its return value, cast the result to **void**. This might be done if you are calling the function to use a side effect.

Example 3-3 Ignoring a function's return value

```
void'(my_func(42));
```

Some simulators such as VCS allow you to ignore the return value without using the above **void** syntax.

If you have a SystemVerilog task that does not consume time, you should make it a **void function** that is a function that does not return a value. Now it can be called from any task or function. For maximum flexibility, any debug routine should be a void function rather than a task so that it can be called from any task or function. Example 3-4 prints values from a state machine.

Example 3-4 Void function for debug

```
function void print_state(...);
  $display("@%0d: state = %0s", $time, cur_state.name);
endfunction
```

3.4   Task and Function Overview

SystemVerilog makes several small improvements to tasks and functions to make them look more like C or C++ routines.[3]

3.4.1   Routine **begin...end** removed

The first improvement you may notice in SystemVerilog is that **begin**...**end** blocks are optional, while Verilog-1995 required them on all but single line routines. The **task** / **endtask** and **function** / **endfunction** keywords are enough to define the routine boundaries.

Example 3-5 Simple task without begin...end

```
task multiple_lines;
  $display("First line");
  $display("Second line");
endtask : multiple_lines
```

3.5   Routine Arguments

Many of the SystemVerilog improvements for routine make it easier to declare arguments and expand the ways you can pass values to and from a routine.

3.5.1   C-style Routine Arguments

SystemVerilog and Verilog-2001 allow you to declare task and function arguments more cleanly and with less repetition. The following Verilog task

---

[3.] In general, a routine definition or call with no arguments does not need the empty parentheses (). This book leaves them out except as needed for clarity.

requires you to declare some arguments twice, once for the direction, and once for the type.

Example 3-6 Verilog-1995 routine arguments

```
task mytask2;
  output [31:0] x;
  reg    [31:0] x;
  input         y;
  ...
endtask
```

With SystemVerilog, you can use the less verbose C-style. Note that you should use the universal input type of `logic`.

Example 3-7 C-style routine arguments

```
task mytask1 (output logic [31:0] x,
              input  logic y);
...
endtask
```

3.5.2   Argument Direction

You can take even more shortcuts with declaring routine arguments. The direction and type default to "input logic" and are sticky, so you don't have to repeat these for similar arguments. Here is a routine header written using the Verilog-1995 style.

Example 3-8 Verbose Verilog-style routine arguments

```
task T3;
  input a, b;
  logic a, b;
  output [15:0] u, v;
  bit [15:0] u, v;
  ...
endtask
```

You could rewrite this as follows.

Example 3-9 Routine arguments with sticky types

```
task T3(a, b, output bit [15:0] u, v);
```

The arguments a and b are input logic, 1 bit wide. The arguments `u` and `v` are 16-bit output bit types.

3.5.3   Advanced Argument Types

Verilog had a simple way to handle arguments: an **input** or **inout** was copied to a local variable at the start of the routine, while an **output** or **inout** was copied when the routine exited. No memories could be passed into a Verilog routine, only scalars.

In SystemVerilog, you can specify that an argument is passed by reference, rather than copying its value. This argument type, **ref**, has several benefits over **input**, **output**, and **inout**. First, you can now pass an array into a routine.

Example 3-10 Passing arrays using ref and const

```
function void print_sum (const ref int a[]);
  int sum = 0;
  for (int i=0; i<a.size; i++) begin
    sum += a[i];
  $display("The sum of the arrays is ", sum);
endfunction
```

SystemVerilog allows you to pass array arguments without the **ref** direction, but the array is copied onto the stack, an expensive operation for all but the smallest arrays.

Example 3-10 also shows the **const** modifier. As a result, the array **a** is initialized when **print_sum** is called, but cannot be modified in the routine.

Always use **ref** when passing arrays to a routine. If you don't want the routine to change the array values, use the **const ref** type. With this, the compiler checks that your routine does not modify the array.

The second benefit of **ref** arguments is that a task can modify a variable and is instantly seen by the calling function. This is useful when you have several threads executing concurrently and want a simple way to pass information. See Chapter 7 for more details on using **fork-join**.

Example 3-11 Using `ref` across threads

```
task bus_read(input logic [31:0] addr,
              ref   logic [31:0] data);

  // Request bus and drive address
  bus.request = 1'b1;
  @(posedge bus.grant) bus.addr = addr;

  // Wait for data from memory
  @(posedge bus.enable) data = bus.data;

  // Release bus and wait for grant
  bus.request = 1'b0;
  @(negedge bus.grant);
endtask

logic [31:0] addr, data;
initial
  fork
    bus_read(addr, data);
    begin
      @data;  // Trigger on data change
      $display("Read %h from bus", data);
    end
  join
```

In Example 3-11, the initial block can access the data from memory as soon as **bus.enable** is asserted, even though the **bus_read** task does not return until the bus transaction completes, which could be several cycles later. Since the **data** argument is passed as **ref**, the **@data** statement triggers as soon as **data** changes in the task. If you had declared **data** as **output**, the **@data** statement would not trigger until the end of the bus transaction.

3.5.4   Default Argument Values

As your testbench grows in sophistication, you may want to add additional controls to your code but not break existing code. For the function in Example 3-10, you might want to print a sum of just the middle values of the array. However, you don't want to go back and rewrite every call to add extra arguments. In SystemVerilog you can specify a default value that is used if you leave out an argument in the call.

Example 3-12 Function with default argument values

```
function void print_sum (ref    int a[],
                         input int start = 0,
                         input int end = -1);
  int sum = 0, last;
  if (last == -1 || last > a.size)
    last = a.size;
  for (int i=start; i<last; i++) begin
    sum += a[i];
  $display("The sum of the arrays is ", sum);
endtask
```

You can call this task in the following ways. Note that the first call is compatible with both versions of the **print_sum** routine.

Example 3-13 Using default argument values

```
print_sum(a);        // Sum a[0:size-1] – default
print_sum(a, 2, 5); // Sum a[2:5]
print_sum(a, 1);    // Start at 1
print_sum(a,, 3);   // Sum a[0:3]
print_sum();        // error: a has no default
```

Using a default value of -1 (or any out-of-range value) is a good way to see if the call specified a value.

3.5.5   Common Coding Errors

The most common coding mistake that you are likely to make with a routine is forgetting that the argument type is sticky with respect to the previous argument, and that the default type for the first argument is a single-bit input. Start with the following simple task header.

Example 3-14 Original task header

```
task sticky(int a, b);
```

The two arguments are input integers. As you are writing the task, you realize that you need access to an array, so you add a new array argument, and use the **ref** type so it does not have to be copied.

Example 3-15 Task header with additional array argument

```
task sticky(ref int array[50],
            int a, b);       // What direction are these?
```

What argument types are **a** and **b**? They take the direction of the previous argument that is a **ref**. Using **ref** for a simple variable such as an **int** is not usually needed, but you would not get even a warning from the compiler, and thus would not realize that you were using the wrong type.

If any argument to your routine is something other than the default input type, specify the direction for all arguments.

Example 3-16 Task header with additional array argument

```
task sticky(ref    int array[50],
            input int a, b);  // Be explicit
```

3.6   Returning from a Routine

SystemVerilog adds the **return** statement to make it easier for you to control the flow in your routines. The following task needs to return early because of error checking. Otherwise, it would have to use an **else** clause, that would cause more indentation and be harder to read.

Example 3-17 Return in a task

```
task load_array(int len, ref int array[]);
  if (len <= 0) begin
    $display("Bad len");
    return;
  end

  // Code for the rest of the task
  ...
endtask
```

A **return** statement can simplify your functions.

Example 3-18 Return in a function

```
function bit transmit(...);
  // Send transaction
  ...
  return ~ifc.cb.error; // Return status: 0=error
endfunction
```

3.7   Local Data Storage

When Verilog was created in the 1980s, its primary goal was describing hardware. Because of this, all objects in the language were statically allocated. In particular, routine arguments and local variables were stored in a fixed location, rather than pushing them on a stack like other programming

languages. After all, how can you build a silicon representation of a recursive routine? However, software engineers who were used to the behavior of stack-based languages such as C were bitten by these subtle bugs, and were limited in their ability to create complex testbenches with libraries of routines.

3.7.1   Automatic storage

In Verilog-1995, if you tried to call a task from multiple places in your testbench, the local variables shared common, static storage, and so the different threads stepped on each other's values. In Verilog-2001 you can specify that tasks, functions, and modules use automatic storage, which causes the simulator to use the stack for local variables.

In SystemVerilog, routines still use static storage by default, for both modules and program blocks. You should always make program blocks (and their routines) use automatic storage by putting the **automatic** keyword in the program statement. In Chapter 5 you will learn about **program** blocks that hold the testbench code. You should always make programs automatic.

Example 3-19 shows a task to monitor when data is written into memory.

Example 3-19 Specifying automatic storage in program blocks

```
program automatic test;
  task wait_for_mem(input [31:0] addr, expect_data,
                    output success);
    while (bus.addr !== addr)
      @(bus.addr);
    success = (bus.data == expect_data);
  endtask
...
endprogram
```

You can call this task multiple times concurrently, as the **addr** and **expect_data** arguments are stored separately for each call. Without the **automatic** modifier, if you called **wait_for_mem** a second time while the first was still waiting, the second call would overwrite the two arguments.

3.7.2   Variable initialization

A similar problem occurs when you try to initialize a local variable in a declaration, as it is actually initialized at the start of simulation. The general solution is to avoid initializing a variable in a declaration to anything other than a constant. Use a separate assignment statement to give you better control over when initialization is done.

The following task looks at the bus after five cycles and then creates a local variable and attempts to initialize it to the current value of the address bus.

Example 3-20 Static initialization bug

```
program initialization; // Buggy version
  task check_bus;
    repeat (5) @(posedge clock);
    if (bus_cmd == `READ) begin
      // When is local_addr initialized?
      reg [7:0] local_addr = addr<<2;  // Bug
      $display("Local Addr = %h", local_addr);
    end
  endtask
endprogram
```

The bug is that the variable **local_addr** is statically allocated, so it is actually initialized at the start of simulation, not when the **begin**...**end** block is entered. Once again, the solution is to declare the program as **automatic**.

Example 3-21 Static initialization fix: use `automatic`

```
program automatic initialization; // Bug solved
...
endmodule
```

3.8   Time Values

SystemVerilog has several new constructs to allow you to unambiguously specify time values in your system.

3.8.1   Time units and precision

When you rely on the **`timescale** compiler directive, you must compile the files in the proper order to be sure all the delays use the proper scale and precision. The **timeunit** and **timeprecision** declarations eliminate this ambiguity by precisely specifying the values for every module. Example 3-22 shows these declarations. Note that if you use these instead of **`timescale**, you must put them in every module that has a delay.

3.8.2   Time literals

SystemVerilog allows you to unambiguously specify a time value plus units. Your code can use delays such as **0.1ns** or **20ps**. Just remember to use **timeunit** and **timeprecision** or **`timescale**. You can make your code

even more time aware by using the classic Verilog **$timeformat** and **$realtime** routines.

Example 3-22 Time literals and $timeformat

```
module timing;
  timeunit 1ns;
  timeprecision 1ps;
  initial begin
    $timeformat(-9, 3, "ns", 8);
    #1      $display("@%t", $realtime); // @1.000ns
    #2ns    $display("@%t", $realtime); // @3.000ns
    #0.1ns  $display("@%t", $realtime); // @3.100ns
    #41ps   $display("@%t,  $realtime); // @3.141ns
  end
endmodule
```

3.9   Conclusion

The new SystemVerilog procedural constructs and task/function features make it easier for you to create testbenches by making the language look more like other programming language such as C/C++. But stick with SystemVerilog for the additional HDL constructs such as timing controls and four-state logic.

Chapter 4

Basic OOP

4.1   Introduction

With procedural programming languages such as Verilog and C, there is a strong division between data structures and the code that uses them. The declarations and types of data are often in a different file than the algorithms that manipulate them. As a result, it can be difficult to understand the functionality of a program, as the two halves are separate.

Verilog users have it even worse than C users, as there are no structures in Verilog, only bit vectors and arrays. If you wanted to store information about a bus transaction, you would need multiple arrays: one for the address, one for the data, one for the command, and more. Information about transaction N is spread across all the arrays. Your code to create, transmit, and receive transactions is in a module that may or may not be actually connected to the bus. Worst of all, the arrays are all static, so if your testbench only allocated 100 array entries, and the current test needed 101, you would have to edit the source code to change the size and recompile. As a result, the arrays are sized to hold the greatest conceivable number of transactions, but during a normal test, most of that memory is wasted.

Object Oriented Programming (OOP) lets you create complex data types and tie them together with the routines that work with them. You can create testbenches and system-level models at a more abstract level by calling routines to perform an action rather than toggling bits. When you work with transactions instead of signal transitions, you are more productive. As a bonus, your testbench is decoupled from the design details, making it more robust and easier to maintain and reuse on future projects.

If you already are familiar with OOP, skim this chapter, as SystemVerilog follows OOP guidelines fairly closely. Be sure to read section 4.18 to learn how to build a testbench. Chapter 8 presents advanced OOP concepts such as inheritance and more testbench techniques; it should be read by everyone.

4.2   Think of Nouns, not Verbs

Grouping data and code together helps you in creating and maintaining large testbenches. How should data and code be brought together? You can start by thinking of how you would perform the testbench's job.

The goal of a testbench is to apply stimulus to a design and then check the result to see if it is correct. The data that flows into and out of the design is

grouped together into transactions. So the easiest way to organize the testbench is around the transactions, and the operations that you perform. In OOP, the transaction is the object that is focus of your testbench.

You can think of an analogy with transportation. When you get into a car, you want to perform discrete actions, such as starting, moving forward, turning, stopping, and listening to music while you drive. Early cars required detailed knowledge about their internals to operate. You had to advance or retard the spark, open and close the choke, keep an eye on the engine speed and be aware of the traction of the tires if you drove on a slippery surface such as a wet road. Today your interactions with the car are at a high level. If you want to start a car, just turn the key in the ignition, and you are done. Get the car moving by pressing the gas pedal; stop it with the brakes. Are you driving on snow? Don't worry: the anti-lock brakes will help you stop safely and in a straight line.

Your testbench should be structured the same way. Traditional testbenches were oriented around the operations that had to happen: create a transaction, transmit it, receive it, check it, and make a report. Instead, you should think about the structure of the testbench, and what each part does. The generator creates transactions and passes them to the next level. The driver talks with the design that responds with transactions that are received by a monitor. The scoreboard checks these against the expected data. You should divide your testbench into blocks, and then define how they communicate.

4.3   Your First Class

A class encapsulates the data together with the routines that manipulate it. Example 4-1 shows a class for a generic packet. The packet contains source and destination addresses, and an array of data values. There are two routines in the **BusTran** class: a function to display the contents of the packet, and another that computes the CRC (cyclic redundancy check) of the data.

To make it easier to match the beginning and end of a named block, you can put a label on the end of it. In Example 4-1 these end labels may look redundant, but in real code with many nested blocks, the labels help you find the mate for a simple end or **endtask**, **endfunction**, or **endclass**.

Example 4-1 Simple `BusTran` class

```
class BusTran;
  bit [31:0] addr, crc, data[8];

  function void display;
    $display("BusTran: %h", addr);
  endfunction : display

  function void calc_crc;
    crc = addr ^ data.xor;
  endfunction : calc_crc

endclass : BusTran
```


Every company has its own naming style. This book uses the following convention: Class names start with a capital letter and do not use underscores, as in **BusTran** or **Packet**. Constants are all upper case, as in **CELL_SIZE**, while variables are lower case, as in **count** or **trans_type**. You are free to use whatever style you want.

4.4   Where to Define a Class

You can define a class in SystemVerilog in a **program**, **module**, **package**, or outside of any these. Classes can be used in programs and modules. This book only shows classes that are used in a program block, as introduced in Chapter 5. Until then, think of a program block as a module that holds your test code. The program holds a single test and contains the objects that comprise the testbench, and the initial blocks to create, initialize, and run the test.

Many verification teams put either a standalone class or a group of closely related classes in a file. Bundle the group of classes with a SystemVerilog **package**. For instance, you might group together all SCSI/ATA transactions into a single **package**. Now you can compile the package separately from the test of the system. Unrelated classes, such as those for transactions, scoreboards, or different protocols, should go into separate files.

See the SystemVerilog LRM for more information on packages.

4.5   OOP Terminology

What separates you, an OOP novice, from an expert? The first thing is the words you use. You already know some OOP concepts from working with

Verilog. Here are some OOP terms, definitions, and rough equivalents in Verilog 2001.

- Class – a basic building block containing routines and variables. The analogue in Verilog is a module.
- Object – an instance of a class. In Verilog, you need to instantiate a module to use it.
- Handle – a pointer to an object. In Verilog, you use the name of an instance when you refer to signals and methods from outside the module. An OOP handle is like the address of the object, but is stored in a pointer that can only refer to one type.
- Property – a variable that holds data. In Verilog, this is a signal such as a register or wire.
- Method – the procedural code that manipulates variables, contained in tasks and functions. Verilog modules have tasks and functions plus initial and always blocks.
- Prototype – the header of a routine that shows the name, type, and argument list. The body of the routine contains the executable code.

This book uses the more traditional terms from Verilog of "variable" and "routine" rather than OOP's "property" and "method." If you are comfortable with the OOP terms, you can skim this chapter.

In Verilog you build complex designs by creating modules and instantiating them hierarchically. In OOP you create classes and instantiate them (creating objects) to create a similar hierarchy.

Here is a brief analogy to explain these OOP terms. Think of a class as the blueprint for a house. This plans describe how the structure of house, but you cannot live in a blueprint. An object is the actual house. Just as one set of blueprints can be used to build a whole subdivision of houses, a single class can be used to build many objects. The house address is like a handle in that it uniquely identifies your house. Inside your house you have things such as lights (on or off), with switches to control them. A class has variables that hold values, and routines that control the values. A class for the house might have many lights. A single call to `turn_on_porch_light()` sets the porch light variable to ON in a single house.

4.6   Creating New Objects

Both Verilog and OOP have the concept of instantiation, but there are some differences in the details. A Verilog module, such as a counter, is instan-

tiated when the netlist is compiled. A SystemVerilog class, such as a network packet, is instantiated at run-time when needed by the testbench. Verilog instances are static, as the hardware does not change during simulation; only signal values change. Stimulus objects are constantly being created and used to drive the DUT and check the results. Later, the objects may be freed so their memory can be used by new ones.[4]

The analogy between OOP and Verilog has a few other exceptions. The top-level Verilog module is not usually explicitly instantiated. However, a SystemVerilog class must be instantiated before it can be used. Next, a Verilog instance name only refers to a single instance, while a SystemVerilog handle can refer to many objects, though only one at a time.

4.6.1   No news is good news

In Example 4-2, **b** is a handle that points to an object of type **BusTran**. You can simplify this by just calling **b** a **BusTran** handle.

Example 4-2 Declaring and using a handle

```
BusTran b;  // Declare a handle
b = new;    // Allocate a BusTran object
```

When you declare the handle **b**, it is initialized to the special value **null**. Next, you call the **new** function to construct the **BusTran** object. **new** allocates space for the **BusTran**, initializes the variables to their default value (0 for 2-state variables and X for 4-state ones), and returns the address where the object is stored. For every class, SystemVerilog creates a default **new** to allocate and initialize an object. See section 4.6.2 for more details on this function.

4.6.2   Custom Constructor

Sometimes OOP terminology can make a simple concept seem complex. What does instantiation mean? When you call **new** to instantiate an object, you are allocating a new block of memory to store the variables for that object. For example, the **BusTran** class has two 32-bit registers (**addr** and **crc**) and an array with eight values (data), for a total of 11 longwords, or 44 bytes. So when you call **new**, SystemVerilog allocates 44 bytes of storage. If you have used C, this step is similar to the **malloc** function. (Note that Sys-

---

4. Back to the house analogy: the address is normally static, unless your house burns down, causing you to construct a new one. And garbage collection is never automatic.

temVerilog uses additional memory for four-state variables and housekeeping information such as the object's type.)

The **new** function does more than allocate memory. It also initializes the values. By default, it sets variables to their default values – 0 for 2-state variables, X for 4-state, etc. You can define your own **new** function so that you can set the values as you prefer. That is why the new function is also called the "constructor," as it builds the object, just as your house is constructed from wood and nails.

You can write your own **new** function. Note that it does not have a type, as it always returns an object of the same type as the class.

Example 4-3 Simple use-defined new function

```
class BusTran;
  logic [31:0] addr, crc, data[8];

  function new;
    addr = 3;
    foreach (data[i])
      data[i] = 5;
  endfunction
endclass
```

The above code sets **addr** and **data** to fixed values but leaves **crc** at its default value of X. (SystemVerilog allocates the space for the object automatically.) You can use the function argument with default values to make a more flexible constructor.

Example 4-4 A new function with arguments

```
class BusTran;
  logic [31:0] addr, crc, data[8];

  function new(logic [31:0] addr=3, d=5);
    this.addr = addr;
    foreach (data[i])
      data[i] = d;
  endfunction
endclass

initial begin
  BusTran b;
  b = new(10);  // data uses default of 5
end
```

How does SystemVerilog know which **new** function to call? It looks at the type of the handle on the left side of the assignment. In Example 4-5, the call to **new** inside the **Driver** constructor calls the **new** function for **BusTran**, even though the one for **Driver** is closer. Since **bt** is a **BusTran** handle, SystemVerilog does the right thing and create an object of type **BusTran**.

Example 4-5 Calling the right new function

```
class BusTran;
...
endclass : BusTran

class Driver;
  BusTran bt;
  function new();      // Driver's new function
    bt = new();        // Call the BusTran new function
  endfunction
endclass : Driver
```

4.6.3   Separating the declaration and construction

You should avoid declaring a handle and calling the constructor, **new**, all in one statement. While this is legal syntax, it can create ordering problems, as the constructor is called before the first procedural statement. You may want to initialize objects in a certain order, but if you call **new** in the declaration, you won't have this control. Additionally, if you forget to use **automatic** storage, the constructor is called at the start of simulation.

4.6.4   The difference between new() and new[]

You may have noticed that this **new()** function looks a lot like the **new[]** operator, described in section 2.4, used to set the size of dynamic arrays. They both allocate memory and initialize values. The big difference is that the **new()** function is called to construct a single object, while the **new[]** operator is building an array with multiple elements. **new()** can take arguments for setting object values, while **new[]** only takes a single value for the array size.

4.6.5   Getting a handle on objects

New OOP users often confuse an object with its handle. The two are very distinct. You **declare** a handle and **construct** an object. Over the course of a simulation, a handle can

point to many objects. This is the dynamic nature of OOP and SystemVerilog. Don't get the handle confused with the object.

In Example 4-6, `b1` first points to one object, then another.

Example 4-6 Allocating multiple objects

```
BusTran b1, b2;    // Declare two handles
b1 = new;          // Allocate first BusTran object
b2 = b1;           // b1 & b2 point to it
b1 = new;          // Allocate second BusTran object
```

Why would you want to create objects dynamically? During a simulation you may need to create hundreds or thousands of transactions. SystemVerilog lets you create new ones automatically, when you need them. In Verilog, you would have to use a fixed-size array large enough to hold the maximum number of transactions.

Figure 4-1 Handles and objects

Note that this dynamic creation of objects is different from anything else offered before in the Verilog language. An instance of a Verilog module and its name are bound together statically during compilation. Even with **automatic** variables, which come and go during simulation, the name and storage are always tied together.

An analogy for handles is people who are attending a conference. Each person is similar to an object. When you arrive, a badge is "constructed" by writing your name on it. This badge is a handle that can be used by the organizers to keep track of each person. When you take a seat for the lecture, space is allocated. You may have multiple badges for attendee, presenter, or organizer. When you leave the conference, your badge may be reused by writing a new name on it, just as a handle can point to different objects through assignment. Lastly, if you lose your badge and there is nothing to identify you, you will be asked to leave. The space you take, your seat, is reclaimed for use by someone else.

4.7   Object Deallocation

Now you know how to create an object – but how do you get rid of it? For example, your testbench creates and send thousands of transactions such as transactions into your DUT. Once you know the transaction completed suc-

cessfully, and you gather statistics, you don't need to keep it around. You should reclaim the memory; otherwise, a long simulation might run out of memory, or at least run more and more slowly.

Garbage collection is the process of automatically freeing objects that are no longer referenced. One way SystemVerilog can tell if an object is no longer being used is by keeping track of the number of handles that point to it. When the last handle no longer references an object, SystemVerilog releases the memory for it.[5]

Example 4-7 Creating multiple objects

```
BusTran b;   // Create a handle
b = new;     // Allocate a new BusTran
b = new;     // Allocate a second one, free the first
b = null;    // Deallocate the second
```

The second line calls **new** to construct an object and store the address in the handle **b.** The next call to **new** constructs a second object and stores its address in **b**, overwriting the previous value. Since there are no handles pointing to the first object, SystemVerilog can deallocate it. (SystemVerilog may delete the object immediately, or wait a while.) The last line explicitly clears the handle so that now the second object can be deallocated.

If you are familiar with C++, these concepts of objects and handles might look familiar, but there are some important differences. A SystemVerilog handle can only point to objects of one type, so they are called "type-safe." In C++, a typical untyped pointer is only an address in memory, and you can set it to any value or modify it with operators such as pre-increment. You cannot be sure that a pointer really is valid. SystemVerilog does not allow any modification of a handle or using a handle of one type to refer to an object of another type. (SystemVerilog's OOP specification is closer to Java than C++.)

Secondly, since SystemVerilog performs automatic garbage collection when no more handles refer to an object, you can be sure your code always uses valid handles. In C / C++, a pointer can refer to an object that no longer exists. Garbage collection in those languages is manual, so your code can suffer from "memory leaks" when you forget to deallocate objects.

SystemVerilog cannot garbage collect an object that is referenced by a handle. If you create linked lists (especially double-linked lists) or circular lists, SystemVerilog does not deallocate the object. You need to manually clear all handles by setting them to **null**. If an object contains a routine that forks off a

---

[5.] The actual algorithm to find unused objects varies between simulators. This section describes reference counting, which is the easiest to understand.

thread, the object is not deallocated while the thread is running. Likewise, any objects that are used by a spawned thread may not be deallocated until the thread terminates. See Chapter 7 for more information on threads.

4.8   Using Objects

Now that you have allocated an object, how do you use it? Going back to the Verilog module analogy, you can refer to variables and routines in an object with the "`.`" notation.

Example 4-8 Using variables and routines in an object

```
BusTran b;          // Declare a handle to a BusTran
b = new;            // Construct a BusTran object
b.addr = 32'h42;    // Set the value of a variable
b.display();        // Call a routine
```

In strict OOP, the only access to variables in an object should be through its public methods such as `get()` and `put()`. This is because accessing variables directly limits your ability to change the underlying implementation in the future. If a better (or simply different) algorithm comes along in the future, you may not be able to adopt it because you would also need to modify all of the references to the variables.

The problem with this methodology is that it was written for large software applications with lifetimes of a decade or more. With dozens of programmers making modifications, stability is paramount. But you are creating a testbench, where the goal is maximum control of all variables to generate the widest range of stimulus values. One of the ways to accomplish this is with constrained-random stimulus generation, which cannot be done if a variable is hidden behind a screen of methods. While the `get()` and `put()` methods are fine for compilers, GUIs, and APIs, you should stick with public variables that can be directly accessed anywhere in your testbench.

4.9   Static Variables vs. Global Variables

Every object has its own local variables that are not shared with any other object. If you have two `BusTran` objects, each has its own `addr`, `crc`, and `data` variables. But sometimes you need a variable that is shared by all objects of a certain type. For example, you might want to keep a running count of the number of transactions that have been created. Without OOP, you would probably create a global variable. Then you would have a global variable that is used by one small piece of code, but is visible to the entire testbench.

4.9.1   Using a static variable

In SystemVerilog you can create a static variable inside a class. This variable is shared between all instances of the class, but its scope is limited to the class. In Example 4-9, the static variable **count** holds the number of objects created so far. It is initialized to 0 in the declaration because there are no transactions at the beginning of the simulation. Each time a new object is constructed, it is tagged with a unique value, and **count** is incremented.
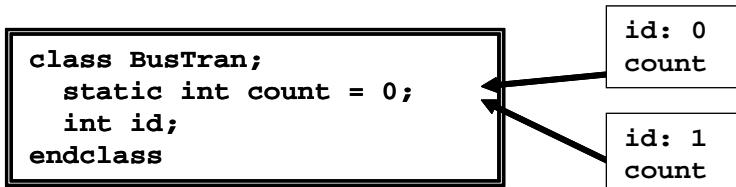
Example 4-9 Class with a static variable

```
class BusTran;
  static int count = 0; // Number of objects created
  int id;               // Unique instance ID
  function new;
    id = count++;       // Set ID, bump count
  endfunction
endclass

BusTran b1, b2;
initial begin
  b1 = new;               // First instance, id=0
  b2 = new;               // Second instance, id=1
  $display("Second id=%d, count=%b", b2.id, b2.count);
end
```

In Example 4-9, there is only one copy of the static variable **count**, regardless of how many **BusTran** objects are created. You can think that **count** is stored with the class and not the object. The variable **id** is not static, so every **BusTran** has its own copy. Now you don't need to make a global variable for the count.

Figure 4-2 Static variables in a class

Using the ID field is a good way to track objects as they flow through a design. When debugging a testbench, you often need a unique value. SystemVerilog does not let you print the address of an object, but you can make an ID field. Whenever you are tempted to make a global variable, consider making a

class-level static variable. A class should be self-contained, with as few out-side references as possible.

4.9.2   Initializing static variables

A static variable is usually initialized in the declaration. You can't easily initialize it in the class constructor, as this is called for every single new object. If you have a more elaborate initialization, you can use an initial block. Just make sure the static variables are initialized before the first object is constructed. In Example 4-10, the handle **s** is still **null** when the **initialize** task is called. This is legal, as the task only uses static variables that are not created in the constructor.

Example 4-10 Initializing a static variable in a task

```
class MyStatic;
  static int count;

  task initialize(int val);
    count = val;
  endtask
endclass

MyStatic s;
initial
  s.initialize(42);
```

4.10   Class Routines

A routine (a.k.a. method) in a class is just a **task** or **function** defined inside the scope of the class. Example 4-11 defines **display()** routines for the **BusTran** and **PCI_Tran**. SystemVerilog calls the correct one, based on the handle type.

Example 4-11 Routines in the class

```
class BusTran;
  bit [31:0] addr, crc, data[8];
  function void display();
    $display("@%0d: BusTran addr=%h, crc=%h",
             addr, crc);
    $write("\tdata[0-7]=");
    foreach (data[i]) $write(data[i]);
    $display();
  endfunction
endclass

class PCI_Tran;
  bit [31:0] addr, data;  // Use realistic names
  function void display();
    $display("@%0d: PCI: addr=%h, data=%h",
             addr, data);
  endfunction
endclass

BusTran b;
PCI_Tran pc;

initial begin
  b = new();      // Construct a BusTran
  b.display();    // Display a BusTran
  pc = new();     // Construct a PCT transaction
  pc.display();   // Display a PCI Transaction
end
```

A routine in a class always uses automatic storage, so you don't have to worry about remembering the **automatic** modifier.

4.11   Defining Routines Outside of the Class

A good rule of thumb is you should limit a piece of code to one "page" to keep it understandable. You may be familiar with this rule for routines, but it also applies to classes. If you can see everything in a class on the screen at one time, you can more easily understand it.

But if each routine takes a page, how can the whole class fit on a page? In SystemVerilog you can break a routine into the prototype (routine name and arguments) inside the class, and the body (the procedural code) that goes after the class.

Here is how you create out-of-block declarations. Copy the first line of the routine, with the name and arguments, and add the **extern** keyword at the beginning. Now take the entire routine and move it after the class body, and add the class name and two colons (**::** the scope operator) before the routine name.

The above classes could be defined as follows.

Example 4-12 Out-of-block routine declarations

```
class BusTran;
  bit [31:0] addr, crc, data[8];
  extern function void display();
endclass

function void BusTran::display();
  $display("@%0d: BusTran addr=%h, crc=%h",
           addr, crc);
  $write("\tdata[0-7]=");
  foreach (data[i]) $write(data[i]);
  $display();
endfunction

class PCI_Tran;
  bit [31:0] addr, data;  // Use realistic names
  extern function void display();
endclass

function void PCI_Tran::display();
  $display("@%0d: PCI: addr=%h, data=%h",
           addr, data);
endfunction
```

A common coding mistake is when the routine prototype does not match the one in the body. SystemVerilog requires that the prototype be identical to the out-of-block routine declaration, except for the class name and scope operator. Additionally, some OOP compilers (g++ and VCS) prohibit you from specifying the default argument values in both the prototype and the body. Since default argument values are important to code that calls a method, not to its implementation, they should only be present in class declaration.

Another common mistake is to leave out the class name when you declare the method. As a result, it is defined at the next higher scope (probably the program's), and the compiler gives an error when the task tries to access class-level variables and routines.

Example 4-13 Out-of-body task missing class name

```
class Broken;
  int id;
  extern function void display;
endclass

function void display; // Missing Broken::
  $display("Broken: id=%0d", id); // Error, id not found
endfunction
```

4.12   Scoping Rules

When writing your testbench, you need to create and refer to many variables. SystemVerilog follows the same basic rules as Verilog, with a few helpful improvements.

A scope is a block of code such as a module, program, task, function, class, or **begin-end** block. The **for** and **foreach** loops automatically create a block so that an index variable can be declared or created local to the scope of the loop.

You can define new variables in a block. New in SystemVerilog is the ability to declare a variable in an unnamed begin-end block, as shown in the **for** loops that declare the index variable.

A name can be relative to the current scope or absolute starting with **$root**. For a relative name, SystemVerilog looks up the list of scopes until it finds a match. If you want to be unambiguous, use **$root** at the start of a name.

Example 4-14 uses the same name in several scopes. Note that in real code, you would use more meaningful names! The name **limit** is used for a global variable, a program variable, a class variable, a task variable, and a local variable in an initial block. The latter is in an unnamed block, so the label created is tool dependent.

Example 4-14 Name scope

```
int limit;                    // $root.limit

program p;
  int limit;                  // $root.p.limit

  class Foo;
    int limit, array[];       // $root.p.Foo.limit

    task print (int limit); // $root.p.Foo.print.limit
      for (int i=0; i<limit; i++)
        $display("%m: array[%0d]=%0d", i, array[i]);
    endfunction
  endclass

  initial begin
    int limit = $root.limit; // $root.p.$unnamed.limit
    Foo bar;

    bar = new;
    bar.array = new[limit];
    bar.print (limit);
    end
endprogram
```

You can declare variables in the **program** or in the **initial** block. If a variable is only used inside a single **initial** block, such as a counter, you should declare it there to avoid possible name conflicts with other blocks.

Declare your classes outside of any **program** or **module** in a **package**. This approach can be shared by all the testbenches, and you can declare temporary variables at the innermost possible level. This style also eliminates a common bug that happens when you forget to declare a variable inside a class. SystemVerilog looks for that variable in higher scopes. If there is a variable with that name in the program block, the class uses it instead, with no warning. In Example 4-15, the function **Bug::display** did not declare the loop variable **i**, so SystemVerilog uses the program level **i** instead. Calling the function changes the value of **test.i**, which is probably not what you want!

Example 4-15 Class uses wrong variable

```
program test;
  int i;  // Program-level variable

  class Bug;
    logic [31:0] data[9];

    // Calling this function changes the program variable
    function void display;
      // Forgot to declare i in next statement
      for (i = 0; i<data.size; i++)
        $display("data[%0d]=%x", i, data[i]);
    endfunction
  endclass
endprogram
```

4.12.1   What is `this`?

When you use a variable name, SystemVerilog looks in the current scope for it, and then in the parent scopes until the variable is found. This is the same algorithm used by Verilog. But what if you are deep inside a class and want to unambiguously refer to a class-level object? This style code is most commonly used in constructors, where the programmer uses the same name for a class variable and an argument.[6] In Example 4-16, the keyword "`this`" removes the ambiguity to let SystemVerilog know that you are assigning the local variable, `oname`, to the class variable, `oname`.

Example 4-16 Using `this` to refer to class variable

```
class Scoping;
  string oname;

  function new(string oname);
    this.oname = oname;    // class oname = local oname
  endfunction

endclass
```

[6.] Some people think this makes the code easier to read; others think it is a shortcut by a lazy programmer.

4.12.2   Referring to a variable out of scope

Inside a class's routines, you can use local variables, class variables, or variables defined in the program. If you forget to declare a variable, SystemVerilog looks up the higher scopes until it finds a match. This can cause subtle bugs if two parts of the code are unintentionally sharing the same variable, perhaps because you forgot to declare it in the innermost scope.

For example, if you like to use the index variable, **i**, be careful that two different threads of your testbench don't concurrently modify this variable by each using it in a **for** loop. Or you may forget to declare a local variable in a class, such as **Buggy**, shown below. If your program block declares a global **i**, the class just uses the global instead of the local that you intended. You might not even notice this unless two parts of the program try to modify the shared variable at the same time. (Threads are covered in Chapter 7.)

Example 4-17 Bug using shared program variable

```
program bug;

  class Buggy;
    int data[10];
    task transmit;
      fork
        for (i=0; i<10; i++)   // i is not declared here
          send(data[i]);
      join_none
    endtask
  endclass

  int i;                       // program-level i
  Buggy b;
  event receive;

  initial begin
    b = new;
    for (i=0; i<10; i++) b.data[i] = i;
    b.transmit;

    for (i=0; i<10; i++)
      @(recieve) $display(data[i]);
    end
endprogram
```
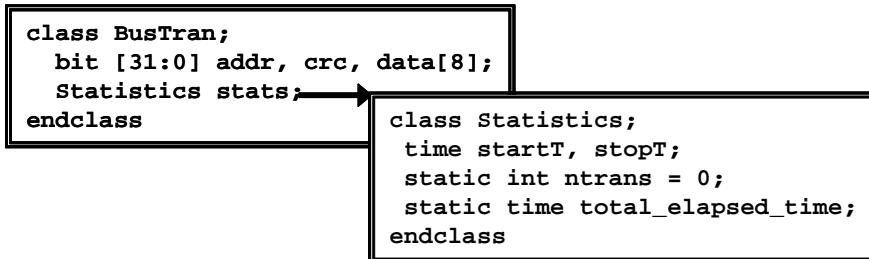
The solution is to declare all your variables in the smallest scope that encloses all uses of the variable. In Example 4-17, declare the index variables inside the **for** loops, not at the program or scope level.

4.13 Using One Class Inside Another

A class can contain an instance of another class, using a handle to an object. This is just like Verilog's concept of instantiating a module inside another module to build up the design hierarchy. Common reasons for using containment are reuse and controlling complexity.

For example, every one of your transactions may have a statistics block, with timestamps on when the transaction started and ended, and information about all transactions.

Figure 4-3 Contained objects

Example 4-18 shows the **Statistics** class.

Example 4-18 Statistics class declaration

```
class Statistics;
  time startT, stopT;        // Transaction time
  static int ntrans = 0;     // Transaction count
  static time total_elapsed_time = 0;

  function time how_long;
    how_long = stopT - startT;
    ntrans++;
    total_elapsed_time += how_long;
  endfunction

  function void start;
    startT = $time;
  endfunction
endclass
```

Now you can use this class inside another.

Example 4-19 Encapsulating the `Statistics` class

```
class BusTran;
  bit [31:0] addr, crc, data[8];
  Statistics stats;          // Statistics handle

  function new();
    stats = new();           // Make instance of stats
  endfunction

  task create_packet();
    // Fill packet with data
    stats.start();
    // Transmit packet
  endtask
endclass
```

The outermost class, **BusTran**, can refer to things in the **Statistics** class using the usual hierarchical syntax, such as **stats.start**.

Remember to instantiate the object; otherwise, the handle **stats** is **null** and the call to **start** fails. This is best done in the constructor of the outer class, **BusTran**.

As your classes become larger, they may become hard to manage. When your variable declarations and routine prototypes grow larger than a page, you should see if there is a logical grouping of items in the class so that it can be split into several smaller ones.

This is also a potential sign that it's time to refactor your code, i.e., split it into several smaller, related classes. See Chapter 8 for more details on class inheritance. Look at what you're trying to do in the class. Is there something you could move into one or more base classes, i.e., decompose a single class into a class hierarchy? A classic indication is similar code appearing at various places in the class. You need to factor that code out into a function in the current class, one of the current class's parent classes, or both.

4.13.1   How big or small should my class be?

Just as you may want to split up classes that are too big, you should also have a lower limit on how small a class should be. A class with just one or two members makes the code harder to understand as it adds an extra layer of hierarchy and forces you to constantly jump back and forth between the parent class and all the children to understand what it does. In addition, look at how often it is

used. If a small class is only instantiated once, you might want to merge it into the parent class.

One Synopsys customer put each transaction variable into its own class for fine control of randomization. The transaction had a separate object for the address, CRC, data, etc. In the end, this approach only made the class hierarchy more complex. On the next project they flattened the hierarchy.

See section 8.5 for more ideas on partitioning classes.

4.13.2   Compilation order issue

Sometimes you need to compile a class that includes another class that is not yet defined. The declaration of the included class's handle causes an error, as the compiler does not recognize the new type. Declare the class name with a **typedef** statement, as shown below.

Example 4-20 Using a `typedef class` statement

```
typedef class Statistics;  // Define lower level class

class BusTran;
  Statistics stats;        // Use Statistics class
  ...
endclass

class Statistics;          // Define Statistics class
...
endclass
```

4.14   Understanding Dynamic Objects
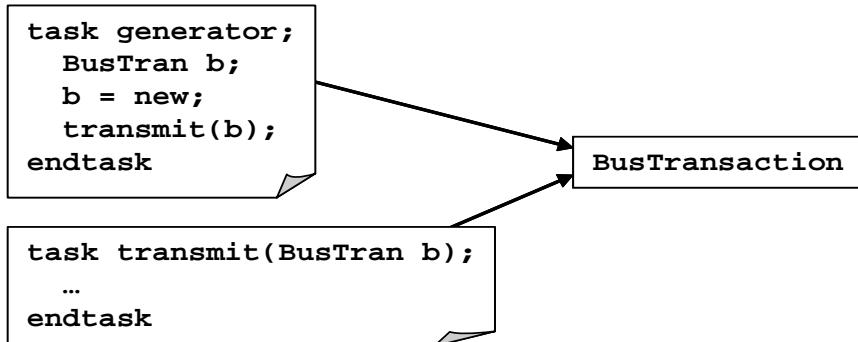
In a statically allocated language such as Verilog, every piece of data usually has a variable associated with it. For example, there may be a wire called **grant**, the integer **count**, and a module instance **i1**. In OOP, there is not the same one-to-one correspondence. There can be many objects, but only a few named handles. A testbench may allocate a thousand transaction objects during a simulation, but may only have a few handles to manipulate them. This situation takes some getting used to if you have only written Verilog code.

In reality, there is a handle pointing to every object. Some handles may be stored in arrays or queues, or in another object, like a linked list. For objects stored in a mailbox, the handle is in an internal SystemVerilog structure. See section 7.6 for more information on mailboxes.

4.14.1   Passing objects to routines

What happens when you pass an object into a routine? Perhaps the routine only needs to read the values in the object, such as **transmit** above. Or, your routine may modify the object, like a routine to create a packet. Either way, when you call the routine, you pass a handle to the object, not the object itself.

Figure 4-4 Handles and objects across routines

In Figure 4-4, the **generator** task has just called **transmit**. There are two handles, **generator.b** and **transmit.btrans**, that both refer to the same object.

When you call a routine with a scalar variable (nonarray, nonobject) and use the **ref** keyword, SystemVerilog passes the address of the scalar, so the routine can modify it. If you don't use **ref**, SystemVerilog copies the scalar's value into the argument variable, so any changes to the argument don't affect the original value.

Example 4-21 Passing objects

```
// Transmit a packet onto a 32-bit bus
task transmit(BusTran bt);
  CBbus.rx_data <= bt.data;
  bt.timestamp = $time;
  ...
endtask

BusTran b;
initial begin
  b = new();       // Allocate the object
  b.addr = 42;     // Initialize values
  transmit(b);     // Pass object to task
end
```

In Example 4-21, the initial block allocates a **BusTran** object and calls the **transmit** task with the handle that points to the object. Using this handle, **transmit** can read and write values in the object. However, if **transmit** tries to modify the handle, the result won't be seen in the initial block, as the **bt** argument was not declared as **ref**.

A routine can modify an object, even if the handle argument does not have a **ref** modifier. This frequently causes confusion for new users, as they mix up the handle with the object. As shown above, **transmit** can write a timestamp into the object. If you don't want an object modified in a routine, pass a copy of it so that the original data is untouched. See section 4.15 for more on copying objects.

4.14.2   Modifying a handle in a task

A common coding mistake is to forget to use **ref** on routine arguments that you want to modify, especially handles. In the following code, the argument **b** is not declared as **ref**, so any change to it is not be seen by the calling code.

Example 4-22 Bad packet creator task, missing ref on handle

```
task create_packet(BusTran bt); // Bug, missing ref
  bt = new();
  bt.addr = 42;
  // Initialize other fields
endtask

BusTran b;
initial begin
  create_packet(b);      // Call bad routine
  $display(b.addr);      // Fails because b=null
  end
```

Even though **create_packet** modified the argument **bt**, the handle **b** remains **null**. You need to declare the argument **bt** as **ref**.

Example 4-23 Good packet creator task with ref on handle

```
task create_packet(ref BusTran bt); // Good
  ...
endtask
```

4.14.3   Modifying objects in flight

A very common mistake is forgetting to create a new object for each transaction in the testbench. In Example 4-24, the **generate_trans** task creates a **BusTran** object with random values, and then transmits it into the design, which takes several cycles.

Example 4-24 Bad generator creates only one object

```
task generator_bad(int n);
  BusTran b;
  b = new();                      // Create one new object
  repeat (n) begin
    b.addr = $random();         // Initialize variables
    $display("Sending addr=%h", b.addr);
    transmit(p);                // Send it into the DUT
  end
endtask
```

What are the symptoms of this mistake? The code above creates only one **BusTran**, so every time through the loop, **generator_bad** changes the object at the same time it is being transmitted. When you run this, the **$display** shows many **addr** values, but all transmitted **BusTrans** have the same value of **addr**. The bug occurs if **transmit** stores the object and keeps using it even after **transmit** returns. If your **transmit** task does not keep a reference to the object, you can recycle the same object over and over.

You need to create a new **BusTran** during each pass through the loop.

Example 4-25 Good generator creates many objects

```
task generator_good(int n);
  BusTran b;
  repeat (n) begin
    b = new();                    // Create one new object
    b.addr = $random();         // Initialize variables
    $display("Sending addr=%h", b.addr);
    transmit(p);                // Send it into the DUT
  end
endtask
```

4.14.4   Arrays of handles

As you write testbenches, you need to be able to store and reference many objects. You can make arrays of handles, each of which refers to an object. Example 4-26 shows that storing of ten bus transactions in an array.

Example 4-26 Using an array of handles

```
task generator();
  BusTran barray[10];
  foreach (barray[i])
    begin
    barray[i] = new();        // Construct each object
    transmit(barray[i]);
    end
endtask
```

The array **barray** is made of handles, not objects. So you need to construct each object in the array before using it, just as you would for a normal handle. There is no way to call **new** on an entire array of handles

4.15   Copying Objects

You may want to make a copy of an object to keep a routine from modifying the original, or in a generator to preserve the constraints. You can either use the simple, built-in copy available with **new**, or you can write your own for more complex classes. Section 8.3 has more on why you should make a copy method.

4.15.1   Copying an object with **new**

Using **new** to copy an object is easy and reliable. A new object is constructed and all variables from the existing object are copied.

Example 4-27 Copying a simple class with new

```
class BusTran;
  bit [31:0] addr, crc, data[8];
endclass

BusTran src, dst;
initial begin
  src = new;          // Create first object
  dst = new src;      // Make a copy with new
  end
```

However, this is a shallow copy, similar to a photocopy of the original, blindly transcribing values from source to destination. If the class contains a handle to another class, only the top level object is copied by **new**, not the lower level one. In Example 4-28, the **BusTran** class contains a handle to the **Statistics** class, shown in Example 4-18.
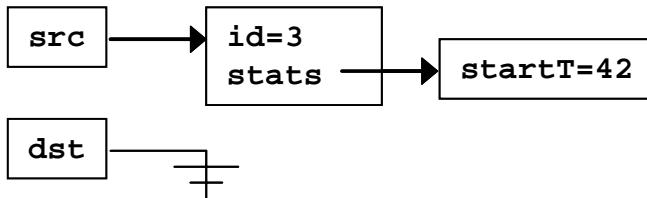
Example 4-28 Copying a complex class with new

```
class BusTran;
  bit [31:0] addr, crc, data[8];
  static int count = 0;
  int id;
  Statistics stats;

  function new;
    stats = new;
    id = count++;
  endfunction
endclass

BusTran src, dst;
initial begin
  src = new;                  // Create first object
  src.stats.startT = 42;
  dst = new src;              // Copy src to dst
  dst.stats.startT = 84;     // Changes stats for dst & src
  end
```
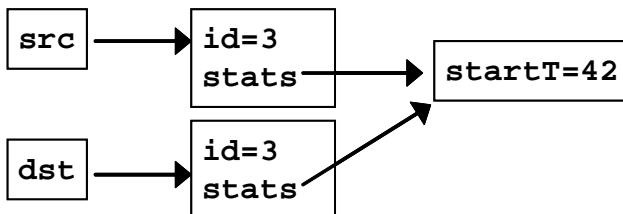
The initial block creates the first **BusTran** object and modifies a variable in the contained object **Statistics**.

Figure 4-5 Objects and handles before copy with new

When you call **new** to make a copy, the **BusTran** object is copied, but not the **Statistics** one. This is because when you use **new** to copy an object, it does not call your own **new** function. Instead, the values of variables and handles are copied. So now both **BusTran** objects point to the same **Statistics** object and both have the same **id**.

Figure 4-6 Objects and handles after copy with new

4.15.2   Writing your own simple copy function

If you have a simple class that does not contain any references to other classes, writing a **copy** function is easy.

Example 4-29 Simple class with copy function

```
class BusTran;
  bit [31:0] addr, crc, data[8];

  function BusTran copy;
    copy = new;              // Construct destination
    copy.addr = addr;        // Fill in data values
    copy.crc  = crc;
    copy.data = data;        // Array copy
  endfunction
endclass
```

Example 4-30 Using `copy` function

```
BusTran src, dst;
initial begin
  src = new;                    // Create first object
  dst = src.copy;
end
```

4.15.3   Writing your own deep copy function

For nontrivial classes, you should always create your own **copy** function. You can make it a deep copy by calling the copy functions of all the contained objects. Your own **copy** function makes sure all your user fields (such as an ID) remain consistent. The downside of making your own **copy** function is that you need to keep it up to date as you add new variables – forget one and you could spend hours debugging to find the missing value.[7]

Example 4-31 Complex class with deep copy function

```
class BusTran;
  bit [31:0] addr, crc, data[8];
  Statistics stats;
  static int count = 0;
  int id;

  function new;
    stats = new;
    id = count++;
  endfunction

  function BusTran copy;
    copy = new;                 // Construct destination
    copy.addr = addr;           // Fill in data values
    copy.crc  = crc;
    copy.data = data;
    copy.stats = stats.copy;  // Call copy for stats
    id = count++;
  endfunction
endclass
```
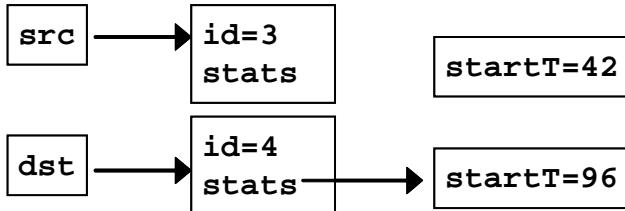
[7.] Perhaps the next version of SystemVerilog may include a deep object copy. However, this still does just a copy, so your constructor (`new` function) won't be called, and fields such as ID will not be updated.

Note that you also need to write a copy for the **Statistics** class, and every other class in the hierarchy.

Figure 4-7 Objects and handles after deep copy

4.16   Public vs. Private

The core of OOP is to encapsulate data and related routines into a class. Data is kept private by default to keep one class from poking around inside another. A class provides a set of accessor routines to access and modify the data. This would also allow you to change the class's implementation without needing to let the users of the class know. For instance, a graphics package could change its internal representation from Cartesian coordinates to polar as long as the user interface (accessor routines) have the same functionality.

Consider the **BusTran** class that has a payload and a CRC so that the hardware can detect errors. In conventional OOP, you would make a routine to set the payload also set the CRC so they would stay synchronized. Thus your objects would always be filled with correct values.

However, testbenches are not like other programs, such as a web browser or word processor. A testbench needs to create errors. You want to have a bad CRC so you can test how the hardware reacts to errors.

OOP languages such as C++ and Java allow you to specify the visibility of variables and routines. By default, everything is private unless labeled otherwise.

In SystemVerilog, everything is public unless labeled private. You should stick with this default so you have the greatest control over the operation of the DUT, which is more important than long-term software stability. For example, making the CRC visible allows you to easily inject errors into the DUT. If the CRC was private, you would have to write extra code to bypass the data-hiding mechanisms, resulting in a larger and more complex testbench.
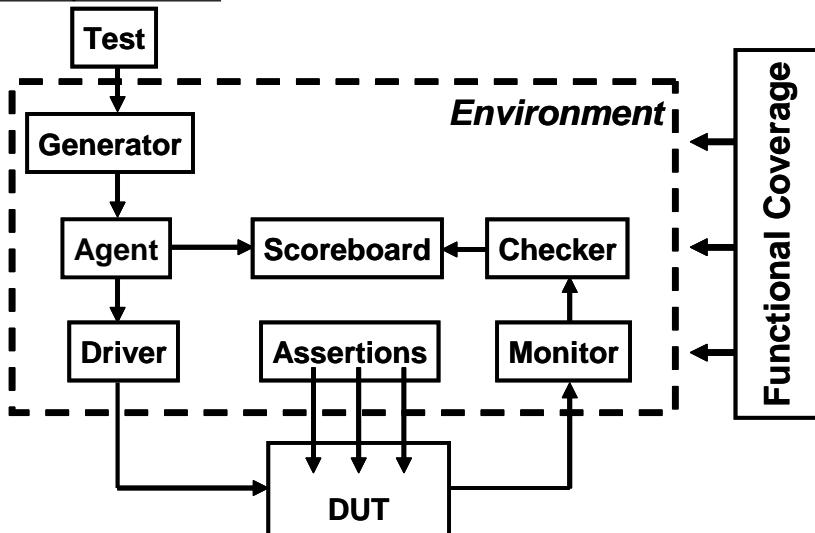
4.17   Straying Off Course

As a new OOP student, you may be tempted to skip the extra thought needed to group items into a class, and just store data in a few variables. Avoid the temptation! A basic DUT monitor samples several values from an interface. Don't just store them in some integers and pass them to the next stage. This saves you a few minutes at first, but eventually you need to group these values together to form a complete transaction. Several of these transactions may need to be grouped to create a higher-level transaction such as a DMA transfer. Instead, immediately put those interface values into a transaction class. Now you can store related information (port number, receive time) along with the data, and easily pass this object to the rest of your testbench.

4.18   Building a Testbench

You are closer to creating a simple testbench from classes. Here is the diagram from Chapter 1. Obviously, the transactions in Example 4-8 are objects, but each block is represented as a class also.

Figure 4-8 Layered testbench

The **Generator**, **Agent**, **Driver**, **Monitor**, **Checker**, and **Scoreboard** are all classes, modeled as transactors (described below). They are instantiated inside the Environment class. For simplicity, the test is at the top of the hierarchy, as is the program that instantiates the Environment class. The Functional coverage definitions can be put inside or outside the Environment class.

A transactor is made of a simple loop that receives a transaction object from a previous block, makes some transformations, and sends it to the following one. Some, such as the **Generator**, have no upstream block, so this transactor constructs and randomizes every transaction, while others, such as the **Driver**, receive a transaction and send it into the DUT as signal transitions.

Example 4-32 Basic Transactor

```
class Transactor;  // Generic class
  Transaction tr;

  task run;
    forever begin
      // Get transaction from upstream block
      ...
      // Do some processing
      ...
      // Send it to downstream block
      ...
      end
  endtask

endclass
```

How do you exchange transactions between blocks? With procedural code you could have one object call the next, or you could use a data structure such as a FIFO to hold transactions in flight between blocks. In Chapter 7, you will learn how to use mailboxes, which are FIFOs with the ability to stall a thread until there is data available.

4.19   Conclusion

Using Object Oriented Programming is a big step, especially if your first computer language was Verilog. The payoff is that your testbenches are more modular and thus easier to develop, debug, and reuse.

Have patience — your first OOP testbench may look more like Verilog with a few classes added. But as you get the hang of this new way of thinking, you begin to create and manipulate classes for both transactions and the transactors in the testbench that manipulate them.

In Chapter 8 you will learn more OOP techniques so your test can change the behavior of the underlying testbench without having to change any of the existing code.

Chapter 5
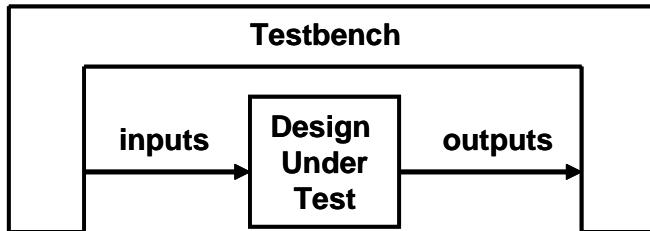
Connecting the Testbench and Design

5.1   Introduction

There are several steps needed to verify a design: generate stimulus, capture responses, determine correctness, and measure progress. But first, you need the proper testbench, connected to the design.

Your testbench wraps around the design, sending in stimulus and capturing the design's response. The testbench forms the "real world" around the design, mimicking the entire environment. For example, a processor model needs to connect to various busses and devices, which are modeled in the testbench as bus functional models. A networking device connects to multiple input and output data streams that are modeled based on standard protocols. A video chip connects to buses that send in commands, and then forms images that are written into memory models. The key concept is that the testbench simulates everything not in the design under test.

Figure 5-1 The testbench – design environment

Your testbench needs a higher-level way to communicate with the design than Verilog's ports and the error-prone pages of connections. You need a robust way to describe the timing so that synchronous signals are always driven and sampled at the correct time and all interactions are free of the race conditions so common to Verilog models.

5.2   Separating the Testbench and Design

In an ideal world, all projects have two separate groups: one to create the design and one to verify it. In the real world, limited budgets may require you to wear both hats. Each team has its own set of specialized skills, such as creating synthesizable RTL code, or figuring out new ways to find bugs in the design. These two groups each read the original design specification and make their own interpretations. The designer has to create code that meets that

spec, while you, the verification engineer, have to find ways to prove the design does not match the spec.

Likewise, your testbench code is in a separate block from design code. In classic Verilog, each goes in a separate module. However, using a module to hold the testbench often causes timing problems around driving and sampling, so SystemVerilog introduces the program block to separate the testbench, both logically and temporally. For more details, see section 5.4.
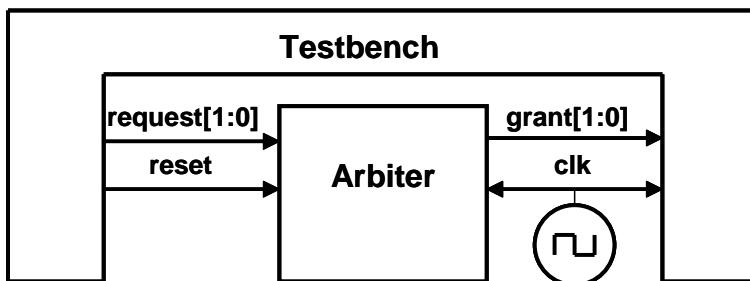
As designs grow in complexity, the connections between the blocks increase. Two RTL blocks may share dozens of signals, which must be listed in the correct order for them to communicate properly. One mismatched or misplaced connection and the design will not work. If it is a subtle bug, such as swapping pins that only toggle occasionally, you may not notice the problem for some time. Worse yet is when you add a new signal between two blocks. You have to edit not only the blocks to add the new port but also the higher-level netlists that wire up the devices. Again, one wrong connection at any level and the design stops working. Or worse, the system only works intermittently!

The solution is the interface, the SystemVerilog construct that represents a bundle of wires, with intelligence such as synchronization, and functional code. An interface can be instantiated like a module but also connected to ports like a signal.

5.2.1   Communication between the testbench and DUT

The next few sections show a testbench connected to an arbiter, using individual signals and again using interfaces. Here is a diagram of the top level design including a testbench, arbiter, clock generator, and the signals that connect them. This is a trivial design, so you can concentrate on the SystemVerilog concepts and not get bogged down in the design. At the end of the chapter, an ATM router is shown.

Figure 5-2 Testbench – Arbiter without interfaces

5.2.2   Communication with ports

The following code fragments show the elements of connecting an RTL block to a testbench. First is the header for the arbiter model. This uses the Verilog-2001 style port declarations where the type and direction are in the header. Most of the code and declarations have been left out for clarity.

As discussed in section 2.2.1, SystemVerilog has expanded the classic **reg** type so that you can use it like a **wire** to connect blocks. In recognition of its new capabilities, the **reg** type has the new name of **logic**. The only place where you cannot use a **logic** variable is a net with multiple drivers, where you must use a net such as **wire**.

Example 5-1 Arbiter model using ports

```
module arb_port (output logic [1:0] grant,
                 input  logic [1:0] request,
                 input  logic reset,
                 input  logic clk);
...
  always @(posedge clk or posedge reset) begin
    if (reset)
      grant <= 2'b00;
    else
      ...
  end
endmodule
```

The testbench is a module with ports. A small piece of the test is shown.

Example 5-2 Testbench using ports

```
module test (input  logic [1:0] grant,
             output logic [1:0] request,
             output logic reset,
             input  logic clk);

  initial begin
    @(posedge clk)        request <= 2'b01;
    $display("@%0d: Drove req=01", $time);
    repeat (2) @(posedge clk);
    if (grant != 2'b01)
      $display("@%0d: a1: grant != 2'b01", $time);
    ...
    $finish;
  end
endmodule
```

The top netlist connects the testbench and DUT.

Example 5-3 Top-level netlist without an interface

```
module top;
  logic [1:0] grant, request;
  logic  clk=0, reset;
  always #5 clk = ~clk;

  arb_port a1 (grant, request, reset, clk);
  test     t1(grant, request, reset, clk);
endmodule
```

In Example 5-3, the netlists are simple, but real designs with hundreds of pins require pages of signal and port declarations. All these connections can be error prone. As a signal moves through several layers of hierarchy, it has to be declared and connected over and over. Worst of all, if you just want to add a new signal, it has to be declared and connected in multiple files. SystemVerilog interfaces can help in each of these cases.

5.3   The Interface Construct
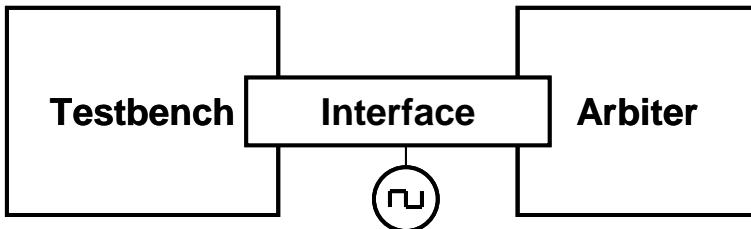
Designs are so complex that even the communication between them may need to be separated out into separate entities. To model this, SystemVerilog uses the interface construct that you can think of as an intelligent bundle of wires. They contain the connectivity, synchronization, and optionally, the functionality of the communication between two or more blocks. They connect design blocks and/or testbenches.

Basic, design-level interfaces are covered in Sutherland (2004). This book covers interfaces to connect design blocks to testbenches.

5.3.1   Using an interface to simplify connections

The first improvement is to bundle the wires together into an interface block. Figure 5-3 shows the testbench and arbiter, communicating using a interface. Note how the interface extends into the two blocks, representing the drivers and receivers that are functionally part of both the test and the DUT. The clock can be part of the interface or a separate port.

Figure 5-3 An interface straddles two modules

The simplest interface is just a bundle of nondirectional signals. Use **logic** so you can drive the signals from procedural statements.

Example 5-4 Simple interface for arbiter

```
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  logic reset;
endinterface
```

This is instantiated in the **top** module and connected as follows.

Example 5-5 Top module using a simple arbiter interface

```
module top;
  bit  clk;
  always #5 clk = ~clk;

  arb_if arbif(clk);
  arb a1 (arbif);
  test t1(arbif);
endmodule : top
```

Example 5-6 shows the testbench. You refer to a signal in an interface by making a hierarchical reference using the instance name: **arbif.request**. Interface signals should always be driven using nonblocking assignments. This is explained in more detail in section 5.5.3.

Example 5-6 Testbench using a simple arbiter interface

```
module test (arb_if arbif);
...
  initial begin
    // reset code left out

    @(posedge arbif.clk);
    arbif.request <= 2'b01;
    $display("@%0d: Drove req=01", $time);
    repeat (2) @(posedge arbif.clk);
    if (arbif.grant != 2'b01)
      $display("@%0d: a1: grant != 2'b01", $time);

    $finish;
  end
endmodule : test
```

Lastly is the device under test, the arbiter, that uses an interface instead of ports.

Example 5-7 Arbiter using a simple interface

```
module arb (arb_if arbif);
...
  always @(posedge arbif.clk or posedge arbif.reset)
    begin
    if (arbif.reset)
      arbif.grant <= 2'b00;
    else
      arbif.grant <= next_grant;
    ...
    end
endmodule
```

You can see an immediate benefit, even on this small device: the connections become cleaner and less prone to mistakes. If you wanted to put a new signal in an interface, you would just have to add it to the interface definition and the modules that actually used it. You would not have to change any module such as **top** that just pass the interface through. This language feature greatly reduces the chance for wiring errors.

5.3.2   Connecting interfaces and ports

If you have a legacy design with ports that cannot be changed to use an interface, you can just connect the interfaces's signals to the individual ports.

Example 5-8 connects the original arbiter from Example 5-1 to the interface in Example 5-4.

Example 5-8 Connecting an interface to a module that uses ports

```
module top;
  bit  clk;
  always #5 clk = ~clk;

  arb_if arbif(clk);
  arb_port a1 (.grant    (arbif.grant),
               .request (arbif.request),
               .reset   (arbif.reset),
               .clk     (arbif.clk));
  test t1(arbif);
endmodule : top
```

5.3.3   Grouping signals in an interface using modports

Example 5-7 uses a point-to-point connection scheme with no signal directions in the interface. The original netlists using ports had this information that the compiler uses to check for wiring mistakes. The **modport** construct in an interface lets you group signals and specify directions. The MONITOR modport allows you to connect a monitor module.

Example 5-9 Interface with modports

```
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  logic reset;

  modport TEST (output request, reset,
                input  grant, clk);

  modport DUT (input request, reset, clk,
               output grant);

  modport MONITOR (input request, grant, reset, clk);

endinterface
```

Here are the arbiter model and testbench, which need to specify the modport in their port connection list. Note that you put the modport name, **DUT** or **TEST**, after the interface name, **arb_if**. Other than the modport name, these are identical to the previous examples.

Example 5-10 Arbiter model with interface using modports

```
module arb (arb_if.DUT arbif);
...
endmodule
```

Example 5-11 Testbench with interface using modports

```
module test (arb_if.TEST arbif);
...
endmodule
```

The top model does not change from Example 5-5, as modports are speci-fied in the module header, not when the module is instantiated.

While the code didn't change much (except that the interface grew larger), this interface more accurately represents the real design, especially the signal direction.

5.3.4   Using modports with a bus design

Not every signal needs to go in every interface. Consider a CPU – memory bus modeled with an **interface**. The CPU is the bus master and drives a subset of the signals, such as **request**, **command**, and **address**. The mem-ory is a slave and receives those signals and drives **ready**. Both master and slave drive **data**. The bus arbiter only looks at **request** and **grant**, and ignores all other signals. So your interface would have three modports for master, slave, and arbiter, plus an optional monitor **modport**.

5.3.5   Creating an interface monitor

You can create a bus monitor using the **MONITOR modport**. The following is a trivial monitor for the arbiter. For a real bus, you could decode the com-mands and print the status: completed, failed, etc.

Example 5-12 Arbiter model with interface using modports

```
module monitor (arb_if.MONITOR arbif);

  always @(posedge arbif.request[0]) begin
    $display("@%0d: request[0] asserted", $time);
    @(posedge arbif.grant[0]);
    $display("@%0d: grant[0] asserted", $time);
  end

  always @(posedge arbif.request[1]) begin
    $display("@%0d: request[1] asserted", $time);
    @(posedge arbif.grant[1]);
    $display("@%0d: grant[1] asserted", $time);
  end
endmodule
```

5.3.6   Interface trade-offs

An interface cannot contain any instances such as modules or other interfaces. There are trade-offs in using interfaces with modports as compared with traditional port connects with signals.

The advantages to using an interface are as follows.

- An interface is ideal for design reuse. When two blocks communicate with a specified protocol using two or more signals, use an interface. If signals are repeated over and over, as in a networking switch, use a virtual interface as described in Chapter 10.

- The interface takes the jumble of signals that you declare over and over in every module or program and puts it in a central location, reducing the possibility of misconnecting signals.

- To add a new signal, you just have to declare it once in the interface, not in higher-level modules, once again reducing errors.

- Modports allow a module to easily tap a subset of signals from an interface. You can specify signal direction for additional checking.

The disadvantages of using an interface are as follows.

- For point-to-point connections, interfaces with modports are almost as verbose as using ports with lists of signals. But all the declarations

are still in one central location, reducing the chance for making an error.

- You must now use the interface name in addition to the signal name, possibly making the modules more verbose.

- If you are connecting two design blocks with a unique protocol that will not be reused, interfaces may be more work than just wiring together the ports.

- It is difficult to connect two different interfaces. A new interface (`bus_if`) may contain all the signals of an existing one (`arb_if`), plus new signals (address, data, etc.). But since interfaces cannot be hierarchical, you have to break out the individual signals and drive them appropriately.

5.3.7   More information and examples

All these examples use SystemVerilog constructs. The SystemVerilog LRM specifies many other ways for you to use interfaces. See Sutherland (2004) for more examples of using interfaces for design.

5.4   Stimulus Timing

The timing between the testbench and the design must be carefully orchestrated. At a cycle level, you need to drive and receive the synchronous signals at the proper time in relation to the clock. Drive too late or sample too early, and your testbench is off a cycle. Even within a single time slot (for example, everything that happens at time 100ns), mixing design and testbench events can cause a race condition, such as when a signal is both read and written at the same time. Do you read the old value, or the one just written? In Verilog, nonblocking assignments helped when a test module drove the DUT, but the test could not always be sure it sampled the last value driven by the design. SystemVerilog has several constructs to help you control the timing of the communication.

5.4.1   Controlling timing of synchronous signals with a clocking block

An interface block uses a clocking block to specify the timing of synchronous signals relative to the clocks. Any signal in a clocking block is now driven or sampled synchronously, ensuring that your testbench interacts with the signals at the right time. Clocking blocks are mainly used by testbenches but also allow you to create abstract synchronous models.

An interface can contain multiple clocking blocks, one per clock domain, as there is single clock expression in each block. Typical clock expressions are **@(posedge clk)** or **@(posedge clk1 or negedge clk2)**.

You can specify a clock skew in the clocking block using the **default**-statement, but the default behavior is that input signals are sampled just before the design executes, and the outputs are driven back into the design during the current time slot. The next section provides more details on the timing between the design and testbench.

Once you have defined a clocking block, your testbench can wait for the clocking expression with **@my_interface.cb** rather than having to spell out the exact clock and edge. Now if you change the clock or edge in the clocking block, you do not have to change your testbench.

In Example 5-13, the clocking block **cb** declares that the signals in the block are active on the positive edge of the clock. The signal directions are relative to the **modport** where they are used. So **request** is an output of the TEST **modport**, and **grant** is an input to the **modport**.

Example 5-13 Interface with a clocking block

```
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  logic reset;

  clocking cb @(posedge clk);    // Declare cb
    output request;
    input grant;
  endclocking

  modport TEST (clocking cb,     // Use cb
                output reset);

  modport DUT (input request, reset, output grant);
endinterface
```

5.4.2   Timing problems in Verilog

Your testbench needs to be separate from the design, not just logically but also temporally. Consider how a hardware tester interacts with a chip for synchronous signals. In a real hardware design, the storage elements latch their inputs at the active clock edge. These values propagate to the storage outputs, and then through the logic clouds to the inputs of the next storage elements. The time from the input of the first storage to the next must be less than a clock cycle. So a hardware tester needs to drive the chip's input just after the clock edge, and read the outputs just before the following edge.

A testbench has to mimic this behavior, as if it were another storage element plus logic. It should drive after the active clock edge, and should sample as late as possible as allowed by the protocol timing specification, just before the active clock edge.

If the DUT and testbench are made of Verilog modules only, this outcome is nearly impossible to achieve. If the testbench drives the DUT at the clock edge, there could be race conditions. What if the clock propagates to some DUT inputs before the TB stimulus, but is a little later to other inputs? From the outside, the clock edges all arrive at the same simulation time, but in the design, some inputs get the value driven during the last cycle, while other inputs get values from the current cycle.

One way around this problem is to add small delays to the system, such as `#0`. This forces the thread of Verilog code to stop and be rescheduled after all other code. But, invariably, a large design has several sections that all want to execute last. Whose `#0` wins out? It could vary from run to run and be unpredictable between simulators. Multiple threads using `#0` delays cause indeterministic behavior.

The next solution is to use a larger delay, `#1`. RTL code has no timing, other than clock edges, so one time unit after the clock, the logic has settled. But what if one module uses a time precision of 1ns, while another used a resolution of just 10ps? Does that `#1` mean 1ns, 10ps, or something else? You want to drive as soon as possible after the clock cycle with the active clock edge, but not during that time, and before anything else can happen. Worse yet, your DUT may contain a mix of RTL code with no delays and gate code with delays.

5.4.3   Testbench – design race condition

Example 5-14 shows a potential race condition between the testbench and design. The race condition occurs when the test drives the **start** signal and then the other ports. The memory is waiting on the **start** signal and could wake up immediately, while the **write**, **addr**, and **data** signals still have their old values. You could delay all these signals slightly by using nonblocking assignments, as recommended by Cummings (2000), but remember that the testbench and the design are both using these assignments. It is still possible to get a race condition between the testbench and design.

Sampling the design outputs has a similar problem. You want to grab the values at the last possible moment, just before the active clock edge. Perhaps you know the next clock edge is at 100ns. You can't sample right at the clock edge at 100ns, as some design values may have already changed. You should sample at **Tsetup** just before the clock edge.

Example 5-14 Race condition between testbench and design

```
module memory(input wire start, write,
              input wire [7:0] addr, data);
  logic [7:0] mem[256];
  always @(posedge start) begin
    if (write)
      mem[addr] = data;
    ...
  end
endmodule

module test(output logic start, write,
            output logic [7:0] addr, data);
  initial begin
    start = 0;                // Initialize signals
    write = 0;
    #10;                      // Short delay
    start = 1;                // Start first command
    write = 1;
    addr = 8'h42;
    data = 8'h5a;
    ...
  end
endmodule
```

5.4.4   The program block and timing regions
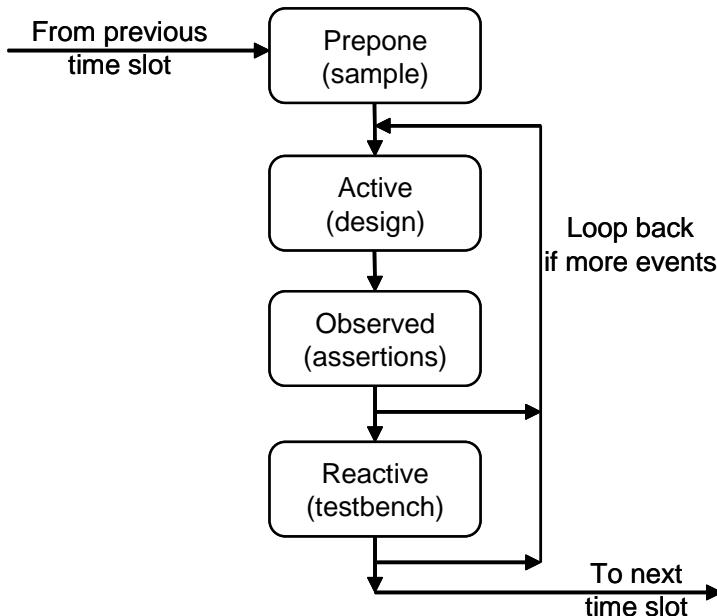
The root of the problem is the mixing of design and testbench events during the same time slot (though even in pure RTL the same problem can happen). What if there were a way you could separate these events temporally, just as you separated the code? At 100ns, your testbench could sample the design outputs before the clock has had a chance to change and any design activity has occurred. By definition, these values would be the last possible ones from the previous time slot. Then, after all the design events are done, your testbench would start.

How does SystemVerilog know to schedule the testbench events separately from the design events? In SystemVerilog, your testbench code is in a program block, which is similar to a module in that it can contain code and variables and be instantiated in other modules. However, a program cannot have any hierarchy such as instances of modules, interfaces, or other programs.

This new division of the time slot was introduced in SystemVerilog. In Verilog, most events executed in the Active region. There are other regions

for nonblocking assignments, PLI execution, etc., but they can be ignored for the purposes of this book. See the IEEE 1800-2005 Language Reference Manual for more details.

Figure 5-4 Main regions inside a SystemVerilog time step

In SystemVerilog, several new regions were introduced. First to execute during a time slot is the Prepone region, which samples signals before any design activity. These samples are used by the testbench. Next is the Active region, where design events run. These include your RTL and gate code plus the clock generator. The third region is the Observed region, where assertions are evaluated. Following that is the Reactive region where the testbench executes. Note that time does not strictly flow forwards — events in the Observed and Reactive regions can trigger further design events in the Active region in the current cycle.

Table 5-1. Primary SystemVerilog scheduling regions

Name	*Activity*
Prepone	Sampling signals before design activity. For testbench input
Active	Simulation of design code in modules

Table 5-1. Primary SystemVerilog scheduling regions

Name	Activity
Observed	Evaluation of SystemVerilog Assertions
Reactive	Execution of testbench code in programs

In Verilog, simulation continues while there are scheduled events, or until a **$finish** is executed. In SystemVerilog there is an implicit **$exit** when the program block terminates — that is, when the all **initial**-blocks complete. An **$exit** just terminates the current program block, while **$finish** terminates the entire simulation. When all program blocks end, your simulation ends, even if there are events pending on the design side or if spawned threads in the testbench are still running.

Example 5-15 shows part of the testbench code for the arbiter. Note that the statement **@arbif.cb** waits for the active edge of the clocking block, **@(posedge clk)**, as shown in Example 5-13.

Example 5-15 Testbench using interface with clocking block

```
program automatic test (arb_if.TEST arbif);
...
  initial begin
    arbif.cb.request <= 2'b01;
    $display("@%0d: Drove req=01", $time);
    repeat (2) @arbif.cb;
    if (arbif.cb.grant != 2'b01)
      $display("@%0d: a1: grant != 2'b01", $time);
    end

endprogram : test
```

Section 5.5 explains more about the driving and sampling of interface signals.
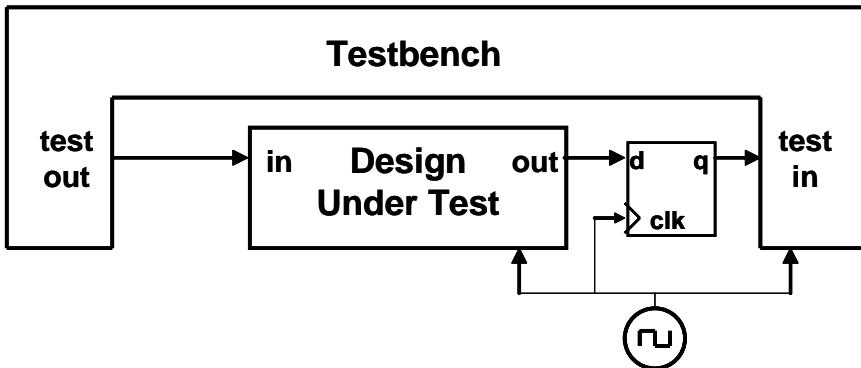
As discussed in section 3.7.1, you should always declare your program block as **automatic** so that it behaves more like the routines in stack-based languages you may have worked with, such as C.

5.4.5 Specifying delays between the design and testbench

The default timing of the clocking block is to sample inputs with a delay of **#1step** and to drive the outputs with a delay of **#0**. The **1step** delay specifies that signals are sampled in the Prepone region, before any design

activity. So you get the output values just before the clock changes. The test-bench outputs are synchronous by virtue of the clocking block, so they flow directly into the design. The program block, running in the Reactive region, retriggers the Active region during the same time slot. If you have a design background, you can remember this by imagining that the clocking block inserts a synchronizer between the design and testbench.

Figure 5-5 A clocking block synchronizes the DUT and testbench

5.5   Interface Driving and Sampling

Your testbench needs to drive and sample signals from the design, prima-rily through interfaces with clocking blocks. The following section uses the arbiter interface from Example 5-13.

Asynchronous signals such as **reset** pass through the interface with no delays. The signals in the clocking block get synchronized as shown below.

5.5.1   Interface synchronization

You can use the Verilog **@** and **wait** constructs to synchronize with the signals in a testbench. The following code does not do anything useful except to show the various constructs.

Example 5-16 Signal synchronization

```
program automatic test(bus_if.TB bus);
  initial begin
    @bus.cb;                       // Continue on active edge
                                   // in clocking block
    repeat (3) @bus.cb;       // Wait for 3 active edges
    @bus.cb.grant;             // Continue on any edge
    @(posedge bus.cb.grant); // Continue on posedge
    @(negedge bus.cb.grant); // Continue on negedge
    wait (bus.cb.grant==1);  // Wait for expression
                                   // No delay if already true
    @(posedge bus.cb.grant or
      negedge bus.reset);     // Wait for several signals
  end
endprogram
```

5.5.2   Interface signal sample

When you read a signal from a clocking block, you get the sample from just before the last clock edge, i.e., from the Prepone region. The following code shows a program block that reads the synchronous grant signal from the DUT. The **arb** module drives **grant** to 1 in the middle of a cycle, and then to 2 at the clock edge.

Example 5-17 Synchronous interface sample and module drive

```
program test(arb_if.TEST arbif);
  initial begin
    $monitor("@%0d: grant=%h", $time, arbif.cb.grant);
    #50;
  end
endprogram

module arb(arb_if.DUT arbif);
  initial begin
        arbif.grant = 1;  // @ 0ns
    #12 arbif.grant = 2;  // @ 12ns
    #18 arbif.grant = 2;  // @ 30ns
  end
endmodule
```
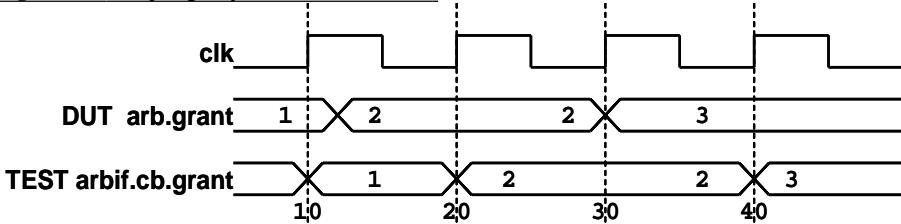
The waveforms show that in the program, **arbif.cb.grant** gets the value from just before the clock edge. When the interface input changes right

at a clock edge (time 30ns), that value does not propagate to the testbench for another cycle (time 40ns).

Figure 5-6 Sampling a synchronous interface

5.5.3   Interface signal drive

Here is a short version of the arbiter test program.

Example 5-18 Testbench using interface with clocking block

```
program automatic test (arb_if.TEST arbif);

  initial begin
    arbif.cb.request <= 2'b01;
    $display("@%0d: Drove req=01", $time);
    repeat (2) @arbif.cb;
    if (arbif.cb.grant != 2'b01)
      $display("@%0d: a1: grant != 2'b01", $time);
    end

endprogram : test
```

Any synchronous signal such as **request** must be prefixed with both the interface name (**arbif**) and the clocking block name (**cb**) in case the signal is used in more than one clocking block. So **arbif.cb.grant** is legal, but **arbif.grant** is not. This is a very common coding mistake and is caught by any SystemVerilog compiler.

You should always drive interface signals with a nonblocking assignment, namely the **<=** operator.[8] This is because the design signal does not change immediately after your assignment – remember that your testbench executes

---

[8.] There are some cases where you could use a blocking assignment, such as the force statement. The LRM is not clear about how a program block can force an interface signal or a signal in the design. Additionally, if an interface signal is passed through a ref argument into a routine, should the routine use a blocking or nonblocking assignment? SystemVerilog is an evolving language.

in the Reactive region while design code is in the Active region. If your test-
bench drives **arbif.cb.request** at 100ns, the same time as **arbif.cb**
(which is **@(posedge clk)** according to the clocking block), **request**
changes in the design at 100ns. But if your testbench tries to drive
arbif.cb.request at time 101ns, between clock edges, the change does
not propagate until the next clock edge. In this way, your drives are always
synchronous. In Example 5-17, **arbif.grant** is driven by a module and can
use a blocking assignment.

   If the testbench drives the synchronous interface signal at the active edge
of the clock, the value propagates immediately to the design. This is because
the default output delay is **#0** for a clocking block. If the testbench drives the
output just after the active edge, the value is not seen in the design until the
next active edge of the clock.

Example 5-19 Interface signal drive
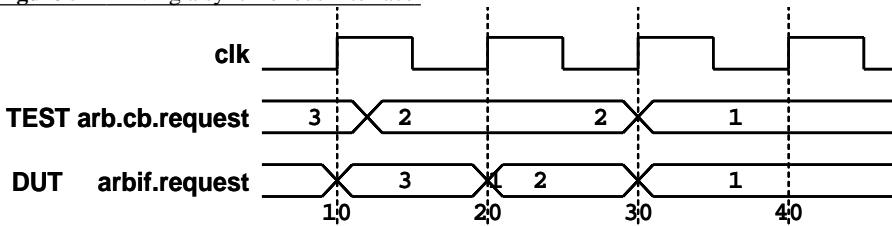
```
busif.cb.request <= 1;      // Nonblocking sync drive
busif.cb.cmd <= cmd_buf;    // Nonblocking sync drive
```

   Example 5-20 shows what happens if you drive a synchronous interface
signal at various points during a clock cycle.

Example 5-20 Driving a synchronous interface

```
program test(arb_if.TEST arbif);
  initial begin
    # 2 arbif.cb.request <= 3;  // @ 2ns
    #10 arbif.cb.request <= 2;  // @ 12ns
    #18 arbif.cb.request <= 1;  // @ 30ns
    #50 finish;
  end
endprogram

module arb(arb_if.DUT arbif);
  initial
    $monitor("@%0d: req=%h", $time, arbif.request);
endmodule
```

Figure 5-7 Driving a synchronous interface

If you want to wait for two clock cycles before driving a signal, you can either use "`repeat (2) @bus.cb;`" or use the cycle delay `##2`. This delay only works as a prefix to a drive of a signal in a clocking block, as it need to know which clock to use for the delay.

Example 5-21 Interface signal drive

```
##2 arbif.cb.request <= 0;  // Wait 2 cycles then assign
##3;  // Illegal - must be used with an assignment
```

5.5.4   Bidirectional signals in the interface

In Verilog, if you want to drive a bidirectional signal such as a port from procedural code, you need a continuous assignment to connect the **reg** to the **wire**. In SystemVerilog, synchronous bidirectional signals in interfaces are easier to use as the continuous assignment is added for you. When you write to the net from a program, SystemVerilog actually writes to a temporary variable that drives the net. Your program reads directly from the wire, seeing the value that is resolved from all the drivers. Design code in a module still uses the classic register plus continuous assignment statement.

Example 5-22 Bidirectional signals in a program and interface

```
interface master_if (input bit clk);
  wire [7:0] data;  // Bidirectional signal

  clocking cb @(posedge clk);
  inout data;
  endclocking

  modport TEST (clocking cb);
endinterface


program test(master_if mif);

  initial begin
    mif.cb.data <= 'z;          // Tri-state the bus
    @mif.cb;
    $displayh(mif.cb.data);     // Read from the bus
    @mif.cb;
    mif.cb.data <= 7'h5a;       // Drive the bus
    @mif.cb;
    mif.cb.data <= 'z;          // Release the bus
  end

endprogram
```

The SystemVerilog LRM is not clear on driving an asynchronous bidirectional signal using an interface. Two possible solutions are to use a cross-module reference and continuous assignment or to use a virtual interface as shown in Chapter 10.

5.5.5 Why are `always` blocks not allowed in a program?

In SystemVerilog you can put tasks, functions, classes, and `initial` blocks in a program, but not `always` blocks. This may seem odd if you are used to Verilog modules, but there are several reasons. SystemVerilog programs are closer to a program in C, with one (or more) entry points, than Verilog's many small blocks of concurrently executing hardware. In a design, an `always` block might trigger on every positive edge of a clock from the start of simulation. A testbench, on the other hand, goes through initialization, drive and respond to design activity, and then completes. When the last `initial` block completes, simulation implicitly ends just as if you had executed `$finish`. If you had an `always` block, it would never stop, so you

would have to explicitly call **$exit** to signal that the program block completed.

But don't despair. If you really need an **always** block, you can use "**initial forever**" to accomplish the same thing.

5.5.6   The clock generator

Now that you have seen the program block, you may wonder if the clock generator should be in a module. The clock is more closely tied to the design than the testbench, and so the clock generator should remain in a module. As you refine the design, you create clock trees, and you have to carefully control the skews as the clocks enter the system and propagate through the blocks.

The testbench is much less picky. It just wants a clock edge to know when to drive and sample signals. Functional verification is concerned with providing the right values at the right cycle, not with fractional nanosecond delays and relative clock skews.

Example 5-23 Bad clock generator in program block

```
program bad_generator (output bit clk, out_sig);
  bit clk=0, out_sig=0;
  initial
    forever #5 clk <= ~clk ;

  initial
    forever @(posedge clk)
      out_sig <= ~out_sig;
endprogram
```

The program block is not the place to put a clock generator. Example 5-23 tries to put the generator in a program block but just causes a race condition. The **clk** and **out_sig** signals both propagate from the Reactive region to the design in the Active region and could cause a race condition depending on which one arrived first.

Avoid race conditions by always putting the clock generator in a module.

If you want to randomize the generator's properties, create a class with random variables for skew, frequency, and other characteristics. You can use this class in the generator module, or in the testbench.

Example 5-24 Good clock generator in module

```
module clock_generator (output bit clk);
  bit clk = 1;
  always #5 clk = ~clk; // Use blocking assignment
endmodule
```

Lastly, don't try to verify the low-level timing with functional verification. The testbenches described in this book check the behavior of the DUT but not the timing, which is better done with a static timing analysis tool.

5.6   Connecting It All Together

Now you have a design described in a module, a testbench in a program block, and interfaces that connect them together. Here is the top-level module that instantiates and connects all the pieces.

Example 5-25 `Top` module using a simple arbiter interface

```
module top;
  bit  clk;
  always #5 clk = ~clk;

  arb_if arbif(.*);
  arb a1 (.*);
  test t1(.*);
endmodule : top
```

This is almost identical to Example 5-5. It uses a shortcut notation `.*` (implicit port connection) that automatically connects module instance ports to signals at the current level if they have the same name and data type.

5.7   Top-Level Scope

Sometimes you need to create things in your simulation that are outside of a program or module so that they are seen by all parts of the simulation. In Verilog, only macros extend across module boundaries, and are often used for creating global constants. SystemVerilog introduces the *compilation unit,* that is a group of source files that are compiled together. The scope outside the boundaries of any **module**, **macromodule**, **interface**, **program**, **package**, or **primitive** is known as the *compilation-unit scope*, also referred to as **$unit**. Anything such as a **parameter** defined in this scope is similar to a global because it can be seen by all lower-level blocks. But it is

not truly global as the **parameter** cannot be seen during compilation of other files.

This leads to some confusion. Some tools, such as Synopsys VCS, compile all the SystemVerilog code together, so **$unit** is global. But Synopsys Design Compiler compiles a single module or group of modules at a time, so **$unit** may be just the contents of one or a few files. Tools from other vendors may compile all files or just a subset at once. As a result, **$unit** is not portable.

This book calls the scope outside blocks the "top-level scope." You can define variables, parameters, data types and even routines in this space. Example 5-26 declares a top-level parameter, **TIMEOUT**, that can be used anywhere in the hierarchy. This example also has a **const** string that holds an error message. You can declare top-level constants either way.

Example 5-26 Top-level scope for arbiter design

```
// root.sv
`timescale 1ns/1ns
parameter int TIMEOUT = 1_000_000;
const string time_out_msg = "ERROR: Time out";
module top;
  test t1(.*);
endmodule

program automatic test;
  ...
  initial begin
    #TIMEOUT;
    $display("%s", time_out_msg);
    $finish;
    end
endprogram
```

The instance name **$root** allows you to unambiguously refer to names in the system, starting with the top-level scope. In this respect, **$root** is similar to "/" in the Unix file system. For tools such as VCS that compile all files at once, **$root** and **$unit** are equivalent. The name **$root** also solves an old Verilog problem. When your code refers to a name in another module, such as **i1.var**, the compiler first looks in the local scope, then looks up to the next higher scope, and so on until it reaches the top. You may have wanted to use **i1.var** in the top module, but an instance named **i1** in an intermediate scope may have sidetracked the search, giving you the wrong variable. You use **$root** to make unambiguous cross-module references by specifying the absolute path.

Example 5-27 shows a program that is instantiated in a module that is explicitly instantiated in the top-level scope. The program can use a relative or absolute reference to the **clk** signal in the module. Note that if the module were implicitly instantiated, that is, if you took out the line **top t1();**, the absolute reference in the program would change to **$root.top.clk**. Use explicit instantiation of the top module if you plan on making cross-module references.

Example 5-27 Cross-module references with $root

```
// root.sv
`timescale 1ns/1ns
parameter TIMEOUT = 1_000_000;
top t1();          // Explicitly instantiate top-level module

module top;
  bit clk;
  test t1(.*);
endmodule

program automatic test;
  ...
  initial begin
    // Absolute reference
    $display("clk=%b", $root.t1.clk)

    // Relative reference
    $display("clk=%b", t1.clk)
endprogram
```

5.8   Program – Module Interactions

The program block can read and write all signals in modules, and can call routines in modules, but a module has no visibility into a program. This is because your testbench needs to see and control the design, but the design should not depend on anything in the testbench.

A program can call a routine in a module to perform various actions. The routine can set values on internal signals, also known as "backdoor load." Next, because the current System-Verilog standard does not define how to force signals from a program block, you need to write a task in the design to do the force, and then call from the program.

Lastly, it is a good practice for your testbench to use a function to get information from the DUT. Reading signal values can work most of the time, but if the design code changes, your testbench may interpret the values incorrectly. A function in the module can encapsulate the communication between the two and make it easier for your testbench to stay synchronized with the design.

5.9   SystemVerilog Assertions

You can create temporal assertions about signals in your design using SystemVerilog Assertions (SVA). The assertions are instantiated similarly to other design blocks and are active for the entire simulation. The simulator keeps track of what assertions have triggered, so you can gather functional coverage data on them.

5.9.1   Procedural assertions

Your testbench procedural code can check the values of design signals and testbench variables and take action if there is a problem. For example, if you have asserted the bus request, you expect that grant will be asserted two cycles later. You could use an `if`-statement.

Example 5-28 Checking a signal with an `if`-statement

```
bus.cb.request <= 1;
@bus.cb;
if (bus.cb.grant != 2'b01)
  $display("Error, grant != 1");
// rest of the test
```

An assertion is more compact than an `if`-statement. However, note that the logic is reversed compared to the `if`-statement above. You want the expression inside the parentheses to be true; otherwise, print an error.

Example 5-29 Simple procedural assertion

```
bus.cb.request <= 1;
@bus.cb;
a1: assert (bus.cb.grant == 2'b01);
// rest of the test
```

If the **grant** signal is asserted correctly, the test continues. If the signal does not have the expected value, the simulator produces a message similar to the following.

Example 5-30 Error from failed procedural assertion

```
"test.sv", 7: top.t1.a1: started at 55ns failed at 55ns
Offending '(bus.cb.grant == 2'b1)'
```

This message says that on line 7 of the file **test.sv**, the assertion **top.t1.a1** started at 55ns to check the signal **bus.cb.grant**, but it failed immediately.

You may be tempted to use the full SystemVerilog Assertion syntax to check an elaborate sequence over a range of time, but use care. Assertions are declarative code, and execute very differently than the surrounding procedural code. In just a few lines of assertions, you can verify complex code; the equivalent procedural code would be far more complicated and verbose.

5.9.2   Customizing the assertion actions

A procedural assertion has optional then- and else-clauses. If you want to augment the default message, you can add your own.

Example 5-31 Creating a custom error message in a procedural assertion

```
a1: assert (bus.cb.grant == 2'b01)
else $error("Grant not asserted");
```

If **grant** does not have the expected value, you'll see an error message.

Example 5-32 Error from failed procedural assertion

```
 "test.sv", 7: top.t1.a1: started at 55ns failed at 55ns
Offending '(bus.cb.grant == 2'b1)'

"test.sv", 76: top.t1.a1: started at 55ns failed at 55ns
Offending '(arbif.cb.grant == 2'b1)'
Error: "test.sv", 76: top.t1.a1: at time 55 ns
Grant not asserted
```

SystemVerilog has four functions to print messages: **$info**, **$warning**, **$error**, and **$fatal**. These are allowed only inside an assertion, not in procedural code, though future versions of SystemVerilog may allow this.

You can use the then-clause to record when an assertion completed successfully.

Example 5-33 Creating a custom error message

```
a1: assert (bus.cb.grant == 2'b01)
  grants_received++;          // Another succesful result
else
  $error("Grant not asserted");
```

5.9.3   Concurrent assertions

The other type of assertion is the concurrent assertion that you can think of as a small model that runs continuously, checking the values of signals for the entire simulation. You need to specify a sampling clock in the assertion. Here is a small assertion to check that the arbiter signals are do not have X or Z values except during reset.

Example 5-34 Concurrent assertion to check for X/Z

```
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  logic reset;

  property request_2state;
    @(posedge clk) disable iff (reset);
    $isunknown(request) == 0;
  endproperty
  assert_request_2state: assert property request_2state

endinterface
```
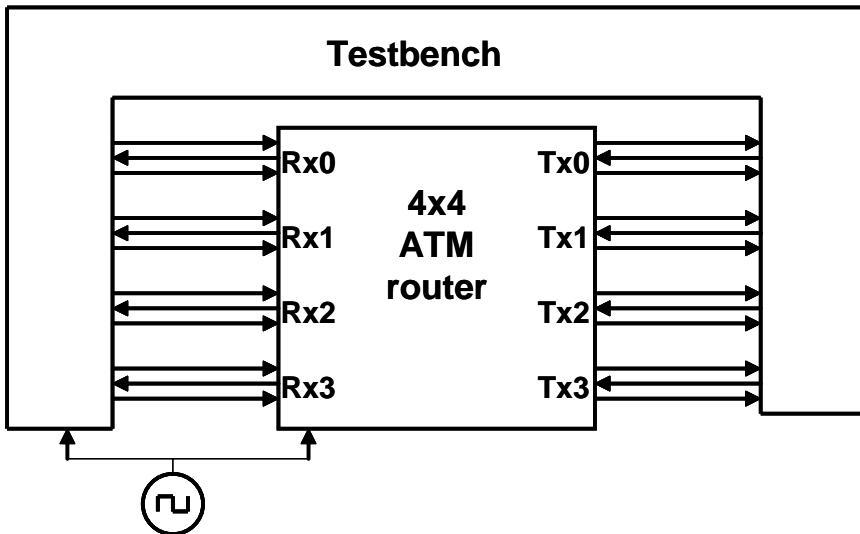
5.9.4   Exploring assertions

There are many other uses for assertions. For example, you can put assertions in an interface. Now your interface not only transmits signal values but also checks the protocol.

This section provides a brief introduction to assertions. For more information on SystemVerilog Assertions, see Vijayaraghhavan (2005).

5.10   The Four-Port ATM Router

The arbiter example is a good introduction to interfaces, but real designs have more than a single input and output. This section shows a four-port ATM router.

Figure 5-8 Testbench – ATM router diagram without interfaces

5.10.1   ATM router with ports

The following code fragments show the tangle of wires you would have to endure to connect an RTL block to a testbench. First is the header for the ATM router model. This uses the Verilog-1995 style port declarations, where the type and direction are separate from the header.

The actual code for the router is crowded out by nearly a page of port declarations.

Example 5-35 ATM router model header without an interface

```
module atm_router(
    // 4 x Level 1 Utopia ATM layer Rx Interfaces
      Rx_clk_0,  Rx_clk_1,  Rx_clk_2,  Rx_clk_3,
      Rx_data_0, Rx_data_1, Rx_data_2, Rx_data_3,
      Rx_soc_0,  Rx_soc_1,  Rx_soc_2,  Rx_soc_3,
      Rx_en_0,   Rx_en_1,   Rx_en_2,   Rx_en_3,
      Rx_clav_0, Rx_clav_1, Rx_clav_2, Rx_clav_3,

    // 4 x Level 1 Utopia ATM layer Tx Interfaces
      Tx_clk_0,  Tx_clk_1,  Tx_clk_2,  Tx_clk_3,
      Tx_data_0, Tx_data_1, Tx_data_2, Tx_data_3,
      Tx_soc_0,  Tx_soc_1,  Tx_soc_2,  Tx_soc_3,
      Tx_en_0,   Tx_en_1,   Tx_en_2,   Tx_en_3,
      Tx_clav_0, Tx_clav_1, Tx_clav_2, Tx_clav_3,

    // Miscellaneous control interfaces
    rst, clk);

// 4 x Level 1 Utopia Rx Interfaces
   output        Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3;
   input [7:0]   Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3;
   input         Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3;
   output        Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3;
   input         Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3;

// 4 x Level 1 Utopia Tx Interfaces
   output        Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3;
   output [7:0]  Tx_data_0,Tx_data_1,Tx_data_2,Tx_data_3;
   output        Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3;
   output        Tx_en_0,  Tx_en_1,  Tx_en_2,  Tx_en_3;
   input         Tx_clav_0,Tx_clav_1,Tx_clav_2,Tx_clav_3;

// Miscellaneous control interfaces
   input         rst, clk;
...
endmodule
```

5.10.2   ATM top-level netlist with ports

Shown next is the top-level netlist.

Example 5-36 Top-level netlist without an interface

```
module top;
  bit clk;
  always #5 clk = !clk;
  wire Rx_clk_0,  Rx_clk_1,  Rx_clk_2,  Rx_clk_3,
       Rx_soc_0,  Rx_soc_1,  Rx_soc_2,  Rx_soc_3,
       Rx_en_0,   Rx_en_1,   Rx_en_2,   Rx_en_3,
       Rx_clav_0, Rx_clav_1, Rx_clav_2, Rx_clav_3,
       Tx_clk_0,  Tx_clk_1,  Tx_clk_2,  Tx_clk_3,
       Tx_soc_0,  Tx_soc_1,  Tx_soc_2,  Tx_soc_3,
       Tx_en_0,   Tx_en_1,   Tx_en_2,   Tx_en_3,
       Tx_clav_0, Tx_clav_1, Tx_clav_2, Tx_clav_3, rst;

  wire [7:0] Rx_data_0, Rx_data_1, Rx_data_2, Rx_data_3,
             Tx_data_0, Tx_data_1, Tx_data_2, Tx_data_3;

  atm_router a1(Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3,
                Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3,
                Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3,
                Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3,
                Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3,
                Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3,
                Tx_data_0,Tx_data_1,Tx_data_2,Tx_data_3,
                Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3,
                Tx_en_0,  Tx_en_1,  Tx_en_2,  Tx_en_3,
                Tx_clav_0,Tx_clav_1,Tx_clav_2,Tx_clav_3,
              rst, clk);

  test       t1 (Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3,
                Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3,
                Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3,
                Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3,
                Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3,
                Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3,
                Tx_data_0,Tx_data_1,Tx_data_2,Tx_data_3,
                Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3,
                Tx_en_0,  Tx_en_1,  Tx_en_2,  Tx_en_3,
                Tx_clav_0,Tx_clav_1,Tx_clav_2,Tx_clav_3,
              rst, clk);
endmodule
```

Example 5-37 shows the top of the testbench module. Once again, note that the ports and wires take up the majority of the netlist.

Example 5-37 Testbench using ports

```
module test(
    // 4 x Level 1 Utopia ATM layer Rx Interfaces
    Rx_clk_0,  Rx_clk_1,  Rx_clk_2,  Rx_clk_3,
    Rx_data_0, Rx_data_1, Rx_data_2, Rx_data_3,
    Rx_soc_0,  Rx_soc_1,  Rx_soc_2,  Rx_soc_3,
    Rx_en_0,   Rx_en_1,  Rx_en_2,    Rx_en_3,
    Rx_clav_0, Rx_clav_1, Rx_clav_2, Rx_clav_3,

    // 4 x Level 1 Utopia ATM layer Tx Interfaces
    Tx_clk_0,  Tx_clk_1,  Tx_clk_2,  Tx_clk_3,
    Tx_data_0, Tx_data_1, Tx_data_2, Tx_data_3,
    Tx_soc_0,  Tx_soc_1,  Tx_soc_2,  Tx_soc_3,
    Tx_en_0,   Tx_en_1,  Tx_en_2,    Tx_en_3,
    Tx_clav_0, Tx_clav_1, Tx_clav_2, Tx_clav_3,

    // Miscellaneous control interfaces
    rst, clk);

// 4 x Level 1 Utopia Rx Interfaces
   input         Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3;
   output [7:0] Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3;
   reg  [7:0]    Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3;
   output        Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3;
   reg           Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3;
   input         Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3;
   output        Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3;
   reg           Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3;

// 4 x Level 1 Utopia Tx Interfaces
   input         Tx_clk_0,  Tx_clk_1, Tx_clk_2, Tx_clk_3;
   input [7:0]  Tx_data_0, Tx_data_1,Tx_data_2,Tx_data_3;
   input         Tx_soc_0,  Tx_soc_1, Tx_soc_2, Tx_soc_3;
   input         Tx_en_0,   Tx_en_1,  Tx_en_2,    Tx_en_3;
   output        Tx_clav_0, Tx_clav_1,Tx_clav_2,Tx_clav_3;
   reg           Tx_clav_0, Tx_clav_1,Tx_clav_2,Tx_clav_3;

// Miscellaneous control interfaces
   output        rst;
   reg           rst;
   input         clk;

   initial begin
     // Reset the device
```
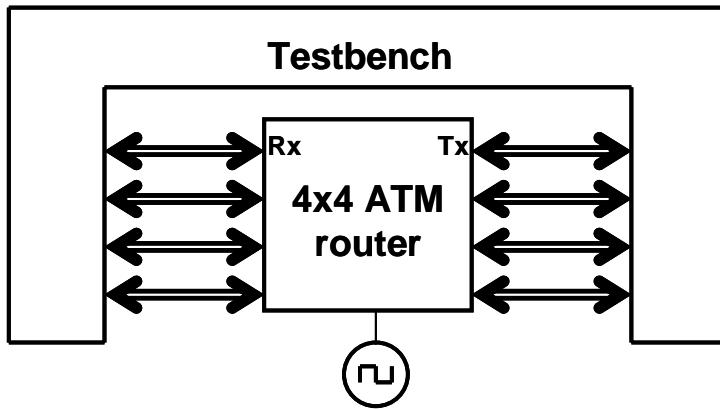
```
    rst <= 1;
    Rx_data_0 <= 0;
    ...
    end
endmodule
```

You just saw three pages of code, and it was all just connectivity — no testbench, no design! Interfaces provide a better way to organize all this information and eliminate the repetitive parts that are so error prone.

5.10.3   Using interfaces to simplify connections

Here is a diagram for the ATM router connected to the testbench, with the signals grouped into interfaces.

Figure 5-9 Testbench - router diagram with interfaces

5.10.4   ATM interfaces

Here are the **Rx** and **Tx** interfaces with modports and clocking blocks.

Example 5-38 Rx interface

```
// Rx interface with modports and clocking block
interface Rx_if (input logic clk);
  logic [7:0] data;
  logic soc, en, clav, rclk;

  clocking cb @(posedge clk);
    output data, soc, clav;  // Directions are relative
    input  en;               // to the testbench
  endclocking : cb

  modport DUT (output en, rclk,
               input  data, soc, clav);

  modport TB (clocking cb);
endinterface : Rx_if
```

Example 5-39 Tx interface

```
// Tx interface with modports and clocking block
interface Tx_if (input logic clk);
  logic [7:0] data;
  logic soc, en, clav, tclk;

  clocking cb @(posedge clk);
      input  data, soc, en;
      output clav;
  endclocking : cb

  modport DUT (output data, soc, en, tclk,
               input  clk, clav);

  modport TB (clocking cb);
endinterface : Tx_if
```

5.10.5   ATM router model using an interface

Here are the ATM router model and testbench, which need to specify the **modport** in their port connection list. Note that you put the **modport** name after the interface name, **Rx_if**.

Example 5-40 ATM router model with interface using modports

```
module atm_router(Rx_if.DUT Rx0, Rx1, Rx2, Rx3,
                  Tx_if.DUT Tx0, Tx1, Tx2, Tx3,
          input logic clk, rst);
...
endmodule
```

5.10.6   ATM top level netlist with interfaces

The top netlist has shrunk considerably, along with the chances of making a mistake.

Example 5-41 Top-level netlist with interface

```
module top;
  bit clk, rst;
  always #5 clk = !clk;

  Rx_if Rx0 (clk), Rx1 (clk), Rx2 (clk), Rx3 (clk);
  Tx_if Tx0 (clk), Tx1 (clk), Tx2 (clk), Tx3 (clk);

  atm_router a1 (Rx0, Rx1, Rx2, Rx3,
                 Tx0, Tx1, Tx2, Tx3, clk, rst);

  test       t1 (Rx0, Rx1, Rx2, Rx3,
                 Tx0, Tx1, Tx2, Tx3, clk, rst);
endmodule : top
```

5.10.7   ATM testbench with interface

Example 5-42 shows the part of the testbench that captures cells coming in from the TX port of the router. Note that the interface names are hard-coded, so you have to duplicate the same code four times for the 4x4 ATM switch. Chapter 10 shows how to simplify the code by using virtual interfaces.

Example 5-42 Testbench using interface with clocking block

```
program test(Rx_if.TB Rx0, Rx1, Rx2, Rx3,
             Tx_if.TB Tx0, Tx1, Tx2, Tx3,
             input logic clk, output logic rst);

  bit [7:0] bytes[ATM_CELL_SIZE];

  initial begin
    // Reset the device
    rst <= 1;
    Rx0.cb.data <= 0;
    ...
    receive_cell0();
    ...
    end

  task receive_cell0();
    @(Tx0.cb);
    Tx0.cb.clav <= 1;          // Assert ready to receive
    wait (Tx0.cb.soc == 1);    // Wait for Start of Cell

    for (int i=0; i<ATM_CELL_SIZE; i++) begin
      wait (Tx0.cb.en == 0);   // Wait for enable
        @(Tx0.cb);

      bytes[i] = Tx0.cb.data;
      @(Tx0.cb);
      Tx0.cb.clav <= 0;        // Deassert flow control
    end
  endtask : receive_cell0

endprogram : test
```

5.11   Conclusion

In this chapter you have learned how to use SystemVerilog's interfaces to organize the communication between design blocks and your testbench. With this design construct, you can replace dozens of signal connections with a single interface, making your code easier to maintain and improve, and reducing the number of wiring mistakes.

SystemVerilog also introduces the program block to hold your testbench and to reduce race conditions between the device under test and the testbench. With a clocking block in an interface, your testbenches will drive and sample design signals correctly relative to the clock.

Chapter 6

Randomization

6.1   Introduction

As designs grow larger, it becomes more difficult to create a complete set of stimuli needed to check their functionality. You can write a directed testcase to check a certain set of features, but you cannot write enough directed testcases when the number of features keeps doubling on each project. Worse yet, the interactions between all these features are the source for the most devious bugs and are the least likely to be caught by going through a laundry list of features.

The solution is to create testcases automatically using constrained-random tests (CRT). A directed test finds the bugs you think are there, but a CRT finds bugs you never thought about, by using random stimulus. You restrict the test scenarios to those that are both valid and of interest by using constraints.

Creating a CRT environment takes more work than creating one for directed tests. A simple directed test just applies stimulus, and then you manually check the result. These results are captured as a golden log file and compared with future simulations to see whether the test passes or fails. A CRT environment needs not only to create the stimulus but also to predict the result, using a reference model, transfer function, or other techniques. However, once this environment is in place, you can run hundreds of tests without having to hand-check the results, thereby improving your productivity. This trade-off of test-authoring time (your work) for CPU time (machine work) is what makes CRT so valuable.

A CRT is made of two parts: the test code that uses a stream of random values to create input to the DUT, and a seed to the pseudo-random number generator (PRNG), shown in section 6.15.1. You can make a CRT behave differently just by using a new seed. This feature allows you to leverage each test so each is the functional equivalent of many directed tests, just by changing seeds. You are able to create more equivalent tests using these techniques than with directed testing.

You may feel that these random tests are like throwing darts. How do you know when you have covered all aspects of the design? The stimulus space is too large to generate every possible input by using **for**-loops, so you need to generate a useful subset. In Chapter 9 you will learn how to measure verification progress by using functional coverage.

There are many ways to use randomization, and this chapter gives a wide range of examples. It highlights the most useful techniques, but you should choose what works best for you.

6.2   What to Randomize

When you think of randomizing the stimulus to a design, the first thing you may think of are the data fields. These are the easiest to create – just call `$random`. The problem is that this approach has a very low payback in terms of bugs found: you only find data-path bugs, perhaps with bit-level mistakes. The test is still inherently directed. The challenging bugs are in the control logic. As a result, you need to randomize all decision points in your DUT. Everywhere control paths diverge, randomization increases the probability that you'll take a different path in each test case.

You need to think broadly about all design input such as the following.

- Device configuration
- Environment configuration
- Primary input data
- Encapsulated input data
- Protocol exceptions
- Delays
- Transaction status
- Errors and violations

6.2.1   Device configuration

What is the most common reason why bugs are missed during testing of the RTL design? Not enough different configurations have been tried! Most tests just use the design as it comes out of reset, or apply a fixed set of initial-ization vectors to put it into a known state. This is like testing a PC's operating system right after it has been installed, and without any applica-tions; of course the performance is fine, and there are no crashes.

Over time, in a real world environment, the DUT's configuration becomes more and more random. For example, a Synopsys customer had to verify a time-division multiplexor switch that had 600 input channels and 12 output channels. When the device was installed in the end-customer's system, chan-nels would be allocated and deallocated over and over, so at any point in time,

there would be little correlation between adjacent channels. In other words, the configuration would seem to be random.

To test this device, the verification engineer had to write several dozen lines of Tcl code to configure each channel. As a result, she was never able to try configurations with more than a handful of channels enabled. Using a CRT methodology, she wrote a testbench that randomized the parameters for a single channel, and then put this in a loop to configure the whole device. Now she had confidence that her tests would uncover bugs that previously would have been missed.

6.2.2   Environment configuration

The device that you are designing operates in an environment containing other devices. When you are verifying the DUT, it is connected to a testbench that mimics this environment. You should randomize the entire environment, including the number of objects and how they are configured.

Another company was creating an I/O switch chip that connected multiple PCI buses to an internal memory bus. At the start of simulation the customer used randomization to choose the number of PCI buses (1–4), the number of devices on each bus (1–8), and the parameters for each device (master or slave, CSR addresses, etc.). Even though there were many possible combinations, this company knew all had been covered.

6.2.3   Primary input data

This is what you probably thought of first when you read about random stimulus: take a transaction such as a bus write or ATM cell and fill it with some random values. How hard can that be? Actually it is fairly straightforward as long as you carefully prepare your transaction classes. You should anticipate any layered protocols and error injection.

6.2.4   Encapsulated input data

Many devices process multiple layers of stimulus. For example, a device may create TCP traffic that is then encoded in the IP protocol, and finally sent out inside Ethernet packets. Each level has its own control fields that can be randomized to try new combinations. So you are randomizing the data and the layers that surround it. You need to write constraints that create valid control fields but that also allow injecting errors.

6.2.5   Protocol exceptions, errors, and violations

Anything that can go wrong, will, eventually. The most challenging part of design and verification is how to handle errors in the system. You need to anticipate all the cases where things can go wrong, inject them into the system, and make sure the design handles them gracefully, without locking up or going into an illegal state. A good verification engineer tests the behavior of the design to the edge of the functional specification and sometimes even beyond

When two devices communicate, what happens if the transfer stops partway through? Can your testbench simulate these breaks? If there are error detection and correction fields, you must make sure all combinations are tried.

The random component of these errors is that your testbench should be able to send functionally correct stimuli and then, with the flip of a configuration bit, start injecting random types of errors at random intervals.

6.2.6   Delays

Many communication protocols specify ranges of delays. The bus grant comes one to three cycles after request. Data from the memory is valid in the fourth to tenth bus cycle. However, many directed tests, optimized for the fastest simulation, use the shortest latency, except for that one test that only tries various delays. Your testbench should always use random, legal delays during every test to try to find that (hopefully) one combination that exposes a design bug.

Below the cycle level, some designs are sensitive to clock jitter. By sliding the clock edges back and forth by small amounts, you can make sure your design is not overly sensitive to small changes in the clock cycle.

The clock generator should be in a module outside the testbench so that it creates events in the Active region along with other design events. However, the generator should have parameters such as frequency and offset that can be set by the testbench during the configuration phase.

(Note that you are looking for functional errors, not timing errors. Your testbench should not try to violate setup and hold requirements. These are better validated using timing analysis tools.)

6.3   Randomization in SystemVerilog

The random stimulus generation in SystemVerilog is most useful when used with OOP. You first create a class to hold a group of related random variables, and then have the random-solver fill them with random values. You

can create constraints to limit the random values to legal values, or to test specific features.

Note that you can randomize individual variables, but this case is the least interesting. True constrained-random stimuli is created at the transaction level, not one value at a time.

6.3.1 Simple class with random variables

Example 6-1 shows a class with random variables, constraints, plus testbench code to use this class.

Example 6-1 Simple random class

```
class Packet;
  // The random variables
  rand bit [31:0] src, dst, data[8];
  randc bit [7:0] kind;
  // Limit the values for src
  constraint c {src > 10;
                src < 15;}
endclass

Packet p;
initial begin
  p = new;  // Create a packet
  assert (p.randomize());
  transmit(p);
end
```

This class has four random variables. The first three use the **rand** modifier, so that every time you randomize the class, the variables are assigned a value. Think of rolling dice: each roll could be a new value or repeat the current one. The **kind** variable is **randc**, which means random cyclic, so that the random solver does not repeat a random value until every possible value has been assigned. Think of dealing cards from a deck: you deal out every card in the deck in random order, then shuffle the deck, and deal out the cards in a different order.

Note that the constraint expression is grouped using curly braces: {}. This is because this code is declarative, not procedural, which uses **begin**...**end**.

The **randomize** function returns 0 if a problem is found with the constraints. The procedural assertion is used to check the result, as shown in section 5.9. You need find the tool-specific switches to cause the assertion to terminate simulation. This book uses **assert** to test the result from **randomize**, but you may want to test the result and then call your special

routine that prints any useful information and then gracefully shuts down then simulation.

The constraint in Example 6-1 is an expression that limits the values for the **src** variable. In this case, SystemVerilog chooses between the values of 11, 12, 13, or 14.

All variables in your classes should be random and public. This gives your test the maximum control over the DUT's stimulus and control. You can always turn off a random variable, as show in section 6.10.2. If you forget to make a variable random, you must edit the environment, which you want to avoid.

6.3.2   Checking the result from randomize

randomize assigns random values to any variable in the class that has been labeled as **rand** or **randc**, and also makes sure that all active constraints are obeyed. Randomization can fail if your code has conflicting constraints (see next section), so you should always check the status. If you don't check, the variables may get unexpected values, causing your simulation to fail.

Example 6-1 checks the status from **randomize** by using a procedural assertion. If randomization succeeds, the function returns 1. If it fails, **randomize** returns 0. The assertion checks the result and prints an error if there was a failure. You should set your simulator's switches to terminate when an error is found. Alternatively, you might want to call a special routine to end simulation, after doing some housekeeping chores like printing a summary report.

6.3.3   The constraint solver

The process of solving constraint expressions is handled by the System-Verilog constraint solver. The solver chooses values that satisfy the constraints. The values come from SystemVerilog's PRNG, that is started with an initial seed. If you give a SystemVerilog simulator the same seed and the same testbench, it always produces the same results.

The solver is specific to the simulation vendor, and a constrained-random test may not give the same results when run on different simulators, or even on different versions of the same tool. The SystemVerilog standard specifies the meaning of the expressions, and the legal values that are created, but does not detail the precise order in which the solver should operate. See section 6.15 for more details on random number generators.

6.4   Constraint Details

Useful stimulus is more than just random values — there are relationships between the variables. Otherwise, it may take too long to generate interesting stimulus values, or the stimulus might contain illegal values. You define these interactions in SystemVerilog using constraint blocks that contain one or more constraint expressions. SystemVerilog solves these expressions concurrently, choosing random values that satisfy all the expressions.

At least one variable in each expression should be random, either **rand** or **randc**. The following class fails when randomized. The solution is to add the modifier **rand** or **randc** to the variable **son**.

Example 6-2 Constraint without random variables

```
class bad;
  bit [31:0] son;  // Error - should be rand or randc
  constraint c_teenager {son > 12;
                         son < 20;}
endclass
```

The **randomize** function tries to assign new values to random variables and to make sure all constraints are satisfied. In Example 6-2, since there are no random variables, **randomize** just checks the value of **son** to see if it is in the bounds specified by the constraint **c_teenager**. Unless the variable happens to fall in the range of 13:19, **randomize** fails.

6.4.1   Constraint introduction

The following sections use this example of a random class with constraints. The specific constructs are explained later in this section.

Example 6-3 Constrained-random class

```
class Stim;
  const bit [31:0] SRC_CONGEST_ADDR = 42;
  typedef enum {READ, WRITE, CONTROL} stim_t;
  randc stim_t type;    // Enumerated var
  rand bit [31:0] len, src, dst;
  bit congestion_test;

  constraint c_stim {
    len < 1000;
    len > 0;
    src inside {0, [2:10], [100:107]};
    if (congestion_test) {
      dst inside {[CONGEST_ADDR-100:CONGEST_ADDR+100]};
    }
  }
endclass
```

6.4.2   Simple expressions

Example 6-3 shows a class with a constraint block, with several expressions. The first two control the values for the **len** variable. As shown above, a variable can be used in multiple expressions.

There can be a maximum of only one relational operator (**<**, **<=**, **==**, **>=**, or **>**) in an expression. If you want to put multiple variables in a fixed order, such as **a**, **b**, and **c**, use multiple expressions.

Example 6-4 Constrain variables to be in a fixed order

```
class bad;
  rand bit [15:0] a, b, c;
  constraint good {0 < a;           // Correct way
                   a < b;
                   b < c;}
  constraint bad  {0 < a < b < c;} // Error, won't work!
endclass
```

6.4.3   Equivalence expressions

You cannot make assignments in a constraint block as it only contains expressions. Instead, use the equivalence operator to set a random variable to a value, e.g., **len==42**, or to build more complex relationships between one

or more random variables, e.g., **len == header.addr_mode * 4 + payload.size**.

6.4.4   Set membership and the inside operator

You can create sets of values with the **inside** operator. SystemVerilog gathers all the values and chooses between the values with equal probability, unless you have other constraints on the variable. As always, you can use variables in the sets. In Example 6-3, the first two expressions could be replaced with **len inside {[1:999]}**.

Example 6-5 Random sets of values

```
rand int c;        // Random variable
int lo, hi;        // Nonrandom variables used as limits
constraint c_range {
  c inside {[lo:hi]};  // lo <= c and c <= hi
}
```

In Example 6-5, SystemVerilog uses the values for **lo** and **hi** to determine the range of possible values. You can use this to parameterize your constraints so that the testbench can alter the behavior of the stimulus generator without rewriting the constraints. A set can contain multiple values and ranges as shown in Example 6-3. Note that if **lo** > **hi**, an empty set is formed, and the constraint fails.

If you want any value, as long as it is not inside a set, invert the constraint.

Example 6-6 Inverted random set constraint

```
constraint c_range {
  !(c inside {[lo:hi]});  // c < lo or c > hi
}
```

All values in the set are chosen equally, even if they appear multiple times.

Example 6-7 Inverted random set constraint

```
constraint c_even_weight {
  (c inside {0,1,1,1,1,1});  // 0 or 1, equal probability
}
```

If you need to weight some values more than others, use the **dist** operator, shown in section 6.4.6.

6.4.5   Using an array in a set

You can choose from a set of values by storing them in an array. Example 6-8 chooses a day of the week from a list of enumerated values. You can change the list of choices on the fly. If you make **choice** a **randc** variable, the simulator tries every possible value before repeating.

The **name** function returns a string with the name of an enumerated value.

Example 6-8 Choosing from an array of possible values

```
class Days;
  typedef enum {SUN, MON, TUE, WED,
                THU, FRI, SAT} DAYS_E;
  DAYS_E choices[$];
  rand DAYS_E choice;
  constraint cday {choice inside choices;}
endclass

Days days;

initial begin
  days = new;

  days.choices = '{Days::SUN, Days::SAT};
  assert (days.randomize());
  $display("Random weekend day %s\n", days.choice.name);

  days.choices = '{Days::MON, Days::TUE, Days::WED,
                   Days::THU, Days::FRI};
  assert (days.randomize());
  $display("Random week day %s", days.choice.name);
end
```

If you want to dynamically add or remove values from a set, think twice before using the **inside** operator because of its performance. For example, perhaps you have a set of values that you want to be chosen just once. You could use **inside** to choose values from a queue, and delete them to slowly shrink the queue. This requires the solver to solve N constraints, where N is the number of elements left in the queue. Instead, use a **randc** variable that points into an array of choices. Choosing a **randc** value takes a short, constant time, while solving a large number of constraints is very expensive, especially for more than a few dozen values.

Example 6-9 Using `randc` to chose array values in random order

```
class RandcInside;
  int array[];                  // Values to choose
  randc bit [15:0] index;       // Index into array

  function new(input int a[]); // Construct & initialize
    array = a;
  endfunction

  function int pick;            // Return most recent pick
    return array[index];
  endfunction

  constraint c_size {index < array.size;}
endclass

  initial begin
    RandcInside ri;

    ri = new('{1,3,5,7,9,11,13});
    repeat (ri.array.size) begin
      assert(ri.randomize());
      $display("Picked %2d [%0d]", ri.pick(),  ri.index);
    end
  end
```

Note that constraints and routines can be mixed in any order.

6.4.6   Weighted Distributions

The **dist** operator allows you to create weighted distributions so that some values are chosen more often than others. A **dist** operator takes a list of values and weights, separated by the **:=** or the **:/** operator. The values and weights can be constants or variables. The values can be a single value or a range such as **[lo:hi]**. The weights are not percentages and do not have to add up to 100. The **:=** operator specifies that the weight is the same for every specified value in the range, while the **:/** operator specifies that the weight is to be equally divided between all the values.

Example 6-10 Weighted random distribution with `dist`

```
rand int src, dst;
constraint c_dist {
   src dist {0:=40, [1:3]:=60};
   // src = 0, weight = 40/220
   // src = 1, weight = 60/220
   // src = 2, weight = 60/220
   // src = 3, weight = 60/220

   dst dist {0:/40, [1:3]:/60};
   // dst = 0, weight = 40/100
   // dst = 1, weight = 20/100
   // dst = 2, weight = 20/100
   // dst = 3, weight = 20/100
}
```

In Example 6-10, **src** gets the value 0, 1, 2, or 3. The weight of 0 is 40, while 1, 2, and 3 have weights of 60, for a total of 220. The probability of choosing 0 is 40/220, and the probability of choosing 1, 2, or 3 is 60/220 each.

Next, **dst** gets the value 0, 1, 2, or 3. The weight of 0 is 40, while 1, 2, and 3 share a total weight of 60, for a total of 100. The probability of choosing 0 is 40/100, while the probability of choosing 1, 2, or 3 is only 20/100 each.

Once again, the values and weights can be constants or variables. You can use variable weights to change distributions on the fly or even to eliminate choices by setting the weight to zero.

Example 6-11 Dynamically changing distribution weights

```
// Bus operation, byte, word, or longword
class BusOp;
  // Operand length
  typedef enum {BYTE, WORD, LWRD } length_t;
  rand length_t len;

  // Random weights for dist constraint
  bit [31:0] w_byte=1, w_word=3, w_lwrd=5;

  constraint c_len {
    len dist {BYTE := w_byte,     // Choose a random
              WORD := w_word,     // length using
              LWRD := w_lwrd};    // variable weights
  }
endclass
```

In Example 6-11, the **len** enumerated variable has three values. The constraint defaults to choosing longword lengths, as **w_lwrd** has the largest value.

6.4.7 Bidirectional Constraints

By now you may have realized that constraint blocks are not procedural code, executing from top to bottom. They are declarative code, all active at the same time. If you constrain a variable with the **inside** operator with the set **[10:50]** and have another expression that constrains the variable to be greater than 20, SystemVerilog only chooses values between 21 and 50.

SystemVerilog constraints are bidirectional, which means that the solver looks at the constraints on both side of an expression. Consider the following constraint:

Example 6-12 Bidirectional constraint

```
rand logic [15:0] b, c, d;
constraint c_bidir {
  b < d;
  c == b;
  d < 30;
  c > 25;
}
```

The SystemVerilog solver looks at all four constraints simultaneously. **b** has to be less than **d**, which has to be less than 30. But **b** is constrained to be equal to **c**, which is greater than 25. Even though there is no direct constraint on the lower value of **d**, the constraint on **c** restricts the choices.

Table 6-1. Solutions for bidirectional constraints

Solution	*b*	*c*	*d*
1	26	26	27
2	27	27	28
3	28	28	29

6.4.8 Conditional constraints

Normally, all constraint expressions are active in a block. What if you want to have an expression active only some of the time? For example, a bus

supports byte, word, and longword reads, but only longword writes. System-
Verilog supports two implication operators, **->** and **if-else**.

When you are choosing from a list of expressions, such as an enumerated
type, the implication operator, **->**, lets you create a case-like block. The
parentheses around the expression are not required, but do make the code eas-
ier to read.

Example 6-13 Constraint block with implication operator

```
class BusOp;
  ...
  constraint c_io {
    (io_space_mode) ->
      addr[31] == 1'b1;
  }
```

If you have a true-false expression, the **if-else** operator may be better.

Example 6-14 Constraint block with if-else operator

```
class BusOp;
  ...
  constraint c_len_rw {
    if (op == READ)
      len inside {[BYTE:LWRD]};
    else
      len == LWRD;
  }
```

In constraint blocks, you use curly braces, **{ }**, to group multiple expres-
sions. The **begin**...**end** keywords are for procedural code.

6.4.9   Choose the right arithmetic operator to boost efficiency

Simple arithmetic operators such as addition and subtraction, bit extracts,
and shifts are handled very efficiently by the solver in a constraint. However,
multiplication, division, and modulo are very expensive with 32-bit values.
Remember that any constant without an explicit size, such as **42**, is treated as
a 32-bit value.

If you want to generate random addresses that are near a page boundary,
where a page is 4096 bytes, you could write the following code, but the solver
may take a long time to find suitable values for **addr**.

Example 6-15 Expensive constraint with mod and unsized variable

```
rand bit [31:0] addr;
constraint slow_near_page_boundary {
  addr % 4096 inside {[0:20], [4075:4095]};
}
```

Many constants in hardware are powers of 2, so take advantage of this by using bit extraction rather than division and modulo. Likewise, multiplication by a power of two can be replaced by a shift.

Example 6-16 Efficient constraint with bit extract

```
rand bit [31:0] addr;
constraint near_page_boundry {
  addr[11:0] inside {[0:20], [4075:4095]};
}
```

6.5   Solution Probabilities

Whenever you deal with random values, you need to understand the probability of the outcome. SystemVerilog does not guarantee the exact solution found by the random constraint solver, but you can influence the distribution. Any time you work with random numbers, you have to look at thousands or millions of values to average out the noise. Changing the tool version or random seed can cause different results. Some simulators, such as Synopsys VCS, have multiple solvers to allow you to trade memory usage vs. performance.

6.5.1   Unconstrained

Start with two variables with no constraints.

Example 6-17 Class `Unconstrained`

```
class Unconstrained;
  rand bit x;         // 0 or 1
  rand bit [1:0] y;   // 0, 1, 2, or 3
endclass
```

There are eight possible solutions. Because there are no constraints, each has the same probability. You have to run thousands of randomizations to see the actual results approach the listed probabilities.[9]

Table 6-2. Solutions for **Unconstrained** class

Solution	x	y	Probability
A	0	0	1/8
B	0	1	1/8
C	0	2	1/8
D	0	3	1/8
E	1	0	1/8
F	1	1	1/8
G	1	2	1/8
H	1	3	1/8

6.5.2   Implication

In Example 6-18, the value of **y** depends on the value of **x**. This is indicated with the implication operator in the following constraint. This example and the rest in this section also behave in the way same with the **if** implication operator.

Example 6-18 Class with implication

```
class Imp1;
  rand bit x;            // 0 or 1
  rand bit [1:0] y;      // 0, 1, 2, or 3
  constraint c_xy {
    (x==0) -> y==0;
  }
endclass
```

Here are the possible solutions and probability. You can see that the random solver recognizes that there are eight combinations of **x** and **y**, but all the solutions where **x==0** (A–D) have been merged together.

---

9. The tables were generated with Synopsys VCS 2005.06 using the run-time switch
   `+ntb_solver_mode=1`.

Table 6-3. Solutions for **Imp1** class

Solution	x	y	Probability
A	0	0	1/2
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/8
G	1	2	1/8
H	1	3	1/8

6.5.3   Implication and bidirectional constraints

Note that the implication operator says that when **x==0**, **y** is forced to 0, but when **y==0**, there is no constraint on **x**. However, implication is bidirectional in that if **y** were forced to a nonzero value, **x** would have to be 1. Example 6-19 has the constraint **y>0**, so **x** can never be 0.

Example 6-19 Class with implication and constraint

```
class Imp2;
  rand bit x;          // 0 or 1
  rand bit [1:0] y;    // 0, 1, 2, or 3
  constraint c_xy {
     y > 0;
    (x==0) -> y==0;
  }
endclass
```

Table 6-4.   Solutions for **Imp2** class

Solution	x	y	Probability
A	0	0	0
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	0
F	1	1	1/3
G	1	2	1/3
H	1	3	1/3

6.5.4   Guiding distribution with **solve...before**

You can guide the SystemVerilog solver using the "**solve**...**before**" constraint as seen in Example 6-20.

Example 6-20 Class with implication and solve...before

```
class SolveBefore;
  rand bit x;          // 0 or 1
  rand bit [1:0] y;    // 0, 1, 2, or 3
  constraint c_xy {
    (x==0) -> y==0;
    solve x before y;
  }
endclass
```

The **solve**...**before** constraint does not change the solution space, just the probability of the results. The solver chooses values of **x** (0, 1) with equal probability. In 1000 calls to **randomize**, **x** is 0 about 500 times, and 1 about 500 times. When **x** is 0, **y** must be 0. When **x** is 1, **y** can be 0, 1, 2, or 3 with equal probability.

Table 6-5.   Solutions for **solve x before y** constraint

Solution	x	y	Probability
A	0	0	1/2
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/8
G	1	2	1/8
H	1	3	1/8

But, if you use the constraint **solve y before x**, you get a very different distribution.

Table 6-6.   Solutions for **solve y before x** constraint

Solution	x	y	Probability
A	0	0	1/8
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/4
G	1	2	1/4
H	1	3	1/4

Only use **solve**...**before** if you are dissatisfied with how often some values occur. Excessive use can slow the constraint solver and make your constraints difficult for others to understand.

6.6   **Controlling Multiple Constraint Blocks**

A class can contain multiple constraint blocks. Your class might naturally divide into two sets of variables, such as data vs. control, so you may want to constrain them separately. Or you might want to have a separate constraint for each test. Perhaps one constraint would restrict the data length to create small transactions (great for testing congestion), while another would make long transactions.

At run-time, you can use the `constraint_mode()` routine to turn constraints on and off. When used with `handle.constraint`, this method controls a single constraint. When used with just `handle`, it controls all constraints for an object.

Example 6-21 Using `constraint_mode`

```
class Packet;
  rand int length;
  constraint c_short {length inside {[1:32]}; }
  constraint c_long  {length inside {[1000:1023]}; }
endclass

Packet p;
initial begin
  p = new;

  // Create a long packet by disabling short constraint
  p.c_short.constraint_mode(0);
  assert (p.randomize());

  transmit(p);

  // Create a short packet by disabling all constraints
  // then enabling only the short constraint
  p.constraint_mode(0);
  p.c_short.constraint_mode(1);
  assert (p.randomize());
  transmit(p);
end
```

6.7   **Valid Constraints**

A good randomization technique is to create several constraints to ensure the correctness of your random stimulus, known as "valid constraints." For

example, a bus read-modify-write command might only be allowed for a longword data length.

Example 6-22 Checking write length with a valid constraint

```
class BusTrans;
  rand enum {BYTE, WORD, LWRD, QWRD} length;
  rand enum {READ, WRITE, RMW, INTR} opc;

  constraint valid_RMW_LWRD {
    (opc == RMW) -> length == LWRD;
  }
endclass
```

Now you know the bus transaction obeys the rule. Later, if you want to violate the rule, use **constraint_mode** to turn off this one constraint. You should have a naming convention to make these constraints stand out, such as using the prefix **valid** as shown above.

6.8   In-line Constraints

As you write more tests, you can end up with many constraints. They can interact with each other in unexpected ways, and the extra code to enable and disable them adds to the test complexity. Additionally, constantly adding and editing constraints to a class could cause problems in a team environment.

Many tests only randomize objects at one place in the code. SystemVerilog allows you to add an extra constraint using **randomize() with**. This is equivalent to adding an extra constraint to any existing ones in effect. Example 6-23 shows a base class with constraints, then two **randomize() with** statements.

Example 6-23 The `randomize()` with statement

```
class Transaction;
  rand bit [31:0] addr, data;
  constraint c1 {addr inside{[0:100],[1000:2000]};}
endclass

Transaction t = new();

initial begin
  int s;
  t = new();

  // addr is 50-100, 1000-1500, data < 10
  assert(t.randomize() with {addr >= 50; addr <= 1500;
                             data < 10;});

  driveBus(t);

  // force addr to a specific value, data > 10
  assert(t.randomize() with {addr == 2000; data > 10;});

  driveBus(t);
  end
```

The extra constraints are added to the existing ones in effect. Use **constraint_mode** if you need to disable a conflicting constraint. Note that inside the **with{}** statement, SystemVerilog uses the scope of the class. That is why Example 6-23 used just **addr**, not **t.addr**.

A common mistake is to surround your in-line constraints with parenthesis instead of curly braces {}. Just remember that constraint blocks use curly braces, so your in-line constraint must use them too. Braces are for declarative code.

6.9   The **pre_randomize** and **post_randomize** Functions

Sometimes you need to perform an action immediately before every **randomize** call or immediately afterwards. For example, you may want to set some nonrandom class variables (such as limits) before randomization starts, or you may need to calculate the error correction bits for random data.
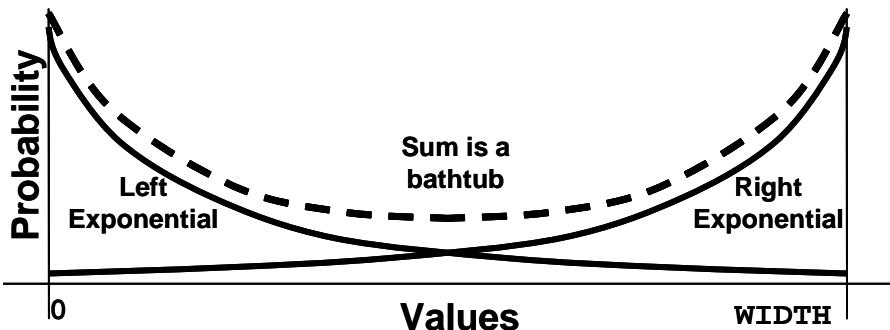
SystemVerilog lets you do this with two special void functions, **pre_randomize** and **post_randomize**. Section 3.3 showed that a void

function does not return a value, but, because it is not a task, does not consume time. If you want to call a debug routine from p**re_randomize** or **post_randomize**, it must be a function.

6.9.1   Building a bathtub distribution

For some applications, you want a nonlinear random distribution. For instance, small and large packets are more likely to find a design bug such as buffer overflow than medium-sized packets. So you want a bathtub shaped distribution; high on both ends, and low in the middle. You could build an elaborate **dist** constraint, but it might require lots of tweaking to get the shape you want. Verilog has several functions for nonlinear distribution such as **$dist_exponential** but none for a bathtub. However, you can build one by using the exponential function twice.

Figure 6-1 Building a bathtub distribution

Example 6-24 Building a bathtub distribution

```
class Bathtub;
  int value;  // Random variable with bathtub dist
  int WIDTH = 50, DEPTH=4, seed=1;

  function void pre_randomize();
    // Calculate the left curve of the tub
    value = $dist_exponential(seed, DEPTH);
    if (value > WIDTH) value = WIDTH;

    // Randomly put this point on the left or right curve
    if ($urandom_range(1))
      value = WIDTH - value;
  endfunction

endclass
```

Every time this object is randomized, the variable **value** gets updated. Across many randomizations, you will see the desired nonlinear distribution.

You can use all the Verilog-1995 distribution functions this way, plus several that are new for SystemVerilog. Some of the useful functions include the following.

- **$dist_exponential** — Exponential decay, as shown in Figure 6-1
- **$dist_normal** — Bell-shaped distribution
- **$dist_poisson** — Bell-shaped distribution
- **$dist_uniform** — Flat distribution
- **$random** — Flat distribution, returning signed 32-bit random
- **$urandom** — Flat distribution, returning unsigned 32-bit random
- **$urandom_range** — Flat distribution over a range

Consult a statistics book for more details on these functions.

6.9.2   Note on void functions

The functions **pre_randomize** and **post_randomize** can only call other functions, not tasks that could consume time. After all, you cannot have a delay in the middle of a call to **randomize**. When you are debugging a randomization problem, you can call your display routines if you planned ahead and made them void functions.

6.10   Constraints Tips and Techniques

How can you create constrained-random tests that can be easily modified? There are several tricks you can use.

6.10.1   Constraints with variables

Most constraint examples in this book use constants to make them more readable. In Example 6-25, **size** is randomized over a range that uses a variable for the upper bound.

Example 6-25 Constraint with a variable bound

```
class bounds;
  rand int size;
  int max_size = 100;
  constraint c_size {
   size inside {[1:max_size]};
  }
endclass
```

By default, this class creates random sizes between 1 and 100, but by changing the variable **max_size**, you can vary the upper limit.

You can use variables in the **dist** constraint to turn on and off values and ranges. In Example 6-26, each bus command has a different weight variable.

Example 6-26 dist constraint with variable weights

```
typedef enum (READ8, READ16, READ32) read_t;
class ReadCommands;
  rand read_t read_cmd;
  int read8_wt=1, read16_wt=1, read32_wt=1;
  constraint c_read {
    read_cmd dist {READ8  := read8_wt,
                   READ16 := read16_wt,
                   READ32 := read32_wt};
    }
endclass
```

By default, this constraint produces each command with equal probability. If you want to have a greater number of **READ8** commands, increase the **read8_wt** weight variable. Most importantly, you can turn off generation of some commands by dropping their weight to 0.

6.10.2   Using nonrandom values

If you have a set of constraints that produces stimulus that is almost what you want, but not quite, you could call **randomize**, and then set a variable to the value you want – you don't have to use the random one. However, your stimulus values may not be correct according to the constraints you created to check validity.

If there are just a few variables that you want to override, use **rand_mode** to make them nonrandom.

Example 6-27 `rand_mode` disables randomization of variables

```
// Packet with variable length payload
class Packet;
  rand bit [7:0] length;
  rand bit [7:0] payload[];
  constraint c_valid {length > 0;
                      payload.size == length;}

  function void display(string msg);
    $write("Packet len=%0d, payload size=%0d, bytes = ",
           length, payload.size);
    for(int i=0; (i<4 && i<payload.size); i++)
      $write(" %0d", payload[i]);
    $display;
  endfunction
endclass

Packet p;
initial begin
  p = new();

  // Randomize all variables
  assert (p.randomize());
  p.display("Simple randomize");

  // Make length nonrandom then randomize packet
  p.length.rand_mode(0);
  p.length = 42;
  assert (p.randomize());
  p.display("Randomize with rand_mode");
end
```

In Example 6-27, the packet length is stored in one variable, while **payload** is a dynamic array. The first half of the test randomizes both the **length** variable and the contents of the **payload** array. The second half calls **rand_mode** to make **length** a nonrandom variable, sets it to 42, then calls **randomize**. The constraint sets the **payload** size at the constant 42, but the array is still filled with random values.

6.10.3   Checking values using constraints

If you randomize an object and then modify some variables, you can check that the object is still valid by checking if all constraints are still obeyed. Call

handle.randomize(null) and SystemVerilog treats all variables as non-random ("state variables") and just ensures that all constraints are satisfied.

6.10.4   Turn constraints off and on

A simple testbench may use a data class with just a few constraints. What if you want to have two tests with very different flavors of data? You could use the implication operators (**->** or **if-else**) to build a single, elaborate constraint controlled by nonrandom variables.

Example 6-28 Using the implication constraint as a `case` statement

```
class Instruction;
  typedef enum {NOP, HALT, CLR, NOT} OPCODE_T;
  rand OPCODE_T opcode;
  bit [1:0] n_operands;
  ...
  constraint c_operands {
    if (n_operands == 0)
      opcode == NOP || opcode == HALT;
    else if (n_operands == 1)
      opcode == CLR || opcode == NOT;
    ...
  }
endclass
```

You can see that having one large constraint can quickly get out of control as you add further expressions for each operand, addressing modes, etc. A more modular approach is to use a separate constraint for each flavor of instruction, and then disable all but the one you need.

While the constraints are simpler with this approach, the process of turning them on and off is more complex. For example, when you turn off all constraints that create data, you are also disabling all the ones that check the data's validity.

Example 6-29 Turning constraints on and off with `constraint_mode`

```
class Instruction;
  rand OPCODE_T opcode;
  …
  constraint c_no_operands {
    opcode == NOP || opcode == HALT;}
  constraint c_one_operand {
    opcode == CLR || opcode == NOT;}
endclass

Instruction instr;
initial begin
  instr = new;

  // Generate an instruction with no operands
  instr.constraint_mode(0);  // Turn off all constraints
  instr.c_no_operands.constraint_mode(1);
  assert (instr.randomize());

  // Generate an instruction with one operand
  instr.constraint_mode(0);  // Turn off all constraints
  instr.c_one_operand.constraint_mode(1);
  assert (instr.randomize());
  end
```

6.10.5   Specifying a constraint in a test using in-line constraints

If you keep adding constraints to a class, it becomes hard to manage and control. Soon, everyone is checking out the same file from your source control system. Many times a constraint is only used by a single test, so why have it visible to every test? One way to localize the effects of a constraint is to use in-line constraints, **randomize() with**, shown in section 6.8. This works well if your new constraint is additive to the default constraints. For the worst case, you can disable any constraint that conflicts with what you are trying to do. For example, if only one test injects a particular flavor of corrupted data, it could first turn off the particular validity constraint that checks for that error.

There are several problems with using in-line constraints. The first is that now your constraints are in multiple locations. If you add a new constraint to the original class, it may conflict with the in-line constraint. The second is that it can be very hard for you to reuse an in-line constraint across multiple tests. By definition, an in-line constraint only exists in one piece of code. You could

put it in a routine in a separate file and then call it as needed. But at that point it has become nearly the same as an external constraint.

6.10.6 Specifying a constraint in a test with external constraints

The body of a constraint does not have to be defined within the class, just as a routine body can be defined externally, as shown in section 4.11. Your data class could be defined in one file, with one empty constraint. Then each test could define its own version of this constraint to generate its own flavors of stimulus.

Example 6-30 Class with an external constraint

```
// packet.sv
class Packet;
  rand bit [7:0] length;
  rand bit [7:0] payload[];
  constraint c_valid {length > 0;
                      payload.size == length;}
  constraint c_external;
endclass
```

Example 6-31 Program defining external constraint

```
// test.sv
program test;
constraint Packet::c_external {length == 1;}
...
endprogram
```

External constraints have several advantages over in-line constraints. They can be put in a file and thus reused between tests. An external constraint applies to all instances of the class, while an in-line constraint only affects the single call to randomize. Consequently, an external constraint provides a primitive way to change a class without having to learn advanced OOP techniques. But you can only add constraints, not alter existing ones, and you need to define the external constraint prototype in the original class.

Like in-line constraints, external constraints can cause problems, as the constraints are spread across multiple files.

A final consideration is what happens when the body for an external constraint is never defined. The SystemVerilog LRM does not currently specify what should happen in this case. Before you build a testbench with many external constraints, find out how your simulator handles missing definitions. Is this an error that prevents simulation, just a warning, or no message at all?

6.10.7   Extending a class

In Chapter 8, you will learn how to extend a class. With this, you can take a testbench that uses a given class, and swap in an extended class that has additional or redefined constraints, routines, and variables. Learning OOP techniques requires a little more study, but the flexibility of this new approach repays with great rewards.

6.11   Common Randomization Problems

You may be comfortable with procedural code, but writing constraints and understanding random distributions requires a new way of thinking. Here are some issues you may encounter when trying to create random stimulus.

6.11.1   Use care with signed variables

When creating a testbench, you may be tempted to use the `int`, `byte`, or other signed types for counters and other simple variables. Don't use them in random constraints unless you really want signed values. What values are produced when the class in Example 6-32 is randomized? It has two random variables and wants to make the sum of them 64.

Example 6-32 Signed variables cause randomization problems

```
class SignedVars;
  rand byte pkt1_len, pk2_len;
  constraint total_len {
    pkt1_len + pk2_len == 64;
  }
endclass
```

Obviously, you could get pairs of values such as (32, 32) and (2, 62). But you could also see (-64, 128), as this is a legitimate solution of the equation, even though it may not be what you wanted. To avoid meaningless values such as negative lengths, use only unsigned random variables.

Example 6-33 Randomizing unsigned 32-bit variables

```
class Vars32;
  rand logic [31:0] pkt1_len, pk2_len;  // unsigned type
  constraint total_len {
    pkt1_len + pk2_len == 64;
  }
endclass
```

Even this version causes problems, as large values of **pkt1_len** and **pkt2_len**, such as **32'h80000040** and **32'h80000000**, wrap around when added together and give **32'd64** / **32'h40**. You might think of adding another pair of constraints to restrict the values of these two variables, but the best approach is to make them only as wide as needed, and to avoid using 32-bit variables in constraints.

Example 6-34 Randomizing unsigned 8-bit variables

```
class Vars8;
  rand logic [7:0] pkt1_len, pk2_len;  // 8-bits wide
  constraint total_len {
    pkt1_len + pk2_len == 8'd64;       // 8-bits wide
  }
endclass
```

6.12   Iterative and Array Constraints

The constraints presented so far allow you to specify limits on scalar variables. What if you want to randomize an array? The **foreach** statement and several array functions let you shape the distribution of the values.

Using the **foreach** constraint creates many constraints and slow down simulation. A good solver can quickly solve hundreds of constraints but may slow down with thousands. Especially slow are nested **foreach** constraints, as they produce N^2 constraints for an array of size N. See section 6.12.5 for an algorithm that used **randc** variables instead of nested **foreach**.

6.12.1   Array size

The easiest array constraint to understand is the **size** function. You are specifying the number of elements in a dynamic array or queue.

Example 6-35 Constraining dynamic array size

```
class dyn_size;
  rand reg [31:0] d[];
  constraint d_size {d.size inside {[1:10]}; }
endclass
```
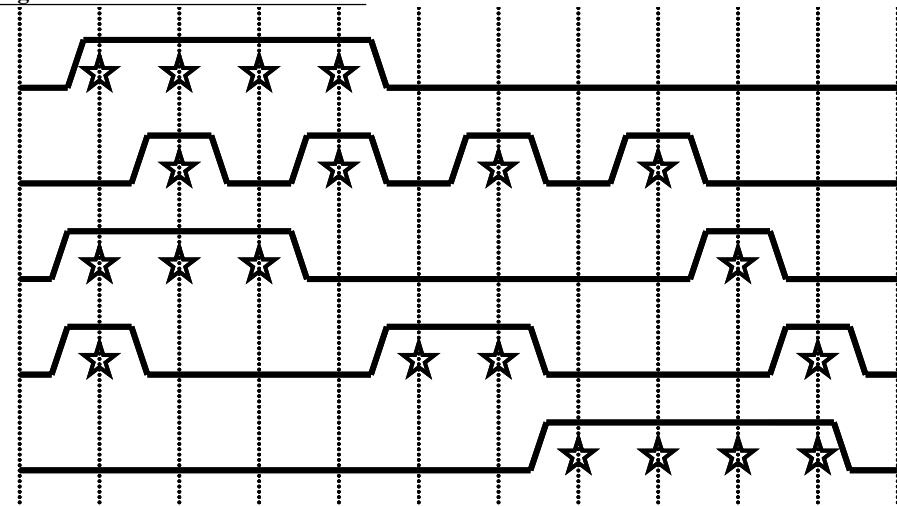
Using the **inside** constraint lets you set a lower and upper boundary on the array size. In many cases you may not want an empty array, that is, **size==0**. Remember to specify an upper limit; otherwise, you can end up

with thousands or millions of elements, which can cause the random solver to take an excessive amount of time.

6.12.2   Sum of elements

You can send a random array of data into a design, but you can also use it to control the flow. Perhaps you have an interface that has to transfer four data words. The words can be sent consecutively or over many cycles. A strobe signal tells when the data is valid. Here are some legitimate strobe patterns, sending four values over ten cycles.

Figure 6-2 Random strobe waveforms

You can create these patterns using a random array. Constrain it to have four bits enabled out of the entire range using the **sum** function.

Example 6-36 Random strobe pattern class

```
parameter MAX_TRANSFER_LEN = 10;

class StrobePat;
  rand bit strobe[MAX_TRANSFER_LEN];
  constraint c_set_four { strobe.sum == 3'h4; }
endclass
```

Example 6-37 Using random strobe pattern class

```
initial begin
  StrobePat sp;
  int count = 0;        // Index into data array

  sp = new();
  assert (sp.randomize);

  foreach (sp.strobe[i]) begin
    bus.cb.strobe = sp.strobe[i];
    // If strobe is enabled, drive out next data word
    if (sp.strobe[i])
      bus.cb.data = data[count++];
  end
end
```

As you remember from Chapter 2, the sum of single-bit variables would normally be a single bit, e.g., 0 or 1. Example 6-36 compares **strobe.sum** to a 3-bit value (**3'h4**), so the sum is calculated with 3-bit precision.

6.12.3   Issues with array constraints

The sum function looks simple but can cause several problems because of Verilog's arithmetic rules.

Start with a simple concept. You want to generate from one to eight transactions, such that the total length of all of them is less than 1024 bytes. Here is a first attempt. The **len** field is a byte in the original transaction.

Example 6-38 First attempt at sum constraint: bad_sum1

```
class bad_sum1;
  rand byte len[];
  constraint c_len {len.sum < 1024;
                    len.size inside {[1:8]};}
  function void display;
    $write("size=%d, sum=%4d ", len.size, len.sum);
    foreach(len[i]) $write("%4d ", len[i]);
    $display;
  endfunction
endclass
```

Example 6-39 Program to try constraint with array sum

```
program automatic test;
  bad_sum1 c;
  initial begin
    c = new;
    repeat (10) begin
      assert (c.randomize());
      c.display;
    end
  end
endprogram
```

Example 6-40 Output from bad_sum1

```
len: sum=  81, val=  62 -20  39
len: sum=  39, val= -27  67   1  76 -97 -58  77
len: sum=  38, val=  60 -22
len: sum=  72, val=-120  29 123 102 -41 -21
len: sum= -53, val= -58 -85-115 112-101 -62
```

This generates some smaller lengths, but the sum is sometimes negative and is always less than 127 — definitely not what you wanted! Try again, this time with an unsigned field. (The **display** function is unchanged.)

Example 6-41 Second attempt at sum constraint: bad_sum2

```
class bad_sum2;
  rand bit [7:0] len[];  // 8 bits
  constraint c_len {len.sum < 1024;
                    len.size inside {[1:8]};}
endclass
```

Example 6-42 Output from bad_sum2

```
len: sum=  79, val=  88 100 246   2  14 228 169
len: sum= 120, val=  74  75 141  86
len: sum=  39, val=  39
len: sum= 193, val=  31 156 172  33  57
len: sum= 173, val=  59 150  25 101 138 212
```

Example 6-42 has a subtle problem: the sum of all transaction lengths is always less than 256, even though you constrained the array sum to be less than 1024. The problem here is that in Verilog, the sum of many 8-bit values is computed using an 8-bit result. Bump the **len** field up to 32 bits using the **uint** type from Chapter 2.

Example 6-43 Third attempt at sum constraint: bad_sum3

```
class bad_sum3;
  rand uint len[];   // 32 bits
  constraint c_len {len.sum < 1024;
                    len.size inside {[1:8]};}
endclass
```

Example 6-44 Output from bad_sum3

```
len:  sum=   245,  val=1348956995  3748256598  985546882
2507174362
len:  sum=   600,  val=2072193829  315191491  484497976
3050698208 2300168220 3988671456 3998079060 970369544
len:  sum=    17,  val=1924767007  3550820640  4149215303
3260098955
len:  sum=   440,  val=3192781444  624830067  1300652226
4072252356 3694386235
len: sum= 864, val=3561488468 733479692
```

Wow – what happened here? This is similar to the signed problem in section 6.11.1, in that the sum of two very large numbers can wrap around to a small number. You need to limit the size based on the comparison in the constraint.

Example 6-45 Fourth attempt at sum_constraint: bad_sum4

```
class bad_sum4;
  rand bit [9:0] len[];   // 10 bits
  constraint c_len {len.sum < 1024;
                    len.size inside {[1:8]};}
endclass
```

Example 6-46 Output from bad_sum4

```
len: sum= 989, val= 787 202
len: sum=1021, val= 564   76 132 235   0   8   6
len: sum= 872, val= 624 101 136  11
len: sum= 978, val= 890   88
len: sum= 905, val= 663 242
```

This does not work either as the individual **len** field fields are more than 8 bits, so the **len** values are often greater than 255. You need to specify that each **len** field is between 1 and 255, but use a 9-bit field so they sum correctly. This requires constraining every element of the array.

6.12.4   Constraining individual array and queue elements

SystemVerilog lets you constrain individual elements of an array using **foreach**. While you might be able to write constraints for a fixed-size array by listing every element, the **foreach** style is more compact. The only practical way to constrain a dynamic array or queue is with **foreach**.

Example 6-47 Simple `foreach` constraint: `good_sum5`

```
class good_sum5;
  rand uint len[];
  constraint c_len {foreach (len[i])
                         len[i] inside {[1:255]};
                      len.sum < 1024;
                      len.size inside {[1:8]};}
endclass
```

Example 6-48 Output from `good_sum5`

```
len: sum=1011, val=  83 249 197 187 152  95  40   8
len: sum=1012, val= 213 252 213  44 196  20  20  54
len: sum= 370, val= 118  76 176
len: sum= 976, val= 233 187  44 157 201  81  73
len: sum= 412, val= 172 167  73
```

The addition of the constraint for individual elements fixed the example. Note that the **len** array can be 10 or more bits wide, but must be unsigned.

You can specify constraints between array elements as long as you are careful about the endpoints. The following class creates an ascending list of values by comparing each element to the previous, except for the first.

Example 6-49 Creating ascending array values with `foreach`

```
class Ascend;
  rand uint d[10];
  constraint c {
    foreach (d[i])      // For every element
      if (i>0)          // except the first
        d[i] > d[i-1]; // compare with previous element
  }
endclass
```

How complex can these constraints become? Constraints have been written to solve Einstein's problem (a logic puzzle with five people, each with five separate attributes), the Eight Queens problem (place eight queens on a chess board so that none can capture each other), and Sudoku.

6.12.5   Generating an array of unique values

How can you create an array of random values that is unique? For example, you may need to assign ID numbers to N bus drivers, which are in the range of 0 to MAX-1 where MAX >=N.

Your may be tempted to use a constraint with nested **foreach** specifying **a[i] != a[j]**. The SystemVerilog solver expands out these equations, so an array of 30 values creates almost 1000 constraints.

Instead, try procedural code in a **post_randomize** function that uses a **randc** variable. You have to put the **randc** variable in a helper class so that you can randomize the same variable over and over.

Example 6-50 `UniqueArray` class

```
// Generate a random array of unique values
class UniqueArray;
  int max_array_size, max_value;
  rand bit [7:0] a[];        // Array of unique values
  constraint c_size {a.size inside {[1:max_array_size]};}

  function new(int max_array_size=2, max_value=1);
  this.max_array_size = max_array_size;
  if (max_value < max_array_size)
    this.max_value = max_array_size;
  else
    this.max_value = max_value;
  endfunction

  // At this point array is allocated, fill w/unique vals
  function void post_randomize;
    RandcRange rr = new(max_value);
    foreach (a[i]) begin
      assert (rr.randomize());
      a[i] = rr.value;
    end
  endfunction

  function void display;
    $write("Size: %3d:", a.size());
    foreach (a[i]) $write("%4d", a[i]);
    $display;
  endfunction
endclass
```

Example 6-51 Unique value generator

```
// Create unique random values in a range 0:max
class RandcRange;
  randc bit [15:0] value;
  int max_value;  // Maximum possible value

  function new(int max_value = 10);
    this.max_value = max_value;
  endfunction

  constraint c_max_value {value < max_value;}
endclass
```

   Here is a program using the **UniqueArray** class.

Example 6-52 Using the UniqueArray class

```
program automatic test;
  UniqueArray ua;
  initial begin
    ua = new(50);                // Array size = 50

    repeat (10) begin
      assert(ua.randomize());  // Create random array
       ua.display;             // Display values
      end
  end
endprogram
```

6.13   Atomic Stimulus Generation vs. Scenario Generation

   Up until now, you have seen atomic random transactions. You have
learned how to make a single random bus transaction, a single network
packet, or a single processor instruction. But your job is to verify that the
design works with real-world stimuli. A bus may have long sequences of
transactions such as DMA transfers or cache fills. Network traffic consists of
extended sequences of packets as you simultaneously read e-mail, browse a
web page, and download music from the net, all in parallel. Processors have
deep pipelines that are filled with the code for routine calls, **for** loops, and
interrupt handlers. Generating transactions one at a time is unlikely to mimic
any of these scenarios.

6.13.1 An atomic generator with history

The easiest way to create a stream of related transactions is to have an atomic generator base some of its random values on ones from previous transactions. The class might constrain a bus transaction to repeat the previous command, such as a write, 80% of the time, and also use the previous destination address plus an increment. You can use the **post_randomize** function to make a copy of the generated transaction for use by the next call to **randomize**.

This scheme works well for smaller cases but gets into trouble when you need information about the entire sequence ahead of time. For example, the DUT may need to know the length of a sequence of network transactions before it starts.

6.13.2 Randsequence

The next way to generate a sequence of transactions is by using the randsequence construct in SystemVerilog. With **randsequence** you describe the grammar of the transaction, using a syntax similar to BNF (Backus-Naur Form).

Example 6-53 Command generator using `randsequence`

```
initial begin
  for (int i=0; i<15; i++) begin
    randsequence (stream)
    stream :  cfg_read := 1 |
              io_read  := 2 |
              mem_read := 5;
    cfg_read : { cfg_read_task; } |
               { cfg_read_task; } cfg_read;
    mem_read : { mem_read_task; } |
               { mem_read_task; } mem_read;
    io_read  : { io_read_task; } |
               { io_read_task; } io_read;

    endsequence
  end // for
end

task cfg_read_task;
  ...
endtask
```

Example 6-53 generates a sequence called **stream**. A **stream** can be either **cfg_read**, **io_read**, or **mem_read**. The random sequence engine randomly picks one. The **cfg_read** label has a weight of 1, while **io_read** is twice as likely to be chosen and **mem_read** is most likely to be chosen, with a weight of 5.

A **cfg_read** can be either a single call to the **cfg_read_task**, or a call to the task followed by another **cfg_read**. As a result, the task is always called at least once, and possibly many times.

One big advantage of **randsequence** is that it is procedural code and you can debug it by stepping though the execution, or adding **$display** statements. When you call **randomize** for an object, it either all works or all fails, but you can't see the steps taken to get to a result.

There are several problems with using **randsequence**. The code to generate the sequence is separate and a very different style from the classes with data and constraints used by the sequence. So if you use both **randomize()** and **randsequence**, you have to master two different forms of randomization. More seriously, if you want to modify a sequence, perhaps to add a new branch or action, you have to modify the original sequence code. You can't just make an extension. As you will see in Chapter 8, you can extend a class to add new code, data, and constraints without having to edit the original class.

6.13.3   Random array of objects

The last form of generating random sequences is to randomize an entire array of objects. You can create constraints that refer to the previous and next objects in the array, and the SystemVerilog solver solves all constraints simultaneously. Since the entire sequence is generated at once, you can then extract information such as the total number of transactions or a checksum of all data values before the first transaction is sent. Alternatively, you can build a sequence for a DMA transfer that is constrained to be exactly 1024 bytes, and let the solver pick the right number of transactions to reach that goal.

6.13.4   Combining sequences

You can combine multiple sequences together to make a more realistic flow of transactions. For example, for a network device, you could make one sequence that resembles downloading e-mail, a second that is viewing a web page, and a third that is entering single characters into web-based form.The techniques to combine these flows is beyond the scope of this book, but you can learn more from the VMM, as described in Bergeron, et al. (2005).

6.14   Random Control

At this point you may be thinking that this process is a great way to create long streams of random input into your design. Or you may think that this is a lot of work if all you want to do is occasionally to make a random decision in your code. You may prefer a set of procedural statements that you can step through using a debugger.

6.14.1   Introduction to `randcase`

You can use **randcase** to make a weighted choice between several actions, without having to create a class and instance. Example 6-54 chooses one of the three branches based on the weight. SystemVerilog adds up the weights $(1+8+1 = 10)$, chooses a value in this range, and then picks the appropriate branch. The branches are not order dependent, the weights can be variables, and they do not have to add up to 100%.

Example 6-54 Random control with `randcase` and `$urandom_range`

```
initial begin
  int len;
  randcase
    1: len = $urandom_range(0, 2);  // 10%: 0, 1, or 2
    8: len = $urandom_range(3, 5);  // 80%: 3, 4, or 5
    1: len = $urandom_range(6, 7);  // 10%: 6 or 7
  endcase
  $display("len=%0d", len);
  end
```

The **$urandom_range** function returns a random number in the specified range.

You can write Example 6-54 using a class and the **randomize** function. For this small case, the OOP version is a little larger. However, if this were part of a larger class, the constraint would be more compact than the equivalent **randcase** statement.

Example 6-55 Equivalent constrained class

```
class LenDist;
  rand int len;
  constraint c
    {len dist {[0:2] := 1, [3:5] := 8, [6:7] := 1}; }
endclass

LenDist lenD;

initial begin
  lenD = new;
  assert (lenD.randomize());
  $display("Chose len=%0d", lenD.len);
  end
```

Code using **randcase** is more difficult to override and modify than random constraints. The only way to modify the random results is to rewrite the code or use variable weights.

Be careful using **randcase**, as it does not leave any tracks behind. For example, you could use it to decide whether or not to inject an error in a transaction. The problem is that the downstream transactors and scoreboard need to know of this choice. The best way to inform them would be to use a variable in the transaction or environment. But if you are going to create a variable that is part of these classes, you could have made it a random variable and used constraints to change its behavior in different tests.

6.14.2   Building a decision tree with **randcase**

You can use **randcase** when you need to create a decision tree. Example 6-56 has just two levels of procedural code, but you can see how it can be extended to use more.

Example 6-56 Creating a decision tree with `randcase`

```
initial begin
  // Level 1
  randcase
  one_write_wt: do_one_write();
  one_read_wt:  do_one_read();
  seq_write_wt: do_seq_write();
  seq_read_wt:  do_seq_read();
  endcase
  end

// Level 2
task do_one_write;
  randcase
  mem_write_wt: do_mem_write();
  io_write_wt:  do_io_write();
  cfg_write_wt: do_cfg_write();
  endcase
endtask

task do_one_read;
  randcase
  mem_read_wt: do_mem_read();
  io_read_wt:  do_io_read();
  cfg_read_wt: do_cfg_read();
  endcase
endtask
```

6.15   Random Generators

How random is SystemVerilog? On the one hand, your testbench depends on an uncorrelated stream of random values to create stimulus patterns that go beyond any directed test. On the other hand, you need to repeat the patterns over and over during debug of a particular test, even if the design and testbench make minor changes.

6.15.1   Pseudorandom number generators

Verilog uses a simple PRNG that you could access with the `$random` function. The generator has an internal state that you can set by providing a seed to `$random`. All IEEE-1364-compliant Verilog simulators use the same algorithm to calculate values.

Example 6-57 shows a simple PRNG, not the one used by SystemVerilog. The PRNG has a 32-bit state. To calculate the next random value, square the state to produce a 64-bit value, take the middle 32 bits, then add the original value.

Example 6-57 Simple pseudorandom number generator
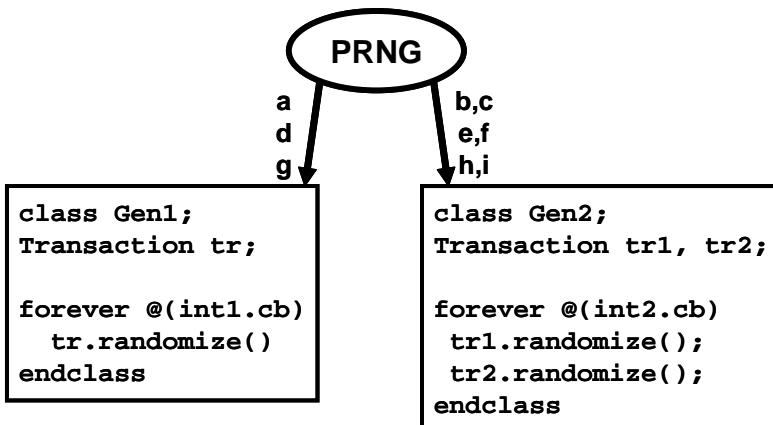
```
reg [31:0] state = 32'h12345678;
function reg [31:0] my_random;
  reg [63:0] s64;
  s64 = state * state;
  state = (s64 >> 16) + state;
  my_random = state;
endfunction
```

You can see how this simple code produces a stream of values that seem random, but can be repeated by using the same seed value. SystemVerilog calls its PRNG for a new values for **randomize** and **randcase**.

6.15.2   Random Stability — multiple generators

Verilog has a single PRNG that is used for the entire simulation. What would happen if SystemVerilog kept this approach? Testbenches often have several stimulus generators running in parallel, creating data for the design under test. If two streams share the same PRNG, they each get a subset of the random values.
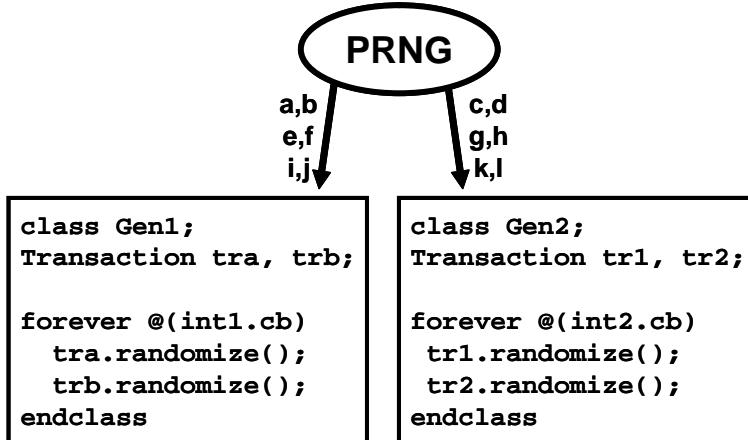
Figure 6-3 Sharing a single random generator

In Figure 6-3, there are two stimulus generators and a single PRNG producing values **a**, **b**, **c**, etc. **Gen2** has two random objects, so during every cycle, it uses twice as many random values as **Gen1**. A problem can occur

when one of the classes changes. `Gen1` gets an additional random variable, and so consumes two random values every time it is called.
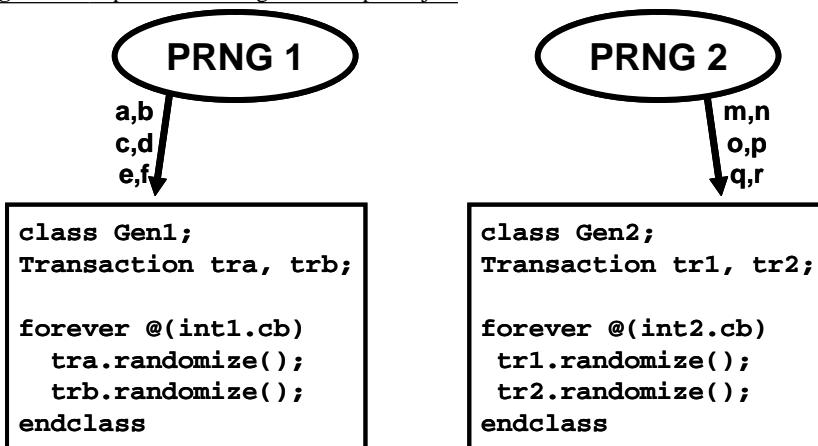
Figure 6-4 First generator uses additional values

This approach changes the values used not only by `Gen1`, but also by `Gen2`.

In SystemVerilog, there is a separate PRNG for every object and thread. Changes to one object don't affect the random values seen by others.

Figure 6-5 Separate random generators per object

6.15.3   Random Stability — hierarchical seeding

Every object and thread have its own PRNG and unique seed. When a new object or thread is started, its PRNG is seeded from its parent's PRNG. Thus a

single seed specified at the start of simulation can create many streams of random stimulus, each distinct.

6.16   Random Device Configuration

An important part of your DUT to test is the configuration of both the internal DUT settings and the system that surrounds it. As described in section 6.2.1, your tests should randomize the environment so that you can be confident it has been tested in as many modes as possible.

Example 6-58 shows how to create a random testbench configuration and modify its results as needed at the test level. The **eth_cfg** class describes the configuration for a 4-port Ethernet switch. It is instantiated in an environment class, which in turn is used in the test. The test overrides one of the configuration values, enabling all 4 ports.

Example 6-58 Ethernet switch configuration class

```
class eth_cfg;
  rand bit [ 3:0] in_use;        // Ports used in test
  rand bit [47:0] mac_addr[4];   // MAC addresses
  rand bit [ 3:0] is_100;        // 100mb mode
  rand int run_for_n_frames;     // # frames in test

  // Force some addr bits when running in unicast mode
  constraint local_unicast {
    foreach (mac_addr[i])
      mac_addr[i][41:40] == 2'b00;
  }

  constraint reasonable {        // Limit test length
    run_for_n_frames inside {[1:100]};
  }
endclass : eth_cfg
```

The configuration class is used in the **Environment** class during several phases. The configuration is constructed in the **Environment** constructor, but not randomized until the **gen_cfg** phase. This allows you to turn constraints on and off before **randomize** is called. Afterwards, you can override the generated values before the **build** phase creates the virtual components around the DUT.

Example 6-59 Building environment with random configuration

```
class Environment;
  eth_cfg cfg;
  eth_src gen[4];
  eth_mii drv[4];

  function new;
    cfg = new;                      // Construct the cfg
  endfunction

  function vid gen_cfg;
    assert(cfg.randomize());  // Randomize the cfg
  endfunction

  // Use random configuration to build the environment
  function void build;
    foreach (src[i])
      if (cfg.in_use[i]) begin
        gen[i] = new(...);
        drv[i] = new(...);
        if (cfg.is_100[i])
           mii[i].set_speed(100);
        end
    endfunction

    task run;
      // Start the testbench transactors
    endtask

    task wrapup;
      // Not currently used
    endtask
enclass : Environment
```

Now you have all the components to build a test, which is described in a program block. The test instantiates the environment class and then runs each step.

Example 6-60 Simple test using random configuration

```
program test;
  Environment env;

  initial begin
    env = new;        // Construct environment
    env.gen_cfg;      // Create random configuration
    env.build;        // Build the testbench environment
    env.run;          // Run the test
    env.wrapup;       // Clean up after test & report
    end
endprogram
```

You may want to override the random configuration, perhaps to reach a corner case. The following test randomizes the configuration class and then enables all the ports.

Example 6-61 Simple test that overrides random configuration

```
program test;
  Environment env;

  initial begin
    env = new;        // Construct environment
    env.gen_cfg;      // Create random configuration

    // Override random in-use – turn all 4 ports on
    env.cfg.in_use = 4'b1111;

    env.build;        // Build the testbench environment
    env.run;          // Run the test
    env.wrapup;       // Clean up after test & report
    end
endprogram
```

6.17   Conclusion

Constrained-random tests are the only practical way to generate the stimulus needed to verify a complex design. SystemVerilog offers many ways to create a random stimulus and this chapter presents many of the alternatives.

A test needs to be flexible, allowing you either to use the values generated by default or to constrain or override the values so that you can reach your goals. Always plan ahead when creating your testbench by leaving sufficient "hooks" so that you can steer the testbench from the test without modifying existing code.
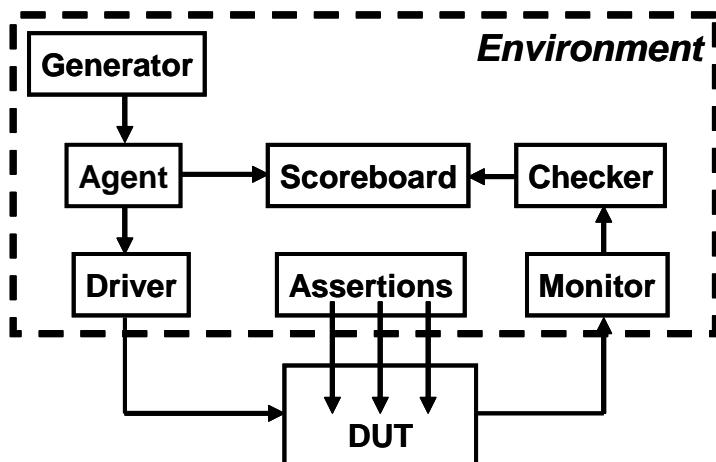
Chapter 7

Threads and Interprocess Communication

7.1   Introduction

In real hardware, the sequential logic is activated on clock edges, while combinational logic is constantly changing when any inputs change. All this parallel activity is simulated in Verilog RTL using **initial** and **always** blocks, plus the occasional gate and continuous assignment statement. To stimulate and check these blocks, your testbench uses many threads of execution, all running in parallel. Most blocks in your testbench environment are modeled with a transactor and run in their own thread.

Figure 7-1 Testbench environment blocks

The SystemVerilog scheduler is the traffic cop that chooses which thread runs next. You can use the techniques in this chapter to control the threads and thus your testbench.

Each of these threads communicates with its neighbors. In Figure 7-1, the generator passes the stimulus to the agent. The environment class needs to know when the generator completes and then tell the rest of the testbench threads to terminate. This is done with interprocess communication constructs such as the standard Verilog events, event control and **wait** constructs, and the SystemVerilog mailboxes and semaphores.[10]
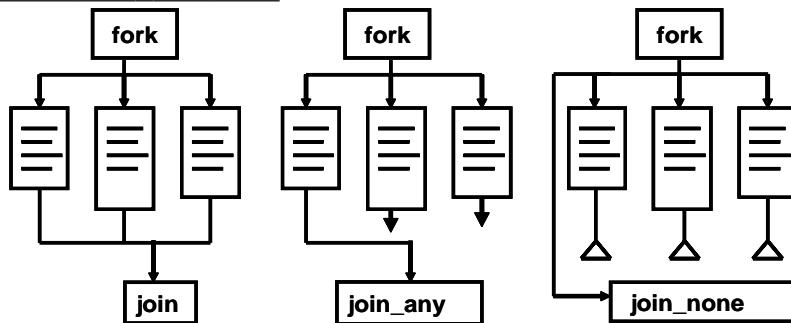
7.2   **Working with Threads**

While all the thread constructs can be used in both modules and program blocks, your testbenches belong in program blocks. As a result, your code always starts with **initial** blocks that start executing at time 0, when the simulator starts. You cannot put an **always** block in a program. However, you can easily get around this by using a **forever** loop in an **initial** block.

Classic Verilog has two ways of grouping statements — with a **begin**...**end** or **fork**...**join**. Statements in a **begin**...**end** run sequentially, while those in a **fork**...**join** execute in parallel. The latter is very limited in that all statements inside the **fork**...**join** have to finish before the rest of the block can continue. As a result, it is rare for Verilog testbenches to use this feature.

SystemVerilog introduces two new ways to create threads — with the **fork**...**join_none** and **fork**...**join_any** statements.

Figure 7-2 Fork...join blocks

Your testbench communicates, synchronizes, and controls these threads with existing constructs such as events, **@** event control, the **wait** and **disable** statements, plus new language elements such as semaphores and mailboxes.

7.2.1   Using **fork**...**join** and **begin**...**end**

Example 7-1 has a **fork**...**join** parallel block with an enclosed **begin**...**end** sequential block, and shows the difference between the two.
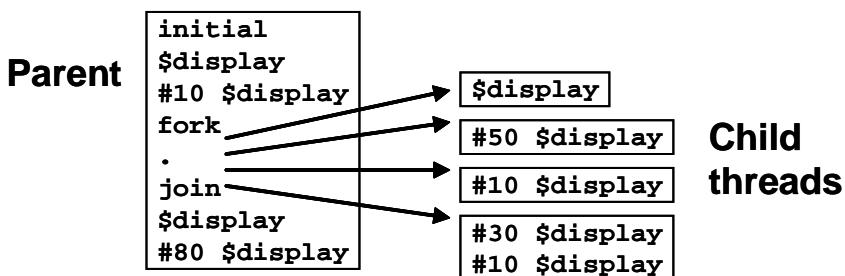
[10.] The SystemVerilog LRM uses "thread" and "process" interchangeably. The term "process" is most commonly associated with Unix processes, in which each contains a program running in its own memory space. Threads are lightweight processes that may share common code and memory, and consume far less resources than a typical process. This book uses the term "thread." However, "interprocess communication" is such a common term that it is used in this book.

Example 7-1 Interaction of `begin...end` and `fork...join`

```
initial begin
     $display("@%0d: start fork...join example", $time);
  #10 $display("@%0d: sequential after #10", $time);
  fork
       $display("@%0d: parallel start", $time);
    #50 $display("@%0d: parallel after #50", $time);
    #10 $display("@%0d: parallel after #10", $time);
    begin
      #30 $display("@%0d: sequential after #20", $time);
      #10 $display("@%0d: sequential after #10", $time);
    end
  join
  $display("@%0d: after join", $time);
  #80 $display("@%0d: final after #80", $time);
end
```

Figure 7-3 `Fork...join` block

Note in the output below that code in the **fork**...**join** executes in parallel, so statements with shorter delays execute before those with longer delays. The **fork**...**join** completes after the last statement, which starts with **#50**.

Example 7-2 Output from `begin...end` and `fork...join`

```
@0: start fork...join example
@10: sequential after #10
@10: parallel start
@20: parallel after #10
@40: sequential after #30
@50: sequential after #10
@60: parallel after #50
@60: after join
@140: final after #80
```

7.2.2   Spawning threads with `fork...join_none`

A **fork**...**join_none** block schedules each statement in the block, but execution continues in the parent thread. The following code is identical to Example 7-1 except that the **join** has been converted to **join_none**.

Example 7-3 Fork...join_none code

```
initial begin
  $display("@%0d: start fork...join_none example",$time);
  #10 $display("@%0d: sequential after #10", $time);
  fork
        $display("@%0d: parallel start", $time);
    #50 $display("@%0d: parallel after #50", $time);
    #10 $display("@%0d: parallel after #10", $time);
    begin
      #30 $display("@%0d: sequential after #20", $time);
      #10 $display("@%0d: sequential after #10", $time);
    end
  join_none
  $display("@%0d: after join_none", $time);
  #80 $display("@%0d: final after #80", $time);
end
```

The diagram for this block is similar to Figure 7-3. Note that the statement after the **join_none** block executes before any statement inside the **fork**...**join_none**.

Example 7-4 Fork...join_none output

```
@0: start fork...join_none example
@10: sequential after #10
@10: after join_none
@10: parallel start
@20: parallel after #10
@40: sequential after #30
@50: sequential after #10
@60: parallel after #50
@90: final after #80
```

7.2.3   Synchronizing threads with `fork...join_any`

A **fork**...**join_any** block schedules each statement in the block. Then, when the first statement completes, execution continues in the parent thread. All other remaining threads continue. The following code is identical to the previous examples, except that the **join** has been converted to **join_any**.

Example 7-5 `Fork...join_any` code

```
initial begin
  $display("@%0d: start fork...join_any example",$time);
  #10 $display("@%0d: sequential after #10", $time);
  fork
       $display("@%0d: parallel start", $time);
    #50 $display("@%0d: parallel after #50", $time);
    #10 $display("@%0d: parallel after #10", $time);
    begin
      #30 $display("@%0d: sequential after #20", $time);
      #10 $display("@%0d: sequential after #10", $time);
    end
  join_any
  $display("@%0d: after join_any", $time);
  #80 $display("@%0d: final after #80", $time);
end
```

Note in the results, the statement **$display("after join_any")** completes after the first statement in the parallel block.

Example 7-6 Output from `fork...join_any`

```
@0: start fork...join_any example
@10: sequential after #10
@10: parallel start
@10: after join_none
@20: parallel after #10
@40: sequential after #30
@50: sequential after #10
@60: parallel after #50
@90: final after #80
```

7.2.4   Creating threads in a class

You can use a **fork**...**join_none** to start a thread, such as the code for a random transactor generator. Example 7-7 shows a generator class with a **run** task that creates N packets. The full testbench has classes for the driver, monitor, checker, and more, all with transactors that need to run in parallel.

Example 7-7 `Generator` class with a `run` task

```
class Generator;

  // Transactor that creates N packets
  task run(int n);
    Packet p;

    fork
      repeat (n) begin
        p = new;
        if (!p.randomize) begin
           $display("Packet randomize failed");
           $finish;
        end
        transmit(p);
      end
    join_none
  endtask
endclass

Generator gen;

initial begin
  gen = new;
  gen.run(10);
  // Start the checker, monitor, and other threads
end
```

There are several points you should notice with Example 7-7. First, the transactor is not started in the **new** task. The constructor should just initialize values, not start any threads. Separating the constructor from the code that does the real work allows you to change any variables before you start executing the code in the object. This allows you to inject errors, modify the defaults, and alter the behavior of the object.

Next, the run task starts a thread in a **fork**...**join_none** block. The thread is an implementation detail of the transactor and should be spawned there, not in the parent class.

7.2.5   Dynamic threads

Verilog's threads are very predictable. You can read the source code and count the **initial**, **always**, and **fork**...**join** blocks to know how many

threads were in a module. SystemVerilog lets you create threads dynamically, and does not require you to wait for them to finish.

In Example 7-8, the testbench generates random transactions and sends them to a DUT that stores them for some predetermined time, and then returns them. The testbench has to wait for the transaction to complete, but does not want to stop the generator.

Example 7-8 Dynamic thread creation

```
program automatic test(busif.TB bus);
  // Code for interface not shown
  task wait_for_tr(Transaction tr);
    fork
      begin
      wait (bus.cb.addr != tr.addr);
      $display("@%0d: Addr match %d", $time, tr.addr);
      end
    join_none
  endtask

  Transaction tr;

  initial
    repeat (10)
      begin
      // Create a random transaction
      tr = new;
      if (!tr.randomize) $finish;

      // Send it into the DUT
      transmit(tr);  // Task not shown

      // Wait for reply from DUT
      wait_for_tr(tr);
  end
endprogram
```

When the **wait_for_tr** task is called, it spawns off a thread to watch the bus for the matching transaction address. During a normal simulation, many of these threads run concurrently. In this simple example, the thread just prints a message, but you could add more elaborate controls.

7.2.6 Automatic variables in threads

A common but subtle bug occurs when you have a loop that spawns threads and you don't save variable values before the next iteration. Example 7-8 only works in a **program** or **module** with automatic storage. If **wait_for_tr** used static storage, each thread would share the same variable **tr**, so later calls would overwrite the value set by earlier ones. Likewise, if the example had the **fork**...**join_none** inside the **repeat** loop, it would try to match incoming transactions using **tr**, but its value would change the next time through the loop. Always use automatic variables to hold values in concurrent threads.

Example 7-9 has a **fork**...**join_none** inside a **for** loop. SystemVerilog schedules the threads inside a **fork**...**join_none** but they are not executed until after the original code blocks, here because of the **#0** delay. So Example 7-9 prints "**3 3 3**" which are the values of the index variable **j** when the loop terminates.

Example 7-9 Bad fork...join_none inside a loop

```
initial begin
  for (int j=0; j<3; j++)
    fork
    $write(j);        // Bug - gets final value of index
    join_none
  #0 $display("\n");
end
```

Example 7-10 Execution of bad fork...join_none inside a loop

```
j    Statement
0    for (j=0; ...
0    Spawn $write(j) [thread 0]
1    j++
1    Spawn $write(j) [thread 1]
2    j++
2    Spawn $write(j) [thread 2]
3    j++
3    join_none
3    #0
3    $write(j)        [thread 0]
3    $write(j)        [thread 1]
3    $write(j)        [thread 2]
3    $display("\n")
```

The **#0** delay blocks the current thread and reschedules it to start later during the current time slot. In Example 7-10, the delay makes the current

thread run after the threads spawned in the **fork**...**join** statement. This delay is useful for blocking a thread, but you should be careful, as excessive use causes race conditions and unexpected results.

You should use **automatic** variables inside a **fork**...**join** statement to save a copy of a variable as shown in Example 7-11.

Example 7-11 Automatic variables in a `fork...join_none`

```
initial begin
  for (int j=0; j<3; j++)
    fork
      automatic int k = j;    // Make copy of index
      $write(k);              // Print copy
    join_none
  #0 $display;
end
```

The **fork**...**join_none** block is split into two parts. The **automatic** variable declaration with initialization runs in the thread inside the **for** loop. During each loop, a copy of **k** is created and set to the current value of **j**. Then the body of the **fork**...**join_none** (**$write**) is scheduled, including a copy of **k**. After the loop finishes, **#0** blocks the current thread, so the three threads run, printing the value of their copy of **k**. When the threads complete, and there is nothing else left during the current time-slot region, SystemVerilog advances to the next statement and the **$display** executes.

Example 7-12 traces the code and variables from Example 7-11. The three copies of the automatic variable **k** are called **k0**, **k1**, and **k2**.

Example 7-12 Steps in executing automatic variable code

```
j  k0 k1 k2   Statement
0             for (j=0; ...
0  0          Create k0, spawn $write(k) [thread 0]
1  0          j++
1  0  1       Create k1, spawn $write(k) [thread 1]
2  0  1       j++
2  0  1  2    Create k2, spawn $write(k) [thread 2]
3  0  1  2    j<3
3  0  1  2    join_none
3  0  1  2    #0
3  0  1  2    $write(k0)      [thread 0]
3  0  1  2    $write(k1)      [thread 1]
3  0  1  2    $write(k2)      [thread 2]
3  0  1  2    $display("\n")
```

7.2.7   Disabling a single thread

Just as you may need to create threads in the testbench, you may also need to stop them. The Verilog **disable** statement works on SystemVerilog threads. Here is the **wait_for_tr** task, this time using a **fork**...**join_any** plus a **disable** to create a watch with a time-out.

Example 7-13 Disabling a thread

```
parameter TIME_OUT = 1000;

task wait_for_tr(Transaction tr);
  fork

    begin
    // Wait for response, or some maximum delay
    fork : timeout_block
      wait (bus.cb.addr != tr.addr);
      #TIME_OUT $display("@%0d: Error: timeout", $time);
    join_any
    disable timeout_block;

    $display("@%0d: Addr match %d", $time, tr.addr);
    end

  join_none
endtask
```

The task and outermost **fork**...**join_none** are identical to Example 7-8. This version has two threads inside a **fork**...**join_any** such that the simple **wait** is done in parallel with a delayed display. If the correct bus address comes back quickly enough, the **wait** construct completes, the **join_any** executes, and then the **disable** kills off the remaining thread. However, if the bus address does not get the right value before the **TIME_OUT** delay completes, the error message is printed, the **join_any** executes, and the **disable** kills the thread with the **wait**.

7.2.8   Disabling multiple threads

Example 7-13 used the classic Verilog **disable** statement to stop the threads in a named block. SystemVerilog introduces the **disable fork** statement so you can stop all child threads that have been spawned from the current thread. Watch out, as you might unintentionally stop too many threads, such as those created from routine calls. You should always surround the target code with a **fork**...**join** to limit the scope of a **disable fork**

statement. Example 7-14 has an additional **begin**...**end** block inside the **fork**...**join** to make the statements sequential.

The following sections show how you can asynchronously disable multiple threads. This can cause unexpected behavior, so you should watch out for side effects when a thread is stopped midstream. You may instead want to design your algorithm to check for interrupts at stable points, then gracefully give up its resources.

The next few examples use the **wait_for_tr** task from Example 7-13. You can just think of this task as doing a **#TIME_OUT**.
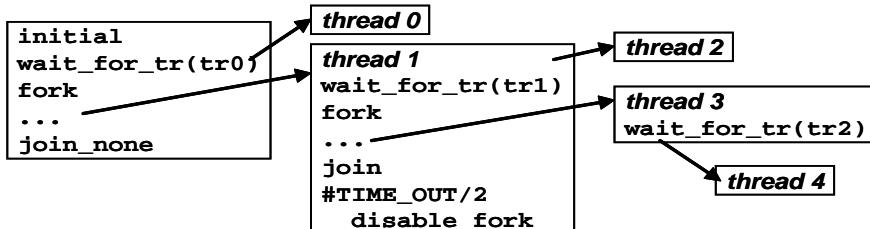
Example 7-14 Limiting the scope of a `disable fork`

```
initial begin
  wait_for_tr(tr0);        // Spawn thread 0

  // Create a thread to limit scope of disable
  fork
    begin
    wait_for_tr(tr1);      // Spawn thread 2
    fork                   // Spawn thread 3
      wait_for_tr(tr2);    // Spawn thread 4
    join

    // Stop threads 1 & 2, but leave 0 alone
    #(TIME_OUT/2) disable fork;
   end
  join
end
```

Figure 7-4 Fork...join block diagram

The code calls **wait_for_tr** that starts thread 0. Next a **fork**...**join** creates thread 1. Inside this thread, one is spawned by the **wait_for_tr** task and one by the innermost **fork**...**join**, which spawns thread 4 by calling the task. After a delay, a **disable fork** stops the child threads. Only threads 2, 3, and 4 are below thread 1, so they are the only ones stopped. Thread 0 is outside the **fork**...**join** block that has the **disable**, so it is unaffected.

Example 7-15 is the more robust version of Example 7-14, with **disable** with a label that explicitly names the threads that you want to stop.

Example 7-15 Using `disable` label to stop threads

```
initial begin
  wait_for_tr(tr0);     // Spawn thread 0
  begin : threads_1_2
    wait_for_tr(tr1);   // Spawn thread 1
    wait_for_tr(tr2);   // Spawn thread 2
  end

  // Stop threads 1 & 2, but leave 0 alone
  #(TIME_OUT/2) disable threads_1_2;
  join
end
```

7.2.9   Waiting for all spawned threads

In SystemVerilog, when all the initial blocks in the program are done, the simulator exits. However, you can spawn many threads, which might still be running. Use the **wait fork** statement to wait for all child threads.

Example 7-16 Using `wait fork` to wait for child threads

```
task run_threads;
  ...                   // Create some transactions

  wait_for_tr(tr1);     // Spawn first thread
  wait_for_tr(tr2);     // Spawn second thread
  wait_for_tr(tr3);     // Spawn third thread

  ...                   // Do some other work

  // Now wait for the above threads to complete
  wait fork;
endtask
```

7.3   Interprocess Communication

All these threads in your testbench need to synchronize and exchange data. At the most basic level, one thread waits for another, such as the environment object waiting for the generator to complete. Multiple threads might try to access a single resource such as bus in the DUT, so the testbench needs to ensure that one and only one thread is granted access. At the highest level, threads need to exchange data such as transaction objects that are passed from the generator to the agent.

7.4   Events

A Verilog event synchronizes threads. It is similar to a phone, where one person waits for a call from another person. In Verilog a thread waits for an event with the **@** operator. This operator is edge sensitive, so it always blocks, waiting for the event to change. Another thread triggers the event with the **->** operator, unblocking the first thread. SystemVerilog enhances the Verilog event in several ways.

First, an event is now a handle to a synchronization object that can be passed around to routines. This feature allows you to share events across objects without having to make the events global. So you can give a phone number to an object to be called later.

There is always the possibility of a race condition in Verilog where one thread blocks on an event at the same time another triggers it. If the triggering thread executes before the blocking thread, the trigger is missed. SystemVerilog introduces the **triggered** function that lets you check whether an event has been triggered, including during the current time-slot. A thread can wait on this function instead of blocking with the **@** operator.

7.4.1   Blocking on the edge of an event

When you run this code, one initial block starts, triggers its event, and then blocks on the other event. The second block starts, triggers its event (waking up the first), and then blocks on the first event. But the second thread locks up because it missed the first event, which is a zero-width pulse.

Example 7-17 Blocking on an event in Verilog

```
event e1, e2;
initial begin
  $display("@%0d: 1: before trigger", $time);
  -> e1;
  @e2;
  $display("@%0d: 1: after trigger", $time);
end

initial begin
  $display("@%0d: 2: before trigger", $time);
  -> e2;
  @e1;
  $display("@%0d: 2: after trigger", $time);
 end
```

Example 7-18 Output from blocking on an event

```
@0: 1: before trigger
@0: 2: before trigger
@0: 1: after trigger
```

7.4.2 Waiting for an event trigger

Instead of the edge-sensitive block **@e1**, use the level-sensitive **wait(e1.triggered)**. This does not block if the event has been triggered during this time step. Otherwise, it waits until the event is triggered.

Example 7-19 Waiting for an event

```
event e1, e2;

initial begin
  $display("\n@%0d: 1: before trigger", $time);
  -> e1;
  wait (e2.triggered);
  $display("@%0d: 1: after trigger", $time);
end

initial begin
  $display("@%0d: 2: before trigger", $time);
  -> e2;
  wait (e1.triggered);
  $display("@%0d: 2: after trigger", $time);
end
```

When you run this code, one initial block starts, triggers its event, and then blocks on the other event. The second block starts, triggers its event (waking up the first) and then blocks on the first event.

Example 7-20 Output from waiting for an event

```
@0: 1: before trigger
@0: 2: before trigger
@0: 1: after trigger
@0: 2: after trigger
```

7.4.3 Passing events

As described above, an event in SystemVerilog can be passed as an argument to a routine. In Example 7-21, an event is used by a transactor to signal when it has completed.

Example 7-21 Passing an event into a constructor

```
class Generator;
  event done;
  function new (event done); // Pass event from TB
    this.done = done
  endfunction

  task run;
    fork
      begin
      ...                    // Create transactions
      -> done;               // Tell the test we are done
      end
    join_none
  endtask
endclass

program automatic test;
  event gen_done;
  Generator gen;

  initial begin
    gen = new(gen_done);     // Instantiate testbench
    gen.run;                 // Run transactor
    wait(gen_done.triggered); // Wait for finish
  end
endprogram
```

7.4.4 Waiting for multiple events

In Example 7-21, you had a single generator that fired a single event. What if your testbench environment class must wait for multiple child processes to finish, such as N generators? The easiest way is to use **wait fork**, that waits for all child processes to end. The problem is that this also waits for all the transactors, drivers, and any other threads that were spawned by the environment. You need to be more selective. You still want to use events to synchronize between the parent and child threads.

You could use a **for** loop in the parent to wait for each event, but that would only work if thread 0 finished before thread 1, which finished before thread 2, etc. If the threads finish out of order, you could be waiting for an event that triggered many cycles ago.

The solution is to make a new thread and then spawn children from there that each block on an event for each generator. Now you can do a **wait fork** because you are being more selective.

Example 7-22 Waiting for multiple threads with `wait fork`

```
event done[N_GENERATORS];

initial begin
  foreach (gen[i]) begin
    gen[i] = new; // Create N generators
    gen[i].run;   // Start them running
  end

  // Wait for all gen to finish by waiting for each event
  foreach (gen[i])
    fork
      automatic int k = i;
      wait (done[k].triggered);
    join_none

  wait fork;  // Wait for all those triggers to finish
end
```

Another way to solve this problem is to keep track of the number of events that have triggered.

Example 7-23 Waiting for multiple threads by counting triggers

```
event done[N_GENERATORS];

initial begin
  foreach (gen[i]) begin
    gen[i] = new; // Create N generators
    gen[i].run;   // Start them running
  end

  // Wait for all generators to finish
  foreach (gen[i])
    fork
      automatic int k = i;
      begin
      wait (done[k].triggered);
      done_count++;
      end
    join_none

  wait fork;      // Wait for all the triggers to finish
end
```

That was slightly less complicated. Why not get rid of all the events and just wait on a count of the number of running generators? This count can be a static variable in the **Generator** class. Note that most of the thread manipulation code has been replaced with a single **wait** construct.

The last block in Example 7-24 waits for the count using the handle **gen[0]**. Any handle to an object gives you access to the static variables.

Example 7-24 Waiting for multiple threads using a thread count

```
class Generator;
  static int thread_count = 0;

  task run;
    thread_count++;          // Start another thread
    fork
      begin
      // Do the real work in here
      // And when done, decrement the thread count
      thread_count--;
      end
    join_none
  endtask
endclass

Generator gen[N_GENERATORS];

initial begin
  // Create N generators
  foreach (gen[i])
    gen[i] = new;

  // Start them running
  foreach (gen[i])
    gen[i].run;

  // Wait for the generators to complete
  wait (gen[0].thread_count == 0);
end
```

7.5   Semaphores

A semaphore allows you to control access to a resource. Imagine that you and your spouse share a car. Obviously, only one person can drive it at a time. You can manage this situation by agreeing that whoever has the key can drive it. When you are done with the car, you will give up the car so that the other

person can use it. The key is the semaphore that makes sure only one person has access to the car. In operating system terminology, this is known as "mutually exclusive access," so a semaphore is known as a mutex and is used to control access to a resource.

Semaphores can be used in a testbench when you have a resource, such as a bus, that may have multiple requestors from inside the testbench but, as part of the physical design, can only have one driver. In SystemVerilog, a thread that requests a key when one is not available always blocks. Multiple blocking threads are queued in FIFO order.

7.5.1   Semaphore operations

There are three operations for a semaphore. You create a semaphore with one or more keys using the **new** method, get one or more keys with **get**, and return one or more keys with **put**. If you want to try to get a semaphore, but not block, use the **try_get** function. It returns 1 if there are enough keys, and 0 if there are insufficient keys.

Example 7-25 Semaphores controlling access to hardware resource

```
program automatic test;
  semaphore sem;          // Create a semaphore
  initial begin
    sem = new(1);         // Allocate with 1 key
    fork
      sequencer;          // Spawn two threads that both
      sequencer;          //   do bus transactions
    join
  end

  task sequencer;
    repeat($urandom%10)   // Random wait, 0-9 cycles
      @bus.cb;
    sendTrans;            // Execute the transaction
  endtask

  task sendTrans;
    sem.get(1);           // Get the key to the bus
    @bus.cb;              // Drive signals onto bus
    bus.cb.addr <= t.addr;
    ...
    sem.put(1);           // Put it back when done
  endtask
endprogram
```

7.5.2   Semaphores with multiple keys

There are two things you should watch out for with semaphores. First, you can put more keys back than you took out. Suddenly you may have two keys but only one car! Secondly, be very careful if your testbench needs to get and put multiple keys. Perhaps you have one key left, and a thread requests two, causing it to block. Now a second thread requests a single semaphore – what should happen? In SystemVerilog the second request is blocked because of the FIFO ordering, even though there are enough keys.

If you need to let smaller requests jump in before larger ones, you can always write your own class. Perhaps your hardware resembles a restaurant and you have a queue of people waiting for tables. A party of two should be able to jump ahead of groups of four in the line if there is space available.

7.6   Mailboxes
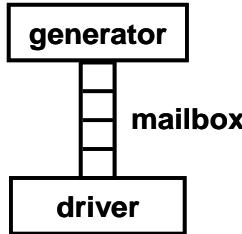
How do you pass information between two threads? Perhaps your generator needs to create many transactions and pass them to a driver. You might be tempted to just have the generator thread call a task in the driver. But then the generator needs to know the hierarchical path to the driver task, making your code less reusable. Additionally, this style forces the generator to run at the same speed as the driver, that can cause synchronization problems if one generator needs to control multiple drivers.

Think of your generator and driver as transactors that are autonomous objects that communicate through a channel. Each object gets a transaction from an upstream object (or creates it, as in the case of a generator), does some processing, and then passes it to a downstream object. The channel must allow its driver and receiver to operate asynchronously. You may be tempted to just use a shared array or queue, but it can be difficult to create code that reads, writes, and blocks between threads.

The solution is a SystemVerilog mailbox. From a hardware point of view, the easiest way to think about a mailbox is that it is just a FIFO, with a source and sink. The source puts data into the mailbox, and the sink gets values from the mailbox. Mailboxes can have a maximum size or can be unlimited. When the source puts a value into a sized mailbox that is full, it blocks until data is removed. Likewise, if a sink tries to remove data from a mailbox that is empty, it blocks until data is put into the mailbox.

Figure 7-5 A mailbox connecting two transactors

A mailbox is an object and thus has to be instantiated by calling the **new** function. This takes an optional **size** argument to limit the number of entries in the mailbox. If the size is 0 or not specified, the mailbox is unbounded and can hold an unlimited number of entries.

You put data into a mailbox with the **put** task, and remove it with the **get** task. A **put** can block if the mailbox is full and a **get** blocks if it is empty. The **peek** task gets a copy of the data in the mailbox but does not remove it.

The data can be a single value, such as an integer, or logic of any size. You can put a handle into a mailbox, not an object. By default, a mailbox does not have a type, so you can put any mix of data into it. Don't do it! Stick to one type per mailbox.

A classic bug is a loop that randomizes objects and puts them in a mailbox, but the object is only constructed once, outside the loop. Since there is only one object, it is randomized over and over. A mailbox only holds handles, not objects, so you end up with a mailbox containing multiple handles that all point to the single object. The code that gets the handles from the mailbox just sees the last set of random values. The solution is to make sure your loop has all three steps of constructing the object, randomizing it, and putting it in the mailbox. This bug is so common that it is also mentioned in section 4.14.3. This type of loop is known as the Factory Pattern and described in section 8.3.

If you don't want your code to block, use the **try_get** and **try_peek** functions. If they are successful, they return a nonzero value; otherwise, they return 0. These are more reliable than the **num** function, as the number of entries can change between when you measure it and when you next access the mailbox.

7.6.1   Mailbox in a testbench

Example 7-26 shows a Generator and Driver exchanging transactions using a mailbox.

Example 7-26 Exchanging objects using a mailbox: the `Generator` class

```
program mailbox_example(bus_if.TB bus, ...);

  class Generator;
    Transaction tr;
    mailbox mbx;

    function new(mailbox mbx);
      this.mbx = mbx;
    endfunction

    task run;
      repeat (10) begin
        tr = new;
        assert(tr.randomize);
        mbx.put(tr);      // Send out transaction
      end
    endtask
  endclass

  class Driver;
    Transaction tr;
    mailbox mbx;

    function new(mailbox mbx);
      this.mbx = mbx;
    endfunction

    task run;
      repeat (10) begin
        mbx.get(tr);        // Fetch next transaction
        @(posedge busif.cb.ack);
        bus.cb.kind <= tr.kind;
        ...
      end
    endtask
  endclass

  mailbox mbx;              // Mailbox connecting gen & drv
  Generator gen;
  Driver drv;
  initial begin
    mbx = new;
    gen = new(mbx);
```

```
    drv = new(mbx);
    fork
      gen.run();              // Spawn the generator
      drv.run();              // Spawn the driver
    join
  end
endprogram
```

7.6.2   Bounded mailboxes

By default, mailboxes are similar to an unlimited FIFO — a producer can put any number of objects into a mailbox before the consumer gets the objects out. However, you may want the two threads to operate in lockstep so that the producer blocks until the consumer is done with the object.

You can specify a maximum size for the mailbox when you construct it. The default mailbox size is 0 which creates an unbounded mailbox. Any size greater than 0 creates a "bounded mailbox." If you attempt to put more objects than this limit and **put** will block until you get an object, creating more room.

Example 7-27 Bounded mailbox

```
program automatic bounded;
  mailbox mbx;

  initial begin
    mbx = new(1);  // Size = 1
    fork

      // Producer
      for (int i=1; i<4; i++) begin
        $display("@%0d: Producer: putting %0d", $time, i);
        mbx.put(i);
        $display("@%0d: Producer: put(%0d) done %0d",
                  $time, i);
      end

      // Consumer
      repeat(3) begin
        int j;
        #1ns mbx.get(j);
        $display("@%0d: Consumer: got %0d", $time, j);
      end
    join_any
  end
endprogram
```

Example 7-27 creates the smallest mailbox which stores a single message. The Producer thread tries to put three messages (integers) in the mailbox, while the Consumer thread slowly gets messages every 1ns. As Example 7-28 shows, the first **put** succeeds, then the Producer tries **put(2)** which blocks. The Consumer wakes up, gets a message 1 from the mailbox, so now the Producer can finish putting the message 2.

Example 7-28 Output from bounded mailbox

```
@0: Producer: before put(1)
@0: Producer: put(1) done
@0: Producer: before put(2)
@1: Consumer: got 1
@1: Producer: put(2) done
@1: Producer: before put(3)
@2: Consumer: got 2
@2: Producer: put(3) done
```

The bounded mailbox acts as a buffer between the two processes. You can see how the Producer gets ahead of the Consumer.

7.6.3   Unsynchronized threads communicating with a mailbox

If you want the producer and consumer threads to run in lockstep, you need an additional handshake. In Example 7-29 the Producer and Consumer are now classes that exchange integers using a mailbox, with no explicit synchronization between the two objects.

Example 7-29 Producer–consumer without synchronization, part 1

```
program automatic unsynchronized;
  class Producer;
    task run;
      for (int i=1; i<4; i++) begin
        $display("Producer: before put(%0d)", i);
        mbx.put(i);
      end
    endtask
  endclass
```

Example 7-30 Producer–consumer without synchronization, continued

```
class Consumer;
  task run;
    int i;
    repeat (3) begin
      mbx.get(i);      // Get integer from mbx
      $display("Consumer: after  get(%0d)", i);
    end
  endtask
endclass

mailbox mbx;
Producer p;
Consumer c;

initial begin
  // Construct mailbox, producer, consumer
  mbx = new;
  p = new;
  c = new;

  // Run the producer and consumer in parallel
  fork
    p.run;
    c.run;
  join
end
endprogram
```

Example 7-31 has no synchronization so the Producer puts all three integers into the mailbox before the Consumer can get the first one. This is because a thread continues running until there is a blocking statement, and the Producer has none. The Consumer thread blocks on the first call to **mbx.get**.

Example 7-31 Producer–consumer without synchronization output

```
Producer: before put(1)
Producer: before put(2)
Producer: before put(3)
Consumer: after  get(1)
Consumer: after  get(2)
Consumer: after  get(3)
```

7.6.4 Synchronized threads using a mailbox and events

You may want the two threads to use a handshake so that the Producer does not get ahead of the Consumer. The Consumer already blocks, waiting for the Producer using a mailbox. The Producer needs to block, waiting for the Consumer to finish the transaction. This is done by adding a blocking statement to the Producer such as an event, a semaphore, or a second mailbox.

Example 7-32 uses an event to block the Producer after it puts data in the mailbox. The Consumer triggers the event after it consumes the data.

If you use **wait(handshake.triggered)** in a loop, be sure to advance the time before waiting again. This **wait** blocks only once in a given time slot, so you need move into another. Example 7-32 uses the edge-sensitive blocking statement **@handshake** instead to ensure that the Producer stops after sending the transaction. The edge-sensitive statement works multiple times in a time slot but may have ordering problems if the trigger and block happen in the same time slot.

Example 7-32 Producer–consumer synchronized with an event

```
program automatic mbx_evt;

  event handshake;

  class Producer;
    task run;
      for (int i=1; i<4; i++) begin
        $display("Producer: before put(%0d)", i);
        mbx.put(i);
        @handshake;
        $display("Producer: after  put(%0d)", i);
      end
    endtask
  endclass
```

Example 7-33 Producer–consumer synchronized with an event, continued

```
class Consumer;
  task run;
    int i;
    repeat (3) begin
      mbx.get(i);
      $display("Consumer: after  get(%0d)", i);
      ->handshake;
    end
  endtask
endclass
...
endprogram
```

Now the Producer does not advance until the Consumer triggers the event.

Example 7-34 Output from producer–consumer with event

```
Producer: before put(1)
Consumer: after  get(1)
Producer: after  put(1)
Producer: before put(2)
Consumer: after  get(2)
Producer: after  put(2)
Producer: before put(3)
Consumer: after  get(3)
Producer: after  put(3)
```

You can see that the Producer and Consumer are in lockstep.

7.6.5   Synchronized threads using two mailboxes

Another way to synchronize the two threads is to use a second mailbox that sends a completion message from the Consumer back to the Producer.

Example 7-35 Producer–consumer synchronized with a mailbox

```
program automatic mbx_mbx2;
  mailbox mbx, rtn;
  class Producer;
    task run;
      int k;
      for (int i=1; i<4; i++) begin
        $display("Producer: before put(%0d)", i);
        mbx.put(i);
        $display("Producer: after  put(%0d)", i);
        rtn.get(k);
        $display("Producer: after  get(%0d)", k);
      end
    endtask
  endclass

  class Consumer;
    task run;
      int i;
      repeat (3) begin
        $display("Consumer: before get");
        mbx.get(i);
        $display("Consumer: after  get(%0d)", i);
        rtn.put(-i);
      end
    endtask
  endclass
endprogram
```

The return message in the **rtn** mailbox is just a negative version of the original integer.

Example 7-36 Output from producer–consumer with mailbox

```
Producer: before put(1)
Producer: after  put(1)
Consumer: before get
Consumer: after  get(1)
Consumer: before get
Producer: after  get(-1)
Producer: before put(2)
Producer: after  put(2)
Consumer: after  get(2)
Consumer: before get
Producer: after  get(-2)
Producer: before put(3)
Producer: after  put(3)
Consumer: after  get(3)
Producer: after  get(-3)
```

7.6.6   Other synchronization techniques

You can also complete the handshake with blocking on a variable or a semaphore. An event is the simplest construct, followed by blocking on a variable. A semaphore is comparable to using a second mailbox, but no information is exchanged.

7.7   Building a Testbench with Threads and IPC

Now that you know how to use threads and IPC, you can construct a basic testbench with transactors.

7.7.1   Basic transactor

The following is the transactor for the agent that sits between the Generator and the Driver.

<u>**Example 7-37**</u> Basic Transactor

```
class Agent;

  mailbox gen2agt, agt2drv;
  Transaction tr;

  function new(mailbox gen2agt, agt2drv);
    this.gen2agt = gen2agt;
    this.agt2drv = agt2drv;
  endfunction

  function build;
  // Empty for now
  endfunction

  task run;
    forever begin
      // Get transaction from upstream block
      gen2agt.get(tr);

      // Do some processing

      // Send it to downstream block
      agt2drv.put(tr);
    end
  endfunction

  task wrapup;
  // Empty for now
  endtask

endclass
```

7.7.2   Environment class

The Generator, Agent, Driver, Monitor, Checker, and Scoreboard classes are instantiated in the Environment class.

Example 7-38 Environment class

```systemverilog
class Environment;

  Generator  gen;
  Agent      agt;
  Driver     drv;
  Monitor    mon;
  Checker    chk;
  Scoreboard scb;
  Config     cfg;
  mailbox gen2agt, agt2drv, mon2chk;

  extern function new;
  extern function void gen_cfg;
  extern function void build;
  extern task run;
  extern task wrapup;
endclass

function Environment::new;
  // Initialize mailboxes
  gen2agt = new;
  agt2drv = new;
  mon2chk = new;

  // Initialize transactors
  gen = new(gen2agt);
  agt = new(gen2agt, agt2drv);
  drv = new(agt2drv);
  mon = new(mon2chk);
  chk = new(mon2chk);
  scb = new;
  cfg = new;
endfunction

function void Environment::gen_cfg;
  assert(cfg.randomize);
endfunction

function void Environment::build;
  gen.build;
  agt.build;
  drv.build;
  mon.build;
```

```
    chk.build;
    scb.build;
endfunction

task Environment::run;
  fork
    gen.run(run_for_n_trans);
    agt.run;
    drv.run;
    mon.run;
    chk.run;
    scb.run(run_for_n_trans);
  join
endtask

task Environment::wrapup;
  fork
    gen.wrapup;
    agt.wrapup;
    drv.wrapup;
    mon.wrapup;
    chk.wrapup;
    scb.wrapup;
  join
endtask
```

7.7.3 Test program

The main test goes in the top-level program.

Example 7-39 Basic test program

```
program automatic test;

  Environment env;

  initial begin
    env = new;
    env.gen_cfg;
    env.build;
    env.run;
    env.wrapup;
  end
endprogram
```

7.8   Conclusion

Your design is modeled as many independent blocks running in parallel, so your testbench must also generate multiple stimulus streams and check the responses using parallel threads. These are organized into a layered testbench, orchestrated by the top-level environment. SystemVerilog introduces powerful constructs such as **fork**...**join_none** and **fork**...**join_any** for dynamically creating new threads, in addition to the standard **fork**...**join**. These threads communicate and synchronize using events, semaphore, mailboxes, and the classic **@** event control and **wait** statement. Lastly, the **disable** command is used to terminate threads.

These threads and the related control constructs complement the dynamic nature of OOP. As objects are created and destroyed, they can run in independent threads, allowing you to build a powerful and flexible testbench environment.

Chapter 8

Advanced OOP and Guidelines

8.1   Introduction

How would you create a complex class for a bus transaction that also includes performs error injection and variable delays? The first approach is to put everything in a large, flat class. This approach is simple to build, easy to understand (all the code is right there in one class) but can be slow to develop and debug. Additionally, such a large class is a maintenance burden, as anyone who wants to make a new transaction behavior has to edit the same file. Just as you would never create a complex RTL design using just one Verilog module, you should break classes down into smaller, reusable blocks.

The next choice is composition. As you learned in Chapter 4, you can instantiate one class inside another, just as you instantiate modules inside another, building up a hierarchical testbench. You write and debug your classes from the top down or bottom up. Look for natural partitions when deciding what variables and methods go into the various classes.

Sometimes it is difficult to divide the functionality into separate parts. Take the example of a bus transaction with error injection. When you write the original class for the transaction, you may not think of all the possible error cases. Ideally, you would like to make a class for a good transaction, and later add different error injectors. The transaction may have data fields and an error-checking CRC field generated from the data. One form of error injection is corruption of the CRC field. If you use composition, you need separate classes for a good transaction, and an error transaction. Testbench code that used good objects would have to be rewritten to process the new error objects. What you need is something that resembles the original class but adds a few new variables and methods. This result is accomplished through inheritance.

Inheritance allows a new class to be derived from an existing one in order to share its variables and routines. The original class is known as the base or super class, while the new one, since it extends the capability of the base class, is called the extended class. Inheritance provides reusability by adding features, such as error injection, to an existing class, the basic transaction, without modifying the base class.
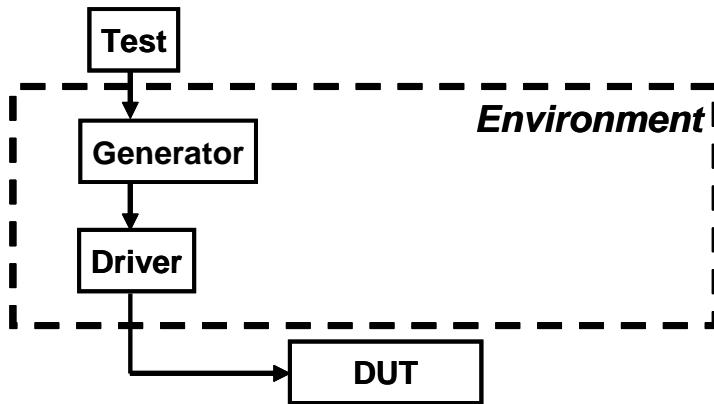
The real power of OOP is that it gives you the ability to take an existing class, such as a transaction, and selectively change parts of its behavior by replacing routines, but without having to change the surrounding infrastructure. With some planning, you can create a testbench solid enough

to send basic transactions, but able to accommodate any extensions needed by the test.

8.2   Introduction to Inheritance

Figure 8-1 shows a simple testbench. A generator creates a transaction, randomizes it, and sends it to the driver. The rest of the testbench is left out.

Figure 8-1 Simplified layered testbench

8.2.1   Base transaction

The base transaction class has variables for the source and destination addresses, eight data words, and a CRC for error checking, plus routines for displaying the contents and calculating the CRC. The **calc_crc** function is tagged as **virtual** so that it can be redefined if needed, as shown in the next section. Virtual routines are explained in more detail later in this chapter.

Example 8-1 Base `Transaction` class
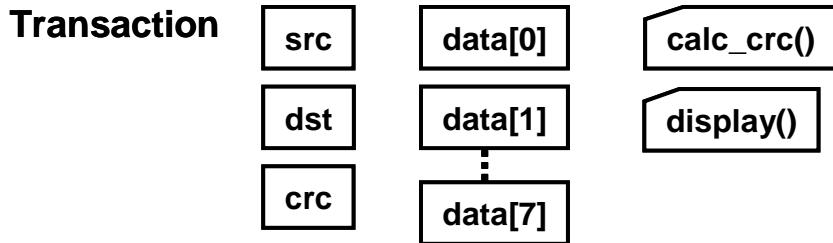
```
class Transaction;
  rand bit [31:0] src, dst, data[8];  // Variables
  bit [31:0] crc;

  virtual function void calc_crc;
    crc = src ^ dst ^ data.xor;
  endfunction

  virtual function void display;
    $display("Tr: src=%h, dst=%h, crc=%h", src,dst,crc);
  endfunction
endclass
```

A diagram for the class shows both the variables and routines.

Figure 8-2 Base Transaction class diagram

8.2.2   Extending the **Transaction** class

Suppose you have a testbench that sends good transactions through the DUT and now want to inject errors. Take an existing transaction class and extend it to create a new class. Following the guidelines from Chapter 1, you want to make as few code changes as possible to your existing testbench. So how can you reuse the existing **Transaction** class? This is done by declaring the new class, **BadTr**, as an extension of the current class. **Transaction** is called the base class, while **BadTr** is known as the extended class.

Example 8-2 Extended Transaction class

```
class BadTr extends Transaction;
  rand bit bad_crc;
  virtual function void calc_crc;
    super.calc_crc();          // Compute good CRC
    if (bad_crc) crc = ~crc;  // Corrupt the CRC bits
  endfunction

  virtual function void display;
    $write("BadTr: bad_crc:%b, ");
    super.display();
  endfunction

endclass : BadTr
```
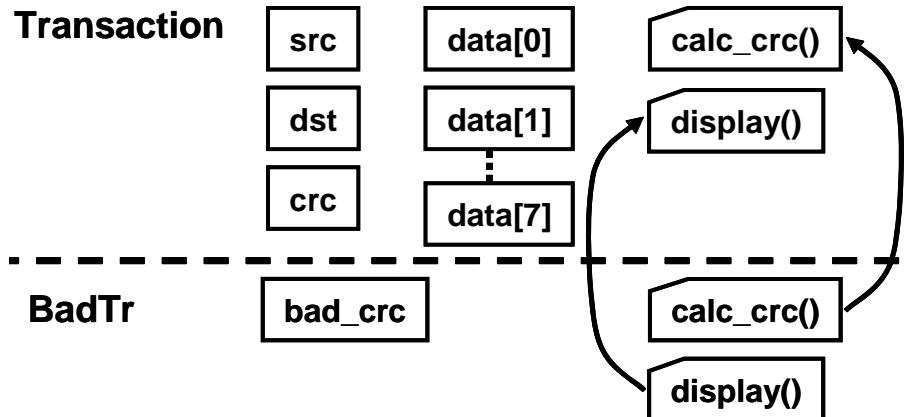
Note that in Example 8-2, the variable **crc** is used without a hierarchical identifier. The **BadTr** class can see all the variables from the original **Transaction** plus its own variables such as **bad_crc**. The **calc_crc** function in the extended class calls **calc_crc** in the base class using the **super** prefix. You can call one level up, but going across multiple levels such

as **super.super.new** is not allowed in SystemVerilog. Not to mention that this style, since it reaches across multiple levels, violates the rules of encapsulation by reaching across multiple boundaries.

Figure 8-3 Extended `Transaction` class diagram

Always declare routines inside a class as virtual so that they can be redefined in an extended class. This applies to all tasks and functions, except the **new** function, which is called when the object is constructed, so there is no way to extend it. SystemVerilog always calls the **new** function based on the handle's type.

8.2.3   Quick OOP glossary

Here is a quick glossary of terms. As explained in Chapter 4, the OOP term for a variable in a class is "property," and a task or function is called a "method." When you extend a class, the original class (such as **Transaction**) is called the parent class or super class. The extended class (**BadTr**) is known as the derived class or sub class. A base class is one that is not derived from any other class. The "prototype" for a routine is just the first line that shows the argument list and return type, if any. The prototype is used when you move the body of the routine outside the class, but is needed to describe how the routine communicated with others, as shown in section 4.11.

8.2.4   Constructors in extended classes

When you start extending classes, there is one rule about constructors (**new** function) to keep in mind. If your base class constructor has any arguments, the extend constructor must have a constructor and must call the base's constructor on its first line.

Example 8-3 Constructor with argument in an extended class

```
class Base1;
  int var;
  function new(int var); // Constructor with an argument
    this.var = var;
  endfunction
endclass

class Extended extends Base1;
  function new(int var);
    super.new(var);  // Must be first line of new
    // Other constructor actions
  endfunction
endclass
```

8.2.5   Driver class

The following driver class receives transactions from the generator and drives them into the DUT.

Example 8-4 Driver class

```
class Driver;
  mailbox gen2drv;

  function new(mailbox gen2drv);
    this.gen2drv = gen2drv;
  endfunction

  task main;
    Transaction tr;
    forever begin
      // Get transaction from upstream generator
      gen2drv.get(tr);

      // Process the transation
      tr.calc_crc;

      // Drive the interface signals to send transaction
      @ifc.cb.src = src;
      ...
    end
  endtask
endclass
```

This class stimulates the DUT with **Transaction** objects. OOP rules say that if you have a handle of the base type (**Transaction**), it can also point to an object of an extended type (**BadTr**). This is because the handle **tr** can only reference the variables **src**, **dst**, **crc**, and **data**, and the routine **calc_crc**. So you can send **BadTr** objects into the driver without changing it.

When the driver calls **tr.calc_crc**, SystemVerilog looks at the type of object stored in **tr**, because the task was declared as virtual. If the object is of type **Transaction**, SystemVerilog calls **Transaction::calc_crc**. If it is of type **BadTr**, SystemVerilog calls **BadTr::calc_crc**.

8.2.6   Simple generator class

The generator for this testbench creates a random transaction and puts it in the mailbox to the driver. The following example shows how you might create the class from what you have learned so far. Note that this avoids a very common testbench bug by constructing the transaction inside the loop instead of outside.

Example 8-5 Generator class

```
// Generator class that uses Transaction objects
// First attempt... too limited
class Generator;
  mailbox gen2drv;
  Transaction tr;

  function new(mailbox gen2drv);
    this.gen2drv = gen2drv;    // this-> class-level var
  endfunction

  function build;                // Empty for now
  endfunction

  task run;
    forever begin
      tr = new;                  // Construct transaction
      assert(tr.randomize);      // Randomize it
      gen2drv.put(tr);           // Send to driver
    end
  endtask

  task wrapup;                    // Empty for now
  endtask
endclass
```

There is a problem with this generator. The **run** task constructs a transaction and immediately randomizes it. This means that the transaction uses whatever constraints are turned on by default. The only way you can change these would be to edit the **Transaction** class, which goes against the verification guidelines. Worse yet, the generator only uses **Transaction** objects — there is no way to use an extended object such as **BadTr**. The fix is to separate the construction of **tr** from its randomization as shown below.

As you can see, the **Generator** and **Driver** classes have a common structure. You can enforce this by having both of them be extensions of a **Transactor** class, with virtual methods for **gen_cfg**, **build**, **run**, and **wrapup**. The VMM has an extensive set of base classes for transactors, data, the and much more.

8.3   Factory Patterns
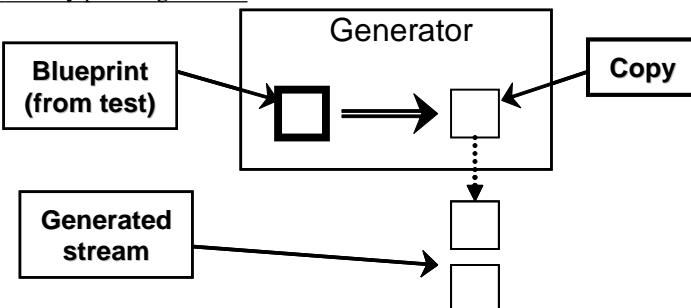
A useful OOP technique is the "factory pattern." If you build a factory to make signs, you don't need to know the shape of every possible sign in advance. You just need a stamping machine and then change the die to cut different shapes. Likewise, when you want to build a transactor generator, you don't have to know how to build every type of transaction; you just need to be able to stamp new ones that are similar to a given transaction.
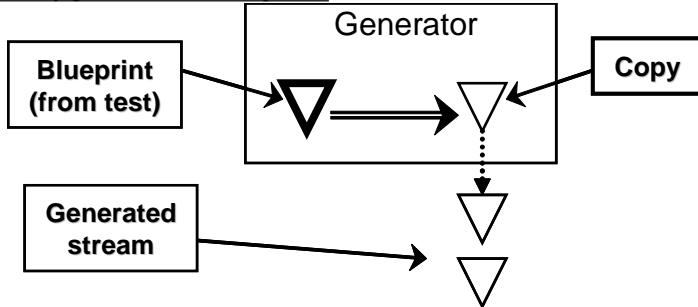
Instead of constructing and then immediately using an object, as in Example 8-5, instead construct a blueprint object (the cutting die), and then modify its constraints, or even replace it with an extended object. Now when you randomize this blueprint, it will have the random values that you want. Make a copy of this object and send the copy to the downstream transactor.

Figure 8-4 Factory pattern generator

The beauty of this technique is that if you change the blueprint object, your factory creates a different-type object. Using the sign analogy, you change the cutting die from a square to a triangle to make Yield signs.

Figure 8-5 Factory generator with new pattern

The blueprint is the "hook" that allows you to change the behavior of the generator class without having to change the class's code.

Here is the generator class using the factory pattern. The important thing to notice is that the blueprint object is constructed in one place (the **build** task) and used in another (the **run** task). Previous coding guidelines said to separate the declaration and construction; similarly, you need to separate the construction and randomization of the blueprint object.

Example 8-6 Generator class using factory pattern

```
class Generator;
  mailbox gen2drv;
  Transaction blueprint;

  function new(mailbox gen2drv);
    this.gen2drv = gen2drv;
  endfunction

  function build;
    blueprint = new;
  endfunction

  task run;
    Transaction tr;
    forever begin
      assert(blueprint.randomize);
      tr = blueprint.copy;      // * see below
      gen2drv.put(tr);          // Send to driver
    end
  endtask
endclass
```

The **copy** function is discussed again in section 8.6. For now, remember that you must add it to the **Transaction** and **BadTr** classes.

8.3.1 The `Environment` class

Chapter 1 discussed the three phases of execution: Build, Run, and Wrap-up. Example 8-7 shows the environment class that instantiates all the testbench components, and runs these three phases.

Example 8-7 Environment class

```
// Testbench environment class
class Environment;
  Generator gen;
  Driver drv;
  mailbox gen2drv;

  function new;
    gen = new(gen2drv);
    drv = new(gen2drv);
  endfunction

  function build;
    gen.build;
    drv.build;
  endfunction

  task run;
    fork
      gen.run;
      drv.run;
    join_none
  endtask

  task wrapup;
    gen.wrapup;
    drv.wrapup;
  endtask

endclass
```

8.3.2 A simple testbench

The test is contained in the top-level program. The basic test just lets the environment run with all the defaults.

Example 8-8 Simple test program using environment defaults

```
program automatic test;

  Environment env;
  initial begin
    env = new;              // Construct the environment
    env.build;              // Build testbench objects
    env.run;                // Run the test
    env.wrap_up;            // Clean up afterwards
  end
endprogram
```

8.3.3   Using the extended `Transaction` class

To inject an error, you need to change the blueprint object from a `Transaction` object to a `BadTr`. You do this between the build and run phases in the environment. The top-level testbench runs each phase of the environment and changes the blueprint. Note how all the references to `BadTr` are in this one file, so you don't have to change the `Environment` or `Generator` classes. You want to restrict the scope of where `BadTr` is used, so a standalone `begin`...`end` block is used in the middle of the `initial` block.

Example 8-9 Injecting extended transaction from test

```
program automatic test;

  Environment env;
  initial begin
    env = new;
    env.build;                  // Construct a blueprint

    begin
      BadTr bad;                // Create a bad transaction
      bad = new;                // Replace blueprint with
      env.drv.tr = bad;         // the "bad" one
    end

    env.run;                    // Run the test
    env.wrap_up;                // Clean up afterwards
  end
endprogram
```

8.4   Type Casting and Virtual Methods

As you start to use inheritance to extend the functionality of classes, you need a few OOP techniques to control the objects and their functionality. In particular, a handle can refer to an object for a certain class, or any extended class. So what happens when a base handle points to a extended object? What happens when you call a method that exists in both the base and extended classes? This section explains what happens using several examples.

8.4.1   Type casting with `$cast`

Type casting or conversion refers to changing an entity of one data type into another. Outside of OOP, you can convert an integer to a real or visa versa. In OOP, the easiest way to think about type casting is to consider a base class and an extended class.
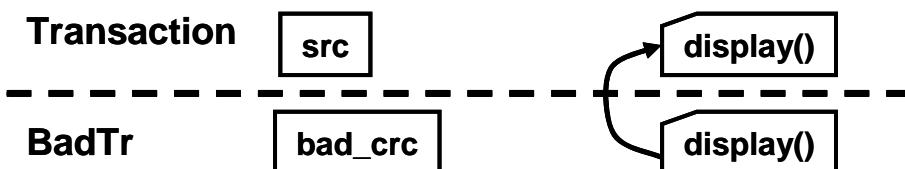
Example 8-10 Base and extended class

```
class Transaction;
  rand bit [31:0] src;
  virtual function void display;
    $display("Transaction: src=%0d", src);
  endfunction
endclass

class BadTr extends Transaction;
  bit bad_crc;
  virtual function void display;
    $display("BadTr: bad_crc=%0d", bad_crc);
  endfunction
endclass

Transaction tr;
BadTr bad, b2;
```

Figure 8-6 Simplified extended transaction

If you assign an extended handle to a base handle, nothing special is needed.

Example 8-11 Copying extended handle to base handle

```
bad = new;           // Construct BadTr extended object
tr = br;             // Base handle points to extended obj
$display(tr.src);    // Display base variable
tr.display;          // Calls BadTr::display
```

When a class is extended, all the variables and routines are inherited, so the integer **src** exists in the extended object. The assignment on the second line is permitted, as any reference using the base handle **tr** is valid, such as **tr.src** and **tr.display**.

But what if you try going in the opposite direction, copying a base object into an extended handle, as shown in Example 8-12? This fails because the base object is missing properties that only exist in the extended class, such as **bad_crc**. The SystemVerilog compiler does a static check of the handle types and will not compile the second line.

Example 8-12 Copying a base handle to an extended handle

```
tr = new;               // Construct base object
bad = tr;               // ERROR: WILL NOT COMPILE
$display(bad.bad_crc);  // bad_crc is not in base object
```

It is not always illegal to assign a base handle to an extended handle. It is allowed when the base handle actually points to an extended object. The **$cast** routine checks the type of object, not just the handle. Now you can copy the address of the extended object from the base handle, **tr**, into the extended handle, **b2**.

Example 8-13 Using $cast to copy handles

```
bad = new;               // Construct BadTr extended object
tr = bad;                // Base handle points to extended obj

// Check object type & copy. Simulation error if mismatch
$cast(b2, tr);

// User check for mismatch, no simulation error
if(!$cast(b2, tr))
  $display("cannot assign tr to b2");

$dislay(b2.bad_crc); // bad_crc exists in original obj
```

When you use **$cast** as a task, SystemVerilog checks the type of the source object at run-time and gives an error if it is not compatible. You can

eliminate this error by using **$cast** as a function and checking the result - 0 for incompatible types, and non-0 for compatible types.

8.4.2   Virtual methods

By now you should be comfortable using handles with extended classes. What happens if you try to call a routine using one of these handles?

Example 8-14 `Transaction` and `BadTr` classes

```
class Transaction;
  rand bit [31:0] src, dst, data[8];  // Variables
  bit [31:0] crc;

  virtual function void calc_crc;      // XOR all fields
    crc = src ^ dst ^ data.xor;
  endfunction
endclass : Transaction

class BadTr extends Transaction;
  rand bit bad_crc;
  virtual function void calc_crc;
    super.calc_crc();                  // Compute good CRC
    if (bad_crc) crc = ~crc;    // Corrupt the CRC bits
  endfunction
endclass : BadTr
```

Here is a block of code that uses handles of different types.

Example 8-15 Calling class methods

```
Transaction tr;
BadTr bad;

initial begin
  tr = new;
  tr.calc_crc;     // Calls Transaction::calc_crc

  bad = new;
  bad.calc_crc;    // Calls BadTr::calc_crc

  tr = bad;        // Base handle points to ext obj
  tr.calc_crc;     // Calls BadTr::calc_crc
end
```

When you use virtual methods, SystemVerilog uses the type of the object, not the handle to decide which routine to call. In the final statement, `tr` points to an extended object (`BadTr`) and so `BadTr::calc_crc` is called.

If you left out the `virtual` modifier on `calc_crc`, SystemVerilog would use the type of the handle (`Transaction`), not the object. That last statement would call `Transaction::calc_crc` – probably not what you wanted.

The OOP term for multiple routines sharing a common name is "polymorphism." It solves a problem similar to what computer architects faced when trying to make a processor that could address a large address space but had only a small amount of physical memory. They created the concept of virtual memory, where the code and data for a program could reside in memory or on a disk. At compile time, the program didn't know where its parts resided — that was all taken care of by the hardware plus operating system at run-time. A virtual address could be mapped to some RAM chips, or the swap file on the disk. Programmers no longer needed to worry about this virtual memory mapping when they wrote code — they just knew that the processor would find the code and data at run-time. See also Denning (2005).

8.4.3   Signatures

There is one downside to using virtual routines – once you define one, all extended classes that define the same virtual routine must use the same "signature," i.e., the same number and type of arguments. You cannot add or remove an argument in an extended virtual routine. This just means you need to plan ahead.

8.5   Composition, Inheritance, and Alternatives

As you build up your testbench, you have to decide how to group related variables and routines together into classes. In Chapter 4 you learned how to build basic classes and include one class inside another. Previously in this chapter, you saw the basics of inheritance. This section shows you how to decide between the two styles, and also shows an alternative.

8.5.1   Deciding between composition and inheritance

How should you tie together two related classes? Composition uses a "has-a" relationship. A packet has a header and a body. Inheritance uses an "is-a" relationship. A `BadTr` is a `Transaction`, just with more information. The following table is a quick guide, with more detail below.

Table 8-1. Comparing inheritance to composition

Question	Inheritance	Composition
1. Do you need to group multiple subclasses together? (SystemVerilog does not support multiple inheritance)	No	Yes
2. Does the higher-level class represent objects at a similar level of abstraction?	Yes	No
3. Is the lower-level information always present or required?	Yes	No
4. Does the additional data need to remain attached to the original class while it is being processed by pre-existing code?	Yes	No

1. Are there several small classes that you want to combine into a larger class? For example, you may have a data class and header class and now want to make a packet class. SystemVerilog does not support multiple inheritance, where one class derives from several classes at once. Instead you have to use composition. Alternatively, you could extend one of the classes to be the new class, and manually add the information from the others.

2. In Example 8-14, the **Transaction** and **BadTr** classes are both bus transactions that are created in a generator and driven into the DUT. Thus inheritance makes sense.

3. The lower-level information such as **src**, **dst**, and **data** must always be present for the Driver to send a transaction.

4. In Example 8-14, the new **BadTr** class has a new field **bad_crc** and the extended **calc_crc** function. The **Generator** class just transmits a transaction and does not care about the additional information. If you use composition to create the error bus transaction, the **Generator** class would have to be rewritten to handle the new type.

If two objects seem to be related by both "is-a" and "has-a," you may need to break them down into smaller components.

8.5.2   Problems with composition

The classical OOP approach to building a class hierarchy partitions functionality into small blocks that are easy to understand. But as discussed in

section 4.16 on public vs. private attributes, testbenches are not standard software development projects. Concepts such as information hiding (using private variables) conflict with building a testbench that needs maximum visibility and controllability. Similarly, dividing a transaction into smaller pieces may cause more problems than it solves.

When you are creating a class to represent a transaction, you may want to partition it to keep the code more manageable. For example, you may have an Ethernet MAC frame and your testbench uses two flavors, normal (**II**) and Virtual LAN (**VLAN**). Using composition, you could create a basic cell **EthMacFrame** with all the common fields such as **da** and **sa** and a discriminant variable, **kind,** to indicate the type. There is a second class to hold the VLAN information, which is included in **EthMacFrame**.

Example 8-16 Building an Ethernet frame with composition

```
// Not recommended
class EthMacFrame;
  typedef enum {II, IEEE} kind_t;
  rand kind_t kind;
  rand bit [47:0] da, sa;
  rand bit [15:0] len;

  ...
  rand Vlan vlan_h;
endclass

class Vlan;
  rand bit [15:0] vlan;
endclass
```

There are several problems with composition. First, it adds an extra layer of hierarchy, so you are constantly having to add an extra name to every reference. The VLAN information is called **eth_h.vlan_h.vlan**. If you start adding more layers, the hierarchical names become a burden.

A more subtle issue occurs when you want to instantiate and randomize the classes. What does the **EthMacFrame** constructor create? Because **kind** is random, you don't know whether to construct a **Vlan** object when **new** is called. When you randomize the class, the constraints set variables in both the **EthMacFrame** and **Vlan** objects based on the random **kind** field. However, randomization only works on objects that have been instantiated. But you can't instantiate this object until **kind** has been chosen.

The only solution to the construction and randomization problems is to always instantiate all object in **EthMacFrame::new**. But if you are always using all alternatives, why divide the Ethernet cell into two different classes?

8.5.3   Problems with inheritance

Inheritance can solve some of these issues. Variables in the extended classes can be referenced without the extra hierarchy as in **eth_h.vlan**. You don't need a discriminant, but you may find it easier to have one variable to test rather than doing type-checking.

**Example 8-17** Building an Ethernet frame with inheritance

```
// Not recommended
class EthMacFrame;
  typedef enum {II, IEEE} kind_t;
  rand kind_t kind;
  rand bit [47:0] da, sa;
  rand bit [15:0] len;
  ...
endclass

class Vlan extends EthMacFrame;
  rand bit [15:0] vlan;
endclass
```
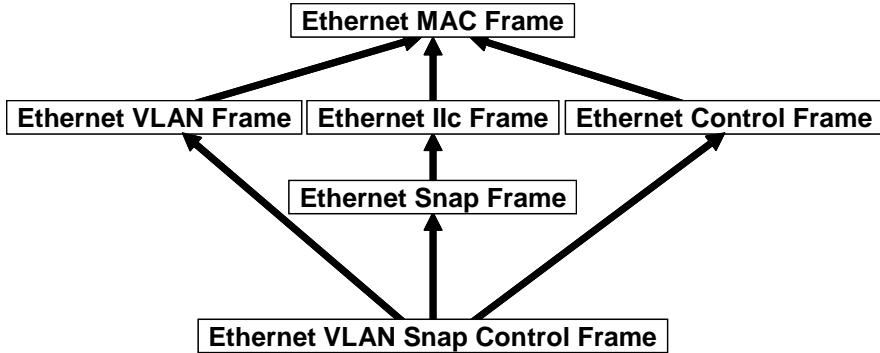
On the downside, a set of classes that use inheritance always requires more effort to design, build, and debug than a set of classes without inheritance. Your code must use **$cast** whenever you have an assignment from a base handle to an extended. Building a set of virtual routines can be challenging, as they all have to have the same prototype. If you need an extra argument, you need to go back and edit the entire set, and possibly the routine calls too.

There are also problems with randomization. How do you make a constraint that randomly chooses between the two kinds of frame and sets the proper variables? You can't put a constraint in **EthMacFrame** that references the **vlan** field.

The final issue is with multiple inheritance. In Figure 8-7, you can see how the VLAN frame is derived from normal MAC frame. The problem is that these different standards reconverged. SystemVerilog does not support multiple inheritance, so you could not create the VLAN / Snap / Control frame through inheritance.

Figure 8-7 Multiple inheritance problem

8.5.4 A real-world alternative

If composition leads to large hierarchies, but inheritance requires extra code and planning to deal with all the different classes, and both have difficult construction and randomization, what can you do? You can instead make a single, flat class that has all the variables and routines. This approach leads to a very large class, but it handles all the variants cleanly. You have to use the discriminant variable often to tell which variables are valid, as shown in Example 8-18. It contains several conditional constraints, which apply in different cases, depending on the value of **kind**.

Example 8-18 Building a flat Ethernet frame

```
class eth_mac_frame;
  typedef enum {II, IEEE} kind_t;
  rand kind_t kind;
  rand bit [47:0] da, sa;
  rand bit [15:0] len, vlan;
  ...
  constraint eth_mac_frame_II {
    if (kind == II) {
      data.size() inside {[46:1500]};
      len == data.size();
  }}
  constraint eth_mac_frame_ieee {
    if (kind == IEEE) {
      data.size() inside {[46:1500]};
      len < 1522;
  }}
endclass
```

Define the typical behavior and constraints in the class, and then use inheritance to inject new behavior at the test level.

8.6   Copying an Object

In Example 8-6, the generator first randomized, and then copied the blueprint to make a new transaction. Take a closer look at your **copy** function.

Example 8-19 Base transaction class with a virtual `copy` function

```
class Transaction;
  rand bit [31:0] src, dst, data[8];  // Variables
  bit [31:0] crc;

  virtual function Transaction copy;
    copy = new;
    copy.src = src;  // Copy data fields
    copy.dst = dst;
    copy.data = data;
    copy.crc = crc;
  endfunction
endclass
```

When you extend the **Transaction** class to make the class **BadTr**, the **copy** function still has to return a **Transaction** object. This is because the extended virtual function must match the base **Transaction::copy**, including all arguments and return type.

Example 8-20 Extended transaction class with virtual `copy` method

```
class BadTr extends Transaction;
  rand bit bad_crc;

  virtual function Transaction copy;
    BadTr bad;
    bad = new;
    bad.src = src;  // Copy data fields
    bad.dst = dst;
    bad.data = data;
    bad.crc = crc;
    bad.bad_crc = bad_crc;
    return bad;
  endfunction

endclass : BadTr
```

8.6.1   The copy_data routine

One optimization is to break the **copy** function in two, creating a separate function, **copy_data**. Now each class is responsible for copying its local data. This makes the **copy** function more robust and reusable. Here is the function for the base class.

Example 8-21 Base transaction class with `copy_data` function

```
class Transaction;
  rand bit [31:0] src, dst, data[8];  // Variables
  bit [31:0] crc;

  virtual function void copy_data(Transaction copy);
    copy.src  = src;    // Copy the data fields
    copy.dst  = dst;
    copy.data = data;
    copy.crc  = crc;
  endfunction

  virtual function Transaction copy;
    copy = new;
    copy_data(copy);
  endfunction
endclass
```

In the extended class, **copy_data** is a little more complicated. Because it extends the original **copy_data** routine, its argument is a **Transaction** handle. The routine can use this when calling **Transaction::copy_data** but when it needs to copy **bad_crc**, it needs a **BadTr** handle, so it has to first cast the base handle to the extended type.

Example 8-22 Extended transaction class with copy_data function

```
class BadTr extends Transaction;
  rand bit bad_crc;

  virtual function void copy_data(Transaction tr);
    BadTr bad;
    super.copy_data(tr);      // Copy base data
    $cast(bad, tr);           // Cast base handle to ext'd
    bad.bad_crc = bad_crc;    // Copy extended data
  endfunction

  virtual function Transaction copy;
    BadTr bad;
    bad = new;                // Construct BadTr
    copy_data(bad);           // Copy data fields
    return bad;
  endfunction

endclass : BadTr
```

8.6.2   Specifying a destination for copy

The existing **copy** routine always constructs a new object. An improvement for **copy** is to specify the location where the copy should be put. This technique is useful when you want to reuse an existing object, and not allocate a new one.

Example 8-23 Base transaction class with copy_data function

```
class Transaction;

  virtual function Transaction copy(Transaction to=null);
    if (to == null) copy = new;  // Construct object
    else            copy = to;   // Use exisiting
    copy_data(copy);
  endfunction
endclass
```

The only difference is the additional argument to specify the destination, and the code to test that a destination object was passed to this routine. If nothing was passed (the default), construct a new object, or else use the existing one.
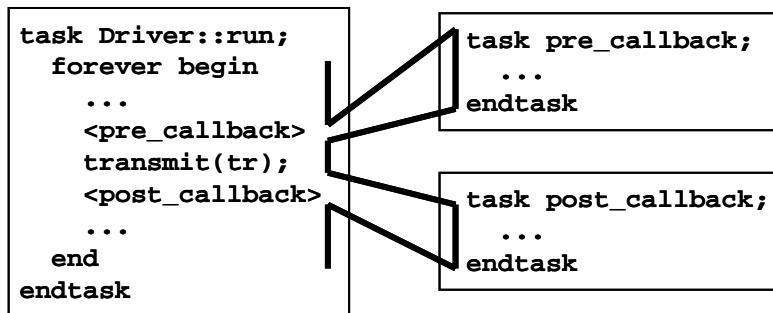
8.7 Callbacks

One of the main guidelines of this book is to create a verification environment that you can use for all tests with no changes. The key requirement is that this testbench must provide a "hook" where the test program can inject new code without modifying the original classes. Your driver may want to do the following.

- Inject errors
- Drop the transaction
- Delay the transaction
- Synchronize this transaction with others
- Put the transaction in the scoreboard
- Gather functional coverage data

Rather than try to anticipate every possible error, delay, or disturbance in the flow of transactions, the driver just needs to "call back" a routine defined in the top-level test. The beauty of this technique is that the callback[11] routine can be defined differently in every test. As a result, the test can add new functionality to the driver using callbacks without editing the **Driver** class.

Figure 8-8 Callback flow

[11.] This OOP-based callback technique is not related to Verilog PLI callbacks or SVA callbacks.

In Figure 8-8, the `Driver::run` task loops forever with a call to a `transmit` task. Before sending the transaction, `run` calls the pre-transmit callback. After sending the transaction, it calls the post-callback task.

8.7.1   Creating a callback

A callback task is created in the top-level test and called from the driver, the lowest level of the environment. However, the driver does not have to have any knowledge of the test – it just has to use a generic class that the test can extend. The driver uses a queue to hold the callback objects, which simplifies adding new objects.

Example 8-24 Base callback class

```
class Driver_cbs;  // Driver callbacks
  virtual task pre_tx(Transaction tr, ref bit drop);
    // By default, callback does nothing
  endtask

  virtual task post_tx(Transaction tr);
    // By default, callback does nothing
  endtask
endclass
```

Example 8-25 Driver class with callbacks

```
class Driver;
  Driver_cbs cbs[$];

  task run;
    bit drop;
    Transaction tr;

    forever begin
      agt2drv.get(tr);
      foreach (cbs[i]) cbs.pre_tx(tr, drop);

      if (!drop) transmit(tr);

      foreach (cbs[i]) cbs.post_tx(tr);
    end
  endtask
endclass
```

8.7.2   Using a callback to inject disturbances

A common use for a callback is to inject some disturbance such as causing an error or delay. The following testbench randomly drops packets using a callback object. Callbacks can also be used to send data to the scoreboard or to gather functional coverage values. Note that you can use **push_back()** or **push_front()** depending on the order in which you want these to be called. For example, you probably want the scoreboard called after any tasks that may delay, corrupt, or drop a transaction. Only gather coverage after a transaction has been successfully transmitted.

Example 8-26 Test using a callback for error injection

```
class Driver_cbs_drop extends Driver_cbs;

  virtual task pre_tx(Transaction tr, ref bit drop);
    // Randomly drop 1 out of every 100 transactions
    drop = ($urandom_range(0,99) == 0);
  endtask

endclass


program automatic test;

  Environment env;

  initial begin
    env = new;
    env.gen_cfg;
    env.build;

    // Callback injection
    begin
    Driver_cbs_drop dcd;
      dcd = new;                   // Create scb callback
      env.drv.cbs.push_back(dcd); // Put into driver's Q
    end

    env.run;
    env.wrapup;
  end

endprogram
```

8.7.3   Connecting to the scoreboard with a callback

The following testbench creates its own extension of the driver's callback class and adds a reference to the driver's callback queue.

Example 8-27 Test using callback for scoreboard

```
class Driver_cbs_scoreboard extends Driver_cbs;

  virtual task pre_tx(Transaction tr, ref bit drop);
    // Put transaction in the scoreboard
    ...
  endtask

endclass


program automatic test;

  Environment env;

  initial begin
    env = new;
    env.gen_cfg;
    env.build;

    begin
      Driver_cbs_scoreboard dcs;
      dcs = new;                      // Create scb callback
      env.drv.cbs.push_back(dcs); // Put into driver's Q
    end

    env.run;
    env.wrapup;
  end

endprogram
```

Always use callbacks for scoreboards and functional coverage. The monitor transactor can use a callback to compare received transactions with expected ones. The monitor callback is also the perfect place to gather functional coverage on transactions that are actually sent by the DUT.

You may have thought of using a separate transactor for functional coverage that is connected to the testbench using a mailbox. This is a poor solution for both coverage and scoreboards, as these are passive testbench components that only wake up when the testbench has data for them, and they

never pass information to a downstream transactor. Additionally, you may sample data from several points in your testbench, but a transactor is designed for a single source. Instead, make the scoreboard and coverage interface passive, and the rest of the testbench can access them with callbacks.

8.7.4   Using a callback to debug a transactor

If a transactor with callbacks is not working as you expect, you can use an additional callback to debug it. You can start by adding a callback to display the transaction. If there are multiple instances, display the hierarchical path, using `$display("%m ...")`. Then move the debug before and after the other callbacks to locate the one that is causing the problem. Even for debug, you want to avoid making changes to the testbench environment.

8.8   Conclusion

The software concept of inheritance, where new functionality is added to an existing class, parallels the hardware practice of extending the design's features for each generation, while still maintaining backwards compatibility.

For example, you can upgrade your PC by adding a larger capacity disk. As long as it uses the same interface as the old one, you do not have to replace any other part of the system, yet the overall functionality is improved.

Likewise, you can create a new test by "upgrading" the existing driver class to inject errors. If you use an existing callback in the driver, you do not have to change any of the testbench infrastructure.

You need to plan ahead if you want use these OOP techniques. By using virtual routines and providing sufficient callback points, your test can modify the behavior of the testbench without changing its code. The result is a robust testbench that does not need to anticipate every type of disturbance (error-injection, delays, synchronization) that you may want as long as you leave a hook where the test can inject its own behavior. The tests become smaller and easier to write as the testbench does the hard work of sending stimulus and checking responses, so the test only has to make small tweaks to cause specialized behavior.

Chapter 9
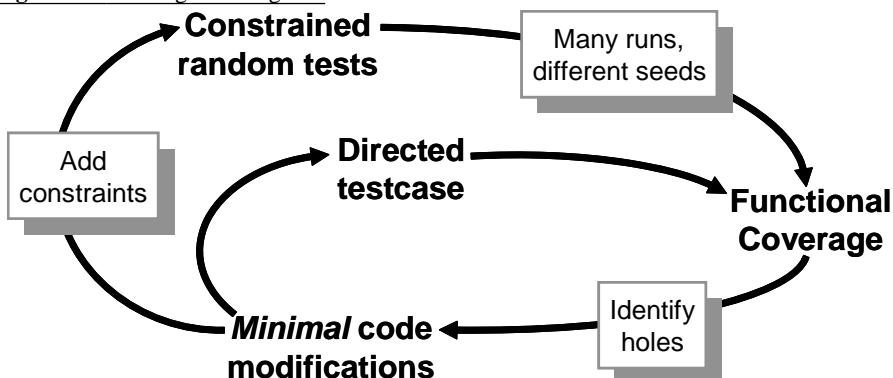
Functional Coverage

9.1   Introduction

As designs become more complex, the only effective way to verify them thoroughly is with constrained-random testing (CRT). This approach elevates you above the tedium of writing individual directed tests, one for each feature in the design. However, if your testbench is taking a random walk through the space of all design states, how do you know if you have reached your destination? Whether you are using random or directed stimulus, you can gauge progress using coverage.

Functional coverage is a measure of which design features have been exercised by the tests. Start with the design specification and create a verification plan with a detailed list of what to test and how. For example, if your design connects to a bus, your tests need to exercise all the possible interactions between the design and bus, including relevant design states, delays, and error modes. The verification plan is a map to show you where to go. For more information on creating a verification plan, see Bergeron (2006).

Figure 9-1 Coverage convergence

Use a feedback loop to analyze the coverage results and decide on which actions to take in order to converge on 100% coverage. Your first choice is to run existing tests with more seeds; the second is to build new constraints. Only resort to creating directed tests if absolutely necessary.

Back when you exclusively wrote directed tests, the verification planning was limited. If the design specification listed 100 features, all you had to do was write 100 tests. Coverage was implicit in the tests — the "register move" test moved all combinations of registers back and forth. Measuring progress
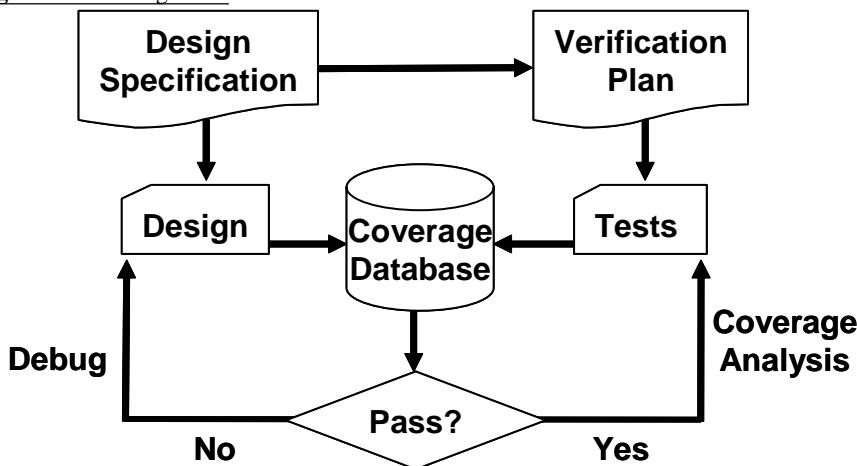
was easy: if you had completed 50 tests, you were halfway done. This chapter uses "explicit" and "implicit" to describe how coverage is specified. Explicit coverage is described directly in the test environment using SystemVerilog features. Implicit coverage is implied by a test — when the "register move" directed test passes, you have hopefully covered all register transactions.

With CRT, you are freed from hand crafting every line of input stimulus, but now you need to write code that tracks the effectiveness of the test with respect to the verification plan. You are still more productive, as you are working at a higher level of abstraction. You have moved from tweaking individual bits to describing the interesting design states. Reaching for 100% functional coverage forces you to think more about what you want to observe and how you can direct the design into those states.

9.1.1   Gathering coverage data

You can run the same random testbench over and over, simply by changing the random seed, to generate new stimulus. Each individual simulation generates a database of functional coverage information, the trail of footprints from the random walk. You can then merge all this information together to measure your overall progress using functional coverage.

Figure 9-2 Coverage flow

You then analyze the coverage data to decide how to modify your tests. If the coverage levels are steadily growing, you may just need to run existing tests with new random seeds, or even just run longer tests. If the coverage growth has started to slow, you can add additional constraints to generate more "interesting" stimuli. When you reach a plateau, some parts of the design are not being exercised, so you need to create more tests. Lastly, when your functional coverage values near 100%, check the bug rate. If bugs are

still being found, you may not be measuring true coverage for some areas of your design.

Each simulation vendor has its own format for storing coverage data and as well as its own analysis tools. You need to perform the following actions with those tools.

– Run a test with multiple seeds. For a given set of constraints (and coverage groups), compile the testbench and design into a single executeable. Now you need to run this constraint set over and over with different random seeds. You can use the Unix system clock as a seed, but be careful, as your batch system may start multiple jobs simultaneously. These jobs may run on different servers or may start on a single server with multiple processors.

– Check for pass/fail. Functional coverage information is only valid for a successful simulation. When a simulation fails because there is a design bug, the coverage information must be discarded. The coverage data measures how many items in the verification plan are complete, and this plan is based on the design specification. If the design does not match the specification, the coverage data is useless. Some verification teams periodically measure all functional coverage from scratch so that it reflects the current state of the design.

– Analyze coverage across multiple runs. You need to measure how successful each constraint set is, over time. If you are not yet getting 100% coverage for the areas that are targeted by the constraints, but the amount is still growing, run more seeds. If the coverage level has plateaued, with no recent progress, it is time to modify the constraints. Only if you think that reaching the last few test cases for one particular section may take too long for constrained-random simulation should you consider writing a directed test. Even then, continue to use random stimulus for the other sections of the design, in case this "background noise" finds a bug.

9.2   Coverage Types

Coverage is a generic term for measuring progress to complete design verification. Your simulations slowly paint the canvas of the design, as you try to cover all of the legal combinations. The coverage tools gather information during a simulation and then post-process it to produce a coverage report. You can use this report to look for coverage holes and then modify existing tests or create new ones to fill the holes. This iterative process continues until you are satisfied with the coverage level.

9.2.1   Code coverage

The easiest way to measure verification progress is with code coverage. Here you are measuring how many lines of code have been executed (line coverage), which paths through the code and expressions have been executed (path coverage), which single-bit variables have had the values 0 or 1 (toggle coverage), and which states and transitions in a state machine have been visited (FSM coverage). You don't have to write any extra HDL code. The tool instruments your design automatically by analyzing the source code and adding hidden code to gather statistics. You then run all your tests, and the code coverage tool creates a database.

Many simulators include a code coverage tool. A post-processing tool converts the database into a readable form. The end result is a measure of how much your tests exercise the design code. Note that you are primarily concerned with analyzing the design code, not the testbench. Untested design code could conceal a hardware bug, or may be just redundant code.

Code coverage measures how thoroughly your tests exercised the "implementation" of the design specification, and not the verification plan. Just because your tests have reached 100% code coverage, your job is not done. What if you made a mistake that your test didn't catch? Worse yet, what if your implementation is missing a feature? The following module is for a D-flip flop. Can you see the mistake?

Example 9-1 Incomplete D-flip flop model missing a path

```
module dff(output logic q, q_l,
           input  logic clk, d, reset_l);

  always @(posedge clk or negedge reset_l) begin
    q <= d;
    q_l <= !d;
  end
endmodule
```

The reset logic was accidently left out. A code coverage tool would report that every line had been exercised, yet the model was not implemented correctly.

9.2.2   Functional coverage

The goal of verification is to ensure that a design behaves correctly in its real environment, be that an MP3 player, network router, or cell phone. The design specification details how the device should operate, while the verification plan lists how that functionality is to be stimulated, verified, and measured. When you gather measurements on what functions were covered,
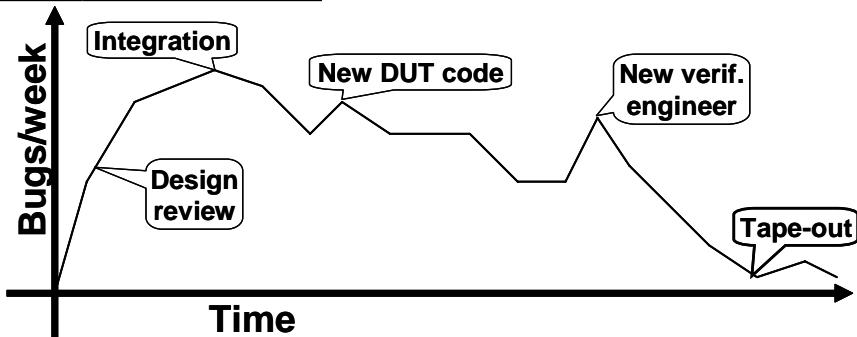
you are performing "design" coverage. For example, the verification plan for a D-flip flop would mention not only its data storage but also how it resets to a known state. Until your test checks both these design features, you will not have 100% functional coverage.

Functional coverage is tied to the design intent and is sometimes called "specification coverage," while code coverage measures the design implementation. Consider what happens if a block of code is missing from the design. Code coverage cannot catch this mistake, but functional coverage can.

9.2.3   Bug rate

An indirect way to measure coverage is to look at the rate at which fresh bugs are found. You should keep track of how many bugs you found each week, over the life of a project. At the start, you may find many bugs through inspection as you create the testbench. As you read the design spec, you may find inconsistencies, which hopefully are fixed before the RTL is written. Once the testbench is up and running, a torrent of bugs is found as you check each module in the system. The bug rate drops, hopefully to zero, as the design nears tape-out. However, just because the bug rates approaches zero, you are not yet done. Every time the rate sags, it is time to find different ways to create corner cases.

Figure 9-3 Bug rate during a project

The bug rate can vary per week based on many factors such as project phases, recent design changes, blocks being integrated, personnel changes, and even vacation schedules. Unexpected changes in the rate could signal a potential problem. As shown in Figure 9-3, it is not uncommon to keep finding bugs even after tape-out, and even after the design ships to customers.

9.2.4   Assertion Coverage

Assertions are pieces of declarative code that check the relationships between design signals, either once or over a period of time. These can be

simulated along with the design and testbench, or proven by formal tools. Sometimes you can write the equivalent check using SystemVerilog procedural code, but many assertions are more easily expressed using SystemVerilog Assertions (SVA).

Assertions can have local variables and perform simple data checking. If you need to check a more complex protocol, such as determining whether a packet successfully went through a router, procedural code is often better suited for the job. There is a large overlap between sequences that are coded procedurally or using SVA. See Vijayaraghavan (2005), Cohen (2005), and Chapters 3 and 7 in the VMM book, Bergeron et al. (2006) for more information on SVA.

The most familiar assertions look for errors such as two signals that should be mutually exclusive or a request that was never followed by a grant. These error checks should stop the simulation as soon as they detect a problem. Assertions can also check arbitration algorithms, FIFOs, and other hardware. These are coded with the **assert property** statement.

Some assertions might look for interesting signal values or design states, such as a successful bus transaction. These are coded with the **cover property** statement. You can measure how often these assertions are triggered during a test by using assertion coverage. A **cover property** observes sequences of signals, while a cover group (described below) samples data values and transactions during the simulation. These two constructs overlap in that a cover group can trigger when a sequence completes. Additionally, a sequence can collect information that can be used by a cover group.

9.3   Functional Coverage Strategies

Before you write the first line of test code, you need to anticipate what are the key design features, corner cases, and possible failure modes. This is how you write your verification plan. Don't think in terms of data values only; instead, think about what information is encoded in the design. The plan should spell out the significant design states.

9.3.1   Gather information, not data

A classic example is a FIFO. How can you be sure you have thoroughly tested a 1K FIFO memory? You could measure the values in the read and write indices, but there are over a million possible combinations. Even if you were able to simulate that many cycles, you would not want to read the coverage report.

At a more abstract level, a FIFO can hold from 0 to N-1 possible values. So what if you just compare the read and write indices to measure how full or

empty the FIFO is? You would still have 1K coverage values. If your testbench pushed 100 entries into the FIFO, then pushed in 100 more, do you really need to know if the FIFO ever had 150 values? Not as long as you can successfully read out all values.

The corner cases for a FIFO are Full and Empty. If you can make the FIFO go from Empty (the state after reset) through Full and back down to Empty, you have covered all the levels in between. Other interesting states involve the indices as they pass between all 1's and all 0's. A coverage report for these cases is easy to understand.

You may have noticed that the interesting states are independent of the FIFO size. Once again, look at the information, not the data values.

Design signals with a large range (more than a few dozen possible values) should be broken down into smaller ranges, plus corner cases. For example, your DUT may have a 32-bit address bus, but you certainly don't need to collect 4 billion samples. Check for natural divisions such as memory and IO space. For a counter, pick a few interesting values, and always try to rollover counter values from all 1's back to 0.

9.3.2   Only measure what you are going to use

Gathering functional coverage data can be expensive, so only measure what you will analyze and use to improve your tests. Your simulations may run slower as the simulator monitors signals for functional coverage, but this approach has lower overhead than gathering waveform traces and measuring code coverage. Once a simulation completes, the database is saved to disk. With multiple testcases and multiple seeds, you can fill disk drives with functional coverage information. But if you never look at the final coverage reports, don't perform the initial measurements.

There are several ways to control cover data: at compilation, instantiation, or triggering. You could use switches provided by the simulation vendor, conditional compilation, or suppression of the gathering of coverage data. The last of these is less desirable because the post-processing report is filled with sections with 0% coverage, making it harder to find the few enabled ones.
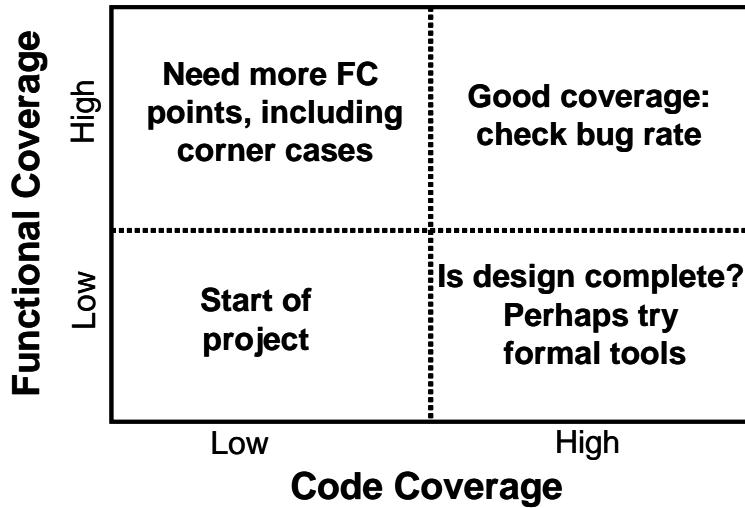
9.3.3   Measuring completeness

Like your kids in the backseat on a family vacation, your manager constantly asks you, "Are we there yet?" How can you tell if you have fully tested a design? You need to look at all coverage measurements and consider the bug rate to see if you have reached your destination.

At the start of a project, both code and functional coverage are low. As you develop tests, run them over and over with different random seeds until you no longer see increasing values of functional coverage. Create additional

constraints and tests to explore new areas. Save test/seed combinations that give high coverage, so that you can use them in regression testing.

Figure 9-4 Coverage comparison

What if the functional coverage is high but the code coverage is low? Your tests are not exercising the full design, and you need to revise your verification plan and add more functional coverage points to locate untested functionality.

A more difficult situation is high code coverage but low functional coverage. Even though your testbench is giving the design a good workout, you are unable to put it in all the interesting states. First, see if the design implements all the specified functionality. If the functionality is there, but your tests can't reach it, you might need a formal verification tool that can extract the design's states and create appropriate stimulus.

The goal is both high code and functional coverage. But don't plan your vacation yet. What is the trend of the bug rate? Are significant bugs still popping up? Worse yet, are they being found deliberately, or did your testbench happen to stumble across a particular combination of states that no one had anticipated? On the other hand, a low bug rate may mean that your existing strategies have run out of steam, and you should look into different approaches. Try different approaches such as new combinations of design blocks and error generators.

9.4   Simple Functional Coverage Example

To measure functional coverage, you begin with the verification plan and write an executable version of it for simulation. In your SystemVerilog test-

bench, sample the values of variables and expressions. These sample locations are known as cover points. Multiple cover points that are sampled at the same time (such as when a transaction completes) are place together in a cover group.

The following design has a transaction that comes in eight flavors. The testbench generates the **port** variable randomly, and the verification plan requires that every value be tried.

Example 9-2 Functional coverage of a simple object

```
program automatic test(busifc.TB ifc);

  class Transaction;
    rand bit [31:0] data;
    rand bit [ 2:0] port;      // Eight port numbers
  endclass

  covergroup CovPort;
    coverpoint tr.port;        // Measure coverage
  endgroup

  Transaction tr = new;

  initial begin
    CovPort ck = new;          // Instantiate group

    repeat (32) begin          // Run a few cycles
      assert(tr.randomize);    // Create a transaction
      ifc.cb.port <= tr.port;  //   and transmit
      ifc.cb.data <= tr.data;  //   onto interface
      ck.sample();             // Gather coverage
      @ifc.cb;                 // Wait a cycle
    end
  end
endprogram
```

Example 9-2 creates a random transaction and drives it out to an interface. The testbench samples the value of the **port** field using the **CovPort** cover group. Eight possible values, 32 random transactions — did your testbench generate them all? Here is part of a coverage report from VCS.

Example 9-3 Coverage report for a simple object

```
Coverpoint Coverage report
CoverageGroup: CovPort
    Coverpoint: tr.port
Summary
    Coverage: 87.50
    Goal: 100
    Number of Expected auto-bins: 8
    Number of User Defined Bins: 0
    Number of Automatically Generated Bins: 7
    Number of User Defined Transitions: 0

    Automatically Generated Bins

    Bin              # hits    at least
    ===============================
    auto[1]           7          1
    auto[2]           7          1
    auto[3]           1          1
    auto[4]           5          1
    auto[5]           4          1
    auto[6]           2          1
    auto[7]           6          1
    ===============================
```

As you can see, the testbench generated the values 1, 2, 3, 4, 5, 6, and 7, but never generated a **port** of 0. The **at least** column specifies how many hits are needed before a bin is considered covered. See section 9.9.3 for the **at_least** option.

To improve your functional coverage, the easiest strategy is to just run more simulation cycles, or to try new random seeds. Look at the coverage report for items with two or more hits. Chances are that you just need to make the simulation run longer or to try new seed values. If a cover point had zero or one hit, you probably have to try a new strategy, as the testbench is not creating the proper stimulus. For this example, the very next random transaction (#33) has a **port** value of 0, giving 100% coverage.

Example 9-4 Coverage report for a simple object, 100% coverage

```
Coverpoint Coverage report
CoverageGroup: CovPort
    Coverpoint: tr.port
Summary
    Coverage: 100
    Goal: 100
    Number of Expected auto-bins: 8
    Number of User Defined Bins: 0
    Number of Automatically Generated Bins: 8
    Number of User Defined Transitions: 0

    Automatically Generated Bins

    Bin             # hits     at least
    ===============================
    auto[0]         1          1
    auto[1]         7          1
    auto[2]         7          1
    auto[3]         1          1
    auto[4]         5          1
    auto[5]         4          1
    auto[6]         2          1
    auto[7]         6          1
    ===============================
```

9.5 Anatomy of a Cover Group

A cover group is similar to a class — you define it once and then instanti-ate it one or more times. It contains cover points, options, formal arguments, and an optional trigger. A cover group encompasses one or more data points, all of which are sampled at the same time.

You should create very clear cover group names that explicitly indicate what you are measuring and, if possible, reference to the verification plan. The name **Parity_Errors_In_Hexaword_Cache_Fills** may seem ver-bose, but when you are trying to read a coverage report that has dozens of cover groups, you will appreciate the extra detail. You can also use the com-ment option for additional descriptive information, as shown in section 9.9.1.

A cover group can be defined in a class or at the program or module level. It can sample any visible variable such as program/module variables, signals from an interface, or any signal in the design (using a hierarchical reference).

A cover group inside a class can sample variables in that class, as well as data values from embedded classes.

Don't define the cover group in a data class, such as a transaction, as doing so can cause additional overhead when gathering coverage data. Imagine you are trying to track how many beers were consumed by patrons in a pub. Would you try to follow every bottle as it flowed from the loading dock, over the bar, and into each person? No, instead you could just have each patron check off the type and number of beers consumed, as shown in van der Schoot (2006).

In SystemVerilog, you should define cover groups at the appropriate level of abstraction. This level can be at the boundary between your testbench and the design, in the transactors that read and write data, in the environment configuration class, or wherever is needed. The sampling of any transaction must wait until it is actually received by the DUT. If you inject an error in the middle of a transaction, causing it to be aborted in transmission, you need to change how you treat it for functional coverage. You need to use a different cover point that has been created just for error handling.

A class can contain multiple cover groups. This approach allows you to have separate groups that can be enabled and disabled as needed. Additionally, each group may have a separate trigger, allowing you to gather data from many sources.

A cover group must be instantiated for it to collect data. If you forget, no error message about null handles is printed at run-time, but the coverage report will not contain any mention of the cover group. This rule applies for cover groups defined either inside or outside of classes.

9.5.1   Defining a cover group in a class

A cover group can be defined in a program, module, or class. In all cases, you must explicitly instantiate it to start sampling. If the cover group is defined in a class, you do not make a separate name when you instance it; you just use the original cover group name.

Example 9-5 is very similar to the first example of this chapter except that it embeds a cover group in a transactor class, and thus does not need a separate instance name.

Example 9-5 Functional coverage inside a class

```
class Transactor;
  Transaction tr;
  mailbox mbx_in;
  covergroup CovPort;
    coverpoint tr.port;
  endgroup

  function new(mailbox mbx_in);
    CovPort = new;               // Instantiate covergroup
    this.mbx_in = mbx_in;
  endfunction

  task main;
    forever begin
      tr = mbx_in.get;        // Get next transaction
      ifc.cb.port <= tr.port; // Send into DUT
      ifc.cb.data <= tr.data;
      CovPort.sample();       // Gather coverage
    end
  endtask

endclass
```

9.6   Triggering a Cover Group

The two major parts of functional coverage are the sampled data values and the time when they are sampled. When new values are ready (such as when a transaction has completed), your testbench triggers the cover group. This can be done directly with the **sample** task, as shown in Example 9-5, or by using a blocking expression in the **covergroup** definition. The blocking expression can use a **wait** or **@** to block on signals or events.

Use the **sample** method if you want to explicitly trigger coverage from procedural code, if there is no existing signal or event that tells when to sample, or if there are multiple instances of a cover group that trigger separately.

Use the blocking statement in the **covergroup** declaration if you want to tap into existing events or signals to trigger coverage.

9.6.1   Sampling using a callback

One of the better ways to integrate functional coverage into your testbench is to use callbacks, as originally shown in section 8.7. This technique allows you to build a flexible testbench without restricting when coverage is collected. You can decide for every point in the verification plan where and when

data is sampled. And if you need an extra "hook" in the environment for a callback, you can always add one in an unobtrusive manner, as a callback only "fires" when the test registers a callback object. You can create many separate callbacks for each cover group, with little overhead. As explained in section 8.7.3, callbacks are superior to using a mailbox to connect the test-bench to the coverage objects. You might need multiple mailboxes to collect transactions from different points in your testbench. A mailbox requires a transactor to receive transactions, and multiple mailboxes cause you to juggle multiple threads. Instead of an active transactor, use a passive callback.

Example 8-25 shows a driver class that has two callback points, before and after the transaction is transmitted. Example 8-24 shows the base callback class, and Example 8-26 has a test with an extended callback class that sends data to a scoreboard. Make your own extension, **Driver_cbs_coverage**, of the base callback class, **Driver_cbs,** to call the **sample** task for your cover group in **post_tx**. Push an instance of the coverage callback class into the driver's callback queue, and your coverage code triggers the cover group at the right time. The following two examples define and use the callback **Driver_cs_coverage**.

Example 9-6 Test using functional coverage callback

```
program automatic test;
  Environment env;

  initial begin
    Driver_cbs_coverage dcc;

    env = new;
    env.gen_cfg;
    env.build;

    // Create and register the coverage callback
    dcc = new;
    env.drv.cbs.push_back(dcc); // Put into driver's Q

    env.run;
    env.wrapup.
  end

endprogram
```

Example 9-7 Callback for functional coverage

```
class Driver_cbs_coverage extends Driver_cbs;
  covergroup CovPort;
    ...
  endgroup

  virtual task post_tx(Transaction tr);
    CovPort.sample();          // Sample coverage values
  endtask
endclass
```

9.6.2   Cover group with an event trigger

In Example 9-8, the cover group **CovPort** is sampled when the testbench triggers the **trans_ready** event.

Example 9-8 Cover group with a trigger

```
event trans_ready;
covergroup CovPort @(trans_ready);
    coverpoint ifc.cb.port;   // Measure coverage
endgroup
```

The advantage of using an event over calling the **sample** routine directly is that you may be able to use an existing event such as one triggered by an assertion, as shown in Example 9-10.

9.6.3   Triggering on a SystemVerilog Assertion

If you already have a SVA that looks for useful events like a complete transaction, you can add an event trigger to wake up the cover group.

Example 9-9 Module with SystemVerilog Assertion

```
module mem(simple_bus sb);
  bit [7:0] data, addr;
  event write_event;

  cover property
   (@(posedge sb.clock) sb.write_ena==1)
    -> write_event;
endmodule
```

Example 9-10 Triggering a cover group with an SVA

```
program automatic test(simple_bus sb);

  covergroup Write_cg @($root.top.m1.write_event);
    coverpoint $root.top.m1.data;
    coverpoint $root.top.m1.addr;
  endgroup

  Write_cg wcg;

  initial begin
    wcg = new;
    // Apply stimulus here
    sb.write_ena <= 1;
    ...
    #10000 $finish;
  end
endprogram
```

9.7   Data Sampling

How is coverage information gathered? When you specify a variable or expression in a cover point, SystemVerilog creates a number of "bins" to record how many times each value has been seen. These bins are the basic units of measurement for functional coverage. If you sample a one-bit variable, a maximum of two bins are created. You can imagine that SystemVerilog drops a token in one or the other bin every time the cover group is triggered. At the end of each simulation, a database is created with all bins that have a token in them. You then run an analysis tool that reads all databases and generates a report with the coverage for each part of the design and for the total coverage.

9.7.1   Individual bins and total coverage

To calculate the coverage for a point, you first have to determine the total number of possible values, also known as the domain. There may be one value per bin or multiple values. Coverage is the number of sampled values divided by the number of bins in the domain.

A cover point that is a 3-bit variable has the domain 0:7 and is normally divided into eight bins. If, during simulation, values belonging to seven bins are sampled, the report will show 7/8 or 87.5% coverage for this point. All these points are combined to show the coverage for the entire group, and then

all the groups are combined to give a coverage percentage for all the simulation databases.

This is the status for a single simulation. You need to track coverage over time. Look for trends so you can see where to run more simulations or add new constraints or tests. Now you can better predict when verification of the design will be completed.

9.7.2   Creating bins automatically

As you saw in the report in Example 9-3, SystemVerilog automatically creates bins for cover points. It looks at the domain of the sampled expression to determine the range of possible values. For an expression that is N bits wide, there are 2^N possible values. For the 3-bit **port** in that example, there are eight possible values. The range of an enumerated type is shown in section 9.7.8. The domain for enumerated data types is the number of named values You can also explicitly define bins as shown in section 9.7.5.

9.7.3   Limiting the number of automatic bins created

The cover group option **auto_bin_max** specifies the maximum number of bins to automatically create, with a default of 64 bins. If the domain of values in the cover point variable or expression is greater than this option, SystemVerilog divides the range into **auto_bin_max** bins. For example, a 16-bit variable has 65,536 possible values, so each of the 64 bins covers 1024 values.

In reality, you may find this approach impractical, as it is very difficult to find the needle of missing coverage in a haystack of auto-generated bins. Lower this limit to 8 or 16, or better yet, explicitly define the bins as shown in section 9.7.5.

The following code takes the chapter's first example and adds a cover point option that sets **auto_bin_max** to two bins. The sampled variable is still **port**, which is three bits wide, for a domain of eight possible values. The first bin holds the lower half of the range, 0–3, and the other hold the upper values, 4–7.

Example 9-11 Using auto_bin_max set to 2

```
covergroup CovPort;
  coverpoint tr.port
    { options.auto_bin_max = 2; }  // Divide into 2 bins
endgroup
```

The coverage report from VCS shows the two bins. This simulation achieved 100% coverage because the eight **port** values were mapped to two bins. Since both bins have sampled values, your coverage is 100%.

Example 9-12 Report with `auto_bin_max` set to 2

```
Bin               # hits    at least
================================
auto[0:3]         15        1
auto[4:7]         17        1
================================
```

Example 9-11 used **auto_bin_max** as an option for the cover point only. You can also use it as an option for the entire group.

Example 9-13 Using `auto_bin_max` for all cover points

```
covergroup CovPort;
  options.auto_bin_max = 2; // Affects port & data
  coverpoint tr.port;
  coverpoint tr.data;
endgroup
```

9.7.4   Sampling expressions

You can sample expressions, but always check the coverage report to be sure you are getting the values you expect. You may have to adjust the width of the computed expression, as shown in section 2.15. For example, sampling a 3-bit header length (0:7) plus a 4-bit payload length (0:15) creates only 2^4 or 16 bins, which may not be enough if your transactions can actually be 0:23 bytes long.

Example 9-14 has a cover group that samples the total transaction length. The cover point has a label to make it easier to read the coverage report. Also, the expression has an additional dummy constant so that the transaction length is computed with 5-bit precision, for a maximum of 32 auto-generated bins.

Example 9-14 Using an expression in a cover point

```
class Transaction;
  rand bit [2:0] hdr_len;      // range: 0:7
  rand bit [3:0] payload_len;  // range: 0:15
  ...
endclass

Transaction tr;

covergroup CovLen;
  len16: coverpoint (tr.hdr_len + tr.payload_len);
  len32: coverpoint (tr.hdr_len + tr.payload_len + 5'b0);
endgroup
```

A quick run with 200 transactions showed that the **len16** had 100% coverage, but this is across only 16 bins. The cover point **len32** had 68% coverage across 32 bins. Neither of these cover points are correct, as the maximum length has a domain of 0:23. The auto-generated bins just don't work, as the maximum length is not a power of 2.

9.7.5   User-defined bins find a bug

Automatically generated bins are okay for anonymous data values, such as counter values, addresses, or values that are a power of 2. For other values, you should explicitly name the bins to improve accuracy and ease coverage report analysis. SystemVerilog automatically creates bin names for enumerated types, but for other variables you need to give names to the interesting states.

The easiest way to specify bins is with the **[ ]** syntax.

Example 9-15 Defining bins for transaction length

```
covergroup CovLen;
  len: coverpoint (tr.hdr_len + tr.payload_len + 5'b0)
    {bins len[] = {[0:23]}; }
endgroup
```

After sampling 2000 random transactions, the group has 95.83% coverage.

Example 9-16 Coverage report for transaction length

```
Bin          # hits     at least
==============================
len_00     13           1
len_01     36           1
len_02     51           1
len_03     60           1
len_04     72           1
len_05     88           1
len_06     127          1
len_07     122          1
len_08     133          1
len_09     138          1
len_0a     115          1
len_0b     128          1
len_0c     125          1
len_0d     111          1
len_0e     115          1
len_0f     134          1
len_10     107          1
len_11     102          1
len_12     70           1
len_13     65           1
len_14     39           1
len_15     30           1
len_16     19           1
len_17     0            1
==============================
```

A quick look at the report shows the problem — the length of 23 (17 hex) was never seen. The longest header is 7, and the longest payload is 15, for a total of 22, not 23! If you change to the **bins** declaration to use 0:22, the coverage jumps to 100%. The user-defined bins found a bug in the test.

9.7.6   Naming the cover point bins

Example 9-17 samples a 4-bit variable, **kind**, that has 16 possible values. The first bin is called **zero** and counts the number of times that **kind** is 0 when sampled. The next three values, 1–3, are all grouped into a single bin, **lo**. The upper eight values, 8–15, are kept in separate bins, **hi_8**, **hi_9**, **hi_a**, **hi_b**, **hi_c**, **hi_d**, **hi_e**, and **hi_f**. Note how **$** in the **hi** bin expression is used as a shorthand notation for the largest value for the sampled variable. Lastly, **misc** holds all values that were not previously chosen, namely 4–7.

Example 9-17 Specifying bin names

```
covergroup CovKind;
  coverpoint tr.kind {
    bins zero = {0};        // A bin for kind==0
    bins lo   = {[1:3]};    // 1 bin for values 1:3
    bins hi[] = {[8:$]};    // 8 separate bins
    bins misc = default;    // 1 bin for all the rest
  }
endgroup // CoverKind
```

Note that the additional information about the **coverpoint** is grouped using curly braces: {}. This is because the bin specification is declarative code, not procedural code that would be grouped with **begin**...**end**. Lastly, the final curly brace is NOT followed by a semicolon, just as an **end** never is.

Now you can easily see which bins have no hits — **hi_8** in this case.

Example 9-18 Report showing bin names

```
Bin        # hits    at least
============================
hi_8          0         1
hi_9          5         1
hi_a          3         1
hi_b          4         1
hi_c          2         1
hi_d          2         1
hi_e          9         1
hi_f          4         1
lo           16         1
misc         15         1
zero          1         1
============================
```

When you define the bins, you are restricting the values used for coverage to those that are interesting to you. SystemVerilog no longer automatically creates bins, and it ignores values that do not fall into a predefined bin. More importantly, only the bins you create are used to calculate functional coverage. You get 100% coverage only as long as you get a hit in every specified bin.

Values that do not fall into any specified bin are ignored. This rule is useful if the sampled value, such as transaction length, is not a power of 2. In general, if you are specifying bins, always use the **default** bin specifier to catch values that you may have forgotten.

9.7.7   Conditional coverage

You can use the **iff** keyword to add a condition to a cover point. The most common reason for doing so is to turn off coverage during reset so that stray triggers are ignored. Example 9-19 gathers only values of **port** when **reset** is 0, where **reset** is active-high.

Example 9-19 Conditional coverage — disable during reset

```
covergroup CoverPort;
  // Don't gather coverage when reset==1
  coverpoint port iff (!bus_if.reset);
endgroup
```

Alternately, you can use the **start** and **stop** functions to control individual instances of cover groups.

Example 9-20 Using stop and start functions

```
initial begin
  CovPort ck = new;           // Instantiate cover group

  // Reset sequence stops collection of coverage data
  #1ns bus_if.reset = 1;
  ck.stop();
  #100ns bus_if.reset = 0; // End of reset
  ck.start();
  ...
end
```

9.7.8   Creating bins for enumerated types

For enumerated types, SystemVerilog creates one bin for each possible value.

Example 9-21 Functional coverage for an enumerated type

```
typedef enum {INIT, DECODE, IDLE} fsmstate_t;
fsmstate_t pstate, nstate;   // declare typed variables
covergroup cg_fsm;
  coverpoint pstate;
endgroup
```

Here is part of the coverage report from VCS, showing the bins for the enumerated types.

Example 9-22 Report with `auto_bin_max` set to 2

```
Bin              # hits    at least
=================================
auto_DECODE      11        1
auto_IDLE        11        1
auto_INIT        10        1
=================================
```

If you want to group multiple values into a single bin, you have to define your own bins. Any bins outside the enumerated values are ignored unless you define a bin with the default specifier. When you gather coverage on enumerated types, **auto_bin_max** does not apply.

9.7.9   Transition coverage

You can specify state transitions for a cover point. In this way, you can tell not only what interesting values were seen but also the sequences. For example, you can check if **port** ever went from 0 to 1, 2, or 3.

Example 9-23 Specifying transitions for a cover point

```
covergroup CoverPort;
  coverpoint port {
    bins t1 = (0 => 1), (0 => 2), (0 => 3);
  }
endgroup
```

You can quickly specify multiple transitions using ranges. The expression **(1,2 => 3,4)** creates the four transitions **(1=>3), (1=>4), (2=>3)**, and **(2=>4)**.

You can specify transitions of any length. Note that you have to sample once for each state in the transition. So **(0 => 1 => 2)** is different from **(0 => 1 => 1 => 2)** or **(0 => 1 => 1 => 1 => 2)**. If you need to repeat values, as in the last sequence, you can use the shorthand form:**(0 => 1[*3] => 2)**. To repeat the value 1 for 3, 4, or 5 times, use **1[*3:5]**.

9.7.10   Wildcard states and transitions

You use the **wildcard** keyword to create multiple states and transitions. Any **x**, **z**, or **?** in the expression is treated as a wildcard for 0 or 1. The following creates a cover point with a bin for even values and one for odd.

Example 9-24 Wildcard bins for a cover point

```
bit [2:0] port;
covergroup CoverPort;
  coverpoint port {
    wildcard bins even = {3'b??0};
    wildcard bins odd  = {3'b??1};
  }
endgroup
```

9.7.11   Ignoring values

With some cover points, you never get all possible values. For instance, a 3-bit variable may be used to store just six values, 0–5. If you use automatic bin creation, you never get beyond 75% coverage. There are two ways to solve this problem. You can explicitly define the bins that you want to cover as show in section 9.7.5. Alternatively, you can let SystemVerilog automatically create bins, and then use **ignore_bins** to tell which values to exclude from functional coverage calculation.

Example 9-25 Cover point with `ignore_bins`

```
bit [2:0] low_ports_0_5;        // Only uses values 0-5
covergroup CoverPort;
  coverpoint low_ports_0_5 {
    ignore_bins hi = {[6,7]}; // Ignore upper 2 bins
  }
endgroup
```

The original range of **low_ports_0_5**, a three-bit variable is 0:7. The **ignore_bins** excludes the last two bins, which reduces the range to 0:5. So total coverage for this group is the number of bins with samples, divided by the total number of bins, which is 5 in this case.

Example 9-26 Cover point with `auto_bin_max` and `ignore_bins`

```
bit [2:0] low_ports_0_5;        // Only uses values 0-5
covergroup CoverPort;
  coverpoint low_ports_0_5 {
    options.auto_bin_max = 4; // 0:1, 2:3, 4:5, 6:7
    ignore_bins hi = {[6,7]}; // Ignore upper 2 values
  }
endgroup
```

If you define bins either explicitly or by using the **auto_bin_max** option, and then ignore them, the ignored bins do not contribute to the calculation of coverage. In Example 9-26, initially four bins are created using

`auto_bin_max`: 0:1, 2:3, 4:5, and 6:7. But then the uppermost bin is eliminated by `ignore_bins`, so in the end only three bins are created. This cover point can have coverage of 0%, 33%, 66%, or 100%

9.7.12   Illegal bins

Some sampled values not only should be ignored, but also should cause an error if they are seen. This is best done in the testbench's monitor code, but can also be done by labeling a bin with `illegal_bins`. Use `illegal_bins` to catch states that were missed by the test's error checking. This also double-checks the accuracy of your bin creation: if an illegal value is found by the cover group, it is a problem either with the testbench or with your bin definitions.

Example 9-27 Cover point with `illegal_bins`

```
bit [2:0] low_ports_0_5;      // Only uses values 0-5
covergroup CoverPort;
  coverpoint low_ports_0_5 {
    illegal_bins hi = {[6,7]}; // Give error if seen
  }
endgroup
```

9.7.13   State machine coverage

You should have noticed that if a cover group is used on a state machine, you can use bins to list the specific states, and transitions for the arcs. But this does not mean you should use SystemVerilog's functional coverage to measure state machine coverage. You would have to extract the states and arcs manually. Even if you did this correctly the first time, you might miss future changes to the design code. Instead, use a code coverage tool that extracts the state register, states, and arcs automatically, saving you from possible mistakes.

However, an automatic tool extracts the information exactly as coded, mistakes and all. You may want to monitor small, critical state machines manually using functional coverage.

9.8   Cross Coverage

A cover point records the observed values of a single variable or expression. You may want to know not only what bus transactions occurred but also what errors happened during those transactions, and their source and destination. For this you need cross coverage that measures what values were seen for two or more cover points at the same time. Note that when you measure

cross coverage of a variable with **N** values, and of another with **M** values, SystemVerilog needs **NxM** cross bins to store all the combinations.

9.8.1   Basic cross coverage example

Previous examples have measured coverage of the transaction kind, and port number, but what about the two combined? Did you try every kind of transaction into every port? The **cross** construct in SystemVerilog records the combined values of two or more cover points in a group. The **cross** statement takes only cover points or a simple variable name. If you want to use expressions, hierarchical names or variables in an object such as **handle.variable**, you must first specify the expression in a **coverpoint** with a label and then use the label in the **cross** statement.

Example 9-28 creates cover points for **tr.kind** and **tr.port**. Then the two points are crossed to show all combinations. SystemVerilog creates a total of 128 (8 x 16) bins. Even a simple cross can result in a very large number of bins.

Example 9-28 Basic cross coverage

```
class Transaction;
  rand bit [3:0] kind;
  rand bit [2:0] port;
endclass

Transaction tr;

covergroup CovPort;
  kind: coverpoint tr.kind;  // Create cover point kind
  port: coverpoint tr.port   // Create cover point port
  cross kind, port;          // Cross kind and port
endgroup
```

A random testbench created 200 transactions and produced the coverage report in Example 9-29. Note that even though all possible **kind** and **port** values were generated, about 1/8 of the cross combinations were not seen.

Example 9-29 Coverage summary report for basic cross coverage

```
Cumulative report for Transaction::CovPort
Summary:
    Coverage: 95.83
    Goal: 100
```

Coverpoint	Coverage	Goal	Weight
kind	100.00	100	1
port	100.00	100	1

Cross	Coverage	Goal	Weight
Transaction::CovPort	87.50	100	1

```
Cross Coverage report
CoverageGroup: Transaction::CovPort
    Cross: Transaction::CovPort
Summary
    Coverage: 87.50
    Goal: 100
    Coverpoints Crossed: kind port
    Number of Expected Cross Bins: 128
    Number of User Defined Cross Bins: 0
    Number of Automatically Generated Cross Bins: 112

    Automatically Generated Cross Bins
```

kind	port	# hits	at least
auto[0]	auto[0]	1	1
auto[0]	auto[1]	4	1
auto[0]	auto[2]	3	1
auto[0]	auto[5]	1	1

```
...
```

9.8.2   Labeling cross coverage bins

If you want more readable cross coverage bin names, you can label the individual cover point bins, and SystemVerilog will use these names when creating the cross bins.

Example 9-30 Specifying cross coverage bin names

```
covergroup CovPortKind;
  port: coverpoint tr.port
   {bins port[] = {[0:$]};
   }
  kind: coverpoint tr.kind
   {bins zero = {0};          // A bin for kind==0
    bins lo   = {[1:3]};      // 1 bin for values 1:3
    bins hi[] = {[8:$]};      // 8 separate bins
    bins misc = default;      // 1 bin for all the rest
  }
  cross kind, port;
endgroup
```

If you define bins that contain multiple values, the coverage statistics change. In the report below, the number of bins has dropped from 128 to 88. This is because **kind** has 11 bins: **zero**, **lo**, **hi_8**, **hi_9**, **hi_a**, **hi_b**, **hi_c**, **hi_d**, **hi_e**, **hi_f**, and **misc**. The percentage of coverage jumped from 87.5% to 90.91% because any single value in the **lo** bin, such as 2, allows that bin to be marked as covered, even if the other values, 0 or 3, are not seen.

Example 9-31 Cross coverage report with labeled bins

```
Summary
    Coverage: 90.91
    Number of Coverpoints Crossed: 2
    Coverpoints Crossed: kind port
    Number of Expected Cross Bins: 88
    Number of Automatically Generated Cross Bins: 80
    Automatically Generated Cross Bins
```

port	kind	# hits	at least
port_0	hi_8	3	1
port_0	hi_a	1	1
port_0	hi_b	4	1
port_0	hi_c	4	1
port_0	hi_d	4	1
port_0	hi_e	1	1
port_0	lo	7	1
port_0	misc	6	1
port_0	zero	1	1
port_1	hi_8	3	1

...

9.8.3 Excluding cross coverage bins

To reduce the number of bins, use `ignore_bins`. With cross coverage, you specify the cover point with `binsof` and the set of values with `intersect` so that a single `ignore_bins` construct can sweep out many individual bins.

Example 9-32 Excluding bins from cross coverage

```
covergroup CovCovport;
  port: coverpoint tr.port
   {bins port[] = {[0:$]};
   }
  kind: coverpoint tr.kind
   {
    bins zero = {0};        // A bin for kind==0
    bins lo   = {[1:3]};    // 1 bin for values 1:3
    bins hi[] = {[8:$]};    // 8 separate bins
    bins misc = default;    // 1 bin for all the rest
  }
  cross kind, port {
    ignore_bins hi = binsof(port) intersect {7};
    ignore_bins md = binsof(port) intersect {0} &&
                     binsof(kind) intersect {[9:10]};
    ignore_bins lo = binsof(kind.lo);
  }
endgroup
```

The first `ignore_bins` just excludes bins where `port` is 7 and any value of `kind`. Since `kind` is a 4-bit value, this statement excludes 16 bins. The second `ignore_bins` is more selective, ignoring bins where `port` is 0 and `kind` is 9, 10, or 11, for a total of 3 bins.

The `ignore_bins` can use the bins defined in the individual cover points. The `ignore_bins lo` uses bin names to exclude `kind.lo` that is 1, 2, or 3. The bins must be names defined at compile-time, such as `zero` and `lo`. The bins `hi_8`, `hi_a`,... `hi_f`, and any automatically generated bins do not have names that can be used at compile-time in other statements such as `ignore_bins`; these names are created at run-time or during the report generation.

Note that `binsof` uses parentheses `()` while `intersect` specifies a range and therefore uses curly braces `{}`.

9.8.4   Excluding cover points from the total coverage metric

The total coverage for a group is based on all simple cover points and cross coverage. If you are only sampling a variable or expression in a **coverpoint** to be used in a **cross** statement, you should set its weight to 0 so that it does not contribute to the total coverage.

Example 9-33 Specifying cross coverage weight

```
covergroup CovPort;
  kind: coverpoint tr.kind
   {bins kind[] = {[0:$]};
    weight = 0;              // Don't count towards total
  }
  port: coverpoint tr.port
   {bins zero = {0};
    bins lo   = {[1:3]};
    bins hi[] = {[8:$]};
    bins misc = default;
    weight = 5;              // Count in total
  }
  cross kind, port
    {weight = 10;}           // Give cross extra weight
endgroup
```

9.8.5   Merging data from multiple domains

One problem with cross coverage is that you may need to sample values from different timing domains. You might want to know if your processor ever received an interrupt in the middle of a cache fill. The interrupt hardware is probably very distinct and may use different clocks than the cache hardware. But a previous design had a bug of this very sort, so you want to make sure you have tested this case.

The solution is to create a timing domain separate from the cache or interrupt hardware. Make copies of the signals into temporary variables and then sample them in a new coverage group that measures the cross coverage.

9.8.6   Cross coverage alternatives

As your cross coverage definition becomes more elaborate, you may spend considerable time specifying which bins should be used and which should be ignored. You may have two random bits, **a** and **b** with three interesting states, {**a==0**, **b==0**}, {**a==1**, **b==0**}, and {**b==1**}.

Example 9-34 shows how you can name bins in the cover points and then gather cross coverage using those bins.

Example 9-34 Cross coverage with bin names

```
class Transaction;
  rand bit a, b;
endclass

covergroup CrossBinNames;
  a: coverpoint tr.a
     { bins a0 = {0};
       bins a1 = {1};
       option.weight=0;}     // Only count cross
  b: coverpoint tr.b
     { bins b0 = {0};
       bins b1 = {1};
       option.weight=0;}     // Only count cross
  ab: cross a, b
     { bins a0b0 = binsof(a.a0) && binsof(b.b0);
       bins a1b0 = binsof(a.a1) && binsof(b.b0);
       bins b0   = binsof(b.b0); }
endgroup
```

Example 9-35 gathers the same cross coverage, but now uses **binsof** to specify the cross coverage values.

Example 9-35 Cross coverage with `binsof`

```
class Transaction;
  rand bit a, b;
endclass

covergroup CrossBinsofIntersect;
  a: coverpoint tr.a
     { option.weight=0; }   // Only count cross
  b: coverpoint tr.b
     { option.weight=0; }   // Only count cross
  ab: cross a, b
       { bins a0b0 = binsof(a) intersect {0} &&
                     binsof(b) intersect {0};
         bins a1b0 = binsof(a) intersect {1} &&
                     binsof(b) intersect {0};
         bins b1   = binsof(b) intersect {1}; }
endgroup
```

Alternatively, you can make a cover point that samples a concatenation of values. Then you only have to define bins using the less complex cover point syntax.

```
covergroup CrossManual;
  ab: coverpoint {tr.a, tr.b}
    { bins a0b0 = {2'b00};
      bins a1b0 = {2'b10};
      wildcard bins b1 = {2'b?1};
    }
endgroup
```

Use the style in Example 9-34 if you already have bins defined for the individual cover points and want to use them to build the cross coverage bins. Use Example 9-35 if you need to build cross coverage bins but have no predefined cover point bins. Use Example 9-36 if you want the tersest format.

9.9 Coverage Options

You can specify additional information in the cover group using options. Options can be placed in the cover group so that they apply to all cover points in the group, or they can be put inside a single cover point for finer control. You have already seen the `auto_bin_max` option. Here are several more.

9.9.1 Cover group comment

You can add a comment into coverage reports to make them easier to analyze. A comment could be as simple as the section number from the verification plan to tags used by a report parser to automatically extract relevant information from the sea of data.

```
covergroup CoverPort;
  option.comment = "Section 3.2.14 Port numbers";
  coverpoint port;
endgroup
```

9.9.2 Per-instance coverage

If your testbench instantiates a coverage group multiple times, by default SystemVerilog groups together all the coverage data from all the instances. But you may have several generators, each creating very different streams of transactions, so you may want to see separate reports. For example, one generator may be creating long transactions while another makes short ones. The following cover group can be instantiated in each separate generator. It keeps track of coverage for each instance.

Example 9-38 Specifying per-instance coverage

```
covergroup CoverLength;
  coverpoint tr.length;
  option.per_instance = 1;
endgroup
```

9.9.3   Coverage threshold using `at_least`

You may not have sufficient visibility into the design to gather robust coverage information. Suppose you are verifying that a DMA state machine can handle bus errors. You don't have access to its current state, but you know the range of cycles that are needed for a transfer. So if you repeatedly cause errors during that range, you have probably covered all the states. So you could set **option.at_least** to 8 or more to specify that after 8 hits on a bin, you are confident that you have exercised that combination.

If you define **option.at_least** at the cover group level, it applies to all cover points. If you define it inside a point, it only applies to that single point.

However, as Example 9-2 showed, even after 32 attempts, the random **kind** variable still did not hit all possible values. So only use **at_least** if there is no direct way to measure coverage.

9.9.4   Printing the empty bins

By default, the coverage report shows only the bins with samples. Your job is to verify all that is listed in the verification plan, so you are actually more interested in the bins without samples. Use the option **cross_num_print_missing** to tell the simulation and report tools to show you all bins, especially the ones with no hits. Set it to a large value, as shown in Example 9-39, but no larger than you are willing to read.

Example 9-39 Report all bins including empty ones

```
covergroup CovPort;
  kind: coverpoint tr.kind;
  port: coverpoint tr.port
  cross kind, port;
  option.cross_num_print_missing = 1_000;
endgroup
```

This option affects only the report, and may be deprecated in a future version of the SystemVerilog IEEE standard.

9.9.5   Coverage goal

The goal for a cover group or point is the level at which the group or point is considered fully covered. The default is 100% coverage.[12] If you set this level below 100%, you are requesting less than complete coverage, which is probably not desirable. This option affects only the coverage report.

Example 9-40 Specifying the coverage goal

```
covergroup CoverPort;
  coverpoint port;
  option.goal = 90;  // Settle for partial coverage
endgroup
```

9.10   Parameterized Cover Groups

As you start writing cover groups, you will find that some are very close to one another. SystemVerilog allows you to parameterize a cover group so that you can create a generic definition and then specify a few unique details when you instantiate it. You cannot pass the trigger into a coverage group instance. Instead, you can put a coverage group into a class and pass the trigger into the constructor.

9.10.1   Pass cover group parameters by value

Example 9-41 shows a cover group that uses a parameter to split the range into two halves. Just pass the midpoint value to the cover groups' **new** function.

Example 9-41 Simple parameter

```
bit [2:0] port;      // Values: 0:7

covergroup CoverPort (int mid);
  coverpoint port
   {bins lo = {[0:mid-1]};
    bins hi = {[mid:$]};
  }
endgroup

CoverPort cp;
initial
  cp = new(5);            // lo=0:4, hi=5:7
```

[12.] The 12/2005 SystemVerilog LRM mentions a default of both 100% and 90%, but the latter is an error.

9.10.2 Pass cover group parameters by reference

You can specify a variable to be sampled with pass-by-reference. Here you want the cover group to sample the value during the entire simulation, not just to use the value when the constructor is called.

Example 9-42 Pass-by-reference

```
bit [2:0] port_a, port_b;

covergroup CoverPort (ref bit [2:0] port, int mid);
  coverpoint port {
    bins lo = {[0:mid-1]};
    bins hi = {[mid:$]};
  }
endgroup

CoverPort cpa, cpb;
initial
  begin
  cpa = new(port_a, 4);  // port_a, lo=0:4, hi=5:7
  cpb = new(port_b, 2);  // port_b, lo=0:1, hi=3:7
  end
```

9.11 Analyzing Coverage Data
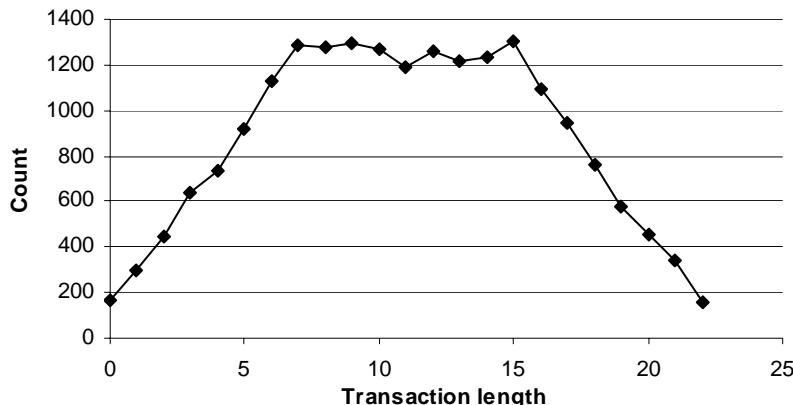
In general, assume you need more seeds and fewer constraints. After all, it is easier to run more tests than to construct new constraints. If you are not careful, new constraints can easily restrict the search space.

If your cover point has only zero or one sample, your constraints are probably not targeting these areas at all. You need to add constraints that "pull" the solver into new areas. In Example 9-15, the transaction length had an uneven distribution. This situation is similar to the distribution seen when you roll two dice and look at the total value.

Example 9-43 Original class for transaction length

```
class Transaction;
  rand bit [2:0] hdr_len;
  rand bit [3:0] payload_len;
  rand bit [4:0] len;
  constraint ct {len == hdr_len + payload_len; }
endclass
```
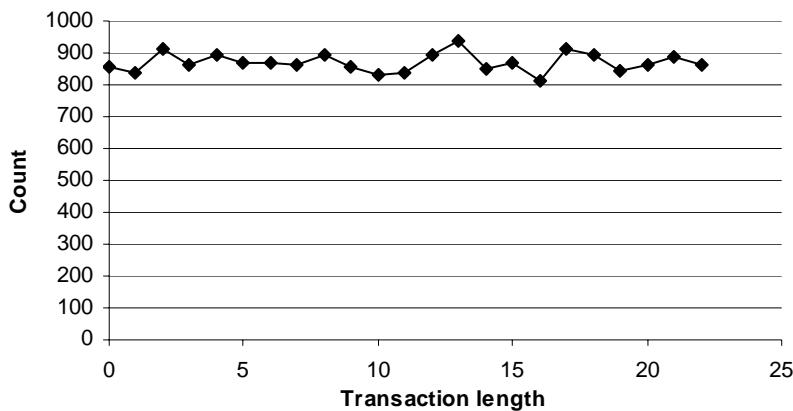
The problem with this class is that **len** is not evenly weighted.

Figure 9-5 Uneven probability for transaction length

If you want to make total length be evenly distributed, use a **solve**...**before** constraint.

Example 9-44 `solve`...`before` constraint for transaction length

```
constraint sb {solve len before hdr_len, payload_len;}
```

Figure 9-6 Even probability for transaction length with `solve`...`before`

The alternative to **solve**...**before** is the **dist** constraint. But it is not effective, as **len** is also being constrained by the sum of the two lengths.

9.12   Measuring Coverage Statistics During Simulation

You can query the level of functional coverage on the fly during simulation by using the **get_coverage** and **get_coverage_inst** functions. This

approach allows you to check whether you have reached your coverage goals, and possibly to control a random test.

The most practical use for these functions is to monitor coverage over a long test. If the coverage level does not advance after a given number of transactions or cycles, the test should stop. Hopefully, another seed or test will increase the coverage.

While it would be nice to have a test that can perform some sophisticated actions based on functional coverage results, it is very hard to write this sort of test. Each test / random seed pair may uncover new functionality, but it may take many runs to reach a goal. If a test finds that it has not reached 100% coverage, what should it do? Run for more cycles? How many more? Should it change the stimulus being generated? How can you correlate a change in the input with the level of functional coverage? The one reliable thing to change is the random seed, which you should only do once per simulation. Otherwise, how can you reproduce a design bug if the stimulus depends on multiple random seeds?

You can query the functional coverage statistics if you want to create your own coverage database. Verification teams have built their own SQL databases that are fed functional coverage data from simulation. This setup allows them greater control over the data, but requires a lot of work outside of creating tests.

Some formal verification tools can extract the state of a design and then create input stimulus to reach all possible states. Don't try to duplicate this in your testbench!

9.13   Conclusion

When you switch from writing directed tests, hand-crafting every bit of stimulus, to constrained-random testing, you might worry that the tests are no longer under your command. By measuring coverage, especially functional coverage, you regain control by knowing what features have been tested.

Using functional coverage requires a detailed verification plan and much time creating the cover groups, analyzing the results, and modifying tests to create the proper stimulus. But this effort is less than would be required to write the equivalent directed tests, and the coverage work helps you better track your progress in verifying your design.

Chapter 10

Advanced Interfaces

10.1   Introduction

In Chapter 5 you learned how to use interfaces to connect the design and testbench. These physical interfaces represent real signals, similar to the wires that connected ports in Verilog-1995. The testbench uses these interfaces by statically connecting to them through ports. However, for many designs, the testbench needs to connect dynamically to the design.

In a network switch, a single Driver class may connect to many interfaces, one for each input channel of the DUT. You wouldn't want to write a unique Driver for each channel — instead you want to write a generic Driver, instantiate it N times, and have it connect each of the N physical interfaces. You can do this in SystemVerilog by using a virtual interface that is merely a handle to a physical interface.

You may need to write a testbench that attaches to several different configurations of your design. In one configuration, the pins of the chip may drive a USB bus, while in another the same pins may drive an I2C serial bus. Once again, you can use a virtual interface in the testbench so you can decide at runtime which drivers to use.

A SystemVerilog interface is more than just signals — you can put executable code inside. This might include routines to read and write to the interface, initial and always blocks that run code inside the interface, and assertions to constantly check the status of the signals. However, do not put testbench code in an interface. Program blocks have been created expressly for building a testbench, including scheduling their execution in the Reactive region, as described in the LRM.

10.2   Virtual Interfaces with the ATM Router

The most common use for a virtual interface is to allow objects in a testbench to refer to items in a replicated interface using a generic handle rather than the actual name. Virtual interfaces are the only mechanism that can bridge the dynamic world of objects with the static world of modules and interfaces.

10.2.1   The testbench with just physical interfaces

Chapter 5 showed how to build an interface to connect a 4x4 ATM router to a testbench. Here are the ATM interfaces (with the DUT **modport** removed for clarity).

Example 10-1 Interface with clocking block

```
// Rx interface with modports and clocking block
interface Rx_if (input logic clk);
  logic [7:0] data;
  logic soc, en, clav, rclk;

  clocking cb @(posedge clk);
    output data, soc, clav;  // Directions are relative
    input  en;                // to the testbench
  endclocking : cb

  modport TB (clocking cb);
endinterface : Rx_if


// Tx interface with modports and clocking block
interface Tx_if (input logic clk);
  logic [7:0] data;
  logic soc, en, clav, tclk;

  clocking cb @(posedge clk);
      input  data, soc, en;
      output clav;
  endclocking : cb

  modport TB (clocking cb);
endinterface : Tx_if
```

These interfaces can be used in a program block as follows. This procedural code is hard coded with interface names such as **Rx0** and **Tx0**.

Example 10-2 Testbench using physical interfaces

```
program automatic test(Rx_if.TB Rx0, Rx1, Rx2, Rx3,
                       Tx_if.TB Tx0, Tx1, Tx2, Tx3,
                       input logic clk, output logic rst);

  bit [7:0] bytes[`ATM_SIZE];

  initial begin
    // Reset the device
    rst <= 1;
    Rx0.cb.data <= 0;
    ...
    receive_cell0;
    ...
  end

  task receive_cell0;
    @(Tx0.cb);
    Tx0.cb.clav <= 1;          // Assert ready to receive
    wait (Tx0.cb.soc == 1);    // Wait for Start of Cell

    for (int i=0; i<`ATM_SIZE; i++) begin
      wait (Tx0.cb.en == 0);   // Wait for enable
        @(Tx0.cb);

      bytes[i] = Tx0.cb.data;
      @(Tx0.cb);
      Tx0.cb.clav <= 0;        // Deassert flow control
    end
  endtask

endprogram
```

10.2.2   Testbench with virtual interfaces

A good OOP technique is to create a class that uses a handle to reference an object, rather than a hard-coded object name. In this case, you can make a single Driver class and a single Monitor class, have them operate on a handle to the data, and then pass in the handle at run-time.

The following program block is still passed the 4 **Rx** and 4 **Tx** interfaces as ports, as in Example 10-2, but it creates an array of virtual interfaces, **vRx** and **vTx**. These can now be passed into the constructors for the drivers and monitors.

Example 10-3 Testbench using virtual interfaces

```
program automatic test(Rx_if.TB Rx0, Rx1, Rx2, Rx3,
                       Tx_if.TB Tx0, Tx1, Tx2, Tx3,
                       output logic rst);

  Driver drv[4];
  Monitor mon[4];
  Scoreboard scb[4];

  virtual Rx_if.TB vRx[4] = '{Rx0, Rx1, Rx2, Rx3};
  virtual Tx_if.TB vTx[4] = '{Tx0, Tx1, Tx2, Tx3};

  initial begin
    foreach (scb[i]) begin
      scb[i] = new(i);
      drv[i] = new(scb[i].exp_mbx, i, vRx[i]);
      mon[i] = new(scb[i].rcv_mbx, i, vTx[i]);
    end
    ...
  end

endprogram
```

The driver class looks similar to the code in Example 10-2, except it uses the virtual interface name **Rx** instead of the physical interface **Rx0**.

Example 10-4 Testbench using virtual interfaces

```
class Driver;
  int stream_id;
  bit done = 0;
  mailbox exp_mbx;
  virtual Rx_if.TB Rx;

  function new(mailbox exp_mbx,
               int      stream_id,
               virtual Rx_if.TB Rx);
    this.exp_mbx = exp_mbx;
    this.stream_id = stream_id;
    this.Rx = Rx;
  endfunction

  task run(int ncells, event driver_done);
    ATM_Cell ac;

    fork // Spawn this as a separate thread
      begin
      // Initialize output signals
      Rx.cb.clav <= 0;
      Rx.cb.soc <= 0;
      @Rx.cb;

      // Drive cells until the last one is sent
      repeat (ncells) begin
        ac = new
        assert(ac.randomize);
        if (ac.eot_cell) break;   // End of transmission
        drive_cell(ac);
      end

      $display("@%0d: Driver::run Driver[%0d] is done",
               $time, stream_id);
      -> driver_done;
      end
    join_none
  endtask // run

  task drive_cell(ATM_Cell ac);
    bit [7:0] bytes[];

    #ac.delay;
```

```
    ac.byte_pack(bytes);

    $display("@%0d: Driver::drive_cell(%0d) vci=%h",
             $time, stream_id, ac.vci);

    // Wait to start on a new cycle
    @Rx.cb;
    Rx.cb.clav <= 1;                  // Assert ready to xfr
    do
      @Rx.cb;
    while (Rx.cb.en != 0)             // Wait for enable low

    Rx.cb.soc <= 1;                   // Start of cell
    Rx.cb.data <= bytes[0];           // Drive first byte
    @Rx.cb;
    Rx.cb.soc <= 0;                   // Start of cell done
    Rx.cb.data <= bytes[1];           // Drive first byte

    for (int i=2; i<`ATM_SIZE; i++) begin
      @Rx.cb;
      Rx.cb.data <= bytes[i];
    end

    @Rx.cb;
    Rx.cb.soc <= 1'bz;                // Tristate SOC at end
    Rx.cb.clav <= 0;
    Rx.cb.data <= 8'bz;               // Clear data lines
    $display("@%0d: Driver::drive_cell(%0d) finish",
             $time, stream_id);

    // Send cell to scoreboard
    exp_mbx.put(ac);

  endtask // drive_cell_t

endclass // Driver
```

10.3   Connecting to Multiple Design Configurations

A common challenge to verifying a design is that it may have several con-
figurations. You could make a separate testbench for each configuration, but
this could lead to a combinatorial explosion as you explore every alternative.
Instead, you can use virtual interfaces to dynamically connect to the optional
interfaces.

10.3.1   A mesh design

Example 10-5 is built of a simple replicated component, an 8-bit counter. This resembles a DUT that has a device such as network chip or processor that is instantiated repeatedly in a mesh configuration. The key idea is that the top-level netlist creates an array of interfaces and counters. Now the testbench can connect its array of virtual interfaces to the physical ones.

Here is the code for the counter's interface, **X_if**.

Example 10-5 Interface for 8-bit counter

```
interface X_if (input logic clk);
    logic [7:0] din, dout;
    logic reset_l, load;

    clocking cb @(posedge clk);
    output din, load;
    input dout;
    endclocking

    always @cb
      $strobe("@%0d:%m: out=%0d, in=%0d, ld=%0d, r=%0d",
              $time, dout, din, load, reset_l);

    modport DUT (input clk, din, reset_l, load,
                 output dout);

    modport TB (clocking cb, output reset_l);
endinterface
```

The counter is simple.

Example 10-6 Counter model using X_if interface

```
// Simple 8-bit counter with load and active-low reset
module DUT(X_if.DUT xi);
  logic [7:0] count;
  assign xi.dout = count;

  always @(posedge xi.clk or negedge xi.reset_l)
    begin
      if (!xi.reset_l)  count = 0;
      else if (xi.load) count = xi.din;
      else              count++;
    end
endmodule
```

The top-level netlist uses a generate statement to instantiate **NUM_XI** inter-faces and counters, but only one testbench.

Example 10-7 Testbench using an array of virtual interfaces

```
parameter NUM_XI = 2;  // Number of design instances

module top;
  // Clock generator
  bit clk;
  initial forever #20 clk = ~clk;

  // Instantiate N interfaces
  X_if xi [NUM_XI] (clk);

  // Instantiate the testbench
  test tb();

  // Generate N DUT instances
  generate
  for (genvar i=0; i<NUM_XI; i++)
    begin : dut
      DUT d (xi[i]);
    end
  endgenerate

endmodule : top
```

The key line in the testbench is where the local virtual interface array, **vxi**, is assigned to point to the array of physical interfaces in the top module, **top.xi**. To simplify Example 10-7, the environment class has been merged with the test, while the generator, agent, and driver layers have been com-pressed into the driver.

The testbench assumes there is at least one counter and thus at least one X interface. If your design could have zero counters, you would have to use dynamic arrays, as fixed-size arrays cannot have a size of zero.

Example 10-8 Counter testbench using virtual interfaces

```
program automatic test;

  virtual X_if.TB vxi[NUM_XI]; // Virtual ifc array
  Driver driver[];

  initial begin
    // Connect local virtual interface to top
    vxi = top.xi;

    // Create N drivers
    driver = new[NUM_XI];
    foreach (driver[i]) begin
        driver[i] = new(vxi[i], i);
        driver[i].reset;
    end

    foreach (driver[i])
      driver[i].load;

    repeat (10) @(vxi[0].cb);
  end

endprogram
```

The **Driver** class uses a single virtual interface to drive and sample signals from the counter.

Example 10-9 `Driver` class using virtual interfaces

```
class Driver;
  virtual X_if xi;
  int id;

  function new(virtual X_if.TB xi, int id);
    this.xi = xi;
    this.id = id;
  endfunction

  task reset;
    fork
      begin
        $display("@%0d: %m: Start reset [%0d]",
                  $time, id);
        // Reset the device
        xi.reset_l <= 1;
        xi.cb.load <= 0;
        xi.cb.din <= 0;
        @(xi.cb)
          xi.reset_l <= 0;
        @(xi.cb)
          xi.reset_l <= 1;
        $display("@%0d: %m: End reset [%0d]",
                  $time, id);
      end
    join_none
  endtask

  task load;
    fork
      begin
        $display("@%0d: %m: Start load [%0d]",
                  $time, id);
        ##1 xi.cb.load <= 1;
        xi.cb.din <= id + 10;

        ##1 xi.cb.load <= 0;
        repeat (5) @(xi.cb);
        $display("@%0d: %m: End load [%0d]",
                  $time, id);
      end
    join_none
  endtask
```

```
endclass
```

10.3.2   Using `typedefs` with virtual interfaces

The type "`virtual  X_if.TB`" can be replaced with a **typedef**, as shown in the following code snippets of the testbench and driver.

Example 10-10 Testbench using a `typedef` for virtual interfaces

```
typedef virtual X_if.TB vX_t;

program automatic test;
  vX_t vxi[NUM_XI];        // Virtual interface array
  Driver driver[];
...
endprogram
```

Example 10-11 Driver using a `typedef` for virtual interfaces

```
class Driver;
  vX_t xi;
  int id;

  function new(vX_t xi, int id);
    this.xi = xi;
    this.id = id;
  endfunction
...
endclass // Driver
```

10.3.3   Passing virtual interface array using a port

The previous examples passed the array of virtual interfaces using a cross module reference (XMR). An alternative is to pass the array in a port. Since the array in the top netlist is static and so only needs to be referenced once, the XMR style makes more sense than using a port that normally is used to pass changing values.

The port style is cleaner in that the testbench does not need to know the hierarchical path to the top netlist. But this code needs to be replicated for each testbench, all of which must be modified if the DUT interface changes, and in real life can be quite large. Additionally, you must provide an argument for ALL possible interfaces you'll be using and attach only to those that the selected configuration uses.

Example 10-12 uses a global parameter to define the number of X interfaces. Here is a snippet of the top netlist.

Example 10-12 Testbench using an array of virtual interfaces

```
parameter NUM_XI = 2;  // Number of instances
module top;

  // Instantiate N interfaces
  X_if xi [NUM_XI] (clk);


  ...
  // Instantiate the testbench
  test tb(xi);

endmodule : top
```

Here is the testbench. It still needs to create an array of virtual interfaces so that it can pass them into the constructor for the driver class.

Example 10-13 Testbench passing virtual interfaces with a port

```
program automatic test(X_if xi [NUM_XI]);

  Driver driver[];
  virtual X_if vxi[NUM_XI];

  initial begin
    // Connect the local virtual interfaces to the top
    if (NUM_XI <= 0) $finish;

    driver = new[NUM_XI];
    vxi = xi;     // Assign the interface array

    for (int i=0; i<NUM_XI; i++) begin
      driver[i] = new(vxi[i], i);
      driver[i].reset;
    end
  ...
  end

endprogram
```

10.4   Procedural Code in an Interface

Just as a class contains both variables and routines, an interface can contain code such as routines, assertions, and **initial** and **always** blocks. Recall that an interface includes the signals and functionality of the communication between two blocks. So the interface block for a bus can contain the

signals and also routines to perform commands such as a read or write. The inner workings of these routines are hidden from the external blocks, allowing you to defer the actual implementation. Access to these routines is controlled using the **modport** statement, just as with signals. A task or function is imported into a **modport** so that it is then visible to any block that uses the **modport**.

These routines can be used by both the design and the testbench. This approach ensures that both are using the same protocol, eliminating a common source of testbench bugs.

Assertions in an interface are used to verify the protocol. An assertion can check for illegal combinations, such as protocol violations and unknown values. These can display state information and stop simulation immediately so that you can easily debug the problem. An assertion can also fire when good transactions occur. Functional coverage code will uses this type of assertion to trigger the gathering of coverage information.

10.4.1   Interface with parallel protocol

When creating your system, you may not know whether to choose a parallel or serial protocol. The interface in Example 10-14 has two tasks, **initiatorSend** and **targetRcv**, that send a transaction between two blocks using the interface signals. It sends the address and data in parallel across two 8-bit buses.

Example 10-14 Interface with tasks for parallel protocol

```
interface simple_if(input logic clk);
  logic [7:0] addr;
  logic [7:0] data;
  bus_cmd_e cmd;
  modport TARGET
   (input  addr, cmd, data,
    import task targetRcv (output bus_cmd_e c,
                                    logic [7:0] a, d));
   modport INITIATOR
    (output addr, cmd, data,
     import task initiatorSend(input bus_cmd_e c,
                                    logic [7:0] a, d)
                );

  // Parallel send
  task initiatorSend(input bus_cmd_e c,
                          logic [7:0] a, d);
    @(posedge clk);
    cmd <= c;
    addr <= a;
    data <= d;
  endtask

  // Parallel receive
  task targetRcv(output bus_cmd_e c, logic [7:0] a, d);
    @(posedge clk);
    a = addr;
    d = data;
    c = cmd;
  endtask

endinterface: simple_if
```

10.4.2   Interface with serial protocol

The interface in Example 10-15 implements a serial interface for sending and receiving the address and data values. It has the same interface and routine names as Example 10-14, so you can swap between the two without having to change any design or testbench code.

Example 10-15 Interface with tasks for serial protocol

```
interface simple_if(input logic clk);
  logic addr;
  logic data;
  logic start = 0;
  bus_cmd_e cmd;

modport TARGET
    (input  addr, cmd, data,
     import task targetRcv (output bus_cmd_e c,
                            logic [7:0] a, d));
modport INITIATOR
    (output addr, cmd, data,
     import task initiatorSend(input bus_cmd_e c,
                               logic [7:0] a, d)
               );

  // Serial send
  task initiatorSend(input bus_cmd_e c,
                           logic [7:0] a, d);
    @(posedge clk);
    start <= 1;
    cmd <= c;
    foreach (a[i]) begin
      addr <= a[i];
      data <= d[i];
      @(posedge clk);
      start <= 0;
    end
    cmd <= IDLE;
  endtask

  // Serial receive
  task targetRcv(output bus_cmd_e c, logic [7:0] a, d);
    @(posedge start);
    c = cmd;
    foreach (a[i]) begin
      @(posedge clk);
      a[i] = addr;
      d[i] = data;
    end
  endtask

endinterface: simple_if
```

10.4.3   Limitations of interface code

Tasks in interfaces are fine for RTL, where the functionality is strictly defined. But these tasks are a poor choice for any type of verification IP. Interfaces and their code cannot be extended, overloaded, or dynamically instantiated based on configuration. Any code for verification needs maximum flexibility and configurability, and so should go in classes in a program block.

10.5   Conclusion

The interface construct in SystemVerilog provides a powerful technique to group together the connectivity, timing, and functionality for the communication between blocks. In this chapter you saw how you can create a single testbench that connects to many different design configurations containing multiple interfaces. Your signal layer code can connect to a variable number of physical interfaces at run-time with virtual interfaces. Additionally, an interface can contain the procedural code that drives the signals and assertions to check the protocol.

In many ways, an interface can resemble a class with pointers, encapsulation, and abstraction. This lets you create an interface to model your system at a higher level than Verilog's traditional ports and wires. Just remember to keep the testbench in the program block.

References

Bergeron, Janick. *Writing Testbenches Using SystemVerilog*. Norwell, MA: Springer, 2006

Bergeron, Janick, Cerny, Eduard, Hunter, Alan, and Nightingale, Andrew. *Verification Methodology Manual for SystemVerilog*. Norwell, MA: Springer, 2005

Cohen, Ben, Venkataramanan, Srinivasan, and Kumari, Ajeetha. *System-Verilog Assertions Handbook for Formal and Dynamic Verification*: VhdlCohen Publishing 2005

Cummings, Cliff. *Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!* Synopsys User Group, San Jose, CA, 2000

Denning, Peter. *The Locality Principle*, Communications of the ACM, 48(7), July 2005, pp. 19–24

Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley 1995

IEEE *IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language*. New York: IEEE 2005 (a.k.a. SystemVerilog Language Reference Manual, or LRM.)

IEEE *IEEE Standard Verilog Hardware Design, Description Language*. New York: IEEE 2001

Sutherland, Stuart, Davidmann, Simon, and Flake, Peter. *SystemVerilog for Design*. Norwell, MA: Kluwer Academic Publishers, 2004

Synopsys, Inc., *Hybrid RTL Formal Verification Ensures Early Detection of Corner-Case Bugs*, http://synopsys.com/products/magellan/magellan_wp.html, 2003

van der Schoot, Hans, and Bergeron, Janick *Transaction-Level Functional Coverage in SystemVerilog*. San Jose, CA: DVCon, February 2006

Vijayaraghavan, Srikanth, and Ramanathan, Meyyappan. *A Practical Guide for SystemVerilog Assertions.* Norwell, MA: Springer, 2005

Wachowski Andy, and Wachowski Larry. *The Matrix.* Hollywood, CA: Warner Brothers Studios, 1999

Index