Writing Testbenches Using SystemVerilog

Almost a hundred years later, after two collapses, the Quebec Bridge is still the longest cantilever bridge in the world. The cantilever technology has been replaced by the better, faster, cheaper suspension technology. SystemVerilog can similarly replace HDLs for better, faster, cheaper verification.

by Janick Bergeron

Springer

Writing Testbenches using SystemVerilog

Writing Testbenches using SystemVerilog

by
Janick Bergeron
Synopsys, Inc.

Janick Bergeron
Verificationguild.com

Writing Testbenches Using SystemVerilog

Library of Congress Control Number: 2005938214

ISBN-10: 0-387-29221-7 ISBN-10: 0-387-31275-7 (e-book)
ISBN-13: 9780387292212 ISBN-13: 9780387312750 (e-book)

Printed on acid-free paper.

© 2006 Springer Science+Business Media, Inc.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, Inc., 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

springer.com

TABLE OF CONTENTS

ABOUT THE COVER

The cover of the first edition of *Writing Testbenches* featured a photograph of the collapse of the Quebec bridge (the cantilever steel bridge on the left[1]) in 1907. The ultimate cause of the collapse was a major change in the design specification that was not verified. To save on construction cost, the engineer in charge of the project increased the span of the bridge from 1600 to 1800 feet, turning the project into the longest bridge in the world, without recalculating weights and stresses.

In those days, engineers felt they could span any distances, as ever longer bridges were being successfully built. But each technology eventually reaches its limits. Almost 100 years after its completion in 1918 (after a complete re-design and a second collapse!), the Quebec bridge is *still* the longest cantilever bridge in the world. Even with all of the advances in civil engineering and composite material, cantilever bridging technology had reached its limits.

You cannot realistically hope to keep applying the same solution to ever increasing problems. Even an evolving technology has its limit. Eventually, you will have to face and survive a revolution that will provide a solution that is faster and cheaper.

1. Photo: © Rock Santerre, Centre de Recherche en Géomatique, Université Laval, Québec City, Québec, Canada

Replacing the Quebec bridge with another cantilever structure is estimated to cost over $600 million today. When it was decided to span the St-Lawrence river once more in 1970, the high cost of a cantilever structure caused a different technology to be used: a suspension bridge. The Pierre Laporte Bridge, visible on the right, has a span of 2,200 feet and was built at a cost of $45 million. It provides more lanes of traffic over a longer span at a lower cost and weight. It is better, faster and cheaper. The suspension bridge technology has replaced cantilever structures in all but the shortest spans.

Directed testcases, as described in the first edition, were the cantilever bridges of verification. Coverage-driven constrained-random transaction-level self-checking testbenches are the suspension bridges. This methodology revolution, introduced by hardware verification languages such as *e* and OpenVera and as described in the second edition of *Writing Testbenches*, make verifying a design better, faster and cheaper. Hardwave verification languages have demonstrated their productivity in verifying today's multi-million gate designs.

SystemVerilog brings the HVL technology to the masses, as a true industry standard, with consistent syntax and simulation semantics and built in the simulators you already own. It is no longer necessary to acquire additional tools nor integrate different languages. Like the Pierre Laporte Bridge, which today carries almost all the traffic across the river, you should use these productive methods for writing the majority of your testbenches.

I'm hoping, with this new book, to facilitate your transition from ad-hoc, directed testcase verification to a state-of-the-art verification methodology using a language you probably have at your fingertip.

PREFACE

If you survey hardware design groups, you will learn that between 60% and 80% of their effort is dedicated to verification. This may seem unusually large, but I include in "verification" all debugging and correctness checking activities, not just writing and running testbenches. Every time a hardware designer pulls up a waveform viewer, he or she performs a verification task. With today's ASIC and FPGA sizes and geometries, getting a design to fit and run at speed is no longer the main challenge. It is to get the *right* design, working as *intended*, at the *right* time.

Unlike synthesizable coding, there is no particular coding style nor language required for verification. The freedom of using any language that can be interfaced to a simulator and of using any features of that language has produced a wide array of techniques and approaches to verification. The continued absence of constraints and historical shortage of available expertise in verification, coupled with an apparent under-appreciation of and under-investment in the verification function, has resulted in several different ad hoc approaches. The consequences of an informal, ill-equipped and understaffed verification process can range from a non-functional design requiring several re-spins, through a design with only a subset of the intended functionality, to a delayed product shipment.

WHY THIS BOOK IS IMPORTANT

In 2000, the first edition of *Writing Testbenches* was the first book specifically devoted to functional verification techniques for hardware models. Since then, several other verification-only books have appeared. Major conferences include verification tracks. Universities, in collaboration with industry, are offering verification courses in their engineering curriculum. Pure verification EDA companies are offering tools to improve productivity and the overall design quality. Some of these pure verification EDA companies have gone public or have been acquired, demonstrating that there is significant value in verification tools and IP. All of these contribute to create a formal body of knowledge in design verification. Such a body of knowledge is an essential foundation to creating a science of verification and fueling progress in methodology and productivity.

In 2003, the second edition presented the latest verification techniques that were successfully being used to produce fully functional first-silicon ASICs, systems-on-a-chip (SoC), boards and entire systems. It built on the content of the first edition—transaction-level self-checking testbenches—to introduce a revolution in functional verification: coverage-driven constrained-random verification using proprietary hardware verification languages.

This book is not really a new edition of the previous *Writing Testbenches* books. Nor is it a completely new book. I like to think of it as the 2½ edition. This book presents the same concepts as the second edition. It simply uses SystemVerilog as the sole implementation vehicle. The languages used in the second edition are still available. Therefore it is still a useful book on its own.

WHAT THIS BOOK IS ABOUT

I will first introduce the necessary concepts and tools of verification, then I'll describe a process for planning and carrying out an effective functional verification of a design. I will also introduce the concept of coverage models that can be used in a coverage-driven verification process.

It will be necessary to cover some SystemVerilog language semantics that are often overlooked in textbooks focused on describing the synthesizable subset or unfamiliar to traditional Verilog users.

These unfamiliar semantics become important in understanding what makes a well-implemented and robust testbench and in providing the necessary control and monitor features.

I will also present techniques for applying stimulus and monitoring the response of a design, by abstracting the physical-level transactions into high-level procedures using bus-functional models. The architecture of testbenches built around these bus-functional models is important to create a layer of abstraction relevant to the function being verified and to minimize development and maintenance effort. I also show some strategies for making testbenches self-checking.

Creating random testbenches involves more than calling the $*random* system task. I will show how random stimulus generators, built on top of bus-functional models, can be architected and designed to be able to produce the desired stimulus patterns. Random generators must be easily externally constrained to increase the likelihood that a set of interesting patterns will be generated.

Transaction-level modeling is another important concept presented in this book. It is used to parallelize the implementation and verification of a design and to perform more efficient simulations. A transaction-level model must adequately emulate the functionality of a design while offering orders of magnitudes in simulation performance over the RTL model. Striking the correct level of accuracy is key to achieving that performance improvement.

This book has one large omission: assertions and formal verification. It is not that they are not important. SystemVerilog includes constructs and semantics for writing assertions and coverage properties using temporal expressions. Formal verification is already an effective methodology for verifying certain classes of designs. It is simply a matter of drawing a line somewhere. There are already books on assertions[1] or formal verification. This book focuses on the bread-and-butter of verification for the foreseeable future: dynamic functional verification using testbenches

1. Cohen, Venkataramanan and Kumari, "*SystemVerilog Assertion Handbook*", VhdlCohen Publishing, 2005

WHAT PRIOR KNOWLEDGE YOU SHOULD HAVE

This book focuses on the functional verification of hardware designs using SystemVerilog. I expect the reader to have at least a basic knowledge of VHDL, Verilog, OpenVera or *e*. Ideally, you should have experience in writing models and be familiar with running a simulation using any of the available VHDL or Verilog simulators. There will be no detailed description of language syntax or grammar. It may be a good idea to have a copy of a language-focused textbook or the *SystemVerilog Language Reference Manual* as a reference along with this book. I do not describe a synthesizable subset, nor limit the implementation of the verification techniques to using that subset. Verification is a complex task: The power of the SystemVerilog language will be used to its fullest.

I also expect that you have a basic understanding of digital hardware design. This book uses several hypothetical designs from various application domains (video, datacom, computing, etc.). How these designs are actually specified, architected and then implemented is beyond the scope of this book. The content focuses on the specification, architecture, then implementation of the *verification* of these same designs.

Once you are satisfied with the content of this book and wish to put it in practice, I recommend you pick up a copy of the *Verification Methodology Manual for SystemVerilog*[1] (VMM). It is a book I co-authored and wrote as a series of very specific guidelines on how to implement testbenches using SystemVerilog. It uses all of the powerful concepts introduced here. It also includes a set of base classes that implements generic functionality that every testbench needs. Why re-invent the wheel? I will refer to the first edition of the VMM at relevant points in this book where further techniques, guidelines or support can be found.

READING PATHS

You should really read this book from cover to cover. However, if you are pressed for time, here are a few suggested paths.

1. Janick Bergeron, Eduard Cerny, Alan Hunter and Andrew Nightingale, "*Verification Methodology Manual for SystemVerilog*", Springer 2005, ISBN 0-387-25538-9

If you are using this book as a university or college textbook, you should focus on Chapter 4 through 6 and Appendix A. If you are a junior engineer who has only recently joined a hardware design group, you may skip Chapters 3 and 7. But do not forget to read them once you have gained some experience.

Chapters 3 and 6, as well as Appendix A, will be of interest to a senior engineer in charge of defining the verification strategy for a project. If you are an experienced designer, you may wish to skip ahead to Chapter 3.

If you have a software background, Chapter 4 and Appendix A may seem somewhat obvious. If you have a hardware design and RTL coding mindset, Chapters 4 and 7 are probably your best friends.

If your responsibilities are limited to managing a hardware verification project, you probably want to concentrate on Chapter 3, Chapter 6 and Chapter 7.

WHY SYSTEMVERILOG?

SystemVerilog is the first truly industry-standard language to cover design, assertions, transaction-level modeling and coverage-driven constrained random verification.

VHDL and Verilog

VHDL and Verilog have shown to be inadequate for verification. Their lack of—or poor—support for high-level data types, object oriented programming, assertions, functional coverage and declarative constraints has prompted the creation of specialized languages for each or all of these areas. Using separate languages creates integration challenges: they may use a different syntax for the same concepts, have different semantics, introduce race conditions and render the simulation less efficient. SystemVerilog unifies all of these areas under a consistent syntax, coherent semantics, with minimal race conditions and with global optimization opportunities.

In my experience, Verilog is a much abused language. It has the reputation for being easier to learn than VHDL, and to the extent that the initial learning curve is not as steep, it is true. SystemVer-

ilog, being a superset of Verilog, benefits from the same smooth initial learning curve. However—like VHDL—Verilog and SystemVerilog provide similar concepts: sequential statements, parallel constructs, structural constructs and the illusion of parallelism.

These concepts *must* be learned. Because of its lax requirements, Verilog lulls the user into a false sense of security. The user believes that he or she knows the language because there are no syntax errors or because the simulation results appear to be correct. Over time, and as a design grows, race conditions and fragile code structures become apparent, forcing the user to learn these important concepts. Languages have the same *area under the learning curve*. VHDL's is steeper but Verilog's goes on for much longer. Some sections in this book take the reader farther down the Verilog learning curve.

SystemVerilog continues in Verilog's footstep. Does that mean that VHDL is dead? Technically, all VHDL capabilities are directly available in SystemVerilog. And with the many capabilities available in SystemVerilog not available in VHDL, there are no longer any *technical* reasons to use VHDL[1]. However, that decision is never a purely technical one. There will be VHDL legacy code for years to come. And companies with large VHDL legacies will continue to use it, if only in a co-simulation capacity. There is also an effort by the VHDL IEEE Working Group to add these missing capabilities to VHDL, known as VHDL-200x. Whether or not you wish—or can afford—to wait for a *me too* language is again a business decision.

Hardware Verification Languages

Hardware verification languages (HVLs) are languages that were specifically designed to implement testbenches efficiently and productively. Commercial solutions include OpenVera from Synopsys and *e* from Cadence. Open-source solutions include the SystemC Verification Library (SCV) from Cadence and Jeda from Juniper Networks. There are also a plethora of home-grown solutions based on Perl, SystemC, C++ or TCL. SystemVerilog includes all of the

1. Before anyone paints me as a Verilog bigot, I wish to inform my readers that I learned VHDL first and have always had a slight preference toward VHDL over Verilog.

features of a hardware verification language. When using SystemVerilog, it is no longer necessary to use a separate language for verification.

Making use of the verification features of SystemVerilog involves more than simply learning a new syntax. Although one can continue to use a Verilog-like directed methodology with SystemVerilog, using it appropriately requires a shift in the way verification is approached and testbenches are implemented. The directed verification strategy used with Verilog is the schematic capture of verification. Using SystemVerilog with a constraint-driven random verification strategy is the synthesis of verification. When used properly, it is an incredible productivity boost (see Figure 2-16 on page 56).

Does SystemVerilog mean that OpenVera, *e* and the other proprietary verification and assertion languages are going to die? The answer is a definite *no*. SystemVerilog was created by merging donated features and technologies from proprietary languages. It did not invent nor create anything new other than integrating these features and technologies under a consistent syntax and semantics. The objectives of SystemVerilog are to lower the adoption and ownership cost of verification tools and to create a broader marketplace through industry-wide support.

SystemVerilog accomplishes these objectives by being an industry standard. But the problem with industry standards is that they remain static while the semiconductor technologies continue to advance. SystemVerilog will more than meet the needs of 90 percent of the users for years to come. But leading edge users, who have adopted HVLs years ago and are already pushing their limits, will not be satisfied by SystemVerilog for very long. Proprietary tools and languages will continue to evolve with those leading edge projects. Hopefully, their trail-blazing efforts will be folded back into a future version of SystemVerilog, where their art can become science.

CODE EXAMPLES

A common complaint I received about the first edition was the lack of complete examples. You'll notice that in this book, like the first two, code samples are still provided only as excerpts. I fundamen-

tally believe that this is a better way to focus the reader's attention on the important point I'm trying to convey. I do not want to bury you under pages and pages of complete but dry (and ultimately mostly irrelevant) source code.

FOR MORE INFORMATION

If you want more information on topics mentioned in this book, you will find links to relevant resources in the book-companion Web site at the following URL:

`http://janick.bergeron.com/wtb`

In the *resources* area, you will find links to publicly available utilities, documents and tools that make the verification task easier. You will also find an errata section listing and correcting the errors that inadvertently made their way in this edition.[1]

ACKNOWLEDGEMENTS

My wife, Danielle, continued to give this book energy against its constant drain. Kyle Smith, my editor, once more gave it form from its initial chaos. Hans van der Schoot, Sean Smith and Chris Spear, my technical reviewers, gave it light from its stubborn darkness. The Wilson Beach, on the shore of Lac St-Jean, made me forget that I had just completed another book. And FrameMaker, my word processing software, once more reliably went where no Word had gone before!

I continue to thank the numerous reviewers from the previous two editions and readers who have reported errors in the published versions, without whom this edition would not have been as good.

I also thank my employer, Synopsys Inc., for supporting this book-writing hobby of mine and for supplying licenses for their VCS™ simulator.

1. If you know of a verification-related resource or an error in this book that is not mentioned in the Web site, please let me know via email at `janick@bergeron.com`. I thank you in advance.

VCS and Vera are trademarks of Synopsys Inc. All other trademarks are the property of their respective owners.

"Everyone knows debugging is twice as hard as writing a program in the first place"

- Brian Kernighan

"*Elements of Programming Style*"

1974

CHAPTER 1 WHAT IS VERIFICATION?

Verification is not a testbench, nor is it a series of testbenches. Verification is a *process* used to demonstrate that the intent of a design is preserved in its implementation. We all perform verification processes throughout our daily lives: balancing a checkbook, tasting a simmering dish, associating landmarks with symbols on a map. These are all verification processes.

In this chapter, I introduce the basic concepts of verification, from its importance and cost, to making sure you are verifying that you are implementing what you want. I present the differences between various verification approaches as well as the difference between testing and verification. I also show how verification is key to design reuse, and I detail the challenges of verification reuse.

WHAT IS A TESTBENCH?

The term "testbench" usually refers to simulation code used to create a predetermined input sequence to a design, then optionally to observe the response. Testbenches are implemented using SystemVerilog, but they may also include external data files or C routines.

Figure 1-1 shows how a testbench interacts with a *design under verification* (DUV). The testbench provides inputs to the design and watches any outputs. Notice how this is a completely closed system: no inputs or outputs go in or out. The testbench is effectively a model of the universe as far as the design is concerned. The verifi-

cation challenge is to determine what input patterns to supply to the design and what is the expected output of a properly working design when submitted to those input patterns.

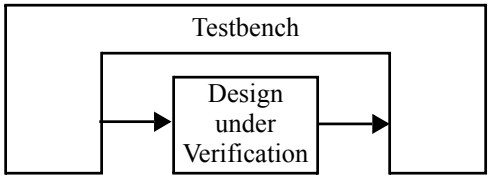


Figure 1-1. Generic structure of a testbench and design under verification

THE IMPORTANCE OF VERIFICATION

70% of design effort goes to verification.

Today, in the era of multi-million gate ASICs and FPGAs, reusable intellectual property (IP), and system-on-a-chip (SoC) designs, verification consumes about 70% of the design effort. Design teams, properly staffed to address the verification challenge, include engineers dedicated to verification. The number of verification engineers can be up to twice the number of RTL designers.

Verification is on the critical path.

Given the amount of effort demanded by verification, the shortage of qualified hardware design and verification engineers, and the quantity of code that must be produced, it is no surprise that, in all projects, verification rests squarely in the critical path. The fact that verification is often considered after the design has been completed, when the schedule has already been ruined, compounds the problem. It is also the reason verification is the target of the most recent tools and methodologies. These tools and methodologies attempt to reduce the overall verification time by enabling parallelism of effort, higher abstraction levels and automation.

Verification time can be reduced through parallelism.

If efforts can be parallelized, additional resources can be applied effectively to reduce the total verification time. For example, digging a hole in the ground can be parallelized by providing more workers armed with shovels. To parallelize the verification effort, it is necessary to be able to write—and debug—testbenches in parallel with each other as well as in parallel with the implementation of the design.

Verification time can be reduced through abstraction.

Providing higher abstraction levels enables you to work more efficiently without worrying about low-level details. Using a backhoe to dig the same hole mentioned above is an example of using a higher abstraction level.

Using abstraction reduces control over low-level details.

Higher abstraction levels are usually accompanied by a reduction in control and therefore must be chosen wisely. These higher abstraction levels often require additional training to understand the abstraction mechanism and how the desired effect can be produced. If a verification process uses higher abstraction levels by working at the transaction- or bus-cycle levels (or even higher ones), it will be easier to create large amount of stimulus. But it may be difficult to produce a specific sequence of low-level zeroes and ones. Similarly, using a backhoe to dig a hole suffers from the same loss-of-control problem: The worker is no longer directly interacting with the dirt; instead the worker is manipulating levers and pedals. Digging happens much faster, but with lower precision and only by a trained operator.

Lower levels of abstraction must remain visible.

It may be necessary to navigate between levels of abstraction. Verification can be accomplished using a bottom-up approach where the interface blocks and physical level details are verified first. The protocol levels can then be verified without having to worry about the physical signals. Verification can also be accomplished using a top-down approach where the protocol-level functionality is verified first using a transaction-level model without any physical-level interfaces. The details of the physical transport mechanisms can then be added later.

The transition between levels of abstraction may also occur dynamically during the execution of a testbench. The testbench may generally work at a high-level of abstraction, verifying the correctness of protocol-level operations. The testbench can then switch to a lower level of abstraction to inject a physical-level parity error to verify that protocol-level operations remain unaffected. Similarly, the backhoe operator can let a worker jump into the hole and use a shovel to uncover a gas pipeline.

Verification time can be reduced through automation.

Automation lets you do something else while a machine completes a task autonomously, faster and with predictable results. Automation requires standard processes with well-defined inputs and outputs. Not all processes can be automated. For example, holes must be dug in a variety of shapes, sizes, depths, locations and in varying

soil conditions, which render general-purpose automation impossible.

Verification faces similar challenges. Because of the variety of functions, interfaces, protocols and transformations that must be verified, it is not possible to provide a general purpose automation solution for verification, given today's technology. It is possible to automate some portion of the verification process, especially when applied to a narrow application domain. Tools automating various portions of the verification process are being introduced. For example, there are tools that will automatically generate bus-functional models from a higher-level abstract specification. Similarly, trenchers have automated digging holes used to lay down conduits or cables at shallow depths.

Randomization can be used as an automation tool.

For specific domains, automation can be emulated using randomization. By constraining a random generator to produce valid inputs within the bounds of a particular domain, it is possible to automatically produce almost all of the interesting conditions. For example, the tedious process of vacuuming the bottom of a pool can be automated using a broom head that, constrained by the vertical walls, randomly moves along the bottom. After a few hours, only the corners and a few small spots remain to be cleaned manually. This type of automation process takes more computation time to achieve the same result, but it is completely autonomous, freeing valuable resources to work on other critical tasks. Furthermore, this process can be parallelized[1] easily by concurrently running several random generators. They can also operate overnight, increasing the total number of productive hours.

RECONVERGENCE MODEL

The *reconvergence model* is a conceptual representation of the verification process. It is used to illustrate what exactly is being verified.

1. Optimizing these concurrent processes to reduce the amount of overlap is another question!

Do you know what you are actually verifying?

One of the most important questions you must be able to answer is: "*What are you verifying?*" The purpose of verification is to ensure that the result of some transformation is as intended or as expected. For example, the purpose of balancing a checkbook is to ensure that all transactions have been recorded accurately and confirm that the balance in the register reflects the amount of available funds.

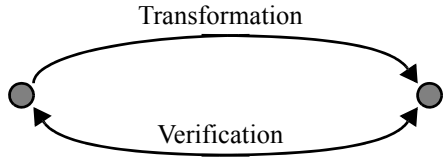


Figure 1-2. Reconvergent paths in verification

Verification is the reconciliation, through different means, of a specification and an output.

Figure 1-2 shows that verification of a transformation can be accomplished only through a second reconvergent path with a common source. The transformation can be any process that takes an input and produces an output. RTL coding from a specification, insertion of a scan chain, synthesizing RTL code into a gate-level netlist and layout of a gate-level netlist are some of the transformations performed in a hardware design project. The verification process reconciles the result with the starting point. If there is no starting point common to the transformation and the verification, *no* verification takes place.

The reconvergence model can be described using the checkbook example as illustrated in Figure 1-3. The common origin is the previous month's balance in the checking account. The transformation is the writing, recording and debiting of several checks during a one-month period. The verification reconciles the final balance in the checkbook register using this month's bank statement.

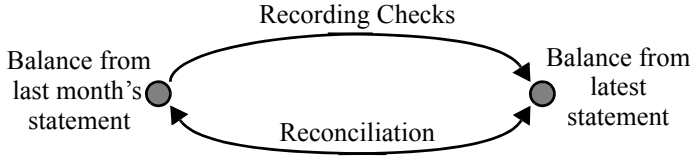


Figure 1-3. Balancing a checkbook is a verification process

THE HUMAN FACTOR

If the transformation process is not completely automated from end to end, it is necessary for an individual (or group of individuals) to

interpret a specification of the desired outcome and then perform the transformation. RTL coding is an example of this situation. A design team interprets a written specification document and produces what they believe to be functionally correct synthesizeable HDL code. Usually, each engineer is left to verify that the code written is indeed functionally correct.

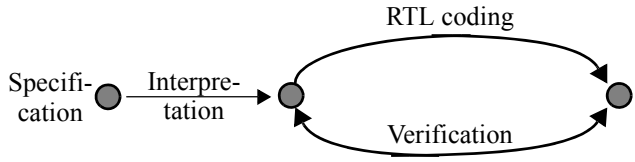


Figure 1-4. Reconvergent paths in ambiguous situation

Verifying your own design verifies against your interpretation, not against the specification.

Figure 1-4 shows the reconvergence model of the situation described above. RTL coding requires interpretation of a written specification. If the same person performs the verification, then the common origin is that interpretation, *not* the specification.

In this situation, the verification effort verifies whether the design accurately represents the *implementer's interpretation* of that specification. If that interpretation is wrong in any way, then this verification activity will never highlight it.

Any human intervention in a process is a source of uncertainty and unrepeatability. The probability of human-introduced errors in a process can be reduced through several complementary mechanisms: automation, *poka-yoke* or redundancy.

Automation

Eliminate human intervention.

Automation is the obvious way to eliminate human-introduced errors in a process. Automation takes human intervention completely out of the process. However, automation is not always possible, especially in processes that are not well-defined and continue to require human ingenuity and creativity, such as hardware design.

Poka-Yoke

Make human intervention foolproof.

Another possibility is to mistake-proof the human intervention by reducing it to simple, and foolproof steps. Human intervention is needed only to decide on the particular sequence or steps required

to obtain the desired results. This mechanism is also known as *poka-yoke* in Total Quality Management circles. It is usually the last step toward complete automation of a process. However, just like automation, it requires a well-defined process with standard transformation steps. The verification process remains an art that, to this day, does not lend itself to fool-proof steps.

Redundancy

Have two individuals check each other's work.

The final alternative to removing human errors is redundancy. It is the simplest, but also the most costly mechanism. Redundancy requires every transformation path to be duplicated. Every transformation accomplished by a human is either independently verified by another individual, or two complete and separate transformations are performed with each outcome compared to verify that both produced the same or equivalent output. This mechanism is used in high-reliability environments, such as airborne and space systems. It is also used in industries where later redesign and replacement of a defective product would be more costly than the redundancy itself, such as ASIC design.

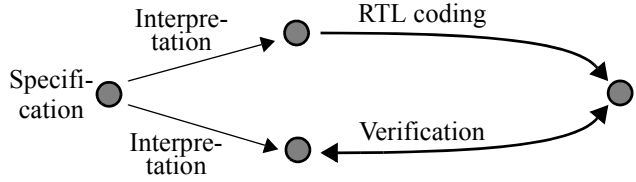


Figure 1-5. Redundancy in an ambiguous situation enables verification

A different person should verify.

Figure 1-5 shows the reconvergence model where redundancy is used to guard against misinterpretation of an ambiguous specification document. When used in the context of hardware design, where the transformation process is writing RTL code from a written specification document, this mechanism implies that a different individual must verify that implementation.

WHAT IS BEING VERIFIED?

Choosing the common origin and reconvergence points determines what is being verified. These origin and reconvergence points are often determined by the tool used to perform the verification. It is important to understand where these points lie to know which transformation is being verified. Formal verification, property checking,

functional verification, and rule checkers verify different things because they have different origin and reconvergence points.

Formal verification does not eliminate the need to write testbenches.

Formal verification is often misunderstood initially. Engineers unfamiliar with the formal verification process often imagine that it is a tool that mathematically determines whether their design is correct, without having to write testbenches. Once they understand what the end points of the formal verification reconvergent paths are, you know what exactly is being verified.

The application of formal verification falls under two broad categories: equivalence checking and property checking.

Equivalence Checking

Equivalence checking compares two models.

Figure 1-6 shows the reconvergence model for equivalence checking. This formal verification process mathematically proves that the origin and output are logically equivalent and that the transformation preserved its functionality.

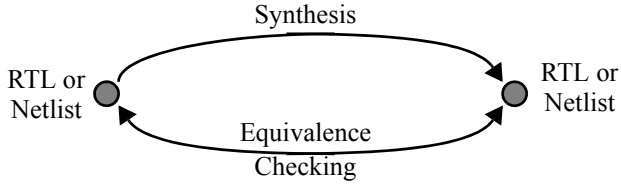


Figure 1-6. Equivalence checking paths

It can compare two netlists.

In its most common use, equivalence checking compares two netlists to ensure that some netlist post-processing, such as scan-chain insertion, clock-tree synthesis or manual modification[1], did not change the functionality of the circuit.

It can detect bugs in the synthesis software.

Another popular use of equivalence checking is to verify that the netlist correctly implements the original RTL code. If one trusted the synthesis tool completely, this verification would not be necessary. However, synthesis tools are large software systems that depend on the correctness of algorithms and library information. History has shown that such systems are prone to error. Equivalence checking is used to keep the synthesis tool honest. In some rare instances, this form of equivalence checking is used to verify

1. Text editors remain the greatest design tools!

that manually written RTL code faithfully represents a legacy gate-level design.

Equivalence checking can also verify that two RTL descriptions are logically identical. Proving their equivalence avoids running lengthy regression simulations when only minor non-functional changes are made to the source code to obtain better synthesis results. Modern equivalence checkers can even deal with sequential differences between RTL models, such as rearchitected FSMs or data pipelines.

Equivalence checking found a bug in an arithmetic operator.

Equivalence checking is a true alternative path to the logic synthesis transformation being verified. It is only interested in comparing Boolean and sequential logic functions, not mapping these functions to a specific technology while meeting stringent design constraints. Engineers using equivalence checking found a design at Digital Equipment Corporation (now part of HP) to be synthesized incorrectly. The design used a synthetic operator that was functionally incorrect when handling more than 48 bits. To the synthesis tool's defense, the documentation of the operator clearly stated that correctness was not guaranteed above 48 bits. Since the synthesis tool had no knowledge of documentation, it could not know it was generating invalid logic. Equivalence checking quickly identified a problem that could have been very difficult to detect using gate-level simulation.

Property Checking

Property checking proves assertions about the behavior of the design.

Property checking is a more recent application of formal verification technology. In it, assertions or characteristics of a design are formally proven or disproved. For example, all state machines in a design could be checked for unreachable or isolated states. A more powerful property checker may be able to determine if deadlock conditions can occur.

Another type of assertion that can be formally verified relates to interfaces. Using the SystemVerilog property specification language, assertions about the interfaces of a design are stated and the tool attempts to prove or disprove them. For example, an assertion

might state that, given that signal ALE will be asserted, then either the DTACK or ABORT signal will be asserted eventually.

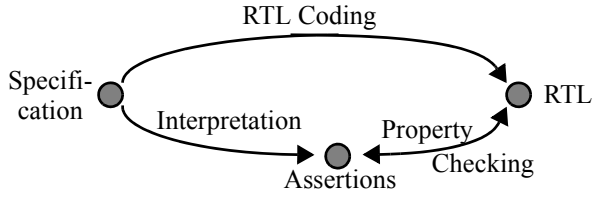


Figure 1-7. Property checking paths

Assertions must not be trivial.

The reconvergence model for property checking is shown in Figure 1-7 . The greatest obstacle for property checking technology is identifying, through interpretation of the design specification, which assertions to prove. Of those assertions, only a subset can be proven feasibly. Furthermore, for a proof to be useful, assertions must not be trivial restatements of the behavior already captured by the RTL code. They should be based on external requirements that the design must meet.

Assertions are good at checking temporal relationships of synchronous signals.

Current technology has difficulties proving high-level assertions about a design to ensure that complex functionality is correctly implemented. It would be nice to be able to concisely assert that, given specific register settings, a sequence of packets will end up at a set of outputs in some relative order. Unfortunately, there are two obstacles in front of this goal. First, property checking technology is limited in its capacity to deal with complex designs. Second, the assertion languages with formal semantics can efficiently describe cycle-based temporal relationships between low-level signals and simple transformations. Their inherent RTL or cycle-based nature makes it difficult to state high-level transformation properties.

Functional Verification

Functional verification verifies design intent.

The main purpose of functional verification is to ensure that a design implements intended functionality. As shown by the reconvergence model in Figure 1-8, functional verification reconciles a design with its specification. Without functional verification, one must trust that the transformation of a specification document into

RTL code was performed correctly, without misinterpretation of the specification's intent.

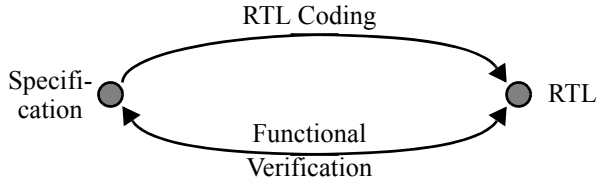


Figure 1-8. Functional verification paths

You can prove the presence of bugs, but you cannot prove their absence.

It is important to note that, unless a specification is written in a formal language with precise semantics,[1] it is *impossible* to prove that a design meets the intent of its specification. Specification documents are written using natural languages by individuals with varying degrees of ability in communicating their intentions. Any document is open to interpretation. One can easily prove that the design does *not* implement the intended function by identifying a single discrepancy. The converse, sadly, is not true: No one can prove that there are *no* discrepancies. Functional verification, as a process, can *show* that a design meets the intent of its specification. But it cannot *prove* it. Similarly, no one can prove that flying reindeers or UFOs do not exist. However, producing a single flying reindeer or UFO would be sufficient to prove the opposite!

FUNCTIONAL VERIFICATION APPROACHES

Functional verification can be accomplished using three complementary approaches: black-box, white-box and grey-box.

Black-Box Verification

Black-box verification cannot look at or know about the inside of a design.

With a black-box approach, functional verification is performed without any knowledge of the actual implementation of a design. All verification is accomplished through the available interfaces, without direct access to the internal state of the design, without knowledge of its structure and implementation. This method suffers from an obvious lack of visibility and controllability. It is often dif-

1. Even if such a language existed, one would eventually have to show that this description is indeed an accurate description of the design intent, based on some higher-level ambiguous specification.

ficult to set up an interesting state combination or to isolate some functionality. It is equally difficult to observe the response from the input and locate the source of the problem. This difficulty arises from the frequent long delays between the occurrence of a problem and the appearance of its symptom on the design's outputs.

Testcase is independent of implementation.

The advantage of black-box verification is that it does not depend on any specific implementation. Whether the design is implemented in a single ASIC, RTL code, transaction-level model, gates, multiple FPGAs, a circuit board or entirely in software, is irrelevant. A black-box functional verification approach forms a true conformance verification that can be used to show that a particular design implements the intent of a specification, regardless of its implementation. A set of black-box testbenches can be developed on a transaction-level model of the design and run, unmodified, on the RTL model of the design to demonstrate that they are equivalent. Black-box testbenches can be used as a set of *golden testbenches*.

My mother is a veteran of the black-box approach: To prevent us from guessing the contents of our Christmas gifts, she never puts any names on the wrapped boxes[1]. At Christmas, she has to correctly identify the content of each box, without opening it, so it can be given to the intended recipient. She has been known to fail on a few occasions, to the pleasure of the rest of the party!

In black-box verification, it is difficult to control and observe specific features.

The pure black-box approach is impractical in today's large designs. A multi-million gates ASIC possesses too many internal signals and states to effectively verify all of its functionality from its periphery. Critical functions, deep into the design, will be difficult to control and observe. Furthermore, a design fault may not readily present symptoms of a flaw at the outputs of the ASIC. For example, the black-box ASIC-level testbench in Figure 1-9 is used to verify a critical round-robin arbiter. If the arbiter is not completely fair in its implementation, what symptoms would be visible at the outputs? This type of fault could only be found through performance analysis using several long simulations to identify discrepancies between the actual throughput of a channel compared with its theoretical throughput. And a two percent discrepancy in a

1. To my wife's chagrin who likes shaking any box bearing her name.

channel throughput in three overnight simulations can be explained so easily as a simple statistical error...

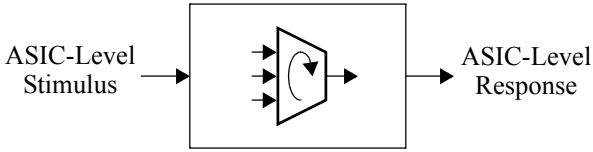


Figure 1-9. Black-box verification of a low-level feature

White-Box Verification

White box verification has intimate knowledge and control of the internals of a design.

As the name suggests, a white-box approach has full visibility and controllability of the internal structure and implementation of the design being verified. This method has the advantage of being able to set up an interesting combination of states and inputs quickly, or to isolate a particular function. It can then easily observe the results as the verification progresses and immediately report any discrepancies from the expected behavior.

White-box verification is tied to a specific implementation.

However, this approach is tightly integrated with a particular implementation. Changes in the design may require changes in the testbench. Furthermore, those testbenches cannot be used in gate-level simulations, on alternative implementations or future redesigns. It also requires detailed knowledge of the design implementation to know which significant conditions to create and which results to observe.

White-box techniques can augment black-box approaches.

White-box verification is a useful complement to black-box verification. This approach can ensure that low-level implementation-specific features behave properly, such as counters rolling over after reaching their end count value or datapaths being appropriately steered and sequenced. The white-box approach can be used only to verify the correctness of the functionality, while still relying on the black- or grey-box stimulus. Assertions are ideal for implementing white-box checks in RTL code. For example, Figure 1-10 shows the black-box ASIC-level environment shown in Figure 1-9 augmented with assertions to verify the functional correctness of the round-robin arbiter. Should fairness not be implemented correctly, the white-box checks would immediately report a failure. The reported error would also make it easier to identify and confirm

the cause of the problem, compared to a two percent throughput discrepancy.

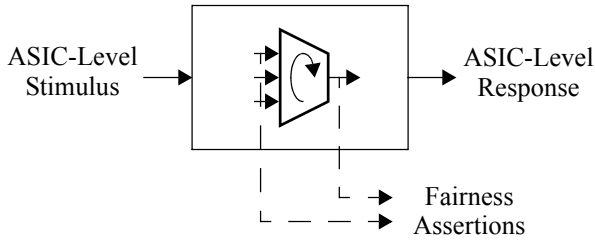


Figure 1-10. White-box checks in black-box environment

Checkered-box is used in system-level verification.

A checkered-box verification approach is often used on SoC design and system-level verification. A system is defined as a design composed of independently designed and verified components. The objective of system-level verification is to verify the system-level features, not re-verify the individual components. Because of the large number of possible states and the difficulty in setting up interesting conditions, system-level verification is often accomplished by treating it as a collection of black-boxes. The independently-designed components are treated as black-boxes, but the system itself is treated as a white-box, with full controllability and observability.

Grey-Box Verification

Grey-box verification is a compromise between the aloofness of a black-box verification and the dependence on the implementation of white-box verification. The former may not fully exercise all parts of a design, while the latter is not portable.

Testcase may not be relevant on another implementation.

As in black-box verification, a grey-box approach controls and observes a design entirely through its top-level interfaces. However, the particular verification being accomplished is intended to exercise significant features specific to the implementation. The same verification of a different implementation would be successful, but the verification may not be particularly more interesting than any other black-box verification. A typical grey-box test case is one written to increase coverage metrics. The input stimulus is designed to execute specific lines of code or create a specific set of conditions in the design. Should the structure (but not the function)

of the design change, this test case, while still correct, may no longer contribute toward better coverage.

Add functions to the design to increase controllability and observability

A typical grey-box strategy is to include some non-functional modifications to provide additional visibility and controllability. Examples include additional software-accessible registers to control or observe internal states, speed up a real-time counter, force the raising of exceptions or modify the size of the processed data to minimize verification time. These registers and features would not be used during normal operations, but they are often valuable during the integration phase of the first prototype systems.

Verification must influence the design.

For non-functional features required by the verification process to exist in a design, verification must be considered as an integral part of a design. When architecting a design, the verifiability of that architecture must be assessed at the same time. If some architectural features promise to be difficult to verify or exercise, additional observability or controllability features must be added. This process is called *design-for-verification*.

White-box cannot be used in parallel with design.

The black-box and grey-box approaches are the only ones that can be used if the functional verification is to be implemented in parallel with the implementation using a transaction-level model of the design (see "Transaction-Level Models" on page 333). Because there is no detailed implementation to know about beforehand, these two verification strategies are the only possible avenue.

TESTING VERSUS VERIFICATION

Testing verifies manufacturing.

Testing is often confused with verification. The purpose of the former is to verify that the design was manufactured correctly. The purpose of the latter is to ensure that a design meets its functional intent.

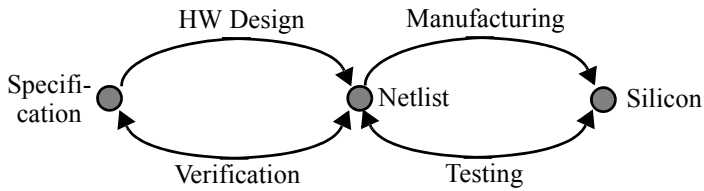


Figure 1-11. Testing vs. Verification

Figure 1-11 shows the reconvergence models for both verification and testing. During testing, the finished silicon is reconciled with the netlist that was submitted for manufacturing.

Testing verifies that internal nodes can be toggled.

Testing is accomplished through test vectors. The objective of these test vectors is not to exercise functions. It is to exercise physical locations in the design to ensure that they can go from 0 to 1 and from 1 to 0 and that the change can be observed. The ratio of physical locations tested to the total number of such locations is called *test coverage*. The test vectors are usually automatically generated to maximize coverage while minimizing vectors through a process called *automatic test pattern generation* (ATPG).

Thoroughness of testing depends on controllability and observability of internal nodes.

Testing and test coverage depends on the ability to set internal physical locations to either 1 or 0, and then observe that they were indeed appropriately set. Some designs have very few inputs and outputs, but these designs have a large number of possible states, requiring long sequences to observe and control all internal physical locations properly. A perfect example is an electronic wristwatch: It has three or four inputs (the buttons around the dial) and a handful of outputs (the digits and symbols on the display). However, if it includes chronometer and calendar functions, it has billions of possible state combinations (hundreds of years divided into milliseconds). At speed, it would take hundreds of years to take such a design through all of its possible states.

Scan-Based Testing

Linking all registers into a long shift register increases controllability and observability.

Fortunately, scan-based testability techniques help reduce this problem to something manageable. With scan-based tests, all registers inside a design are hooked-up in a long serial chain. In normal mode, the registers operate as if the scan chain was not there (see Figure 1-12(a)). In scan mode, the registers operate as a long shift register (see Figure 1-12(b)).

To test a scannable design, the unit under test is put into scan mode, then an input pattern is shifted through all of its internal registers. The design is then put into normal mode and a *single* clock cycle is applied, loading the result of the normal operation based on the scanned state into the registers. The design is then put into scan mode again. The result is shifted out of the registers (at the same

time the next input pattern is shifted in), and the result is compared against the expected value.

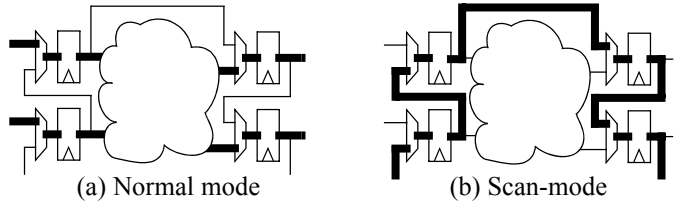


Figure 1-12. Scan-based testing

Scan-based testing puts restrictions on design.

This increase in controllability and observability, and thus test coverage, comes at a cost. Certain restrictions, called *design-for-testability*, are put onto the design to enable the insertion of a scan chain and the automatic generation of test patterns. These restrictions include, but are not limited to: fully synchronous design, no derived or gated clocks and use of a single clock edge. The topic of design-for-testability is far greater and complex than this simple introduction implies. For more details, books and papers specializing on the subject should be consulted.

The benefits of scan-based testing far outweighs the drawbacks of these restrictions.

Hardware designers introduced to scan-based testing initially rebel against the restrictions imposed on them. They see only the immediate area penalty and their favorite design technique rendered illegal. However, the increased area and additional design effort are quickly outweighed when a design can be fitted with one or more scan chains, when test patterns are generated and high test coverage is achieved automatically, at the push of a button. The time saved and the greater confidence in putting a working product on the market far outweighs the added cost for scan-based design.

Design for Verification

Design practices need to be modified to accommodate testability requirements. Isn't it acceptable to modify those same design practices to accommodate verification requirements?

Verification must be considered during specification.

With functional verification requiring more effort than design, it is reasonable to require additional design effort to simplify verification. Just as scan chains are put in a design to improve testability without adding to the functionality, it should be standard practice to add non-functional structures and features to facilitate verification. This approach requires that verification be considered at the outset

of a project, during its specification phase. Not only should the architect of the design answer the question, "What is this supposed to do?" but also, "How is this thing going to be verified?"

Typical design-for-verification techniques include well-defined interfaces, clear separation of functions in relatively independent units, providing additional software-accessible registers to control and observe internal locations and providing programmable multiplexers to isolate or bypass functional units.

DESIGN AND VERIFICATION REUSE

Today, design reuse is a fact of life. It is the best way to overcome the difference between the number of transistors that can be manufactured on a single chip and the number of transistors engineers can take advantage of in a reasonable amount of time. This difference is called the *productivity gap*. Design reuse was originally thought to be a simple concept that would be easy to put in practice. The reality proved—and continues to prove—to be more problematic.

Reuse Is About Trust

You won't use what you do not trust.

The major obstacle to design reuse is cultural. Engineers have little incentive and willingness to incorporate an unknown design into their own. They do not trust that the other design is as good or as reliable as one designed by themselves. The key to design reuse is gaining that trust.

Proper functional verification demonstrates trustworthiness of a design.

Trust, like quality, is not something that can be added to a design after the fact. It must be built-in, through the best possible design practices. Trustworthiness can be demonstrated through a proper verification process. By showing the user that a design has been thoroughly and meticulously verified according to the design specification, trust can be built and communicated much faster. Functional verification is the only way to demonstrate that the design meets, or even exceeds, the quality of a similar design that an engineer could do himself or herself.

Verification for Reuse

Reusable designs must be verified to a greater degree of confidence.

If you create a design, you have a certain degree of confidence in your own abilities as a designer and implicitly trust its correctness. Functional verification is used only to confirm that opinion and to augment that opinion in areas known to be weak. If you try to reuse a design, you can rely only on the functional verification to build that same level of confidence and trust. Thus, reusable designs must be verified to a greater degree of confidence than custom designs.

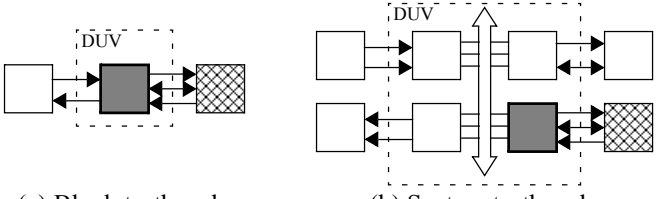
All claims, possible configurations and uses must be verified.

Because reusable designs tend to be configurable and programmable to meet a variety of possible environments and applications, it is necessary to verify a reusable design under all possible configurations and for all possible uses. All claims made about the reusable design must be verified and demonstrated to users.

Verification Reuse

Testbench components can be reused also.

If portions of a design can be reused, portions of testbenches can be reused as well. For example, Figure 1-13 shows that the bus-functional model used to verify a design block (a) can be reused to verify the system that uses it (b).



(a) Block testbench (b) System testbench

Figure 1-13. Reusing BFMs from block-level testbench in system-level testbench

Verification reuse has its challenges.

There are degrees of verification reuse, some easier to achieve, others facing difficulties similar to design reuse. Reusing BFMs across different testbenches and test cases for the same design is a simple process of properly architecting a verification environment. Reusing testbench components or test cases in a subsequent revision of the same design presents some difficulties in introducing the verification of the new features. Reusing a testbench component between two different projects or between two different levels of abstraction has many challenges that must be addressed when designing the component itself.

Salvaging is not reuse.

Salvaging is reusing a piece of an existing testbench that was not expressly designed to be reused. The suitability of the salvaged component will vary greatly depending on the similarities between the needs of the design to be verified and those of the original design. For example, a BFM that was designed to verify an interface block (as in Figure 1-13 (a)) may not be suitable for verifying a system using that interface block.

Block- and system-level testbenches put different requirements on a BFM.

Block-level verification must exercise the state machines and decoders used in implementing the interface protocol. This verification requires a transaction-level BFM with detailed controls of the protocol signals to vary timing or inject protocol errors. However, the system-level verification must exercise the high-level functionality that resides behind the interface block. This verification requires the ability to encapsulate high-level data onto the interface transactions. The desired level of controllability resides at a much higher level than the signal-level required to verify the interface block.

THE COST OF VERIFICATION

Verification is a necessary evil. It always takes too long and costs too much. Verification does not directly generate a profit or make money: After all, it is the design being verified that will be sold and ultimately make money, not the verification. Yet verification is indispensable. To be marketable and create revenues, a design must be functionally correct and provide the benefits that the customer requires.

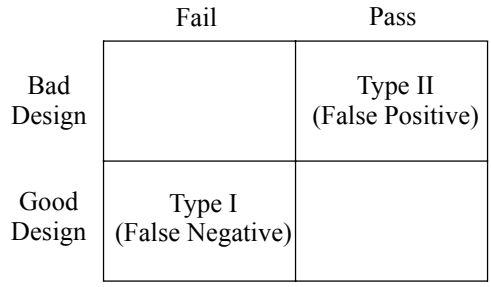
As the number of errors left to be found decreases, the time—and cost—to identify them increases.

Verification is a process that is never truly complete. The objective of verification is to ensure that a design is error-free, yet one cannot prove that a design is error-free. Verification can show only the *presence* of errors, not their *absence*. Given enough time, an error *will* be found. The question thus becomes: Is the error likely to be severe enough to warrant the effort spent identifying it? As more and more time is spent on verification, fewer and fewer errors are found with a constant incremental effort expenditure. As verification progresses, it has diminishing returns. It costs more and more to find each remaining error.

Functional verification is similar to statistical hypothesis testing. The hypothesis under test is: "*Is my design functionally correct?*"

The answer can be either *yes* or *no*. But either answer could be wrong. These wrong answers are Type II and Type I mistakes, respectively.

Figure 1-14. Type I & II mistakes



False positives must be avoided.

Figure 1-14 shows where each type of mistake occurs. Type I mistakes, or *false negatives*, are the easy ones to identify. The verification is finding an error where none exist. Once the misinterpretation is identified, the implementation of the verification is modified to change the answer from “no” to "yes," and the mistake no longer exists. Type II mistakes are the most serious ones: The verification failed to identify an error. In a Type II mistake, or *false positive* situation, a bad design is shipped unknowingly, with all the potential consequences that entails.

Shipping a bad design may result in simple product recall or in the total failure of a space probe after it has landed on another planet. Similarly, drug companies faces Type II mistakes on a regular basis with potentially devastating consequences: In spite of positive clinical test results, is a dangerous drug being released on the market?

With the future of the company potentially at stake, the 64-thousand dollar question in verification is: "*How much is enough?*" The functional verification process presented in this book, along with some of the tools described in the next chapter attempt to answer that question.

The 64-million dollar question is: "*When will I be done?*” Knowing where you are in the verification process, although impossible to establish with certainty, is much easier to estimate than how long it will take to complete the job. The verification planning process described in Chapter 3 creates a tool that enables a verification

manager to better estimate the effort and time required to complete the task at hand, to the degree of certainty required.

SUMMARY

Verification is a process, not a set of testbenches.

Verification can be only accomplished through an independent path between a specification and an implementation. It is important to understand where that independence starts and to know what is being verified.

Verification can be performed at various levels of the design hierarchy, with varying degrees of visibility within those hierarchies. I prefer a black-box approach because it yields portable testbenches. Augment with grey and white-box testbenches to meet your goals.

Consider verification at the beginning of the design. If a function would be difficult to verify, modify the design to give the necessary observability and controllability over the function.

Make your verification components reusable across different testbenches, across block and system-level testbenches and across different projects.

CHAPTER 2 VERIFICATION TECHNOLOGIES

As mentioned in the previous chapter, one of the mechanisms that can be used to improve the efficiency and reliability of a process is automation. This chapter covers technologies used in a state-of-the-art functional verification environment. Some of these technologies, such as simulators, are essential for the functional verification activity to take place. Others, such as linting or code coverage technologies, automate some of the most tedious tasks of verification and help increase the confidence in the outcome of the functional verification. This chapter does not contain an exhaustive list of verification technologies, as new application-specific and general purpose verification automation technologies are regularly brought to market.

It is not necessary to use all the technologies mentioned.

As a verification engineer, your job is to use the necessary technologies to ensure that the verification process does not miss a significant functional bug. As a project manager responsible for the delivery of a working product on schedule and within the allocated budget, your responsibility is to arm your engineers with the proper tools to do their job efficiently and with the necessary degree of confidence. Your job is also to decide when the cost of finding the next functional bug has increased above the value the additional functional correctness brings. This last responsibility is the heaviest of them all. Some of these technologies provide information to help you decide when you've reached that point.

A tool may include more than one technology.

This chapter presents various verification technologies separately from each other. Each technology is used in multiple EDA tools. A specific EDA tool may include more than one technology. For example, "super linting" tools leverage linting and formal technologies. Hybrid- or semi-formal tools use a combination of simulation and formal technologies.

Synopsys tools are mentioned.

Being a Synopsys employee at the time of writing this book, the commercial tools I mention are provided by Synopsys, Inc. Other EDA companies supply competitive products. All trademarks and service marks, registered or not, are the property of their respective owners.

LINTING

Linting technology finds common programmer mistakes.

The term "lint" comes from the name of a UNIX utility that parses a C program and reports questionable uses and potential problems. When the C programming language was created by Dennis Ritchie, it did not include many of the safeguards that have evolved in later versions of the language, like ANSI-C or C++, or other strongly-typed languages such as Pascal or ADA. *lint* evolved as a tool to identify common mistakes programmers made, letting them find the mistakes quickly and efficiently, instead of waiting to find them through a dreaded segmentation fault during execution of the program.

lint identifies real problems, such as mismatched types between arguments and function calls or mismatched number of arguments, as shown in Sample 2-1. The source code is syntactically correct and compiles without a single error or warning using gcc version 2.96.

However, Sample 2-1 suffers from several pathologically severe problems:

1. The *my_func* function is called with only one argument instead of two.
2. The *my_func* function is called with an integer value as a first argument instead of a pointer to an integer.

Problems are found faster than at runtime.

As shown in Sample 2-2, the *lint* program identifies these problems, letting the programmer fix them before executing the pro-

Sample 2-1. Syntactically correct K&R C source code

```
int my_func(addr_ptr, ratio)
   int   *addr_ptr;
   float ratio;
{
   return (*addr_ptr)++;
}

main()
{
   int my_addr;
   my_func(my_addr);
}
```

gram and observing a catastrophic failure. Diagnosing the problems at runtime would require a debugger and would take several minutes. Compared to the few seconds it took using *lint*, it is easy to see that the latter method is more efficient.

Sample 2-2. *Lint* output for Sample 2-1

```
src.c(3): warning: argument ratio unused in
function my_func
src.c(11): warning: addr may be used before set
src.c(12): warning: main() returns random value
to invocation environment
my_func: variable # of args.    src.c(4)  ::
src.c(11)
my_func, arg. 1 used inconsistently
src.c(4)  ::  src.c(11)
my_func returns value which is always ignored
```

Linting does not require stimulus.

Linting has a tremendous advantage over other verification technologies: It does not require stimulus, nor does it require a description of the expected output. It performs checks that are entirely static, with the expectations built into the linting tool itself.

The Limitations of Linting Technology

Linting can only identify a certain class of problems.

Other potential problems were also identified by *lint*. All were fixed in Sample 2-3, but *lint* continues to report a problem with the invocation of the *my_func* function: The return value is always ignored. Linting cannot identify all problems in source code. It can only find problems that can be statically deduced by looking at the code structure, not problems in the algorithm or data flow.

For example, in Sample 2-3, linting does not recognize that the uninitialized *my_addr* variable will be incremented in the *my_func* function, producing random results. Linting is similar to spell checking; it identifies misspelled words, but cannot determine if the wrong word is used. For example, this book could have several instances of the word “with” being used instead of “width”. It is a type of error the spell checker (or a linting tool) could not find.

Sample 2-3. Functionally correct K&R C source code

```
int my_func(addr_ptr)
   int *addr_ptr;
{
   return (*addr_ptr)++;
}

main()
{
   int my_addr;
   my_func(&my_addr);
   return 0;
}
```

Many false negatives are reported.

Another limitation of linting technology is that it is often too paranoid in reporting problems it identifies. To avoid letting an error go unreported, linting errs on the side of caution and reports potential problems where none exist. This results into a lot of false errors. Designers can become frustrated while looking for non-existent problems and may abandon using linting altogether.

Carefully filter error messages!

You should filter the output of linting tools to eliminate warnings or errors known to be false. Filtering error messages helps reduce the frustration of looking for non-existent problems. More importantly, it reduces the output clutter, reducing the probability that the report of a real problem goes unnoticed among dozens of false reports. Similarly, errors known to be true positive should be highlighted. *Extreme* caution must be exercised when writing such a filter: You must make sure that a true problem is not filtered out and never reported.

Naming conventions can help output filtering.

A properly defined naming convention is a useful technique to help determine if a warning is significant. For example, the report in Sample 2-4 about a latch being inferred on a signal whose name

ends with "_lat" could be considered as expected and a false warning. All other instances would be flagged as true errors.

Sample 2-4. Output from a hypothetical SystemVerilog linting tool

```
Warning: file decoder.sv, line 23: Latch
inferred on reg "address_lat".
Warning: file decoder.sv, line 36: Latch
inferred on reg "next_state".
```

Do not turn off checks.

Filtering the output of a linting tool is preferable to turning off checks from within the source code itself or via a command line option. A check may remain turned off for an unexpected duration, potentially hiding real problems. Checks that were thought to be irrelevant may become critical as new source files are added.

Lint code as it is being written.

Because it is better to fix problems when they are created, you should run *lint* on the source code while it is being written. If you wait until a large amount of code is written before linting it, the large number of reports—many of them false—will be daunting and create the impression of a setback. The best time to identify a report as true or false is when you are still intimately familiar with the code.

Enforce coding guidelines.

Linting, through the use of user-defined rules, can also be used to enforce coding guidelines and naming conventions[1]. Therefore, it should be an integral part of the authoring process to make sure your code meets the standards of readability and maintainability demanded by your audience.

Linting SystemVerilog Source Code

Linting SystemVerilog source code catches common errors.

Linting SystemVerilog source code ensures that all data is properly handled without accidentally dropping or adding to it. The code in Sample 2-5 shows a SystemVerilog model that looks perfect, compiles without errors, but eventually produces incorrect results.

Problems may not be obvious.

The problem lies with the use of the *byte* type. It is a signed 8-bit value. It will thus never be equal to or greater than 255 as specified in the conditional expressions. The counter will never saturate and

1. See Appendix A for a set of coding guidelines.

Sample 2-5. Potentially problematic SystemVerilog code

```
module saturated_counter(output done,
                         input  rst,
                         input  clk);

byte counter;
always_ff @(posedge clk)
begin
   if (rst) counter <= 0;
   else if (counter < 255) counter++;
end

assign done = (counter == 255);

endmodule
```

roll over instead. A simple change to "*bit [7:0]*" fixes the problem. But identifying the root cause may be difficult using simulation and waveforms. It is even possible that the problem will never be exercised because no testbench causes the counter to (normally) saturate or the correct effect of the saturation is never checked in the self-checking structure. Linting should report this discrepancy immediately and the bug would be fixed in a few minutes, without a single simulation cycle.

Linting may detect some race conditions.

Sample 2-6 shows another example. It is a race condition between two concurrent execution branches that will yield an unpredictable result (this race condition is explained in details in the section titled "Write/Write Race Conditions" on page 180). This type of error could be easily detected through linting.

Sample 2-6. Race condition in SystemVerilog code

```
begin
   integer i;
   ...
   fork
      i = 1;
      i = 0;
   join
   ...
end
```

Linting may be built in simulators.

SystemVerilog simulators may provide linting functionality. Some errors, such as race conditions, may be easier to identify during a simulation than through static analysis of the source code. The race condition in Sample 2-6 is quickly identified when using the *+race* command line option of VCS.

Linting tools may leverage formal technology.

Linting tools may use formal technologies to perform more complex static checks. Conversely, property checking tools may also provide some *lint*-like functionality. Some errors, such as unreachable lines of code or FSM transitions, require formal-like analysis of the conditions required to reach executable statements or states and whether or not these conditions can be produced.

Code Reviews

Reviews are performed by peers.

Although not technically linting, the objective of code reviews is essentially the same: Identify functional and coding style errors before functional verification and simulation. Linting can only identify questionable language uses. It cannot check if the intended behavior has been coded. In code reviews, the source code produced by a designer is reviewed by one or more peers. The goal is not to publicly ridicule the author, but to identify problems with the original code that could not be found by an automated tool. Reviews can identify discrepancies between the design intent and the implementation. They also provide an opportunity for suggesting coding improvements, such as better comments, better structure or the use of assertions.

Identify qualitative problems and functional errors.

A code review is an excellent venue for evaluating the maintainability of a source file, and the relevance of its comments and assertions. Other qualitative coding style issues can also be identified. If the code is well understood, it is often possible to identify functional errors or omissions.

Code reviews are not new ideas either. They have been used for many years in the software design industry. They have been shown to be the most effective quality-enhancing activity. Detailed information on how to conduct effective code reviews can be found in the *resources* section at:

```
http://janick.bergeron.com/wtb
```

SIMULATION

Simulate your design before implementing it.

Simulation is the most common and familiar verification technology. It is called “simulation” because it is limited to approximating reality. A simulation is never the final goal of a project. The goal of all hardware design projects is to create real physical designs that

can be sold and generate profits. Simulation attempts to create an artificial universe that mimics the future real design. This type of verification technology lets the designers interact with the design before it is manufactured, and correct flaws and problems earlier.

Simulation is only an approximation of reality.

You must never forget that a simulation is an approximation of reality. Many physical characteristics are simplified—or even ignored—to ease the simulation task. For example, a four-state digital simulation assumes that the only possible values for a signal are 0, 1, unknown, and high-impedance. However, in the physical—and analog—world, the value of a signal is a continuous function of the voltage and current across a thin aluminium or copper wire track: an infinite number of possible values. In a discrete simulation, events that happen deterministically 5 nanoseconds apart may be asynchronous in the real world and may occur randomly.

Simulation is at the mercy of the descriptions being simulated.

Within that simplified universe, the only thing a simulator does is execute a description of the design. The description is limited to a well-defined language with precise semantics. If that description does not accurately reflect the reality it is trying to model, there is no way for you to know that you are simulating something that is different from the design that will be ultimately manufactured. Functional correctness and accuracy of models is a big problem as errors cannot be proven *not* to exist.

Stimulus and Response

Simulation requires stimulus.

Simulation is not a static technology. A static verification technology performs its task on a design without any additional information or action required by the user. For example, linting and property checking are static technologies. Simulation, on the other hand, requires that you provide a facsimile of the environment in which the design will find itself. This facsimile is called a testbench. Writing this testbench is the main objective of this book. The testbench needs to provide a representation of the inputs observed by the design, so the simulation can emulate the design's responses based on its description.

The simulation outputs are validated externally, against design intents.

The other thing that you must not forget is that a simulation has no knowledge of your intentions. It cannot determine if a design being simulated is correct. Correctness is a value judgment on the outcome of a simulation that must be made by you, the engineer. Once the design is subjected to an approximation of the inputs from its

environment, your primary responsibility is to examine the outputs produced by the simulation of the design's description and determine if that response is appropriate.

Event-Driven Simulation

Simulators are never fast enough.

Simulators are continuously faced with one intractable problem: They are never fast enough. They are attempting to emulate a physical world where electricity travels at the speed of light and millions of transistors switch over one billion times in a second. Simulators are implemented using general purpose computers that can execute, under ideal conditions, in the order of a billion sequential instructions per second. The speed advantage is unfairly and forever tipped in favor of the physical world.

Outputs change only when an input changes.

One way to optimize the performance of a simulator is to avoid simulating something that does not need to be simulated. Figure 2-1 shows a 2-input XOR gate. In the physical world, even if the inputs do not change (Figure 2-1(a)), voltage is constantly applied to the output, current is continuously flowing through the transistors (in some technologies), and the atomic particles in the semiconductor are constantly moving. The *interpretation* of the electrical state of the output as a binary value (either a logic 1 or a logic 0) does not change. Only if one of the inputs change (as in Figure 2-1(b)), can the output change.

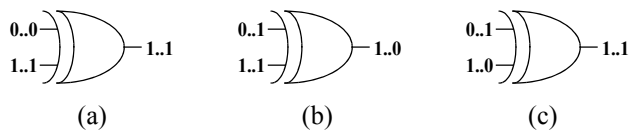


Figure 2-1. Behavior of an XOR gate

Change in values, called *events*, drive the simulation process.

Sample 2-7 shows a SystemVerilog description (or model) of an XOR gate. The simulator could choose to execute this model continuously, producing the same output value if the input values did not change. An opportunity to improve upon that simulator's performance becomes obvious: do not execute the model while the inputs are constants. Phrased another way: Only execute a model when an input changes. The simulation is therefore driven by

changes in inputs. If you define an input change as an *event*, you now have an *event-driven* simulator.

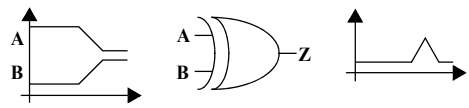
Sample 2-7. SystemVerilog model for an XOR gate

```
assign Z = A ^ B;
```

Sometimes, input changes do not cause the output to change.

But what if both inputs change, as in Figure 2-1(c)? In the logical world, the output does not change. What should an event-driven simulator do? For two reasons, the simulator should execute the description of the XOR gate. First, in the real world, the output of the XOR gate *does* change. The output might oscillate between 0 and 1 or remain in the "neither-0-nor-1" region for a few hundredths of picoseconds (see Figure 2-2). It just depends on how accurate you want your model to be. You could decide to model the XOR gate to include the small amount of time spent in the unknown (or x) state to more accurately reflect what happens when both inputs change at the same time**.**

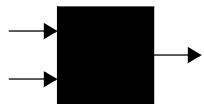
Figure 2-2. Behavior of an XOR gate when both inputs change



Descriptions between inputs and outputs are arbitrary.

The second reason is that the event-driven simulator does not know apriori that it is about to execute a model of an XOR gate. All the simulator knows is that it is about to execute a description of a 2-input, 1-output function. Figure 2-3 shows the view of the XOR gate from the simulator's perspective: a simple 2-input, 1-output black box. The black box could just as easily contain a 2-input AND gate (in which case the output might very well change if both inputs change), or a 1024-bit linear feedback shift register (LFSR).

Figure 2-3. Event-driven simulator view of an XOR gate



The mechanism of event-driven simulation introduces some limitations and interesting side effects that are discussed further in Chapter 4.

Acceleration options are often available in event-driven simulators

Simulation vendors are forever locked in a constant battle of beating the competition with an easier-to-use, faster simulator. It is possible to increase the performance of an event-driven simulator by simplifying some underlying assumptions in the design or in the simulation algorithm. For example, reducing delay values to identical unit delays or using two states (0 and 1) instead of four states (0, 1, x and z) are techniques used to speed-up simulation. You should refer to the documentation of your simulator to see what acceleration options are provided. It is also important to understand what the consequences are, in terms of reduced accuracy, of using these acceleration options.

Cycle-Based Simulation

Figure 2-4 shows the event-driven view of a synchronous circuit composed of a chain of three 2-input gates between two edge-triggered flip-flops. Assuming that Q1 holds a zero, Q2 holds a one and all other inputs remain constant, a rising edge on the clock input would cause an event-driven simulator to simulate the circuit as follows:

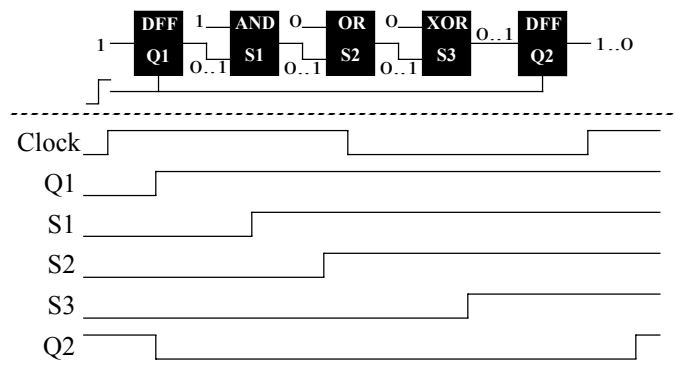


Figure 2-4. Event-driven simulator view of a synchronous circuit

1. The event (rising edge) on the clock input causes the execution of the description of the flip-flop models, changing the output value of Q1 to one and of Q2 to zero, after some small delay.

2. The event on Q1 causes the description of the AND gate to execute, changing the output S1 to one, after some small delay.
3. The event on S1 causes the description of the OR gate to execute, changing the output S2 to one, after some small delay.
4. The event on S2 causes the description of the XOR gate to execute, changing the output S3 to one after some small delay.
5. The next rising edge on the clock causes the description of the flip-flops to execute, Q1 remains unchanged, and Q2 changes back to one, after some small delay.

Many intermediate events in synchronous circuits are not functionally relevant.

To simulate the effect of a single clock cycle on this simple circuit required the generation of six events and the execution of seven models (some models were executed twice). If all we are interested in are the final states of Q1 and Q2, not of the intermediate combinatorial signals, then the simulation of this circuit could be optimized by acting only on the significant events for Q1 and Q2: the active edge of the clock. Phrased another way: Simulation is based on clock cycles. This is how cycle-based simulators operate.

The synchronous circuit in Figure 2-4 can be simulated in a cycle-based simulator using the following sequence:

Cycle-based simulators collapse combinatorial logic into equations.

1. When the circuit description is compiled, all combinatorial functions are collapsed into a single expression that can be used to determine all flip-flop input values based on the current state of the fan-in flip-flops.

   For example, the combinatorial function between Q1 and Q2 would be compiled from the following initial description:

   ```
   S1 = Q1 & '1'
   S2 = S1 | '0'
   S3 = S2 ^ '0'
   ```

   into this final single expression:

   ```
   S3 = Q1
   ```

   The cycle-based simulation view of the compiled circuit is shown in Figure 2-5.
2. During simulation, whenever the clock input rises, the value of all flip-flops are updated using the input value returned by the pre-compiled combinatorial input functions.

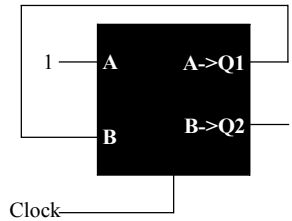


Figure 2-5. Cycle-based simulator view of a synchronous circuit

The simulation of the same circuit, using a cycle-based simulator, required the generation of two events and the execution of a single model. The number of logic computations performed is the same in both cases. They would have been performed whether the "A" input changed or not. As long as the time required to perform logic computation is smaller than the time required to schedule intermediate events,[1] and there are many registers changing state at every clock cycle, cycle-based simulation will offer greater performance.

Cycle-based simulations have no timing information.

This great improvement in simulation performance comes at a cost: All timing and delay information is lost. Cycle-based simulators assume that the entire simulation model of the design meets the setup and hold requirements of all the flip-flops. When using a cycle-based simulator, timing is usually verified using a static timing analyzer.

Cycle-based simulators can only handle synchronous circuits.

Cycle-based simulators further assume that the active clock edge is the only significant event in changing the state of the design. All other inputs are assumed to be perfectly synchronous with the active clock edge. Therefore, cycle-based simulators can only simulate perfectly synchronous designs. Anything containing asynchronous inputs, latches or multiple-clock domains *cannot* be simulated accurately. Fortunately, the same restrictions apply to static timing analysis. Thus, circuits that are suitable for cycle-based simulation to verify the functionality are suitable for static timing verification to verify the timing.

Co-Simulators

No real-world design and testbench is perfectly suited for a single simulator, simulation algorithm or modeling language. Different

1. And it is. By a long shot.

components in a design may be specified using different languages. A design could contain small sections that cannot be simulated using a cycle-based algorithm. Some portion of the design may contain some legacy blocks coded in VHDL or be implemented using analog circuitry.

Multiple simulators can handle separate portions of a simulation.

To handle the portions of a design that do not meet the requirements for cycle-based simulation, most cycle-based simulators are integrated with an event-driven simulator. As shown in Figure 2-6, the synchronous portion of the design is simulated using the cycle-based algorithm, while the remainder of the design is simulated using a conventional event-driven simulator. Both simulators (event-driven and cycle-based) are running together, cooperating to simulate the entire design.

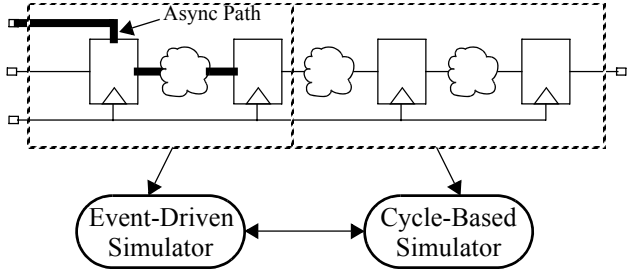


Figure 2-6. Event-driven and cycle-based co-simulation

Other popular co-simulation environments provide VHDL, SystemVerilog, SystemC, assertions and analog co-simulation. For example, Figure 2-7 shows the testbench (written in SystemVerilog) and a mixed-signal design co-simulated using a SystemVerilog digital simulator and an analog simulator.

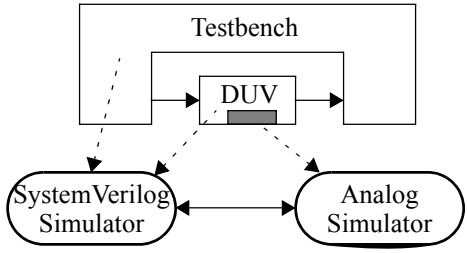


Figure 2-7. Digital and analog co-simulation

All simulators operate in locked-step.

During co-simulation, all simulators involved progress along the time axis in lock-step. All are at simulation time T_1 at the same time and reach the next time T_2 at the same time. This implies that the speed of a co-simulation environment is limited by the slowest simulator. Some experimental co-simulation environments implement *time warp* synchronization where some simulators are allowed to move ahead of the others.

Performance is decreased by the communication and synchronization overhead.

The biggest hurdle of co-simulation comes from the communication overhead between the simulators. Whenever a signal generated within a simulator is required as an input by another, the current value of that signal, as well as the timing information of any change in that value, must be communicated. This communication usually involves a translation of the event from one simulator into an (almost) equivalent event in another simulator. Ambiguities can arise during that translation when each simulation has different semantics. The difference in semantics is usually present: the semantic difference often being the requirement for co-simulation in the first place.

Translating values and events from one simulator to another can create ambiguities.

Examples of translation ambiguities abound. How do you map SystemVerilog's 128 possible states (composed of orthogonal logic values and strengths) into VHDL's nine logic values (where logic values and strengths are combined)? How do you translate a voltage and current value in an analog simulator into a logic value and strength in a digital simulator? How do you translate an x or z value into a 2-state *C++* value?

Co-simulation should not be confused with single-kernel simulation.

Co-simulation is when two (or more) simulators are cooperating to simulate a design, each simulating a portion of the design, as shown in Figure 2-8. It should not be confused with simulators able to read and compile models described in different languages. For example, Synopsys' VCS can simulate a design described using a mix of SystemVerilog, VHDL, OpenVera and SystemC. As shown in Figure 2-9, all languages are compiled into a single internal representation or machine code and the simulation is performed using a single simulation engine.

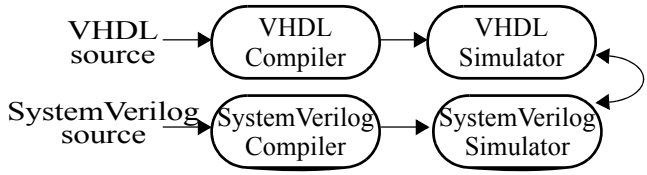


Figure 2-8. Co-simulator

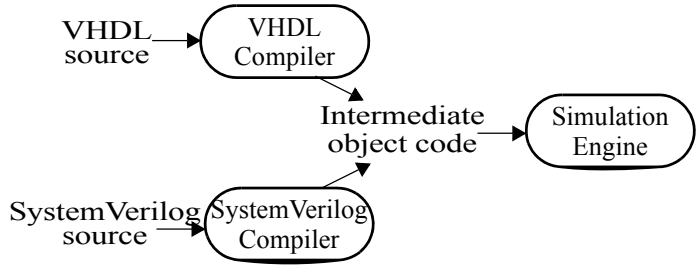


Figure 2-9. Mixed-language simulator

VERIFICATION INTELLECTUAL PROPERTY

You can buy IP for standard functions.

If you want to verify your design, it is necessary to have models for all the parts included in a simulation. The model of the RTL design is a natural by-product of the design exercise and the actual objective of the simulation. Models for embedded or external RAMs are also required, as well as models for standard interfaces and off-the-shelf parts. If you were able to procure the RAM, design IP, specification or standard part from a third party, you should be able to obtain a model for it as well. You may have to obtain the model from a different vendor than the one who supplies the design component.

It is cheaper to buy models than write them yourself.

At first glance, buying a simulation model from a third-party provider may seem expensive. Many have decided to write their own models to save on licensing costs. However, you have to decide if this endeavor is truly economically fruitful: Are you in the modeling business or in the chip design business? If you have a shortage of qualified engineers, why spend critical resources on writing a model that does not embody any competitive advantage for your company? If it was not worth designing on your own in the first place, why is writing your own model suddenly justified?

Your model is not as reliable as the one you buy.

Secondly, the model you write has never been used before. Its quality is much lower than a model that has been used by several other companies before you. The value of a functionally correct and reliable model is far greater than an uncertain one. Writing and verifying a model to the *same degree of confidence* as the third-party model is always more expensive than licensing it. And be assured: No matter how simple the model is (such as a quad 2-input NAND gate, 74LS00), you'll get it wrong the first time. If not functionally, then at least with respect to timing or connectivity.

There are several providers of verification IP. Many are written using a proprietary language or C code; others are provided as non-synthesizeable SystemVerilog source code. For intellectual property protection and licensing technicalities, most are provided as compiled binary or encrypted models. Verification IP includes, but is not limited to functional models of external and embedded memories, bus-functional models for standard interfaces, protocol generators and analyzers, assertion sets for standard protocols and black-box models for off-the-shelf components and processors.

WAVEFORM VIEWERS

Waveform viewers display the changes in signal values over time.

Waveform viewing is the most common verification technology used in conjunction with simulators. It lets you visualize the transitions of multiple signals over time, and their relationship with other transitions. With a waveform viewer, you can zoom in and out over particular time sequences, measure time differences between two transitions, or display a collection of bits as bit strings, hexadecimal or as symbolic values. Figure 2-10 shows a typical display of a waveform viewer showing the inputs and outputs of a 4-bit synchronous counter.

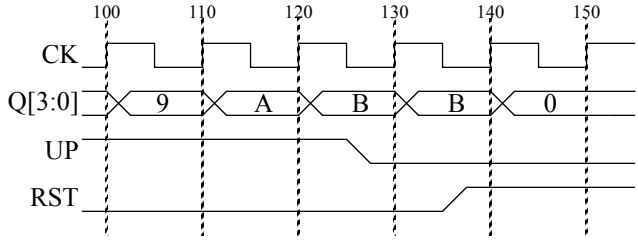


Figure 2-10. Hypothetical waveform view of a 4-bit synchronous counter

Waveform viewing is used to debug simulations.

Waveform viewing is indispensable during the authoring phase of a design or a testbench. With a viewer, you can casually inspect that the behavior of the code is as expected. They are needed to diagnose, in an efficient fashion, why and when problems occur in the design or testbench. They can be used interactively during the simulation, but more importantly offline, after the simulation has completed. As shown in Figure 2-11, a waveform viewer can play back the events that occurred during the simulation that were recorded in some trace file.

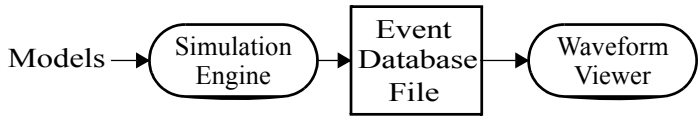


Figure 2-11. Waveform viewing as post-processing

Recording waveform trace data decreases simulation performance.

Viewing waveforms as a post-processing step lets you quickly browse through a simulation that can take hours to run. However, keep in mind that recording trace information significantly reduces the performance of the simulator. The quantity and scope of the signals whose transitions are traced, as well as the duration of the trace, should be limited as much as possible. Of course, you have to trade-off the cost of tracing a greater quantity or scope of signals versus the cost of running the simulation over again to get a trace of additional signals that turn out to be required to completely diagnose the problem. If it is likely or known that bugs will be reported, such as the beginning of the project or during a debugging iteration, trace all the signals required to diagnose the problem. If no errors are expected, such as during regression runs, no signal should be traced.

Do not use waveform viewing to determine if a design passes or fails.

In a functional verification environment, using a waveform viewer to determine the correctness of a design involves interpreting the dozens (if not hundreds) of wavy lines on a computer screen against some expectation. It can be an acceptable verification method used two or three times, for less than a dozen signals. As the number of signals and transitions increases, so does the number of relationships that must be checked for correctness. Multiply that by the duration of the simulation. Multiply again by the number of simulation runs. Very soon, the probability that a functional error is missed reaches one hundred percent.

Some viewers can compare sets of waveforms.

Some waveform viewers can compare two sets of waveforms. One set is presumed to be a golden reference, while the other is verified for any discrepancy. The comparator visually flags or highlights any differences found. This approach has two significant problems.

How do you define a set of waveforms as "golden"?

First, how is the golden reference waveform set declared "golden"? If visual inspection is required, the probability of missing a significant functional error remains equal to one hundred percent in most cases. The only time golden waveforms are truly available is in a redesign exercise, where cycle-accurate backward compatibility must be maintained. However, there are very few of these designs. Most redesign exercises take advantage of the process to introduce needed modifications or enhancements, thus tarnishing the status of the golden waveforms.

And are the differences really significant?

Second, waveforms are at the wrong level of abstraction to compare simulation results against design intent. Differences from the golden waveforms may not be significant. The value of all output signals is not significant all the time. Sometimes, what is significant is the relative relationships between the transitions, not their absolute position. The new waveforms may be simply shifted by a few clock cycles compared to the reference waveforms, but remain functionally correct. Yet, the comparator identifies this situation as a mismatch.

Use assertions instead.

You should avoid using waveform viewing to check a response. It should be reserved for debugging. Instead of looking for specific signal relationships in a waveform viewer, specify these same relationships using assertions. The assertions will be continuously and reliably checked, for the entire duration of the simulation, for all simulations. They will provide a specification of the intended functionality of the design. Should your design be picked up by another designer, your original intent will be communicated along with your original code.

CODE COVERAGE

Did you forget to verify some function in your code?

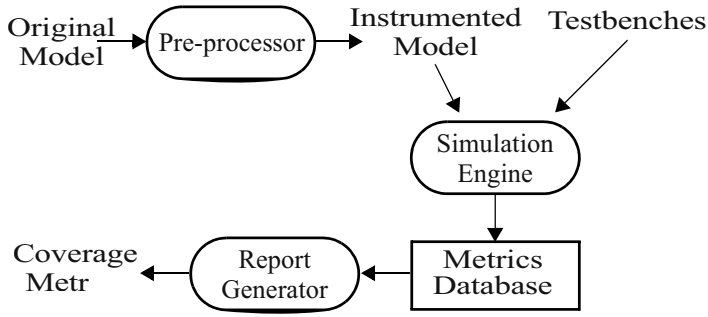
Code coverage is a technology that can identify what code has been (and more importantly *not* been) executed in the design under verification. It is a technology that has been in use in software engineering for quite some time. The problem with a design containing an unknown bug is that it looks just like a perfectly good design. It is

impossible to know, with one hundred percent certainty, that the design being verified is indeed functionally correct. All of your testbenches simulate successfully, but are there sections of the RTL code that you did not exercise and therefore not triggered a functional error? That is the question that code coverage can help answer.

Code must first be instrumented.

Figure 2-12 shows how code coverage works. The source code is first
points at strategic locations of the source code to record whether a particular construct has been exercised. The instrumentation mechanism varies from tool to tool. Some may use file I/O features available in the language (i.e., use *$write* statements in SystemVerilog). Others may use special features built into the simulator.

Figure 2-12. Code coverage process



No need to instrument the testbenches.

Only the code for the design under verification needs to be covered and thus instrumented. The objective of code coverage is to determine if you have forgotten to exercise some code in the design. The code for the testbenches need not be traced to confirm that it has executed. If a significant section of a testbench was not executed, it should be reflected in some portion of the design not being exercised. Furthermore, a significant portion of testbench code is executed only if errors are detected. Code coverage metrics on testbench code are therefore of little interest.

Trace information is collected at runtime.

The instrumented code is then simulated normally using all available, uninstrumented, testbenches. The cumulative traces from all simulations are collected into a database. From that database, reports can be generated to measure various coverage metrics of the verification suite on the design.

The most popular metrics are statement, path and expression coverage.

Statement Coverage

Statement and block coverage are the same thing.

Statement coverage can also be called block coverage, where a block is a sequence of statements that are executed if a single statement is executed. The code in Sample 2-8 shows an example of a statement block. The block named *acked* is executed entirely whenever the expression in the *if* statement evaluates to *true*. So counting the execution of that block is equivalent to counting the execution of the four individual statements within that block.

Sample 2-8. Block vs. statement execution

```
if (dtack == 1'b1) begin: acked
   as     <= 1'b0;
   data   <= 16'hZZZZ;
   bus_rq <= 1'b0;
   state  <= IDLE;
end: acked
```

But block boundaries may not be that obvious.

Statement blocks may not be necessarily clearly delimited. In Sample 2-9, two statement blocks are found: one before (and including) the *wait* statement, and one after. The *wait* statement may have never completed and the execution was waiting forever. The subsequent sequential statements may not have executed. Thus, they form a separate statement block.

Sample 2-9. Blocks separated by a *wait* statement

```
address <= 16'hFFED;
ale     <= 1'b1;
rw      <= 1'b1;
wait (dtack == 1'b1);
read_data = data;
ale     <= 1'b0;
```

Did you execute all the statements?

Statement, line or block coverage measures how much of the total lines of code were executed by the verification suite. A graphical user interface usually lets the user browse the source code and quickly identify the statements that were not executed. Figure 2-13 shows, in a graphical fashion, a statement coverage report for a small portion of code from a model of a modem. The actual form of

the report from any code coverage tool or source code browser will likely be different.

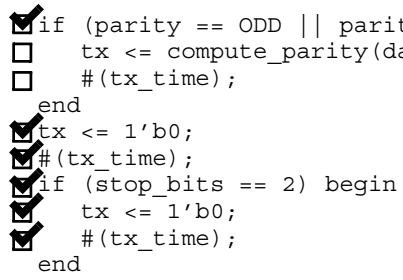


Figure 2-13. Example of statement coverage

Why did you not execute all statements?

The example in Figure 2-13 shows that two out of the eight executable statements—or 25 percent—were not executed. To bring the statement coverage metric up to 100 percent, a desirable goal[1], it is necessary to understand what conditions are required to cause the execution of the uncovered statements. In this case, the parity must be set to either ODD or EVEN. Once the conditions have been determined, you must understand why they never occurred in the first place. Is it a condition that can never occur? Is it a condition that should have been verified by the existing verification suite? Or is it a condition that was forgotten?

Add testcases to execute all statements.

If the conditions that would cause the uncovered statements to be executed should have been verified, it is an indication that one or more testbenches are either not functionally correct or incomplete. If the condition was entirely forgotten, it is necessary to add to an existing testbench, create an entirely new one or make additional runs with different seeds.

Path Coverage

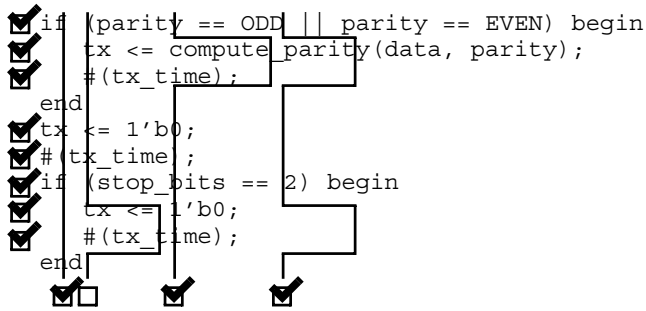
There is more than one way to execute a sequence of statements.

Path coverage measures all possible ways you can execute a sequence of statements. The code in Sample 2-10 has four possible paths: the first *if* statement can be either true or false. So can the second. To verify all paths through this simple code section, it is

1. But not necessarily achievable. For example, the *default* clause in a fully specified *case* statement should never be executed.

necessary to execute it with all possible state combinations for both *if* statements: false-false, false-true, true-false, and true-true.

Sample 2-10. Example of statement and path coverage



Why were some sequences not executed?

The current verification suite, although it offers 100 percent statement coverage, only offers 75 percent path coverage through this small code section. Again, it is necessary to determine the conditions that cause the uncovered path to be executed. In this case, a testcase must set the parity to neither ODD nor EVEN and the number of stop bits to two. Again, the important question one must ask is whether this is a condition that will ever happen, or if it is a condition that was overlooked.

Limit the length of statement sequences.

The number of paths in a sequence of statements grows exponentially with the number of control-flow statements. Code coverage tools give up measuring path coverage if their number is too large in a given code sequence. To avoid this situation, keep all sequential code constructs (*always* and *initial* blocks, *tasks* and *functions*) to under 100 lines.

Reaching 100 percent path coverage is very difficult.

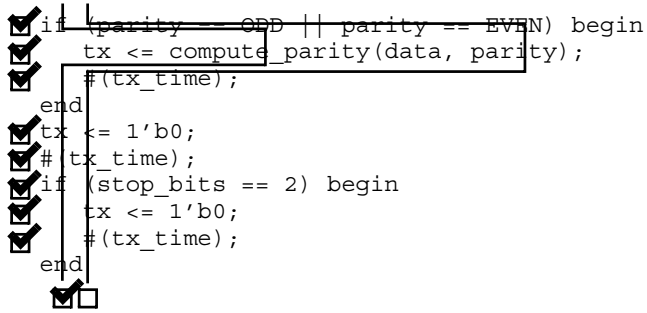
Expression Coverage

There may be more than one cause for a control-flow change.

If you look closely at the code in Sample 2-11, you notice that there are two mutually independent conditions that can cause the first *if* statement to branch the execution into its *then* clause: parity being set to either ODD or EVEN. Expression coverage, as shown in Sample 2-11, measures the various ways decisions through the code

are made. Even if the statement coverage is at 100 percent, the expression coverage is only at 50 percent.

Sample 2-11. Example of statement and expression coverage



Once more, it is necessary to understand why a controlling term of an expression has not been exercised. In this case, no testbench sets the parity to EVEN. Is it a condition that will never occur? Or was it another oversight?

Reaching 100 percent expression coverage is extremely difficult.

FSM Coverage

Statement coverage detects unvisited states.

Because each state in an FSM is usually explicitly coded using a choice in a *case* statement, any unvisited state will be clearly identifiable through uncovered statements. The state corresponding to an uncovered *case* statement choice was not visited during verification.

FSM coverage identifies state transitions.

Figure 2-14 shows a bubble diagram for an FSM. Although it has only five states, it has significantly more possible transitions: 14 possible transitions exist between adjoining states. State coverage of 100 percent can be easily reached through the sequence Reset, A, B, D, then C. However, this would yield only 36 percent transition coverage. To completely verify the implementation of this FSM, it is necessary to ensure the design operates according to expectation for all transitions.

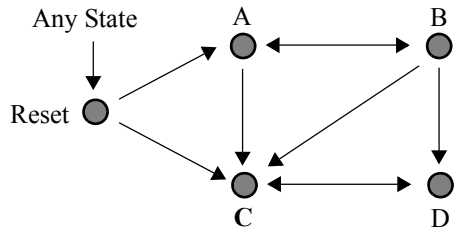


Figure 2-14. Example FSM bubble diagram

FSM coverage cannot identify unintended or missing transitions.

The transitions identified by FSM coverage tools are automatically extracted from the implementation of the FSM. There is no way for the coverage tool to determine whether a transition was part of the intent, or if an intended transition is missing. It is important to review the extracted state transitions to ensure that only and all intended transitions are present.

What about unspecified states?

The FSM illustrated in Figure 2-14 only shows five specified states. Once synthesized into hardware, a 3-bit state register will be necessary (maybe more if a different state encoding scheme, such as one-hot, is used). This leaves three possible state values that were not specified. What if some cosmic rays zap the design into one of these unspecified states? Will the correctness of the design be affected? Logic optimization may yield decode logic that creates an island of transitions within those three unspecified states, never letting the design recover into specified behavior unless reset is applied. The issues of design safety and reliability and techniques for ensuring them are beyond the scope of this book. But it is the role of a verification engineer to ask those questions.

Formal verification may be better suited for FSM verification.

The behavior of a FSM is a combination of its state transition description and the behavior of its input signals. If those input signals are themselves generated by another FSM or follow a specific protocol, it is possible that certain transitions cannot be taken or states cannot be reached. A property checker tool may be able to formally determine which states are reachable and which transitions are possible—including invalid states and transitions. It may also be able to formally verify that a specific state encoding, such as *one-hot*, is never violated.

What Does 100 Percent Code Coverage Mean?

Completeness does not imply correctness.

The short answer is: The entire design implementation was executed. Code coverage indicates how *thoroughly* your entire verification suite exercises the source code. But it does not provide an indication, in any way, about the *correctness* or *completeness* of the verification suite. Figure 2-15 shows the reconvergence model for automatically extracted code coverage metrics. It clearly shows that it does not help verify design intent, only that the RTL code, correct or not, was fully exercised.

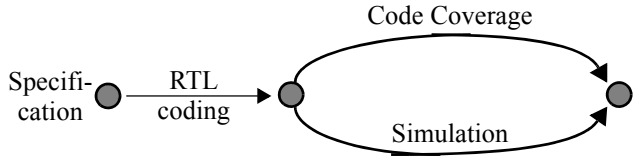


Figure 2-15. Reconvergent paths in automated code coverage

Results from code coverage should be interpreted with a grain of salt. They should be used to help identify corner cases that were not exercised by the verification suite or implementation-dependent features that were introduced during the implementation. You should also determine if the uncovered cases are relevant and deserve additional attention, or a consequence of the mindlessness of the coverage tool.

Code coverage lets you know if you are not done.

Code coverage indicates if the verification task is *not* complete through low coverage numbers. A high coverage number is by no means an indication that the job is over. For example, the code in an empty *module* will always be 100 percent covered. If the functionality that ought to be implemented in that *module* is not verified, all testbenches will pass without errors. Code coverage is an additional indicator for the completeness of the verification job. It can help increase your confidence that the verification job is complete, but it should not be your only indicator.

Code coverage tools can be used as profilers.

When developing models for simulation only, where performance is an important criteria, code coverage can be used for *profiling*. The aim of profiling is the opposite of code coverage. The aim of profiling is to identify the lines of codes that are executed most often. These lines of code become the primary candidates for performance optimization efforts.

FUNCTIONAL COVERAGE

Did you forget to verify some condition?

Functional coverage is another technology to help ensure that a bad design is not hiding behind passing testbenches. Although this technology has been in use at some companies for quite some time, it is a recent addition to the arsenal of general-purpose verification tools. Functional coverage records relevant metrics (e.g., packet length, instruction opcode, buffer occupancy level) to ensure that the verification process has exercised the design through all of the interesting values. Whereas code coverage measures how much of the implementation has been exercised, functional coverage measures how much of the original design specification has been exercised.

It complements code coverage.

High functional coverage does not necessarily correlate with high code coverage. Whereas code coverage is concerned with recording the mechanics of code execution, functional coverage is concerned with the intent or purpose of the implemented function. For example, the decoding of a CPU instruction may involve separate *case* statements for each field in the opcode. Each *case* statement may be 100 percent code-covered due to combinations of field values from previously decoded opcodes. However, the particular combination involved in decoding a specific CPU instruction may not have been exercised.

It will detect errors of omission.

Sample 2-12 shows a *case* statement decoding a CPU instruction. Notice how the decoding of the RTS instruction is missing. If you rely solely on code coverage, you will be lulled in a false sense of completeness by having 100 percent coverage of this code. For code coverage to report a gap, the unexercised code must apriori exist. Functional coverage does not rely on actual code. It will report gaps in the recorded values whether the code to process them is there or not.

Sample 2-12. Example of coding error undetectable by code coverage

```
enum {ADD, SUB, JMP, RTS, NOP} opcode;
...
case (opcode)
   ADD: ...
   SUB: ...
   JMP: ...
   default: ...
endcase
```

It must be manually defined.

Code coverage was quickly adopted into verification processes because of its low adoption cost. It requires very little additional action from the user: usually the specification of an additional command-line option when compiling your code. Functional coverage, because it is a measure of values deemed to be interesting and relevant, must be manually specified. Since relevance and interest are qualities that are extracted from the intent of the design, functional coverage is not something that can be automatically extracted from the RTL source code. Your functional coverage metrics will be only as good as what you implement.

Metrics are collected at runtime and graded.

Like code coverage, functional coverage metrics are collected at runtime, during a simulation run. The values from individual runs are collected into a database or separate files. The functional coverage metrics from these separate runs are then merged for offline analysis. The marginal coverage of individual runs can then be graded to identify which runs contributed the most toward the overall functional coverage goal. These runs are then given preference in the regression suite, while pruning runs that did not significantly contribute to the objective.

Coverage data can be used at runtime.

SystemVerilog provides a set of predefined methods that let a testbench dynamically query a particular functional coverage metric. The testbench can then use the information to modify its current behavior. For example, it could increase the probability of generating values that have not been covered yet. It could decide to abort the simulation should the functional coverage not have significantly increased since the last query.

Although touted as a powerful mechanism, it is no silver bullet. Implementing the dynamic feedback mechanism is not easy: You have to correlate your stimulus generation process with the functional coverage metric, and ensure that one will cause the other to converge toward the goal. Dynamic feedback works best when there is a direct correlation between the input and the measured coverage, such as instruction types. It may be more efficient to achieve your goal with three or four runs of a few simple testbenches without dynamic feedback than with a single run of a much more complex testbench.

Coverage Points

Did I generate all interesting and relevant values?

A coverage point is the sampling of an individual scalar value or expression. The objective of a coverage point is to ensure that all interesting and relevant values have been observed in the sampled value or expression. Examples of coverage points include, but are not limited to, packet length, instruction opcode, interrupt level, bus transaction termination status, buffer occupancy level, bus request patterns and so on.

Define *what* to sample.

It is extremely easy to record functional coverage and be inundated with vast amounts of coverage data. But data is not the same thing as information. You must restrict coverage to only (but all!) values that will indicate how thoroughly your design has been verified. For example, measuring the value of the read and write pointers in a FIFO is fine if you are concerned about the full utilization of the buffer space and wrapping around of the pointer values. But if you are interested in the FIFO occupancy level (Was it ever empty? Was it ever full? Did it overflow?), you should measure and record the *difference* between the pointer values.

Define *where* to sample it.

Next, you must decide where in your testbench or design is the measured value accurate and relevant. For example, you can sample the opcode of an instruction at several places: at the output of the code generator, at the interface of the program memory, in the decoder register or in the execution pipeline. You have to ensure that a value, once recorded, is indeed processed or committed as implied by the coverage metric.

For example, if you are measuring opcodes that were executed, they should be sampled in the execution unit. Sampling them in the decode unit could result in false samples when the decode pipeline is flushed on branches or exceptions. Similarly, sampling the length of packets at the output of the generator may yield false samples: If a packet is corrupted by injecting an error during its transmission to the design in lower-level functions of the testbench, it may be dropped.

Define *when* to sample it.

Values are sampled at some point in time during the simulation. It could be at every clock cycle, whenever the address strobe signal is asserted, every time a request is made or after randomly generating a new value. You must carefully choose your sampling time. Over-

sampling will decrease simulation performance and consume database resources without contributing additional information.

The sampled data must also be stable so race conditions must be avoided between the sampled data and the sampling event (see “Read/Write Race Conditions” on page 177). To reduce the probability that a transient value is being sampled, functional coverage in SystemVerilog can be sampled at the end of the simulation cycle, before time is about to advance (see “The Simulation Cycle” on page 163) when the *strobe* option is used.

Define *why* it is covered.

If functional coverage is supposed to measure interesting and relevant values, it is necessary to define what makes those values so interesting and relevant. For example, measuring the functional coverage of a 32-bit address will yield over 4 billion “interesting and relevant” values. Not all values are created equal—but most are. Values may be numerically different but functionally equivalent. By identifying those functionally equivalent values into a single bin, you can reduce the number of interesting and relevant values to a more manageable size. For example, based on the decoder architecture, addresses 0x00000001 through 0x7FFFFFFF and addresses 0x80000000 through 0x8FFFFFFE are functionally equivalent, reducing the number of relevant and interesting values to 4 bins (min, 1 to mid, mid to max-1, max).

It can detect invalid values.

Just as you can define bins of equivalent values, it is possible to define bins of invalid or unexpected values. Functional coverage can be used as an error detecting mechanism, just like an *if* statement in your testbench code. However, you should not rely on functional coverage to detect invalid values. Functional coverage is an optional runtime data collection mechanism that may not be turned on at all times. If functional coverage is not enabled to improve simulation performance and if a value is defined as invalid in the functional coverage only, then an invalid value may go undetected.

It can report holes.

The ultimate purpose of functional coverage is to identify what remains to be done. During analysis, the functional coverage reporting tool can compare the number of bins that contain at least one sample against the total number of bins. Any bin that does not contain at least one sample is a hole in your functional coverage. By enumerating the empty bins, you can focus on the holes in your test cases and complete your verification sooner rather than continue to exercise functionality that has already been verified.

For this enumeration to be possible, the total number of bins for a coverage point must be relatively small. For example, it is practically impossible to fill the coverage for a 32-bit value without broad bins. The number of holes will be likely in the millions, making enumeration impossible. You should strive to limit the number of possible bins as much as possible.

Cross Coverage

Did I generate all interesting *combinations* of values?

Whereas coverage points are concerned with individual scalar values, cross coverage measures the presence or occurrence of combinations of values. It helps answer questions like, "Did I inject a corrupted packet on all ports?" "Did we execute all combinations of opcodes and operand modes?" and "Did this state machine visit each state while that buffer was full, not empty and empty?" Cross coverage can involve more than two coverage points. However, the number of possible bins grows factorially with the number of crossed points.

Similar to coverage points.

Mechanically, cross coverage is identical to coverage points. Specific values are sampled at specific locations at specific points in time with specific value bins. The only difference is that two or more values are sampled instead of one. To ensure that crossed values are consistent, they must all be sampled at the same time. In SystemVerilog, only coverage points specified within the same *covergroup* can be crossed.

Transition Coverage

Did I generate all interesting *sequences* of values?

Whereas cross coverage is concerned with combinations of scalar values at the same point in time, transition coverage measures the presence or occurrence of sequences of values. Transition coverage helps answer questions like, "Did I perform all combinations of back-to-back read and write cycles?" "Did we execute all combinations of arithmetic opcodes followed by test opcodes?" and "Did this state machine traverse all significant paths?" Transition coverage can involve more than two consecutive values of the same coverage point. However, the number of possible bins grows factorially with the number of transition states.

Similar to coverage points.

Mechanically, transition coverage is identical to coverage points. Specific values are sampled at specific locations at specific points

in time with specific bins. The only difference is that a sample is said to have occurred in a bin after two or more consecutive coverage point samples instead of one. The other difference is that transitions can overlap, hence two transition samples may be composed of the same coverage point sample.

Similar to FSM path coverage.

Conceptually, transition coverage is identical to FSM path coverage (see "FSM Coverage" on page 46). Both record the consecutive values at a particular location of the design (for example, a state register), and both compare against the possible set of paths. But unlike FSM coverage tools, which are limited to state registers in RTL code, transition coverage can be applied to any coverage points in testbenches and the design under verification.

Transition coverage reflects intent.

Because transition coverage is manually specified from the intent of the design or the implementation, it provides a true independent path to verifying the correctness of the design and the completeness of the verification. It can detect invalid transitions as well as specify transitions that may be missing from the implementation of the design.

What Does 100 Percent Functional Coverage Mean?

It indicates completeness of the test suite, not correctness.

Functional coverage indicates which interesting and relevant conditions were verified. It provides an indication of the *thoroughness* of the implementation of the verification plan. Unless some bins are defined as invalid, it cannot provide an indication, in any way, about the *correctness* of those conditions or of the design's response to those conditions. Functional coverage metrics are only as good as the functional coverage model you have defined. An overall functional coverage metric of 100 percent means that you've covered all of the coverage points you included in the simulation. It makes no statement about the *completeness* of your functional coverage model.

Results from functional coverage should also be interpreted with a grain of salt. Since they are generated by additional testbench constructs, they have to be debugged and verified for correctness before being trusted. They will help identify additional interesting conditions that were not included in the verification plan.

Functional coverage lets you know if you are done.

When used properly, functional coverage becomes a formal specification of the verification plan. Once you reach 100 percent functional coverage, it indicates that you have created and exercised all of the relevant and interesting conditions you originally identified. It confirms that you have implemented everything in the verification plan. However, it does not provide any indication of the completeness of the verification plan itself or the correctness of the design under such conditions.

If a metric is not interesting, don't measure it.

It is extremely easy to define functional coverage metrics and generate many reports. If coverage is not measured according to a specific purpose, you will soon drown under megabytes of functional coverage reports. And few of them will ever be close to 100 percent. It will also become impossible to determine which report is significant or what is the significance of the holes in others. The verification plan (see the next chapter) should serve as the functional specification for the coverage models, as well as for the rest of the verification environment. If a report is not interesting or meaningful to look at, if you are not eager to look at a report after a simulation run, then you should question its existence.

VERIFICATION LANGUAGE TECHNOLOGIES

Verilog is a simulation language, not a verification language.

Verilog was designed with a focus on describing low-level hardware structures. Verilog-2001 only introduced support for basic high-level data structures. Verilog thus continued to lack features important in efficiently implementing a modern verification process. These shortcomings were the forces being the creation of hardware verification languages, such as Synopsys' OpenVera. Having demonstrated their usefulness, the value-add functionality of HVLs has been incorporated in SystemVerilog.

Verification languages can raise the level of abstraction.

As mentioned in Chapter 1, one way to increase productivity is to raise the level of abstraction used to perform a task. High-level languages, such as C or Pascal, raised the level of abstraction from assembly-level, enabling software engineers to become more productive. Similarly, the SystemVerilog verification constructs are able to raise the level of abstraction compared to plain Verilog. SystemVerilog can provide an increase in level of abstraction while maintaining the important concepts necessary to interact with hardware: time, concurrency and instantiation. The SystemVerilog fea-

tures that help raise the level of abstraction include: *class*, object-oriented *class* extensions and temporal properties.

SystemVerilog can automate verification.

If higher levels of abstraction and object-orientedness were sufficient, then *C++* would have long been identified as the best solution[1]: It is free and widely known. SystemVerilog provides additional benefits, as shown in Figure 2-16. It can automate a portion of the verification process by randomly generating stimulus, collecting functional coverage to identify holes then easily add or modify constraints to create more stimulus targeted to fill those holes. To support this productivity cycle, SystemVerilog offers constrainable random generation, functional coverage measurement and an object-oriented code extension mechanism.

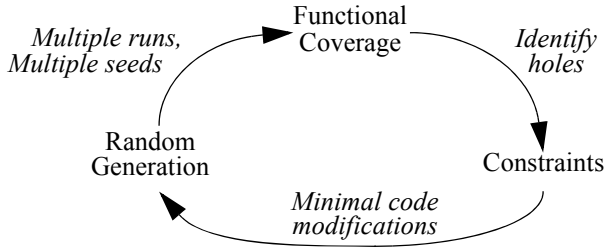


Figure 2-16. SystemVerilog productivity cycle

SystemVerilog can implement a coverage-driven constrained random approach.

SystemVerilog can be used as if it was a simple souped-up version of Verilog. SystemVerilog will make implementing directed testbenches easier than plain Verilog—especially the self-checking part. But if you want to take advantage of the productivity cycle shown in Figure 2-16, the verification process must be approached—and implemented—in a different fashion.

This change is just like taking advantage of the productivity offered by logic synthesis tools: It requires an approach different from schematic capture. To successfully implement a coverage-driven constrained random verification approach, you need to modify the way you plan your verification, design its strategy and implement the testcases. This new approach is described in “Coverage-Driven Random-Based Approach” on page 101.

1. C++ still lacks a native concept of time, concurrency and instantiation.

ASSERTIONS

Assertions detect conditions that should always be true.

An assertion boils down to an *if* statement and an error message should the expression in the *if* statement become false. Assertions have been used in software design for many years: the *assert()* function has been part of the ANSI C standard from the beginning. In software for example, assertions are used to detect conditions such as NULL pointers or empty lists. VHDL has had an *assert* statement from day one too, but it was never a popular construct.

Hardware assertions require a form of temporal language.

An immediate assertion, like an *if* statement, simply checks that, at the time it is executed, the condition evaluates to TRUE. This simple zero-time test is not sufficient for supporting assertions in hardware designs. In hardware, functional correctness usually involves behavior over a period of time. Some hardware assertions such as, "This state register is one-hot encoded." or "This FIFO never overflows." can be expressed as immediate, zero-time expressions. But checking simple hardware assertions such as, "This signal must be asserted for a single clock period." or "A request must always be followed by a grant or abort within 10 clock cycles." require that the assertion condition be evaluated over time. Thus, assertions require the use of a temporal language to be able to describe relationships over time.

There are two classes of assertions.

Assertions fall in two broad classes: those specified by the designer and those specified by the verification engineer.

- Implementation assertions are specified by the designers.
- Specification assertions are specified by the verification engineers.

Implementation assertions verify assumptions.

Implementation assertions are used to formally encode the designer's assumptions about the interface or implementation of the design or conditions that are indications of misuse or design faults. For example, the designer of a FIFO would add assertions to detect if it ever overflows or underflows or that, because of a design limitation, the *write* and *read* pulses are ever asserted at the same time. Because implementation assertions are specified by the designer, they will not detect discrepancies between the functional intent and the design. But implementation assertions will detect discrepancies between the design assumptions and the implementation.

Specification assertions verify intent.

Specification assertions formally encode expectations of the design based on the functional intent. These assertions are used as a functional error detection mechanism and supplement the error detections performed in the self-checking section of testbenches. Specification assertions are typically *white-box* strategies because the relationships between the primary inputs and outputs of a modern design are too complex to be described efficiently in SystemVerilog's temporal languages. For example, rather than relying on the scoreboard to detect that an arbiter is not fair, it is much simpler to perform this check using a white-box assertion.

Simulated Assertions

The OVL started the storm.

Assertions took the hardware design community by storm when Foster and Bening's book[1] introduced the concept using a library of predefined Verilog modules that implemented a set of common design assertions. The library, available in source form as the *Open Verification Library,*[2] was a clever way of using Verilog to specify temporal expressions. Foster, then at Hewlett-Packard, had a hidden agenda: Get designers to specify design assertions he could then try to prove using formal methods. Using Verilog modules was a convenient solution to ease the adoption of these assertions by the designers. The reality of what happened next proved to be even more fruitful.

They detect errors close in space and time to the fault.

If a design assumption is violated during simulation, the design will not operate correctly. The cause of the violation is not important: It could be a misunderstanding by the designer of the block or the designer of the upstream block or an incorrect testbench. The relevant fact is that the design is failing to operate according to the original intent. The symptoms of that low-level failure are usually not visible (if at all) until the affected data item makes its way to the outputs of the design and is flagged by the self-checking structure.

An assertion formally encoding the design assumption immediately fires and reports a problem at the time it occurs, in the area of the design where it occurs. Debugging and fixing the assertion failure

1. Harry Foster and Lionel Bening, "*Principles of Verifiable RTL Design,*" second edition, Kluwer Academic Publisher, ISBN 0-7923-7368-5.
2. See `http://www.eda.org/ovl`.

(whatever the cause) will be a lot more efficient than tracing back the cause of a corrupted packet. In one of Foster's projects, 85% of the design errors where caught and quickly fixed using simulated assertions.

Your model can tell you if things are not as assumed.

SystemVerilog provides a powerful assertion language. But it also provides constructs designed to ensure consistent results between synthesis and simulation. Sample 2-14 shows an example of a synthesizeable *unique case* statement, which can be used to replace the *full case* directive shown in Sample 2-13. In both cases, the synthesis tool is instructed that the *case* statement describes all possible non-overlapping conditions. But it is possible for an unexpected condition to occur during simulation. If that were the case, the simulation results would differ from the results produced by the hardware implementation. If a pragma is used, as in Sample 2-13, the unexpected condition would only be detected if it eventually produces an incorrect response. If the *unique case* statement is used, any unexpected condition will be immediately reported near the time and place of its occurence.

Sample 2-13. *full case* directive

```
case (mode[1:0]) // synopsys full_case
2'b00: ...
2'b10: ...
2'b01: ...
endcase
```

Sample 2-14. *unique case* statement

```
unique case (mode[1:0])
2'b00: ...
2'b10: ...
2'b01: ...
endcase
```

Formal Assertion Proving

Is it possible for an assertion to fire?

Simulation can show only the presence of bugs, never prove their absence. The fact that an assertion has never reported a violation throughout a series of simulations does not mean that it can never be violated. Tools like code and functional coverage can satisfy us that a portion of a design was thoroughly verified—but there will (and should) always be a nagging doubt.

Property checking can mathematically prove or disprove an assertion.

Formal tools called *property checkers* or *assertion provers* can mathematically prove that, given an RTL design and some assumptions about the relationships of the input signals, an assertion will always hold true. If a counter example is found, the formal tool will provide details on the sequence of events that leads to the assertion violation. It is then up to you to decide if this sequence of events is possible, given additional knowledge about the environment of the design.

Some assertions are used as assumptions.

Given total freedom over the inputs of a design, it may be possible to violate assertions about its implementation. The proper operation of the design may rely on the proper behavior of the inputs, subject to limitations and rules that must be followed. These input signals usually come from other designs that do not behave (one hopes!) erratically and follow the rules. When proving some assertions on a design, it is thus necessary to supply assertions on the inputs or state of the design. The latter assertions are not proven. Rather, they are assumed to be true and used to constrain the solution space for the proof.

Assumptions need to be proven too.

The correctness of a proof depends on the correctness of the assumptions[1] made on the design inputs. Should any assumption be wrong, the proof no longer stands. An assumption on a design's inputs thus becomes an assertion to be proven on the upstream design supplying those inputs.

Semi-formal tools combine property checking with simulation.

Semi-formal tools are hybrid tools that combine formal methods with simulation. Semi-formal tools are used to bridge the gap between the capacity of current formal analysis engines and the size and complexity of the design to be verified. Rather than try to prove all assertions from the reset state, they use intermediate simulation information—such as the current state of a design—as a starting point for proving or disproving assertions.

Use formal methods to prove cases uncovered in simulation.

Formal verification does not replace simulation or make it obsolete. Simulation (including simulated assertions) is the lawnmower of the verification garden: It is still the best technology for covering broad swaths of functionality and for weeding out the easy-to-find

1. The formal verification community calls these input assertions "constraints." I used the term "assumptions" to differentiate them from random-generation constraints, which are randomization concepts.

and some not-so-easy-to-find bugs. Formal verification puts the finishing touch on those hard-to-reach corners in critical and important design sections and ensures that the job is well done. Using functional coverage metrics collected from simulation (for example, request patterns on an arbiter), conditions that remain to be verified are identified. If those conditions would be difficult to create within the simulation environment, it may be easier to prove the correctness of the design for the remaining uncovered cases.

Formal verification should replace ad hoc unit-level verification.

When a designer completes the coding of a design unit—a single or a few modules implementing some elementary function—he or she verifies that it works as intended. This verification is casual and usually the waveform viewer is used to visually inspect the correctness of the response. As mentioned in "Waveform Viewers" on page 39, assertions should be used to specify the signal relationships that define the implementation as "correct" instead of looking for them visually. Once these relationships are specified using assertions, why not try to prove or disprove them using formal technology instead of simulating the design?

Assertion specification is a complex topic.

This simple introduction to assertions does not do justice to the richness and power—and ensuing complexity—of assertions. Entire books have already been written about the subject and should be consulted for more information. Chapter 3 and 7 of the *Verification Methodology Manual for SystemVerilog* provide a lot of guidelines for using assertions with simulation and formal technologies.

REVISION CONTROL

Are we all looking at the same thing?

One of the major difficulties in verification is to ensure that what is being verified is actually what will be implemented. When you compile a SystemVerilog source file, what is the guarantee that the design engineer will use that *exact same file* when synthesizing the design?

When the same person verifies and then synthesizes the design, this problem is reduced to that person using proper file management discipline. However, as I hope to have demonstrated in Chapter 1, having the same person perform both tasks is not a reliable functional verification process. It is more likely that separate individuals perform the verification and synthesis tasks.

Files must be centrally managed.

In very small and closely knit groups, it may be possible to have everyone work from a single directory, or to have the design files distributed across a small number of individual directories. Everyone agrees where each other's files are, then each is left to his or her own device. This situation is very common and very dangerous: How can you tell if the designer has changed a source file and maybe introduced a functional bug since you last verified it?

It must be easy to get at all the files, from a single location.

This methodology is not scalable either. It quickly breaks down once the team grows to more than two or three individuals. And it does not work at all when the team is distributed across different physical or geographical areas. The verification engineer is often the first person to face the non-scalability challenge of this environment. Each designer is content working independently in his or her own directories. Individual designs, when properly partitioned, rarely need to refer to some other design in another designer's working directory. As the verification engineer, your first task is to integrate all the pieces into a functional entity. That's where the difficulties of pulling bits and pieces from heterogeneous working environments scattered across multiple file servers become apparent.

The Software Engineering Experience

HDL models are software projects!

For over 30 years, software engineering has been dealing with the issues of managing a large number of source files, authored by many different individuals, verified by others and compiled into a final product. Make no mistake: Managing a synthesis-based hardware design project is no different than managing a software project.

Free and commercial tools are available.

To help manage files, software engineers use source control management systems. Some are available, free of charge, either bundled with the UNIX operating systems (RCS, CVS, SCCS), or distributed by the GNU project (RCS, CVS) and available in source form at:

```
ftp://prep.ai.mit.edu/pub/gnu
```

Commercial systems, some very sophisticated, are also available.

All source files are centrally managed.

Figure 2-17 shows how source files are managed using a source control management system. All accesses and changes to source

files are mediated by the management system. Individual authors and users interact solely through the management system, not by directly accessing files in working directories.

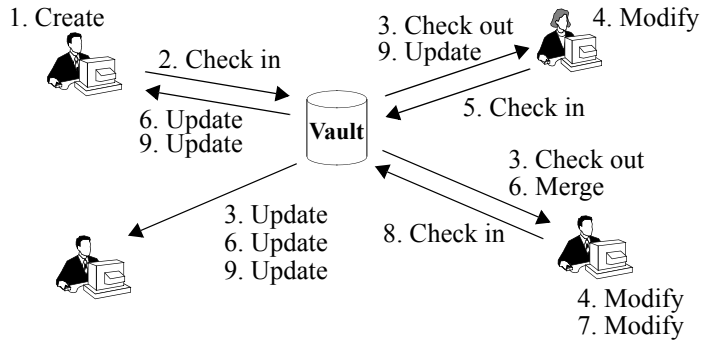


Figure 2-17. Data flow in a source control system

The history of a file is maintained.

Source code management systems maintain not only the latest version of a file, but also keep a complete history of each file as separate *versions*. Thus, it is possible to recover older versions of files, or to determine what changed from one version to another. It is a good idea to frequently *check in* file versions. You do not have to rely on a backup system if you ever accidentally delete a file. Sometimes, a series of modifications you have been working on for the last couple of hours is making things worse, not better. You can easily roll back the state of a file to a previous version known to work.

The team owns all the files.

When using a source management system, files are no longer owned by individuals. Designers may be nominally responsible for various sections of a design, but anyone—with the proper permissions—can make any change to any file. This lets a verification engineer fix bugs found in RTL code without having to rely on the designer, busy trying to get timing closure on another portion of the design. The source management system mediates changes to files either through exclusive locks, or by merging concurrent modifications.

Configuration Management

Each user works from a *view* of the file system.

Each engineer working on a project managed with a source control system has a private *view* of all the source files (or a subset thereof) used in the project. Figure 2-18 shows how two users may have two

different views of the source files in the management system. Views need not be always composed of the latest versions of all the files. In fact, for a verification engineer, that would be a hindrance. Files checked in on a regular basis by their authors may include syntax errors, be simple placeholders for future work, or be totally broken. It would be very frustrating if the model you were trying to verify kept changing faster than you could identify problems with it.

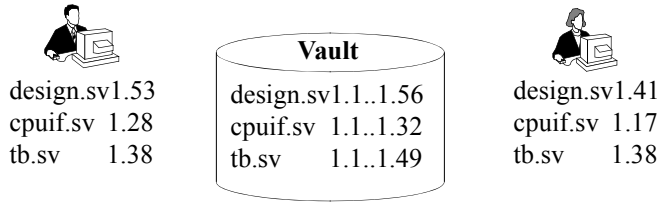


Figure 2-18. User views of managed source files

Configurations are created by tagging a set of versions.

All source management systems use the concept of symbolic tags that can be attached to specific versions of files. You may then refer to particular versions of files, or set of files, using the symbolic name, without knowing the exact version number they refer to. In Figure 2-18, the user on the left could be working with the versions that were tagged as “ready to simulate” by the author. The user on the right, the system verification engineer, could be working with the versions that were tagged as “golden” by the block-level verification engineer.

Configuration management translates to tag management.

Managing releases becomes a problem of managing tags, which can be a complex task. Table 2-1 shows a list of tags that could be used in a project to identify the various versions of a file as it progresses through the design process. Some tags, such as the “Version_M.N” tag, never move once applied to a specific version. Others, such as the “Submit” tag, move to newer versions as the development of the design progresses. Before moving a tag, it may be a good idea to leave a trace of the previous position of a tag. One possible mechanism for doing so is to append the date to the tag name. For example, the old “Submit” version gets tagged with the new tag “Submit_060302” on March 2nd, 2006 and the “Submit” tag is moved to the latest version.

Table 2-1. Example tags for release management

Tag Name	Description
Submit	Ready to submit to functional verification. Author has verified syntax correctness and basic level of functionality.
Bronze	Passes a basic set of functional testcases. Release is sufficiently functional for integration.
Silver	Passes all functional testcases.
Gold	Passes all functional testcases and meets coding coverage guidelines (requires additional corner-case testcases).
To_Synthesis	Ready to submit to synthesis. Usually matches “Silver” or “Gold”.
To_Layout	Ready to submit to layout. Usually matches “Gold”.
Version_M.N	Version that was manufactured. Matches corresponding “To_Layout” release. Future versions of the same chip will move tags beyond this point.
ON_YYMMDD	Some meaningful release on the specified date.

Working with Releases

Views can become out-of-date as new versions of files are checked into the source management system database and tags are moved forward.

Releases are specific configurations.

The author of the RTL for a portion of the design would likely always work with the latest version of the files he or she is actively working on, checking in and updating them frequently (typically at relevant points of code development throughout the day and at the end of each day). Once the source code is syntactically correct and its functionality satisfies the designer (by proving all embedded assertions or using a few ad hoc testbenches), the corresponding version of the files are tagged as ready for verification.

Users must update their view to the appropriate release.

You, as the verification engineer, must be constantly on the lookout for updates to your view. When working on a particularly difficult testbench, you may spend several days without updating your view to the latest version ready to be verified. That way, you maintain a consistent view of the design under test and limit changes to the testbenches, which you make. Once the actual verification and debugging of the design starts, you probably want to refresh your view to the latest "ready-to-verify" release of the design before running a testbench.

Update often.

When using a concurrent development model where multiple engineers are working in parallel on the same files, it is important to check in modifications often, and update your view to merge concurrent modifications even more often. If you wait too long, there is a greater probability of collisions that will require manual resolution. The concept of concurrently modifying files then merging the differences sounds impossibly risky at first. However, experience has shown that different functions or bug fixes rarely involve modification to the same lines of source code. As long as the modifications are separated by two or three lines of unmodified code, merging will proceed without any problems. Trust me, concurrent development is the way to go!

You can be notified of new releases.

An interesting feature of some source management systems is the ability to issue email notification whenever a significant event occurs. For example, such a system could send e-mail to all verification engineers whenever the tag identifying the release that is ready for verification is moved. Optionally, the e-mail could contain a copy of the descriptions of the changes that were made to the source files. Upon receiving such an e-mail, you could make an informed decision about whether to update your view immediately.

ISSUE TRACKING

All your bug are belong to us!

The job of any verification engineer is to find bugs. Under normal conditions, you should expect to find functional irregularities. You should be *really* worried if no problems are being found. Their occurrence is normal and do not reflect the abilities of the hardware designers. Even the most experienced software designers write code that includes bugs, even in the simplest and shortest routines. Now that we've established that bugs *will* be found, how will you deal with them?

Bugs must be fixed.

Once a problem has been identified, it *must* be resolved. All design teams have informal systems to track issues and ensure their resolutions. However, the quality and scalability of these informal systems leaves a lot to be desired.

What Is an Issue?

Is it worth worrying about?

Before we discuss the various ways issues can be tracked, we must first consider what is an issue worth tracking. The answer depends highly on the tracking system used. The cost of tracking the issue should not be greater than the cost of the issue itself. However, do you want the tracking system to dictate what kind of issues are tracked? Or, do you want to decide on what constitutes a trackable issue, then implement a suitable tracking system? The latter position is the one that serves the ultimate goal better: Making sure that the design is functionally correct.

An issue is *anything* that can affect the functionality of the design:

1. Bugs found during the execution of a testbench are clearly issues worth tracking.
2. Ambiguities or incompleteness in the specification document should also be tracked issues. However, typographical errors definitely do not fit in this category.
3. Architectural decisions and trade-offs are also issues.
4. Errors found at all stages of the design, in the design itself or in the verification environment should be tracked as well.
5. If someone thinks about a new relevant testcase, it should be filed as an issue.

When in doubt, track it.

It is not possible to come up with an exhaustive list of issues worth tracking. Whenever an issue comes up, the only criterion that determines whether it should be tracked, is its effect on the correctness of the final design. If a bad design can be manufactured when that issue goes unresolved, it *must* be tracked. Of course, all issues are not created equal. Some have a direct impact on the functionality of the design, others have minor secondary effects. Issues should be assigned a priority and be addressed in order of that priority.

You may choose not to fix an issue.

Some issues, often of lower importance, may be consciously left unresolved. The design or project team may decide that a particular problem or shortcoming is an acceptable limitation for this particu-

lar project and can be left to be resolved in the next incarnation of the product. The principal difficulty is to make sure that the decision was a conscious and rational one!

The Grapevine System

Issues can be verbally reported.

The simplest, and most pervasive issue tracking system is the *grapevine*. After identifying a problem, you walk over to the hardware designer's cubicle (assuming you are not the hardware designer as well!) and discuss the issue. Others may be pulled into the conversation or accidentally drop in as they overhear something interesting being debated. Simple issues are usually resolved on the spot. For bigger issues, everyone may agree that further discussions are warranted, pending the input of other individuals. The priority of issues is implicitly communicated by the insistence and frequency of your reminders to the hardware designer.

It works only under specific conditions.

The grapevine system works well with small, closely knit design groups, working in close proximity. If temporary contractors or part-time engineers are on the team, or members are distributed geographically, this system breaks down as instant verbal communications are not readily available. Once issues are verbally resolved, no one has a clear responsibility for making sure that the solution will be implemented.

You are condemned to repeat past mistakes.

Also, this system does not maintain any history. Once an issue is resolved, there is no way to review the process that led to the decision. The same issue may be revisited many times if the implementation of the solution is significantly delayed. If the proposed resolution turns out to be inappropriate, the team may end up going in circles, repeatedly trying previous solutions. Without history, you are condemned to repeat it. There is no opportunity for the team to learn from its mistakes. Learning is limited to individuals, and to the extent that they keep encountering similar problems.

The Post-It System

Issues can be tracked on little pieces of paper.

When teams become larger, or when communications are no longer regular and casual, the next issue tracking system that is used is the 3M Post-It™ note system. It is easy to recognize at a glance: Every team member has a number of telltale yellow pieces of paper stuck around the periphery of their computer monitor.

If the paper disappears, so does the issue.

This evolutionary system only addresses the lack of ownership of the grapevine system: Whoever has the yellow piece of paper is responsible for its resolution. This ownership is tenuous at best. Many issues are "resolved" when the sticky note accidentally falls on the floor and is swept away by the janitorial staff.

Issues cannot be prioritized.

With the Post-It system, issues are not prioritized. One bug may be critical to another team member, but the owner of the bug may choose to resolve other issues first simply because they are simpler and because resolving them instead reduces the clutter around his computer screen faster. All notes look alike and none indicate a sense of urgency more than the others.

History will repeat itself.

And again, the Post-It system suffers from the same learning disabilities as the grapevine system. Because of the lack of history, issues are revisited many times, and problems are recreated repeatedly.

The Procedural System

Issues can be tracked at group meetings.

The next step in the normal evolution of issue tracking is the procedural system. In this system, issues are formally reported, usually through free-form documents such as e-mail messages. The outstanding issues are reviewed and resolved during team meetings.

Only the biggest issues are tracked.

Because the entire team is involved and the minutes of meetings are usually kept, this system provides an opportunity for team-wide learning. But the procedural system consumes an inordinate amount of precious meeting time. Because of the time and effort involved in tracking and resolving these issues, it is usually reserved for the most important or controversial ones. The smaller, less important—but much more numerous—issues default back to the grapevine or Post-It note systems.

Computerized System

Issues can be tracked using databases.

A revolution in issue tracking comes from using a computer-based system. In such a system, issues must be seen through to resolution: Outstanding issues are repeatedly reported loud and clear. Issues can be formally assigned to individuals or list of individuals. Their resolution need only involve the required team members. The computer-based system can automatically send daily or weekly status reports to interested parties.

A history of the decision making process is maintained and archived. By recording various attempted solutions and their effectiveness, solutions are only tried once without going in circles. The resolution process of similar issues can be quickly looked-up by anyone, preventing similar mistakes from being committed repeatedly.

But it should not be easier to track them verbally or on paper.

Even with its clear advantages, computer-based systems are often unsuccessful. The main obstacle is their lack of comparative ease-of-use. Remember: The grapevine and Post-It systems are readily available at all times. Given the schedule pressure engineers work under and the amount of work that needs to be done, if you had the choice to report a relatively simple problem, which process would you use:

1. Walk over to the person who has to solve the problem and verbally report it.
2. Describe the problem on a Post-It note, then give it to that same person (and if that person is not there, stick it in the middle of his or her computer screen).
3. Enter a description of the problem in the issue tracking database and never leave your workstation?

It should not take longer to submit an issue than to fix it.

You would probably use the one that requires the least amount of time and effort. If you want your team to use a computer-based issue tracking system successfully, then select one that causes the smallest disruption in their normal work flow. Choose one that is a simple or transparent extension of their normal behavior and tools they already use.

I was involved in a project where the issue tracking system used a proprietary X-based graphical interface. It took about 15 seconds to bring up the entire interface on your screen. You were then faced with a series of required menu selections to identify the precise division, project, system, sub-system, device and functional aspect of the problem, followed by several other dialog boxes to describe the actual issue. Entering the simplest issue took *at least* three to four minutes. And the system could not be accessed when working from home on dial-up lines. You can guess how successful that system was...

Email-based systems have the greatest acceptance.

The systems that have the most success invariably use an e-mail-based interface, usually coupled with a Web-based interface for administrative tasks and reporting. Everyone on your team uses e-mail. It is probably already the preferred mechanism for discussing issues when members are distributed geographically or work in different time zones. Having a system that simply captures these e-mail messages, categorizes them and keeps track of the status and resolution of individual issues (usually through a minimum set of required fields in the e-mail body or header), is an effective way of implementing a computer-based issue tracking system.

METRICS

Metrics are essential management technologies.

Managers love metrics and measurements. They have little time to personally assess the progress and status of a project. They must rely on numbers that (more or less) reflect the current situation.

Metrics are best observed over time to see trends.

Metrics are most often used in a static fashion: "What are the values today?" "How close are they to the values that indicate that the project is complete?" The odometer reports a static value: How far have you travelled. However, metrics provide the most valuable information when observed over time. Not only do you know where you are, but also you can know how fast you are going, and what direction you are heading. (Is it getting better or worse?)

Historical data should be used to create a baseline.

When compared with historical data, metrics can paint a picture of your learning abilities. Unless you know how well (or how poorly) you did last time, how can you tell if you are becoming better at your job? It is important to create a baseline from historical data to determine your productivity level. In an industry where the manufacturing capability doubles every 18 months, you cannot afford to maintain a constant level of productivity.

Metrics can help assess the verification effort.

There are several metrics that can help assess the status, progress and productivity of functional verification. Two have already been introduced: code and functional coverage.

Code-Related Metrics

Code coverage may not be relevant.

Code coverage measures how thoroughly the verification suite exercises the source code being verified. That metric should climb

steadily toward 100 percent over time. From project to project, it should climb faster, and get closer to 100 percent.

However, code coverage is not a suitable metric for all verification projects. It is an effective metric for the smallest design unit that is individually specified (such as an FPGA, a reusable component or an ASIC). But it is ineffective when verifying designs composed of sub-designs that have been independently verified. The objective of that verification is to confirm that the sub-designs are interfaced and cooperate properly, not to verify their individual features. It is unlikely (and unnecessary) to execute all the statements.

The number of lines of code can measure implementation efficiency.

The total *number of lines of code* that is necessary to implement a verification suite can be an effective measure of the effort required in implementing it. This metric can be used to compare the productivity offered by new verification technologies or methods. If they can reduce the number of lines of code that need to be written, then they should reduce the effort required to implement the verification.

Lines-of-code ratio can measure complexity.

The *ratio of lines of code* between the design being verified and the verification suite may measure the complexity of the design. Historical data on that ratio could help predict the verification effort for a new design by predicting its estimated complexity.

Code change rate should trend toward zero.

If you are using a source control system, you can measure the *source code changes* over time. At the beginning of a project, code changes at a very fast rate as new functionality is added and initial versions are augmented. At the beginning of the verification phase, many changes in the code are required by bug fixes. As the verification progresses, the rate of changes should decrease as there are fewer and fewer bugs to be found and fixed. Figure 2-19 shows a plot of the expected code change rate over the life of a project. From this metric, you are able to determine if the code is becoming stable, or identify the most unstable sections of a design.

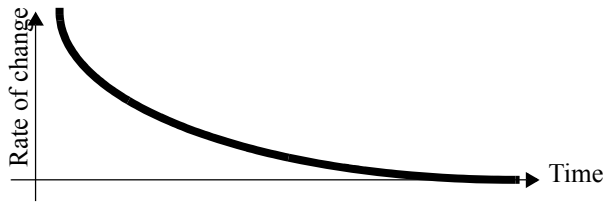


Figure 2-19. Ideal code change rate metric over time

Quality-Related Metrics

Quality is subjective, but it can be measured indirectly.

Quality-related metrics are probably more directly related with the functional verification than other productivity metrics. Quality is a subjective value, yet, it is possible to find metrics that correlate with the level of quality in a design. This is much like the number of customer complaints or the number of repeat customers can be used to judge the quality of retail services.

Functional coverage can measure testcase completeness.

Functional coverage measures the range and combination of input and output values that were submitted to and observed from the design, and of selected internal values. By assigning a weight to each functional coverage metric, it can be reduced to a single functional coverage grade measuring how thoroughly the functionality of the design was exercised. By weighing the more important functional coverage measures more than the less important ones, it gives a good indicator of the progress of the functional verification. This metric should evolve rapidly toward 100 percent at the beginning of the project then significantly slow down as only hard-to-reach functional coverage points remain.

A simple metric is the number of known issues.

The easiest metric to collect is the *number of known outstanding issues*. The number could be weighed to count issues differently according to their severity. When using a computer-based issue tracking system, this metric, as well as trends and rates, can be easily generated. Are issues accumulating (indicating a growing quality problem)? Or, are they decreasing and nearing zero?

Code will be worn out eventually.

If you are dealing with a reusable or long-lived design, it is useful to measure the *number of bugs found during its service life*. These are bugs that were not originally found by the verification suite. If the number of bugs starts to increase dramatically compared to historical findings, it is an indication that the design has outlived its useful life. It has been modified and adapted too many times and needs to be re-designed from scratch. Throughout the normal life

cycle of a reusable design, the number of outstanding issues exhibits a behavior as shown in Figure 2-20.

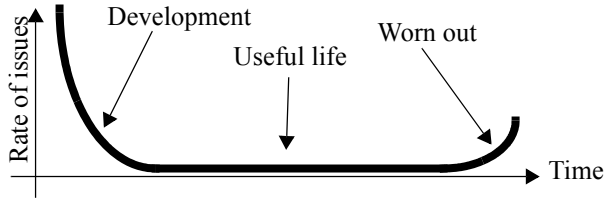


Figure 2-20. Number of outstanding issues throughout the life cycle of a design

Interpreting Metrics

Whatever gets measured gets done.

Because managers rely heavily on metrics to measure performance (and ultimately assign reward and blame), there is a tendency for any organization to align its behavior with the metrics. That is why you must be extremely careful to select metrics that faithfully represent the situation and are correlated with the effect you are trying to measure or improve. If you measure the number of bugs found and fixed, you quickly see an increase in the number of bugs found and fixed. But do you see an increase in the quality of the code being verified? Were bugs simply not previously reported? Are designers more sloppy when writing their code since they'll be rewarded only when and if a bug is found and fixed?

Make sure metrics are correlated with the effect you want to measure.

Figure 2-21 shows a list of file names and current version numbers maintained by two different designers. Which designer is more productive? Do the large version numbers from the designer on the left indicate someone who writes code with many bugs that had to be fixed? Or, are they from a cautious designer who checkpoints changes often?

Figure 2-21. Using version numbers as a metric

alu_e.vhd	1.15	cpuif_e.vhd	1.2
alu_rtl.vhd	1.234	cpuif_rtl.vhd	1.4
decoder_e.vhd	1.12	regfile_e.vhd	1.1
decoder_rtl.vhf	1.155	regfile_rtl.vhf	1.7
dpath_e.vhd	1.7	addr_dec_e.vhd	1.3
dpath_rtl.vhd	1.176	addr_dec_rtl.vhd	1.6

On the other hand, Figure 2-22 shows a plot of the code change rate for each designer. What is your assessment of the code quality from

the designer on the left? It seems to me that the designer on the right is not making proper use of the revision control system.

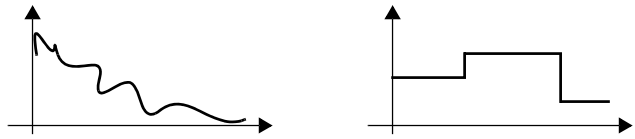


Figure 2-22. Using code change rate as a metric

SUMMARY

Despite reporting many false errors, linting and other static code checking technologies are still the most efficient mechanism for finding certain classes of problems.

Simulators are only as good as the model they are simulating. Simulators offer many performance enhancing options and the possibility to co-simulate with other languages or simulators.

Assertion-based verification is a powerful addition to any verification methodology. This approach allows the quick identification of problems, where and when they occur.

Verification-specific SystemVerilog features offer an increase in productivity because of their specialization to the verification task and their support for coverage-driven random-based verification.

Use code and functional coverage metrics to provide a quantitative assessment of your progress. Do not focus on reaching 100 percent at all cost. Do not consider the job done when you've reached your initial coverage goals.

Use a source control system and an issue tracking system to manage your code and bug reports.

CHAPTER 3 THE VERIFICATION PLAN

In this chapter, I describe the verification plan as a specification of the functional verification process and of the testbench infrastructure that will be necessary to support it. It is used to define what is first-time success, how a design is verified and which testbenches are written.

The design project that sits before you will propel your company to new levels of market share and profitability. A few system architects have designed and specified a system that should meet performance and cost goals. Several design leaders, using the system specification, have been working on writing detailed functional specification documents for each of the ASICs and FPGAs that are required to build this new product. Teams of hot-shot hardware designers are being assembled to implement each ASIC or FPGA. Using the detailed specification documents for each device, they are coming up with a detailed implementation schedule. So far, it appears that the project will meet its production deadline.

You are in charge of the verification for this design. Not only must this product be on time, but also it must be functionally correct. The commercial success and profitability of the product depends on it. You have been asked by the project manager to produce a detailed schedule for the verification and define your staffing requirements. How can you determine either?

THE ROLE OF THE VERIFICATION PLAN

Traditionally, verification is an ad-hoc process.

In a traditional verification process, your decision would be simple. In fact, your own position would not exist. Verification would be left to each hardware designer to do as they wish. It would be performed as time allows. And everybody's fingers would be crossed hoping that system integration will be smooth and that any serious design flaws can be fixed or worked-around by the software. Many devices would be implemented in FPGAs, trading additional per-unit costs for flexibility in fixing problems later found during system integration.

Technologies exist to help determine when you are done.

The technologies described in the previous chapter will help during your verification effort. Code coverage, functional coverage, bug discovery rate and code change rates are metrics that indicate how much progress you have made toward your goal. But they are like stock market indices or batting averages: They provide a snapshot of the current situation and, if recorded over time, show trends and progression. However, they cannot be used to predict the future.[1]

Specifying the Verification

You need a method to determine when you will be done.

Today's question is about producing a schedule. You must determine, as reliably as possible, when the verification will be completed to the required degree of confidence. Unless you have a detailed specification of the work that needs to be accomplished, you cannot determine how many people you need, nor how long it is going to take or even if you are doing work that needs to be done. That's what the verification plan is about.

Start from the design specification.

Before you can decide on a plan of attack for the verification, a specification document for the design to be verified must exist. And it must exist in written form. "Folklore" specifications that describe the design as, "The same thing as we did before, but at twice the clock rate and with these additional features." are insufficient. Often, the design specification is implemented using two separate documents written at different abstraction levels.

1. However, many financial and sports pundits make a good living predicting an essentially random process. With enough pundits, you can always find one that has made the correct "prediction".

- The first is the architectural specification, which details the functional requirements of the device.
- The second is the implementation specification, which describes the particular implementation of the architecture down to the block level.

The verification plan can start to be written once the architectural specification document is complete. It can be augmented with implementation-specific testcases once the implementation document is complete.

The specification document is the golden reference.

The specification document is the common source for the verification and implementation efforts. It is the golden reference and the rule of law. Later, when discrepancies are found between the response expected by the testbench and the one produced by the design under verification, the specification document arbitrates and decides which one has the correct answer. If necessary, the specification document should be elaborated to remove any ambiguity. The specification document must exist before the implementation. The implementation must follow the specification. If the specification depends on or is a consequence of the implementation, it will be impossible to verify because the specification will change every time the implementation changes.

The verification plan is the specification document for the verification effort.

Today's million-gate ASIC and SoC designs cannot proceed without a detailed specification document being written first. With the verification effort being 100 percent to 200 percent of the RTL design effort, why should it proceed without a specification document of its own? The verification plan is the specification document for the verification effort.

Defining First-Time Success

If, and only if, it is in the plan, will it be verified.

The verification plan provides a forum for the entire design team to define what *first-time success* is. It is a mechanism that ensures all essential features are appropriately verified. If you want first-time success, you must identify which features must be exercised under which conditions and what the expected response should be. The verification plan documents which features are a priority and which ones are optional. In the face of schedule pressure, the decision to drop features from the first-time success requirements becomes a conscious one. The alternative is to live with whatever happens to

work when the decision to ship the design cuts off the verification effort like a guillotine. Some of the features, essential for market acceptance, might fall in the basket.

From the verification plan, a detailed schedule can be created.

The verification plan creates a line in the sand that cannot be crossed without endangering the success of the project in the market place. Once the plan is written, you know how many testcases must be created, how complex they need to be and how they depend on each other. You can define a detailed verification schedule, and allocate tasks to resources, parallelizing verification as much as possible. Once the RTL passes all of the testcases, and you are satisfied with the coverage and bug-rate metrics, the design can be shipped. Not before.

The team owns the verification plan.

It is important for everyone involved with the design project to realize that they have a stake in the verification plan. The responsibility of an RTL designer is not to design RTL. That's only a means to an end. His or her responsibility is to produce a working design. The entire design team must contribute to the verification plan, to make sure that it is complete and correct.

This process is not revolutionary.

The process used to write a verification plan is not new. It has been used for decades by NASA, the FAA and aerospace companies to ensure that the ultra-reliable systems they were implementing met their specifications. This process has been used for software as well as for hardware designs.

LEVELS OF VERIFICATION

Verification can be performed at various levels of granularity.

The first question, when planning the verification, is to determine the level of granularity for the verification effort. A design is potentially composed of several levels. Some have a physical partition, such as printed circuit boards, FPGAs and ASICs. Others have a logical partition, such as synthesized units and blocks, reusable cores or sub-systems. As illustrated in Figure 3-1, each level of granularity is best suited for a particular verification objective.

Deciding between levels of granularity involves trade-offs.

Smaller partitions are easier to verify because they offer greater controllability and observability. It is easier to set up interesting conditions and state combinations and to observe if the response is as expected in a block than in a system. With larger partitions, the

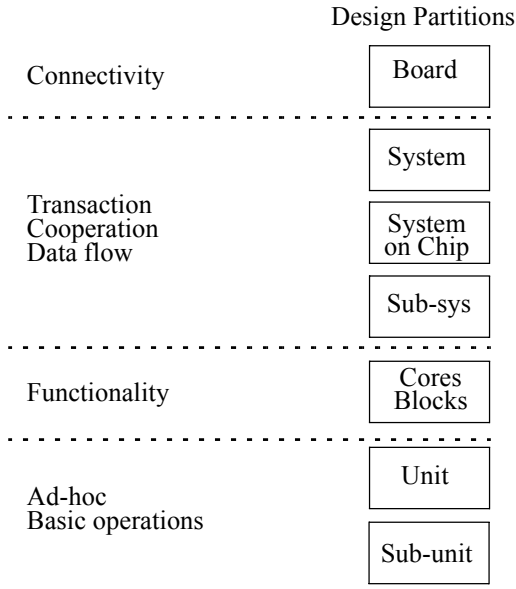


Figure 3-1. Application of different levels of verification

integration of the smaller partitions it contains is implicitly verified at the cost of lower controllability and observability.

Verifying at a given level of granularity requires stable interfaces.

Because the verification requires a significant implementation effort, any partition being verified must have relatively stable interfaces and intended functionality. If the interfaces keep changing, or functionality keeps being moved from one partition to another, the testbenches will constantly need to be changed with little progress being made. Once you've decided on specific partitions to be verified, their interface and overall functionality must be specified early on and remain as stable as possible. Ideally, each verified partition should have its own specification document or, at a minimum, its own section in the specification document.

Unit-Level Verification

Implementation determines the content of this partition.

Design units are *modules*. They are created to facilitate the implementation or the synthesis process. They vary from the relatively small (e.g., FIFOs and state machines) to the complex (e.g., PCI slave interface and DSP datapaths). Their interfaces and functionality tend to vary a lot over time, as implementation details highlight

shortcomings in the initial design. They usually do not have an independent specification document to verify against either.

Use ad-hoc verification for design units.

Because these design units do not have a specification document to very against, they are better left to an ad-hoc verification process. The designer himself verifies the basic operation of the unit by proving assertions embedded in the unit or by using casual testbenches. The objective of this verification is to ensure that there are no syntax errors in the RTL code, and that basic functionality is operational. It is not to create a regressionable test suite and obtain high code coverage.

They are too numerous to thoroughly verify independently.

The high number of design units in any project makes a verification process implemented at that level too time consuming. Each would require a custom verification environment, as described in Chapters 5 and 6. The precious verification resources would spend an inordinate amount of time creating stimulus generators and response monitors for a myriad of ever-changing interfaces. Writing a lot of simple testbenches is just as much work, if not more, as writing a few complex ones. And verification at the subsystem or system-level would still be required to verify the integration of these design units.

Unit-level verification may be required in some cases.

Not all units are created equal. For the highly sensitive and complex functional units, it may be more efficient to perform unit-level verification to have sufficient levels of controllability and observability and reach the desired level of confidence. A design unit is then considered a *block*.

Block and Core Verification

Design blocks are verified independently.

A design block is composed of one or more design units. A design block is the smallest partition to be independently verified. It is that independent verification that differentiates a unit from a block. The verification plan identifies the design blocks. Identifying the appropriate blocks is critical in balancing the total number of verification environments and testbenches that will need to be written and the required controllability and observability to verify the complete design. Blocks need not all be of the same size nor at the same level of design hierarchy. Some blocks may be large, others may be small, but they tend to require a similar amount of verification effort.

Reusable design cores are independent of any particular use.

Reusable cores are blocks designed to an independent specification. They are intended to be used as-is and unchanged in many different designs. Their reusability can be limited to a single product, the entire product family, or they could be applicable to any product requiring their functionality. They must be designed—and thus verified—independent of any one usage. It is a good idea to use assertions (see "Assertions" on page 57) to specify restrictions and requirements on the inputs of reusable components. They help ensure that the reused components are always used as intended.

Architect the design to facilitate block-level verification.

Your design should be architected to make block-level verification as relevant and complete as possible. Partition the design so the features to be verified are completely contained within a block and can be verified on a stand-alone basis. Once verified, these features can be assumed to work during the verification of the higher levels. If the features to be verified at the block level require interaction with other blocks, they have to be re-verified at a higher-level where the features are fully contained, to ensure that the integration correctly implements them.

Minimize the number of unique interfaces.

Reusable components and blocks should be designed using standardized interfaces. These interfaces can be designed to standard on-chip buses, or industry-standard external physical interfaces. The verification components used to stimulate and monitor these interfaces can be themselves reused across the various verification environments used to verify different reusable components, different blocks or the systems where they are used. The verification effort can be leveraged across multiple components, thus minimizing the overall investment in verification. Chapter 6 will detail how to architect a testbench to promote the creation and use of reusable verification components.

Blocks need a regression test suite.

Blocks are expected to be functionally correct. When they are modified, either to fix problems that were found, or to enhance their functionality, you must make sure that they remain functionally correct. This is accomplished by implementing a *regression suite* that verifies the correctness of the block after any modification. Checking the equivalence of the new version with the previous version using formal verification would not really work unless the modifications were not functional. Adding functionality or fixing problems, by definition, makes the new version of the design not equivalent to the previous one.

They need thorough code and functional coverage.

Design blocks must be verified as thoroughly as possible. Their functionality must be assumed as correct when performing system-level verification. A system-level verification environment is the wrong environment to verify the functionality of a design block.

ASIC and FPGA Verification

The physical partition is an ideal verification level.

ASICs and FPGAs are physical partitions. They form a natural partition for verification because their interfaces and functionality do not change very much after the initial specification of the system and the completion of their specification documents.

They may have to be treated as systems.

The ever increasing densities offered by semiconductor technology enables ever increasing integration of complex functionality into a single device. To manage this complexity from a design and verification standpoint, devices are often designed as a collection of independently designed and verified blocks, usually reusable but not necessarily so. In that case, the ASIC is called a System-on-a-Chip (SoC) and its verification resembles a system-level verification process, as described in the next section. The bulk of the functional verification is performed using block-level verification.

FPGAs now require an ASIC-like verification process.

Traditionally, FPGAs were able to survive an ad-hoc, or even a completely missing, verification process. Their ease of programmability, often without additional component costs, allowed their functionality to be modified up to the last minute. But today's million-gate FPGAs, even with only 50 percent effective usage, can implement functions that are too complex to verify and debug during integration. Their functionality must be verified from the RTL code, before synthesis and implementation.

System-Level Verification

A system need not follow physical boundaries.

Everybody's definition of a *system* is different. In this book, a system is a *logical* partition composed of independently verified blocks or sub-systems. A system could thus be composed of a few reusable components and cover a subset of an SoC ASIC. A system could also be composed of several ASICs physically located on separate printed circuit boards, as illustrated in Figure 3-2.

The verification focuses on interaction.

Individual blocks are specified and designed by separate individuals or teams with assumptions about how they will interact with other blocks. These assumptions made by different people are a

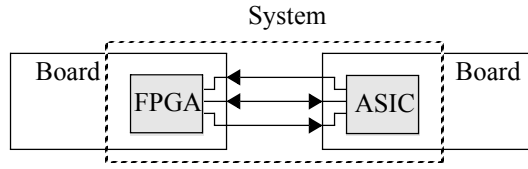


Figure 3-2. Logical system partition

prime source of bugs. System-level verification thus focuses on the interactions among the individual blocks instead of the functionality implemented in each one. The latter is better verified at the block-level. The system verification engineer has to rely on the individual blocks being functionally correct.

The testcase defines the system.

Since systems are logical partitions, they can be composed of any number of blocks, regardless of their physical location. Which system to use and verify depends on the testcases that are determined to be interesting and significant. To minimize the simulation overhead, it is preferable to use the smallest possible system necessary to execute the specified testcase. However, the number of possible systems being very large, a set of "standard" systems should be defined. The same system is used for many testcases even if, in some cases, some of the included blocks are not required.

Board-Level Verification

Board-level models are generated from the board design tool.

The primary objective of board-level verification is to confirm that the "system" captured by the board design tool is correct. Unlike a logical system model, the model for the board design must be automatically generated by the board capture tool. When verifying the board design, or any other physical partition, you must make sure that what is being verified is what will be manufactured. There must be a direct link between the captured design and what is simulated. Automatic generation of the board-level model from the capture tool provides that link. A logical system model has no such restriction: It can be manually generated for the system of interest.

Many components on a board do not fit in a digital simulation environment.

The main difficulty with board-level models is obtaining suitable models for all the components. Also, generating a model out of a board design tool involves introducing approximations. For example, how do you represent capacitors in a digital simulation environment? Analog devices, connectors, opto-couplers and other

components used in board-level designs do not translate easily in a digital simulation environment either.

VERIFICATION STRATEGIES

Decide on a black- or white-box approach for various levels of granularity.

Given the functionality that needs to be verified, you must decide on a strategy for carrying out the verification. You must decide on the level of granularity where verification will be accomplished. You must also decide on the invasiveness of the verification approach that will be used for each level of granularity. Testcases can be either white-box or black-box, depending on the visibility and knowledge you have of the internal implementation of each unit under verification (see "Black-Box Verification" on page 11 and "White-Box Verification" on page 13).

Decide on the level of abstraction where the testcases will be specified.

You also need to decide the level of abstraction where the bulk of the verification will be performed. With higher levels of abstraction, you have less detailed control over the timing and coordination of the stimulus and response, but it is easier to generate large amount of stimulus and observe the response over a long period of time. If detailed controls are required to perform certain testcases, it may be necessary to work at a lower level of abstraction.

A processor interface could be verified at the cycle or device driver level.

For example, verifying a processor interface can be accomplished at the individual read and write cycle levels. But that requires each testcase to have an intimate knowledge of the memory-mapped registers and how to program them. That same interface could be driven at the device driver level. The testcase would have access to a set of high-level procedural calls to perform complete operations. Each operation is composed of many individual read and write cycles to specific memory-mapped registers, but the testcase is removed from these implementation details.

Verifying the Response

Plan how you will check the response.

Deciding how to apply the stimulus is relatively easy. You are under complete control of its timing and content. It is verifying the response that is difficult. You must plan how you will determine the expected response, then how to verify that the design provided the response you expected. The section titled, "Self-Checking Testbenches" on page 292 suggests several techniques for implementing output verification.

Some responses are difficult to verify in the simulation.

Throughout this book, implementing self-checking testbenches is recommended (see "Simple Output" on page 216). But, it can sometimes be difficult for a testbench to verify a response that can be recognized immediately as right or wrong by a human. For example, verifying a graphic engine involves checking the output picture for expected content. A self-checking simulation would be very good at verifying individual pixels in the picture. But a human would be more efficient in recognizing a filled red circle. The verification strategy must find a way to automate these types of testcases.

Detect errors as early as possible.

It may be more efficient to have the simulation produce a set of outputs that can be later compared against a set of reference outputs. The result of a simulation can be further processed outside of the simulator to determine success or failure. However, it is more efficient to detect problems as early as possible. When the response is checked within the simulation, the error is identified while the model is near the state that produced the error. It is then easier to diagnose and fix the cause of the error.

FROM SPECIFICATION TO FEATURES

Identify features.

The first step in writing a verification plan is to identify the features that will be verified. From the specification document, you enumerate all the features that are described and thus must be verified. Other team members, especially the system architects and RTL designers, contribute additional features to be verified. These additional features may not have been obvious in the specification to someone unfamiliar with the purpose or characteristics of the design. Other features may become significant once a particular implementation is chosen. In *The Art of Verification*[1], Haque, Michelson and Khan propose using a methodical approach for extracting significant and relevant features to verify by first looking at the interfaces, then the functions, then finally the corner cases implied by the chosen architecture.

1. Faisal Haque, Jon Michelson and Khizar Khan, "*The Art of Verification with VERA,*" http://www.verificationcentral.com

Enumerate interface-based features.

For every interface on the design to be verified, enumerate every feature it suggests that must be verified. The interface-based features can be obtained by asking questions such as:

- What transactions must be applied?
- What range of values?
- What sequences of transactions?
- What are the relevant transaction densities?
- What protocol violations should the design be able to sustain?
- What are the relevant interactions between this interface and other interfaces or internal design structures?
- Do transactions on an interface need to be synchronized with those of another interface?

A subset of the interface-based feature list for a Universal Asynchronous Receiver Transmitter (UART) is shown in Sample 3-1.

Sample 3-1. Some of the interface-based features of a UART design

1. The Clear-To-Send (CTS) pin must be asserted when the UART can accept a new word to be transmitted via the CPU interface.
2. The Data Terminal Ready (DTR) pin must be asserted when there is a received word ready to be read by the CPU interface.
3. Read and write cycles to addresses other than 0 through 4 are ignored.
4. Back-to-back read/read, read/write, write/write and write/read cycles within the address space are supported.
5. All bits in the configuration registers are readable, writable and non-volatile.

Identify function-based features.

Following the major data paths through the design[1], enumerate every transformation and decision that must be verified. The function-based features can be obtained by asking questions such as:

- What are all the relevant configurations?

1. As specified in the architecture specification document, *not* in the implementation.

- What are the possible transformations that can be performed on the data?
- What are the possible sequences of transformation?
- What are the sensitive data values for triggering transformations?
- What are the sensitive values that affect each transformation?
- Where should the transformed data end up?
- How is the data ordering affected?
- What error detection mechanisms exist and how are they triggered?
- How do error mechanisms report errors?
- What happens to erroneous data?

A subset of the function-based feature list for a UART is shown in Sample 3-2.

Sample 3-2. Some of the function-based features of a UART design

1. Data bits are sent and received serially with the least significant bit first.
2. Data bytes are sent in the same order in which they were written.
3. Data bytes are read in the same order in which they were received.
4. Parity is generated according to configured mode.
5. Parity is checked according to configured mode.

List architecture-based features.

Finally, based on detailed knowledge of the architecture of the design, identify the conditions that will stress the design and push it toward its limit. The architecture-based features can be obtained by asking questions such as:

- Can I overflow or underflow a buffer? If so, what should happen?
- Where are the resource bottlenecks?
- Can multiple requests for these resources occur at the same time?
- Can a transformation path affect, prevent or block another?

A subset of the architecture-based feature list for a UART is shown in Sample 3-3.

Sample 3-3. Some of the architecture-based features of a UART design

1. Receiving one more byte while the receive buffer is full will cause that byte to be dropped.
2. The Clear-to-Send (CTS) signal reflects the status of the transmit buffer (asserted when not full).
3. Data is received and transmitted in full-duplex.

Label each feature.

Features should be labeled and have a short description. The feature should be described in terms of what conditions need to be verified and the expected result, not how it is to be implemented. Each feature should be cross-referenced to the section or paragraph in the specification document that describes it in detail. Ideally, the specification document should also contain a cross-reference to the feature list in the verification plan. Specify features for the proper level of verification. The feature label should be used in error messages when it is found to be violated. Including feature labels in error messages will help in identifying what was assumed to have gone wrong and in assessing if the behavior is indeed incorrect.

Assign features to a suitable verification level.

When enumerating features, be careful to include them in the verification plan for the proper verification level. Some features are better verified at the block level, while others must be verified at the (sub)system level. Very often, there will be a large number of features concerned with verifying a critical function or unit in your design. If the design partition implementing that function or containing that unit is not being verified independently, now is the time to reconsider your verification approach. It may be an indication that the unit needs to be verified independently to achieve the necessary level of confidence.

Block-Level Features

They are fully contained within the block being verified.

A block can be a unit, a reusable component, or an entire ASIC. Block-level features are fully contained within the block being verified. They do not involve system-level interaction with other blocks. Their correctness can be determined without depending on a subsequent verification of the integration of the block into a high-level system.

The bulk of the features will be block-level features. These features are assumed to be functional when the block is used in a system-level verification.

System-Level Features

Minimize system-level features.

A system can be a subset of an ASIC, a few ASICs from different boards, an entire board design or the complete product. Because of the large size and long runtime of system-level simulations, it is necessary to minimize the features verified at this level. Whenever something is identified as a system-level feature, question whether it can be verified as a block-level feature instead. For example, in the design illustrated in Figure 3-3, the *MX* block can select between the data from blocks *ID0* or *ID1* under software control. Is the switching feature a system-level feature? The answer is *no*. The switching feature is entirely contained within the *MX* block and is thus a block-level feature.

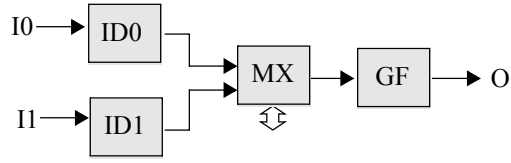


Figure 3-3. Example of a system structure

System-level features include connectivity, flow control and inter-operability.

System-level features are usually limited to connectivity, flow-control and inter-operability. For example, the connectivity from the input ports to the output port would be a system-level feature. In verifying the connectivity, it is necessary to switch the input from the *ID0* stream to the *ID1* stream. But the switching is not the primary objective of the verification and would be assumed to work.

Another system testcase would be verifying that full input FIFOs in the *MX* block creates back-pressure through the *ID0* and *ID1* blocks and stops the flow of data until the congestion clears.

Error Types to Look For

Assume design tools do not introduce functional errors.

When listing features to be verified, there is an implicit assumption about the errors that are likely to occur and should be found. Functional verification must focus on finding functional errors in the design. It is not the responsibility of functional verification to make

sure that the design tools are bug-free. Functional simulation ensures that the design was implemented as specified without interpretation errors or problems introduced by the designers. For example, running all functional testbenches on the gate-level netlist only verifies that the synthesis tool works properly. Formal verification and static timing analysis are better technologies to accomplish this task.

Likely errors are different based on the capture technology used.

The types of errors that can be made are different when using different capture technologies. When schematic capture is used, connectivity errors, such as reversed bit orders in a bus, or misconnected individual bits within a bus, are very common. In an RTL coding and logic synthesis environment, this type of error is not likely to occur: If a bus is properly connected, either all the bits work, or none do. Linting can detect some connectivity problems such as multiple drivers on a wire or an output that goes nowhere and would be a better technology for identifying these types of problems.

Look for functional errors.

Common errors in a synthesis-based design flow include wrong polarities, protocol violations or incorrect computations. The type of stimulus that proved useful in the days of schematic capture, such as walking ones and zeroes may not be as useful in an RTL design verification. A pair of patterns of alternating ones and zeroes, for example “0xAAAA” followed by “0x5555”, is usually sufficient.

Using *signatures* in the data stream is another efficient technique to detect functional errors. A signature can be as simple as a sequential number to help detect missing or repeating data items. A signature can also encode either the source or the expected destination of a data item. For example, the data associated with an address in a write cycle could contain a portion of the address and an identification of the bus master issuing the cycle. The section titled, “Data Tagging” on page 295 details how to use signatures to verify a class of designs.

Prioritize

Prioritize the features.

Not all features are created equal. Once they are enumerated, they must be prioritized. Some features are *must-have* for the design to properly function or to meet the demands of the market. This is the

stage that defines first-time success. These features must operate properly for the design to be shipped. The completion of the verification of these features gates the successful completion of the project and the testbenches verifying these features are often on the critical path. The *must-have* features need to be thoroughly verified for all possible configuration and usage options.

Less important features receive less attention.

The *should-have* features are secondary for the commercial success of the design. They may simply offer expansion capabilities or differentiation from the competition. The main objective is to verify their basic functionality for correct operation. If time and resources allow, more detailed verification of these features may be accomplished. The verification of these features may be cancelled if schedule pressure forces the reallocation of resources to the verification of more important features.

Some features are verified only as time allows.

The *nice-to-have* features are purely optional. They are verified only as time allows, usually in a primitive fashion. The reality of today's design schedule almost guarantees that they'll never be verified!

Make an informed decision when cutting back on the verification effort.

The prioritization of the features to be verified lets a project manager make informed decisions when schedule pressures make it necessary to eliminate some planned activities. The verification effort can be trimmed starting with features that were predetermined to be less important. If a greater impact of the project completion date is required and *must-have* features are dropped from the verification, the decision will be a conscious one as these priorities were clearly identified as critical to the initial marketing objectives. Cutting the verification effort of *must-have* features requires a conscious re-evaluation of the marketing objectives for the project.

Design for Verification

Hard-to-verify features will be identified.

At this stage of the verification planning, hard-to-verify features will be identified. They can be difficult to verify because the chosen partition lacks suitable controllability or observability of the features. An example would be the verification that an embedded 64-bit counter properly rolls over and that the processing algorithm works properly across the roll-over point. The difficulty may be because of a poor choice in verification granularity. In that case, a smaller partition containing the hard-to-verify features should be used. The difficulty may also be due to the choice of implementa-

tion architecture or an artifact of the design itself. If a smaller partition cannot be used, or would not ease the verification of these features, a grey- or white-box approach must be taken.

Modify the design to aid verification.

The advantage of planning the verification up front is that you can still influence the implementation of the design. If some features prove to be too difficult to verify given the current architecture and feature set of the design, have the design modified to include additional features to aid in their verification. Hardware design engineers will no doubt complain about adding functionality that is not really needed by the design. However, if the alternative is to create a design you cannot verify, what choice do they have? These features have always proven to be useful during lab integration of sample parts.

Provide state pre-load functions.

If the design contains long counters or other state conditions which require hundreds or thousands of cycles to reach from reset, make sure they can be pre-loaded to an arbitrary value via a memory-mapped register. Ideally, their current value should be available for read back through the same register. In the previous example, a series of 8 bytes in the address space of the design could be allocated to pre-loading and reading back the value of the 64-bit counter.

Provide datapath by-pass paths.

The correct implementation of long data paths can also be difficult to verify if you do not have detailed control over all the operands. For example, speech synthesizers are simple digital signal processing designs with a datapath that shapes random noise[1]. You have complete control over the coefficients applied to the data samples to form specific sounds. However, you do not have control over one critical element: the primary input data value. That's an internally-generated random number. To properly verify the operation of this datapath, you need control over its initial input value.

As shown in Figure 3-4, the design should include a mechanism to use a programmable constant input value instead of a random number as input to the datapath. Conversely, you should also be able to

1. It is used to produce consonant sounds, such as the *sh* sound. It is then mixed with a shaped base frequency used to produce vowel sounds, such as the *a* sound, which hopefully creates intelligible speech.

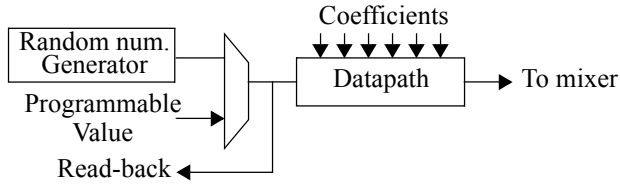


Figure 3-4. Verifiable datapath for a speech synthesizer

read back the output of the random number generator to ensure that it is indeed producing random numbers.

Pop quiz: Why is the read-back point located after the multiplexer that selects between the normal operation using the random number generator and the programmable static value, and not at the output of the random number generator?[1]

Provide sample points.

If observability is the problem but not controllability, adding sample points readable through memory-mapped registers can help ease the verification of some features. If the address space allocated to the design is at a premium, these sample points could be multiplexed into a single address location, using a second address to select which point is currently being sampled.

Provide error injection mechanism.

If the design includes error and exception detection mechanisms, you may want to have provisions to force the detection of an error or exception. For example, verifying the maskability of interrupts is very time consuming if the design has to be coerced into every exception condition. The same task is rendered considerably easier if a simple register write can manually raise the same interrupts. Of course, the task of verifying that the exception condition raises the interrupt remains. The decision to include error injection should be carefully considered. If it is for hardware verification only, it may not be properly documented for the software engineers. This feature may be accidentally turned on when a device driver writes a value that was thought to be inoffensive.

1. You want to verify that, when the datapath is put into normal operation mode, the multiplexer is functionally correct and the input value is indeed coming from the random number generator.

DIRECTED TESTBENCHES APPROACH

With directed testbenches, individual features are verified using individual testbenches. The stimulus is manually crafted to exercise that feature. The response is verified against the symptoms that would appear should the feature not be correctly implemented.

Use for small number of testcases.

Before you embark on the directed testbenches path, you need to consider its lack of scalability. This approach can be managed and completed if the total number of testcases is in the low hundreds. But as the number of testcases grows, so does the number of testbenches. A project with over a thousand identified testcases would require over a year to complete using a directed approach. For a larger number of testcases, some form of testbench automation is necessary to complete the task within an acceptable time frame. Currently, the best method of testbench automation is the coverage-driven random-based approach (see page 101).

Group into Testcases

Group features with similar verification requirements.

Features naturally fall into groups. Some features require similar configuration, granularity or verification strategy to perform their verification. To maximize productivity, these features should be grouped together and assigned to the same verification engineer. For example, all features related to the CPU interface should be grouped together. As another example, verifying the baud rate, number of data bits and parity generation of a UART falls within the same group. Each group of feature verification forms a *testcase*.

Cross-reference into the feature list.

Each testcase should be labeled and given a short description of its objective. Its description should contain a list of the features verified in this testcase. The feature list should also be annotated with cross-references to the testcases where a particular feature is being verified. If a feature does not have a cross-reference to a testcase, it is not being verified.

Define dependencies.

The description of a testcase should also contain a list of the features assumed to be operational and functionally correct. From these dependencies, you can determine the order in which the testcases must be written, and identify any parallelism opportunities in the testbench development effort.

Specify the testcase stimulus.

The sequence and characteristics of the stimulus for the testcase must also be described. For example, describe the various operations or bus cycles that must be performed. It is a good idea to fill all non-relevant or background data with random values or transactions.

Specify the acceptance criteria.

More than just the expected response, the testcase specification must state how the response will be determined as valid. This includes expected values, timing and protocol. For example, the output of a packet processor could be determined as correct solely on the basis of the destination address matching the output port where it appeared. Or, a more stringent requirement could be specified, such as packets from different sources showing up in the proper order and interleaved with a proper distribution.

Specify what errors to look for.

One of the more explicit ways of describing acceptance criteria is to state exactly which errors to look for. For example, making sure that a packet comes out with a correct CRC value. Another example is to describe events that are mutually exclusive, such as the assertion of the *full* and *empty* flags in a FIFO. Being explicit about what errors to look for lets a verification engineer, who is not intimately familiar with the design, implement a highly reliable testbench.

Inject errors to make sure they are detected.

Never trust a testbench that does not produce error messages. Every testcase should include some error injection mechanism to make sure that errors are detected and reported by the testbench. The absence of an error message would be a failure condition for that testcase. For example, a testcase verifying the parity generation in a UART should purposefully misconfigure the parity in the UART to make sure that the testbench detects a wrong parity. Of course, the testbench must not abort the simulation as soon as the error message is issued and must declare success if and only if the error message is issued.

Define functional coverage points.

The purpose of a directed testcase is implicit in its directness. The stimulus of a directed testcase is hard-coded. Therefore you know what it will do. If it executes without error, the targeted function will have been exercised. But what if the design changes in a way that the targeted function is no longer exercised without producing an error? For example, a directed testcase designed to fill a FIFO would no longer accomplish its goal should the size of the FIFO be increased. Directed testcases should include functional coverage

points to positively confirm that they continue to accomplish what they were intended to do.

From Testcases to Testbenches

Testcases naturally fall into groups.

Just like features, testcases naturally fall into groups. They require a similar configuration of the design, use the same abstraction level for the stimulus and response, generate similar stimulus, determine the validity of the response using a similar strategy, or verify closely-related features. For example, the testcase verifying that a UART properly transmits data can be grouped with the testcase that verifies its configuration controls. Both need similar stimulus (a variety of data words to transmit), and both verify the correctness of the output in a similar fashion (is the data value identical, with no parity error).

Group testcases into testbenches.

Each group of testcases is then divided into testbenches. A popular division, the one used in this book, is one testcase per testbench. The minimization of SystemVerilog compilation time, or the time spent back-annotating a large gate-level netlist with a correspondingly large Standard Delay File (SDF) may dictate that a minimum number of testbenches be created by grouping several testcases into a single testbench.

Cross-reference testbenches with testcases.

Each testbench should be labeled and uniquely identified. This identifier should be used as the filename where the top-level code for the testbench is implemented. For each testbench, enumerate the list of testcases it implements. Then cross-reference each testbench into the testcase list. The description of a testcase should contain the name of the testbench where it is implemented. If a testbench is not identified, a testcase has not yet been implemented.

Allocate each group to an engineer.

Regardless of the division of testcases into testbenches, allocate each group of testcases to a verification engineer. Testcases in the same group have similar implementation requirements. They can build on the implementation of previous testcases in the group. The first testbench takes the longest to write. But as engineers responsible for each testcase group gain experience and debug their verification infrastructure, a lot can be reused, often through cut-and-paste, in subsequent testbenches. The name of the individual to whom a testbench has been assigned should be recorded in the verification plan. That person is responsible for implementing the testbench according to its specification.

Verifying Testbenches

How do you verify that testbenches implement the verification plan?

The purpose of the verification effort and writing testbenches is to verify that a design meets its specification. If the verification plan is the specification for the verification effort, how do you verify that the testbenches implement their specification? How can you prevent a significant portion of a testcase from being skipped because of human error? Testbenches often include temporary code structures to bypass large sections to speed up the debugging of a critical section. How can you make sure that they are taken out, returning the testbench to implementing the entire set of testcases it is supposed to contain?

Verify testbenches through peer reviews.

As described in "The Human Factor" on page 5, one way to verify a transformation performed by a human (in this case, writing a testbench from a specification), is to provide redundancy. Once completed, testbenches should be reviewed by other verification engineers to ensure that they implement the specification of the testcases they contain. For more details, refer to section "Code Reviews" on page 29. The simulation output log should also be reviewed to ensure that the execution of the testbench follows the specification as well. To that effect, the testbench should produce regular notice messages. It should state what stimulus is about to be generated, and what error or response is being checked. The output log should ultimately contain, in a bullet form, the specification of the testcases that have been executed.

Directed testbenches may become obsolete.

What if there is a design change and a directed testbenches, although successful, no longer exercises the feature is was designed to verify? For example, the size of a memory or FIFO could be increased. Any testcase involved in verifying the correct operation of that memory or FIFO would still be successful, but it would no longer verify the entire memory or FIFO. How can you ensure that directed testcases remain relevant and useful?

Use functional coverage.

Another redundant path is functional coverage measurement. By specifying, through a functional coverage model, what you expect a directed testcase to accomplish, you can obtain a positive confirmation that the testcase was indeed executed. After a directed testbench is run, the functional coverage metrics should meet 100 percent of the goal. Since the stimulus was manually coded, it is deterministic and should fill 100 percent of the relevant and interesting coverage points. For example, a directed testcase that is sup-

posed to verify a FIFO should fill a functional coverage of that FIFO. Should it fail to fill or empty the FIFO, as defined in the coverage model, the functional coverage metric for the test will not reach 100 percent.

Measuring Progress

Testcase completion measures progress.

In a directed testbench approach, progress is measured using a simple table. On one dimension, all of the testcases are listed. On the other, the current status of each testcase is tracked throughout its lifetime: assigned, coded, running, passing, reviewed/covered. Figure 3-5 shows the progress of a directed testbench approach. Initially, little progress is made because the verification infrastructure is being developed and the design is being debugged. Once the first testcase completes successfully, the progress will accelerate as less and less bugs remain to be found, and more and more verification infrastructure code is reused. This acceleration may not translate into an accelerated testcase completion rate as testcases become increasingly complex to implement.

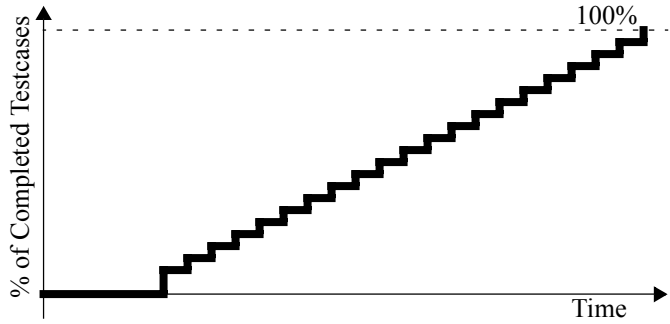


Figure 3-5. Progress of a directed testbench approach

When are you done?

The completion of all testcases does not necessarily indicate that the verification task is over. Code coverage metrics can indicate that the original set of testcases is not as thorough as imagined and additional testcases must be created to increase the code coverage scores to more acceptable levels. In reality, “done” is usually defined when you have to ship the design and you are confident enough that the must-have features are working properly.

COVERAGE-DRIVEN RANDOM-BASED APPROACH

The SystemVerilog productivity cycle ("Verification Language Technologies" on page 55) rests on constrained random verification. You can use SystemVerilog to implement a directed testcase approach as described in the previous section. Its high-level programming language constructs would facilitate the implementation of testcases. However, it would only increase the slope of the testcase completion curve somewhat (Figure 3-5), not alter the nature of the curve. Changing the curve itself requires changing how verification is approached.

Random verification still provides valid stimulus.

Random verification does not mean that you randomly apply zeroes and ones to every input signal in the design. This would not represent an accurate usage of the design and would not accomplish anything. With random verification, the inputs are subjected to valid individual operations, such as a read cycle or an ethernet packet. It is the sequence and timing of these operations and the content of the data transferred that is random. Through the addition of constraints, a random testbench can be steered toward exercising specific features.

Measuring Progress

There are too many testcases.

Today's multi-million gate ASICs contain hundreds of features to be verified for hundreds of different combinations of data values. Assuming a bug-free design and a team of highly productive engineers who can code and debug a self-checking testbench in three days, a team of 10 verification engineers (a rarity by today's standards) would require over seven months to implement 500 testcases. The number of testcases cannot be reduced. Throwing more engineers at the problem quickly produces diminishing returns. The only way to reduce the verification time is to write more testcases in less time. In other words, exercise the same functionality with less code.

Testcases exercise more than the target feature.

Although each testbench, when verifying a testcase, considers the target feature in isolation, applying stimulus to the design exercises other features at the same time. Since progress, in a directed approach, is tracked by associating features with testbenches, how can you track progress against features that are not explicitly coded and verified in a testbench?

Measure functional coverage, not features.

The solution is to measure progress against functional coverage points that will identify whether a feature has been exercised. The objective becomes filling a functional coverage model of your design rather than writing a series of testcases. You could fill this coverage model using large directed testbenches. Or you could let a random testbench create the testcases and exercise the features for you. Figure 3-6 shows progress using a coverage-driven approach with a random testbench against the traditional directed approach. The former trades-off longer initial testbench development time for more productive feature coverage in the long run. The promised ultimate productivity gain should not be measured on this qualitative plot: It depends highly on your commitment to this approach, and your experience in writing random generators that can be constrained easily (“Random Stimulus” on page 307).

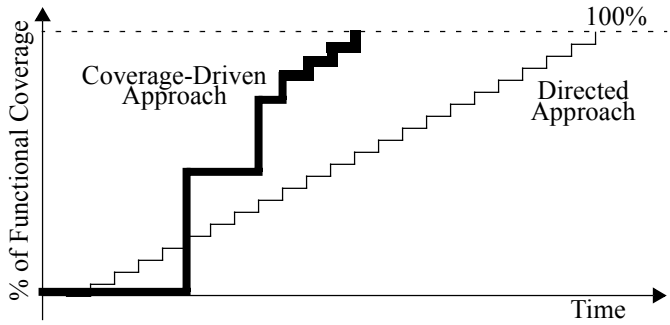


Figure 3-6. Progress of a coverage-driven testbench approach

You will develop more confidence.

Directed testcases can only find bugs you were looking for. Random simulations will create conditions that you have not thought of when writing your verification plan. They create unexpected conditions and hit corner cases. They also reduce the bias introduced by the verification engineer when coding directed testbenches. Instead of creating input sequences that are easy or familiar to code, they create more thorough stimulus. Because your design will have been exercised under a larger number of conditions (compared to a directed approach during the same time period), the overall quality of the design will be higher.

This approach requires commitment.

Using a constraint-driven approach requires commitment. Under pressure, it is too easy to fall back to writing directed testcases. A critical component of this approach is that you need to simulate your testbench and your design to know how much functional cov-

erage you have achieved. If the RTL model is not available on time (and it never is), how can you debug your self-checking random testbench? How can you show that the verification team is making progress towards its functional coverage goals? The easy answer is to start writing testcases as directed pseudo-random testbenches that implicitly fill functional coverage points. That puts you back on the staircase curve. A better approach is to stage the RTL delivery to enable simulations as early as possible and to use a transaction-level model of the design under verification. For more details on transaction-level models, see "Transaction-Level Models" on page 333.

From Features to Functional Coverage

Start with functional coverage.

In a coverage-driven approach, functional coverage is used to identify which testcases were executed instead of explicitly coding those testcases. Thus, it is important to implement functional coverage models and collect functional coverage measurements right from the start. Functional coverage is not like code coverage. The latter is often added to the verification process toward the end to measure how thoroughly the code is being exercised and to identify implementation code that was not exercised. Functional coverage is used from the beginning of the project to record which testcases and conditions were automatically created by the random generator. If you are not using functional coverage in tandem with your random environment, I'm afraid you are only doing directed testcases with random stimulus filling.

Measure symptoms of data indicative of feature.

Each feature presents a characteristic or symptom in the input data stream, the design configuration or the internal state of the design that must be exercised. Functional coverage must identify, then record, those characteristics and symptoms. Sample 3-4 shows a description of the functional coverage items used to identify that the interface-based features identified in Sample 3-1 have been exercised.

Define your goal.

Functional coverage can help you measure your progress only if your goals are explicitly defined. It will also make analysis of the functional coverage easier. The progress will be measured against a constant goal. If the goals are intellectually defined every time you analyze a functional coverage report, then these goals are subject to human error. There will also be a tendency to minimize the impor-

Sample 3-4. Functional coverage for interfaced-based features of a UART design

1. Level of the Clear-to-Send (CTS) pin.
2. Level of the Data-Ready (DTR) pin.
3. CPU cycle kind crossed with address.
4. CPU cycle kind transition.
5. CPU cycle kind crossed with address crossed with data.

tance of holes toward the end of the project as you subconsciously justify your progress against the looming deadline. Sample 3-5 describes the functional coverage goals for each of the functional coverage points identified in Sample 3-4. Different features may use the same coverage point but imply a broader goal.

Sample 3-5. Functional coverage goals for interfaced-based features of a UART design

1. A least one value of 0 and 1 observed.
2. At least one value of 0 and 1 observed.
3. At least one read and write cycle with address greater 4.
4. All combinations of read and write cycles.
5. At least one read and write cycle for each address equal to configuration register and with individual bits equal to 0 and 1.

Understand the complexity of the goal.

Don't make your goal more accurate or precise than it needs to be. The more coverage point bins that must be filled to meet your goal, the more work it is going to be. With cross-coverage, the number of bins that must be filled grows exponentially. For example, it is not realistic to attempt to cover all possible values for a 32-bit address for both read and write cycles: That is over 8 billion values. Define bins for equivalent values or combinations of values to minimize the number of samples required to meet your goal. Functional coverage tools have a practical limit on the total number of bins that must be filled to provide a measure of the coverage.

Question, reduce, inform.

It is very easy to collect a large number of functional coverage metrics. But the more functional coverage data you have, the harder it becomes to analyze the results. Always question the relevance of a functional coverage point. If you start to ignore some coverage point reports or are not looking forward to the next report, you should probably not collect it. There is a fundamental difference between data and information. Haphazard functional coverage points only provide data that must be analyzed. Well-chosen func-

tional coverage points with well-defined goals provide information that is immediately meaningful. For example, if a FIFO must be exercised across its operating range, measuring the values of the read and write pointers would be data. Measuring the *difference* between the read and write pointers with goals stated as empty, full and neither would be much more meaningful. Crossed with some critical pointer regions (such as roll-over points), this latter functional coverage point will provide much more relevant information.

Functional coverage definition is an evolving art.

Developing a good functional coverage model of your verification plan is not easy. This section has described the necessary steps only in the broadest of terms. Functional coverage modeling is a topic that can and should be developed into a science with well-defined processes. The book *Functional Verification Coverage Measurement and Analysis* by Andrew Piziali is an excellent text on functional coverage modeling. Although it uses the *e* language as implementation medium, the coverage modeling process outlined in the book can just as easily be implemented using SystemVerilog.

From Features to Testbench

Identify how correctness will be determined.

Note that the functional coverage points described above do not make any reference to the correctness of the results. Correctness is the responsibility of the self-checking portion of the testbench. Given the features that must be verified, you have to determine how its correctness is going to be confirmed. The process is similar to identifying the expected response in a directed testcase exercising that feature. The difference is that you do not know the timing or ordering of the stimulus that will trigger the feature. Errors can be detected by a failure of the random testbench to operate properly, by explicitly comparing output data against expected data in the self-checking structure, or white-box assertions on the design itself. The list of error detection mechanisms becomes a detailed specification of the self-checking random testbench. Sample 3-6 shows the error detection mechanism that will confirm the correctness of the features identified in Sample 3-1.

Identify termination mechanisms.

It is easy to terminate a directed testcase: Once you are done applying the stimulus necessary to exercise the target feature, you simply terminate the simulation. But a random testbench is not about exercising a single feature. How do you know when to stop? You have to plan for several termination mechanisms that can be triggered or

Sample 3-6. Error detection for interfaced-based features of a UART design

1. Data source will wait for Clear-to-Send (CTS) pin to be asserted before writing the next data to send. If it is not functional, no data will be transmitted.
2. Data sink will wait for the Data-Terminal-Ready (DTR) pin to be asserted before reading the next received data. If it is not asserted, no data will be received.
3. Covered by #4.
4. Verify that read cycles return expected values given the previous values written, writability of bits and reset value, size and presence of registers.
5. Covered by #4.

turned off through additional constraints on the random testbench or a simple procedural call at the beginning of a simulation run. Any one termination mechanism, once triggered will cause the orderly shutdown of the simulation.

There are several popular termination mechanisms. A watchdog timer is useful to prevent deadlocked simulations: It must be reset at regular intervals, otherwise the simulation is terminated. A time bomb helps prevent run-away simulations: It will terminate the simulation after a predetermined amount of time. An idle detector will stop the simulation when all of the output interfaces have been idle for some time. A simple data item counter will terminate the simulation after a specified number of data items was sent to or received from the design. Functional coverage feedback can terminate the simulation if the metrics are not significantly increasing or if the coverage goal has been reached.

You can run for a long time, or you can run many times.

You can generate a lot of random data using two strategies: You can run a random source for a long time, or you can run a random source many times, for a short time, each time with a different seed. If the random source is truly random and the seeds are chosen as not to repeat a previous sequence, then the quality of the resulting random data should be the same. However, the effects on a simulation of each strategy are quite different.

Plan for many short runs.

A long simulation will be cumbersome to reproduce if the error is detected toward the end. Furthermore, since a single simulation run will typically use a single configuration of the device, you will have less opportunities to verify different configurations. Your device may also find itself in a particular corner of the state space and

remain in that corner. Continuing to apply more stimulus under those conditions is unlikely to yield increased functional coverage.

Using the many-short-runs strategy, you can reproduce a problem more quickly, run many more configurations and quickly traverse the state space. If your device requires a lot of simulation cycles to reach certain states after reset, consider state-forcing mechanisms as described in "Design for Verification" on page 93.

From Features to Generators

Identify stimulus requirements.

To be able to fill your coverage goals, it will be necessary for the random generators to generate the necessary stimulus, with the necessary characteristics and necessary timing. It is very simple to generate a single packet or instruction with random content. But this simple random generation approach is likely insufficient if you need the ability to generate packets of different lengths, lots of consecutive packets of the same length, straight-through instruction sequences, nested loop structures, invalid or corrupted data or synchronized data across multiple random streams.

A random generator that will be able to exercise the required features does not happen by accident. It has to be designed and architected to produce the required data sequences. Sample 3-7 shows the random generator requirements necessary to exercise the features identified in Sample 3-1.

Sample 3-7. Generator requirements for interfaced-based features of a UART design

1. Generate send data stream.
2. Generate receive data stream.
3. Generate read or write cycles.
4. Covered by #3.
5. Covered by #3.

Constraints become preferable to more seeds.

What if, after multiple random simulations, some of your functional coverage points remain unfilled? You could run more simulations with additional seeds, or you could add constraints to your testbench to increase the probability (hopefully to 100 percent) of filling at least one of the remaining functional coverage points. The latter, although requiring more work on your part, is likely to be the more productive avenue, especially if hundreds of previous runs

failed to produce the necessary inputs to fill those coverage points. The gap in your functional coverage measurements could also be a symptom of a functional problem in your random generators or verification environment. It is possible that a lingering constraint is preventing the generation of input sequences that will cause the functional coverage points to be filled.

Identify constrainable dimensions.

When architecting your random generator, it is necessary to consider the available constraint mechanisms. Traditionally, different random streams were produced by physically altering the code of the random generator. As shown in Figure 3-7, altering the code of the random generator effectively created a different random generator for each testbench.

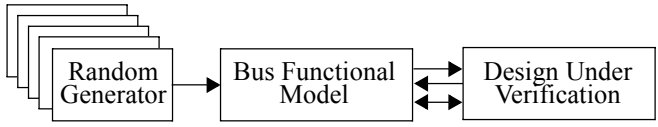


Figure 3-7. Different random generators

To minimize the amount of duplicated code and the amount of new code that must be written to fill additional functional coverage points (and thus to be more productive), it is better to design a random generator that can be constrained easily, from the outside, as illustrated in Figure 3-8. Writing a random generator that can be constrained easily from the outside does not happen by accident. The section titled “Random Stimulus” on page 307 shows how to write such generators. Sample 3-8 shows the constraint mechanisms that must be available in the generators to exercise the features in Sample 3-1.

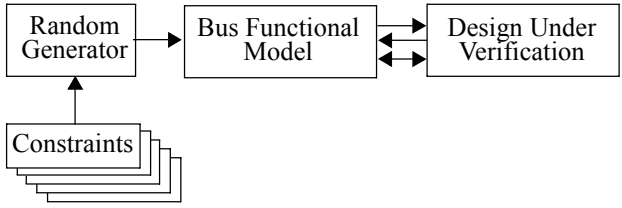


Figure 3-8. Constraining a single random generator

Sample 3-8. Constraint requirements for interfaced-based features of a UART design

1. No constraints.
2. No constraints.
3. Must be able to constrain address.
4. Must be able to constrain type of cycle in sequences of cycles.
5. Must be able to constrain address and data.

Randomly generate the device or testbench configutation.

It is easy to conceive of randomly generating data streams throughout a simulation. But you can just as easily randomly generate data that is used only once, at the beginning of the simulation. For example, configurable or programmable devices are often verified using only a few configurations hard-coded in the testbench or an external file. And most simulations are usually run using one of those configurations.

Why not randomly generate the device configuration then download it into the device? The device configuration descriptor is then used by the self-checking structure to predict the response according to the current configuration. Similarly, you could randomly generate the configuration of the testbench. For example, when verifying an ethernet switch, why not randomly generate the number of devices on each port, their speed and their station MAC addresses? Then use functional coverage measurement on the randomly generated configuration to know which configurations and combinations of configuration parameters were verified.

Constrain configurations if necessary.

Your self-checking structure does not yet support all possible device configurations? Or, you are unable to "compile" all possible configurations into register writes? Or, you are migrating from a Verilog testbench that can use only two configurations through *$readmemh* tasks? No problem. Simply constrain the configuration generator to generate only the supported configurations. Once you are able to support additional configurations, remove the constraints accordingly.

Directed Testcases

Identify low-probability testcases.

There are some features that will have a low probability of being exercised through random stimulus. For example, verifying that interrupt bits are maskable would require that the mask bits be randomly set to one and zero while the associated interrupt bits were

set and cleared (i.e., fill the cross-coverage of the interrupt bit value with the mask bit value, for all interrupt bits). Given that interrupts usually signal exceptional events in a design, it will likely take a very long time for random stimulus to completely verify this feature. These features are probably better verified using directed testcases.

May be implemented using constraints.

Writing a directed testcase does not necessarily imply using a directed procedural implementation. If the random generators were designed to be highly constrainable, it is possible to constrain them so much that they will produce directed stimulus. For example, to verify the maskability of a particular interrupt, you would constrain the CPU cycle generator to write a zero then a one in the appropriate position at the interrupt mask register address. You would then constrain the data generators to cause the interrupt condition. Once the condition is detected, constrain the CPU cycle generator to write a zero and then a one in the mask bit again. An assertion would verify that the external interrupt would be asserted only when the interrupt condition is not masked. However, if the most productive approach is to write a directed procedural testcase, the random environment can be suspended to allow access to the transaction layer of the bus-functional model.

The first testcases are the simplest but also the toughest.

When a new version of the design first hits the verification team, it is subjected to a few simple testcases. The objective of these testcases is to verify that the basic functionality of the design operates correctly. Once it passes these initial trivial tests, it will be subjected to high volumes of traffic to thoroughly verify the design.

These first trivial testcases, although very simple, are the toughest ones to pass. You may spend weeks running the same simple tests. Because they are used on immature code, they catch the most bugs. These early trivial testcases usually involve performing a write cycle followed by a read cycle, or transmitting a single packet, or executing a few straight-through instructions.

Trivial testcases can be random.

Because of their simplicity, you could be tempted to write the first trivial testcases as directed testcases. Given well-designed generators, they are usually much simpler to write as constrained tests. For example, constraining the test to two cycles, where the first one must be a write cycle, the second must be a read cycle and both addresses must be the same. Or, constrain the packet generator to generating only one packet for the entire simulation. Or, constrain

the instruction generator to generate only arithmetic opcodes without any branches. Once the initial trivial tests pass successfully, you remove these constraints and let the now-debugged random environment loose on the design.

See Chapter 6 of the VMM.

For guidelines on how to implement functional coverage models using SystemVerilog, see the section titled "Functional Coverage Implementation" starting on page 266 of the *Verification Methodology Manual for SystemVerilog*.

SUMMARY

Write a verification plan. It is the specification for all testcases and supporting testbench functions. Implement and verify from a common specification. Do not verify an implementation.

Define the various levels of granularity used to verify the design: block, unit, reusable core, FPGA, ASIC, subsystem, system, board. Trade off greater visibility and controllability for fewer testbenches and more integration tests.

Define the self-checking strategy that will be used to detect errors.

Identify features from the design specification, and enumerate which features must be verified.

Consider verification early in the design phase. Architect the design as needed to make it as easy to verify as possible.

You can use a directed testbench approach if the number of testcases is small. For each feature, specify a testcase. Implement each testcase in a separate testbench.

Define a functional coverage model from the enumerated features. From those same features, identify the degrees of freedom and constraint dimensions of the generators required to generate the stimulus that will exercise each feature. Use your coverage model to decide which feature to target next and how to best exercise it.

CHAPTER 4 HIGH-LEVEL MODELING

A skilled verification engineer must break the "RTL mindset" that most hardware engineers, out of necessity, have grown into. To efficiently accomplish the verification task, you must be well versed in behavioral (i.e., non-synthesizable and highly algorithmic) and transaction-level descriptions. To reliably and correctly use the high-level constructs of SystemVerilog, it is necessary to understand the side effects of the simulation algorithm and the limitations of the language—and to understand ways to circumvent those side effects and limitations. This understanding was not required to write RTL models successfully.

HIGH-LEVEL VERSUS RTL THINKING

This section illustrates the differences between the approaches to writing an RTL model and to writing a high-level model.

Many guidelines help code RTL models.

All experienced hardware design engineers are very comfortable with writing synthesizable models. The models conform to a well-defined subset of the SystemVerilog language and follow one of a few coding styles. Numerous RTL coding guidelines have been published.[1] They help designers obtain efficient implementations:

1. See "*IEEE P1364.1 Standard for Verilog Register Transfer Level Synthesis*" prepared by the Verilog Synthesis Interoperability Working Group of the Design Automation Standards Committee.

low area, high speed or low power. Guidelines, such as the ones shown in Sample 4-1, can help a novice designer avoid undesirable hardware components, such as latches, internal buses or tristate buffers. More importantly, guidelines such as the ones shown in Sample 4-2, can help maintain identical behavior between the synthesizable model and the gate-level implementation.

Sample 4-1. RTL coding guidelines to avoid undesirable hardware structures

1. To avoid latches, set all outputs of combinatorial blocks to default values at the beginning of the block.
2. To avoid internal buses, do not assign *reg*s from two separate *always* blocks.
3. To avoid tristate buffers, do not assign the value *1'bz*.

Sample 4-2. RTL coding guidelines to maintain simulation behavior

1. All inputs must be listed in the sensitivity list of a combinatorial block.
2. The clock and asynchronous reset must be in the sensitivity list of a sequential block.
3. Use a nonblocking assignment when assigning to a *reg* intended to be inferred as a flip-flop.

The adherence to the synthesizable subset and proper coding guidelines can be verified easily using linting (more details are in the section titled "Linting" on page 24). After several months of experience, the subset becomes very natural to hardware designers. It matches their mental model of a hardware design: state machines, operators, multiplexers, decoders, latches and clocks etc.

Do not use RTL-like code when writing testbenches.

The synthesizable subset is adequate for describing the implementation of a particular design. The subset is dictated by the synthesis technology, not by someone with a warped sense of humor playing a practical joke on the entire industry. It is designed to describe hardware structures and logical transformations between registers, matching the capability of logic synthesis technology. However, this subset quickly becomes insufficient when writing testbenches that were never intended to be implemented in hardware. SystemVerilog has a rich set of constructs and statements. If you have an RTL mindset when writing testbenches and limit yourself to using a coding style designed to describe relatively low-level hardware structures, you will not take full advantage of SystemVerilog's

power. The verification task will be needlessly tedious and complicated.

Contrasting the Approaches

The example below shows a simple handshaking protocol. Your task is to write the SystemVerilog code that implements the simple handshaking protocol shown in Figure 4-1. The protocol detects that an acknowledge signal (ACK) is asserted (high) after a requesting signal (REQ) is asserted (high). Once the acknowledge signal is detected, the requesting signal is deasserted, and the protocol then waits for the acknowledge signal to be deasserted.

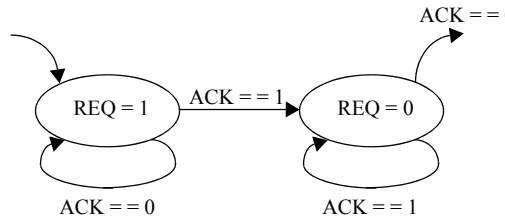


Figure 4-1. State diagram for handshaking protocol

RTL-Thinking Example. A hardware designer, with an RTL mindset, will immediately implement the state machine shown in Figure 4-1. The corresponding SystemVerilog code is shown in Sample 4-3. This relatively simple behavior required 21 lines of code and two *always* blocks to describe, and two additional states in a potentially more complex state machine.

Focus on behavior, not implementation.

High-Level-Thinking Example. A verification engineer, with a high-level mindset, will instead focus on the *behavior* of the protocol, not its implementation as a state machine. The corresponding code is shown in Sample 4-4. The functionality can be described behaviorally using only four statements.

High-level models are faster to write.

Modeling this simple protocol using high-level constructs should require less than 10% of the time required to model it using synthesizable constructs. Not only is there less code to write (20%), but it is also simpler, requiring less effort to ensure that it is correct.

High-level models simulate faster.

Another benefit of high-level modeling is the increase in simulation performance. Assuming that there is a long delay between a change in the request and the corresponding acknowledgement, the simula-

Sample 4-3. Synthesizable SystemVerilog code for simple handshaking protocol

```
enum {...,
      MAKE_REQ,
      RELEASE,
      ...} state, next_state;
...
always_comb
begin
   next_state <= state;
   case (state)
   ...
   MAKE_REQ: begin
      req <= 1'b1;
      if (ack) next_state <= RELEASE;
   end
   RELEASE: begin
      req <= 1'b0;
      if (!ack) next_state <= ...;
   end
   ...
   endcase
end

always_ff @(posedge clk)
begin
   if rst state <= ...;
   else state <= next_state;
end
```

Sample 4-4. High-level SystemVerilog code for simple handshaking protocol

```
always
begin
   ...
   req <= '1';
   wait (ack);
   req <= '0';
   wait (!ack);
   ...
end
```

tion of the synthesizable model would still execute the *always_ff* block at every rising edge of the clock (because that *always* block is sensitive to that transition of the clock signal). The *always* block containing the high-level description would wait for the proper condition of the acknowledge signal, resuming execution only when the protocol is satisfied. If the acknowledge signal replies after a 10 clock-cycle delay, this represents a reduction of process execution from 20 in the synthesizable version to 2 in the behavioral one, or a 1000 percent increase in simulation performance.

YOU GOTTA HAVE STYLE!

The synthesizable subset puts several constraints on the coding style you may use. Even with these restrictions, many less experienced hardware designers manage to write RTL code that is difficult to understand and maintain. There are no such restrictions with high-level modeling. With this complete and thorough freedom, it is not surprising that even experienced designers produce testbench code that is unmaintainable, fragile and not portable.

A Question of Discipline

Write maintainable, robust code.

There are no laws against writing bad code. If you do, the consequences do not involve personal fines or prison terms. However, the consequences do involve a real economic cost to your employer. Your code *will* need to be modified: either to fix a functional error, to extend its functionality or to adapt it to a new design. When (not *if*) your code needs to be modified, it will take the person in charge of making that modification more time than would otherwise have been required had the code been written properly the first time. Under extreme conditions, your code may even have to be re-written entirely.[1]

My first job after graduating from university was to design and implement a portion of a logic synthesis tool using the C language. In those days, I had been writing code in various languages for over eight years, and I measured my performance as a software engineer by the cleverness of my implementations of algorithms. I felt really proud of myself when I was able to craft a complex computation into a "poetic" one-liner.

Invest time now, save support time later.

I soon came to realize the error of my ways. During the eight previous years, I always wrote "disposable" code: The programs were either short-lived (school assignments or personal projects), or they had a narrow audience (utilities for university professors or a learning aid for a particular class). Never had I written a program that would live for several years and be used by dozens of persons, each with their own sophisticated needs and attempting to use my program in ways I had never intended or even thought of. As I found

1. Do not think, "*It won't be my problem.*" You may very well be that person and you may not be able to understand your own code weeks later.

myself having to fix many problems reported by users, I had difficulties understanding my own code written only weeks before. I quickly learned that time invested in writing better code up front would be saved many times over in subsequent support efforts.

Optimize the Right Thing

You should *always* strive for maintainability. Maintainability is important even when writing synthesizable code. Before optimizing some aspect of your code, make sure it really needs improvement. If your code meets all of its constraints, it does not need to be optimized. Maintainability is the most important aspect of any code you write because understanding and supporting code is the most expensive activity.

Saving lines actually costs money.

There is no economic reason to reduce the number of lines of code. Unless, of course, it also improves the maintainability. Saving one line of code, with an average of 50 characters per line, saves only 50 bytes on the storage medium. With 40GB hard drives costing less than $80 in 2002[1], this represents a savings of one hundred thousandth of one cent ($0.00000001). The time saved in typing, assuming an extremely slow typing speed of one character per second and a loaded labor rate for an engineer at $100,000 a year[2], amounts to $0.69. However, if saving that line reduces the understandability of the code where it will require an additional five minutes to figure out its operation, the additional cost incurred amounts to $4.17. The total *loss* from reducing code by one line equals $3.48. And that is for a single line and a single instance of maintenance.

Optimizing performance costs money.

Similar costs are incurred when optimizing code for performance. These optimizations usually reduce maintainability and must be done only when absolutely required. If the code meets its constraints as is, do not optimize it. That principle applies to synthesizable code as well. The example in Sample 4-5 is a design example provided in the Vera distribution. It is a synthesizable description of a 2-bit round-robin arbiter.

1. 93% cheaper in the 3 years since the first edition of this book!
2. That, however, is pretty much the same...

Sample 4-5. Synthesizable code for 2-bit round-robin arbiter

```
/*
###############################################
# PROPRIETARY AND CONFIDENIAL                 #
# SYSTEMS SCIENCE INC.                        #
# COPYRIGHT (c) 1995 BY SYSTEMS SCIENCE INC. #
###############################################
*/
module rrarb(request, grant, reset, clk);
input  [1:0] request;
output [1:0] grant;
input        reset;
input        clk;
wire         winner;
reg          last_winner
reg    [1:0] grant;
wire   [1:0] next_grant;

assign next_grant[0] =
   ~reset & (request[0] &
               (~request[1] | last_winner));

assign next_grant[1] =
   ~reset & (request[1] &
               (~request[0] | ~last_winner));

assign winner =
   ~reset & ~next_grant[0] &
   (last_winner | next_grant[1]);

always @ (posedge clk)
begin
   last_winner = winner;
   grant = next_grant;
end
endmodule
```

RTL code can be too close to schematic capture.

Several aspects of maintainable code were used in Sample 4-5: Identifiers are meaningful and the code is properly indented. However, the continuous assignment statements implementing the combinatorial decoding suggest that the author was thinking in terms of boolean equations, maybe even working from a schematic design, not in terms of functionality of the design.

This approach simplifies the understanding of the final implementation at the cost of functional understanding. From each concurrent statement, it is easy to figure out the logic gates and flip-flops necessary to implement. But try to figure out what happens to the con-

tent of the *last_winner* register when there are no requests, or add a third request and grant signal pair. Understanding or modifying the functionality is much more difficult. Other potential problems are the race conditions created by using the blocking assignments in the *always* block (for more details, see “Read/Write Race Conditions” on page 177).

Specify function first, optimize implementation second—and only if needed.

The code shown in Sample 4-6 implements the same function, but it is described with respect to its functionality, not its gate-level implementation. The code sample simplifies the understanding of the function but makes no attempt at describing the final implementation. It is much easier to figure out what happens to the content of the *last_winner* register when there are no requests or to add a new request and grant signal pair. The synthesized results should be close to that of the previous model. The synthesized results should not be a concern until it is demonstrated that the results do not meet area, timing or power constraints. Your primary concern should be maintainability, unless shown otherwise.

Sample 4-6. Synthesizable code for 2-bit round-robin arbiter

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

reg [1:0] grant;
reg last_winner;
always_ff @ (posedge clk)
begin
   grant <= 2'b00;
   if (reset) last_winner <= 0;
   else if (request != 2'b00) begin: find_winner
      reg winner;
      case (request)
         2'b01: winner = 0;
         2'b10: winner = 1;
         2'b11: winner = last_winner+1;
      endcase
      grant[winner] <= 1'b1;
      last_winner   <= winner;
   end: find_winner
end
endmodule
```

Good Comments Improve Maintainability

If reducing the number of lines of code actually increases the overall cost of a design, the same argument applies to comments. One could argue that reducing the number of lines of code can yield a better program, since there are fewer statements to understand and debug. However, the primary purpose of comments is explicitly to improve maintainability of code. No one can argue that reducing their number can lead to better code.

You can write bad comments.

However, just as there is bad code, there are bad comments. Obsolete or outdated comments are worse than no comments at all since they create confusion. Comments that are cryptic or assume some particular knowledge may not be very useful either. One of the most common mistakes in commenting code, illustrated in Sample 4-7, is to describe in written language what the code actually does.

Sample 4-7. Poor comment in SystemVerilog

```
// Increment addr
addr++;
```

Unless you are trying to learn SystemVerilog, this comment is self-evident and redundant. It does not add any information. Any reader familiar with SystemVerilog would have understood the functionality of the statement. Comments should describe the intent and purpose of the code, as illustrated in Sample 4-8. It is information that is not readily available to someone unfamiliar with the design.

Sample 4-8. Proper comments in SystemVerilog

```
// In burst mode, the bytes are written in
// consecutive addresses. Need to access the
// next address to verify that the next byte
// was properly saved.
addr++;
```

Assume an inexperienced audience.

When commenting code, you should assume that your audience is composed of junior engineers who are familiar with the language, but not with the design. Ideally, it should be possible to strip a file of all of its source code and still understand its functionality based on the comments alone.

STRUCTURE OF HIGH-LEVEL CODE

This section describes techniques to structure and encapsulate high-level code for maximum maintainability. Encapsulation can be used to hide implementation details and package reusable code elements.

RTL models are structured according to implementation needs.

Structuring code is the process of allocating portions of the functionality to different modules or entities. These modules or entities are then connected together to provide the complete functionality of the design. There are many guidelines covering the structure of synthesizable code. That structure has a direct impact on the ease of meeting timing requirements. The structure of a synthesizable model is dictated by the limitations of the synthesis tools, often with little regard to the functionality.

Testbenches are structured according to functional needs.

A testbench implemented using high-level SystemVerilog code does not face similar restrictions. You are free to structure your code any way you like. For maintainability reasons, high-level code is structured according to functionality or need. If a function is particularly complex, it is easier to break it up into smaller, easier to understand subfunctions. Or, if a function is required more than once, it is easier to code and verify it separately. Then you can use it as many times as necessary with little additional efforts. SystemVerilog code can be structured using *task*, *function*, *class*, *module*, *program, interface, package* or *inheritance.*

Encapsulation Hides Implementation Details

Encapsulation is an application of the structuring principle. The idea behind encapsulation is to hide implementation details and decouple the usage of a function from its implementation. That way, the implementation can be modified or optimized without affecting the users, as long as the interface is not modified.

Keep declarations as local as possible.

The simplest encapsulation technique is to keep declarations as local as possible. This technique avoids accidental interactions with another portion of the code where the declaration is also visible. A common problem in SystemVerilog is illustrated in Sample 4-9: Two *always* blocks contain a *for-loop* statement using the register *i* as an iterator. However, the declaration of *i* is global to both blocks.

They will interfere with each other's execution and produce unexpected results.

Sample 4-9. Improper encapsulation of local objects in Verilog

```
int i;

always
begin
   for (i = 0; i < 32; i = i + 1) begin
      ...
   end
end

always
begin
   for (i = 15; i >= 0; i = i - 1) begin
      ...
   end
end
```

In SystemVerilog, put local declarations in *begin/end* blocks.

In SystemVerilog, you can declare registers local to a *begin*/*end* block. A proper way of encapsulating the declarations of the iterators so they do not affect the module-level environment is to declare them locally in each *always* block, as shown in Sample 4-10. Properly encapsulated, these local variables cannot be accidentally accessed by other *always* or *initial* blocks and create unexpected behavior.

Sample 4-10. Proper encapsulation of local objects in SystemVerilog

```
always
begin
   int i;
   for (i = 0; i < 32; i = i + 1) begin
      ...
   end
end

always
begin
   int i;
   for (i = 15; i >= 0; i = i - 1) begin
      ...
   end
end
```

SystemVerilog *tasks* and *functions* can contain local variables.

Other locations where you can declare local registers in SystemVerilog include *tasks* and *functions*, after the declaration of their arguments. An example can be found in Sample 4-11.

Sample 4-11. Local declarations in tasks and functions

```
task send(input [7:0] data);
   reg          parity;

   ...
endtask

function [31:0] average(input [31:0] val1,
                        input [31:0] val2);
   reg   [32:0] sum;

   sum = val1 + val2;
   average = sum / 2;
endfunction
```

Minimize the scope of local variables.

In SystemVerilog, local variable declarations can be located at the beginning of any *begin/end* block. And since blocks can be located anywhere in sequential code to create local scope regions, local variables can be created while minimizing their scope and potential undesirable interaction. For example, Sample 4-12 shows how a local iterator variable can be created in the middle of a long sequence of statements by creating a local scope region.

Sample 4-12. Local declarations in SystemVerilog

```
function bit eth_frame::compare(eth_frame to);
   compare = 0;
   ...
   if (this.data_len !== to.data_len) return;
   begin
      int i;
      for (i = 0; i < this.data_len; i++) begin
         if (this.data[i] !== to.data[i])
            return;
      end
   end
   ...
   compare = 1;
endfunction: compare
```

Inline the declaration of iterator variables.

The scope of variables used solely as *for-loop* iterators can be further reduced by declaring them within the loop statement, often eliminating the need for a *begin/end* block. Sample 4-13 shows how the local iterator variable can be declared and created within the

for-loop statement. Note that the loop iterator variable is an *automatic* variable. It is allocated every time the loop is entered and freed once the loop exits.

Sample 4-13. Local declarations in SystemVerilog

```
function bit eth_frame::compare(eth_frame to);
   compare = 0;
   ...
   if (this.data_len !== to.data_len) return;
   for (int i = 0; i < this.data_len; i++) begin
      if (this.data[i] !== to.data[i])
         return;
   end
   ...
   compare = 1;
endfunction: compare
```

Encapsulating Useful Subprograms

Some functions and procedures are useful across an entire project or between many testbenches. One possibility would be to replicate them wherever they are needed. This obviously increases the required maintenance effort. It also duplicates information that was already captured. SystemVerilog has *classes*, *modules*, *programs*, *interfaces* and *packages* to encapsulate any declaration used in more than one partition.

Example: error reporting routines.

One example of procedures that are used by many testbenches are the error reporting routines. To have a consistent error reporting format (which can be parsed easily later to check the result of a regression), a set of standard routines are used to issue messages during simulation. When using SystemVerilog, they should be implemented as *void functions*, within a message *class*. The *Verification Methodology Manual for SystemVerilog* defines a very flexible message reporting class named *vmm_log* and detailed guidelines for using it. See the section titled "Message Service" starting on page 134 of the *Verification Methodology Manual for SystemVerilog* for more details.

Functions should be packaged in a *class* and used via a local instance.

It is preferable to use *classes* to encapsulate shared declarations. They can be used anywhere. They offer a protection mechanism for data and procedures that users should not be allowed to use directly. And they can be user-extended in case a user wants to add to or modify the packaged declarations. Functions and tasks in a class—called *methods*—are accessed through a local instance of the class.

The SystemVerilog implementation, shown in Sample 4-14 and used in Sample 4-15, can be compiled on its own since the functions are contained within a compilation unit. The encapsulating class will be added to the *$root* name space—and thus should be carefully named to avoid collisions. *Classes* unlike *modules*, must be explicitly instantiated using the *new* constructor.

Sample 4-14. Packaging of functions and tasks in SystemVerilog

```
class syslog;

int warnings = 0;
int errors = 0;

function void warning(input string msg);
   $write("WARNING at %t: %s", msg);
   warnings++;
endfunction: warning
...
endclass: syslog
```

Sample 4-15. Using tasks packaged using a *class* in SystemVerilog

```
module testcase;

syslog log = new;

initial
begin
   ...
   if (...) log.error("Unexpected response");
   ...
   log.terminate;
end
endmodule
```

static variables can replace global variables.

It is also possible to include global variables, such as an error counter in the *class*. In the usage example shown in Sample 4-15, there will be an instance of error and warning counters for each instance of the *class*. In this case, it would be preferable to have a single instance of those counter values. This can be accomplished by declaring them as *static*, as shown in Sample 4-16.

This is basic object-oriented programming.

Encapsulating procedures and the state variables they operate on in the same construct is the primary technique in object-oriented programming. SystemVerilog's *class* is an object-oriented construct. It supports inheritance and polymorphism. These important object-oriented concepts will be introduced in "Object-Oriented Programming" on page 147.

Sample 4-16. Global variables in SystemVerilog

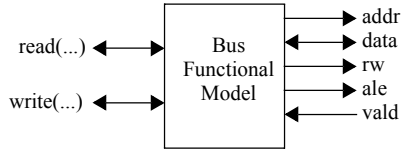
```
class syslog;

static int warnings = 0;
static int errors = 0;

function void warning(input string msg);
   $write("WARNING at %t: %s", msg);
   warnings++;
endfunction: warning
...
endclass: syslog
```

Encapsulating Bus-Functional Models

In Chapter 5, I describe how stimulus applied to the design under verification via complex waveforms and protocols can be implemented using tasks. These tasks, called *bus-functional models*, are typically used by many testbenches throughout a project. If they model a standard interface, such as a PCI bus or a Utopia interface, they can even be reused between different projects. Properly packaging these tasks facilitates their use and distribution.

Figure 4-2 shows a block diagram of a bus-functional model. On one side, it drives and samples low-level signals according to a predefined protocol. On the other side, tasks are available to initiate a transaction with the specified data values. The latter is called a *procedural interface*.

Figure 4-2. Block diagram of a bus-functional model



Task arguments are passed *by value* only.

In SystemVerilog, you might be tempted to implement the bus-functional model using a task where the low-level signals are passed to the tasks, so it can be reused to drive different sets of signals. By default, SystemVerilog arguments are passed *by value* when the task is called and when it returns. At no other time can a value flow into or out of a task via its interface. For example, the task shown in Sample 4-17 would never work. The assignment to the *bus_rq* variable cannot affect the outside until the task returns. The task cannot return until the *wait* statement sees that the *bus_gt*

signal was asserted. But the value of *bus_gt* cannot change from the value it had when the task was called.

Sample 4-17. By default, task arguments in SystemVerilog are passed *by value*

```
class arbiter;
...
// This task will not work...
task request(output logic bus_rq,
             input  logic bus_gt);
   // The new value does not "flow" out
   bus_rq <= 1'b1;
   // And changes do not "flow" in
   wait bus_gt == 1'b1;
endtask: request
...
endclass: arbiter
```

Task arguments can be passed *by reference*.

SystemVerilog arguments can be passed *by reference* if the *ref* attribute is added to the argument declaration. Passing by reference is like passing a pointer as the argument. When passed by reference, any change on the outside is immediately reflected inside the task. And any change made inside the task is immediately reflected outside. For example, the task shown in Sample 4-18 would work. The assignment to the *bus_rq* variable will affect the outside. The task will return when the *wait* statement sees that the *bus_gt* signal was asserted.

Sample 4-18. By default, task arguments in SystemVerilog are passed *by value*

```
class arbiter;

task request(ref output logic bus_rq,
             ref input  logic bus_gt);
   // The new value will "flow" out
   bus_rq <= 1'b1;
   // And changes will "flow" in
   wait bus_gt == 1'b1;
endtask: request

endclass: arbiter
```

Encapsulate signals and BFM tasks in *interface*.

However, only *variables* can be passed by reference through task arguments. Physical signals are *nets* and thus cannot be used as task arguments. Physical signals can be encapsulated in an *interface*, as shown in Sample 4-19. The bus-functional model tasks can also be located in the *interface*, with the physical signals directly accessible. This also simplifies calling the tasks as the (potentially numer-

ous) signals need not be enumerated on the argument list for every call.

Sample 4-19. Encapsulating physical signals and tasks in *interface*.

```
interface arbiter;
logic bus_rq;
logic bus_gt;

task request;
   bus_rq <= 1'b1;
   wait bus_gt == 1'b1;
endtask: request
endinterface: arbiter
```

Pass signals to *classes* using a *virtual interface*.

Bus-functional model tasks can be encapsulated in an *interface*. However, *interfaces* do not support the object-oriented programming model in SystemVerilog. Only *classes* do. It is not possible to protect local declarations in an interface. Nor is it possible to extend it to add new bus-functional methods or modify the existing ones to inject errors. Bus-functional model tasks can still be encapsulated in a *class* while physical signals are bundled in an *interface*. Sample 4-20 shows a class encapsulating bus-functional model tasks for an ethernet MII interface and an interface encapsulating its physical signals. Notice how the interface is passed as an argument to the constructor and saved in a local variable using the *virtual* attribute. Each instance of the bus-functional model *class* operates on a single set of physical signals through the *virtual interface*, maintaining internal state variables belonging to that one interface.

Specify the *virtual interface* binding when instantiating the bus-functional class.

To create an instance of a bus-functional model encapsulated in a class, call its constructor. Each instance will be connected to the *interface* specified in that instance's constructor. Thus, different instances of the same class can be connected to different physical signals. Sample 4-21 shows an example instantiating two instances of the MII bus-functional model, each connected to a different *interface*.

See Chapter 4 of the VMM.

See the section titled "Transactors" starting on page 161 of the *Verification Methodology Manual for SystemVerilog* provides detailed guidelines and further techniques on encapsulating bus-functional models in classes and interface signals in interfaces.

Sample 4-20. Class with virtual interface

```
interface eth_mii_if;
   logic tx_clk, txd, tx_en;
   logic rx_clk, rxd, rx_dv;
   logic col, crs;
endinterface: eth_mii_if

class eth_mii_mac;
   local virtual eth_mii_if sigs;

   task new(virtual eth_mii_if sigs);
      this.sigs = sigs;
   endtask: new

   task send(input eth_frame frame);
      @ (posedge this.sigs.tx_clk);
      ...
   endtask: sends

   task receive(output eth_frame frame);
      @ (posedge this.sigs.rx_clk);
      ...
   endtask: receive
endclass: eth_mii_mac
```

Sample 4-21. Instantiating bus-functional model class

```
module testbench;

eth_mii_if if0();
eth_mii_if if1();

eth_switch dut(if0, if1, ...);

eth_mii_mac bfm[2];

initial
begin
   bfm[0] = new(if0);
   bfm[1] = new(if1);
   ...
end
endmodule: testbench
```

DATA ABSTRACTION

Synthesizable models are limited to bits and vectors.

The limitation of logic synthesis technology has forced the synthesizable subset into dealing only with data formats that are clearly implementable: bits, vectors of bits and integers. SystemVerilog adds *enums* and *structs* to the synthesizable set, but the various ele-

ments of the *struct* must themselves be bits, bit vectors or integers. High-level models have no such restrictions. You are free to use any data representation that fits your need.

Work at the same level as the design under verification.

You must be careful not to let an RTL mindset artificially limit your choice, or to keep you from moving to a higher level of abstraction. You should approach the verification problem at the same level of granularity as the "unit of work" for the design. For an ethernet switch, it is an ethernet MAC frame. For an IP cell router, the unit of work is an entire IP packet. For a SONET framer, the unit of work is a SONET frame. For a video compressor, the unit of work is either a video line or an entire frame, depending on the granularity of the compression. The interesting conditions and testcases are much easier to set up at that level than at the low-level bit interface.

SystemVerilog provides excellent support for abstracting data into high-level representations. This section will only outline how certain data types can be used. You are invited to consult a book on the language to learn the details of all available data types.

2-state Data Types

Some types are 4-state.

The SystemVerilog types *logic*, *reg*, *wire*, *integer* and *time* are 4-state types. Each bit in these types can represent 0, 1, X or Z. This means that each bit of data requires at least two bits of implementation. Low-level hardware modeling requires 4-state logic to more accurately represent the possible logic values on a physical signal. But in high-level modeling, the values X or Z are usually not required nor useful.

Prefer 2-state types.

If a 4-state representation is not absolutely required, use 2-state types in preference to 4-state types. They will require less memory to implement than their 4-state counterpart and will simulate faster. The 2-state types are *bit*, *int*, *real* and *shortreal*.

Struct, Class

Structs and *classes* are used to represent information composed of various smaller pieces of different types. They can be used to model packets, frames, instructions, commands, floating-point numbers, etc. Sample 4-22 shows a *struct* used to model an IEEE single-precision floating-point number. Sample 4-23 shows the declaration for a *class* used to represent an ATM cell. An ATM cell is a fixed-

length 53-byte packet with 48 bytes of user data. Which one should be used?

Sample 4-22. *Struct* for a floating-point value.

```
typedef struct {
   bit        sign;
   bit [24:0] mantissa;
   bit [ 5:0] exponent;
} ieee_sp_float
```

Sample 4-23. *Class* for an ATM cell

```
class atm_cell;
   bit [11:0] vpi;
   bit [15:0] vci;
   bit [ 2:0] pt;
   bit        clp;
   bit [ 7:0] hec;
   bit [ 7:0] payload[48];
endclass: atm_cell
```

Structs are bundles of bits, like *integer*.

A *struct* is an integral type, just like *integer* or *reg*. Whenever you declare a variable of a *struct* type, the necessary number of bits is automatically allocated. If you assign a *struct* variable to another, or pass a *struct* as an argument to a function or task, as shown in Sample 4-24, all of the bits are copied. A *struct* simply creates additional structure to the collection of bits it represents. It is possible to specify a literal value for a *struct*. Furthermore, if a *struct* is declared *packed*[1], as shown in Sample 4-25, the bits of the *struct* fields are laid out consecutively in memory, as shown in Figure 4-3. This allows the *struct* to be transparently converted to and from bit vectors or integer values.

Sample 4-24. *Struct* variables.

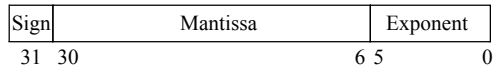
```
function ieee_sp_float abs(ieee_sp_float v);
   v.sign = 0;
   abs = v;
endfunction: abs

ieee_sp_float v1, v2;
v1 = {1, 24'h800, 6'h0};
v2 = v1;
v1 = abs(v1);
```

1. There are restrictions on the types that can be used within a *packed struct*. See the SystemVerilog LRM for more details.

Sample 4-25. *Packed struct* for a floating-point value.

```
typedef struct packed {
   bit        sign;
   bit [24:0] mantissa;
   bit [ 5:0] exponent;
} ieee_sp_float
```

Figure 4-3. Layout of a *packed struct*.



Classes are dynamic and must be explicitly instantiated.

A *class* is a dynamic type. Whenever you declare a variable of a *class* type, all you get is a reference[1] initialized to *null*. The necessary number of bits to hold the content of the *class* must be explicitly allocated. If you assign a *class* variable to another, or pass a *class* as an argument to a function or task, only the reference is copied, not the actual *class* instance. It is not possible to specify a literal value for a *class*. A *class* can be converted to and from bit vectors by using explicit packing and unpacking functions.

Sample 4-26. *Class* variables.

```
class atm_cell;
   ...
   function atm_cell copy();
      copy = new;
      copy.vpi = this.vpi;
      ...
   endfunction: copy
endclass: atm_cell
...
initial
begin
   atm_cell c1, c2, c3;
   c1 = new;
   c1.vpi = 12'hABC;
   ...
   c2 = c1;         // c1 & c2 refer to same
                    // instance
   c3 = c2.copy(); // c2 & c3 differ
                    // but have same content
end
```

Structs are not object-oriented.

The biggest difference between *class* and *struct* is that the *class* is the only type in SystemVerilog that supports the object-oriented

1. A reference is similar to a pointer.

programming model. It can contain tasks and functions to operate on its content. It can be extended to add or modify an existing class. It can protect its content against unauthorized access or usage. Methods can be declared *virtual* to support polymorphism and make their operation instance-specific. More on *classes* and object-oriented programming will be presented in "Object-Oriented Programming" on page 147.

Use *classes*.

Because of their greater flexibility compared to *struct*, *classes* are preferred when writing testbenches and transaction-level models. *structs*, because of their closer association with bits, are more suitable for describing a synthesizable model of the design. Because testbenches typically have to create thousands of stimulus datum and record their corresponding expected response, the dynamic nature of *class* instances will help minimize memory consumption. The ability to move references to large chunks of data, instead of the data itself, will lead to a more runtime efficient testbench as well.

Union

Unions are variable *structs*.

Unions provide different or optional fields on top of the same bits. *Packed unions* are useful for having the ability to look at the same bits through different types or layout. For example, the first few bytes of an ethernet frame may carry link-layer information. It may be useful to have the ability to look at these bytes as straight payload bytes or structured LLC information, as shown in Sample 4-27.

Sample 4-27. LLC information in payload using *union*.

```
class eth_frame;
   ...
   bit [15:0] typ_len;
   union packed {
      bit [7:0] data[1500];
      struct packed {
         bit [7:0] dsap;
         bit [7:0] ssap;
         bit [7:0] control;
         bit [7:0] data[1497];
      } llc;
   } payload;
   bit [31:0] fcs;
endclass: eth_frame;
```

Pure *unions* are unsafe.

Unions allow bits to be written using one type and read using another type, a potential type loophole. For example, an ethernet frame may or may not contain four additional header bytes carrying virtual LAN label information. This could be modeled using a pure *union* as shown in Sample 4-28. The VLAN label data could be used whether or not it is present: nothing prevents its access if the frame does not contain it, as shown in Sample 4-29. Furthermore, the model of the frame requires an indication to let users know whether to use the *vlan.label* or *vlan.no_label* field.

Sample 4-28. Optional fields using *union*.

```
class eth_frame;
   ...
   bit is_labelled;
   union {
      void no_label;
      struct {
         bit [ 2:0] user_pri;
         bit        cfi;
         bit [11:0] id;
      } label;
   } vlan;
   bit [15:0] typ_len;
   ...
endclass: eth_frame;
```

Sample 4-29. Type loophole in *union*.

```
eth_frame fr = new;
fr.is_labelled = 0;
...
if (fr.vlan.label.id == 12'hABC) ...
```

Tagged unions provide value safety.

A *tagged union* provides type safety by allowing access to a union field only if it is appropriately tagged. The virtual LAN label information in an ethernet frame can be modelled using a *tagged union* as shown in Sample 4-30. There is no longer a need for an explicit indication of which flavor of the union to use as it is built in the *tagged union* itself. As shown in Sample 4-31, the VLAN label fields cannot be accessed in an unlabeled frame.

Avoid *unions*.

Although *unions* appear simple, elegant and attractive, they create more problems than they solve when the time comes to randomize them. If different fields of different types share the same bits, what are the semantics of constraining the different fields? Should there be a single solution for the shared bits that satisfy all the constraints, from the different type perspectives? Or should only the

Sample 4-30. Optional fields using *tagged union*.

```
class eth_frame;
   ...
   union tagged {
      void no_label;
      struct {
         bit [ 2:0] user_pri;
         bit        cfi;
         bit [11:0] id;
      } label;
   } vlan;
   bit [15:0] typ_len;
   ...
   function new(bit is_labelled);
      if (is_labelled) begin
         this.vlan = tagged label {0, 0, 0};
      else this.vlan = tagged no_label;
   endfunction: new
   ...
endclass: eth_frame;
```

Sample 4-31. Invalid access in *tagged union*.

```
eth_frame fr = new(0);
...
if (fr.vlan.label.id == 12'hABC) ... // Error!
if (fr matches tagged tag) begin
   if (fr.vlan.label.id == 12'hABC) ... // OK
end
```

constraints from one perspective be satisfied? In that case, which one? *Tagged unions* could solve the latter dilemma by using the tag to determine which perspective to use. Unfortunately, the SystemVerilog language does allow the value of the tag itself to be constrained. It would thus not be possible to randomly select between different tag alternatives, possibly constrained by other field values.

Unions should be implemented in-line.

Sample 4-32 shows a class modeling an ethernet MAC frame with optional virtual LAN labelling. When the *is_labelled* class property is non-zero (i.e., true), the *cfi*, *user_pri* and *vlan_id* class properties are assumed to be valid and "exist". When the *is_labelled* class property is zero (i.e., false), these subsequent properties are not relevant and do not "exist". When referring to the content of the class, like in a pure *union*, it will be necessary to check the *is_labelled* class property to determine whether the other class properties "exist" and process the frame accordingly. Because the optional class properties are always present and are separate from other class properties, they can be constrained and solved for like any other

class property. However, the optional class properties remain accessible at all times, without a mechanism to prevent their use, so the same programmer discipline required to use a pure *union* is required.

Sample 4-32. In-lined optional class properties.

```
class eth_frame;
   bit [47:0] da;
   bit [47:0] sa;
   bit        is_labelled;  // VLAN control
   bit [ 2:0] user_pri;   // VLAN
   bit        cfi;        // VLAN
   bit [11:0] vlan_id;    // VLAN
   bit [15:0] len_typ;
   ...
endclass: eth_frame
```

Unions can be implemented using composition.

Inlining all possible optional class properties requires that memory be allocated for all of them, even if they are not going to be used. If the number and size of optional class properties is small, that is not a problem. If a data model needs to present several different variations, each independent of each other—for example PCI Express control frames—it may be more appropriate to use *composition* to model optional class properties. Sample 4-33 shows a class modeling an ethernet MAC frame with optional virtual LAN labelling using composition. When the *vlan* class property is non-*null*, the *cfi*, *user_pri* and *vlan_id* class sub-properties are valid and exist. When the *vlan* class property is *null*, these class sub-properties do not exist. When referring to the content of the class, like in a pure *union*, it will be necessary to check the *vlan* class property to determine whether the other class sub-properties are present and process the frame accordingly. Composition, like *tagged unions*, has a built-in mechanism to prevent references to absent class properties. If an absent class property is accessed, the *null* sub-class reference will cause a run-time error. However, composition has some additional requirements related to randomization, constraining and solving. Because randomization does not allocate memory, it will be necessary to allocate all of the subclasses composing the randomized class in the *pre_randomize()* method, then prune the class from unnecessary composed subclasses based on the result of the randomization in the *post_randomize()* method.

Sample 4-33. Optional class properties implemented using composition.

```
class eth_vlan_label;
   bit [ 2:0] user_pri;
   bit        cfi;
   bit [11:0] vlan_id;
endclass: eth_vlan_label

class eth_frame;
   bit [47:0]     da;
   bit [47:0]     sa;
   eth_vlan_label vlan;
   bit [15:0]     len_typ;
   ...
endclass: eth_frame
```

Inheritance should not be used to replace *unions*.

You may be tempted to replace *unions* using inheritance[1] as shown in Sample 4-34. First, a base class represents the data model without any optional class property. Then, derived classes are used to add various optional class properties. Although very object-oriented, this approach creates several limitations.

- First, you will not be able to randomly generate a mix of datum, sometimes with optional class properties, sometimes not. Each derivative creates a new data type. Once an instance of a type is created, it cannot be modified. Each time it will be randomized, it will produce random values for that type and no other.
- Second, you will not be able to create combinations of optional class properties easily. Because multiple inheritance is not supported, you cannot recombine multiple derived classes into a single one containing multiple optional class properties. With single inheritance, you would have to create a new type for each possible combination of optional class properties. This quickly grows to a significant number, which grows exponentially when testbench-specific additions must be made to the data model.

See Chapter 4 of the VMM.

See guidelines 4-68 to 4-72 of the *Verification Methodology Manual for SystemVerilog* specifies guidelines for modeling data structures with optional data fields.

1. Inheritance will be discussed in more detail in “Inheritance” on page 153.

Sample 4-34. Optional class properties implemented using single inheritance.

```
class eth_frame;
   bit [47:0] da;
   bit [47:0] sa;
   bit [15:0] len_typ;
   ...
endclass: eth_frame

class eth_vlan_frame extends eth_frame;
   bit [ 2:0] user_pri;
   bit        cfi;
   bit [11:0] vlan_id;
   ...
endclass: eth_vlan_frame

class eth_control_frame extends eth_frame;
   bit [15:0] opcode;
   ...
endclass: eth_control_frame

class eth_control_vlan_frame extends eth_frame;
   bit [ 2:0] user_pri;
   bit        cfi;
   bit [11:0] vlan_id;
   bit [15:0] opcode;
   ...
endclass:eth_control_vlan_frame
```

Arrays

There are *packed* and *unpacked* arrays.

SystemVerilog defines two types of array: *packed* and *unpacked*. *Packed* arrays can only be made of single-bit types and their dimensions are specified to the left of the variable name, as shown in Sample 4-35. *Packed* arrays are implemented as consecutive bits and can be implicitly used as signed or unsigned integer values. *Unpacked* arrays can be of any type. They are declared with their dimensions specified to the right of the variable name, as shown in Sample 4-36. Unlike *packed* arrays, the content of an *unpack* array cannot be treated like an integral value. Each element of an *unpacked* array is an individual value.

Sample 4-35. Packed array declarations.

```
bit              [ 7:0] a_byte;
logic     [3:0]  [ 7:0] q_quadword;
eth_frame        [31:0] scoreboard;   // INVALID
```

Sample 4-36. Unpacked array declarations.

```
bit       eight_bits[8];
logic     sixteen_bits[4][4];
eth_frame scoreboard[32];
```

Only *unpacked* arrays will be considered.

From a high-level modeling perspective, *packed* arrays are no different than any other scalar types. Therefore, only *unpacked* arrays will be considered in this section.

Single-dimensional arrays are useful data structures for representing linear information such as ordered data sequences, look-up tables or memories. Two-dimensional arrays are used for planar data such as graphics or video frames. Three-dimensional arrays are not frequently used, but they could find an application in representing data for video compression applications such as MPEG. Arrays with greater numbers of dimensions have rare applications, especially in hardware verification.

Dynamic arrays have unspecified size.

A *dynamic* array is an *unpacked* array with an unspecified dimension size. The actual size of the array is specified and allocated at runtime. The size can be modified—shrunk or grown—at any time during the simulation.

Sample 4-37. Dynamic array declaration and sizing.

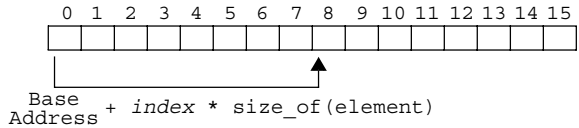
```
eth_frame sb[];
...
sb = new[32];                 // 32 elements
sb = new[sb.size()*2] (sb); // Double its size
sb.delete();                    // Flush it
```

Arrays are typically located in consecutive memory elements.

As shown in Figure 4-4, array elements are typically located in consecutive memory locations. They are accessed by computing their address using an offset from a base address. Random access, array truncation and element replacement are efficient operations. But element insertion, element deleting and array lengthening are expensive operations that require copying a potentially large number of elements to maintain the integrity of the consecutive memory locations. For these types of operations, a *queue* may be more

appropriate. See “Queues” on page 141 for more information on *queues*.

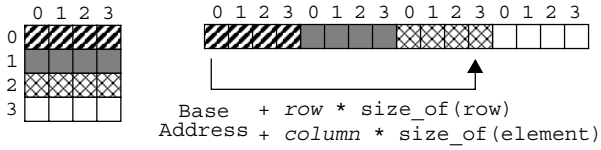
Figure 4-4. Array elements in memory



Multi-dimensional arrays must be mapped onto a single dimensional structure.

Even though SystemVerilog offers multi-dimensional arrays, they must be mapped to the linear hardware memory of the host computer. That hardware memory is only one-dimensional. Figure 4-5 shows a two-dimensional array mapped into a linear memory. Indexing an element requires knowing the length of each preceding row. To make these (often highly repeated) calculations efficient in large multi-dimensional arrays, it is necessary to have fixed-sized dimensions. That is why *dynamic* arrays are limited to single dimension arrays.

Figure 4-5. Mapping a 4x4 array to a linear memory



Use arrays of classes of arrays for multi-dimensional dynamic arrays.

Multi-dimensional dynamic arrays can be emulated by using a dynamic array of *classes* containing a dynamic array. Since this multi-dimensional array is composed of independent one-dimensional arrays, the size and number of each dimension can vary. However, looking up one element will require looking up each individual dimension. Sample 4-38 shows a definition, instantiation and reference of a two-dimensional dynamic array of RGB values.

Queues

Queues are implemented using links.

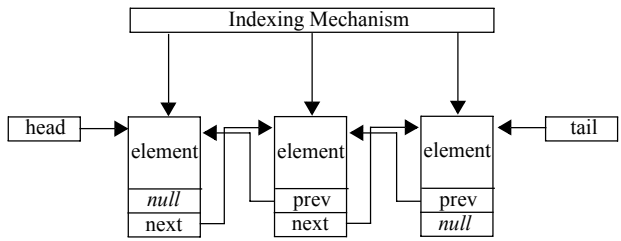
Queues are used to represent ordered linear information and, as such, are very similar to one-dimensional arrays. However, they guarantee constant-time insertion and removal of individual elements either at the end or the beginning of the queue. To support the constant time insertion and deletion, they fundamentally differ from arrays in their implementation. As shown in Figure 4-6, queue elements are located in independent memory locations. The linear and

Sample 4-38. Two-dimensional dynamic array.

```
class rgb;
   bit [7:0] red;
   bit [7:0] green;
   bit [7:0] blue;
endclass: rgb

class line;
   rgb pixels[];
   function new(int unsigned n);
      this.pixels = new [n];
   endfunction: new
endclass: line

class picture;
   line lines[];
   function new(int unsigned x,y);
      this.lines = new [x];
      foreach (this.lines[i]) begin
         this.lines[i] = new(y);
      end
   endfunction: new
endclass: picture

...
picture vga = new(480, 640);
rgb center = vga.lines[240].pixels[320];
```

ordered relationship is created by a series of pointers, starting from a *head* pointer, that points to each subsequent element in the queue. Queues are frequently doubly-linked in the reverse direction, starting from a *tail* pointer, to facilitate some queue operations[1]. To support constant time access to its elements via an ordinal reference, an indexing mechanism is super-imposed on the linked elements.

Figure 4-6. Doubly-linked queue elements in memory



1. The actual implementation of queues in a specific simulator may differ.

Queues can be more efficient than arrays.

Because of their different implementation, *queues* are easier to use than arrays if their grow and shrink many times during a simulation. Arrays must allocate consecutive memory for their entire size, whereas the memory used by *queues* grows and shrinks by one element at a time as the number of elements they contain increases or decreases. Elements can be appended or prepended at little cost. All that is required is a re-orientation of the pointer sequence to include or remove the element.

Queues have a rich set of predefined methods.

Queues come with a rich set of operators, such as appending or prepending to a *queue*, removing the element at the head or tail, inserting or deleting an element at an arbitrary offset within the *queue,* finding out its length or iterating over all of its elements.

Associative Arrays

They are arbitrarily indexed.

Associative arrays are used for non-ordinal or non-consecutive indexing operations. The elements in an associative array are not considered ordered from index *0* through index *N-1*. Instead, they are randomly stored based on an efficient indexing mechanism that can use any type—even a *string* or *class* reference—as indexing value. The element is stored in the array associated with that arbitrary index value. Pre-defined methods exist to check the existence of an element at a particular index, delete an existing element, or iterate through all of the elements in the array.

They can be used to model large memories.

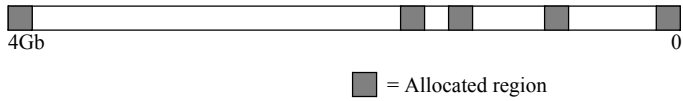
One of the best applications of an associative array is to model a large memory. In system-level simulations, you may have to provide a model for a large amount of memory. With the amount of memory available in today's systems, and the overhead associated with modeling them, you may find that you do not have a computer with enough resources to simulate your system-level model efficiently. For example, if you model a memory with 32-bits of addressable bytes using an array of *logic*, the amount of memory consumed by this array alone exceeds 8GB (four logic values requiring 2 bits to represent each *logic* bit times 8 bits per byte times 4GB).

Only the sections of the memory currently in use need to be modeled.

In any simulation, it is unlikely that all memory locations are required. Usually, the accesses are limited to a few regions within the memory address space. An associative array can be used to model a very large memory in a fashion similar to a cache memory. Only regions of the memory that are currently in use are stored in

the array. When a particular location is accessed, the array is checked for the region of interest, allocating a new region as necessary. Figure 4-7 shows a conceptual view of an address space where only the portions that are actively used are physically allocated. This type of partial memory model is called *sparse memory model*.

Figure 4-7. Sparse memory model



An associative array can be used to model a sparse memory.

Sample 4-39 shows a *class* encapsulating a sparse memory model. It has a method to locate and return any address of interest. Conversely, a method is provided to store a specified value at a specified memory location. This model could also include usage assertions, such as reading uninitialized memory locations or overwriting unread memory values.

Sample 4-39. Sparse memory model using an associative array.

```
class sparse_memory;
   local logic [7:0] mem[bit [63:0]];

   function logic [7:0] read(bit [63:0] addr);
      read = 8'hXX;
      if (this.mem.exists(addr)) begin
         read = this.mem[addr];
      end
   endfunction: read

   function void write(bit   [63:0] addr,
                       logic [ 7:0] data);
      this.mem[addr] = data;
   endfunction: read
endclass: spare_memory
```

They can improve scoreboarding performance.

Looking up a large scoreboard for a specific expected response, given an observed response can be an expensive operation. For example, it may be necessary to look up packets based on the content of their destination address. Looking up an ethernet MAC frame based on a 48-bit destination address would require an array with over 281 billion elements. Assuming only minimum-size MAC frames, this array would consume 18 million gigabytes! Using an associative array, it is much more efficient to allocate only those locations for which we already have an expected MAC frame.

Files

External input files complicate configuration management.

I recommend avoiding external input files for testbenches. Configuration management of the testbench and the design under verification is complex enough. Without good practices, it is very difficult to make sure that you are simulating the right version of the correct model together with the proper implementation of the right testbench. If you must add to the mix making sure you have the right version of input files, often generated by scripts from some other format of some other files, configuration management grows exponentially in complexity. For example, many use files to initialize SystemVerilog memories, as shown in Sample 4-40.

Sample 4-40. Initializing a memory using an external file

```
module testcase;

reg [7:0] pattern [0:55];

initial $readmemh(pattern, "pattern.memh");

endmodule
```

Understanding the implementation of the testcase now requires looking at two files and understanding their interaction. If the file always contains the same data for the same testcase, it can be replaced with an explicit initialization of the memory in the SystemVerilog code, as shown in Sample 4-41. Now, only a single file needs to be managed and understood. In some cases, using external files is unavoidable, such as when using input data that was produced by an external program or recorded from actual data streams.

Sample 4-41. Explicitly initializing a memory

```
module testcase;

reg [7:0] pattern [0:55] = {8'h00,
                            8'hFF,
                            ...
                            8'hC0};

endmodule
```

Files can program bus-functional models.

Programmable testbenches are architected around programmable bus-functional models and checkers, and can be programmed via an external input file. The "program" can be as simple as a sequence of data patterns or as complex as a pseudo assembly language with opcodes and operands interpreted by an engine implemented in

SystemVerilog. However, this approach to programmable bus-functional models makes it impossible to drive these bus-functional models from higher level bus-functional models, such as random data generators. It will also make it extremely difficult to coordinate or synchronize a particular operation of the bus-functional model with some external events or other bus-functional model. It requires the introduction of synthetic synchronization instructions.

Use SystemVerilog as the BFM programming language.

If a new synchronization mechanism is required, a new instruction must be added. It is easier to "program" a bus-functional model using the testbench language (which is a rich high-level language) with calls to the procedural interface of the bus-functional model. The program, being part of the testbench code, has visibility over the necessary states of the design or other bus-functional model to coordinate and synchronize them effectively. The procedural interface can also be accessed from higher-level bus-functional models.

Do not use input files to define constraints

An input file can also be a set of limit values for constraints in a randomly-driven bus-functional model. But this limits testcases to modifying constraint boundary conditions. Testcases cannot add entirely new constraints or relax existing ones.

External files can eliminate recompilation.

Using external input files can save a lot of compilation time if you use a compiled simulator. If you can modify your testcase by modifying external input files, it is not necessary to recompile the model of the design under verification nor the testbench. For large designs, this compilation time can be significant, especially for a gate-level design with SDF back-annotation. However, your SystemVerilog simulator, such as VCS, may provide an incremental compile and link usage model to minimize the recompilation time between simulations if only the testbench changes.

From High-Level to Physical-Level

It is very unlikely that high-level data types are directly usable by any device that must be verified. Any complex data structure has to be mapped to bits, bytes, addresses and registers. They are sent to or received from the design using a physical-level interface using a more basic data representation, such as a bit, byte or word, usually including synchronization, framing or handshaking signals. In Chapter 5, I show techniques using bus-functional models for applying high-level data to a design via a low-level physical interface (and vice-versa on the output side).

Packed data types have implicit mapping.

All *packed* data types in SystemVerilog have an implicit mapping to physical bits. They have a well-defined mapping to integer or packed single-dimension array of bits. These are then more useful for the design specification than testbench, where the granularity of data items tends to be coarser and higher level.

Encapsulate streaming operations.

The *streaming* operators can be used to translate to and from a series of discrete variables, such as a set of *class* properties, and a stream of physical-level values. As described by guidelines 4-77 and 4-78 of the *Verification Methodology Manual for SystemVerilog*, these operations should be encapsulated in *packing* and *unpacking* methods.

OBJECT-ORIENTED PROGRAMMING

SystemVerilog is object-oriented.

Object-oriented programming is a methodology that has been used with great success in software engineering for many years. It is the next evolutionary step in language design after structured programming (i.e., the removal of the "go to" statement and introduction of subprograms and control structures). Object-oriented used to be one of those buzzwords used to describe almost everything. In this book, object-oriented is used to identify a methodology that makes use of (and a language that supports) classes, inheritance and polymorphism. SystemVerilog's *class* data type meets this definition of object-oriented.

Classes

Objects are data and procedures together.

Classes are a collection of variables and subprograms that create object types. Objects are instances of a class. A class defines an object's state as a collection of data members. A class also defines all possible operations on the object using methods. In SystemVerilog, data members are called *class properties* while methods are *functions* and *tasks* declared within the class. Sample 4-39 shows an example of a *class* declaration.

Everything is modeled as an object.

Physical data types, such as packets, frames and cells, are modeled as objects. Their various fields are data members. Methods exist to calculate and check the value of any CRC or error protection field, translate the field values to and from a sequence of physically transmitted bytes and segment a large object into a series of smaller ones. Processor instructions are modeled as objects. Their opcode

and various operands are data members. Methods exist to produce the object code value, relocate branch destination values and display the current value as an assembly code statement. Floating point values—which are not directly supported by SystemVerilog—are modeled as objects. Data members represent their integer and fractional parts. Methods perform arithmetic and logical operations, translate to and from a fixed-point value or display the floating-point number using a user-specified format.

Bus-functional models are also objects.

Bus-functional models are objects. Their interface signals are data members. Methods implement the procedural interface. A design configuration is modeled as an object. The routing table, coefficient array or interface configuration parameters are modeled using data members. Methods are used to download the configuration specified in the data members in the design, or generate a random design configuration.

Scoreboards are objects.

A scoreboard (see “Scoreboarding” on page 300 for more details on scoreboards) is modeled as an object. The queues of expected output data sequences are data members. Methods exist to transform a new input data value according to the current design configuration, compare an output data value against expected ones and check the scoreboard for any losses at the end of the simulation.

Testbenches can be objects.

An entire testbench can be an object in which the bus-functional models and scoreboards sub-objects are data members. Methods implement the reset procedure, main testcase sequence and termination procedures.

Data members consume memory. Methods do not.

By default, each data member is local to each object instance. Methods, however, are global to the class. For example, if there are 1,000 instances of the *eth_frame* class (see Sample 4-32), there are 1,000 instances of the *da* property but only a single instance of the code implementing the *to_bytes* function. The memory required to implement the data members will be replicated for each object instance. Methods will not consume more memory, whether an object is instantiated only once or 1,000 times. If memory consumption starts to be a problem, focus on the data members of the objects with the most instances.

Data members can be global.

It is often necessary to have information that will be shared among all instances of an object. For example, if each object has its own instance of an error counter, it would be difficult to determine the

number of error messages that were produced during a simulation. If that error counter is global, that task becomes much easier. Another example is automatically assigning unique ID numbers to each object instance. A global ID number source is the only way to ensure uniqueness. In SystemVerilog, a data member global to the class is declared using the *static* attribute, as shown in Sample 4-42.

Sample 4-42. Global data members

```
class sim_status;
   static integer n_errors = 0;
endclass: sim_status

class eth_frame;
   sim_status status;
   ...
      if (this.compute_fcs() !== this.fcs) begin
         printf("Bad FCS");
         this.status.n_errors++;
      end
   ...
endclass: eth_frame

class eth_mii_mac;
   sim_status status;
   ...
      if (col === 1'b1 && crs !== 1'b1) begin
         printf("Collision without carrier\n");
         this.status.n_errors++;
      end
   ...
endclass: mii
```

module, *interface*, *struct* and *package* are not objects.

You may be tempted to conclude that all of SystemVerilog is object-oriented since *modules*, *interfaces* and *packages* can contain both variables (data members) and functions and tasks (methods). These constructs are not object oriented because *modules* are not true objects. *modules* and *interfaces* are hierarchical constructs, not data structures. They cannot be assigned. nor passed to tasks or functions as arguments. They cannot be compared or used in expressions. *packages* may also look like objects: They can contain data (shared variables) and *tasks* and *functions* (methods). In addition to presenting all of the same non-object behaviors that *modules* and *interface* do, packages cannot be instantiated. Even though a package may be used in many files, a single instance exists for the entire elaborated simulation model. And if that weren't enough,

none of these constructs support inheritance nor polymorphism, which are other key aspects of object-orientedness.

Objects have public and private declarations.

The concept behind objects is to encapsulate data and its transformation operations to present to the user a coherent and stable interface. As the object evolves or is modified, the interface visible to the user should remain constant. To help enforce this, objects usually have two separate declaration spaces: *public* and *private* declarations. Public declarations are accessible from the outside of the object (i.e., by the users), whereas private declarations are only accessible from within the object (i.e., by the implementer).

If the public interface is never modified (or modified in such a way as not to impact the user), the entire private implementation can be modified or re-written without affecting the users. By default, declarations are public. In SystemVerilog, to make a declaration private, use the *local* attribute, as shown in Sample 4-39. Any *local* data members or methods will not be externally accessible.

Keep non-randomized data members private.

Once a data member has been made public, you can count on other objects to make direct use of it. It is not possible to control its access, nor ensure that its value remains consistent with other data members. For example, the *length* property in Sample 4-43 can be modified independently of the *data* property. It is very easy for a user to corrupt the internal state of the object by directly operating on the data members. Traditional object-oriented practice commands that all accesses to an object be done through methods. These methods can ensure that the value of the properties are coherent at all times, as shown in Sample 4-44. However, if you end up with a pair of *set_data()* and *get_data()* methods for each data member, then internal coherency is probably not required, or you are providing methods with too low a level of abstraction. You might as well make the data members public.

Make *rand* data members public.

SystemVerilog places an additional requirement for making a data member public: It is not possible to externally constrain a private data member since it is not accessible. All randomized data members must be public to allow them to be constrained. More on constraints and randomization will be discussed in “Random Stimulus” on page 307.

Sample 4-43. Unsafe object state

```
class byte_list;
   integer length = 0;
   bit [7:0] data[];
endclass: byte_list

program ignoramus_use;
   byte_list my_list = new;

initial begin
   my_list.length = 100;
   my_list.data = new[1] ({1});
   // List is corrupted: array has 1 element,
   // NOT 100.
end
endprogram
```

Sample 4-44. Safe object state

```
class byte_list;
   local integer length = 0;
   local bit [7:0] data[];
   extern task resize(integer length);
endclass: byte_list

program ignoramus_user;
   byte_list my_list = new;

initial begin
   my_list.length = 100;  // Syntax error!
   my_list.resize(100);
   // List now has 100 element
end
endprogram
```

Reference and instance are different things.

When declaring an class variable, all you are declaring is a *reference* (or a pointer) to an instance of a specific class. By default, class variables do not refer to any instance (Figure 4-8(a)). They must first be initialized by allocating an instance of a class created using *new*. When assigning a class variable to another, all you are doing is making a copy of the reference, not a copy of the instance (Figure 4-8(b)). If one of these two references modifies the instance, it is modified for both class variables since they both refer to the same instance. A common mistake is to put references to instances in a *queue*, but keep using the same reference to generate new values, as shown in Sample 4-45. Since there is a single call to *new*, a single instance of the ATM cell object exists. The *queue* ends up containing several references to the last value of the cell, instead of references to 10 different random cells as expected. Similarly,

when comparing two class variables, you are comparing their reference, not their content. If both variables refer to the same instance, the comparison will be true (Figure 4-8(b)). If both objects refer to two different objects (even if they have identical content), the comparison will be false (Figure 4-8(c)).

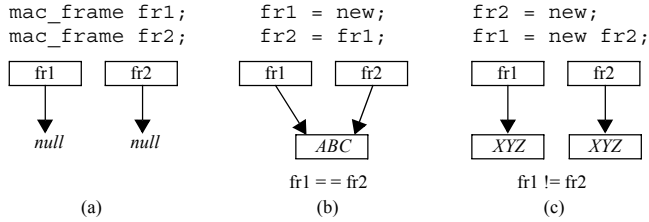


Figure 4-8. Class reference vs. class instance

```
atm_cell cell = new;
atm_cell cells[$];

while (cells.size() < 10) begin
   cell.randomize();
   cells.push(cell);
end
```

Sample 4-45. Common mistake with object reference

Comparison and copying can be shallow or deep.

Because copying or comparing class variables only deals with the references, methods must be used to perform comparison or copy functions. If the instances being compared also contain references to other objects, how are these handled? If only the references are used, then the operation is said to be *shallow* (Figure 4-9(a)). If the operation is applied recursively down the hierarchy of instances, the operation is said to be *deep* (Figure 4-9(b)). In SystemVerilog, the *new* constructor can be used to perform a shallow copy operation, but comparison and deep copy methods must be written manually for each *class*.

```
class rgb;
   byte red;
   byte green;
   byte blue;
endclass: rgb
class pixel;
   int x;
   int y;
   rgb color;
endclass: pixel
pixel dot = new;
```

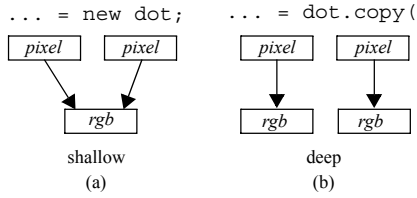


Figure 4-9. Shallow and deep operations

Standard methods must be provided.

When defining a new class, you should always provide the methods usually required for using and manipulating instances of that class. You should try to provide these methods using a consistent name and argument to avoid dealing with meaningless differences between various classes. For example, methods to display the object in a meaningful human-readable format always prove indispensable. Other methods that should be included are deep-copy, deep-compare, packing to and from a list of bytes, words or quads, calculating and checking error-protection fields, checking the internal consistency of data members and converting to and from other object types.

See Chapter 4 of the VMM.

The section titled "Data and Transactions" starting on page 140 of the *Verification Methodology Manual for SystemVerilog* specifies several guidelines for modeling data using classes and the methods that are required to provide with each class.

Inheritance

Classes can build upon other classes.

What if there is a class that does almost everything you need, but it is missing only that one little feature, what do you do? The traditional approach would dictate that you make a copy of the useful code, call it something else, and make the necessary modifications. But this has just created additional code that must be maintained and understood. With inheritance, your needs can be built upon existing classes—even those you do not have source code for—by only specifying the desired difference in behavior. Any unchanged behavior is automatically *inherited* from the original class. Any changes made to the original class are also automatically inherited by the new class, reducing maintenance efforts. The original class is called the *parent* or *base* class. The new class inheriting from the base class is called a *derived* class.

Derived classes can overload parent members and methods.

The difference in functionality between a derived class and a parent class can be expressed by adding new data members and methods, adding to the parent's methods or replacing data members and methods with new ones. For example, the verification of your design requires that you inject ethernet MAC frames with corrupted FCS values. But the ethernet MAC frame class shown in Sample 4-32 always has a good FCS value. You can create a new ethernet MAC frame object that can have a bad FCS value, based on the value of a control property,[1] using inheritance, as shown in Sample

4-46. Notice how the new version of the *compute_fcs()* method adds functionality to the parent methods. In SystemVerilog, you can refer to overloaded data members and methods in the parent class using the *super* prefix.

Sample 4-46. Adding functionality through inheritance

```
class eth_frame_may_be_bad extends eth_frame;
   bit is_bad;

   function bit [31:0] compute_fcs();
      compute_fcs = super.compute_fcs();
      if (this.is_bad) begin
         bit [4:0] i = random;
         compute_fcs[i] ^= 1;
      end
   endfunction: compute_fcs
endclass: eth_frame_may_be_bad
```

Children take after their parent.

Because derived classes are extensions of their base class, they remain valid instances of their base class. As shown in Sample 4-47, they can be assigned to base class variables without any type conversion (i.e., automatic downcasting). From that point on, they will be viewed as if they were an instance of the base class. An instance of a derived class, referred to as a base class, can be assigned back to a derived class variable with explicit upcasting. Upcasting an instance of the base class or of a derived class on a different inheritance branch (or *lineage*) causes an error. To prevent runtime errors when you cannot rely on implicit knowledge of an object's lineage, it is always possible to test the lineage of an object by using the $cast() system task in a scalar context. it will return 0 (i.e. false) if the casting operation is not allowed.

This compatibility of a derived class with its base class makes all code and models that already operate on the base class available to operate on the derived class—without their knowledge. For example, as shown in Sample 4-48, the derived MAC frame class with potential bad FCS values can be downcasted to the base class and

1. Why not make this derived class always a bad frame? Because generating a stream containing a mix of good and bad frames would require instantiating a mix of different classes. This way, only one class needs to be instantiated. The class will decide on its own whether the frame is good. And this approach is easier to constrain.

sent through the existing MII bus-functional model from Sample 4-20.

Sample 4-47. Downcasting and upcasting an inheritance tree.

```
class instruction; ...
class arith_instr extends instruction; ...
class branch_instr extends instruction; ...
class cond_branch_instr extends branch_instr; ...

instruction       instr;
arith_instr       arith;
branch_instr      br;
cond_branch_instr br_if;
...
instr = arith;     // OK: downcasting
arith = br;        // ERROR: different lineage
instr = br_if;     // OK: downcasting

$cast(br, instr);  // upcasting

$cast(arith, instr); // ERROR: instance on wrong
                     //        lineage

if ($cast(arith, instr)) begin
   ... // OK: Exec'd if correct lineage
end
```

Sample 4-48. Using code written for the base class with a derived class

```
eth_mii              bfm   = new(...);
eth_frame_may_be_bad tx_fr = new;

bfm.send(tx_fr);
```

Declarations can be semi-private.

Derived classes cannot access any private declaration in the base class. It is often necessary to let a derived class have more intimate knowledge of the internal state and implementation of a base class than what is visible through the public interface. Declarations can have the *protected* attribute. This makes them visible *only* to the base class and any derived classes. Protected data members and methods are "semi-private" declarations shared only between a base class and its derived classes. When implementing a derived class, it is assumed that you have a more intimate knowledge of a base class (how it works, what it depends on) than a casual user. That is why it is possible to gain greater visibility into a base class (if allowed by the base class author). As a user of a class, do not casually create derived classes simply to get at the protected mem-

bers and methods. Usually, these are not safeguarded as well as the public interface and are more subject to being modified.

Inheritance and instance are different things.

When building a new class upon an existing class, should you inherit from it or simply instantiate it as a data member in the new class? That depends on the relationship between the two classes. If the new class is a different way of looking at the older class, but fundamentally represents the same object, then inheritance should be used. The corruptible ethernet MAC frame in Sample 4-46 is a perfect example. The objective of this new class is not to create a completely separate and new object. It is to add a new capability to the existing one. Whether good or bad, this new ethernet frame can still be viewed and treated like the old one—albeit with a small, hidden difference.

Another example is the pixel and RGB objects shown in Figure 4-9. If you already have a *color* object and want to build a *pixel* object that has a color specification, you should instantiate the color object as a data member of the pixel object, not derive from it. Why? Because a pixel is not a color. A color is only an attribute of a pixel, just like its position on the screen.

Polymorphism

Polymorphism means multiple forms.

The term “polymorphism” means to have many forms. The concept of polymorphism was hinted at when I talked about the automatic downcasting of a derived class when using existing methods that deal with the base class, as shown in Sample 4-48. A class has the ability to take the form of an instance of any of its base classes. You can create an entire genealogy of classes. Classes on different branches can be treated as if they were the same class, when viewed as a common base class. If all classes are derived from a single root class, they can all be viewed as instances of that root base class. Polymorphism lets you write “generic” methods that can deal with any objects based on a “generic” base class.

A class can be designed to be used only as a base.

Polymorphism does not happen by accident. You have to plan your class genealogy to isolate common and useful functions in base classes. Sometimes, the common information exists, but does not make sense as a complete object. It would be a mistake to create an instance of such a base class because the base class is not designed to represent an object on its own. Rather, the base class is designed

to take advantage of polymorphism and create a single set of operations generic to that base class.

For example, ATM cells come in two flavors: UNI and NNI. Both have the same size, both have a large number of common fields. They only differ by their interpretation of the first 12 bits, as shown in Figure 4-10. To take advantage of the polymorphism between the two ATM cell flavors, you can create an *any_atm_cell* class that contains all the common fields. Derived classes will be used to add the fields unique to each flavor. But the class *any_atm_cell* should not be allowed to be instantiated on its own: it is *not* a valid ATM cell! A user must always use one of the derived classes. It is possible to have the compiler enforce this usage by declaring the base class as *virtual*. As shown in Sample 4-49, attempting to "new" an instance of a virtual class is an error.

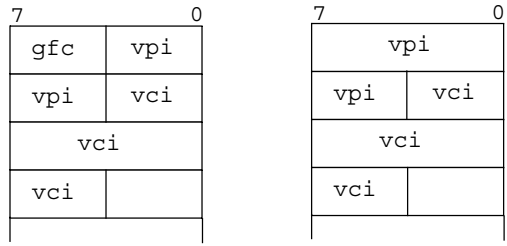


Figure 4-10. Differences in UNI and NNI ATM cells

Which method is called?

Dealing with class references and type casting is relatively simple. The value of a reference stays the same while the casting operation lets the compiler provide runtime type checking. The bigger question is, when a method is overloaded or extended in a derived class, like the *compute_fcs()* method in Sample 4-46, which version is called when an instance of a derived class is referred to as an instance of its base class?

A parent can act like a child.

By default, the original method in the base class is called. If the methods are declared as *virtual*, the *overloaded method* is called. Sample 4-50 shows how a method in the base class (which is never overloaded) makes use of a virtual method. The header error check (HEC) computation requires operating on all header bytes—which vary depending on the cell's flavor. When the *compute_hec()* method invokes the *virtual pack_header()* method, what is called is

Sample 4-49. Using *virtual* classes

```
virtual any_atm_cell;
   bit [11:0] vci;
   ...
endclass: any_atm_cell

class uni_atm_cell extends any_atm_cell;
   bit [3:0] gfc;
   bit [7:0] vpi;
endclass: uni_atm_cell

class nni_atm_cell extends any_atm_cell;
   bit [11:0] vpi;
endclass: nni_atm_cell
...
any_atm_cell a_cell;
uni_atm_cell uni_cell = new;
nni_atm_cell nni_cell = new;

a_cell = new;       // ERROR: Cannot instantiate
                    // a virtual class
a_cell = uni_cell;
a_cell = nni_cell;
$cast(nni_cell, a_cell);
$cast(uni_cell, a_cell); // Runtime error: a_cell
                         // refers to a nni_cell
                           // instance
```

the extended method found in the particular extension corresponding to the instance. The list of bits returned by *pack_header()* will always contain all of the header bits, regardless of the actual flavor of the ATM cell instance.

Methods are usually *virtual*.

In SystemVerilog, most methods should be declared virtual to give the possibility of being adapted to the extensions of each derived class and maintain the behavior expected by existing code that uses the original base class.

Randomization is a virtual process.

The predefined method *randomize()* and constraint blocks are virtual. This means that, when randomizing a base class variable that refers to a derived class instance, the derived class is being randomized, subject to the constraints and overloaded *pre_randomize()* and *post_randomize()* methods in the derived class.

No multiple inheritance.

Classes can be derived from a single base class only. It is not possible to have a class be derived from more than one base class, nor is it possible to recombine different derivatives of a common base

Sample 4-50. Using *virtual* methods

```
class any_atm_cell;
   ...
   function bit [7:0] compute_hec();
      logic [7:0] header[4];
      pack_header(header);
      ...
   endfunction: compute_hec
   virtual function void
      pack_header(ref logic [7:0] bits[4]);
endclass: any_atm_cell

class uni_atm_cell extends any_atm_cell;
   ...
   virtual function void
      pack_header(ref logic [7:0] bits[4]);
      bits = >> 8 {gfc, vpi, vci, clp, pt};
   endfunction: pack_header
endclass: uni_atm_cell

class nni_atm_cell extends any_atm_cell;
   ...
   virtual function void
      pack_header(ref logic [7:0] bits[4]);
      bits = >> 8 {vpi, vci, clp, pt};
   endfunction: pack_header
endclass: nni_atm_cell
```

class into a single class that encompasses both extensions. The lack of multiple inheritance is often mentioned by OO purists as a *sine qua non* condition that favors C++. Multiple inheritance appears to be an elegant solution to the modeling of unions in classes, instead of inlining or composition. However, multiple inheritance would not address the requirements posed by randomization and constraints (see "Union" on page 134), two concepts absent in C++ and other general-purpose object-oriented languages.

THE PARALLEL SIMULATION ENGINE

C and C++ lack essential concepts for hardware modeling.

Why hasn't C been used as a hardware description language instead of creating Verilog, VHDL, SystemVerilog and many others? Because the basic C language lacks three fundamental concepts necessary to model hardware designs: connectivity, time and concurrency. Basic C++ also lacks the necessary features to support the verification productivity cycle: randomization, constrainability and functional coverage measurement.

Connectivity, Time and Concurrency

Connectivity is the ability to describe a design using simpler blocks then connecting them together. Schematic capture tools are perfect examples of connectivity support.

Time is the ability to represent how the internal state of a design evolves over time and to control its progression and rate. This concept is different from *execution time* which is a simple measure of how long a program runs.

Concurrency is the ability to describe actions that occur at the same time, independently of each other.

C and C++ can be extended.

Many extensions and coding styles for C or C++ exist that include some or all of these concepts. SystemC is a set of C++ classes to introduce the concept of connectivity, time and concurrency. The SystemC Verification Library is a set of C++ classes that provides support for randomization, constraints and temporal expressions.

The connectivity, time and concurrency concepts are very important to understand when learning to model using a modeling language. Each language implements them in a different fashion, some easier to understand than the others.

For example, connectivity in SystemVerilog is implemented by directly instantiating modules and interfaces within modules, and connecting the pins of the modules and interfaces to wires or registers. Time is implemented by using timing control statements such as @ and *wait*. Concurrency is implemented through separate *always* and *initial* blocks. Concurrency is described in further detail in the following sections.

The Problems with Concurrency

You write better testbenches when you understand concurrency.

When writing testbenches, it becomes necessary to understand how concurrency is implemented and how concurrency affects the execution of the various components of the testbench and how they create potential race conditions.

Many testbenches are written with a severe lack of understanding of concurrency. In the best case, the execution and overall control structure of the testbench code is difficult to follow and maintain. In

the worst case, the testbench fails to execute properly on a different simulator, on different versions of the same simulator or when using different command-line options. The understanding of concurrency is often what separates the experienced designer from the newcomers.

There are two problems with concurrency. The first one is in describing concurrent systems. The second is executing them.

Concurrent systems are difficult to describe.

Since computers were created, computer scientists have tried to figure out a way to take advantage of the increased performance offered by multi-processor machines. They are relatively easy to build and many parallel architectures have been designed. However, they proved much more difficult to program. I do not know if that difficulty originated with the mindset imposed by the early Von Neumann architecture still used in today's processors, or by an innate limitation of our intellect.

Concurrent systems are described using a hybrid approach.

Human beings are adept at performing relatively complex tasks in parallel. For example, you can drive in heavy traffic while carrying a conversation with a passenger. But it seems that we are better at describing a process or following instructions in a sequential manner. For example, a recipe is always described using a sequence of steps. The description of concurrent systems has evolved into a hybrid approach. Individual processes running in parallel with each other are themselves described using sequential instructions. For example, a dessert recipe includes instructions for the cake and the icing as separate instructions that can be performed in parallel, but the instructions themselves follow a sequential order.

SystemVerilog models are concurrent threads described sequentially.

A similar principle is used in SystemVerilog. The concurrent threads are the *always* and *initial* blocks, the continuous signal assignment statements and statements in *fork/join* statements. The exact behavior of each concurrent construct is described individually using sequential statements.

Every *always* and *initial* block, every continuous assignment and every forked statement in a SystemVerilog model execute in parallel with each other, but internally each executes sequentially. It is a common misconception that SystemVerilog's *initial* blocks mean "initialize". *initial* blocks are identical to *always* blocks except that they execute only once. They are removed from the simulation once the last statement in the *initial* block executes. They are executed in

no particular order compared to other execution threads[1]. Because they are regular simulation threads, assignments made in *initial* blocks cause events on the assigned variables. However, assignment of initializer values specified in variable declarations, as shown in Sample 4-51, are not execution threads: they are performed before the start of the simulation (in lieu of the default *'x* initial value) and thus do not cause any events on the initialized variables..

Sample 4-51. Initialized variable declaration.

```
int i = 0;
...
class sim_results;
   static n_errors = 0;
endclass: sim_results
```

Emulating Parallelism on a Sequential Processor

Concurrent threads must be executed on single processor machines.

If you look inside the workstation that you use to simulate your model, you will see that there is a single processor. Even if you have a multi-processor machine, you can always write a model with one more concurrent thread than you have processors available. How do you execute a parallel description on a single processor, which is itself a sequential machine?

Multi-tasking operating systems are like simulators.

If you use a modern computer, you probably have a windowing graphical interface. During normal day-to-day use, you are very likely to have several windows open at once, each of them running a different application. On multi-user machines, there may be several others running a similar environment on the same computer. The applications running in all of these windows appear to work all in parallel even though there is a single sequential processor to execute them. How is that possible? You probably answered *time-sharing*. With time-sharing, each application uses the entire processor for small portions of time. Each application has its turn according to priority and activity. If the performance of the processor and operating system is high enough, the interruptions in the execution of a program are below our threshold of detection: It appears as if each program runs smoothly 100% of the time, in parallel with all the others.

1. Simulators would be free to execute *initial* blocks first.

Simulators are time-sharing engines.

A simulator works using the same principle. Each *always* and *initial* block or thread has the simulation engine for some portion of time. Each *appears* to be executing in parallel with the others when, in fact, they are each executed sequentially, one after another. There is one important difference in the time-sharing process of a simulator. Unlike a multi-tasking operating system, the simulator assumes that the various parallel threads cooperate to obtain fair access to the simulation resources.

Simulators do not have time slice limits.

In an operating system, every thread has a limit on the amount of processor time it can have during each execution slice. Once that limit is exhausted, the thread is kicked out of the processor to be replaced by another. There is no such limit in a simulator. Any execution thread keeps executing until it explicitly requests to be kicked out. Thus, it is possible in a simulation to have an execution thread grab the simulation engine and never let it go. Ensuring that the parallel threads properly cooperate in a simulation is a large part of understanding how concurrency is implemented.

Processes simulate until they execute a timing statement.

In SystemVerilog, an execution thread simulates, and keeps simulating, until an active *timing control* statement—@, #, or *wait*—is executed[1]. When the *timing control* statement is executed, the executing thread is kicked out of the simulation engine and replaced by another one. This thread remains "out of circulation" until the condition it is waiting for is realized. If an execution thread does not execute some form of an active timing control statement[2], it remains in the simulation engine, locking all other processes out.

The Simulation Cycle

There are *module* and *program* threads.

There are two kinds of concurrent threads in SystemVerilog: module threads and program threads. Module threads are intended to model the design whereas program threads are intended to model the testbench. Threads are made of *always* and *initial* blocks, concurrent signal assignments and forked threads. Program threads are composed of *initial* blocks, concurrent signal assignments and forked thread*s* defined inside *program* blocks. Program threads are

1. That is not strictly true but that is what happens in practice.
2. Some timing control statements can be inactive if the condition they are supposed to wait for is already true.

also composed of class methods[1] instantiated in a *program* block and invoked by a program thread. Program threads further include pass/fail action block in properties. All other threads are module threads.

A *program* thread can execute as a *module* thread.

Note that the definition of a program thread depends on the location of the code, as shown in Sample 4-52. If a program thread calls a *task* implemented in a *module* and the execution of the thread is blocked within that module *task*, the thread will resume as a module thread. It will return to a program thread once its execution blocks in a non-module *task*. Because module code cannot call program code, it is not possible for a module thread to execute as program thread. To avoid this confusing switch in execution semantics, program threads should not invoke module code—a good testbench coding practice regardless.

Sample 4-52. Module and program threads.

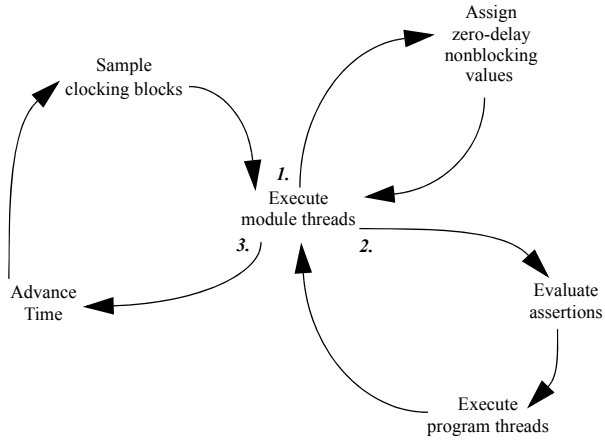
```
module tb_top;

bit clk = 0;
always #10 clk = ~clk; // Module thread

task wait_for_clk;
   @ (posedge clk);    // Module thread
endtask

endmodule

program test;

initial begin
   @ (posedge tb_top.clk);    // Program thread
   tb_top.wait_for_clk;
   @ (posedge tb_top.clk);    // Program thread
end
endprogram
```

1. The execution of class methods as module or program threads is not clearly defined in the SystemVerilog standard and is being clarified by the P1800 Working Group. Verify with your simulator which exact interpretation is being used.

Simulators execute module threads at the current time, then assign zero-delay nonblocking values.

Figure 4-11 shows the SystemVerilog simulation cycle. For a given timestep, the simulation engine first samples any value used by *clocking* blocks and *properties*. The simulation engine then executes all of the module threads that can be executed. While executing, these module threads may perform assignments of future values using nonblocking assignments. Once all module threads have executed (i.e., they are all waiting for something), the simulator assigns any nonblocking values scheduled for the current timestep (i.e. zero-delay assignments). Module threads sensitive to the new values are then executed. This cycle continues until there are no more module threads that can be executed at the current timestep and there are no more zero-delay nonblocking values.

Figure 4-11. SystemVerilog simulation cycle



Program threads are executed after *assertions* are evaluated.

When there are no more available module threads that can be executed, the simulator evaluates all assertions using the values sampled at the beginning of the timestep. The simulator then executes all program threads that can be executed. While executing, the program threads may make nonblocking assignments. Once all program threads have executed (i.e., they are all waiting for something), the simulator assigns any nonblocking values scheduled for the current timestep (i.e. zero-delay assignments) made from the program threads. Any module threads sensitive to the new values are then executed, restarting the module thread execution cycle. This cycle continues until there are no more program threads that can be executed at the current timestep and there are no more zero-delay nonblocking values.

Simulators then advance time or starve.

If there is nothing left to be done at the current time, there *must* be either:

1. A thread waiting for a specific amount of time
2. A nonblocking value to be assigned after a non-zero delay

If one of the conditions is present, the simulator advances time to the next time period where there is useful work to be done. The simulator then assigns a nonblocking value, which causes threads sensitive to the signals assigned these values to be executed, or executes threads that were waiting. If neither of the conditions are true, then the simulation stops on its own, having reached a quiescent state and suffering from event starvation.

Simulators do not increment time step by step.

The simulator does *not* increment time by a basic time unit, timestep or time increment. Regardless of the simulation resolution, the simulation advances time as far as necessary, in a single step, to the next point in time where there is useful work to do. Usually, that point in time is the delay in the clock generator. Increasing the simulation time resolution should not significantly decrease the simulation performance of a behavioral or RTL model.

Simulation progresses but time does not advance in zero-delay cycles.

The state of the simulation progresses along two axes: zero-time and simulation time. As threads are simulated and new values are assigned after zero delays, the state of the simulation evolves and progresses, but time does not advance. Since time does not advance, but the state of the simulation evolves, these zero-delay cycles where threads are evaluated and zero-delay nonblocking values are assigned are called *delta-cycles*[1]. The simulation progresses first along the delta axis then along the real-time axis, as shown in Figure 4-12. It is possible to write models that simulate entirely along the delta-time axis. It is also possible to write models that are unintentionally stuck in delta cycles, preventing time from advancing.

Program threads eliminate race conditions.

Having separate module and program threads allow testbenches to react to the results of design states and assertions without any race conditions. Program threads are evaluated once all transient events in the design have been flushed out. They are executed based on the

1. I borrowed this term from VHDL. SystemVerilog does not explicitly defines a term for zero-delay cycles.

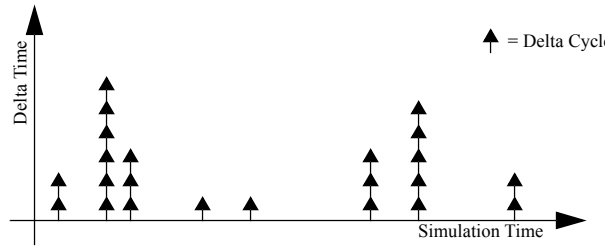


Figure 4-12. Time progression along two axis

final—for this timestep—values of module variables. It is still possible to have race conditions between module threads and between program threads, but not across them.

Synchronous design signals behave differently to program threads.

RTL coding guidelines requiring the use of nonblocking assignments to all inferred flip-flops—and enforced by the *always_ff* block—ensure that no race conditions exist between module threads. In traditional Verilog, with only one kind of execution thread, the same technique could be used to prevent races between the design and the testbench. But in SystemVerilog, with testbenches executing as program threads, a different behavior is observed. Because program threads are executed after all nonblocking assignments are made, any testbench *always* block sensitive to the clock, will execute after the state of the synchronous variable has been updated. A program thread will see the new value of the synchronous variable whereas a module thread would see its previous value, as illustrated in Sample 4-53. In that example, only the *$write* statement in the module *always* block would display a value of zero at the first clock edge after reset. The *$write* statement in the program *initial* block would always display a value of one.

Intra-assignment delay could be used.

When sampling synchronous signals at the active clock edge, the intent is to sample their *current* value, immediately before it is updated by that active clock edge. The RTL coding style in Verilog worked because of the nature of the nonblocking assignment which creates an infinitesimal delay between the clock and the updating of synchronous variables. In SystemVerilog, that infinitesimal delay is not long enough to maintain the current state of a synchronous variable into the program thread execution cycle. One solution would be to add an intra-assignment delay to all nonblocking assignments to synchronous variables, as shown in Sample 4-54. But this is not accepted by all synthesis tools, nor would it be compatible with the large body of existing RTL code out there.

Sample 4-53. Synchronous variables in module and program threads.

```
module design(...);
...
bit flag;
always_ff @(posedge clk)
begin
   if (rst) flag <= 0;
   else     flag <= 1;
end

always @(posedge clk)
begin
   if (!rst) $write("Mod: flag = %b\n", flag);
end

endmodule: design

program test;

initial begin
   forever @ (posedge design.clk)
   begin
      if (!design.rst) begin
         $write("Pgm: flag = %b\n",
                design.flag);
      end
   end
end
endprogram: test
```

Sample 4-54. Synchronous variables assigned with intra-assignment delay.

```
module design(...);
...
bit flag;
always_ff @(posedge clk)
begin
   if (rst) flag <= #1 0;
   else     flag <= #1 1;
end
endmodule: design
```

Synchronous module signals must be sampled using a *clocking* block.

Instead of relying on the proper coding of the assignment statements in the design, using a *clocking* block resolves the problem by using the *sampled* value of a synchronous variable. These values are sampled when first going into a simulation timestep, before the variables have had a chance of being updated. Therefore, the sampled value is the true value before the active edge of the clock, the one the testbench intends to use. Sample 4-55 shows the same code as Sample 4-53, but with a consistent view of synchronous vari-

ables between the design and the testbench. More details on how the clocking block is used with *interfaces* and transactors will be presented in "Physical Interfaces" on page 238.

Sample 4-55. Synchronous variables in module and program threads sampled using a clocking block.

```
program test;
...
clocking cb @ (posedge design.clk);
   input rst  = design.rst;
   input flag = design.flag;
endclocking: cb

initial begin
   forever @(cb)
   begin
      if (!cb.rst) $write("Pgm: flag = %b\n",
                          cb.flag);
   end
end
endprogram: test
```

Parallel vs. Sequential

Use sequential descriptions as much as possible.

As explained earlier, humans can understand sequential descriptions much easier than concurrent descriptions. Anything that is described using a single sequence of statements is easier to understand and maintain than the equivalent behavior described using parallel constructs. The independence of their location and ordering in the source file adds to the complexity of concurrent descriptions. A concurrent description that would be relatively easy to understand can be obfuscated by simply separating the pertinent concurrent statements with a few other unrelated concurrent constructs. Therefore, functionality should be described using sequential constructs as much as possible.

A frequent misuse of sequential constructs involves the initialization of variables. For example, Sample 4-56 shows a clock generator implemented using two concurrent constructs: an *initial* and an *always* block.

Sample 4-56. Misuse of concurrency

```
reg clk;
initial clk = 1'b0;
always #50 clk = ~clk;
```

However, generating a clock is an inherently sequential process: It starts at one value then toggles between one and zero at a constant rate. A better description, using a single concurrent construct, is shown in Sample 4-57. Better yet, to avoid generating a clock event at time zero, the *clk* variable should be initialized via an initializer value, as shown in Sample 4-58.

Sample 4-57. Proper use of concurrency

```
reg clk;
initial
begin
   clk = 1'b0;
   forever #50 clk = ~clk;
end
```

Sample 4-58. Better use of concurrency

```
reg clk = 1'b0;
always #50 clk = ~clk;
```

Deterministic sequential behavior does not need concurrency.

Another less obvious case of misused concurrency happens when the behavior of the various processes is deterministically sequential because of the data flow. For example, Sample 4-59 shows an *always* block labeled *P2* that can execute only once the *always* block labelled *P1* triggers the event *do*. The *P1* block then waits for the completion of block *P2* before resuming its execution. The sequence of execution cannot be other than the first half of *P1, P2*, then the second half of *P1*.

The implementation in Sample 4-60 shows the equivalent functionality, implemented using a single block. Not only is the execution flow easier to follow, but also it does not require the control events *do* and *done*.

Fork/Join Statement

Control flow may alternate between sequential and concurrent regions.

The overall control flow for a testcase often involves a sequence of sequential steps followed by concurrent ones. For example, verifying a configuration of a design may require configuring the device through several consecutive reads and writes via the CPU interface, then concurrently sending and receiving data. This process is then

Sample 4-59. Deterministic sequential execution.

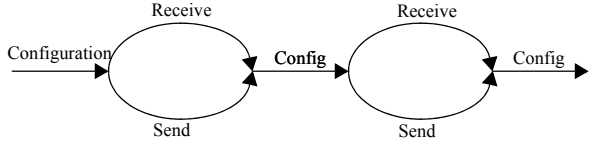
```
event do, done;

always
begin: P1
   // First half of P1
   ...
   -> do;
   @(done);
   // Second half of P1
   ...
end: P1

always
begin: P2
   @(do);
   // All of P2
   ...
   -> done;
end: P2
```

Sample 4-60. Simplified sequential execution.

```
always
begin: P1_2
   // First half of P1
   ...
   // All of P2
   ...
   // Second half of P1
   ...
end: P1_2
```

repeated for another configuration. Figure 4-13 shows a control flow diagram of such a control structure.

Figure 4-13. Series of sequential and concurrent control flows

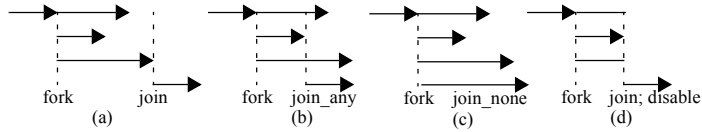


Implement using a *fork/join* statement.

The easiest way to implement this type of control flow structure is to use a *fork/join* statement. This statement dynamically creates concurrent threads within a region of sequential code. SystemVerilog has many flavors of the *join* statement to define how sequential execution resumes after the *fork*/*join* statement, as illustrated in

Figure 4-14(a). For example, the code in Sample 4-61 waits for the maximum of Ta, Tb and Tc.

Figure 4-14. Execution threads in *fork/join* statements



Sample 4-61. Example of using the *fork/join* statement

```
initial
begin
   ...
   fork
      #(Ta);
      #(Tb);
      #(Tc);
   join
   ...
end
endmodule
```

The *join* condition can have many flavors.

The *join* condition may have different flavors. By default, the code after a *fork/join* statement resumes only once all of the branches have completed their respective execution. Sometimes it may be useful to continue execution as soon as one of the branches completes its execution, as illustrated in Figure 4-14(b) or simply fork the subthreads and continue the execution of the main thread without interruption, as illustrated in Figure 4-14(c). Sample 4-62 shows how the *join_none* statement is used to generate an event at regular intervals until it is eventually acknowledged.

Sample 4-62. Variant of the *fork/join* statement.

```
...
fork: req_until_ack
   forever begin
      ->req;
      #10;
   end
join_none
@(ack);
disable req_until_ack;
...
```

join_any does not terminate other branches.

Sample 4-63 shown another way of implementing the same functionality. Notice the presence of the *disable* statement after the

join_any. The other branches of the *fork/join_any* statement keep executing after execution resumes after the *join*, as illustrated in Figure 4-14(b). Your functionality may require that they be allowed to complete in parallel. But if they must be aborted, as is the case in Sample 4-63, they must be aborted explicitly using the *disable* statement.

Sample 4-63. *fork/join_any* statement

```
...
fork: req_until_ack
   @(ack);
   forever begin
      ->req;
      #10;
   end
join_any
disable req_until_ack;
...
```

Use *disable fork* with care.

Instead of using the named *disable* statement to forcibly terminate the execution of all branches inside the named *fork/join* statement, it can be terminated by using the *disable fork* statement. However, the *disable fork* statement will terminate all subthreads started by the thread using that statement. This may terminate threads that were previously started and not targeted for termination. For example, the code in Sample 4-64 uses a base class to implement some generic functionality in concurrent threads. The class extension accidentally terminates the generic functionality by using the *disable fork*.

The Difference Between Driving and Assigning

Assignments write a value to a memory location.

Regular programming languages provide variables that can contain arbitrary values of the appropriate type. They are implemented as simple memory locations. Assigning to these variables is the simple process of storing a value into that memory location. SystemVerilog *variables* operate in the same way. When an assignment is completed, whether blocking or nonblocking, the newly assigned value overwrites any previous value in the memory location. Previous assignments have no effects on the final result. Regular assignments behave like a multiplexer. A single value from all of the potential contributors is somehow selected.

Sample 4-64. Unintended consequences of using *disable fork*

```
class xactor;
   virtual task main();
      fork
         // Generic functionality
         ...
      join_none
   endtask: main
endclass: xactor

class my_bfm extends xactor;
   virtual task main();
      super.main();
      forever begin
         fork
            // Extended functionality
           ...
         join_any
         ...
         disable fork; // Also kills generic
                       // functionality
      end
   endtask: main
endclass: my_bfm
```

The last assignment determines the value.

For example, in Sample 4-65, the value of the register *R* goes from x to 5 to 4 to 3 to 2 to 1, then finally to 0. Since *R* is a variable shared by all three concurrent blocks, a single memory location exists. Whatever value was assigned last by a concurrent block, is the value stored in the variable..

SystemVerilog has the concept of a wire.

The variable is sufficient for ordinary sequential programming languages. When describing hardware, a construct that can describe the behavior of a wire used to connect multiple devices together must be provided. Figure 4-15 shows a wire, presumably part of a data bus, connected to several devices. Each device, using a tristate driver, can drive a value onto the wire. The final logic value on the wire depends on *all* the individual values being driven, not just the last one, like a variable.

Individual values from connected devices must be driven continuously onto the wire.

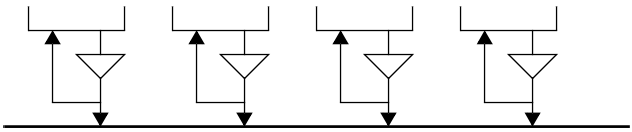
To model connectivity via a wire properly, any value driven by a device must be driven continuously onto that wire, in parallel with the other driving values. The final value on that wire depends on all of the continuously driven individual values.

For example, on a tristate wire, the individual driven values of z, 1, weak-0 and z would produce a final result of 1. Figure 4-16 shows the implementation of the wire driver in SystemVerilog.

Sample 4-65. Assignments to a shared variable.

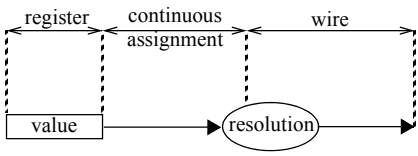
```
module assignments;
int R;

initial R <= #20 3;

initial
begin
   R = 5;
   R = #35 2;
end

initial
begin
   R <= #100 1
   #15 R = 4;
   #220;
   R = 0;
end

endmodule
```

Figure 4-15. Multiple drivers on a wire



In SystemVerilog, this continuous drive is implemented using a continuous assignment while the final value is determined by the type of wire being used (*wire*, *wor*, *wand* or *trireg*) and the strength of the individual driven values..

Figure 4-16. Implementation of continuous drive

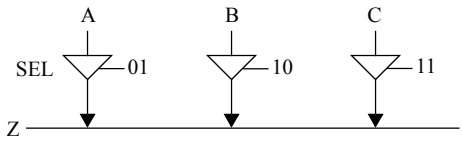


Each concurrent construct has its own, single driver.

Parallel drivers on a wire require concurrent constructs to describe them. Many inexperienced engineers, when learning to code for synthesis try to implement the design shown in Figure 4-17 using the code shown in Sample 4-66. Unfortunately, since a single register is used with variable assignments in sequential code, a *multi-*

plexer is synthesized instead of the expected parallel drivers. The proper solution requires three concurrent constructs, one for each driver, and is shown in Sample 4-67.

Figure 4-17. Simple design with three tristate drivers



Sample 4-66. Implementation using a multiplexer

```
module simple(A, B, C, SEL, Z);
input        A, B, C;
input  [1:0] SEL;
output       Z;

reg Z;
always @ (A or B or C or SEL)
begin
   case (SEL)
   2'b00: Z = 1'bz;
   2'b01: Z = A;
   2'b10: Z = B;
   2'b11: Z = C;
   endcase
end
endmodule
```

Sample 4-67. Implementation using three tristate drivers

```
module simple(A, B, C, SEL, Z);
input        A, B, C;
input  [1:0] SEL;
output       Z;

assign Z = (SEL == 2'b01) ? A : 1'bz;
assign Z = (SEL == 2'b10) ? B : 1'bz;
assign Z = (SEL == 2'b11) ? C : 1'bz;

endmodule
```

RACE CONDITIONS

The simulation cycle creates race conditions.

If you refer to Figure 4-11 and the section titled "Emulating Parallelism on a Sequential Processor" on page 162, you will see that parallel threads are executed one after another, during the same timestep. The order in which the threads are executed is *not deter-*

ministic. Race conditions exist when multiple concurrent threads compete for the same shared resource over the same time period.

RTL coding guidelines hide race conditions

Race conditions are conveniently eliminated when limiting yourself to writing synthesizable code. But once you start using all the features of the language, you may find yourself with code that is not portable across different simulators, different versions of the same simulator or by using different command-line arguments. Any change in the simulation algorithm that causes concurrent threads to be executed in a different order will yield different simulation results.

Shared variables can create race conditions.

All variables in SystemVerilog are shared among concurrent threads within their scope (except for *automatic* variables). Depending on the order in which these concurrent threads read or write these shared variables, different values may be observed.

Read/Write Race Conditions

A *read/write* race condition happens when two concurrent threads attempt to read and write the same shared variable in the same timestep. If you look at the code in Sample 4-68, you will notice that the first *always* block assigns the variable *count* while the second one displays it. But *both threads execute at the rising edge of the clock*.

Sample 4-68. Example of a read/write race condition

```
module rw_race(clk);
input clk;

integer count;

always @ (posedge clk)
begin
   count = count + 1;
end

always @ (posedge clk)
begin
   $write("Count is equal to %0d\n", count);
end

endmodule
```

The execution order determines the final result.

Let's assume that the current value of *count* is 10. If the first block is executed first, the value of *count* is updated to 11. When the second block is executed, the value 11 is displayed. However, if the second block executes first, the value of 10 is displayed, the value of *count* being incremented only when the first block executes later.

Some read/write race conditions can be solved by using nonblocking assignments.

This type of race condition can be solved easily by using a nonblocking assignment, such as shown in Sample 4-69. Referring again to Figure 4-11: When the first block executes, the nonblocking assignment *schedules* the new value of 11, with a delay of zero, to the next timestep. When the second block executes, the value of *count* is *still* 10. The new value is assigned to *count* only when all blocks executing at this timestep are executed, creating a delta cycle.

Sample 4-69. Avoiding a read/write race condition using a nonblocking assignment

```
module rw_race(clk);
input clk;

integer count;

always @ (posedge clk)
begin
   count <= count + 1;
end

always @ (posedge clk)
begin
   $write("Count is equal to %0d\n", count);
end

endmodule
```

Prefer sequential over parallel code.

Using a nonblocking or signal assignment resolves the race condition by introducing an infinitesimal delay between the *write* and the *read* operation. You should avoid creating parallel threads when a single sequential thread would do the same job, as shown in Sample 4-70.

Continuous assignments create races.

A more insidious read/write race condition can occur between *always* or *initial* blocks and continuous assignments. Examine the code in Sample 4-71 closely. What value of *out* will be displayed? The answer depends on the simulator and the command line you are using.

Sample 4-70. Avoiding a read/write race condition using sequential code

```
module rw_race(clk);
input clk;

integer count;

always @ (posedge clk)
begin
   $write("Count is equal to %0d\n", count);
   count <= count + 1;
end

endmodule
```

Sample 4-71. A riddle

```
module rw_race;

wire [7:0] out;
assign out = count + 1;

integer count;
initial
begin
   count = 0;
   $write("Out = %b\n", out);
end

endmodule
```

Some simulators do not interrupt blocks to execute continuous assignments.

When the *initial* block assigns a new value to *count*, a simulator could choose to schedule the execution of the continuous assignment for the next timestep, since it is sensitive to *count*. The execution of the *initial* block is not interrupted and the value of *out* displayed is the one it had after initialization, since the continuous assignment has not yet been executed.

Some do.

Another simulator could choose to execute the continuous assignment as soon as count is assigned in the *initial* block. The execution of the *initial* block is interrupted after the assignment to *count* while the continuous assignment is executed. The execution of the *initial* block resumes immediately afterwards.

This type of race condition cannot be avoided easily.

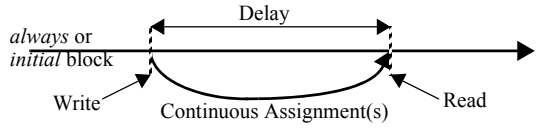
Unfortunately, this type of error condition is not as easy to avoid or eliminate as the one between two blocks. When writing high-level code, you must be careful about the timing between assignments to registers in the right-hand side of a continuous assignment and reading the wire driven by it. To make matters worse, the race condition may involve non-zero delays as well as multiple continuous

assignment statements, such as in Sample 4-72. A *read/write* race condition occurs if the delay between the time the right-hand side of a continuous assignment is updated, and the time any wire on the left-hand side is read, is equal to the propagation delay of all intervening continuous assignments. Figure 4-18 illustrates the timing of these race conditions. The only way to avoid such race conditions is to avoid using continuous assignments for internal decoding logic.

Sample 4-72. Another read/write race condition

```
module rw_race;

wire [7:0] out, tmp;
integer count;
assign #1 out = tmp - 1;
assign #3 tmp = count + 1;

initial
begin
   count = 0;
   #4;
   // "out" will be 0 or x's.
   $write("Out = %b\n", out);
end

endmodule
```

Figure 4-18. Timing of a *read/write* race condition



Write/Write Race Conditions

A *write/write* race condition occurs when two concurrent threads write to the same register at the same timestep. If you look at the code in Sample 4-73, you will notice that both branches of the *fork/join* statement assign the variable *flag* under different conditions and both execute at any change of the clock. This setup creates a *write/write* race condition if both conditions are true.

The execution order determines the final result.

If you refer one more time to Figure 4-11 and the section titled "Emulating Parallelism on a Sequential Processor" on page 162, you will see that both threads are executed one after another, during

Sample 4-73. Example of a write/write race condition

```
program test;

bit flag;

initial
begin
   fork
      @(clk) if (<cond1>) flag = 0;
      @(clk) if (<cond2>) flag = 1;
   join_none
end
endprogram: test
```

the same timestep. Again, the order in which the threads execute is *not deterministic*. Let's assume that both conditions are true. If the first thread is executed first, the value of *flag* is updated to zero. When the second thread is executed, the value of *flag* is updated to one. However, if the second thread executes first, the value of *flag* is updated to one, then it is updated to zero when the first thread executes later.

Sample 4-74. Another example of a write/write race condition

```
module ww_race(clk);
input clk;

reg flag;

always @ (posedge clk)
begin
   if (<cond1>) flag <= 0;
end

always @ (posedge clk)
begin
   if (<cond2>) flag <= 1;
end

endmodule
```

Nonblocking assignments do not solve the problem.

You might be tempted to use the same solution to eliminate the race condition as was used to eliminate the *read/write* race condition, as shown in Sample 4-74. Using nonblocking assignments simply moves the *write/write* race condition from the variable assignment to the scheduling of the nonblocking value. If the first thread executes first, the nonblocking value zero is scheduled for the next timestep. When the second thread executes, the nonblocking value

one is also scheduled for the next timestep, overwriting the previously scheduled value of zero. If the threads execute in the opposite sequence, the scheduled value of zero overwrites the previously scheduled value of one.

There is no way out of this one.

There is no mechanism to prevent this type of race condition. The logic of your model must make sure that both conditions are never true at the same time. It would be a good practice to put an assertion in your model to verify that it is indeed always the case.

Pop quiz!

Why can't you have a *write/write* race condition on a wire?[1]

Initialization Races

initial blocks are no different then *always* blocks.

The most frequent race conditions can be found at the beginning of the simulation, when all threads are executed for the first time. Everything is initialized to its respective specified initializer value or *x* if none are specified. Then the simulation starts normally. It is a common misconception that *initial* blocks are used to initialize variables. *Initial* blocks are *identical* to *always* blocks, except that they execute only once, whereas *always* blocks execute forever, as if they were stuck in an infinite loop.

Initial blocks are not executed first.

When the simulation is started, the *initial* and *always* blocks are executed one after another, in *any order*. The *initial* blocks are *not* executed first—although doing so would not be illegal and some simulators may do just that. Most simulators, for no other reason than to be compatible with Verilog-XL and legacy code containing race conditions, first execute blocks in the same order as they are specified in the file[2]. But the subsequent execution order is not so deterministic.

When simulating the code in Sample 4-75 using an XL-compliant simulator, the first *always* block would be executed and suspended immediately, waiting for the rising edge of the clock. The *initial* block is executed next, assigning the new value of one to the variable named *clk*, which was previously initialized to *x*. A transition

1. Because wires are driven, not assigned. The value from each parallel construct would contribute to the final logic value on the wire, without overwriting the other.
2. You should avoid depending on this behavior.

from *x* to one being considered a rising edge, the first *always* block sees the event and is scheduled to be executed again at the next timestep. However, since the last *always* block was not yet executed, and thus is not waiting for the rising edge of the clock, it does not see this edge. When the last block is finally executed, it is also immediately suspended, waiting for the *next* rising edge on *clk*. An XL-compliant simulator would therefore execute the body of the first *always* block, but not the second. However, that is not a requirement. If a simulator chooses to execute the *initial* block first, the body of neither block would execute at time zero.

Sample 4-75. Race condition at simulation startup

```
module init_race;
reg clk;

always @ (posedge clk)
begin
   $write("Block #1 at %t\n", $time);
end

initial clk = 1'b1;

always @ (posedge clk)
begin
   $write("Block #3 at %t\n", $time);
end

endmodule
```

Guidelines for Avoiding Race Conditions

Race conditions can be avoided if you follow strict coding guidelines. These guidelines differ from typical RTL coding guidelines because of the stricter rules on using blocking vs. nonblocking assignment or the use of continuous assignment statements. RTL coding guidelines are designed to fit the model to the inferred hardware structure. Testbenches use the full language, and as such require guidelines designed to fit the model to the underlying simulation engine.

1. If a variable is declared outside of a concurrent thread structure, assign to it using a nonblocking assignment. Reserve the blocking assignment for variables local to the thread.
2. Assign to a variable from a single concurrent thread.
3. Use continuous assignments to drive inout pins only. Avoid using them to model internal combinatorial functions. Prefer

sequential code in a large *always* block to several continuous assignments.

4. Use initializers to assign initial values. Do not assign any value at time 0.
5. Use *clocking* blocks to sample synchronous module variables in program threads.

Semaphores

Use semaphores.

The problem with guidelines is that there is no way to ensure that everyone follows them. Competing access to shared resources by concurrent threads is an old problem with an equally old solution: the semaphore. When traffic coming from multiple directions (the concurrent thread) needs to cross an intersection (the shared resource), a traffic light (the semaphore) is used to make sure that only one direction of traffic gets to cross the intersection at the same time. A semaphore can be used to ensure that only one thread executes their portion of code that can potentially create a race condition.

A semaphore is a write/write race condition put to good use.

A semaphore is a shared variable that is set by a single execution thread only if the semaphore is currently cleared. That thread is then responsible for clearing the semaphore after completing its access to the shared resource. Sample 4-76 shows an implementation of a semaphore[1] in SystemVerilog. The *in_use* variable indicates whether the semaphore is set. If the *lock* task is invoked while the *in_use* variable is set to 1, the thread waits until the *lock* task is eventually cleared. The *unlock* task clears the semaphore.

This would not work on a true parallel system.

The key to the proper operation of the semaphore implementation shown in Sample 4-76 is the *while* loop in the *lock* task. Let's assume that three concurrent threads are vying for the semaphore by calling the *lock* task at the exact same simulation cycle. Because concurrent threads are really executed sequentially, one at the time, one of these threads (lets call it #1) will execute first. The *in_use* register being equal to 1'bx, the condition of the *while* loop will be false and it will set the semaphore and return from the *lock* task.

1. Computer scientists have a very narrow definition of a semaphore that is probably not met by this implementation. However, it is good enough for now.

Sample 4-76. A SystemVerilog semaphore

```
module semaphore;

bit in_use;

task lock;
   while (in_use === 1'b1) wait (in_use !== 1'b1);
   in_use = 1'b1;
endtask

task unlock;
   in_use = 1'b0;
endtask
endmodule
```

Threads #2 and #3 run, one after another, and get to the *while* loop. Because *in_use* is now set to 1'b1, they enter the *while* loop and wait for *in_use* to be not identical to 1'b1. Eventually thread #1 will release the semaphore by calling the *unlock* task. This will wake up threads #2 and #3. One of them (let's pick #2) will run first, find the condition of the *while* loop false, set the semaphore then return from the *lock* task. When thread #3 runs, it finds the condition of the *while* loop still true (because of thread #2) and waits again.

This semaphore may not be fair.

This simple semaphore implementation relies on the ordering of thread execution to ensure that access is fair. If the simulator implements a first-in-first-out execution order on the *wait* statement (i.e. the thread that has been waiting the longest is run first), then the semaphore will be fair. If it uses a last-in-first-out execution order (i.e. the thread with the most recent invocation of the *wait* statement executes first), then this semaphore will be completely unfair. This simple implementation would also not work if a thread does not suspend its execution between the *unlock* and *lock* task calls: Since *in_use* had just been cleared by the *unlock* task, the *while* loop would not be entered and the thread would acquire the semaphore again.

Semaphores are built-in SystemVerilog.

SystemVerilog comes with a predefined semaphore object. Unlike the user-defined semaphore shown in Sample 4-76, the predefined semaphore is guaranteed to be fair, would work in a potential parallel implementation of SystemVerilog and includes the concept of keys, where a number of identical shared resources can be managed using a single semaphore. See Sample 4-83 for an example of using a predefine semaphore.

PORTABILITY ISSUES

Two compliant simulators can produce different results.

In my many years of consulting in design verification, I have yet to see a *single* testbench that simulates with identical results on different simulators. Half the time, these same testbenches can produce different results by using different command-line options or use a different version of the same simulator! Yet, all simulators are fully compliant with the IEEE standard. Most of the time, the differences are due to race conditions (see "Race Conditions" on page 176). Sometimes, the differences are due to different interpretations of the standard: Many implementation details were left unspecified or existing discrepancies between simulators were also declared "unspecified". Simulator vendors are thus free to implement these unspecified portions of the standard any way they want, yielding different simulation results.

The primary cause is the author's lack of experience.

The primary cause of the simulation differences are the authors. SystemVerilog *appears* easy to learn because it produces the expected response rather quickly. Making sure that the results are reproducible under different conditions is another matter. Learning the idiosyncrasies of the language are what takes time and differentiates an experienced modeler from a new one. It is possible to write testbenches that will simulate with identical results on all simulators and with all command-line options.

Events from Overwritten Scheduled Values

If a scheduled value is overwritten by another scheduled value, can the original value cause an event? The answer to that question is left undefined by the SystemVerilog standard. If you look at the code in Sample 4-77, will anything be displayed at time 10?

Overwriting a scheduled value may generate an event.

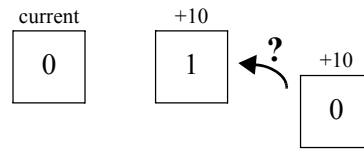
Figure 4-19 shows the queue of scheduled future values for register *strobe* just before the last statement of the *initial* block is about to execute. After executing that last statement, and scheduling the new value of zero after ten time units in the future, what happens to the previously scheduled value of one? Is it removed? Is it left there? If so, which value will be assigned to *strobe* ten time units from now? Only zero (and thus not generating an event on *strobe*) or both in zero-time (and generating an event)? The answer to this question is

Sample 4-77. Overwriting scheduled values

```
module events;

reg stobe;

always @ (strobe)
begin
   $write("Stobe is %b\n", strobe);
end

initial
begin
   strobe = 1'b0;
   strobe <= #10 1'b1;
   strobe <= #10 1'b0;
end

endmodule
```

simulator dependent. In asynchronous descriptions, avoid overwriting previously scheduled values using nonblocking assignments.

Figure 4-19. Event queue on *strobe*



Disabled Scheduled Values

Nonblocking assignment values may be affected by the *disable* statement.

The SystemVerilog standard does not specify what happens to still-pending values that were scheduled using a nonblocking assignment within a block that is disabled. Consider the code in Sample 4-78. When a reset condition is detected, the *always* block modeling the CPU interface is disabled to restart it from the beginning. What should happen to the various values assigned to the CPU interface signals *data* and *dtack* using nonblocking assignments, but that may not have been assigned to the variables yet? Depending on the simulator you are using, these values may be removed from the scheduled value queue and never make it to the intended variables, or they may remain unaffected by the *disable* statement. Avoid disabling a block where nonblocking assignments are performed.

Sample 4-78. Nonblocking assignments potentially affected by a *disable* statement

```
module cpuif(...);

always
begin: if_logic
   ...
   data  <= #(Ta) read_val;
   dtack <= #(Tack) 1'b1;
   @ (negedge ale);
   data  <= #(Thold) 32'bz;
   dtack <= #(Thold) 1'b0;
   ...
end

always wait (reset == 1'b1)
begin
   disable if_logic;
   wait (reset != 1'b1);
end
endmodule
```

Output Arguments on Disabled Tasks

Output values may not make it out of disabled tasks.

Another area where the behavior of SystemVerilog is left unspecified is the value of output arguments in disabled tasks. Look at the code in Sample 4-79. The *read* task has an output argument returning the value that was read. Within the task, a *disable* statement is used to abort its execution at the end of the read cycle. Because the entire task was disabled, whether the value of *rdat* is copied out into the register *actual* used to invoke the task is not specified in the SystemVerilog standard.

Use the *return* statement instead.

In some simulators, the value of *actual* is updated with the value of *rdat*, effectively completing the read cycle. In some others, the value of *actual* remains unchanged, leaving the read cycle incomplete. This unspecified behavior can be avoided easily by using the *return* statement instead of disabling the task itself, as shown in Sample 4-80.

Non-Re-Entrant Tasks

This is not an unspecified behavior.

Unless a task is declared as *automatic*, they are not re-entrant. Non-re-entrant tasks are not really an unspecified behavior in SystemVerilog. All simulators have non-re-entrant tasks because every declaration in a SystemVerilog model, except for *classes*, is static. By default, no declaration is dynamically allocated upon invocation of a subprogram or entry into a block of code.

Sample 4-79. Unspecified behavior of disabled tasks

```
task read(input  [7:0] radd,
          output [7:0] rdat);
   ...
   if (valid) begin
      rdat = data;
      disable read;
   end
   ...
endtask

initial
begin: test_procedure
   reg [7:0] actual;

   read(8'hF0, actual);
   ...
end
```

Sample 4-80. Using *return* instead of *disable*.

```
task read(input  [7:0] radd,
          output [7:0] rdat);
   ...
   if (valid) begin
      rdat = data;
      return;
   end
   ...
endtask
```

The same memory space is used for all invocations of a task.

When you declare a task or a function, the memory space for its arguments and all other locally declared variables is allocated at compile time. There is a single location for the subprogram and all of its local variables. The memory is not allocated at runtime each time the task or function is invoked. Every time a subprogram is invoked, the *same* memory space is used. This reuse of memory space does not cause problems in functions or in tasks that do not include @, # or *wait* statements because the local data space is used in a single invocation. The memory space is no longer in use by the time a second invocation is made. However, if a task contains timing control statements, it may still be active when a second invocation is made.

A second invocation clobbers the data space of an active prior invocation.

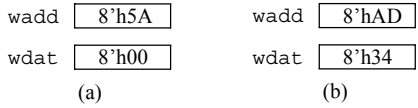
Examine the code in Sample 4-81. The task named *write* contains timing control statements and is invoked from two different *initial* blocks. In Figure 4-20(a), the content of the arguments, local to the task, is shown after the invocation from the first *initial* block. While

this first invocation is waiting, the second *initial* block is executed[1] and invokes the *write* task again, setting its local arguments to the values shown in Figure 4-20(b). When the first invocation resumes, it continues its execution, using the arguments provided by the second invocation: Its data space was overwritten. The first invocation goes on to write the value 8'h34 at address 8'h5A.

Sample 4-81. Non-re-entrant task

```
task write(input [7:0] wadd,
           input [7:0] wdat);
   ad_dt <= wadd;
   ale   <= 1'b1;
   rw    <= 1'b1;
   @ (posedge rdy);
   ad_dt <= wdat;
   ale   <= 1'b0;
   @ (negedge rdy);
endtask

initial write(8'h5A, 8'h00);
initial write(8'hAD, 8'h34);
```

Figure 4-20. Task data space



Concurrent task activations may not be so obvious.

The concurrent invocation of the same task in Sample 4-81 is pretty obvious. But most of the time, the conditions where a task is activated more than once are much more obscure. In a large verification environment, with numerous tasks invoked under a complex control structure, it is very easy to concurrently activate a task and corrupt an entire testcase without you, or the simulator, being aware of it.

automatic tasks *are* re-entrant.

In SystemVerilog, tasks can be made re-entrant by declaring them *automatic*. This causes the arguments and local variables to be dynamically created upon invocation of the task. Because these variables are no longer static, they cannot be referred to externally using a hierarchical name, nor can they be displayed on a waveform

1. This specific execution order is only an example. The *initial* blocks could execute in reverse order with equally catastrophic results.

viewer. Furthermore, making a task re-entrant only solves the local data part of the problem. Separate threads still exist with the potential for race conditions to shared variables. For example, if the task in Sample 4-81 were made re-entrant by adding the keyword *automatic*, would the problem be solved? No. Even though each thread would have their correct respective values of the address and data to write, both will assign to the *same shared variables ad_dt*, *ale* and *rw*, creating write/write race conditions (see "Write/Write Race Conditions" on page 180).

Use a semaphore to detect concurrent task activation.

The best approach to avoid this fatal condition is to use a semaphore to detect concurrent activation or protect against the write/write race condition. When using *automatic* tasks, a semaphore will ensure proper operation of the task, as shown in Sample 4-82.

Sample 4-82. Using a semaphore in a re-entrant task

```
semaphore sem = new(1);

task automatic write(input [7:0] wadd,
                     input [7:0] wdat);

   sem.get(1);
   ...
   sem.put(1);
endtask
```

A semaphore does not help non-re-entrant tasks.

Semaphores can only help protect shared resources if they are used *before* the shared resource is accessed. The solution shown in Sample 4-82 would not work for a non-re-entrant task because the data space of the task was already corrupted. It is too late. One solution would be to use the semaphore before the non-re-entrant task is invoked, as shown in Sample 4-83. What if someone forgets to use the semaphore before calling the task?

You can detect concurrent task activation.

A modified version of the semaphore can be used to detect concurrent activation of a non-re-entrant task. As shown in Sample 4-84, the state of the semaphore indicates whether the task is currently activated. If the task is invoked while the key in the semaphore is checked out, the simulation is terminated. Because the data space of the task has already been clobbered, it is not possible to recover from the error. Terminating the simulation is the only option. The

Sample 4-83. Using a semaphore with a non-re-entrant task

```
semaphore sem = new(1);

task write(...);
...
endtask

initial
begin
   sem.get(1);
   write(8'h5A, 8'h00);
   sem.put(1);
end

initial
begin
   sem.get(1);
   write(8'hAD, 8'h34);
   sem.put(1);
end
```

problem must be fixed by retiming the access to the task (usually through a semaphore) to ensure that no concurrent invocation takes place.

Sample 4-84. Guarding non-re-entrant task

```
task write(input [7:0] wadd,
           input [7:0] wdat);

   semaphore sem = new(1);

   if (!sem.try_get(1)) $stop;

   ad_dt <= wadd;
   ale   <= 1'b1;
   rw    <= 1'b1;
   @ (posedge rdy);
   ad_dt <= wdat;
   ale   <= 1'b0;
   @ (negedge rdy);

   sem.put(1);
end
endtask
```

I always put a semaphore around any non-re-entrant task. This let my model tell me immediately if I misused it. I could immediately fix the problem, without having to diagnose a testbench failure back to a concurrent task activation. The time invested in adding the

semaphore was well worth it. If the task I wrote was to be used by others, the message produced by the concurrent activation detection specifically stated that the error was not in my task code, but in *their* use of it and to go look for a concurrent activation. This has saved me many technical support calls.

Static vs. Automatic Variables

Variables are *static* by default.

Unless declared in a dynamic context—within a *class* or an *automatic* task—all variables in SystemVerilog are static by default. Static variables are not really an unspecified behavior in SystemVerilog but can be a source of unexpected behavior. A single copy exists for a static variable. It is created and initialized at the beginning of the simulation and is reused by all threads referencing that variable. A variable in a dynamic scope or explicitly declared as *automatic* is created and initialized every time a thread enters the scope in which the dynamic variable is declared.

Static variables are initialized only once.

Variables are initialized only when they are created. Because static variables are created only once, at the beginning of the simulation, they are thus initialized only once. For someone with a non-Verilog background, that may produce some unexpected behavior, as illustrated in Sample 4-85. The variable *count* is a static variable. It will be initialized to zero only once, at the beginning of the simulation. Even if the design operates correctly, the value of *count* will eventually exceed the limit of 10 because it is never reset back to zero when the task is invoked. The expected behavior can be obtained by either declaring the *count* variable as *automatic* as shown in Sample 4-86, or explicitly initializing it to zero at the beginning of the task as shown in Sample 4-87.

Sample 4-85. Static variables in loops.

```
task bus_request;
   int count = 0;
   req <= 1;
   while (gnt != 1) begin
      @ (posedge clk);
      count++;
   end
   if (count > 10) $write(...);
endtask: bus_request
```

Sample 4-86. Automatic variables in loops.

```
task bus_request;
   automatic int count = 0;
   req <= 1;
   while (gnt != 1) begin
      @ (posedge clk);
      count++;
   end
   if (count > 10) $write(...);
endtask: bus_request
```

Sample 4-87. Initializing static variables in loops.

```
task bus_request;
   int count;
   count = 0;
   req <= 1;
   while (gnt != 1) begin
      @ (posedge clk);
      count++;
   end
   if (count > 10) $write(...);
endtask: bus_request
```

Only one instance of static variables exists.

Static variables are created only once, at the beginning of the simulation. Only one copy exists, no matter how many threads enter the scope where they are declared. Again, this may produce some unexpected behavior, as illustrated in Sample 4-88. Variable *id* is static. Only one copy exists. Therefore, every thread forked off inside the *for-loop* will be sharing the same variable and checking for the same index within the *ack* array. The intended functionality can be obtained by declaring the *id* variable *automatic* as shown in Sample 4-89

Sample 4-88. Static variables in *fork/join* statements.

```
int i;
...
for (i = 0; i < 10; i++) begin
   fork
      begin
         int id = i;
         while (ack[id] != 1) @ (posedge clk);
         ...
      end
   join_none
end
```

Sample 4-89. Automatic variables in *fork/join* statements.

```
int i;
...
for (i = 0; i < 10; i++) begin
   fork
      begin
         automatic int id = i;
         while (ack[id] != 1) @ (posedge clk);
         ...
      end
   join_none
end
```

SUMMARY

When writing testbenches, think function, not implementation. Abandon the RTL coding mindset. Do not think in terms of logic, registers and state machines. Think in terms of data transformation, program state and execution flow.

Your first objective is to write maintainable code. Write relevant comments that describe your intent, not the code. Optimize for performance only when necessary.

Minimize the scope of your variables as much as possible. Declare local variables in the scope where they are needed.

Package reusable subprograms and bus-functional models in classes to facilitate their reuse. Maintain separate name spaces as much as possible. Make sure that it is possible to have multiple instances of a bus-functional model connected to different interface signals without interference or collisions.

Use data abstraction. Collect related data into classes, arrays and queues.

Separate public interfaces from private implementation. Plan your class inheritance and take advantage of polymorphism to create generic bus-functional models and utility subprograms.

Understand the concurrency model used in simulating SystemVerilog. It will help write more efficient models and avoid race conditions. Use semaphores to protect shared resources.

Understand the unspecified portion of the SystemVerilog standard. This portion is a source of non-portability between different SystemVerilog simulators, versions and command-line options.

CHAPTER 5 STIMULUS AND RESPONSE

The purpose of writing testbenches is to apply stimulus to a design and observe the response. That response must then be compared against the expected behavior.

Generating stimulus is the process of providing input signals to the design under verification as shown in Figure 5-1. From the perspective of the stimulus generator, every input of the design is an output of the generator.

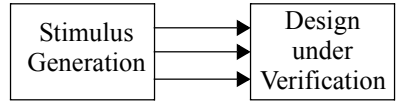


Figure 5-1. Stimulus generation

Monitoring is the process of observing output signals from the design under verification as shown in Figure 5-2. From the perspective of the response monitor, every output of the design is an input of the monitor.

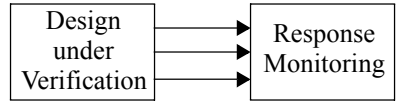


Figure 5-2. Response monitoring

This chapter shows how to apply stimulus and observe response.

In this chapter, I show how to generate stimulus and observe responses. I also show how to abstract data flowing to and from the design from a physical level composed of 1's, 0's and elapsed time to a transaction level composed of data objects and procedures. The greatest challenge with stimulus is making sure it is an accurate representation of the environment, not just a simple case. When monitoring responses, one has to be careful not to miss any data and detect as many errors as possible.

The next chapter shows how to structure a testbench.

In the next chapter, I show how to best structure the stimulus generators and response monitors to create a layered self-checking environment. Constrainable random generation is then added on top of the stimulus generators and response monitors. If you prefer a top-down perspective, I recommend you start with the next chapter then come back to this one.

REFERENCE SIGNALS

Clock signals must be generated with care.

Because a clock signal has a very simple repetitive pattern, it is one of the first and most fundamental signals to generate. It is also the most critical signal to generate accurately. Many other signals use clock signals to synchronize themselves.

Use a module thread.

Generate the clock signals using a module thread. Program threads are designed to be reactive to the events occurring in the design. Clock signals are the primary cause of these events. The design reacts to clock events. The *always* blocks generating the clock signals should be inside a *module*, as shown in Sample 5-1.

Explicitly initialize the clock variable.

The code to generate a 50 percent duty-cycle 100MHz clock signal is shown in Sample 5-1. Using a statement like `clk = ~clk` depends on the proper initialization of the clock signal to a value different than the default value of 1'bx. Initializing the clock variables using an explicit initializer value also prevents the generation of clock events at time zero, potentially creating initialization race conditions, as described in "Initialization Races" on page 182.

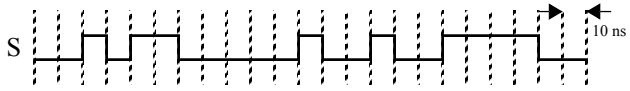
Sample 5-1. Generating a 50% duty-cycle clock

```
module tb_top;
bit clk = 0;
always #5 clk = ~clk;
...
endmodule
```

Any repetitive waveform is easy to generate.

Waveforms with deterministic edge-to-edge relationships with an easily identifiable period also are easy to generate. It is a simple process of generating each edge in sequence, at the appropriate time. For example, Figure 5-3 outlines an apparently complex waveform. However, Sample 5-2 shows that it is simple to generate.

Figure 5-3. Apparently complex waveform



Sample 5-2. Generating a deterministic waveform

```
always
begin
   S = 1'b0; #20ns;
   S = 1'b1; #10ns;
   S = 1'b0; #10ns;
   S = 1'b1; #20ns;
   S = 1'b0; #50ns;
   S = 1'b1; #10ns;
   S = 1'b0; #20ns;
   S = 1'b1; #10ns;
   S = 1'b0; #20ns;
   S = 1'b1; #40ns;
   S = 1'b0; #20ns;
   ...
end
```

Time Resolution Issues

Integer division may speed-up the clock.

When generating waveforms in SystemVerilog, you must select the appropriate timescale and precision to properly place the edges at the correct offset in time. When using an expression, such as `cycle/2`, to compute delays, you must make sure that integer operations do not truncate a fractional part. For example, the clock generated in Sample 5-3 produces a period of 14 nanoseconds because of truncation caused by the integer division.

The time-scale may affect the timing of edges.

If the precision of the currently active timescale is not sufficiently high, delay values are rounded up or down. When this happens to the delay values of clock signals, it shifts the relative position of the clock edges. For example, the clock generated in Sample 5-4 pro-

Sample 5-3. Truncation errors in stimulus generation

```
‘timescale 1ns/1ns
module testbench;
...
bit clk = 0;
parameter cycle = 15;
always
begin
   #(cycle/2);  // Integer division
   clk =~clk;
end
endmodule
```

duces a period of 16 nanoseconds because of rounding the result of the real division to an integer value.

Sample 5-4. Rounding errors in stimulus generation

```
‘timescale 1ns/1ns
module testbench;
...
bit clk = 0;
parameter cycle = 15;
always
begin
   #(cycle/2.0);   // Real division
   clk = ~clk;
end
endmodule
```

Because the timescale in Sample 5-5 offers the necessary precision for a 7.5 nanoseconds half-period, only this model generates a 50 percent duty-cycle clock signal with a precise 15 nanoseconds period.

Sample 5-5. Proper precision in stimulus generation

```
‘timescale 1ns/100ps
module testbench;
...
bit clk = 0;
parameter cycle = 15;
always
begin
   #(cycle/2.0);
   clk = ~clk;
end
endmodule
```

Aligning Signals in Delta-Time

Delta delays are functionally equivalent to real delays.

In the specification shown in Figure 5-4, the transition of *clk2* is aligned with a transition on *clk*. There are many ways of generating these two signals. Depending on the approach used, these aligned transitions may occur in the same delta cycle, or in different delta cycles. Although delta-cycle delays are considered zero-delays by the simulator, functionally they have the same effect as real delays.

A derived waveform, such as the one shown in Figure 5-4, apparently is easy to generate. A simple *always* block, sensitive to the proper edge of the original signal as shown in Sample 5-6, and voila! Even the waveform viewer shows that it is right!

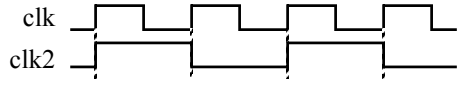


Figure 5-4. Derived waveform specification

Sample 5-6. Improperly generating a derived waveform

```
always @(posedge clk)
begin
   clk2 <= ~clk2;
end
```

Watch for delta delays in derived waveforms.

The problem is not visually apparent. Because of the simulation cycle (See “The Simulation Cycle” on page 163.), there is a delta cycle between the rising edge of the base clock signal and the transition on the derived clock signal, as shown in Figure 5-5. Any data transferred from the base clock domain to the derived clock domain goes through this additional delta cycle delay. In a zero-delay simulation, such as a transaction-level or RTL model, this additional delta-cycle delay can have the same effect as an entire clock cycle delay.

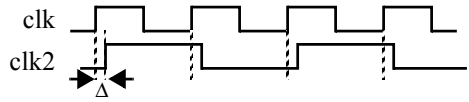


Figure 5-5. Delta delay in derived waveform

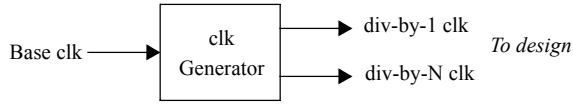
Propagation delays make it work in the real world.

Why is it that generating divided clocks in simulation the same way it is done in the real world does not work? Because, in synchronous designs, there is always a race condition between the *clk*-to-D-input path and the *clk*-to-*clk2* path of adjacent flip-flops. This constant race condition is solved by making sure that the delay through the *clk*-to-D path is always longer than the delay through the *clk*-to-*clk2* path. In the real world, these signal propagation delays will never be zero. Device physics and clock skew management provides a simple solution. In zero-time transaction-level or RTL models, propagation delays are composed of delta cycles. If the number of delta cycles in the *clk*-to-D path is smaller than the number of delta cycles in the *clk*-to-*clk2* path, an entire clock cycle delay will be lost.

Align derived signals in delta-time.

The solution is surprisingly similar to that used in the real world. It is necessary to minimize the delta-cycle skew between the base and derived signals. This skew can be completely eliminated by aligning their respective edges in delta time. The only way to perform this task is to re-derive the base signal through a divide-by-1 operation, as shown in Sample 5-7 and illustrated in Figure 5-6. The base signal is never used by other threads. Instead, they must use the divide-by-1 signal.

Figure 5-6. Generation of aligned derived signals



Sample 5-7. Properly generating a derived waveform

```
always @(clk)
begin
   clk1 <= clk;
   if (clk == 1'b1) clk2 <= ~clk2;
end
```

Differential data signals need not be aligned.

When generating a differential data signal pair, it is not necessary to align both polarities in the same delta cycle. Adding an inversion delay in one of the phase signals only adds to the clock-to-D-input path delay. This technique goes in the right direction to solve the race condition. As shown in Sample 5-8, the inversion of the *d* signal in the connection to the *dn* pin may introduce an additional delta cycle in the *d*-to-*dn* path compared to the *d*-to-*dp* path.

Sample 5-8. Generating differential data signals

```
wire [15:0] d;

bfm cpu(..., .d(d), ...);
design dut (..., .dp(d), .dn(~d), ...);
```

Clock Multipliers

Implemented using PLLs.

Many designs have a very high-speed front-end interface that is driven using a multiple of a recovered clock or the lower-frequency system clock. This clock multiplication is performed using an internal or external PLL (phase locked loop). PLLs are inherently analog circuits. They are very costly to simulate in a digital simulator. When an internal PLL is used, the analog component that implements the PLL is often modeled as an empty module. It is up to the testbench to create an appropriate multiplied clock signal in a behavioral fashion.

The reference clock could become the derived clock.

A simple strategy is to reverse the role of the reference and derived clock. Since clock dividers are so easy to model, you could generate the high-frequency clock then use it to derive the lower-frequency system-clock. Sample 5-9 shows a model for a multiply-by-4 clock generator using the divide-by-4 strategy. But this only works under two conditions: The reference clock is also an input to the design, and the frequency of the reference clock is known and fixed.

Sample 5-9. Generating clock multiples by division

```
bit clk1 = 0;
bit clk4 = 0;
always
begin
   repeat (4) #10 clk4 <= ~clk4;
   clk1 <= ~clk1;
end
```

Synchronize the multiplied clock to the reference clock.

The first condition can be eliminated by synchronizing the multiplied clock signal with the reference clock. It will be possible to generate the multiplied clock signal even if the reference clock is supplied by the design. Sample 5-10 shows a model of a multiply-by-4 clock generator, synchronized with an input reference clock. But the problem of the hard-coded multiplied clock period remains. This model assumes a reference clock with an 80 nanoseconds period. What if the reference clock has a different frequency in a

different simulation run? How can we generalize this model into a generic clock-multiplier PLL model?

Sample 5-10. Synchronizing multiplied clock to input reference clock

```
always @(clk)
begin
   clk4 <=       ~clk4;
   clk4 <= #10ns  clk4;
   clk4 <= #20ns ~clk4;
   clk4 <= #30ns  clk4;
end
```

Measure the period of the reference clock.

Why not let the model learn the period of the reference signal? You can measure the time difference between two consecutive edges, divide this value by 4, and voila! A generic PLL model. Sample 5-11 shows a PLL model with a continuous measure of the reference signal. As the frequency of the reference clock signal changes, the frequency of the multiplied clock will adapt.

Sample 5-11. Adaptive clock multiplier model

```
module pll(input  bit ref_clk,
           output bit out_clk);
parameter FACTOR = 4;

initial
begin
   real stamp;

   out_clk = 1'b0;
   @(ref_clk);
   stamp = $realtime;

   forever begin
      real period;

      @(ref_clk);
      period = ($realtime - stamp)/FACTOR;
      stamp = $realtime;

      repeat (FACTOR-1) begin
         out_clk = ~out_clk;
         #(period);
      end
      out_clk = ~out_clk;
   end
end
endmodule
```

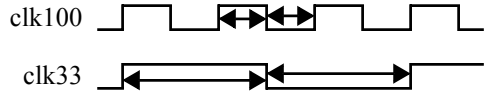
Watch that timescale!

The usual words of caution (see “Time Resolution Issues” on page 199) apply regarding the precision of the timescale. The computed period of the multiplied signal is a real value that will likely have a fractional part. The actual delay value between two consecutive edges of the multiplied clock will be the computed value rounded to the current timescale precision. If the size of this error is small enough compared to the period of the reference signal, this should not cause a problem.

Asynchronous Reference Signals

Figure 5-7 shows a specification for two unrelated clock signals. They are used by two separate clock domains in the design under verification. *clk100* is a 100 MHz signal while *clk33* is a 33 MHz signal. You could be tempted to model these two clock signals as shown in Sample 5-12, using the higher frequency signal as a reference to generate the lower-frequency one with a divide-by-3 strategy. This approach will indeed generate the waveforms shown in Figure 5-7. But that is only one of the possible solutions, and one that may not highlight some classes of problems.

Figure 5-7. Unrelated waveform specification



Sample 5-12. Improperly generating unrelated waveform

```
always @(clk100)
begin
   int count = 0;

   count = count + 1;
   if (count == 3) begin
      clk33 <= ~clk33;
      count = 0;
   end
end
```

Alignment on paper is not a specification.

The problem comes from the *inference* that the waveforms are aligned simply because they are aligned in the figure. There are no explicit or implicit timing relationships between the two signals as there is no timing arrow going from an edge in one waveform to an edge in the other waveform. Drawing tools have a grid system that facilitates drawing straight lines. But they also have the side effect

of aligning objects. When writing a specification, you must be careful that these implicit alignments do not create the illusion of a relationship. When reading a specification, do not assume a relationship unless it is explicitly stated.

Generate unrelated signals in separate threads.

Sample 5-13 shows a better way to generate these unrelated clock signals. Since they are not synchronized in any way, they are generated using separate concurrent threads. This separation will make it easier to modify the frequency of one signal without affecting the frequency of the other. Also, notice how each signal is explicitly skewed at the beginning of the simulation to avoid having the edges aligned at the same simulation time. This approach is a good practice to highlight potential problems in the clock-domain crossing portion of the design. By varying these initial signal skew values, it will be possible to verify the correct functionality of the design across different asynchronous clock relationships.

Sample 5-13. Generating unrelated waveforms

```
bit clk100 = 0;
initial #2 forever #5 clk100 = ~clk100;

bit clk33 = 0;
initial #5 forever #15 clk33 = ~clk33;
```

Random Generation of Reference Signal Parameters

All signals are related in simulation.

In the previous section, I explained why unrelated signals should be modeled as separate threads and skewed with respect to each other to avoid creating an implicit relationship that does not exist between them. The truth is: There is no way to accurately model unrelated signals. Each waveform is described with respect to the current simulation time. Because all waveforms are described using the same built-in reference, they are all implicitly related. Even though I made my best effort to avoid modeling any relationship between the two clock signals generated in Sample 5-13, they *are* related because of the deterministic nature of the simulator. Unless I manually modify one of the timing parameters, they will maintain the same relationship in all simulations.

Asynchronous means random.

When we say that two signals are asynchronous to each other, we are saying that they have a random phase relationship. That phase relationship will be different every time and cannot be predicted. When I specified explicit skew delay values in Sample 5-13, I intro-

duced certainty where there wasn't any. These delay values should be generated randomly to increase the chances that, over the thousands of simulation runs the design will be subjected to, any problem related to clock skews will be highlighted.

Avoid using *$random*.

One solution would be to call *$random* to generate the delay values, as shown in Sample 5-14. But this strategy can only produce evenly distributed delay values. It would not be possible to modify the distribution of delay values toward more interesting corner cases, or constrain the delay values against each other to create interesting conditions.

Sample 5-14. Generating unrelated waveforms using random skew

```
bit clk100 = 0;
initial #({$random} % 10)
   forever #5 clk100 = ~clk100;

bit clk33 = 0;
initial #({$random} % 30)
   forever #15 clk33 = ~clk33;
```

Use skew variables initialized with *$random*.

Instead, as shown in Sample 5-15, use skew variables initialized using *$random* to determine the initial skew of each asynchronous waveform. Although it appears that little has been gained compared with Sample 5-14, this approach allows a testbench to modify or constrain the skew values. Notice the *#1* delay inserted before the actual skew delay. This allows the testbench code, written in a *program thread*, to run and potentially replace the skew values before they are used. Sample 5-16 shows how a testbench can generate and replace new skew values with constrained random values that are deemed more interesting.

Sample 5-15. Generating unrelated waveforms using skew variables

```
bit clk100 = 0;
int clk100_skew = {$random} % 10;
initial #1 #(clk100_skew)
   forever #5 clk100 = ~clk100;

bit clk33 = 0;
int clk33_skew = {$random} % 30;
initial #1 #(clk33_skew)
   forever #15 clk33 = ~clk33;
```

Sample 5-16. Randomly generating new skew values

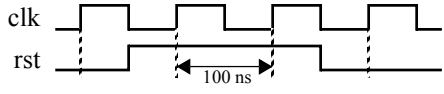
```
program test;
initial
begin
   std::randomize(tb_top.clk100_skew,
                  tb_top.clk33_skew) with {
      tb_top.clk100_skew == tb_top.clk33_skew;
   };
   ...
end
endprogram
```

Applying Reset

Synchronized signals must be properly modeled.

The first signal to be generated after the clock signals is the hardware reset signal. The reset signal must be shaped properly to reset the design correctly. The generation of a synchronous reset signal should also reflect its synchronization with any clock signal. For example, consider the specification for a reset signal shown in Figure 5-8. The code in Sample 5-17 shows how such a waveform is generated frequently.

Figure 5-8. Reset waveform specification



Sample 5-17. Improperly generating a synchronous reset

```
bit clk = 0;
always #50 clk = ~clk;

bit rst = 0;
initial
begin
   #150 rst = 1'b1;
   #200 rst = 1'b0;
end
```

Race conditions can be created easily between synchronized signals.

There are two problems with the way these two waveforms are generated in Sample 5-17. The first problem is functional: There is a race condition between the *clk* and *rst* signals. At simulation time 150, and again later at simulation time 350, both variables are assigned at the same timestep. Because the *blocking* assignment is used for both assignments, one of them is assigned first. A block sensitive to the falling edge of *clk* may execute before or after *rst* is assigned. From the perspective of that block, the specification

shown in Figure 5-8 could appear to be violated. The race condition can be eliminated by using *nonblocking* assignments, as shown in Sample 5-18. Both *clk* and *rst* signals are assigned between timesteps when no blocks are executing. If the design under verification uses the falling edge of *clk* as the active edge, *rst* is already—and reliably—assigned.

Sample 5-18. Race-free generation of a synchronous reset

```
bit clk = 0;
always #50 clk <= ~clk;

bit rst = 0;
initial
begin
   #150 rst <= 1'b1;
   #200 rst <= 1'b0;
end
```

Lack of maintainability can introduce functional errors.

The second problem, which is just as serious as the first one, is maintainability of the description. You could argue that the first problem is more serious, since it is functional. The entire simulation can produce the wrong result under certain conditions. Maintainability has no such functional impact. Or has it? What if you made a change as simple as changing the phase or frequency of the clock. How would you know to change the generation of the reset signal to match the new clock waveform?

Conditions in real life are different than those in this book.

In the context of Sample 5-18, with Figure 5-8 nearby, you would probably adjust the generation of the *rst* signal. But outside of this book, in the real world, these two blocks could be separated by hundreds of lines, or even be in different files. The specification is usually a document three centimeters thick, printed on both sides. The timing diagram shown in Figure 5-8 could be buried in an anonymous appendix, while the pressing requirements of changing the clock frequency or phase was stated urgently in an email message. And you were busy debugging this other testbench when you received that pesky email message! Would you know to change the generation of the reset signal as well? I know I would not.

Model the synchronization within the generation.

Waiting for an apparently arbitrary delay cannot guarantee synchronization with respect to the delay of the clock generation. A much better way of modeling synchronized waveforms is to include the synchronization in the generation of the dependent signals, as shown in Sample 5-19. The proper way to synchronize the *rst* sig-

nal with the *clk* signal is for the generator to wait for the significant synchronizing event, whenever it may occur. The timing or phase of the clock generator can be modified now, without affecting the proper generation of the *rst* waveform. From the perspective of a design sensitive to the falling edge of *clk*, *rst* is reliably assigned one delta-cycle after the clock edge.

Sample 5-19. Proper generation of a synchronous reset

```
bit clk = 0;
always #50 clk = ~clk;

bit rst = 0;
initial
begin
   @ (negedge clk);
   rst <= 1'b1;
   @ (negedge clk);
   @ (negedge clk);
   rst <= 1'b0;
end
```

Reset may need to be applied repeatedly during a simulation.

There is a problem with the way the *rst* waveform is generated in Sample 5-19. The *initial* block runs only once and is eliminated from the simulation once completed. There is no way to have it execute again during a simulation. What if it were necessary to reset the device under verification multiple times during the same simulation? An example is the "hardware reset" testcase that verifies proper reset operation: After setting some internal registers, the hardware reset must be applied to verify that these registers return to their reset value. The ability to control reset application is also very useful. This control lets testbenches perform preparatory operations before resetting the design and starting the actual stimulus.

Generate reset from within a module task.

The proper mechanism to encapsulate statements that you may need to repeat during a simulation is to use a *task* as shown in Sample 5-20. To repeat the reset signaling, simply call the task. To maintain the behavior of using an *initial* block to reset the device under verification automatically at the beginning of the simulation (which may or may not be desirable), simply call the task in an *initial* block.

Reset task should be in *program*.

It will be possible to invoke the reset task in the module from the testbench *program* threads. The testbench will thus be able to reset the design any time the reset is required. An alternative would be to put the reset task in a program task, as shown in Sample 5-21.

Sample 5-20. Encapsulating the generation of a synchronous reset

```
module tb_top;

bit clk = 0;
bit rst = 0;
always #50 clk = ~clk;
...
task hw_reset
   rst = 1'b0;
   @ (negedge clk);
   rst <= 1'b1;
   @ (negedge clk);
   @ (negedge clk);
   rst <= 1'b0;
endtask: hw_reset
initial hw_reset;
...
endmodule
```

Because modules cannot call program tasks, it will be necessary for each testbench to call the reset program task to reset the DUT. However, this gives the testbench better control over the reset parameters and its coordination with other device stimuli, not just the clock. The section titled "Simulation Control" starting on page 124 of the *Verification Methodology Manual for SystemVerilog* defines a simulation sequence methodology that includes a predefined step for applying hardware reset to the design under verification.

Sample 5-21. Synchronous reset program task.

```
program test;

task hw_reset
   tb_top.rst <= 1'b0;
   @ (negedge tb_top.clk);
   tb_top.rst <= 1'b1;
   @ (negedge tb_top.clk);
   @ (negedge tb_top.clk);
   tb_top.rst <= 1'b0;
endtask: hw_reset

initial
begin
   hw_reset;
   ...
end
endprogram
```

Are you paying attention?

Pop quiz: What is missing from the *hw_reset* task in Sample 5-20 and Sample 5-21? The answer can be found in this footnote.[1]

SIMPLE STIMULUS

In this section, I explain how to generate deterministic waveforms. Various techniques are developed to generate stimulus signals in the best way. I also demonstrate how to encapsulate and package signal generation operations using simple bus-functional models.

Applying Synchronous Data Values

There is a race condition between the clock and data signal.

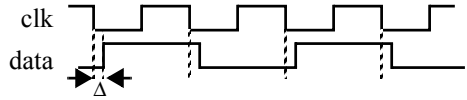
Sample 5-22 shows how you could generate a zero-delay synchronous data waveform. This approach is identical to the way flip-flops are inferred in an RTL model. As illustrated in Figure 5-9, there is a delay between the edge on the clock and the transition on *data,* but the delay is a single delta cycle. In terms of simulation time, there is no delay. For RTL models, this infinitesimal clock-to-Q delay is sufficient to model the behavior of synchronous circuits properly. However, this delay assumes that all clock edges are aligned in delta time (see "Aligning Signals in Delta-Time" on page 201). If you are generating both clock and data signals from the outside of the model of the design under verification, you have no way of ensuring that the total number of delta-cycle delays between the clock and the data is maintained and that the data signal will arrive before the clock.

Sample 5-22. Zero-delay generation of synchronous data

```
initial
begin
   @ (negedge tb_top.clk);
   tb_top.data <= ...;
   @ (negedge tb_top.clk);
   tb_top.data <= ...;
   ...
end;
```

1. The task *hw_reset* contains delay control statements. The task should contain a semaphore to detect concurrent activation. You can read more about this issue in "Non-Re-Entrant Tasks" on page 188.

Figure 5-9. Synchronous data waveforms



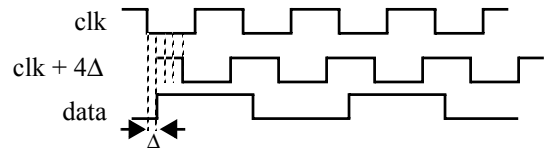
The clock may be delayed more than the data.

For many possible reasons, the clock signal may be delayed by more delta cycles than its corresponding data signal. These delays could be introduced by using different I/O pad models for the clock and data pins. They could also be introduced by the clock distribution network, which does not exist on the data signal. If the clock signal is delayed more than the data signal, even in zero-time as shown in Figure 5-10, the effect is the same as removing an entire clock cycle from the data path.

Delay the data from the active clock edge.

Interface specifications never specify zero-delay values. A physical interface always has a real delay between the active edge of a clock signal and its synchronous data. When generating synchronous data, always provide a real delay between the active edge and the transition on the data signal, as shown in Sample 5-23, or synchronize the data signal with the inactive edge of the clock.

Figure 5-10. Delta delays in clock path



Sample 5-23. Non-zero-delay generation of synchronous data

```
initial
begin
   @ (negedge tb_top.clk);
   tb_top.data <= #1 ...;
   @ (negedge tb_top.clk);
   tb_top.data <= #1 ...;
   ...
end
```

Use a *clocking* block.

Properly generating synchronous data requires that values generated for each cycle from different statements follow the exact same proper approach. Should the timing requirements or synchronization of the synchronous signal be modified, all statements driving that signal must be consistently updated. Using a *clocking* block decouples synchronization specification from functional specifica-

tion. The *clocking* block defines the active clock edge and the hold time of the synchronous signal. The code generating the stimulus only needs to worry about the consecutive values of the signal, as shown in Sample 5-24.

Sample 5-24. Using a *clocking* block to drive synchronous values.

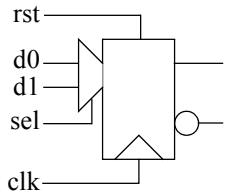
```
clocking cb @ (negedge tb_top.clk);
   output #1 data = tb_top.data;
endclocking: cb

initial
begin
   @(cb);
   cb.data <= ...;
   @(cb);
   cb.data <= ...;
   ...
end
```

Abstracting Waveform Generation

Input vectors are difficult to write and maintain.

Using synchronous test values, also known as test vectors, to verify a design is rather cumbersome. They are hard to interpret and difficult to specify correctly. For example, using cycle-by-cycle values to verify a synchronously resetable D flip-flop with a 2-to-1 multiplexer on the input, as shown in Figure 5-11, could be stimulated using the vectors shown in Sample 5-25.

Figure 5-11. 2-to-1 input sync reset D flip-flop



Use tasks to encapsulate operations.

It would be easier if the operation accomplished by the test vectors were abstracted. The device under verification can perform only one of three things:

- A synchronous reset,
- Load from input *d0,* or
- Load from input *d1*

Sample 5-25. Test vectors for 2-to-1 input sync reset D flip-flop

```
clocking cb @ (negedge tb_top.clk);
   output #1 data = {tb_top.rst,
                     tb_top.d0,
                     tb_top.d1,
                     tb_top.sel};
endclocking: cb

initial
begin
   // Input
   @(cb); cb.data <= 4'b1110;
   @(cb); cb.data <= 4'b0100;
   @(cb); cb.data <= 4'b1111;
   @(cb); cb.data <= 4'b0011;
   @(cb); cb.data <= 4'b0010;
   @(cb); cb.data <= 4'b0011;
   @(cb); cb.data <= 4'b1111;
   ...
end
```

Instead of providing test vectors to perform these operations repeatedly, why not provide subprograms that perform these operations? All that will be left is to call the subprograms in the appropriate order with the appropriate data.

Try to apply the worst possible combination of inputs.

The task to perform the synchronous reset is very simple. It needs to assert the *rst* input, then wait for the active edge of the clock. But what about the other inputs? You could decide to leave them unchanged, but is that the worst possible case? What if the reset was not functional and the device loaded one of the inputs and that input was set to 0? It would be impossible to differentiate the wrong behavior from the correct one. To create the worst possible condition, both *d0* and *d1* inputs must be set to 1. The *sel* input can be set randomly, since either input selection should be functionally identical. An implementation of the *reset* procedure is shown in Sample 5-26.

Sample 5-26. Abstracting the synchronous reset operation

```
task sync_reset;
begin
   cb.rst <= 1'b1;
   cb.d0  <= 1'b1;
   cb.d1  <= 1'b1;
   cb.sel <= $random;
   @(cb);
endtask: sync_reset
```

Pass input values as arguments to the subprogram.

The other operations this design can perform is to load input *d0* or *d1*. The *task* to perform the "load *d0*" operation is shown in Sample 5-27. Unlike resetting the design, loading data can have different input values: It can load either a one or a zero. The value of the input to load is passed as an argument to the task. The worst condition is created when the other input is set to the complement of the input value. If the device is not functioning properly and is loading from the wrong input, then the result will be clearly wrong.

Sample 5-27. Abstracting the load operation

```
task load_d0(input bit data);
   cb.rst <= 1'b0;
   cb.sel <= 1'b0;
   cb.d0  <=  data;
   cb.d1  <= ~data;
   @ (cb);
endtask: load_d0
```

Stimulus generated with abstracted operations is easier to write and maintain.

Once operation abstractions are available, providing the proper stimulus to the design under verification is easy to write and understand. Compare the code in Sample 5-28 with the code of Sample 5-25. If the polarity of the *rst* input were changed, which verification approach would be easiest to understand and modify?

Sample 5-28. Verifying the design using operation abstractions

```
initial
begin
   sync_reset;
   load_d0(1);
   sync_reset;
   load_d1(1);
   load_d0(0);
   load_d1(1);
   sync_reset;
   ...
end
```

SIMPLE OUTPUT

Generating stimulus is only half of the job. Actually, it is more like 25 percent of the job. The other parts, verifying that the output is as expected and collecting functional coverage measurements, is much more time consuming and error prone. There are various ways the output can be checked against expectations. The outputs have varying degrees of applicability and repeatability. In this sec-

tion, I will review techniques, some good, some not so good, for verifying simple responses.

Visual Inspection of Response

Results can be printed.

The most obvious method for verifying the output of a simulation is to inspect the results visually. The visual display can be an ASCII printout of the input and output values at specific points in time, as shown in Sample 5-29.

Sample 5-29. ASCII view of simulation results

```
      r  s
      sddeqq
Time  t01l b
------------
0100  1110xx
0105  111001
0200  010001
0205  010010
0300  111110
0305  111101
0400  001101
0405  001110
0500  001010
0505  001010
0600  001110
0605  001110
0700  111110
0705  111101
...
```

Producing Simulation Results

To print simulation results, you must model the signal sampling.

The specific points in time that are significant for a particular design or testbench are always different. Which signals are significant is also different and may change as the simulation progresses. If you know which time points and signals are significant for determining the correctness of the simulation results, you have to be able to model that knowledge. Producing the proper simulation results involves modeling the behavior of the signal sampling.

Many sampling techniques can be used.

There are many sampling techniques, each as valid as the other. The correct sampling technique depends on your needs and on what makes the simulation results significant. Just as you have to decide which input sequence is relevant for the functionality you are trying to verify, you must also decide on the output sampling that is rele-

vant for determining the success or failure of the function under verification.

You can sample at regular intervals.

The simplest sampling technique is to sample the relevant signals at a regular interval. The interval can be an absolute delay value, as illustrated in Sample 5-30, or the interval can be a reference signal such as the clock, as illustrated in Sample 5-31. Note how the *$strobe* statement is used instead of *$write* or *$display*. This ensures that the displayed values are the final, stable values for the current simulation cycle and not some intermediate transient value..

Sample 5-30. Sampling at a delay interval

```
parameter INTERVAL = 10;
always
begin
   #(INTERVAL);
   $strobe(...);
end
```

Sample 5-31. Sampling based on a reference signal

```
always @(negedge clk)
begin
   $strobe(...)
end
```

You can sample based on a signal changing value.

Another popular sampling technique is to sample a set of signals whenever one of them changes. This technique is used to reduce the amount of data produced during a simulation when signals do not change at a constant interval.

To sample a set of signals, simply make an *always* block sensitive to the signals whose changes are significant, as shown in Sample 5-32. The set of signals displayed and monitored can be different. SystemVerilog has a built-in task, called *$monitor*, to perform this sampling when the set of displayed and monitored signals are identical.

An example of using the *$monitor* task is shown in Sample 5-33. The behavior of the *$monitor* statement in Sample 5-33 is different than the *always* block in Sample 5-32: the former will display on any change of the *rst*, *d0*, *d1*, *sel*, *q* or *qb* signals, whereas the latter will only display on changes of the *q* or *qb* signals. Note that simu-

lations are limited to a single active *$monitor* task. Any subsequent call to *$monitor* replaces the previous monitor.

Sample 5-32. Sampling based on signal changes

```
always @(q, qb)
begin
   $strobe("...", rst, d0, d1, sel, q, qb);
end
```

Sample 5-33. Sampling using the *$monitor* task

```
initial
begin
   $monitor("...", rst, d0, d1, sel, q, qb);
end
```

Minimizing Sampling

To improve simulation performance, minimize sampling.

The use of an output device on a computer slows down the execution of any program. Therefore, recording simulation output reduces the performance of the simulation. To maximize the speed of a simulation, minimize the amount of simulation output produced during its execution.

An active *$monitor* task can be turned on and off by using the *$monitoron* and *$monitoroff* tasks, respectively. If you are using an explicit sampling *always* block, you should include sampling minimization techniques in your model, as illustrated in Sample 5-34. A very efficient way of minimizing sampling is to have the stimulus turn on the sampling when an interesting section of the testcase is entered, as shown in Sample 5-35.

Sample 5-34. Minimizing sampling

```
always
begin
   wait (<interesting_condition>);
   while (<interesting_condition>) begin
      @ (q, qb;)
      $strobe("...", rst, d0, d1, sel, q, qb);
   end
end
```

Sample 5-35. Controlling the sampling from the stimulus

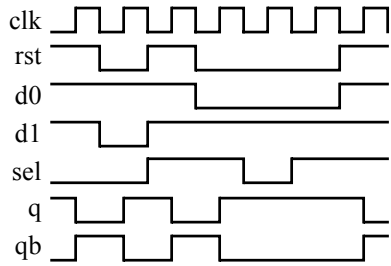
```
initial
begin
   $monitor("...", rst, d0, d1, sel, q, qb);
   $monitoroff;
   sync_reset;
   load_d0(1);
   sync_reset;
   $monitoron;
   load_d1(1);
   load_d0(0);
   load_d1(1);
   sync_reset;
   $monitoroff;
   ...
end
```

Visual Inspection of Waveforms

Results are better viewed when plotted over time.

Waveform displays usually provide a more intuitive visual representation of simulation results. Figure 5-12 shows the same information as Sample 5-29, but using a waveform view. The waveform view has the advantage of providing a continuous display of many values over the entire simulation time, not just at specific time points as in a text view. Therefore, you need not specify or model a particular sampling technique. The signals are continuously sampled, usually into an efficient database format. Sampling for waveforms must be turned on explicitly. It is a tool-dependent[1] process that is different for each tool.

Figure 5-12. Waveform view of simulation results



1. SystemVerilog has a standard waveform database called the VCD file. Although all waveform viewers can display simulation results from a VCD file, all of the more advanced viewers use their own proprietary database to store additional signal information.

Minimize the number and duration of sampled signals.

The default behavior is to sample all signals during the entire simulation. The waveform sampling process consumes a significant portion of the simulation resources. Reducing the number of signals sampled, or reducing the duration of the sampling, increases the simulation performance. However, it is a trade-off with running a simulation multiple times to obtain traces of signals that were found to be necessary for diagnosing the cause of a functional error in a previous iteration. During a typical verification process, all signals should be sampled at the beginning, when the number of bugs is significant and their location is unknown. As the code stabilizes and simulations move to greater levels of integration, less and less signals are sampled. During regression runs, no signals are sampled. The rule of thumb is: If you expect the simulation to fail, sample a lot of signals; if you expect it to pass, don't sample any.

Self-Checking Testbenches

Visual inspection is not acceptable.

The model of the D flip-flop with a 2-to-1 input mux being verified has a functional error. Can you identify it using either views of the simulation results in Sample 5-29 or Figure 5-12? How long did it take to diagnose the problem?[1]

Code the response with the stimulus.

This example was for a very simple design, over a very short period of time, and for a very small number of signals (and you knew there was a bug). Imagine visually inspecting simulation results spanning hundreds of thousands of clock cycles, and involving hundreds of input and output signals. Then imagine repeating this visual inspection for every testbench and for every simulation of every testbench. The probability that you will miss identifying an error is equal to one hundred percent. You must automate the process of comparing the simulation results against the expected outputs.

Input and Output Vectors

Specify the expected output values for each clock cycle.

The first step in automating output verification is to include the expected output with the input stimulus for every clock cycle. The vector application task in Sample 5-24 can be easily modified to include the comparison of the output signals with the specified out-

1. The logic value on input *d0* is ignored and a 1 is always loaded.

put vector, as shown in Sample 5-36. The testcase becomes a series of input/output test vectors, as shown in Sample 5-37.

Sample 5-36. Application of input and verification of output data vectors

```
task apply_vector(input [...] in_data,
                  input [...] out_data);
   cb.in_data <= in_data;
   @(cb);
   fork
      begin
         @(cb)
         if (cb.out_data !== out_data) ...;
      end
   join_none
endtask: apply_vector
```

Sample 5-37. Input/output test vectors for 2-to-1 input sync reset D flip-flop

```
initial
begin
   // In: rst, d0, d1, sel
   // Out: q, qb
   apply_vector(4'b1110, 2'b00);
   apply_vector(4'b0100, 2'b10);
   apply_vector(4'b1111, 2'b00);
   apply_vector(4'b0011, 2'b10);
   apply_vector(4'b0010, 2'b01);
   apply_vector(4'b0011, 2'b10);
   apply_vector(4'b1111, 2'b00);
   ...
end
```

Test vectors require synchronous interfaces.

The main problem with input and output test vectors (other than the fact that they are very difficult to specify, maintain and debug), is that they require perfectly synchronous interfaces. If the design under verification contains interfaces in different clock domains, each interface requires its own test vector stream. If any interface contains asynchronous signals, the signals have to be either externally synchronized before vectors are applied, or treated as synchronous signals, therefore under-constraining the verification.

Golden Vectors

A set of reference simulation results can be used.

The next step toward automation of response verification is the use of golden vectors. It is a simple extension of the manufacturing test process where devices are physically subjected to a series of qualifying test vectors. A set of reference output results, determined to be correct, are kept in a file or database. The simulation outputs are

captured in a similar format during a simulation. They are then compared against the reference results. Golden vectors have an advantage over input/output vectors because the expected output values need not be specified in advance.

Text files can be compared using *diff*.

If the simulation results are kept in ASCII files, the simplest comparison process involves using the UNIX *diff* utility. The *diff* output for the simulation results shown in Sample 5-29 is shown in Sample 5-38. You can appreciate how difficult the subsequent task of diagnosing the functional error will be.

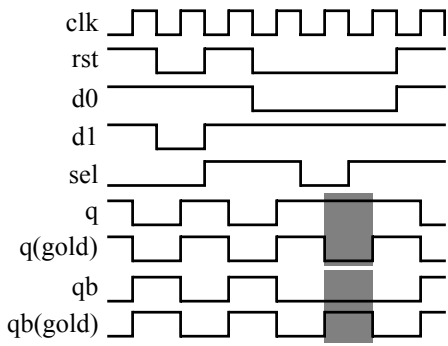
Sample 5-38. *diff* output of comparing ASCII view of simulation results

```
14c2
>0505 001010
>0600 001110
--------
<0505 001001
<0600 001110
...
```

Waveforms can be compared by a specialized tool.

Waveform comparators can be used also. They are tools similar to waveform viewers and are usually built into one. Waveform comparators compare two sets of waveforms then highlight the differences on a graphical display. The display of a waveform comparator might look something like the results illustrated in Figure 5-13. Identifying the problem is easier since you have access to the entire history of the simulation in a single view.

Figure 5-13. Waveform differences in simulation results



Golden vectors must still be inspected visually.

The main problem with golden simulation results is that they need to be inspected visually to be determined as "golden". This self-checking technique only reduces the number of times a set of simulation responses must be verified visually, not the need for visual inspection. The result from each testbench must *still* be manually confirmed as good.

Golden vectors do not adapt to changes.

Another problem: Reference simulation results do not adapt to modifications in the design under verification that may only affect the timing of the result, without affecting its functional correctness. For example, an extra register may be added in the datapath of a design to help meet timing constraints. All that was added was a pipeline delay. The functionality was not modified. Only the latency was increased. If that latency is irrelevant to the functional correctness of the overall system, the reference vectors must be updated to reflect that change.

Golden vectors require a significant maintenance effort.

Reference simulation results must be inspected visually for every testcase, and modified or regenerated whenever a change is made to the design, each time requiring visual inspection. Using reference vectors is a high-maintenance, low-efficiency self-checking strategy. Verification vectors should be used *only* when a design must be 100 percent backward compatible with an existing device, signal for signal, clock cycle for clock cycle. In those circumstances, the reference vectors never change and never require visual inspection as they are golden by definition.

Separate the reference vectors along clock domains.

Reference simulation results also work best with synchronous interfaces. If you have multiple interfaces in separate clock domains, it is necessary to generate reference results for each domain in a separate file. If a single file is used, the asynchronous relationship between the clock domains may result in the samples from different domains being written in a different order. The ordering difference is not functionally relevant, but would be flagged as an error by the comparison tool.

Self-Checking Operations

For simple operations on simple devices, it may be possible to verify the response on an operation-by-operation basis. For example, the task shown in Sample 5-26 can include the verification that the flip-flop was reset properly as shown in Sample 5-39. Similarly, the task used to apply the stimulus to load data from *d0* shown in Sam-

ple 5-27 can be modified to include the verification of the output, as shown in Sample 5-40. The testcase shown in Sample 5-28 now becomes entirely self-checking.

Sample 5-39. Verifying the sync reset operation

```
task sync_reset;
begin
   cb.rst <= 1'b1;
   cb.d0  <= 1'b1;
   cb.d1  <= 1'b1;
   cb.sel <= $random;
   @(cb);
   fork
      begin
         @(cb);
         if (cb.q  !== 1'b0 ||
             cb.qb !== 1'b1) ...
      end
   join_none
endtask: sync_reset
```

Sample 5-40. Verifying the load operation

```
task load_d0(input data);
   cb.rst <= 1'b0;
   cb.sel <= 1'b0;
   cb.d0  <=  data;
   cb.d1  <= ~data;
   @ (cb);
   fork
      begin
         @(cb);
         if (cb.q  !== data ||
             cb.qb !== ~data) ...
      end
   join_none
endtask: load_d0
```

Make sure the output is verified properly.

The problem with output verification is that you can't identify a functional discrepancy if you are not looking at it. Using an *if* statement to verify the output in the middle of a stimulus thread only looks at the output value for a brief instant. That may be acceptable, but this technique does not say anything about the *stability* of that output. For example, the tasks in Sample 5-39 and Sample 5-40 only check the value of the output at a single point in time.

Figure 5-14 shows the complete specification for the flip-flop. The verification sampling point is shown as well.

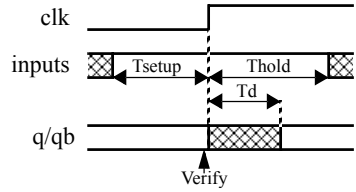


Figure 5-14. Timing specification for the flip-flop

Make sure you verify the output over the entire significant time period.

To verify the functionality of the design properly and completely, it is necessary to verify that the output is stable, except for the short period after the rising edge of the clock. That could be verified easily using a static timing analysis tool and a set of suitable constraints to verify against. If you want to perform the verification as part of a functional simulation, the stability of the output cannot be verified from the same task that applies the input. Stability is a property that must be checked at all times, not just after applying new stimulus. Therefore, a concurrent stability check thread must exist, independent of the stimulus thread, to be ready to react to any unexpected changes, as shown in Sample 5-41. The stimulus task still checks the correctness of the output value. The stability monitor simply verifies that the output remains stable, whatever its value.

Low-level checks may have to be located in the design.

Notice how the stability check in Sample 5-41 is located in the flip-flop design *module* itself, not the testbench *program*. The preference in the simulation cycle for module threads over program threads would filter out any transient output value made from a module thread. These transient values may not be visible to the program threads if they eventually resolve to the same initial value. The check could have been implemented in a separate module, using cross-module (white-box) references to observe the appropriate signals within the design module, but it would have required one such module for every instance of the design module. Locating the check inside the design module simplifies overall maintenance and it can be surrounded by appropriate directives to eliminate it from synthesis.

A *property* could not have been asserted.

Properties are cycle-based sequences of boolean expressions. The stability check requires an asynchronous, self-timed expression. Note that the stability check *initial* block is an assertion. An asser-

Sample 5-41. Verifying the stability of flip-flop outputs

```
module muxff(...);
...
`ifndef SYNTHESIS
initial
begin
   // wait for the first clock edge
   @ (posedge clk);
   forever begin
      // Ignore changes for Td after clock edge
      #(Td);
      // Watch for a change before the next clk
      fork: stability_mon
         @ (q or qb) $write("...");
         @ (posedge clk);
      join_any
      disable stability_mon;
   end
end
`endif
endmodule
```

tion is simply a check for a property to be true. It does not need to make use of the *property* construct. Some assertions are better implemented using behavioral code.

COMPLEX STIMULUS

This section introduces more complex stimulus generation scenarios through the use of bus-functional models. I start with reactive stimulus, where the stimulus or its timing depends on answers from the device under verification. I also show how to avoid wasting precious simulation cycles by getting caught in deadlock conditions.

Generating inputs may require cooperating with the design.

Applying stimulus to a clock or reset input or applying cycle-by-cycle test vectors is straightforward. You are under complete control of the timing of the input signal. However, if the interface being driven contains handshaking or flow-control signals, the generation of the stimulus requires cooperation with the design under verification.

Feedback Between Stimulus and Design

Without feedback, verification can be under-constrained.

Figure 5-15 shows the specification for a simple bus arbiter. If you were to verify the design of the arbiter using test vectors applied at every clock cycle, as described in “Input and Output Vectors” on page 221, you would have to assume a specific delay between the assertion of the *req* signal and the assertion of the *grt* signal. Any delay value between one and five clock cycles would be functionally correct, but the only reliable choice is a delay of five cycles. Similarly, a delay of three clock cycles would have to be made for the release portion of the verification. These choices, however, severely under-constrain the verification. If you want to stress the arbiter by issuing requests as fast as possible, you would want to know when the request was granted and released, so it could be reapplied as quickly as possible.

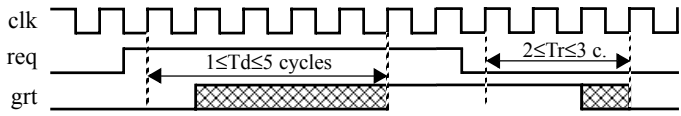


Figure 5-15. Specification for a simple arbiter

Stimulus generation can wait for feedback before proceeding.

If, instead of using input and output test vectors, you are using encapsulated operations to verify the design, you can modify the operation to wait for feedback from the design under verification before proceeding. You should also include any timing and functional verification in the feedback monitoring to ensure that the design responds in an appropriate manner. Sample 5-42 shows the *bus_request* operation task. The task samples the *grt* signal at every clock cycle, and immediately returns once it detects that the bus was granted. With a similarly implemented *bus_release* task, a testcase that stresses the arbiter under maximum load can be written easily, as shown in Sample 5-43.

Recovering from Deadlocks

A deadlock may prevent the testcase from running to completion.

There is a risk inherent to using feedback in generating stimulus: The stimulus now depends on the proper operation of the design under verification to complete. If the design does not provide the feedback as expected, the stimulus generation may be halted, waiting for a condition that will never occur. For example, consider the *bus_request* task in Sample 5-42. What happens if the *grt* signal is

Sample 5-42. Verifying the bus request operation

```
program test;

clocking cb @(posedge tb_top.clk);
   output req = tb_top.req;
   input  grt = tb_top.grt;
endclocking: cb

task bus_request;
   automatic int cycle_count = 0;
   cb.req <= 1'b1;
   while (cb.grt != 1'b1) begin
      @(cb);
      cycle_count++;
   end
   assert 1 <= cycle_count && cycle_count <= 5;
end: bus_request
...
endprogram: test
```

Sample 5-43. Stressing the bus arbiter

```
program test;
...
initial
begin
   repeat (10) begin
      bus_request;
      bus_release;
   end
end
endprogram: test
```

never asserted? The task remains stuck in the *while* loop and never returns.

A deadlocked simulation appears to be running correctly.

If this were to occur, the simulation would still be running, merrily going around and around the *while* loop. The simulation time would advance at each tick of the clock. The CPU usage of your workstation would show near 100 percent usage. The only symptom that something is wrong would be that no messages are produced on the simulation's output log and the simulation runs for much longer than usual. If you are watching the simulation run and expect regular messages to be produced during its execution, you would quickly recognize that something is wrong and manually interrupt it.

A deadlocked simulation wastes regression runs.

But what if there is no one watching the simulation, such as during a regression run? Regressions are large scale simulation runs where all available testcases are executed. They are used to verify that the functionality of the design under verification is still correct after modifications. Because of the large number of testcases involved in a regression, the process is automated to run unattended, usually overnight and on many computers. If a design modification creates a deadlock situation, all testcases scheduled to execute subsequently will never run, as the deadlocked testcase never terminates. The opportunity of detecting other problems in the regression run is wasted. It will be necessary to wait for another 24-hour period before knowing if the new version of the design meets its functional specification.

Eliminate the possibility of deadlock conditions.

When generating stimulus, you must make sure that there is no possibility of a deadlock condition. You must assume that the feedback condition you are waiting for may never occur. If the feedback condition fails to happen, you must then take appropriate action. It could include terminating the testcase, or jumping to the next portion of the testcase that does not depend on the current operation, or retrying the operation after some delay. Sample 5-42 was modified as shown in Sample 5-44 to use an assertion to avoid the deadlock condition created if the arbiter failed and the *grt* signal was never asserted.

Sample 5-44. Avoiding deadlock in the bus request operation

```
program test;

clocking cb @(posedge tb_top.clk);
   output req = tb_top.req;
   input  grt = tb_top.grt;
endclocking: cb

property grt_within_5;
   $rose(tb_top.req)
      |-> ##[1:5] $rose(tb_top.ack);
endproperty
assert grt_within_5 @(posedge tb_top.clk);

task bus_request;
   cb.req <= 1'b1;
   while (cb.grt != 1'b1) @(cb);
end: bus_request
...
endprogram: test
```

Operation tasks could return status.

If a failure of the feedback condition is detected, terminating the simulation on the spot, as shown in Sample 5-44, is easy to implement. If you want more flexibility in handling a non-fatal error, you might want to let the testcase handle the error recovery, instead of handling it inside the operation task. The task must provide an indication of the status of the operation's completion back to the testcase. Sample 5-45 shows the *bus_request* task that includes an *OK* status flag indicating whether the bus was granted. The testcase is then free to retry the bus request operation until it succeeds, as shown in Sample 5-46. Notice how the testcase takes care of avoiding its own deadlock condition if the bus request operation never succeeds.

Sample 5-45. Returning status in the bus request operation

```
program test;

clocking cb @(posedge tb_top.clk);
   output req = tb_top.req;
   input  grt = tb_top.grt;
endclocking: cb

task bus_request(output bit ok);
   automatic int cycle_count = 0;
   ok = 0;
   cb.req <= 1'b1;
   while (cb.grt != 1'b1) begin
      @(cb);
      if (cycle_count++ > 100) begin
         cb.req <= 1'b0;
         return;
      end
   end
   assert 1 <= cycle_count && cycle_count <= 5;
   ok = 1;
end: bus_request
...
endprogram: test
```

Asynchronous Interfaces

Test vectors under-constrain asynchronous interfaces.

Creating synchronous input data and verifying synchronous output values is simple. The inputs are all applied at the same time. The outputs are all verified at the same time. And this process is repeated at regular intervals. In every design, there is some reference signal that can be used to synchronize generation and sampling operations. But many interfaces, although implemented using

Sample 5-46. Handling failures in the *bus_request* task

```
program test;
...
initial
begin
   bit ok;
   int attempts = 0;

   forever begin
      bus_request(ok);
      if (ok) break;
      attempts++;
      assert attempts < 5;
   end
   ...
end
endprogram: test
```

synchronous finite state machines and edge-triggered flip-flops, are *specified* in an asynchronous fashion. The implementer has arbitrarily chosen a clock to streamline the physical implementation of the interface. If that clock is not part of the specification, it should not be part of the verification. For example, Figure 5-16 shows an asynchronous specification for a bus arbiter. Given a suitable clock frequency, the synchronous specification shown in Figure 5-15 would be functionally equivalent.

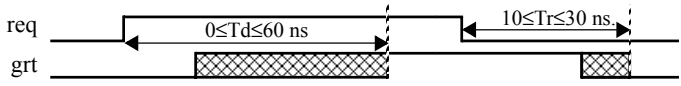


Figure 5-16. Asynchronous specification for a simple arbiter

Verify the synchronous implementation against the asynchronous specification.

Even though a clock may be present in the implementation, if it is not part of the specification, you cannot use it to generate stimulus nor to verify the response. You would be verifying against a particular implementation, not the specification. If a clock is present, and the timing constraints make reference to clock edges, then you *must* use it to generate stimulus and verify response. For example, a PCI bus is synchronous. A verification of a PCI interface must use the PCI system clock to verify any implementation.

High-level code does not require a clock like RTL code.

Testbenches are written using high-level code. Transaction-level models do not require a clock. A clock is an artifice of the implementation methodology and is required only for RTL code. The bus

request phase of the asynchronous interface specified in Figure 5-16 can be verified asynchronously with the *bus_request* task shown in Sample 5-47. Notice how the bus request operation does not use a clock for timing control. Also, notice how it uses the definitely non-synthesizeable *fork*/*join* statement to wait for the rising edge of *grt* for a maximum of 60 nanoseconds.

Sample 5-47. Verifying the asynchronous bus request operation

```
task bus_request(output bit ok);
   req = 1'b1;
   fork: wait_for_grt
      #60ns;
      @ (posedge grt);
   join_any
   disable wait_for_grt;
   ok = (grt == 1'b1);
endtask: bus_request
```

Consider all possible failure modes.

There is one problem with the bus request operation in Sample 5-47. What if the arbiter was functionally incorrect and left the *grt* signal always asserted? Both models would never see a rising edge on the *grt* signal. They would eventually exhaust their maximum waiting period then detect *grt* as asserted, indicating a successful completion. To detect this possible failure mode, the bus request operation must verify that the *grt* signal is not asserted prior to asserting the *req* signal, as shown in Sample 5-48.

Sample 5-48. Verifying all failure modes in the asynchronous bus request operation

```
task bus_request(output bit ok);
   if (grt == 1'b1) begin
      ok = 0;
      return;
   end
   req = 1'b1;
   fork: wait_for_grt
      #60ns;
      @ (posedge grt);
   join_any
   disable wait_for_grt;
   ok = (grt == 1'b1);
endtask: bus_request
```

Were you paying attention?

Pop quiz: what is missing from all those task implementations?[1]

BUS-FUNCTIONAL MODELS

Operations are abstracted through bus-functional models.

Although I have avoided using the term *bus-functional model*, all of the tasks abstracting operations on the design shown earlier are bus-functional models, albeit very simple ones. Operations, also known as *transactions*, encapsulated using *tasks* can be very complex. The examples shown earlier were very simple and dealt with only a few signals. Real-life interfaces are more complex. But they can be encapsulated just as easily. These transactions may even return values to be verified against expected response or modify the stimulus sequence. As shown in Figure 4-2, a bus-functional model abstracts transactions on a physical-level interface into a procedural interface. Bus-functional models can be used to generate stimulus as well as monitor the response of a design. Very often, a single bus-functional model performs both operations.

CPU Transactions

CPU interfaces are popular bus functional models.

The first image that probably came to your mind when you read the term "bus-functional model" was an interface to a processor. Abstracted processor bus transactions are the most popular and common bus-functional models. Figure 5-17 shows the specification for the write cycle for an Intel 386SX processor bus. Sample 5-49 shows the corresponding bus-functional model procedure.

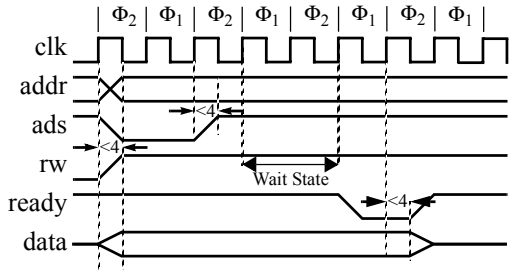


Figure 5-17. Specification for the write cycle of a 386sx processor

1. They all include timing control statements. They should have a semaphore to detect concurrent activation. See "Non-Re-Entrant Tasks" on page 188.

Sample 5-49. Model for the write cycle operation

```
task write_cycle(input bit [23:0] wadd,
                 input bit [31:0] wdat);
   do @ (cb) while (cb.phi != 2);
   cb.addr <= wadd;
   cb.ads  <= 1'b0;
   cb.rw   <= 1'b1;
   cb.data <= wdat;
   repeat (2) @ (cb);
   cb.ads  <= 1'b1;
   do @(cb) while (cb.phi != 2 ||
                   cb.ready != '0');
   cb.data <= 'z;
endtask: write_cycle
```

Bus models can adapt to a different number of wait states.

To generate stimulus for this interface using synchronous test vectors, you would have to assume a specific number of wait cycles to complete the write operation at the right time. With high-level models of the transaction, you need not enforce a particular number of wait cycles and adapt to any valid bus timing. In Sample 5-49, the wait cycles are introduced by the timing control statement inside the *do* loop.

Bus-functional procedures can return values.

All of the abstracted transactions shown so far were unidirectional. Data always flowed from the testbench through the bus-functional task where the data was applied to the design and outputs were checked for correctness. What if determining the correctness of the output required visibility over multiple operations? What if only the relevant output values for this testcase were known and the others were to be ignored? Bus-functional tasks can just as easily sample output and return it instead of comparing the output against supplied expected responses. The sampled value can then be processed by the testbench where the value can be dealt with according to the needs of the testcase. For example, Sample 5-50 shows the *read* operation of the 386SX interface. Notice how the value read is not compared against an expected value. The value read is instead returned through an *output* argument.

You can perform read-modify-write operations.

It now becomes easy to perform read-modify-write operations. With abstracted transactions and the full power of a high-level language, you can perform a *read* operation that returns whatever value was read at the specified address, manipulate the read value, then use the modified value in a subsequent write transaction. Sample 5-51 shows a portion of a testcase where the *read_cycle* and

Sample 5-50. Model for the read cycle operation

```
task read_cycle(input  bit [23:0] radd,
                output bit [31:0] rdat);
   do @(cb) while (phi != 2);
   cb.addr <= radd;
   cb.ads  <= 1'b0;
   cb.rw   <= 1'b0;
   repeat (2) @ (cb);
   cb.ads  <= 1'b1;
   do @(cb) while (cb.phi != 2 ||
                   cb.ready != 1'b0);
   rdat = cb.data;
endtask: read_cycle
```

write_cycle tasks are used to perform a read-modify-write operation.

Sample 5-51. Performing a read-modify-write operation

```
program test;
...
initial
begin
   const bit [23:0] cfg_reg = 24'h000316;
   bit [31:0] tmp;
   ...
   read_cycle(cfg_reg, tmp);
   tmp(13:9) := "01101";
   write_cycle(cfg_reg, tmp);
   ...
end
endprogram: test
```

From Bus-Functional Tasks to Bus-Functional Model

Bus-functional tasks are packaged into bus-functional models.

A complete bus-functional model is composed of many bus-functional tasks. Each transaction supported by a particular physical interface is implemented using a different procedure. Collected together in a *class* as described in "Encapsulating Bus-Functional Models" on page 127, they create a complete bus-functional model for a specific interface.

Tasks may be re-entrant, but bus-functional models are not.

In "Non-Re-Entrant Tasks" on page 188, I discussed the problem caused by non-re-entrant tasks. By encapsulating the transaction procedures in a class, which is dynamic, making the tasks re-entrant by default, you'd think that the problem would be solved, right? Wrong. Sample 5-52 shows the bus-functional tasks for the i386SX packed into a class. Although the *read* and *write* tasks are now fully

re-entrant, what happens if two separate threads concurrently invoke the same task? The local data space of the task is preserved, but not the value of the (static) interface signals. The two concurrent transactions will interfere with each other trying to execute at the same time on the same physical interface. The same problem will occur even if two different bus-functional tasks in the same bus-functional model are concurrently invoked.

Sample 5-52. Packaged i386SX bus-functional model

```
class i386sx;
   virtual i386sx_if sigs;
   ...
   virtual task read(input  bit [23:0] radd,
                     output bit [31:0] rdat);
      this.sigs.cb.addr <= radd;
      ...
   endtask: read

   virtual task write(input bit [23:0] wdd,
                      input bit [31:0] wdat);
      this.sigs.cb.addr <= wadd;
      ...
   endtask: write
endclass: i386sx
```

Put a semaphore on the bus-functional model.

If two or more threads must read from (or write to) the design, the operations must be coordinated. To pipeline concurrent operations, it is necessary to put a semaphore around the *entire* bus-functional model. Much like a semaphore was used to detect concurrent invocation of a non-re-entrant task, it will be used to detect concurrent invocation of transactions in a non-re-entrant bus-functional model. Sample 5-53 shows how a bus-functional model can be protected against concurrent transactions using a semaphore. It is up to you to decide, should the semaphore detect a concurrent transaction, whether to wait for the bus-functional model to become available or to terminate with an error.

SystemVerilog is not significantly better for physical-level bus-functional models.

With almost all of the examples in the previous sections looking like pure Verilog, I would not be surprised if you double-checked the title of this book to make sure it says “SystemVerilog”. SystemVerilog is not significantly better than good old Verilog in implementing physical-level bus-functional models. Low-level bus-functional models simply translate an abstracted representation of a transaction into 1’s and 0’s applied or sampled at individual clock cycles. Signal assignments, signal sampling and waiting for the

Sample 5-53. Protected i386SX bus-functional model

```
class i386sx;
   local semaphore sem;
   virtual i386sx_if sigs;
   ...
   virtual task read(input  bit [23:0] radd,
                     output bit [31:0] rdat);
      if (!this.sem.try_get(1)) ...;
      this.sigs.cb.addr <= radd;
      ...
      this.sem.put(1);
   endtask: read

   virtual task write(input bit [23:0] wdd,
                      input bit [31:0] wdat);
      if (!this.sem.try_get(1)) ...;
      this.sigs.cb.addr <= wadd;
      ...
      this.sem.put(1);
   endtask: write
endclass: i386sx
```

next clock edge uses the same statements as in pure Verilog because it does not need the increased levels of abstraction offered by the features that make SystemVerilog.

SystemVerilog is significantly better above the physical level.

SystemVerilog becomes clearly superior once we stop dealing with the physical interface because of its support for high-level data types, object-orientedness and randomization. SystemVerilog allows for a simpler transaction-layer interface. The transaction descriptors will be easier to model and manipulate using object-oriented methods and transaction descriptors will be easier to generate using constrainable randomization.

Physical Interfaces

Collect all signals in an *interface.*

Bus-functional models encapsulated in classes can access physical signals in one of two ways: through hierarchical—white-box—references or through a *virtual interface*. Using hierarchical references would make the bus-functional model class specific to a particular set of interface signals. It would not be possible to reuse the model in a different testbench or instantiate it more than once in the same testbench without copying it and modifying the references. The only mechanism that will make bus-functional models reusable within or across testbenches is the *virtual interface*. This implies

that all physical-level signals required by the bus-functional model be encapsulated in an *interface* as shown in Sample 5-54.

Sample 5-54. Physical interface for MII bus-functional model

```
interface mii_mac_if;

wire       tx_clk;
reg  [3:0] txd;
reg        tx_en;
reg        tx_er;
wire       rx_clk;
wire [3:0] rxd;
wire       rx_dv;
wire       rx_er;
wire       crs;
wire       col;
...
endinterface: mii_mac_if
```

Define *clocking* blocks for each clocking domain.

Synchronous signals are better sampled and driven through *clocking* blocks. It simplifies the maintenance of the synchronization and delay specifications and properly samples synchronous data from the module domain to the program domain. Each clock domain requires a separate clocking block, as shown in Sample 5-55. Note how the asynchronous signals *crs* and *col* are not included in any *clocking* blocks.

Sample 5-55. Clocking domains in physical interface.

```
interface mii_mac_if;
...
clocking tx @(posedge tx_clk);
   output #1 txd, tx_en, tx_er;
endclocking: tx

clocking rx @(posedge rx_clk);
   input #1 rxd, rx_dv, rx_er;
endclocking: rx

endinterface: mii_mac_if
```

Can use a single *interface* for all perspectives on a physical interface.

The interface definitions in Sample 5-54 and Sample 5-55 imply a definite direction to the signals in that interface. Data is transmitted on the *tx...* signals and received on the *rx...* signals. That is correct if the bus-functional model implements the MAC-layer functionality in the ethernet protocol. If the bus-functional model were to implement the PHY-layer functionality, the data flow would need to be reversed. In reality, you often need both bus-functional models,

because the system you are designing has components on either side of that physical interface, and both components need to be independently verified. You'll also see how bus-functional models are useful for writing transaction-level models in Chapter 7, which requires the complementary flavor of bus-functional models. Because both bus-functional model flavors are usually needed, you have the choice of implementing two *interface* declarations—one for each flavor—or a single *interface* declaration that supports both flavors.

Define all signals as *wire* or *logic*.

If an interface is declared to be agnostic to the perspective or personality a bus-functional model can have on its signals, then it must not imply any directionality in the signals themselves. The *interface* becomes a collection of *wires* that are used to exchange information between a bus-functional model and a design or between two bus-functional models. Sample 5-56 shows the same signals as in Sample 5-54, but this time without any implied directions.

Sample 5-56. Physical interface signals

```
interface mii_if;

wire       tx_clk;
wire [3:0] txd;
wire       tx_en;
wire       tx_er;
wire       rx_clk;
wire [3:0] rxd;
wire       rx_dv;
wire       rx_er;
wire       crs;
wire       col;
...
endinterface: mii_if
```

Synchronous signals in *clocking* blocks are defined as *inout.*

clocking blocks also indicate directionality. To allow the *clocking* blocks to be used by any bus-functional model, regardless of its perspective on the physical interface, the synchronous signals in each clock domain must be defined as *inout*, as shown in Sample 5-57, compared to Sample 5-55.

Don't wait for clock edges.

It is only natural to use a Verilog coding style when coding using SystemVerilog. But this style can create problems. Consider the MII MAC-layer bus-functional model in Sample 5-58. This is obviously a model for a synchronous interface active on the negative edge of the clock, right? Wrong. Timing synchronization is speci-

Sample 5-57. Bidirectional clocking domains in physical interface.

```
interface mii_if;
...
clocking tx @(posedge tx_clk);
   input #1 output #1 txd, tx_en, tx_er;
endclocking: tx

clocking rx @(posedge rx_clk);
   input #1 output #1 rxd, rx_dv, rx_er;
endclocking: rx

endinterface: mii_if
```

fied in the *clocking* blocks of the *interface* declaration, not by the sequential statements. Should the synchronization be changed, only the *clocking* blocks should need changing. This style may also use wrong input values because of the module vs program thread simulation cycles. A better style, shown in Sample 5-59, lets the *clocking* block synchronization mechanism define the timing of the transaction.

Sample 5-58. Verilog coding style in SystemVerilog

```
class mii_mac_bfm;
   virtual mii_if sigs;
   ...
   virtual task rx(output eth_frame frame);
      ...
      do @ (posedge this.sigs.rx_clk)
         while (this.sigs.rx_dv != 1);
      do @ (posedge this.sigs.rx_clk)
         while (this.sigs.rx_dv == 1 &&
                this.sigs.rxd   == 4'b0101);
      if (this.sigs.rxd != 4'b0111) ...
      while (this.sigs.rx_dv == 1) begin
         @ (posedge this.sigs.rx_clk);
         byte[7:4] = this.sigs.rxd;
         ...
         @ (posedge this.sigs.rx_clk);
         byte[3:0] = this.sigs.rxd;
         ...
      end
      ...
   endtask: rx
endclass: mii_mac_bfm
```

Specify *virtual interface* to connect to via the constructor.

All class-encapsulated bus-functional model examples so far had a *virtual interface* data member that was used to access the physical interface of the bus-functional model. This data member must be

Sample 5-59. Proper coding style in SystemVerilog

```
class mii_mac_bfm
   virtual mii_if sigs;
   ...
   virtual task rx(output eth_frame frame);
      ...
      do @ (this.sigs.rx)
         while (this.sigs.rx_dv != 1);
      do @ (this.sigs.rx)
         while (this.sigs.rx_dv == 1 &&
                this.sigs.rxd   == 4'b0101);
      if (this.sigs.rxd != 4'b0111) ...
      while (this.sigs.rx_dv == 1) begin
         @ (this.sigs.rx);
         byte[7:4] = this.sigs.rxd;
         ...
         @ (this.sigs.rx);
         byte[3:0] = this.sigs.rxd;
         ...
      end
      ...
   endtask: rx
endclass: mii_mac_bfm
```

set somehow. Module pins are connected when the module is instantiated. This way, the same module may be used more than once but connected to different signals. Similarly, a virtual interface in a bus-functional model is connected when the bus-functional model is instantiated. Because the bus-functional model is encapsulated in a *class*, it is instantiated when its constructor is invoked. Therefore, the *virtual interface* connection is specified as a constructor argument, as shown in Sample 5-60. The virtual interface is connected when a new instance of the bus-functional model is created by calling the constructor and specifying an *interface* instance it is bound to via a cross-module reference, as shown in Sample 5-61.

Sample 5-60. Virtual interface as constructor argument.

```
class mii_mac_bfm;
   virtual mii_if sigs;
   ...
   function new(virtual mii_if sigs);
      this.sigs = sigs;
   endfunction: new
endclass: mii_mac_bfm
```

Sample 5-61. Binding a virtual interface in a class instance.

```
module tb_top;
mii_if if0();
...
endmodule

program test;
mii_mac_bfm mac = new(tb_top.if0);
...
endprogram: test
```

See the VMM.

The section titled “Signal Layer” on page 107 of the *Verification Methology Manual for SystemVerilog* provides additional guidelines and techniques for implementing and encapsulating physical interfaces.

Configurable Bus-Functional Models

Protocols can have configurable elements.

A protocol specification may contain configuration options. For example, the assertion level for a particular control signal may be configurable to either high or low. Each option has a small impact on the operation of the interface. Taken individually, you could create a different task for each configuration. The problem would be relegated to the testcase in deciding which flavor of the operation to invoke. You would also have to maintain several nearly identical models.

Simple configurable elements become complex when grouped.

Taken together, the number of possible configurations explodes factorially.[1] It would be impractical to provide a different task for each possible configuration. It is much easier to include configurability in the bus-functional model itself. An RS-232 interface, shown in Figure 5-18, is the perfect example of a highly configurable, yet simple interface. Not only is the polarity of the parity bit configurable, but also its presence, as well as the number of data bits transmitted. And to top it all, because the interface is asynchronous, the duration of each pulse is also configurable. Assuming eight possible baud rates, five possible parities, seven or eight data bits, and

1. Exponential growth follows a K^n curve. Factorial growth follows a n! curve, where n! = 1 x 2 x 3 x 4 x ... x (n-2) x (n-1) x n.

one or two stop bits, there are 160 possible combinations of these four configurable parameters.

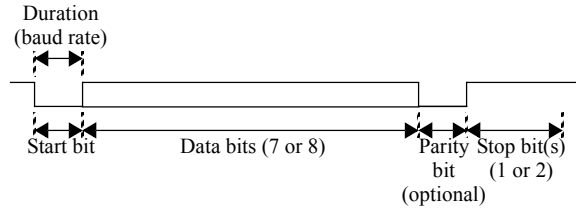


Figure 5-18. Specification for the RS-232 interface

Write a configurable bus-functional model.

Instead of writing 160 flavors of the same transaction, it is much easier to model the configurability itself, as shown in Sample 5-62. Configuration parameters tend to remain static during an entire simulation (unless the corresponding design can be re-configured on-the-fly, and this on-the-fly reconfiguration is the objective of the test). They must also be consistent across different bus-functional tasks within the same bus-functional model. Rather than passing "constant" information through the interface of each bus-functional task, it is better located in the bus-functional model encapsulating structure (see "Encapsulating Bus-Functional Models" on page 127) alongside the bus-functional tasks where it can be accessed directly.

Create a *configuration class*.

Configuration parameters should be implemented as properties in a *configuration class*. An instance of the configuration class would be passed to the bus-functional model via its constructor, alongside the interface binding. Using a separate configuration class will make it easier to create random configurations and to ensure that multiple instances of the bus-functional model have an identical configuration. The current configuration should be kept in a *local* class property to prevent it from being modified without the bus-functional model knowing about it or at the wrong time. If it is possible for a bus-functional model to be reconfigured during a simulation—such as the RS-232 model—a *reconfigure* method should be provided. That method can check that the configuration is valid, that it is an appropriate time for the bus-functional model to be reconfigured—e.g. it is idle—and to perform the necessary operations to notify the bus-functional model tasks of the new configuration. What important safety measure is missing from Sample 5-62?[1]

Sample 5-62. Model for a configurable bus-functional model

```
class rs232_cfg;
   int unsigned baud_rate;
   enum {NONE, ODD, EVEN, MARK, SPACE} parity;
   bit data8;
   bit stop2;
endclass: rs232_cfg

class rs232;
   virtual rs232_if sigs;
   local rs232_cfg  cfg;

   function new(virtual rs232_if sigs,
                rs232_cfg cfg);
      this.sigs = sigs;
      this.cfg  = cfg;
   endfunction: new

   function void reconfigure(rs232_cfg cfg);
      ...
      this.cfg = cfg;
   endfunction: reconfigure

   task send(bit [7:0] data);
      time duration = 1s / this.cfg.baud_rate;
      int i;

      this.sigs.tx <= 1'b0;
      #(duration);
      i = (this.sigs.data8) ? 8 : 7;
      while (i-- > 0) begin
         this.sigs.tx <= data[i];
         #(duration);
      end
      ...
      this.sigs.tx <= 1'b1;
      #(duration * (this.sigs.stop2+1));
   endtask: send
   ...
endclass: rs232
```

See the VMM.

Guidelines 4-104 through 4-107 of the *Verification Methodology Manual for SystemVerilog* specify similar guidelines for implementing configurable transactors.

1. The entire bus-functional model should be protected using a semaphore to prevent concurrent access to the interface signals. See "From Bus-Functional Tasks to Bus-Functional Model" on page 236.

RESPONSE MONITORS

Response transactions can be encapsulated.

Earlier in this chapter, we encapsulated input transactions to abstract the stimulus generation from individual signals and waveforms to generating sequences of operations. A similar abstraction can be used for verifying the response. The repetitiveness of output signals within a transaction can be taken care of and verified inside the bus-functional model. Then the testbench only needs to worry about the correctness of the data carried by the transaction. Sample 5-63 is an example of a bus-functional procedure for an RS-232 monitor.

Verifying the data in the response monitor is too restrictive.

The response verification operation, as encapsulated in Sample 5-63, has a very limited application. It can be used only to verify that the response matches a pre-defined expected value, with no parity error. Can you imagine other possible uses? What if the response can be any value within a predetermined set or range? What if the response is to be ignored until a specific sequence of output values is seen? What if the response, once verified, needs to be fed back to the stimulus generation? What if the parity value is expected to be incorrect? What if responses were to be ignored if the parity bit is invalid? The usage possibilities are endless. It is not possible, a priori, to determine all of them nor to provide a single interface that satisfies all of their needs.

Sample 5-63. RS-232 serial receive bus-functional monitor

```
class rs232;
   ...
   task receive(input bit [7:0] expect);
      ...
      @ (negedge this.sigs.rx); // Wait 4 start
      #(duration * 0.5);     // Shift sample 50%
      data[7] = 1'b0;
      i = (this.sigs.data8) ? 8 : 7;
      while (i-- > 0) begin
         #(duration);
         data[i] = this.sigs.rx;
      end
      if (data !== expect) ...
      ...
   endtask: receive
   ...
endclass: rs232
```

Separate monitoring from value verification.

The most flexible implementation for an response transaction bus-functional task is simply to return to the caller whatever output value was just received. It will be up to a "higher authority" to determine if this value is correct or not. The RS-232 receiver was modified in Sample 5-64 to return the byte received without verifying its correctness. The correctness of the parity is also returned.

Sample 5-64. RS-232 serial receive bus-functional task without verifying correctness of response received

```
class rs232;
   ...
   task receive(output bit [7:0] data,
                output bit       valid);
      ...
      @ (negedge this.sigs.rx); // Wait 4 start
      #(duration * 0.9); // Sample @ 90% of pulse
      data[7] = 1'b0;
      i = (this.sigs.data8) ? 8 : 7;
      while (i-- > 0) begin
         #(duration);
         data[i] = this.sigs.rx;
      end
      ...
      valid = (parity === this.sigs.rx);
      ...
   endtask: receive
   ...
endclass: rs232
```

Consider all possible failure modes.

The bus-functional task shown in Sample 5-64 has some potential problems and limitations. What if the output signal being monitored is dead and the start bit is never received? This task will wait forever. It may be a good idea to provide a maximum delay to wait for the start bit via an additional argument, as shown in Sample 5-65, or to compute a sensible maximum delay based on the baud rate. Notice how a default argument value is used in the task definition to avoid forcing the user to specify a value when it is not relevant, as shown in Sample 5-65, or to avoid modifying existing code that was written before the additional argument was added.

Do not arbitrarily constrain the transaction.

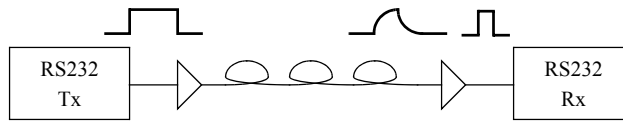
The width of pulses is not verified in the implementation of the RS-232 receive operation in Sample 5-64. Should it? If you assume that the task is used in a controlled 100 percent digital environment, then verifying the pulse width might make sense. This task also could be used in system-level verification, where the serial signal was digitized from a noisy analog transmission line as illustrated in Figure 5-19. In that environment, the shape of the pulse, although

Sample 5-65. Providing an optional time-out for the RS-232 serial receive transaction

```
class rs232;
   ...
   task receive(output bit [7:0] data,
                output bit       valid,
                input  time      timeout = 0);
   begin: receive_task
      ...
      fork: timer
         if (timeout > 0) begin
            #(timeout);
            data = 8'hXX;
            valid = 0;
            disable receive_task
         end
      join_none
      @ (negedge this.sigs.rx); // Wait 4 start
      disable timer
      ...
   end: receive_task
   endtask: receive
   ...
endclass: rs232
```

unambiguously carrying valid data, most likely does not meet the rigid requirements of a clean waveform for a specific baud rate. Just as in real life, where modems fail to communicate properly if their baud rates are not compatible, an improper waveform shape is detected as invalid data being transmitted.

Figure 5-19. Modification to the serial signal in a real system



Generation and monitoring pertains to the ability to initiate a transaction.

We have already seen that input transactions sometimes have to monitor some output signals from the design under verification. The same is true for a response monitor. Sometimes, the monitor has to provide data back as an answer to an "output" transaction. This reporting blurs the line between stimulus and response. Isn't a stimulus bus-functional task that verifies the control or feedback signals from the design also doing response checking? Isn't a monitor task that replies with control flow signals back to the design also doing stimulus generation? The terms *generator* and *monitor* become meaningless if they are attached to the direction of the sig-

nals being generated or monitored. They regain their meaning if you attach them to the initiation of transactions. If a task *initiates* the transaction under full control of the testbench, it is a *stimulus generator*. If the task sits there and waits for a transaction *to be initiated by the design*, then it is a *response monitor*.

Monitors must always be monitoring.

Bus-functional model tasks must be invoked by the testbench. Invoking a bus-functional task either initiates a stimulus transaction or initiates the expectation of a response transaction. What if the design happens to initiate a response transaction but the testbench had not called the appropriate bus-functional model response task? At best, the design will detect that the testbench is not ready to receive data because some control flow signals were left at an appropriate level at the completion of the previous response transactions. But this would result in back-pressure building up inside the design and would not verify the design under maximum throughput. Typically however, output data would "spill" and a gap would be created in the output data stream. At worst, the design will fail to operate correctly because the output transaction protocol will be violated due to missing feedback signals. What if the testbench invokes the response task just a few cycles too late, after the design has already initiated a response transaction? A transaction protocol violation is likely to be reported. To avoid the false errors introduced by the misalignment of the response transaction in the testbench and the design, response monitors should always be active and monitoring the design output interface.

Autonomous Monitors

Decouple timing of transaction and timing of response checking.

Since the testbench is not responsible for the initiation of the response transaction, why give it the responsibility for the initiation of the response monitoring procedure? Testbenches are not usually interested in the timing of the output transaction; testbenches are interested only in verifying that the output data is correct. Therefore, we can decouple the monitoring of the physical interface signals from the retrieval of the output data. As illustrated in Figure 5-20, an independent thread can continuously monitor the output transactions. Output data is extracted from each transaction and put into a FIFO. The testbench retrieves the next output data that was received from the front of the FIFO.

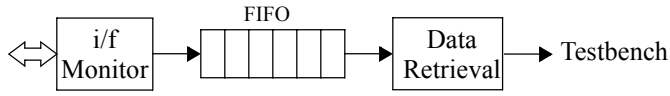


Figure 5-20. Structure of an autonomous monitor

Add a concurrent thread in the bus-functional model.

The bus-functional response task is invoked within a concurrent thread running inside the bus-functional model, as shown in Sample 5-66. An independent thread can be created by simply forking a *forever* loop. Note how the *receive_thread()* task is declared as *local*. This will prevent a user from calling the internal-only method directly. Note also how the receive thread is started in the constructor. This will cause the bus-functional monitor to start operating as soon as it is instantiated, without additional requirements from the user.

Sample 5-66. Autonomous RS-232 response monitor

```
class rs232;
   local bit [8:0] fifo[$];
   ...
   function new(...);
      ...
      fork
         this.receive_thread();
      join_none
   endfunction: new

   local task receive_thread();
      forever begin
         automatic bit [8:0] resp;
         this.receive(resp[7:0], resp[8]);
         this.fifo.push_back(resp);
      end
   endtask: receive_thread
   ...
endclass: rs232
```

Output data is buffered in a queue.

As response data is extracted from the response transactions, it is added to a FIFO to be retrieved later by the testbench. For simple interfaces, such as RS-232, the buffered data is a simple byte, concatenated with its parity-correctness indicator. For more complex interfaces, such as SONET/SDH, the buffered data would be an entire frame. It is not possible to predict how many such data items will need to be buffered before we get the testbench's attention. Therefore, it must be accumulated in a queue (see "Queues" on

page 141) that will grow as data is collected from the output and shrink as data is retrieved by the testbench.

Data collection could be optional.

What if, for a particular testcase, a testbench does not need to examine the response from an interface? Data would accumulate in the FIFO, consuming an ever increasing amount of memory. Hopefully, the simulation would terminate before running out of memory—but that is not likely. The active monitor on that interface is still requried because the protocol may need to be terminated and any protocol-level errors must be checked for—even if the transaction-level response is to be ignored. It should be possible to configure the monitor to extract data from output transactions, verify adherence to the protocol and providing necessary feedback signals but turn off data accumulation. As shown in Sample 5-67, protocol correctness will be monitored without data being accumulated.

Sample 5-67. Optional data collection

```
class rs232;
   ...
   local task receive_thread();
      forever begin
         automatic bit [8:0] resp;
         this.receive_data(resp[7:0], resp[8]);
         if (!this.cfg.sink) begin
            this.fifo.push_back(resp);
         end
      end
   endtask: receive_thread
   ...
endclass: rs232
```

Can provide blocking or non-blocking model.

The bus-functional monitor should provide a task to retrieve the next response transaction that was received. This raises a question: What do we do when there is no response data for the testbench? One solution is to wait for data to become available. But what if the testbench needs to turn its attention elsewhere while the response retrieval task is stuck waiting for data? A better solution is to give the choice to the testbench whether to wait if there is no response data. The testbench should be able to ask the bus-functional monitor if their is response data available before attempting to retrieve it, as shown in Sample 5-68.

Sample 5-68. Nonblocking data retrieval procedure

```
class rs232;
   ...
   local bit [8:0] fifo[$];

   function bit data_avail();
      return this.fifo.size() > 0;
   endfunction: data_avail

   task receive(output bit [7:0] data,
                output           valid);
      while (!this.data_avail())
         @(this.fifo);
      {valid, data} = this.fifo.pop_front();
   endtask: receive
   ...
endclass: rs232
```

Decoupling implies that transaction timing is not relevant.

Decoupling the monitoring of the physical signals from the verification of the response requires that the timing of the response not be relevant to determine functional correctness. As long as the response comes out, the design works. This decoupling is actually one of the great benefits of using high-level testbenches: As the design is modified and pipeline stages are added or removed, the testbench need not be modified. But what if it is functionally important that response comes out after a specific (or range of) number of clock cycles? If it is important, it must be stated in the design specification document. If it is in the design specification document, it must be verified. If it must be verified, the testbench must be able to verify the timing of the response.

Transaction timing can be verified through time-stamping.

Verifying timing does not mean that it must be verified where and when the transaction started (or completed). Timing also can be verified by comparing timestamps. To satisfy the need of both types of testbenches, one where transaction timing is relevant, the other where it is not, timing information should be added to the extracted data. The testbench is then free to compare or ignore the timing information. There is one problem though: *Where* is the timestamp information stored? The response data structure may not have additional fields to store that information. It may not be possible to modify the original data structure to add the necessary field. If the original data structure is potentially reusable in other projects or testbenches, you are adding project- or testbench-specific information to a shared object. If everyone did the same, it quickly would grow into an unmaintainable mess that could not be trusted to be

functionally correct. Instead, extend the original object to add your required information, leaving the original data structure intact. Sample 5-69 shows an extension and timestamping example.

Sample 5-69. Timestamping output data

```
class stamped_eth_frame extends eth_frame;
   time started;
   time completed;
endclass: stamped_eth_frame

class mii_mac_bfm;
   ...
   local task rx(output stamped_eth_frame fr);
      ...
      fr = new;
      fr.started = $time;
      ...
      fr.completed = $time;
   endtask: rx
   ...
endclass: mii_mac_bfm
```

See *vmm_channels*.

The section titled "Transaction-Level Interfaces" starting on page 171 of the *Verification Methodology Manual for SystemVerilog*" specifies a mechanism—the channel—that can be used readily for implementing autonomous monitors and handle the decoupling of monitoring and data retrieval functions.

Slave Generators

Response monitors may need to reply with "input" data.

How do you verify an interface used by the design to fetch data? Typical examples include the instruction fetch interface on a processor or an external memory interface on a design. The transactions are initiated by the design, not the testbench. Therefore, it falls under the category of response monitor. But the transactions do not produce any response data. Instead, they require and consume input data. It is the responsibility of the testbench to supply the data to complete the transaction in a timely manner. Of course, in those cases, the correctness of the data is implied. It will have to be verified elsewhere when it (or its descendent) shows up at another interface.

Slave generators must ask the testbench.

Because of the time-sensitive nature of the transaction, it is not possible to decouple the monitoring of the output interface and the generation of the reply data. The testbench must be ready to supply

stimulus data at all times in response to a transaction initiated by the design. The difficulty is how to get the testbench's attention when required.

The testbench must always call a *standby* task.

By having the testbench constantly call a *standby* task, its attention can be obtained by returning from it. The testbench can then execute testcase-specific code to generate the input data that is supplied by immediately calling the standby task again. Sample 5-70 shows a bus-functional model monitoring the instruction fetch interface of a CPU. Rather than pre-generating the code before the simulation and statically loading it into a memory model, this bus-functional model lets the testbench dynamically generate instructions on-the-fly. Notice the *fetch* task. It is the standby task used by the testbench to provide the instruction opcode fetched from the specified address. Sample 5-71 shows how a testbench could use this standby task to implement a random instruction generator stream.

Sample 5-70. Bus-functional model with standby task

```
class code_mem;
   ...
   event fetch, ready;
   local task monitor_thread();
      forever begin
         @ (negedge this.sigs.as);
         fetch.address = addr;
         -> this.fetch;
         @ (this.ready);
         this.sigs.data <= fetch.opcode;
         this.sigs.rdy  <= 1'b0;
         ...
      end
   endtask: monitor_thread

   task fetch(output [31:0] address,
              input  [31:0] opcode);
      -> this.ready;
      @ (this.fetch);
   endtask: fetch
endclass: code_mem
```

Standby tasks create reusable slave bus-functional models.

Why bother with these standby task calls? Why not simply go in the bus-functional model, add the code we need directly in there and be done with it? That would be the simple way out, but one that will create maintenance challenges later on. This approach makes one big assumption: that you have access to the source code to begin

Sample 5-71. Using the standby task

```
program test;

code_mem pmem = new(...);

initial
forever begin
   bit [31:0] addr, opcode;
   pmem.fetch(addr, opcode);
   opcode = generate_opcode(addr);
end
...
endprogram
```

with. If you wrote the bus-functional model yourself, you do. But it also could be a very large bus-functional model purchased from a third party who will only supply compiled or encrypted code to protect their interests. What if different testbenches need different extensions to the bus-functional model? Are you going to create a different copy for each? What about reusing that bus-functional model in the next revision of the project or in a different project altogether? By specializing a bus-functional model to the specific needs of your testbench(es), you have made reusing it and incorporating upgrades and bug fixes more difficult. You saved a little initially, but lost a lot more in the long run.

See the reactive response completion model in VMM.

The section titled “Reactive Response” starting on page 192 of the *Verification Methodology Manual for SystemVerilog* shows how the generic *vmm_channel* mechanism can be used to implement an even more flexible interface mechanism to slave generators. That interface mechanism is called a request/response completion model.

Multiple Possible Transactions

The next transaction on an output interface may not be predictable.

You may be in a situation where more than one type of transaction can happen on an output interface. Each would be valid and you cannot predict which specific operation will come next. Then how do you decide which *standby* task to call? An example would be a processor that executes an unknown stream of instructions. You cannot predict (without detailed knowledge of the processor architecture and instruction streams) whether a read or a write cycle will appear next on the data memory interface.

Use a transaction descriptor.

How do you write a response monitor when you do not know what kind of transaction comes next? You must write a bus-functional task that identifies the next transaction after it has started. It verifies the preamble to all transactions on the output interface until it becomes unique to a specific transaction. It then builds a transaction descriptor containing any information collected so far to identify, to the testbench via the standby task, which transaction is currently underway. It is then up to the testbench to supply the necessary (and correct) information to complete the verification of the transaction.

Sample 5-72 shows a transaction descriptor for a read or write cycle. The response monitor bus-functional model in Sample 5-73 identifies whether the next transaction from the design is a read or a write cycle and fills in as much of the descriptor as it can. Since the address already has been sampled by the time the decision of the type of cycle was made, the address will be valid in both read and write cycles. If a *read* cycle is observed, the transaction descriptor is completely filled in and no further response is expected from the testbench. If a *write* cycle is observed, the *data* class property is left unfilled. That will be the response expected from the testbench and driven back to the design. Sample 5-74 shows how this transaction descriptor is used by the testbench

Sample 5-72. Transaction descriptor.

```
class ram_trans;
   enum {READ, WRITE} kind;
   bit [31:0] addr;
   bit [31:0] data;
endclass: ram_trans
```

See the reactive response completion model in VMM.

The section titled “Reactive Response” starting on page 192 of the *Verification Methodology Manual for SystemVerilog* shows how the generic *vmm_channel* mechanism can be used to implement a more flexible interface mechanism to slave generators that handles multiple possible transactions. That interface mechanism is called a request/response completion model.

Sample 5-73. Monitoring many possible output transactions

```
class ram_bfm;
   ...
   ram_trans tr;
   local task monitor_thread();
      forever begin
         do
            @ (negedge this.sigs.ale);
         while (this.sigs.cs == 1'b1);
         this.tr = new;
         this.tr.addr = this.sigs.addr;
         this.tr.kind = (this.sigs.rw == 1'b1) ?
            ram_trans::READ : ram_trans::WRITE;
         this.tr.data = this.sigs.data;
         -> do_cycle;
         if (this.tr.kind == ram_trans::WRITE)
         begin
            @ (cycle_done);
            this.sigs.data <= tr.data;
         end
      end
   endtask: monitor_thread()

   task mem_cycle(inout ram_trans tr);
      this.tr = tr;
      @ (do_cycle);
      ->cycle_done;
      tr = this.tr;
   endtask: mem_cycle
endclass: ram_bfm
```

Sample 5-74. Handling many possible output operations

```
...
forever begin
   ram_trans tr;

   mem_cycle(tr);
   case (tr.kind)
   ram_trans::READ :
      tr.data = read_cycle(tr.addr);
   ram_trans::WRITE:
      write_cycle(tr.addr, tr.data);
   endcase
end
...
```

TRANSACTION-LEVEL INTERFACE

Testbenches are removed from the physical level.

As illustrated in Figure 5-21, the purpose of bus-functional models is to remove the testbench from the repetitive physical-level details. The bus-functional model lets the testbench concentrate on the data to be supplied, on the data that was produced and how it is supposed to have been transformed. Once you have a reliable set of bus-functional models, it makes writing testbenches faster and easier. In this section, I will describe how to design a transaction-level interface. The next chapter will describe how to structure a testbench by combining bus-functional models with coherent transaction-level interfaces.

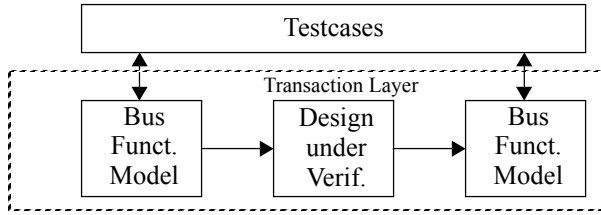


Figure 5-21. Transaction level testbench

The transaction interface must be designed.

Throughout this chapter, the procedural interface of the bus-functional models evolved from the particular transaction being discussed. It was optimized according to the particular point I was trying to make while presenting advantages and disadvantages of various alternatives. Each task was written and evolved independently of each other. The purpose of this chapter thus far was the generation and monitoring of physical-level signals, not the design of a transaction interface. Using this process to write a complete bus-functional model will likely result in an awkward and clumsy transaction-level interface dictated by the physical-level details, not the requirements of the testbenches that must use it. The transaction interface of a bus-functional model must be designed and planned, just like the testcases or the design.

Declare.

When designing a bus-functional model, write the transaction-level interface first. This is akin to writing the header file in C. Code the descriptors, methods, task and function declarations that make up the entire transaction-level interface of your bus-functional model. Leave the body or the implementation of each method empty. This style will let you focus on the transaction-level interface across all

of the transactions. You must start thinking about the needs of the testcases, not the implementation of the bus-functional models. This is the stage where you address questions like "How does the testbench know when and how this transaction completed?" and "How much work can I do without the testbench's attention?" You have to strike a balance between abstraction and controllability. You have to consider the requirements of each testcase, the self-checking mechanism and functional coverage measurement.

Document.

Next, write the user documentation for the bus-functional model. Yes, documentation. It is the only way to ensure that the documentation will reflect the content of the bus-functional model accurately and that it will exist at all. It also presents an opportunity to think about the purpose of each element of the transaction interface and the usage model of the bus-functional model. The documentation will have to describe the functionality and interaction of each element, often highlighting inconsistencies or difficulties that were not considered when coding the interface. Review and iterate over the declaration and the documentation until your have specified a bus-functional model that will meet all of your requirements.

Implement.

With the declaration and documentation of each task completed, the implementation of the bus-functional model becomes a simple coding exercise. During the implementation, you will discover misunderstanding in the specification of the physical interface specification. You will encounter functionality that cannot be implemented as intended. You will find inconsistencies in the bus-functional model specification. Update the transaction interface *and* the documentation as required.

Procedural Interface vs Dataflow Interface

The task called may decide the transaction.

So far, all transaction-level interfaces shown in code samples were procedural interfaces. The nature of transactions was determined by the task being called. If I wanted to execute a *read* cycle, I simply invoked a *read* task. If I wanted to execute a *write* cycle, I invoked a *write* task. It is a simple model that works well on the stimulus side, but breaks down quickly on the response or slave side. As described in "Multiple Possible Transactions" on page 255, a transaction descriptor and a single task has to be used to deal with the unpredictable nature of observed transactions.

Tasks complicate randomization.

A procedural interface also starts to break down when random stimulus is required. As described in "Random Stimulus" on page 307, randomization and constraints in SystemVerilog are built on top of the object-oriented framework and *classes*—and for good reasons also explained in that section. Using a transaction descriptor makes the transaction immediately randomizable and constrainable. A procedural approach requires that the randomization be explicitly modeled using random control flows and random variables, such as *randcase* or *$random*. Procedural random description cannot leverage the constraint language nor the constraint solver built into SystemVerilog.

Tasks complicate transaction-level connectivity.

In all previous examples, the testcases were implemented directly on top of the bus-functional models connected to the physical interfaces of the DUV. Testcases "connected" to bus-functional models by calling the tasks in their procedural interface. But what if those testcases had to be run on a similar design with a slightly different physical interface and thus a slightly different bus-functional model? They would all have to be modified to change the old task calls to the new ones.

A *mailbox* can be used as transaction-level interface.

If the required transactions are described using a descriptor, all that is required to have that transaction executed is to pass it to the bus-functional model for execution. By using a *mailbox* to exchange transaction descriptors, testcases and bus-functional models only need to agree on a particular mailbox to exchange a particular stream of transactions. Connecting to a different transactor simply requires that the new transactor understands the same descriptors and uses the same mailbox. None of the testcases need to be modified. Using a mailbox carrying transaction descriptors to connect testcases and transactors is like using a wire carrying electrons to connect two design components.

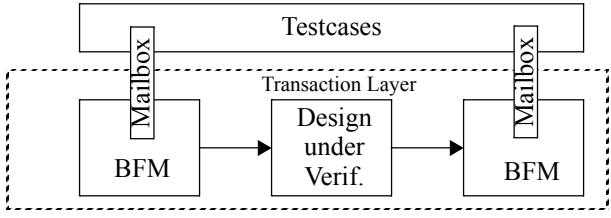


Figure 5-22. Using mailboxes to connect testcases to bus-functional models

Mailboxes are a transaction-level interface.

Sample 5-75 shows a bus-functional model with a mailbox-based transaction-level interface. The *mailbox* is the encapsulation mechanism for the transaction-level interface, much like the *virtual interface* was the encapsulation mechanism for the physical-level interface. In fact, compare the implementation of the two interface levels in Sample 5-75 and Sample 5-60. Both are specified via the constructor, both are maintained in a class property.

Sample 5-75. Mailbox-based transaction-level interface

```
class bus_trans;
   enum {READ, WRITE} kind;
   ...
endclass: bus_trans;
typedef mailbox #(bus_trans) bus_trans_mbox;

class bus_master;
   ...
   bus_trans_mbox inbox;

   function new(...,
                bux_trans_mbox inbox);
      ...
      this.inbox = inbox;
      fork
         this.execute_thread();
      join_none
   endfunction: new

   local task execute_thread();
      forever begin
         bus_trans tr;
         this.inbox.get(tr);
         case (tr.kind)
            ...
         endcase
      end
   endtask: extecute_thread;
endclass: bus_master
```

Mailboxes enable transaction-level bus-functional models.

The traditional bus-functional models, like all the bus-functional models shown so far, are physical-level bus-functional models. They are tied to a specific physical-level interface and provide a transaction-level interface to abstract the physical-level transactions. With mailboxes providing a transaction-level connectivity mechanism, bus-functional models need not be tied to physical interfaces but can be purely implementation-independent transaction-level bus-functional models. For example, the PCI Express

protocol has a well-defined Transaction Level (TL) and Datalink Level (DL) behavior but no implementations. A TL and DL PCI Express bus-functional model could be written as a pure transaction-level BFM. These higher-level bus-functional models have a transaction-level interface on both sides, as illustrated in Figure 5-23.

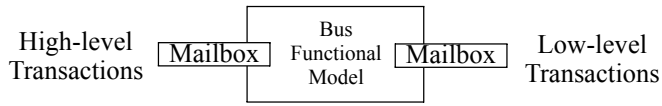


Figure 5-23. High-level bus-functional model

Use a mailbox for each data stream.

A mailbox will transfer transaction descriptors, in order, from a source to a destination. The source is the thread that calls the *put()* method. The destination is the thread that calls the *get()* method. There is no way to enforce the flow of transactions through a mailbox. To avoid confusion, they should remain unidirectional, with transactions flowing always in the same direction. Bus-functional models that transmit and receive information should use two mailboxes: one for transmitted transactions and the others for received transactions. Because transactions are transmitted in order, a bus-functional model that has multiple priorities or different classes of service should use one mailbox per priority or class of service. This will avoid blocking a high-priority transaction because the mailbox is filled with low-priority transactions.

Testbench requirements favor dataflow interfaces.

Notice how all of the arguments in this section had nothing to do whatsoever with the physical protocol implementing the transactions. All of the previous examples focused on the requirements of those physical protocols and used a procedural interface to meet them. But once you consider the needs of the testbench—the user of the bus-functional model—it becomes apparent that a descriptor-based interface is more useful.

See *vmm_channels*.

The section titled “Tranaction-Level Interfaces” starting on page 171 of the *Verification Methodology Manual for SystemVerilog* specifies a very flexible and powerful mailbox-like mechanism called *vmm_channel*s. It also shows how various completion models can be implemented using that mechanism.

You may start with a procedural interface.

You may decide that your testcase requirements do not make a dataflow transaction-level interface a worthwhile investment. You’d

rather stick to a procedural interface. That's quite alright. But remember that even procedural interfaces need to use transaction descriptors—see section titled "Multiple Possible Transactions" on page 255. If it turns out that a dataflow interface is required, it can be added in front of a procedural interface. Simply add the code shown in Sample 5-75 then invoke the appropriate task in the *case* statement in the *execute_thread()* task.

What is a Transaction?

A transaction is an atomic operation.

Earlier in this chapter, a transaction was defined as an operation on a physical interface. With mailboxes allowing bus-functional models to be free from physical interfaces, a transaction becomes a more generic concept. A *transaction* is the smallest operation or data transfer that can be executed to completion by a bus-functional model. Note that what is "smallest" for a high-level bus-functional model can be divided into smaller lower-level transactions, which are in turn the smallest operations that can be executed on a lower-level bus-functional model. For example, a transaction for a USB function BFM is an entire USB transfer, which is implemented by executing USB transactions on a USB device BFM which is in turn implemented by transmitting and receiving USB packets on a USB port BFM which is implemented by toggling or monitoring a physical signal.

Transactions can transfer a variable number of bytes.

Most of the transactions used so far were simple fixed-size transactions. The amount of data transmitted to or from the design was identical in each occurrence of the transactions. For example, an RS-232 interface always transmits a single byte. Most physical interfaces nowadays deal with variable-length data. For example, all ethernet interfaces deal with MAC frames between 64 and 1,518 bytes long. In a PCI interface, the maximum number of bytes that can be transferred in a read or write cycle is not even specified. A variable-length protocol can be built on top of a fixed-length physical interface. For example, a PPP transaction over an RS-232 link will transmit and receive variable-length packets, one byte at a time. How do you design a transaction interface that can handle those differences? The easiest solution is to provide enough memory for the largest amount of data and a "length" specification indicating how many data elements are actually valid. But always specifying the maximum number of data is inefficient and wastes memory.

Use a queue or an array.

Use a queue or an array as part of the transaction descriptor or as the type of the task argument. Sample 5-76 shows the implementation of an ethernet frame, using an array for the variable data. Sample 5-77 shows that same variable-length ethernet frame used as an argument of a task, making it possible to send ethernet frames of any (even invalid) lengths efficiently. Sample 5-78 shows that same variable-length ethernet frame used in a mailbox-based transaction-level interface

Sample 5-76. Variable-length transaction descriptor.

```
class eth_frame;
   bit [47:0] da;
   bit [47:0] sa;
   bit [15:0] len_typ;
   bit [ 7:0] data[];
   bit [31:0] fcs;
   ...
endclass: eth_frame
typedef mailbox #(eth_frame) eth_frame_mbox;
```

Sample 5-77. Using a variable-length transaction descriptor.

```
class mii_mac_bfm;
   ...
   task send(eth_frame fr);
      ...
   endtask: send
   ...
endclass: mii_mac_bfm
```

Sample 5-78. Using a variable-length transaction descriptor with mailboxes.

```
class mii_mac_bfm;
   ...
   function new(...,
                eth_frame_mbox tx,
                eth_frame_mbox rx);
      ...
   endfunction: new
   ...
endclass: mii_mac_bfm
```

How do we know when and how a transaction completed?

Monitors report on observed transactions, including any error that may have occured. Status class properties in the transaction descriptor reported by the monitor indicate the status of the transaction. A monitor may even choose to filter out bad transactions as they would not be recognized by the design anyway. But what about stimulus transactions? It is easy to determine *which* transaction to execute and, for master bus-functional models, *when* to exe-

cute them. But how do we know when their execution has completed and whether or not they executed succesfully?

Blocking Transactions

The transaction completes when the task returns.

In a procedural interface, the unspoken assumption was that, when the task used to execute a transaction returned, the transaction execution was completed. The task could also include *output* arguments to report on the execution status.

Supports in-order transactions only.

This execution and completion model can only support in-order transactions. Because of the blocking nature of the procedural interface, it is not possible to submit multiple transactions and let the bus-functional model choose which one to execute next. If out-of-order execution is required, then a different mechanism for indicating the completion and status of a transaction is needed.

put() cannot implement blocking transactions.

In a dataflow interface, transactions are submitted for execution in a mailbox by calling its *put()* method. However, for *put()* to block, the mailbox must already be full—i.e. there must already be a transaction currently being executed. When the mailbox is empty, *put()* immediately returns before the bus-functional model even has had a chance to execute it. Furthermore, a mailbox can be configured as full with more than one transaction descriptor in it or even to be unbounded, causing repeated invocations of *put()* to immediately return. Therefore, you cannot assume that the transaction has completed once *put()* returns.

See the VMM.

The section titled “In-Order Atomic Execution Model” starting on page 177 of the *Verification Methodology Manual for SystemVerilog* shows how a *vmm_channel* can be used to implement blocking transactions

Nonblocking Transactions

Transactions may be executed without blocking the testbench.

What if the testbench needs to be able to perform other tasks while transactions are being executed? With a blocking procedural interface, an additional thread and a completion indication must be created to allow the execution of the testcase to continue, as shown in Sample 5-79.

Sample 5-79. Forking a blocking interface.

```
initial
begin
   bus_status status;
   event      done;
   fork
      begin
         bfm.read(..., status);
         -> done;
      end
   join_none
   ...
end
```

Signal the completion in the descriptor.

This same mechanism can be encapsulated in the transaction descriptors used by the dataflow transaction-level interface. The completion of the transaction is indicated by triggering the event in the transaction descriptor, as shown in Sample 5-80. The bus-functional model can also annotate the transaction descriptor with status information that will be assumed valid and available to the testbench once the event has been triggered. The testcase, after submitting the transaction to the mailbox, simply waits for the event to be triggered, as shown in Sample 5-81.

Sample 5-80. Event-based transaction completion

```
class bus_trans;
   enum {READ, WRITE} kind;
   ...
   event done;
   enum {UNKNOWN, OK, ERROR} status = UNKNOWN;
endclass: bus_trans;
typedef mailbox #(bus_trans) bus_trans_mbox;

class bus_master;
   ...
   local task execute_thread();
      forever begin
         bus_trans tr;
         this.inbox.get(tr);
         case (tr.kind)
            ...
         endcase
         tr.status = ...;
         -> tr.done;
      end
   endtask: extecute_thread;
endclass: bus_master
```

Sample 5-81. Waiting for transaction completion

```
initial
begin
   bus_trans tr = new();
   tr.kind = READ;
   ...
   inbox.put(tr);
   @tr.done;
   if (tr.status != bus_trans::OK) ...
   ...
end
```

Blocking is a special case.

A blocking transaction is a special case of a nonblocking transaction. It can be implemented my immediately waiting for the transaction completion indication, as shown in Sample 5-81.

Supports out-of-order execution.

This transaction completion and status indication mechanism can support an out-of-order transaction execution model. Multiple transactions can be submitted, either by calling the procedural interface multiple times[1], or by submitting multiple transaction descriptors in the input mailbox.

See the VMM.

The section titled “Out-of-Order Atomic Execution Model” starting on page 182 of the *Verification Methodology Manual for SystemVerilog* shows how a *vmm_channel* can be used to implement nonblocking transactions

Split Transactions

Transactions may be composed of sub-transactions.

Many high-performance bus protocols have split transactions. For example, a *read* transaction could be composed of separate address and data tenures. The bus master can perform the address tenure. While the target device performs the work and buffers data, the master can perform other bus transactions to other devices. The master either polls or is interrupted by the first device when it is ready to complete the *read* transaction. The master then performs the second tenure, transferring data from the device, completing the *read* transaction. The same target device may be able to handle several split and non-split transactions concurrently. Split transactions may also include out-of-order completion.

1. Make sure the task is declared as *automatic* or these concurrent invocations will clobber each other! See section titled "Non-Re-Entrant Tasks" on page 188.

Provide tenure procedures.

Whatever transaction-level interface you decide on, the bus-functional model will have to perform those tenures. They should be implemented as separate sub-transactions. As long as they exist, you should make them public to enable a testbench to have detailed control over the atomic tenures on the physical interface. For example, a testbench may need to create a specific sequence of tenures and transactions to exercise a particular corner case. If the testbench is too far removed from the physical interface, it may not be possible to create. The low-level tenure sub-transactions should be used only by the very few testcases responsible for verifying the physical interface logic.

Provide an interface to the complete transactions.

For any verification project, the bulk of the testcases are concerned with higher-level functionality, not physical interface logic. They do not depend on the split nature of the transaction and do not require detailed control of the physical interface. Having to deal with the tenures would make writing those testbenches tedious and cumbersome. You should provide an interface to the split transaction that makes it look like an atomic operation. Internally, it would execute the tenures as required. But from the testbench's perspective, it would appear like an ordinary transaction. The transaction execution would wait until the split transaction completes and returns with the completed data and status.

Provide nonblocking transaction interface.

To support a mix of split and non-split transactions, with non-split transactions executing between the tenures of a split transaction, you should provide a nonblocking transaction interface. A procedural transaction-level interface may very well wait while the initial tenure of the split transaction is applied. But it should be nonblocking in that it would not wait for the completion of the split transaction. It would return instead with an indication of the status of the split transaction—whether it was accepted by the target device—and a mechanism for alerting the testbench that the transaction has been completed. Sample 5-82 shows an implementation of such a procedural interface: the response of the task is a reference to a transaction completion descriptor that contains completion and status indication. Sample 5-83 shows how such a procedure would be used. It is easy to fork separate threads that will wait for the completion of the split transaction and verify their correctness. Implementing a nonblocking transaction completion mechanism using a

mailbox-based interface has already been shown in the previous section.

Sample 5-82. Nonblocking procedural transaction interface

```
class split_read_resp;
   enum {DECLINED, PENDING, RETRY, ABORT, OK}
      status;
   bit [7:0] data[];
   event completed;
endclass: split_read_resp

class bfm;
   ...
   task split_read(input  bit [23:0] addr,
                   input  int len,
                   output split_read_resp resp);
      resp = new;
      if (this.setup_split_read(addr, len)) begin
         resp.status = DECLINED;
         return;
      end
      resp.status = PENDING;
      this.register_split_read(addr, len,
                               result);
   endtask: split_read
endclass: bfm
```

Sample 5-83. Using a nonblocking procedural transaction-level interface

```
split_read_resp resp;

master0.split_read('h000FFF, 32, resp);
if (resp.status == split_read_resp::PENDING)
   fork
      check_split_response(0x000FFF, 32, resp);
   join_none
...
task check_split_response(bit [23:0]      addr,
                          int             len,
                          split_read_resp resp);
   @resp.completed;
   if (data.size() != len) ...
   ...
endtask: check_split_response
```

See the VMM.

The section titled "Non-Atomic Transaction Execution" starting on page 185 of the *Verification Methodology Manual for SystemVerilog* shows how *vmm_channels* can be used to implement split transactions.

Exceptions

Transactions may be retried.

In many protocols, transactions may fail not because they are invalid but because one of the parties involved in the transaction is busy or is out-of-sync. The transaction should be retried at a later time. Only after a certain number of retries is a transaction considered failed. Should the bus-functional model do the retrying or should you let the testbench worry about it?

Let the bus-functional model do the retry.

The bus-functional model should make its best possible effort to complete a transaction. That includes retrying transactions that did not complete initially. Testbenches should not be burdened with the repetitive retry operations. On the other hand, a testbench may need to have control over the number of retries, or whether to even allow a transaction to be retried. The transaction-level interface could have a parameter specifying the maximum number of attempts. Once the transaction has been tried for the specified number of attempts, it is considered failed. A testbench that does not wish a transaction to be retried would simply specify a single attempt. A default value for the number of attempts (usually specified in the protocol) could also be provided so it would not need to be specified for each invocation—only when a different value is required. Sample 5-84 shows an example of an MII ethernet procedural interface with control over the number of transmission attempts.

Sample 5-84. Retried transaction

```
class eth_mac_bfm;
   ...
   task send(input  mac_frame frame,
             output bit       success,
             input  int       attempts = 10);
      while (attempts-- > 0) begin
         ...
            success = 1;
            return;
         ...
      end
      success = 0;
   endtask: send
endclass: eth_mac_bfm
```

Can add exception controls to transaction descriptor.

If a dataflow interface is used, the additional parameters can be included in the transaction descriptor, as shown in Sample 5-85. That is fine if there is a one-to-one correspondence between the transaction descriptor and the bus-functional model that can inject

or report the exception. But what if the same transaction descriptor can be used on different bus-functional models and protocol, each with different exceptions? For example, an ethernet frame can be transmitted using MII, RMII, SMII, GMII, XGMII or XAUI physical interfaces, each with their specific exceptions. Should the ethernet frame descriptor thus contain the necessary exception controls for all of these physical interfaces?

Sample 5-85. Exception controls in transaction descriptor.

```
class eth_frame;
   ...
   event done;
   bit success,

   int attempts = 10;
   ...
endclass: eth_frame
```

Transactions are composed of physical symbols.

For most interfaces, the transaction data is not transmitted in parallel, in a single cycle. Instead, the transaction data is divided into *symbols* transmitted sequentially over multiple cycles. For example, a byte transmitted over an RS-232 physical interface is translated into one-bit symbols. An ethernet frame is transmitted as 4-bit symbols on a MII interface and as 2-bit symbols on a RMII interface. Some symbols may be added to the transaction data by the protocol for framing, synchronization, or error protection. For example, an ethernet frame is prefixed with an 8-byte preamble and a USB packet is prefixed with an 8-bit synchronization pattern.

Symbol-level parameters must be controllable.

For each symbol, a protocol usually has several possible exceptions, as well as symbol-level flow control and status indication. But a transaction-layer interface deals with information for the entire transaction, not individual symbols. This is fine when verifying functionality that resides behind the interface. But when verifying the implementation of the interface itself, it is necessary to have detailed control over all relevant symbol-level parameters. One possible solution is to provide symbol-level parameters for each symbol necessary to execute a transaction in the transaction-level interface or transaction descriptor, as shown in Sample 5-86.

Interpret the exception controls when executing the transaction.

The bus-functional model simply needs to look at the appropriate exception controls and execute them when appropriate, as shown in Sample 5-87. Although you should plan for all exceptions required by your verification plan, there is no need to implement all of them

Sample 5-86. Symbol-level exception controls in transaction descriptor.

```
class mii_symbol_except;
   enum {NONE, SKIP, CORRUPT, DISABLE} kind;
endclass: mii_symbol_except

class eth_frame;
   ...
   event done;
   bit success,

   int attempts = 10;
   mii_symbol_except mii_exceptions[int];
   ...
endclass: eth_frame
```

at once. You should start with a bus-functional model that cannot inject any exceptions and get the basic functionality of the design debugged first. Once the design's reaction to exceptions must be verified, implement them as needed.

Sample 5-87. Executing symbol-level exceptions

```
class mii_mac_bfm;
   task send(eth_frame fr);
      bit [3:0] symbols[];
      ...
      foreach (symbols[i]) begin
         if (fr.mii_exceptions.exists[i]) begin
            ...
         end
         ...
      end
      ...
   endtask: send
endclass: mii_mac_bfm
```

All exceptions must be implemented in the model.

This approach is simple and direct. But it requires that either the exception and its control already exist in the bus-functional model, or that you are able to modify the bus-functional model to add a new exception to it. What if you do not have access to the source code of the bus-functional model and it does not support the exception you need? What if your job is to create a reusable bus-functional model and you do not want users constantly modifying—and breaking—your model yet you do not want to always have to add new exceptions to it?

A bus-functional model can execute user-defined code.

What if a bus-functional model could be extended with user-defined code? This would mean that the user could make it do things it was not originally coded to do. It would mean it would not have to be able to do everything from day one. It would mean a simpler bus-functional model that could meet the unpredictable needs of different verification environments. *Callbacks* allow bus-functional models to execute user-defined code. Callbacks are invoked by the bus-functional models at appropriate points in the execution of the transaction. In C, callbacks are implemented using pointers to functions. But pointers to functions do not exist in SystemVerilog. Experienced C users often point to this apparent lack as a fatal flaw in SystemVerilog. They are wrong. In an object oriented programming model, *virtual methods* are used instead of pointers to functions. And SystemVerilog supports those just fine.

Use *virtual callback* method.

In SystemVerilog, callbacks are implemented using *virtual* methods. The default implementation of a callback should be to return an innocuous value to eliminate the requirement that all testbenches overload the callback method to render a bus-functional model functionally correct. Testbench-specific code can replace the default implementation by providing an overloaded definition in a derived class. Information can be passed between the bus-functional model and the callback method through arguments or through public or *protected* class properties. Sample 5-88 shows a callback invoked just before an ethernet frame is to be transmitted by a MII interface. The callback is extended in a testbench, as shown in Sample 5-89, to introduce a delay before the transmission of every frame. The bus-functional model in Sample 5-88 is able to accept only one callback extension. The *cbs* class property could be made into a *queue* to accept a series of callback extensions, each dealing with different aspects of a testcase or testbench.

Callbacks can modify default behavior.

It is easy to write a bus-functional model that implements and executes a protocol as fast as possible and without any errors. But when it comes time to verify the design interfacing to that protocol, that is not the most interesting. You need to be able to make the bus-functional model deviate from its default behavior. Callbacks can be used to do that. The nice thing about callbacks is that you need not extend them. So by default, a bus-functional model will operate as fast as possible and without errors. But for those times where you need it to do something different, a callback is there to give you that control. For example, as shown in Sample 5-89, a

Sample 5-88. Bus-functional model callback method

```
class mii_mac_cbs;
   virtual task pre_frame_tx(mii_mac_bfm bfm,
                             eth_frame    fr,
                             ref bit      drop);
   endtask: pre_frame_tx
   ...
endclass mii_mac_cbs

class mii_mac_bfm;
   mii_mac_cbs cbs = null;
   ...
   task send(eth_frame fr);
      if (this.cbs != null) begin
         bit drop = 0;
         this.cbs.pre_frame_tx(this.fr, drop);
         if (drop) return;
      end
   endtask: send
   ...
endclass: mii_mac_bfm
```

Sample 5-89. Overloading the callback method

```
class my_mii_mac_cbs extends mii_mac_cbs;
   virtual task pre_frame_tx(mii_mac_bfm bfm,
                             eth_frame   fr,
                             ref bit     drop);
      repeat (10) @(bfm.sigs.tx);
   endtask: pre_frame_tx
endclass: my_mii_mac_cbs

initial
begin
   my_mii_mac_cbs cb = new;
   bfm.cbs = cb;
   ...
end
```

callback can be used to potentially modify the answer of a USB device to an IN transaction. By default, the answer is an ACK. But the callback can be extended to answer with a NAK, STALL or not answer at all.

Provide symbol-level callbacks.

Callbacks can be defined to provide controllability down to the symbol level. Before transmitting or receiving a symbol, a callback should be called with the necessary arguments to let a testbench modify the default symbol-level behavior. For example, Sample 5-91 shows a symbol-level callback method for a slave PCI memory

Sample 5-90. Callback to modify default behavior

```
class usb_trans;
   typedef enum {ACK, NAK, STALL, NONE} answer;
   ...
endclass: usb_trans
...
class usb_device_cbs;
   virtual function void
      pre_in_ack(usb_device_bfm        bfm,
                 usb_trans             tr,
                 ref usb_trans::answer answ);
   endtask: pre_in_ack
endclass: usb_device_cbs
```

read interface. It controls symbol-level flow control by introducing time advances that will delay the assertion of the *target-ready* signal. It also controls byte-enable indications, has the possibility of aborting the transaction or signaling a parity error for this symbol.

Sample 5-91. Symbol-level callback method

```
class pci_slave_cbs;
   virtual task pre_symbol(ref bit [31:0] data,
                           ref bit [ 3:0] be,
                           ref bit        abort,
                           ref bit        perr);
   endtask: pre_symbol
endclass pci_slave_cbs

class pci_slave_bfm;
   ...
   task mem_read_tx();
      ...
      while (...) begin
         ...
         this.sigs.trdy <= 1'b1;
         if (this.cbs != null)
            this.cbs.pre_symbol(data, be,
                                abort, perr);
         if (abort) begin
            this.sigs.abrt = 1'b0;
            return;
         end
         this.sigs.trdy   <= 1'b0;
         this.sigs.data   <= data;
         this.sigs.cmd_be <= be;
         ...
      end
      ...
   endtask: mem_read_tx
endclass: pci_slave_bfm
```

Use symbol-level callbacks to inject symbol-level errors.

From the transaction-level interface, it is simple to inject transaction-level errors such as a bad CRC or a packet that is too short. But how can you inject errors at the symbol level, such as corrupting a symbol, violating the handshake protocol or unexpectedly terminating a transaction? As long as you have control over the symbol-level parameters in a callback, why not take this opportunity to use the symbol-level callback to inject symbol-level errors? Simply add parameters to the symbol-level callback methods for the errors that can be injected. By default, they are set to not inject any errors. If they are not modified in the callback, no errors will be injected.

Sample 5-92 shows the transaction-level and symbol-level callbacks for a MII ethernet interface. From the symbol-level callback, it is possible to corrupt the symbol, cause the TX_EN signal to be deasserted, cause the TX_ER signal to be asserted or abort the frame altogether. When implementing a bus-functional model, it is necessary to provide every mechanism for breaking the protocol that the design should be able to sustain.

Sample 5-92. Transaction and symbol-level callbacks.

```
class mii_mac_cbs;
   virtual task pre_frame_tx(mii_mac_bfm bfm,
                             eth_frame   fr,
                             ref bit     drop);
   endtask: pre_frame_tx

   virtual function void
      pre_symbol_tx(mii_mac_bfm bfm,
                    eth_frame   fr,
                    ref bit [3:0] symbol,
                    ref bit       tx_en,
                    ref bit       tx_er,
                    ref bit       abort);
   endfunction: pre_symbol_tx
   ...
endclass mii_mac_cbs
```

Time may not be allowed to advance during a callback.

Is time allowed to advance inside a callback—i.e. can a callback method be blocking? The answer is: It depends. If the transaction protocol includes handshaking and flow-control indicators, it is possible to have time advance while a callback is executed. This would introduce delays in the execution of the protocol. Other transaction protocols may suffer a total breakdown if any delay is introduced, in which case the callback must execute and return in zero-time.

SystemVerilog has a built-in mechanism for enforcing time restrictions on callbacks: Use a *task* for callback methods that are allowed to advance time and use a *function* returning a *void* type for callback methods that must execute in zero-time. For example, as shown in Sample 5-92, delay can be inserted before the transmission of an ethernet frame, but once it has started, no delays can be introduced before a symbol is transmitted.

See the VMM.

The section titled "Callback Methods" starting on page 198 of the *Verification Methodology Manual for SystemVerilog* provides detailed guidelines for implementing callbacks in transactors.

SUMMARY

Model your clock signals in a *module*. Be careful about time resolution issues, delta cycle alignment and implicit synchronization of asynchronous signals.

Encapsulate repetitive physical-level operations into bus-functional tasks. Collect all of the bus-functional tasks for a physical interface or protocol into a bus-functional model. Detect concurrent activation of bus-functional tasks within the same bus-functional model using a semaphore.

Design an effective transaction-level interface with a suitable transaction completion and status notification mechanism.

Provide callbacks in bus-functional models and response monitors to enable access to symbol-level protocol parameters and inject symbol-level errors.

CHAPTER 6 ARCHITECTING TESTBENCHES

A testbench need not be a monolithic block. Although Figure 1-1 shows the testbench as a large thing that surrounds the design under verification, it need not be implemented that way. The design is also shown in a single block, and it is surely not implemented as a single unit. Why should the testbench be any different? Figure 6-1 depicts the architecture of a generic testbench. In this chapter, I will describe how to implement each component.

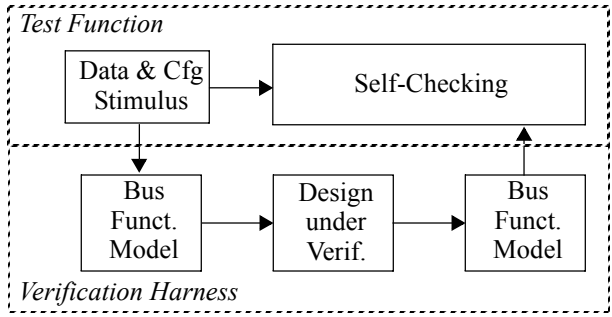


Figure 6-1. Typical testbench architecture

The previous chapter was about low-level testbench components.

In Chapter 5, we focused on the stimulus and monitoring of the low-level signals going into and coming out of the device under verification. I showed how to abstract them into transactions using bus-functional models. The emphasis was on the stimulus and response of interfaces and the need for managing separate execution threads underneath a useful transaction-level interface. If you

prefer a bottom-up approach to writing testbenches, I suggest you start with the previous chapter.

This chapter focuses on the structure of the testbench.

This chapter concentrates on implementing the many testcases and filling the functional coverage models that were identified in your verification plan. I show how best to structure the bus-functional models into a transaction-level verification harness. This verification harness will create the platform on top of which the self-checking structure and stimulus sources will be built. I also describe how to create random generators that can be constrained easily, with a minimum of modifications, to create a complete verification environment.

A hypothetical design will be used to illustrate the concepts.

Figure 6-2 shows the interfaces around a hypothetical ATM switch node design. A management interface allows a processor to read and write internal registers to configure the switch node. The bus-functional models shown in Sample 6-1 and Sample 6-2 are also assumed to be available. This design and bus-functional models will be used throughout this chapter to illustrate important concepts. .

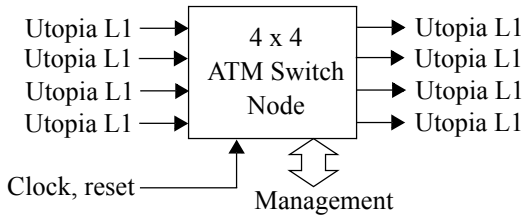


Figure 6-2. 4x4 ATM switch design

VERIFICATION HARNESS

Create a transaction-level verification harness.

All of the testbenches have to interface through an instantiation to the same design under verification. It is safe to assume that they all require the use of the same bus-functional models to generate stimulus and to monitor response. Instead of a monolithic block, the testbenches should be designed with a layer of physical-level bus-functional models. This physical-level layer, common to all testbenches for the design under verification, is called the *verification harness*. The *test functions* required to implement the testcases identified in the verification plan are built on top of the verification

Sample 6-1. Utopia Level 1 ATM-layer bus-functional model

```
class atm_cell;
   ...
endclass: atm_cell
...
interface utopia_L1_if;
   logic       clk;
   logic [7:0] data;
   logic       soc;
   logic       enb;
   logic       clav;

   clocking cb @ (posedge clk);
      ...
   endclocking: cb
endinterface: utopia_L1_if;
...
class utopia_L1_atm_bfm;
   ...
   extern function
      new(virtual utopia_L1_if tx_sigs,
          virtual utopia_L1_if rx_sigs);
   extern task send(atm_cell cell);
   extern task receive(output atm_cell cell);
endclass: utopia_L1_atm_bfm
```

Sample 6-2. Management interface bus-functional model

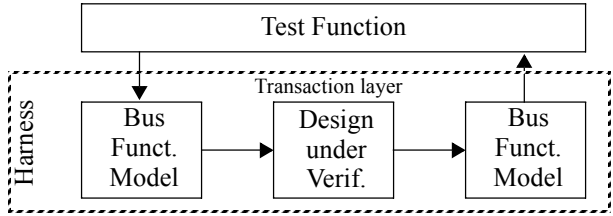
```
interface utopia_mgmt_if;
   logic [15:0] addr;
   logic [15:0] data;
   logic        rd;
   logic        wr;
   logic        rdy;
endinterface: utopia_mgmt_if
...
class utopia_mgmt_master_bfm;
   ...
   extern function
      new(virtual utopia_mgmt_if sigs);
   extern task write(input bit [15:0] wadd,
                     input bit [15:0] wdat);
   extern task read(input  bit [15:0] radd,
                    output bit [15:0] rdat);
endclass: utopia_mgmt_master_bfm
```

harness, as illustrated in Figure 6-3. The *test function* and the *harness* together form a *testbench*.

Encapsulate the verification harness.

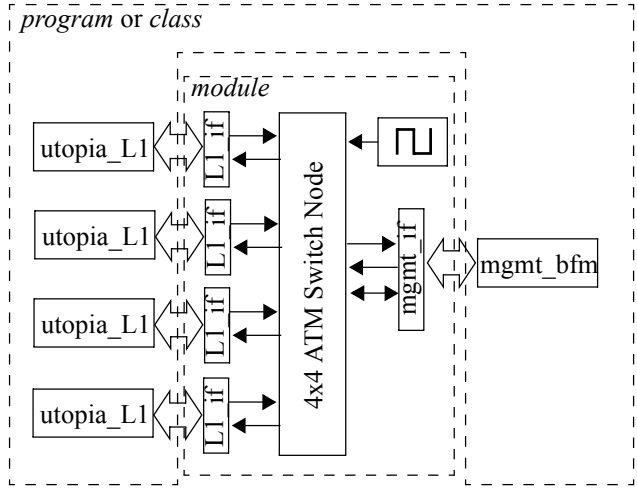
The encapsulation of a transaction-level verification harness is relatively simple. The design under verification, the clock generators and the top-level signals and interfaces are instantiated in a top-

Figure 6-3. Structure of a transaction-level testbench



level *module.* The bus-functional model instances and their connectivity to the interface instances are encapsulated in a *program* or a *class* eventually instantiated in a program. The various test functions then instantiate the encapsulated verification harness. Figure 6-4 depicts the structure of the verification harness for the ATM switch node and the location of the components in the top-level *module* or encapsulating *program* or *class*.

Figure 6-4. Verification harness for 4x4 ATM switch design



Interface instances and DUV in top-level *module.*

The design under verification and the wires connecting to its top-level pins are instantiated in the top-level *module*. The clock generators are also coded in that same module. This will ensure that the design is simulated as a set of module threads. The top-level module for the example design is shown in Sample 6-3.

Sample 6-3. Top-level module for ATM switch node

```
module top;
utopia_L1_if   tx_0(), tx_1(), tx_2(), tx_3();
utopia_L1_if   rx_0(), rx_1(), rx_2(), rx_3();
utopia_mgmt_if mgmt();
...
reg            reset = 0;
reg            clk   = 0;

switch_node dut(tx_0.clk, ..., clk, reset);

always #5 clk = ~clk;

endmodule: top
```

Bus-functional models in top-level *program* or *class*.

The bus-functional models are instantiated in a top-level *program* or *class* that will eventually be instanted in a *program*. This ensures that the testbench executes as program threads and can reliably react to events and assertions in the design. Sample 6-4 shows the bus-functional models instantiated in a *program* whereas Sample 6-5 shows them instantiated in a *class*. Both look very similar but I prefer to use a *class* as it has a few advantages over a *program* which will be discussed in the next section.

Sample 6-4. Top-level *program* for ATM switch node

```
program harness;

utopia_L1_bfm  atm0 = new(top.tx_0, rx_0);
utopia_L1_bfm  atm1 = new(top.tx_1, rx_1);
utopia_L1_bfm  atm2 = new(top.tx_2, rx_2);
utopia_L1_bfm  atm3 = new(top.tx_3, rx_3);
utopia_mgmt_bfm cpu = new(top.mgmt);

task reset;
   top.rst <= 1;
   repeat (3) @(negedge top.clk);
   top.rst <= 0;
endtask

endprogram
```

Test functions use transaction interfaces in the verification harness.

A complete verification harness provides a transaction-level abstraction of the design to be verified. It provides a foundation on which the data generation mechanism, the self-checking structure and the functional coverage measurements are built. Test functions are implemented by using the transaction-level interface elements in the bus-functional models and verification harness itself. These interface elements are accessed using hierarchical references in a

Sample 6-5. Top-level *class* for ATM switch node

```
class harness;

   utopia_L1_bfm   atm0 = new(top.tx_0, rx_0);
   utopia_L1_bfm   atm1 = new(top.tx_1, rx_1);
   utopia_L1_bfm   atm2 = new(top.tx_2, rx_2);
   utopia_L1_bfm   atm3 = new(top.tx_3, rx_3);
   utopia_mgmt_bfm cpu  = new(top.mgmt);

   task reset;
      top.rst <= 1;
      repeat (3) @(negedge top.clk);
      top.rst <= 0;
   endtask

endclass: harness
```

single instance of the verification harness. Sample 6-6 shows a partial test function that configures the device then injects an ATM cell in one of the ports.

Sample 6-6. Test function using a verification harness

```
program my_test;

harness th = new;

initial
begin
   atm_cell cell;
   th.cpu.write(16'h0001, 16'h0010);
   ...
   th.atm0.send(cell);
   ...
   $finish;
end

endprogram: my_test
```

See the VMM.

The section titled "Testbench Architecture" starting on page 104 of the *Verification Methodology Manual for SystemVerilog* provides additional techniques and guidelines for architecting a verification harness and test functions.

DESIGN CONFIGURATION

Most designs require configuration.

Unless you are verifying a very simple design, or an implementation unit of a much larger design, it will be necessary to perform

certain configuration operations before it will be possible to apply stimulus to and observe response from the design. Configuration may be as simple as enabling some data path, or it may be as complicated as generating, then downloading, firmware code. It may involve writing to internal registers, writing to an embedded memory or setting external pins to particular levels.

Avoid using one or two configurations.

Because of the often complex nature of a device configuration, it is not unusual for verification to proceed with only one or two device configurations. Device configurations are maintained as a series of register write operations that "magically" produce a configuration. The testbenches are then written according to the configuration usually loaded. Unfortunately, this will likely prevent some bugs from being uncovered. Some unexpected correlation may exist between different configuration parameters. If these parameters are not exercised, the correlation will not be highlighted.

Abstracting Design Configuration

Model the configuration.

Instead of relying on an implicit knowledge of the current design configuration, why not create a high-level model of the configurable elements of the design? That model could then be used by the self-checking structure (see "Self-Checking Testbenches" on page 292) to determine the correctness of the response. For example, a design could have an input pin that can be used to select between two different management interfaces. As illustrated in Sample 6-7, you can use an enumerated type to model the currently selected interface. That enumerated type can then be passed to the verification harness to instantiate the proper bus-functional model based on the interface configuration. The verification harness would also use the value to determine the polarity used to drive the interface selection pin.

A harness *class* is easier to make configurable.

If the configurability of the device requires a configurable verification harness, putting the harness in a *class* will make implementing and controlling the configuration much easier than in a *program*. A *program* is like a *module*. The structure of a program is more or less predefined and can only be controlled during elaboration using *if-* and *for-generate* statements. Once the simulation has started, the structure of the verification harness *program* cannot be modified. The structure of a *class*, on the other hand, is determined at runtime by its constructor. When implementing the constructor, you

Sample 6-7. Modeling interface configuration using *e*

```
class utopia_mgmt_intel_bfm
   extends utopia_mgmt_bfm;
   ...
endclass: utopia_mgmt_intel_bfm

class utopia_mgmt_motorola_bfm
   extends utopia_mgmt_bfm;
   ...
endclass: utopia_mgmt_motorola_bfm
...
class device_cfg;
   enum {INTEL, MOTOROLA} mgmt_mode;
   ...
endclass: device_cfg

class harness;
   ...
   utopia_mgmt_bfm cpu;;
   ...
   function new(device_cfg cfg);
      case (cfg.mgmt_mode)
      device_cfg::INTEL: begin
         utopia_mgmt_intel_bfm m = new(...);
         this.cpu = m;
         top.int_mot = 0;
      end
      device_cfg::MOTOROLA: begin
         utopia_mgmt_motorola_bfm m = new(...);
         this.cpu = m;
         top.int_mot = 1;
      end
      endcase
   endfunction
   ...
endclass: harness
```

have access to the full power of the SystemVerilog language to determine how to construct the verification harness. And because it is constructed at runtime, not elaboration time, it is possible to have the testcase influence the configuration of the design and the verification harness before it is constructed. That is why I prefer to encapsulate my verification harnesses in classes rather than programs. As a secondary benefit, I have access to the object-oriented programming features of the harness class to make testcase-specific extensions to the harness or create a harness base class where I can locate device-independent functionality.

See vmm_env.

The section titled “Simulation Control” starting on page 124 of the *Verification Methodology Manual for SystemVerilog* defines a base class *vmm_env* and usage guidelines for creating a verification harness class.

Do not model the implementation of the configuration.

Even though the configuration of the device is expressed in terms of ones and zeroes in various bit fields in various registers, it is not necessary to use the same approach when modeling a device configuration. Instead of maintaining an image of the register values, model the purpose and function of the configuration. A high-level description of the device configuration will be much easier to use in the self-checking structure and won’t necessitate the interpretation of low-level bit fields.

For example, the configuration of the switch table in the example ATM switch node design could be implemented as a series of bits in a register. If bit *x* in register *y* is set, then any cell with a VPI value equal to *y* is forwarded to output port *x*. As shown in Sample 6-8, the same information can be modeled in a more abstract fashion by using an array of a queues of integers. Cells with a VPI value of *y* are forwarded to all ports whose number is found in the queue at index *y* of the array.

Sample 6-8. Modeling configuration function, not implementation

```
class to_ports;
   bit [1:0] number[$];
endclass: to_ports
class device_cfg;
   ...
   to_ports table[256];
   ...
endclass: device_cfg;
```

Collect all device configuration information in a single *class*.

As shown in Sample 6-7 and Sample 6-8, it is good practice to collect all device configuration information under a single descriptor. This technique makes it easier to pass it to the verification harness and the self-checking structure. Collecting device configuration information will also make it possible to create constraints and relationships between various configuration items and include methods to ensure internal consistency.

Design configuration may include test configuration.

The “design” configuration may include configuration parameters that are outside of the design itself but influence the stucture or behavior of the testbench in which it sits. If your design can be used

in different system configurations, why limit the testbench structure to just one of the possibilities? For example, a USB hub design could be surrounded by a configurable number of devices. Some devices would be low speed, others full speed. Some could have asynchronous endpoints, others have multiple interfaces with alternate settings. Some devices actually could be another instance of the USB hub with further devices connected to it. Based on the test configuration, the necessary instances of the design under verification are created and connected to the necessary instances of bus-functional models.

Configuring the Design

Compile the configuration description into bit fields.

Once the device configuration is captured in an instance of the configuration descriptor, it will be necessary to configure the design to match. This step will necessitate the translation of the various configuration items into the appropriate bit field values in the appropriate registers. This step may seem like a daunting task—and it usually is—but it is simply coding the process you would have to perform intellectually otherwise. This translation will provide for a better documentation of the device configuration process. This translation will also ensure that configuring the design to match is repeatable, should the location of various bits fields be reorganized or their encoding modified. Sample 6-9 shows how the switch table in Sample 6-8 could be compiled into the corresponding bit field values in the corresponding registers.

Sample 6-9. Translating a high-level configuration descriptor

```
class device_cfg;
   ...
   to_ports table[256];

   task apply(utopia_mgmt_bfm cpu);
      bit [15:0] entry;

      for (this.table[i]) begin
         entry = 0;
         for (this.table[i].number[j]) begin
             entry[table[i].number[j]] = 1;
         end
         cpu.write(16'h0800 + i, entry);
      end
   endtask: apply
endclass: device_cfg
```

Grow the configuration capability.

It is not necessary to implement the translation process of the entire configuration descriptor from day one. The first simulations will likely be performed using a simple device configuration, leaving the bulk of the configuration parameters in their default state. Therefore, translate only that part of the configuration descriptor that is relevant for these first simulations. As more and more configuration parameters are being verified or are supported by the self-checking structure, they should be added to the translation process similarly. Eventually, you will end up with the entire configuration descriptor appropriately translated and programmed into the device.

Assert that unsupported configurations are not used.

While the configuration translation process does not support certain configuration parameters, you must ensure that they are not accidentally used in a simulation. The translation procedure must check that all unsupported configuration parameters are at their default values. Sample 6-10 shows the translation process for the management interface configuration signal. Because one of the modes is not currently available (because the bus-functional model may not be ready yet), it will report an error if the unsupported configuration is attempted.

Sample 6-10. Detecting unsupported configurations

```
class harness;
   ...
   utopia_mgmt_bfm cpu;;
   ...
   function new(device_cfg cfg);
      case (cfg.mgmt_mode)
      device_cfg::INTEL: begin
         $write("ERROR: Intel-style mgmt not
                available yet...");
         $finish;
      end
      device_cfg::MOTOROLA: begin
         utopia_mgmt_motorola_bfm m = new(...);
         this.cpu = m;
         top.int_mot = 1;
      end
      endcase
   endfunction
   ...
endclass: harness
```

Random Design Configuration

Randomize the configuration.

Once you have a descriptor capable of coherently describing any possible configuration of your design, why bother specifying it manually? You'd probably always be specifying the same configuration anyway, which is exactly the problem we were trying to avoid. If you can generate random instructions, packets or data items, why not generate a random device configuration as well? By using a different randomly generated device configuration in each simulation run, you quickly will cover many more combinations. If an unintended correlation between parameters exists, it is likely to be exposed. Sample 6-11 shows a randomizable configuration descriptor. Because the configuration descriptor is already encapsulated in a *class*, all that was required was the use of the *rand* attribute on the class properties modelling the configurable parameters.

Sample 6-11. Randomizable configuration descriptor

```
class to_ports;
   rand bit [1:0] number[$];
endclass: to_ports
class device_cfg;
   rand enum {INTEL, MOTOROLA} mgmt_mode;
   rand to_ports table[256];

   task apply(utopia_mgmt_bfm cpu);
      ...
   endtask: apply
endclass: device_cfg
```

Add constraints to match limitations.

But what about the limitations of your configuration translation procedure? If an unsupported configuration is generated, it will cause an error to be reported. You should maintain a set of constraints that match the current limitation in the device configuration support. As more and more of the configuration parameters are supported, constraints are removed. Sample 6-12 shows the constraints that would be added to all simulation runs temporarily to prevent a configuration, unsupported by the verification harness shown in Sample 6-10, from being generated. Of course, at the end of your verification project, that constraint block should be empty because

your verification harness should support all functions required by your testcases.

Sample 6-12. Support limitation constraints

```
class device_cfg;
   ...
   constraint unsupported {
      mgmt_mode != INTEL;
   }
endclass: device_cfg
```

Add constraints to generate simple debug configurations.

Likewise, a completely random configuration is not likely to be useful for the first simulations. In the early stage of a project, a design will contain many functional bugs. They will be easier to identify and debug if a simple configuration is used. The first simulations should be executed with constraints on the configuration descriptor to generate a simple configuration. Once the design simulates successfully, these constraints are removed to increase immediately the number of configuration combinations that can be verified.

Use functional coverage to identify configurations that were verified.

If the device configuration is generated randomly, how do you know which configurations you have verified? Simple: A device configuration is treated just like a feature of the design. All interesting and relevant configurations should be identified in the verification plan. They should be included in the functional coverage model of the design. Functional coverage measurements will identify which configurations were indeed verified and the ones that remain to be verified.

Randomize configuration in harness.

To ensure that the design and test configuration is randomized as often as possible, it should be randomized in the verification harness as shown in Sample 6-13. It also avoids having to repeatedly code the configuration randomization in every testcase. Each testcase will have to call a *configure()* method to have the selected configuration applied to the device under verification as shown in Sample 6-14. This gives the opportunity to the testcase of modifying the randomized configuration, or re-randomize it under different constraints before applying it. It is also a better approach than having each testcase invoke the *device_cfg::apply()* method directly because device configuration may impact the configuration of the verification harness itself. By encapsulating the configuration process in a harness method, the harness is free to perform all nec-

essary steps to configure the device and the verification harness in a consistent and coherent fashion.

Sample 6-13. Randomizing configuration in harness

```
class harness;
   device_cfg cfg;
   ...
   function new();
      ...
      if (!cfg.randomize()) ...
   endfunction: new

   task configure();
      this.cfg.apply(this.cpu);
   endtask: configure
   ...
endclass: harness
```

Sample 6-14. Testcase with random device configuration

```
program test;

harness th = new;

initial
begin
   ...
   th.configure();
   ...
end
endprogram: test
```

See vmm_env.

Guidelines 4-33, 4-34, 4-40 and 4-42 of the *Verification Methodology Manual for SystemVerilog* describe how a configurable verification environment should be implemented when using the *vmm_env* base class.

SELF-CHECKING TESTBENCHES

Testbenches must be self-checking.

As discussed in "Verifying the Response" on page 86, visually inspecting simulation results to determine functional correctness is not an acceptable long-term strategy. Whatever intellectual process you would go through to identify an error visually in the simulation result must be coded in your testbench. This technique will let the testbench detect errors and declare success or failure on its own. Coding error detection into your testbenches will free you to work

on other tasks while the design is autonomously subjected to hundreds of simulations.

Define what to check.

The problem with verification is that you cannot find an error where you are not looking. It is therefore necessary, during the verification planning stage, to identify all of the failure modes that must be checked for and how they can be detected. Typical correctness criteria include data transformation, data ordering, protocol correctness, data losses and design state. The requirement of the self-checking mechanism must be specified and reviewed to ensure that a potential failure will not go undetected.

Some checks are implemented as assertions.

Some failure modes will be easier to detect using assertions on the design itself. Checks that should be formally verified must also be implemented using assertions. Many are implementation-specific failure modes, such as buffer overflows, and should be coded directly in the RTL. Others will be to check for signal-level relationships on interface signals. These can be coded in the *interface* declaration containing the signals in question.

Many are implemented as testbench code.

Failure modes dealing with higher level errors, such as data transformation, ordering or computations, are better detected behaviorally in the testbench itself. The expressiveness of the property language does not really lend itself well for high-level or end-to-end response checking.

It will be the most complex portion of your testbench.

After the completion of a project, you will find that the largest, most complex component of the testbenches is the self-checking structure. It will have been the portion that required the most authoring and maintenance effort. The self-checking structure is also the most critical portion as it is responsible for declaring the functional correctness of the design. It will embody a duplication of the specified functionality of the design under verification.

Why is this section so short?

If the self-checking structure is the most complex and largest portion of a verification project, why is it such a small portion of this book? That's because the bulk of the functionality in the self-checking structure will be to model the expected functionality of the design under verification. That is unique to every design and cannot be described in a generic fashion in a book. Each class of function requires different approaches and different mechanisms for identifying failures. Each class of design could be the topic of its own book.

General implementation techniques are presented.

This section presents various techniques for implementing the self-checking structure. Which one to use depends on the class of design under verification. The techniques can also be used in combination. Some techniques depend on the availability of reference models. Others rely on the availability of unmodified data payloads. Once you have specified the requirements of the self-checking structure, use the necessary techniques to implement them.

See the VMM.

The section titled "Self-Checking Structures" starting on page 246 of the *Verification Methodology Manual for SystemVerilog* describe similar and additional self-checking techniques and provides guidelines for implementing them.

Hard Coded Response

Some techniques require hard-coded stimulus and configuration.

The self-checking strategy used to verify the muxed flip-flop in "Self-Checking Testbenches" on page 221 relied on hard coded response checking. The function and configuration of the device under verification was very simple. The response could be checked for each individual input value. To hardcode a response in a testbench requires a known configuration and a known input stream. It is therefore only applicable to directed testcases. Sample 6-15 shows the pseudo-code for a directed testcase with a hardcoded response on the ATM switch node design. The objective of this testcase is to verify that cells from every input port can be switched to every output port.

It must be replicated for each testcase.

Each testcase is supposed to verify a different feature of the design. Each testcase needs a different configuration or a different input data stream. Each testcase will thus yield a different response. If a hard-coded response strategy is used, it will be necessary to replicate the response checking in each testbench.

Errors can slip through easily.

Because the response being checked is crafted to the testcase, it tends to ignore other potential problems. It is assumed that the other functions operate correctly and that any problem would be caught by the testcase targeting those functions. Should an unexpected correlation or corner case exist, it will likely go undetected if it is accidentally created in a testcase that focuses on different features.

Sample 6-15. Pseudo-code for hardcoded response

```
Program configuration:
   for x in 0..3:
      vpi x -> output port #x
for out_port in 0..3:
   fork
   {
      for in_port in 0..3:
         generate atm cell with:
            vpi == out_port;
            vci == in_port;
         send cell on port(in_port);
   }
   {
      for in_port in 0..3:
         wait for cell on port(out_port);
         assert cell.vpi == out_port;
         assert cell.vci == in_port;
   }
   join
```

Data Tagging

Transactions must have untouched data fields.

Many designs use some of the input information for processing, sometimes transforming it, but leave other portions of the input untouched and forward it, intact, all the way through the design to an output. Examples abound in the datacom industry. They include ethernet hubs, IP routers, ATM switches and SONET framers.

Use the untouched fields to encode the expected tranformation.

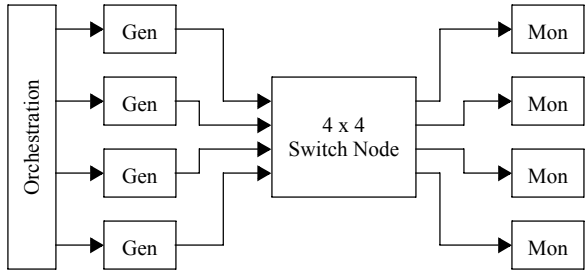
The portion of the data input that passes untouched through the design under verification can be put to good use. It is often called *payload* and the term *packet* or *frame* often is used to describe the unit of data processed by the design. You must first determine, through a proper check, that the payload information is indeed not modified by the design. Subsequently, the payload information can be used to describe the expected destination, position and transformation for this packet. For each packet received, the output monitor uses the information in the payload to determine if the packet was processed appropriately.

This simplifies the testbench control structure.

This self-checking strategy usually lends itself to the simplest self-checking structures. All of the intelligence is located in independent output monitors. The control of this type of testbench is simple because all the processing (stimulus and specification of the expected response) is performed in a single location: the stimulus generator. Some minor orchestration between the generators may

be required in some testcases when it is necessary to synchronize traffic patterns to create interesting scenarios. Figure 6-5 shows the structure of a testbench using data tagging to verify the example switch node design example.

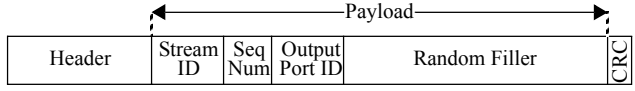
Figure 6-5. Testbench structure for the switch node design



Include all necessary information in the payload to determine functional correctness.

The payload must contain all necessary information to determine if a particular packet came out of the appropriate output, in the proper sequence and with the appropriate transformation of its control information. For example, assume the success criteria is that the packets for a given input stream be received in the proper order by the proper output port. The payload should contain a unique stream identifier, a sequence number and an output port identifier, as shown in Figure 6-6.

Figure 6-6. Example packet payload structure



The output monitor needs to verify that the output identifier matches its own identifier. It also needs to verify that the sequence number is equal to the previously received sequence number in that stream plus one, as outlined in Sample 6-16. A CRC value is used to verify that the payload was indeed not modified by the design.

Use data tagging in collaboration with scoreboarding.

Should it be possible for a packet to have a payload too short to contain all of the tag information, another self-checking strategy—such as scoreboarding—must be used in concert with data tagging. When present in the payload, the tag information is used by the output monitor to quickly search the scoreboard to confirm correctness of the received object. When not available, the scoreboard is

Sample 6-16. Implementation using payload information to determine functional correctness

```
forever begin
   atm_cell cell;

   th.uL1_0.receive(cell);
   // Cell was corrupted?
   if (cell.payload[47] !== cell.payload_crc())
   begin
      ...
      continue;
   end
   // Cell is for this port?
   if (cell.payload[0] !== my_id) ...
   // Packet in correct sequence?
   if (last_seq[cell.payload[1]] + 1 !=
       {cell.payload[2], cell.payload[3]}) ...
   // Reset sequence number
   last_seq[cell.payload[1]] =
      {cell.payload[2], cell.payload[3]};
end
```

searched normally, to ensure that the received object is indeed expected. For more details on scoreboarding, see "Scoreboarding" on page 300.

Cannot be used within system-level context.

Data tagging assumes that some portion of the stimulus is available to be used without interference from the design under verification. This may be true for a stand-alone design block. But every payload has an ultimate system-level purpose. Once put in a system, the payload fields may be used to carry higher-level information. For example, the payload bytes in an ethernet frame may carry IP packets. If you intend to reuse your block-level self-checking structure at the system-level, data tagging may not be an appropriate strategy.

Reference Models

You can use a reference model.

As illustrated in Figure 6-7, the reference model and the design under verification are subjected to the same stimulus and their output is monitored and compared for discrepancies constantly.

Reference models rarely exist.

The problem with this strategy is that reference models rarely exist. Reference models are available only during a re-design exercise where backward compatibility is required and when they form an integral part of the specification. Pure backward compatible re-designs are rare as the re-design is often used as an opportunity to increase performance, add to the number of ports or add new fea-

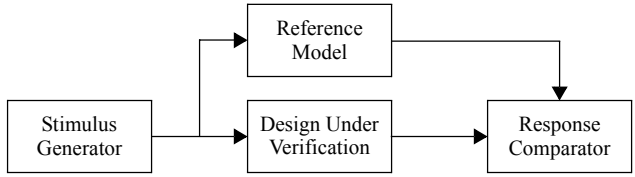


Figure 6-7. Using a reference model to predict output

tures. That only leaves reference models that exist as part of the specification.

It is a popular strategy for processors.

Some classes of designs are not fully specified on paper. Rather, they are specified using an executable model that was used to explore architectural and performance trade-offs. Because the model is the specification, it is golden by definition. It is the typical approach used for general-purpose, digital signal and graphics processors.

The model need not run concurrently.

Often, the difficulty of integrating the reference model with the design simulation prevents it from being simulated concurrently with the design. The output is thus compared in a post-processing step, as illustrated in Figure 6-8. The input can be generated externally similarly when the reference model includes a suitable data generator, as depicted in Figure 6-9.

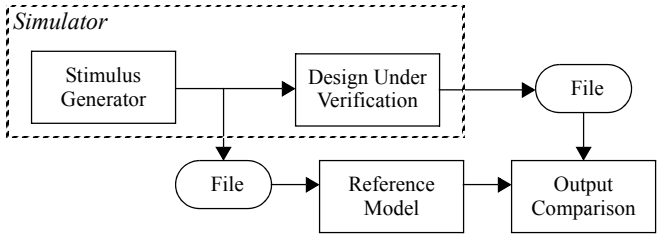


Figure 6-8. External output comparison

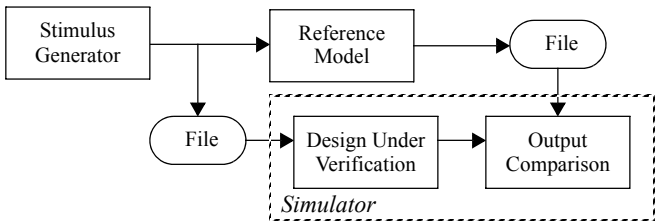


Figure 6-9. External input generation

It is a force behind C-based verification.

The fact that these reference models are usually implemented in C or C++ is a force behind using C or C++ as a simulation language for the design and verification. It is thought that by using a common language the design and verification can proceed smoothly from system-level and architectural-specification down to detailed implementation.

SystemVerilog can interface with C models

SystemVerilog has a well-defined interface mechanism to C/C++ models. It is possible to develop a testbench in SystemVerilog that uses a reference model written in C or C++. The technicalities of integrating such models are beyond the scope of this book.

Rerefence models can be written in SystemVerilog too!

Reference models are often transaction-level models written as part of the architectural exploration of the design specification. It is often used to make trade-offs between hardware and software functions and analyze the overall performance of the system. Because this is the area where SystemC is well suited, there is a belief that these models must therefore be written in SystemC. That may very well be the case—but that need not be the case. Chapter 7 introduces transaction-level models using SystemVerilog and the techniques presented can be used to create reference models.

Transfer Function

Model the data transformation.

A transfer function is used to reproduce any data transformation performed by the design to determine which output value to expect, as illustrated in Figure 6-10. The transfer function is often used in concert with a scoreboard (see “Scoreboarding” on page 300). The transfer function uses the design configuration descriptor to perform the same transformation. Data transformation is not limited to computation and modification of fields and values inside each data item. Data transformation also includes the generation of new data items (for example IP segments from a TCP packet), the identification of ordering and destination for the data item and computation of the next state of the design when executing an instruction.

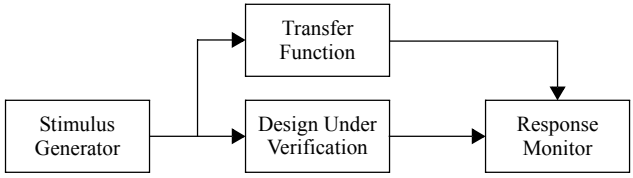


Figure 6-10. Reproducing the transformation to predict output

It's not the same as a reference model.

Isn't a transfer function the same thing as a reference model? Figure 6-10 sure looks like Figure 6-7![1] The answer is no, for several reasons. First, a transfer function does not exist a priori. Second, it is not golden by definition. As simulations will be run and errors reported, there will be as many errors in the transfer function as in the design itself. Third, a transfer function model does not predict the response as accurately as a reference model. When a packet or an instruction is sent from the stimulus generator to the transfer function, it is transformed then stored into the scoreboard or forwarded to the response monitor. The response monitor will have to perform more intelligent checks to deal with the uncertainty in the transfer function when checking the observed response. With a reference model, the response checking is a simple observed-against-expected comparison process.

Scoreboarding

A scoreboard is a data structure.

The definition of scoreboard is definitely not standardized across the industry. For some, it is the entire self-checking structure, including the transfer function or reference model, the expected data storage mechanism and the output comparison function. In this book, the definition of scoreboard is limited to the data structure used to hold the expected data for ease of comparison against the monitored output values.

A scoreboard holds expected data.

A scoreboard is a data structure that holds data expected to be received by the output monitor. As illustrated in Figure 6-11, the transfer function adds data to the scoreboard. Any data received by the output monitor is compared against the data in the scoreboard. If an identical data item is found, the design produced the expected response. If an identical data item is not found in the scoreboard, an error is reported. At the end of the simulation, any data items left in the scoreboard were lost in the design—which may or may not be an error.

The data structure depends on the self-checking requirements.

Just as there is no single definition of scoreboard, there isn't a single scoreboard kind or structure. Each scoreboard is designed to meet the needs of the self-checking requirements. Some scoreboards are simply scalar variables holding just one data item at a

1. In fact, one was cut and pasted from the other!

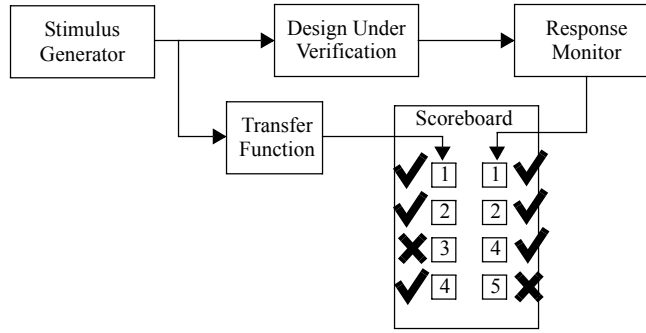


Figure 6-11. Scoreboarding

time. Some scoreboards are arrays of queues of data items. A scoreboard may be centralized into a single data structure or it may be distributed in the comparison functions attached to each output monitor. The self-checking structure of a testbench may be composed of a single scoreboard or of a series of scoreboards daisy-chained into one another.

Use a queue if ordering is important.

If the design is supposed to maintain the original order of the input data stream, the scoreboard is usually implemented using a *queue*. The output produced by a functionally correct design would be found, in order, in the queue. If multiple data streams are multiplexed onto a single data stream with no ordering relationship between them, use one queue per input stream. Any data item received from the design must be found at the head of one of the queues. The scoreboard for the example ATM switch node design would be composed of four sets of four queues: one set per output port, one queue per input port. An ATM cell would be added to the appropriate queue based on the input port where it is injected and the expected destination ports.

Optimize the look-up function.

When a new data item is received from the design by an output monitor, it must be compared against a data item in the scoreboard. For simple designs, it may be necessary only to look at the data item at the head of a very specific queue. For more complex designs where data losses are possible or ordering is difficult to predict, output data may come in an apparently random order. It is therefore important to make the look-up operation as efficient as possible to identify the output data as valid or not quickly. If you have to search through all of the data items in the entire scoreboard, simulations

will take forever to run. Associative arrays can be useful to minimize the cost of look-up operations.

Refer to a data structure book.

Designing a scoreboard is about designing a suitable data structure that will meet the self-checking requirements of the design. It has to be efficient, both in terms of runtime and storage. A scoreboard that is expected to hold thousands of very large packets must be given a lot of careful attention. You have to watch out for memory leaks as objects are discarded after being compared. It would be pointless for me to describe in this book what has been the object of several other books. Lists, hash functions, circular buffers, lookup tables, queues, indexing strategies and the like have already been described better than I ever could. I recommend you look up the computer science section of your local technical bookstore for a textbook on data structures.

See the VMM.

The section titled "Scoreboarding" starting on page 249 of the *Verification Methodology Manual for SystemVerilog* specifies guidelines for implementing scoreboards.

Integration with the Transaction Layer

The self-checking structure must be visible globally.

The self-checking structure must be accessible from almost every component of the testbench. The configuration generator needs to pass the device configuration descriptor to it. The testcases and stimulus generators need to forward generated input data to it. The bus-functional models need to inform the self-checking structure of any unexpected events that occurred during the data transmission. Output monitors must indicate that new output data has been received to verify its correctness.

Encapsulate the self-checking structure.

Whatever strategy is used to make the self-checking structure visible to all components of the testbench, it will be easier if it is encapsulated as a single object. I prefer to use a *class* as it will be possible to pass references to the self-checking structure around if necessary while making global hierarchical references still possible through a well-known reference. Provide a nonblocking transaction-level interface that will be used to notify the self-checking structure of new data being injected or received. Sample 6-17 shows the defini-

tion of a possible self-checking structure for the example ATM switch node.

Sample 6-17. Definition of the self-checking structure for the ATM switch node

```
class self_check;
   cfg: device_cfg;

   extern function void sent(atm_cell cell,
                             int      on_port;
   extern function void
      received(atm_cell cell,
               int      on_port);
endclass: self_check
```

Put a reference to the self-checking structure in a global variable.

One way to make the self-checking structure visible to all components of the testbench is to have a reference to its instance in a global variable. Since everything is globally accessible in SystemVerilog, the question simply becomes on deciding where to put it and under what name. Personally, I like to instantiate it in the harness, under the name *sb* or *sc*, as shown in Sample 6-18.

Sample 6-18. Self-checking structure instance.

```
class harness;
   ...
   self_check sb;
   ...
   function new();
      this.sb = new;
   endfunction: new
   ...
   task configure();
      this.sb.cfg = this.cfg;
      this.cfg.apply(this.cpu);
   endtask: configure
endclass: harness
```

For each transaction on the design, notify the scoreboard.

Once the self-checking structure is globally visible, it is simple to implement monitoring threads in the verification harness or extend the callbacks in the bus-functional models to invoke the proper transaction-level procedures in the self-checking structure at the proper time. Sample 6-19 shows how to complete the integration of the self-checking structure with the transaction-layer verification harness.

See the VMM.

Guidelines 5-43 through 5-48 and the section titled “Integration with the Transactors” starting on page 253 of the *Verification Meth-*

Sample 6-19. Integrating the self-checking structure in the verification harness

```
fork
   forever begin
      atm_cell cell;
      this.atm0.receive(cell);
      this.sb.received(cell, 0);
   end
join_none
```

odology Manual for SystemVerilog specifies guidelines for encapsulating and integrating scoreboards in a verification environment.

DIRECTED STIMULUS

Stimulus is hardcoded.

Directed stimulus is specified in the verification plan and hardcoded for each testcase. Executing a testcase requires simulating the testbench that includes the directed stimulus for that testcase. It is used to implement each directed testcase specified using the approach defined in “Directed Testbenches Approach” on page 96. Sample 6-20 shows a directed testcase used to debug a CPU interface: It “generates” a write cycle followed by a read cycle at the same address and verifies that the readback value is correct.

Sample 6-20. Directed stimulus

```
program simple;

harness th = new;

initial
begin
   bit [15:0] actual;

   th.cpu.write(16'h00FF, 16'hABCD);
   th.cpu.read(16'h00FF, actual);
   if (actual !== 16'hABCD) ...
   $finish;
end
endprogram: simple
```

Procedural interfaces imply directed tests.

A procedural transaction-level interface, like the one used to write the testcase in Sample 6-20, usually imply directed testcases and make creating random stimulus more difficult. A dataflow transaction-level interface, on the other hand, makes writing random stimulus much easier (see “Random Stimulus” on page 307) but appears to make writing directed stimulus impossible. Not true. In fact,

dataflow interfaces support directed stimulus just as well as procedural interfaces. The difference is that a directed transaction descriptor must be created before calling the injection procedure. Sample 6-21 shows the same testcase as Sample 6-20, this time using a dataflow interface.

Sample 6-21. Directed dataflow-based stimulus

```
program simple;

harness th = new;

initial
begin
   cpu_trans tr = new;

   tr.kind = cpu_trans::WRITE;
   tr.addr = 16'h00FF;
   tr.data = 16'hABCD;
   th.cpu.inbox.put(tr);
   @(tr.done);
   th.cpu.inbox.put(tr);
   @(tr.done);
   if (tr.data !== 16'hABCD) ...
end
endprogram: simple
```

Can include random filling.

Directed stimulus need not be specified 100 percent. The part that is coded explicitly usually only pertains to the testcase being implemented. The data that is deemed irrelevant for this testcase is usually filled with random—but valid—values. For example, the content of an ethernet frame could be filled with random values, except for the VLAN label that is the objective of the testcase. The sequence of VLAN label values would be hardcoded in the testcase while the remaining data fields would be generated randomly. Random filling works best when using transaction descriptors and dataflow transaction-level interfaces. Sample 6-22 shows an example of directed stimulus for an instruction stream. The content of the operands is randomized while the actual sequence of opcodes is hardcoded.

Other streams can be generated randomly.

Directed testcases often concentrate on a single data stream when creating a stimulus sequence. The other streams are left idle or can be generated randomly. In the ATM switch node example, directed stimulus can be specified for the cell stream on port #0 while random traffic is injected in the other ports, as shown in Sample 6-23.

Sample 6-22. Random-filled directed instruction sequence stimulus

```
instr.randomize() with {
   opcode == CMP;
};
...
repeat (3) begin
   instr.randomize() with {
      opcode == NOP;
   };
end
...
instr.randomize() with {
   opcode == BLT;
};
...
```

Sample 6-23. Random background stimulus

```
program test;

harness th = new();

initial
begin
   th.configure();
   fork
      bg_noise(1);
      bg_noise(2);
      bg_noise(3);
   join_none
   ...
   th.atm[0].inbox.put(cell);
   ...
end

task bg_noise(int on_port);
   atm_cell celll;

   forever begin
      cell = new;
      cell.randomize();
      th.atm[on_port].inbox.put(cell);
   end
endtask: bg_noise
endprogram: test
```

Directed stimulus implements testcases.

Even though some random stimulus was used, the nature of directed stimulus is always tied to a specific testcase. Each directed testbench can be tied to a specific testcase. It was written specifically to implement that testcase and no other. If there are one hundred directed testcases to write, there will be one hundred sets of

directed stimulus. Any random stimulus included in directed testcases is usually ignored in the response checking.

See the VMM.

The section titled "Directed Stimulus" starting on page 219 of the *Verification Methodology Manual for SystemVerilog* specifies guidelines for implementing directed stimulus in a random verification environment.

RANDOM STIMULUS

Generators create the stimulus.

Random stimulus is created by a random generator that can be constrained according to the requirement of the verification plan. Executing a testcase requires simulating the random testbench with a seed that will hit the functional coverage point that corresponds to the testcase. They are used to create automated verification environments specified using the approach defined in "Coverage-Driven Random-Based Approach" on page 101.

Random generation is more than calling *$random*.

Random stimulus generation in SystemVerilog has evolved far beyond random generation of individual scalar values using the *$random* system task. The generation components of a verification environment are designed to generate two subsets of all possible stimuli autonomously. The first is that subset that is legal, i.e., the possible inputs. The second is that subset of the first that is defined by the functional coverage models of the verification plan.

Atomic Generation

Generating a stream of random data is easy.

Sample 6-24 shows a simple random ATM cell generator. It should be encapsulated in a *class*, like a bus-functional model. In fact, generators are output-only bus-functional models. Their output is a transaction-level interface, the stream of generated random transactions. The same code could be used to generate CPU instructions, bus cycles, digitized signal samples or any other input data stream required by the design under verification.

Define termination mechanisms.

The simple random generator in Sample 6-24 will always generate 100 cells then terminate. This number is completely arbitrary and is unlikely to satisfy the needs of all testcases. During initial design debug stage, generating just a single data item is often required. A random testbench must run for much longer to increase the likelihood that functional coverage points will be hit. Generators should

Sample 6-24. Simple random generator

```
class atm_gen;
   ...
   function new();
      ...
      fork
         this.main();
      join_none
   endfunction: new

   task main();
      atm_cell cell;

      repeat (100) begin
         cell = new;
         if (!cell.randomize()) ...
         ...
      end
   endtask: main
endclass: atm_gen
```

have several termination mechanisms that can be armed at the start of the simulation (such as the number of objects to generate) or externally triggered by the testbench. Sample 6-25 shows a generator that, by default, will generate an infinite number of objects. It will also not start immediately, leaving time for the testbench to configure the device before starting to generate stimulus. The generator also can be suspended at anytime by calling the *stop()* method. Sample 6-26 shows how a debug testcase can configure the generator to generate a single cell on a randomly selected port and no cells on the others.

How to connect the generator to the BFM?

A random generator "bus-functional model" is like a monitor bus-functional model: it produces an output stream of transaction descriptors. In this case, they are not observed transactions but randomly generated transactions. This stream of transactions must be forwarded to the bus-functional model that will apply them to the design under verification. How are random generators integrated with the rest of the bus-functional models in the verification harness?

Can use a forwarding thread.

The simplest solution is to create an execution thread that will simply forward transactions from the generator to the bus-functional model. The forwarding thread can also perform any translation required between the transaction-level interface of the generator and the transaction-level interface of the bus-functional model. It

Sample 6-25. Random generator with termination mechanisms

```
class atm_gen;
   ...
   int stop_after = -1;

   function new();
      ...
   endfunction: new

   function void start();
      fork
         this.main();
      join_none
   endfunction: start

   function void stop();
      this.stop_after = 0;
   endfunction: stop

   virtual task main();
      atm_cell cell;

      while (this.stop_after != 0) begin
         cell = new;
         if (!cell.randomize()) ...
         ...
         if (this.stop_after > 0) begin
            this.stop_after--;
         end
      end
   endtask: main
endclass: atm_gen
```

Sample 6-26. Debug testcase injecting a single cell on random port

```
program test;

harness th = new;
initial
begin
   bit [1:0] the_one = $random;
   foreach (th.gen[i]) begin
      th.gen[i].stop_after = (i == the_one);
   end
   ...
end
endprogram
```

can also notify the self-checking structure of the transaction about to be executed. Sample 6-27 shows how a forwarding thread can move transaction descriptors from an output mailbox in the genera-

tor and translate them to a procedural interface in the bus-functional model. Notice how the output mailbox is created with a bound of one. This will ensure that the mailbox will fill up and cause the generation thread to block. The generation thread will resume once the mailbox is empty.

Sample 6-27. Forwarding thread to integrate generator and BFM.

```
class harness;
   cpu_trans_mbox  outbox;
   cpu_trans_gen   gen;
   utopia_mgmt_bfm cpu;
   ...
   function new();
      ...
      this.outbox = new(1);
      this.gen = new(this.outbox);
      fork
         forever begin: forwarding_thread
            cpu_trans tr;
            this.outbox.get(tr);
            case (tr.kind)
            cpu_trans::WRITE:
               this.cpu.write(tr.addr, tr.data);
            cpu_trans:READ:
               this.cpu.read(tr.addr, tr.data);
            endcase
         end
      join_none
   endfunction: new
   ...
endclass: harness
```

Can use shared mailboxes.

If the generate and bus-functional model it needs to connect to both use mailboxes and the same transaction descriptor, they can be directly connected to each other by sharing the same mailbox. As shown in Sample 6-28, the output mailbox of the generator is the input mailbox of the bus-functional model. With directly connected bus-functional models, callbacks must be used to tap the flow of transactions between the two bus-functional models and notify the scoreboard of the transaction being executed or observed.

Adding constraints is hard.

What if a testcase requires that a stream of cells with the same VPI value be injected? Or that only write cycles within a narrow address range be generated? Or samples with negative values? Or no branch instructions? Adding constraints to the simple generator shown in Sample 6-25 requires that the entire generation method be replaced, as shown in Sample 6-29. This results in a lot of duplicated code

Sample 6-28. Directly connected generator and BFM.

```
class harness;
   utopia_mgmt_trans_mbox gen_to_cpu;
   utopia_mgmt_trans_gen  gen;
   utopia_mgmt_bfm        cpu;
   ...
   function new();
      ...
      this.gen_to_cpu = new(1);
      this.gen = new(.outbox(this.gen_to_cpu));
      this.cpu = new(.inbox(this.gen_to_cpu));
   endfunction: new
   ...
endclass: harness
```

and a methodology similar to using directed stimulus. And the new generator would have to be substituted in the verification harness instead of the plain unconstrained generator. As illustrated in Figure 6-12, a different random generator would be created for each testcase. The idea behind the productivity cycle depicted in Figure 2-16 is to write just one generator that can be steered toward the uncovered functional coverage points by adding constraints with as little modifications as possible, as illustrated in Figure 6-13.

Sample 6-29. Replacing random generation method

```
class my_atm_gen extends atm_gen;
   task main();
      atm_cell cell;

      while (this.stop_after != 0) begin
         bit ok;
         cell = new;
         ok = cell.randomize() with {
            vpi == 0;
         };
         if (!ok) ...
         ...
         if (this.stop_after > 0) begin
            this.stop_after--;
         end
      end
   endtask: main
endclass: my_atm_gen
```

Figure 6-12. Different random generators

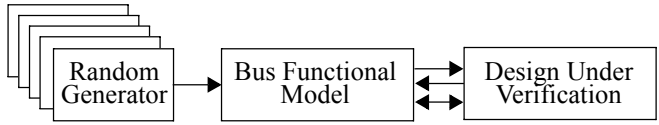
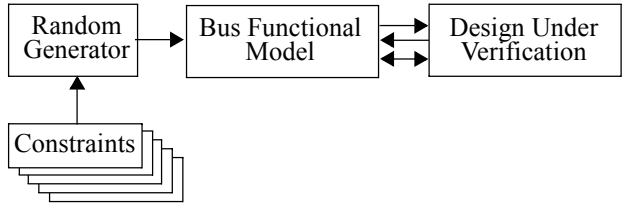


Figure 6-13. Constraining a single random generator



Adding Constraints

You can add constraints to the randomized type.

The simplest mechanism for adding constraints is to add them to the class being randomized. For example, the constraint forcing the generation of a stream of ATM cells with a VPI value equal to 0 can be added to the ATM cell class as shown in Sample 6-30. The problem with this approach is that the constraint will apply to every instance of the object in all future simulations and models. If the objective was to inject a specific condition for a specific testcase or simulation, this approach will not work.

Sample 6-30. Adding constraints to the generated class

```
class atm_cell;
   ...
   constraint vpi_is_0 {
      vpi == 0;
   }
endclass: atm_cell
```

You can turn constraints ON and OFF.

If a constraint is not supposed to apply at all times, it should be turned OFF in the constructor. It can then be turned ON only when required. For example, the constraint added to the ATM cell class in Sample 6-30 is turned OFF in Sample 6-31. The constraint will apply to a randomized instance only if it is explicitly turned ON.

Always randomize a public class property.

The problem with the simple generator in Sample 6-25 is that it randomizes a local variable to generate data items. The variable is not visible externally to allow the testcase to turn the relevant con-

Sample 6-31. Adding constraints turned OFF by default

```
class atm_cell;
   ...
   constraint vpi_is_0 {
      vpi == 0;
   }

   function new();
      this.vpi_is_0.constraint_mode(0);
   endfunction: new
endclass: atm_cell
```

straint blocks ON. By randomizing a public class property instead of a local variable, as shown in Sample 6-32, you can control the constraints in the randomized instance as shown in Sample 6-33. For a constraint to apply to a single stream, turn the relevant constraint block ON in the randomized class property for the generator instance of that stream.

Sample 6-32. Randomizing a public class property instead of a local variable

```
class atm_gen;
   ...
   atm_cell randomized_cell;
   ...
   virtual task main();
      while (this.stop_after != 0) begin
         this.randomized_cell = new;
         ok = this.randomized_cell.randomize();
         if (!ok) ...
         this.outbox.put(this.randomized_cell);
         if (this.stop_after > 0) begin
            this.stop_after--;
         end
      end
   endtask: main
endclass: atm_gen
```

Sample 6-33. Controlling constraints in randomized instance

```
program corner_case;

harness th = new;
initial
begin
   ...
   th.gen[0].randomized_cell.
      vpi_is_0.constraint_mode(1);
   ...
end
endprogram: corner_case
```

Always randomize the same instance.

There is still a problem with the generator in Sample 6-32: It keeps randomizing a different instance. The randomized instance keeps changing and must be controlled as shown Sample 6-33 before each and every randomization. That's too much work. Instead, randomize a single instance whose value is then copied into a new instance. These new instances will create the stream of generated data items, as shown in Sample 6-34.

Sample 6-34. Randomizing a single instance

```
class atm_gen;
   ...
   atm_cell randomized_cell;
   ...
   function new();
      this.randomized_cell = new;
   endfunction: new
   ...
   virtual task main();
      while (this.stop_after != 0) begin
         atm_cell cell;

         ok = this.randomized_cell.randomize();
         if (!ok) ...
         cell = this.randomized_cell.copy();
         this.outbox.put(cell);
         if (this.stop_after > 0) begin
            this.stop_after--;
         end
      end
   endtask: main
endclass: atm_gen
```

Add constraints to a derived class.

The constraint in Sample 6-31 was added to the original class modeling the ATM cell. If all of the constraints required to generate all of the input conditions required by all testcases to meet your coverage goals are added to that one class, it will soon become unmanageable. Furthermore, a generic model, such as an ATM cell, can be reused across projects. It should not be polluted with project or testcase-specific additions. Constraints should be added in a derived class as shown in Sample 6-35. Use a different extension for each testcase.

Replace the randomized instance with an instance of the derived class.

This technique does not appear to be helpful, as the generator is still making use of the base class, not the derived class in Sample 6-35. Therefore the new constraints are not used. One solution would be to change the generator to use an instance of the derived class, but

Sample 6-35. Adding constraints in a derived class

```
class constrained_atm_cell extends atm_cell;
   constraint vpi_is_0 {
      vpi == 8'h00;
   }
endclass: constrained_atm_cell
```

you'll end-up modifying the generator for each constraint set. Remember that a derived class is compatible with its base class and that the *randomize()* method is a virtual method. We can simply sneak an instance of the derived class in lieu of the original randomized instance, and the generator won't even be aware that it is now generating a data stream subject to additional constraints! Sample 6-36 shows how to do so for a specific instance of a generator.

Sample 6-36. Adding constraints via a derived class.

```
class constrained_atm_cell extends atm_cell;
   ...
endclass: constrained_atm_cell

program corner_case;

harness th = new;
initial
begin
   constrained_atm_cell cell = new;
   th.gen[0].randomized_cell = cell;
   ...
end
endprogram: corner_case
```

Extend *pre_randomize*() or *post_randomize*(), as needed.

Constraints are powerful, but sometimes cannot express a particular condition that needs to be generated. You can execute procedural code before or after the randomization process by extending the predefined *pre_randomize*() and *post_randomize*() methods. You could use *pre_randomize*() to initialize some non-randomized fields or call *constraint_mode()* or *rand_mode()*. You could use *post_randomize*() to compute or corrupt CRC values, as illustrated in Sample 6-37. CRC values must be computed—and thus corrupted—in the *post_randomize*() method because method calls should not be used in constraint expressions[1]. When overloading the *pre_randomize*() or *post_randomize*() methods, do not forget to invoke their original version in the parent class using *super.pre_randomize*() or *super.post_randomize*(). This approach

will ensure that any procedural operations required to randomize the parent class successfully are executed.

Sample 6-37. Randomly corrupted HEC value

```
class may_be_bad_atm_cell extends atm_cell;
   rand bit is_bad;
   function void post_randomize();
      super.post_randomize();
      if (is_bad) begin
         bit [7:0] hec = $random;
         while (hec == this.hec) begin
            hec = $random;
         end
      end
   endfunction: post_randomize
endclass: may_be_bad_atm_cell
```

See the VMM.

The sections titled "Random Stimulus" starting on page 213 and "Atomic Generation" starting on page 231 of the *Verification Methodology Manual for SystemVerilog* specifies guidelines for implementing highly constrainable and reusable stimulus generators and using the predefined *vmm_atomic_gen*.

Constraining Sequences

Generating atomic elements is not interesting.

In the previous section, data items were generated randomly, independent of each other. This technique is going to create some interesting conditions but is unlikely to generate all of the conditions required to meet your functional coverage goals. The design under verification has data and temporal behavior, each of which must be verified. The temporal properties of applied stimuli must be as flexible and diverse as the data properties to verify the temporal behavior easily. You must include a mechanism that will make it possible to express constraints describing a sequence of data items that will exercise the temporal features of the design.

Provide a unique data item identifier.

It is possible to express stream-specific constraints using a unique stream identifier in a conditional expression. The same mechanism can be used to specify constraints applicable to data items at a spe-

1. Because methods are usually invoked without arguments and usually make use of random variables in the class being randomized, the solver would not know which variables to solve before calling the method. Thus the method may be called with unsolved-for variable values.

cific index within a sequence. The random generator from Sample 6-34 has been modified in Sample 6-38 to increment an object index after each random generation. Constraints specific to the position of the object within the sequence can then be specified using that unique object identifier, as shown in Sample 6-39.

Sample 6-38. Generating constrainable sequences

```
class atm_gen;
   ...
   int cell_idx = 0;
   atm_cell randomized_cell;

   virtual task main();
      while (this.stop_after != 0) begin
         ...
         this.randomized_cell.cell_idx =
            cell_idx++;
         ok = this.randomized_cell.randomize();
         ...
      end
   endtask: main
endclass: atm_gen
```

Sample 6-39. Specifying sequence-specific constraints

```
class constrained_atm_cell extends atm_cell;
   constraint rotating_vpi {
      vpi == cell_idx % 4;
   }
endclass: constrained_atm_cell
```

Randomize an array.

Using a unique object identifier allows specifying sequence-specific constraints. But they can be specified only as independent values. It is not possible to express constraints that refer to previously or subsequently generated values. For example, how would you generate a sequence of ATM cells with random VPI values with no two consecutive identical values? The solution is to randomize an array instead of a single object. Array constraints can refer to any elements in the array. Sample 6-40 shows the previous generator, modified to generate an array instead of a single object. Sample 6-41 shows how to add constraints to avoid generating two identical consecutive VPI values.

How long is a sequence?

You may have noticed that, in Sample 6-40, the sequence is implemented using a dynamic array. Presumably, the length of the array determines the length of the sequence. Using a dynamic array thus allows for variable-length sequences. But how long is a sequence?

Sample 6-40. Generating sequences using an array

```
class atm_cell_seq;
   rand atm_cell randomized_cells[];
endclass: atm_cell_seq

class atm_gen;
   ...
   atm_cell_seq randomized_seq;

   virtual task main();
      while (this.stop_after != 0) begin
         ...
         ok = this.randomized_seq.randomize();
         ...
      end
   endtask: main
endclass: atm_gen
```

Sample 6-41. Specifying sequence-specific array constraints

```
class my_atm_cell_seq extends atm_cell_seq;
   constraint not_conseq_vpi {
      foreach (cells[i]) {
         if (i > 0) {
            cells[i].vpi != cells[i-1].vpi;
         }
      }
   }
endclass: my_atm_cell_seq
```

How is the size of the dynamic array determined? Can the length of a sequence be random as well?

Randomize for the longest possible sequence.

In SystemVerilog, randomization and memory allocation are two different things. Memory is allocated first—either statically or via constructors—then the content of the allocated memory is randomized. This means that the size of a dynamic array must be decided *before* it is randomized. This implies that the dynamic array must be as long as the longest possible sequence. To implement random-length sequences, simply randomize a *length* class property, constrained to be less than or equal to the length of the dynamic array, as shown in Sample 6-42.

Define a sequence for a trivial testcase.

During a consulting engagement, I had spent several days helping a customer write a random-based self-checking environment to verify a CPU interface on an RTL design. After explaining and implementing the concepts shown in this section, the engineer I was working with interjected, “But the first testcase I’ll want to run,

Sample 6-42. Random-length sequences.

```
class atm_cell_seq;
   rand atm_cell randomized_cells[];
   rand int length;

   constraint random_length {|
      length > 0;
      length <= randomized_cells.size();
   }
endclass: atm_cell_seq

class atm_gen;
   ...
   atm_cell_seq rand_seq;

   virtual task main();
      while (this.stop_after != 0) begin
         ...
         ok = this.rand_seq.randomize();
         for (int i = 0;
               i <= this.rand_seq.length;
               i++) begin
            cell = this.rand_seq.cells[i].copy();
            ...
         end
         ...
      end
   endtask: main
endclass: atm_gen
```

when the RTL is delivered tomorrow, is a simple *write* followed by a *read*. I don't need this fancy generation and constraint mechanism yet." I replied that his first testcase was simply a very simple random sequence: Constrain the sequence of transactions to a length of two, the first transaction to be a *write* cycle and the second transaction to be a *read* cycle at the same address as the previous one. Instead of writing a separate testcase, only a few additional lines creating a simple sequence were sufficient. Once the initial debug of the design was over, these constraints were removed, subjecting the RTL code to a lot of different input sequences with no additional testbench development effort. The sequence constraints can be found in Sample 6-43.

Need to define multiple sequences.

The sequences defined using the constraint mechanism shown in this section would be generated over and over by the generator. Each simulation would generate the same sequence in each generator. It is more efficient to have multiple interesting sequences be

Sample 6-43. Defining a first-test sequence

```
class trivial_seq extends cpu_trans_seq;
   constraint trivial {
      length == 2;
      trans[0].kind == WRITE;
      trans[1].kind == READ;
      trans[0].addr == trans[1].addr;
   }
endclass: trivial_seq
```

generated randomly, one after another in a single simulation. Different instances of the same generator could generate the same sequence at the same time or generate different sequences.

Define scenarios to increase functional coverage.

Left unconstrained, random generators will generate valid but most likely uninteresting input sequences. By defining scenarios, generators will be constrained to generate a series of constrained sequences, focused on interesting cases. One scenario is usually the "random" sequence i.e., no constraints at all. As holes in functional coverage are identified, scenarios are added to the verification environment to steer the generators toward the uncovered areas of the solution space.

Scenarios can define directed testcases.

If you have the necessary control variables, it is possible to specify a directed testcase as a series of constraints. By specifying a set of constraints for which there is only one solution, you have created a scenario that implements a directed testcase. If scenarios can also be defined procedurally, a directed testcase can also be described as a procedural scenario. A directed testcase then becomes one possible scenario amongst all possible scenarios. It could be randomly generated as part of a longer stream of random scenarios.

Defining Random Scenarios

Randomize a *kind* class property.

To be able to generate multiple scenarios one after the other, it is necessary to randomize the array differently, according to different scenario constraints. Each randomization of the array defines a scenario. The generator in Sample 6-42 is modified in Sample 6-44 to include a random *kind* class property. This randomly-selected property selects the scenario that will be generated. Scenarios are defined in individual constraint blocks, conditionally based on the value of the *kind* class property.

Sample 6-44. Generating scenarios

```
class atm_cell_seq;
   typedef enum {ATOMIC} scenario_name;
   rand scenario_name kind;
   rand atm_cell randomized_cells[];
   rand int length;

   constraint random_length {
      length > 0;
      length <= randomized_cells.size();
   }

   constraint atomic_scenario {
      if (kind == ATOMIC) length == 1;
   }
   ...
endclass: atm_cell_seq
```

Define a new scenario by adding an enumeral and a constraint block.

The default *atomic* scenario simply generates a single random cell. To create new scenarios representing interesting sequences of transactions, add a new enumeral to the enumerated type identifying the scenario, then specify the scenario constraints in a separate constraint block, as shown in Sample 6-45.

Sample 6-45. Defining new scenarios

```
class atm_cell_seq;
   typedef enum {ATOMIC, SAME_VPI} scenario_name;
   ...
   constraint same_vpi {
      if (kind == SAME_VPI) {
         foreach (cells[i]) {
            if (i > 0) {
               cells[i].vpi == cells[i-1].vpi;
            }
         }
      }
   }
   ...
endclass: atm_cell_seq
```

Testcases can control distribution.

Specific testcases may want to control the distribution or selection of the randomly generated scenarios. Some testcases may want to restrict the stimulus to a particular scenario. Others may want to favor some scenarios over others. This testcase specific distribution can be created by deriving a testcase-specific scenario definition class with additional distribution constraints on the *kind* class property and substituted in the randomized sequence instance, as shown in Sample 6-46.

Sample 6-46. Modifying scenario distribution.

```
class my_scenarios extends atm_cell_seq;
   kind dist {1 :/ ATOMIC; 10 :/ SAME_VPI};
endclass: my_scenarios

program test;
harness th = new;

initial
begin
   my_scenarios sc = new;
   th.gen[0].rand_seq = sc;
   ...
end
endprogram: test
```

Defining Procedural Scenarios

Some scenarios are easier to define procedurally.

Sometimes scenarios are easier to define procedurally than with constraints. For example, generating one thousand times the same transaction is easier and more efficient to accomplish by generating a single transaction that is then repeated one thousand times using a *repeat* statement. Another example involves waiting for some condition in the design. Constraints are solved in zero-time. They cannot be solved for a few transactions, then wait for a FIFO to fill up, then solved for the remaining transactions. But a procedural definition can. It can generate and apply stimulus while a FIFO is not full, then switch to different stimulus once the FIFO is detected as full.

Use a *virtual method*.

By default, a random scenario is applied through a bus-functional model as fast as possible. If that default behavior is implemented in a *virtual task*, that behavior can be modified for some or all scenarios. Sample 6-47 shows the sequence generator using an *apply()* virtual task to execute a scenario. This *apply()* method can be used to specify procedural scenarios, as shown in Sample 6-48. Note how procedural scenarios are defined using the randomly selected *kind* class property. This allows random scenarios to be randomly intermixed with other procedural or random scenarios.

Use the *randsequence* statement.

Scenarios can also be described using the sequence generator. A sequence generator ruleset is defined using the *randsequence* statement. A scenario is described as a *production rule*, often making use of other production rules. Thus scenarios can be described in terms of other scenarios. A second type of production rule describes a weighted choice between equivalent scenarios. For someone with

Sample 6-47. Executing scenarios via a *virtual method*.

```
class atm_cell_seq;
   ...
   virtual task apply(atm_cell_mbox outbox);
      for (int i = 0; i < this.length; i++) begin
         outbox.put(this.cells[i].copy());
      end
   endtask: apply
endclass: atm_cell_seq

class atm_gen;
   ...
   atm_cell_seq rand_seq;

   virtual task main();
      while (this.stop_after != 0) begin
         ...
         ok = this.rand_seq.randomize();
         this.rand_seq.apply(this.outbox);
         ...
      end
   endtask: main
endclass: atm_gen
```

Sample 6-48. Defining a procedural scenario.

```
class atm_cell_seq;
   typedef enum {ATOMIC, SAME_VPI,
                 REPEAT_100} scenario_name;
   ...
   virtual task apply(atm_cell_mbox outbox);
      case (kind)
      REPEAT_100: begin
         this.cells[0].randomize();
         repeat(100) begin
            outbox.put(this.cells[0].copy());
         end
      end
      default: super.apply(outbox);
      endcase
   endtask: apply
endclass: atm_cell_seq
```

an RTL background, the reverse-YACC syntax of the sequence generator is really bizarre. But once you realize it is a simple front-end to small subprograms and the *randcase* statement, it becomes easy to understand.

A production rule is like a task definition.

A production rule is similar to defining a small task or function. When invoked, it will “execute” its definition. For example, the

ruleset in Sample 6-49 is functionally equivalent to defining the tasks in Sample 6-50. A rule can return values and be passed arguments. Refer to section 12.16 of the SystemVerilog Language Reference Manual for a complete description of the sequence generator.

Sample 6-49. Sequence generator ruleset

```
virtual task apply(atm_cell_mbox outbox);
   case (kind)
   SEQGEN:
      randsequence (SCENARIOS) {
         SCENARIOS: RANDOM_CELL := 1
                  | SAME_VPI := 2;

         RANDOM_CELL: {
            this.cells[0].randomize();
            outbox(this.cells[0].copy());
         }

         SAME_VPI:
            RANDOM_CELL
            {
               this.cells[0].vpi.rand_mode(0);
            }
            repeat (this.length-1) RANDOM_CELL
            {
               this.cells[0].vpi.rand_mode(1);
            }
      }
   ...
   endcase
endtask: apply
```

The choice weights can be expressions.

The weights assigned to the different choices in an alternative rule can be expressions, just like the weights in the equivalent *randcase* statement. By making these weights public properties of the sequence descriptor, each simulation run can pick and choose which scenarios will be enabled and the probability a particular scenario will be generated. For example, in Sample 6-51, the simulation run will only use the “debug testcase” scenario in the descriptor shown in Sample 6-52.

Do not create random-length sequences using recursive rules.

Rules can be recursive. For example, Sample 6-53 shows a ruleset that creates a sequence of ATM cells. The number of cells in the sequence is random, determined by the number of times the *CELL_STREAM* rule is selected over the *RANDOM_CELL* rule. This style works just fine except for one thing: The length of the

Sample 6-50. Equivalent task definitions

```
virtual task apply(atm_cell_mbox outbox);
   task SCENARIOS();
      randcase
      1: RANDOM_CELL();
      2: SAME_VPI();
      endcase
   endtask: SCENARIOS

   task RANDOM_CELL();
      this.cells[0].randomize();
      outbox.put(this.cells[0].copy());
   endtask: RANDOM_CELL

   task SAME_VPI();
      RANDOM_CELL();
      this.cells[0].vpi.rand_mode(0);
      repeat (this.length-1) RANDOM_CELL();
      this.cells[0].vpi.rand_mode(1);
   endtask: SAME_VPI

   case (kind)
   SEQGEN: SCENARIOS();
   ...
   endcase
endtask: apply
```

Sample 6-51. Selecting scenarios

```
program initial_debug;
harness th = new;
initial
begin
   th.gen.null_weight = 0;
   ...
end
endprogram: initial_debug
```

sequence is not distributed evenly. The probability that the length of the sequence is one is 10 percent. The probability that it is two is 9 percent (0.9). The probability that it is three is 8.1 percent (0.1 x 0.9 x 0.9 x 0.1). The probability that it is N is $0.9^N/10$. Furthermore, the length of the sequence cannot be constrained other than by playing with the selection weights.

Generate the length first, then generate the sequence.

A better approach is to generate the length of the sequence first. That value can be subjected to constraints and have an even distribution. Once the length of the sequence is decided, you generate the sequence using a *repeat* statement. To that effect, the already-exist-

Sample 6-52. Variable weight scenario selection

```
class trans_seq
   trans trs[]
   ...
   integer null_weight = 1;
   integer debug_weight = 1;

   virtual task apply(trans_mbox outbox);
      case (kind)
      SEQGEN:
         randsequence (SCENARIOS) {
            SCENARIOS: NULL    := null_weight
                     | DBG_TST := debug_weight;

            ...
         }
      ...
      endcase
   endtask: apply
endclass: trans_seq
```

Sample 6-53. Recursive rules

```
CELL_STREAM: CELL_STREAM RANDOM_CELL  := 90
           | RANDOM_CELL              := 10;

RANDOM_CELL: {
   this.cells[0].randomize();
   outbox.put(this.cells[0].copy());
}
```

ing *length* class property in the sequence descriptor can be used, as shown in Sample 6-49.

Rulesets and rules cannot be virtual.

An important limitation of the sequence generator is that production rules and entire rulesets cannot be virtual. It is not possible to extend an existing generator using a sequence generator ruleset without modifying its source code. If you want to be able to add new production rules or modify existing rules or entire rulesets, I suggest you use the equivalent task style, as shown in Sample 6-50. If each task or function is defined as *virtual*, it will be simple to extend a ruleset by creating a derived generator class. Virtual tasks and functions can still make use of the sequence generator and the *randsequence* statement to describe their respective scenarios.

See the VMM.

The section titled “Scenario Generation” starting on page 232 of the *Verification Methodology Manual for SystemVerilog* specifies

Sample 6-54. Generating variable-length sequences

```
class seq_length;
   integer value;
   constraint valid {
      value > 0;
   }
   constraint reasonable {
      value < 50;
   }
endclass: seq_length

class atm_gen;
   seq_length len;
   ...
   task main();
      ...
      randseq (...) {
         ...
         CELL_STREAM: {
            void = this.len.randomize();
         } <repeat (this.len.value) RANDOM_CELL>;
         ...
      }
   endtask: main
endclass: atm_gen
```

guidelines for implementing highly flexible scenario generators and using the predefined *vmm_scenario_gen*.

SYSTEM-LEVEL VERIFICATION HARNESSES

More than one harness is needed.

A project is rarely composed of a single design block, requiring only one verification harness and one set of testbenches. A typical project involves multiple blocks tied together into a system. Each of those blocks is verified independently. The block integration into a system is then verified. Each block and system requires its own verification harness. When creating or procuring bus-functional models for a project, you must consider the needs of all harnesses, not just those for a single block.

Use system-level transactions.

What is a transaction at the block level is usually not a transaction at the system level. For example, a block may simply interact with an ethernet interface to recover ethernet frames. As far as the block is concerned, a transaction is an ethernet frame. But an entire system may extract the IP fragments from the payload of ethernet frames, reassemble these IP fragments into IP packets then perform

IP-layer routing functions. The IP packets are then segmented and encapsulated back into ethernet frames to be retransmitted. In such a system, a transaction is an entire IP packet, not an ethernet frame. The bus-functional models in a system-level harness must support system-level transactions.

Layered Bus-Functional Models

Bus-functional models should implement protocol layers.

Traditionally, bus-functional models were tied to a specific physical interface. As shown in Figure 6-14, a monolithic IP-packet bus-functional model would have accepted IP packets on its transaction-level interface side and produce physical signals on the other. This type of bus-functional model would not be able to reuse or provide any functionality that was required by the block-level harnesses. Each harness would require its own bus-functional model with its corresponding transaction-level abstraction. Similarly, the transaction-level behavior of monolithic bus-functional models could not be reused from or provided to other bus-functional models that implemented the same system-level functionality but on different physical-level interfaces. Instead, system-level bus-functional models should be layered according to the layers of the protocol they are implementing, as shown in Figure 6-15. The transaction-level interface mechanism used to encapsulate monitors and generators can be used to create bus-functional models independently from physical interfaces.

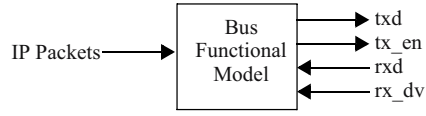


Figure 6-14. Monolithic IP-level bus-functional

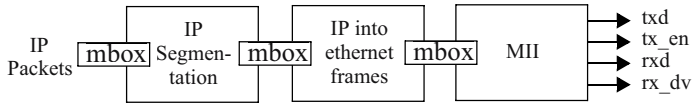


Figure 6-15. Layered IP-level bus-functional

Layers are somewhat arbitrary.

Protocol layers may be obvious in datacom systems where the protocols themselves are layered according to the ISO model. For example, a USB application would be layered according to packets,

transactions and transfers. But even non-datacom applications have similar layers. For example, a graphic system could be composed of pixels, vertices, surfaces and 3D object layers. If you can decompose your system-level functionality into separate functional blocks, then you can decompose your system-level transactions into similar functional layers.

Bus-functional models can be shared across harnesses.

With bus-functional models following the same layers as the functionality they implement, they can be composed into the appropriate combination to create the necessary abstraction layers in system-level verification harnesses. Figure 6-16 shows three different (partial) verification harnesses, for two blocks and one system, each sharing several bus-functional models.

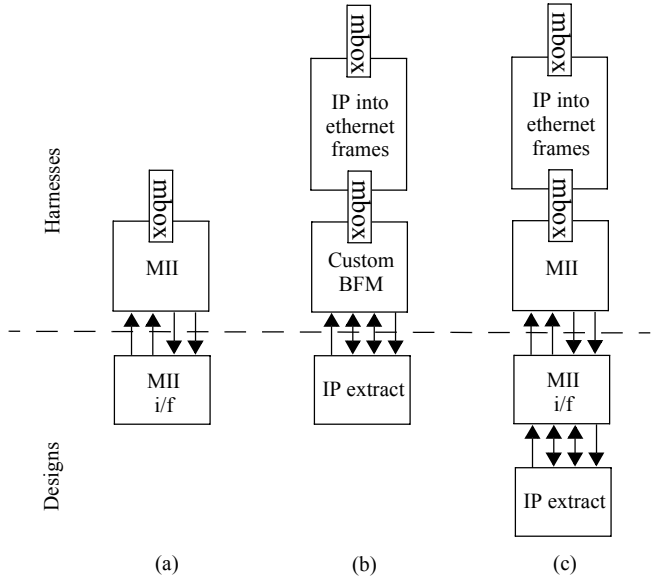


Figure 6-16. Sharing bus-functional model between harnesses.

Tests may be reused.

If the abstraction layer and functionality are maintained across block of system-level harnesses, the testcases written on one harness can be reused on the other. For example, the IP fragment extraction functionality verified in Figure 6-16(b) is independent of the physical interface used to transfer ethernet frames. All tests written on that harness should be reusable, unmodified, on the system-level harness shown in Figure 6-16(c).

Block-level tests may be used as integration tests.

For some blocks and systems, it may be useful to execute block-level tests within the system context to verify that the integration of the block has not affected its functionality. Note that this process does not actually verify any new system-level functionality or increase the block-level coverage. It is simply ensuring that block-level functionality is preserved within the context of the system. In a bus-centric system, it may even be useful to execute block level tests from multiple blocks concurrently to verify the correctness of concurrent block-level operations.

Layered BFMs help port block-level tests.

Figure 6-17 shows how a bus-functional model can be introduced in a block-level stimulus path to undo the function of the design block hiding the system bus interface that used to be visible at the block level. If the access block is a bridge, a simple address remapping function needs to be introduced. If the access block is a processor, write and read cycles simply need to be turned into store and load instructions. If the access block is an MPEG4 decoder, you might have picked the wrong block to go through.

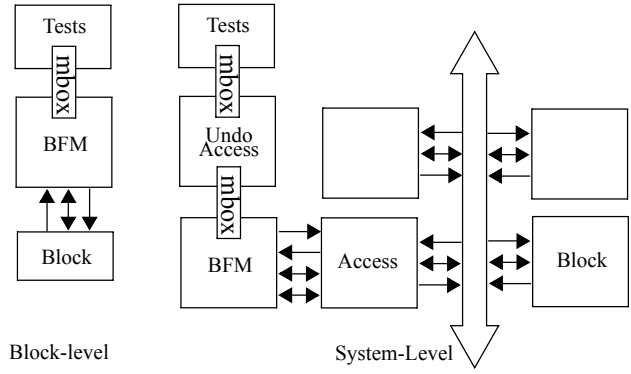


Figure 6-17. Porting block-level tests to the system-level.

See the VMM.

Chapter 8 of the *Verification Methodology Manual for SystemVerilog* specifies guidelines and additional abstraction techniques for verifying system-level designs.

SUMMARY

Encapsulate bus-functional tasks in a class. Provide a suitable transaction-level interface. Layer your bus-functional models according to the functional layers in the application.

Create a transaction-level test harness encapsulating all of the bus-functional model instances and clock generators connected to the design under verification.

Provide a high-level device configuration descriptor. Interpret the descriptor value to configure the design under verification at the beginning of the simulation.

Generate the device configuration randomly. Use functional coverage measurements to determine which configuration (or combination of configurations) has been verified. Use constraints to limit configurations to currently supported or interesting values and combinations.

Make your testbench self-checking. Build the self-checking structure on top of the transaction-level test harness.

Self-checking can be implemented using a reference model, by tagging data, by using a transfer function with a scoreboard or any combination of the above.

Generate directed stimulus by invoking the proper transaction-level procedures directly.

When writing random generators, provide a constraint mechanism that can describe all of the interesting and relevant input sequences.

Provide unique identifiers for each generator instance and data instance to allow stream-specific and order-specific constraints to be expressed.

Write random generators that generate random sequences. Define scenarios to increase your functional and code coverage.

Directed and initial debug testcases can be described as tightly constrained random scenarios.

CHAPTER 7 SIMULATION MANAGEMENT

Simulation must be managed.

In "Revision Control" on page 61, I described how tools can help manage source code. In "Issue Tracking" on page 66, I described how issues and bugs can be tracked to ensure they are resolved. In this chapter, I address the simulation management issues. I describe how to debug your testbenches efficiently using transaction-level models. Often overlooked but important topics, such as terminating your simulation, reporting errors and determining success or failure are covered. We also discuss configuration management: How do you know you are simulating what you think you are simulating?

TRANSACTION-LEVEL MODELS

This section demonstrates how transaction-level models can benefit a design project. These benefits can be realized only if the model is written with the proper perspective. This section also shows how to model exceptions properly and explains how to demonstrate a transaction-level model to be equivalent to an RTL model.

Testbenches need a model to be debugged.

You have decided which testcases and functional coverage measurements are needed to verify a design functionally. Your best verification engineers are developing the verification harness, self-checking structure and random-generators. Hardware design engineers are working furiously on the RTL model, but it will not be available for several weeks. Meanwhile, the verification harness and self-checking structure continue to be written. When all is said and done, the amount of code written for the verification will sur-

pass the amount of RTL code. You are looking at writing thousands of lines of code without being able to debug them. Furthermore, when writing a constrainable random environment, you need a model to exercise your generators to ensure that they offer the constraint capabilities required to generate interesting input scenarios.

Transaction-level models are used to debug testbenches.

What if someone walked up to you and offered you a model, available at the same time as early versions of the verification environment can be simulated, that runs one hundred times faster than the RTL model and that looks and feels just like the real thing? You could start debugging your verification environment even before the RTL is ready. Because this model simulates faster, the debug cycles would be shorter. By the time the RTL is available to simulate, you'd probably have most of the scenarios covering your entire input functional coverage model defined and debugged. The design schedule could be shortened and the verification would no longer be squarely on the critical path. Sounds too good to be true? I'm offering exactly such a model: It is called a *transaction-level* model.

Transaction-level models can be written using SystemVerilog.

There is a tendency in the industry to associate transaction-level models with SystemC. Although SystemC is perfectly suited for writing transaction-level models, SystemVerilog is just as good. Unless you need a model written in C/C++ to integrate with software or ship to your own customers, there is no real practical reason to introduce another language in your verification process. SystemVerilog offers all of the necessary constructs for writing transaction-level models. And with today's direct compilation technology, there is no technical reason for a model written at the same level of abstraction, to run significantly faster when written in SystemC compared to SystemVerilog.

Transaction-Level versus Synthesizable Models

Transaction-level models are not synthesizable.

A model that can be translated automatically into a gate-level implementation by a synthesis tool, such as Synopsys' Design Compiler, is called a *Register-Transfer-Level* or *RTL* model. It also may be called a *synthesizable* model. A transaction-level model describes the black-box functionality of a design, without having to be synthesizable. The Virtual Socket Interface Alliance uses the term *functional* model.

High-level code is not just for testbenches.

In "High-Level versus RTL Thinking" on page 113, I described the characteristics of high-level code compared with synthesizable code. Using high-level descriptions for testbenches is acceptable easily by most design engineers. After all, the testbench will never be implemented in hardware so they never give any thought as to how they would go about it. Their mind hasn't been influenced by an implementation architecture or by a synthesizable description of the testbench's functionality. They are still open to describing this functionality using high-level code.

Writing a transaction-level model requires a different mindset than writing an RTL model.

Writing a truly transaction-level model of a design requires a greater mental leap. You may have already started to think of a design's functionality in terms of state machines, datapaths, operators, memory interfaces and other implementation details. This mindset can be created simply because the functional specification document was written with these implementation details in mind. To write a proper transaction-level model, you have to focus on the *functionality*, not the implementation. If the implementation starts to color your thinking, you'll simply write what I call an "RTL++" model.

Example of Transaction-Level Modeling

RTL++ models may be synthesizable using behavioral synthesis.

For example, consider the specification in Sample 7-1. How would you write a transaction-level description of this functionality? Most would write something similar to the description shown in Sample 7-2. Clearly, this description is not synthesizable using logic synthesis tools. However, it happens to be synthesizable using behavioral synthesis tools such as Synopsys' Behavioral Compiler. The design is synthesizable behaviorally because the description was tainted by the specification: There is an implicit state machine and everything happens at the active edge of the clock.

Sample 7-1. Specification of a debounce circuit

The debounce circuit samples the input at every clock cycle. The debounced version of the input changes state only when eight consecutive samples of the input have the same polarity.

Sample 7-2. RTL++ description of debounce circuit

```
reg debounced;
always @ (posedge clk)
begin: debounce
   if (bouncing != debounced) begin
      repeat (7) begin
         @ (posedge clk);
         if (bouncing == debounced)
            disable debounce;
      end
      debounced <= bouncing;
   end
end
```

A transaction-level model should be refined into a synthesizable model.

The objective of a transaction-level model is to represent the functionality of a design faithfully, in a way that is easy to write and simulate. The transaction-level model is designed to help verification, and indirectly, the implementation. When written properly, a transaction-level model cannot be refined into a model synthesizable by today's logic synthesis tools.

For example, what is the *functionality* of the debounce circuitry specified in Sample 7-1? It prevents pulses on the primary input, narrower than 8 clock periods, from making it to the debounced output. The functionality is similar to a buffer with a significant inertial delay. This behavior can be modeled using a single SystemVerilog statement, as shown in Sample 7-3. The continuous assignment statement uses the inertial delay model built in SystemVerilog. If required, please refer to a suitable SystemVerilog reference book for a detailed description of inertial delays.

Sample 7-3. Transaction-level description of debounce circuitry

```
assign #(8*cycle) debounced = bouncing;
```

Delays cannot be synthesized.

The description in Sample 7-3 is far from being synthesizable. It is not possible to synthesize a specific inertial delay. The other limitation of that description is the need to know the sampling clock period. It could be specified using a *parameter* or a transaction-level model configuration descriptor, but the transaction-level model would not adjust to different clock periods as the real imple-

mentation would. If this is an important requirement, the clock period could be determined at runtime by sampling two consecutive edges. Sample 7-4 shows how this sampling could be performed. Notice how the clock cycle is measured only once to improve simulation performance. It is unlikely that the clock period will change significantly during a simulation. Computing the clock period at every clock cycle simply would consume simulation resources without accomplishing additional work.

Sample 7-4. Measuring the clock period in the debounce circuitry

```
module debounce(input  bouncing,
                output debounced,
                input  clk);

time cycle = 8 * 10ns;

initial
begin
   time stamp;

   @ (posedge clk);
   stamp := $time;
   @ (posedge clk);
   cycle = $time - stamp;
end

assign #(8*cycle) debounced = bouncing;
endmodule
```

Characteristics of a Transaction-Level Model

They are partitioned for maintenance.

A transaction-level model is partitioned differently from a synthesizable model. The latter is partitioned to help the synthesis process. Partitioning is decided along implementation lines, producing a design with several instances arranged in a wide and shallow structure.

Transaction-level models are partitioned according to generally accepted software engineering practices. Transaction-level models tend to be partitioned according to main functional boundaries to avoid maintaining one large file, or to allow more than one author to write it. Duplication of function in a model, such as many interfaces of the same type, is also implemented using multiple instances of a single description. Transaction-level models tend to

have very few instances creating a narrow and shallow structure of large blocks.

They do not use a clock.

A clock signal is an implementation artifice for synchronous design methodologies. These implementation methodologies are functionally irrelevant. A transaction-level model does not change state synchronously with a clock. Instead, a transaction-level model uses many different synchronization mechanisms—one of which could be a clock edge. While an RTL model continuously recomputes and updates the value of inferred registers, a transaction-level model performs computations only when necessary.

Consider the RTL model in Sample 4-3 on page 116: The *always_ff* block is executed every time the clock rises. The signal named *state* is assigned at every rising edge of the clock signal, regardless of the value of *next_state*.

The equivalent transaction-level model in Sample 4-4 on page 116, on the other hand, does not even use a clock. Instead, it acts on the only functionally significant event: the change in *ack.* This behavioral model changes the only functionally significant state, the state of the *req* output.

A clock would be used only when data needs to be sampled or produced synchronously with a clock signal. Examples of synchronous interfaces include PCI or Utopia Level 1. The clock signals for synchronous interfaces are usually externally generated and are not used any further by the transaction-level model.

Transaction-level models do not use FSMs.

Synthesizable models are littered with finite state machines. They are the primary synchronous design methodology for implementing control algorithms. When writing software using a language like C++, you would not usually implement it as a series of cooperating finite state machines. The language does not lend itself very well to that.

Instead, the control algorithm and the data transformations would be part of the control flow of the program. The model's state would depend on the current values of the variables and the location of the statement under execution in the program sequence.

Transaction-level models follow a similar strategy. Consider the example in Sample 4-3 on page 116. The state of the RTL model is determined by the value of the state register and the current input values. The same code is executed over and over. On the other hand, the state of the behavioral model shown in Sample 4-4 on page 116 depends only on which *wait* statement is being executed currently.

Data can remain at a high-level of abstraction.

The skills of the hardware engineer reside in mapping a complex functionality into the very primitive resources available in hardware. Everything must be implemented using a binary value, with a small number of bits, and reduced to integer arithmetic. A transaction-level model can make use of the high-level data types provided by the language, such as enumerals, floating-point numbers, classes, multi-dimensional arrays and queues. The section titled "Data Abstraction" on page 130 illustrates many examples of using high-level data abstraction instead of using representations suitable for implementation.

Data structures are designed for ease-of-use, not implementation.

In a synthesizable model, the format of the data structures are organized to make implementation possible. For example, imagine that a routing table in a packet router is composed logically of 256-bit records with various fields. The router is specified to support 1,024 possible routes and the table is maintained by an external processor through a 16-bit wide interface.

The physical implementation of the routing table is likely to use a 16-bit RAM with 16K locations. Whenever the routing engine performs a table lookup, it has to read a block of 16 words to build the entire 256-bit routing record.

If the table maintenance via the CPU interface has a much lower frequency than packet routing, a transaction-level model would instead optimize the data structure for the table look-up and routing operation. The routing table would be implemented using an array with 1,024 locations, each containing a complete 256-bit routing entry. It could probably use a associative array to minimize memory usage as well. The table would look the same from the CPU's perspective, with each 16-bit access being performed at the right offset within the record identified by the upper 10 bits of addresses. Sam-

ple 7-5 shows an implementation of the CPU access into the routing table of the transaction-level model.

Sample 7-5. Mapping a narrow access in a wide data structure

```
bit [255:0] table[1024];

always
begin: cpu_access
   bit [255:0] entry;
   ...
   entry = table[addr[13:4]];
   if (read) data = entry[addr[3:0]*16 +: 16];
   else begin
      entry[addr[3:0]*16 +: 16] = data;
      table[addr[13:4]] = entry;
   end
   ...
end
```

Interfaces may be implemented using bus-functional models.

The testbench is a transaction-level model of the environment. To make implementation more efficient, Chapters 5 and 6 explained how bus-functional models are used and located in a testcase-independent verification harness. The bus-functional models abstract data from the physical level to a functional level where they are simpler to process using high-level code.

The same strategy can be used when writing a transaction-level model that must present pin-true physical interfaces. Bus-functional models are used for each interface around the periphery of the model. Data is transformed at the transaction level and moved from bus-functional model to bus-functional model according to the function of the device. And as Figure 7-1 shows, you will likely be able to reuse the bus-functional models written for the testbench in your transaction-level model.

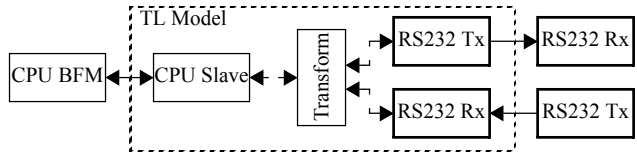


Figure 7-1. Structure of a UART test harness and behavioral model

Modeling Reset

Reset is part of the RTL coding style.

Modeling exceptions can take of lot of time and introduce a lot of intricacies in an otherwise simple algorithm description. When writing a synthesizable description, modeling the effect of reset on the state elements is defined in the supported coding style. For example, Sample 7-6 shows how an asynchronous reset is modeled to reset a finite state machine. Resetting an entire RTL model is accomplished by including the logic to handle the reset exception in each *always* block that infers a register.

Sample 7-6. Modeling an asynchronous reset in RTL

```
always_ff @ (posedge clk
             iff rst == 1 or negedge rest)
begin
   if (rst) state <= IDLE;
   else begin
      case (state)
         ...
      endcase
   end
end
```

Transaction-level models must reset variables and execution points.

As described in the previous section, the state of a transaction-level model is not just composed of the values of the variables. It also includes the location of the statement currently being executed in the sequence of statements describing each concurrent execution thread. To reset a transaction-level model, you need not just reset the content of the variables. You must also reset the execution to a specific statement, usually at the top of the *always* blocks. For example, resetting the *always* block shown in Sample 7-7 would require resetting the variables and signal drivers to their initial values, as well as restarting the execution of the *always* block at the top.

Disable all the blocks on reset

Resetting a transaction-level model in SystemVerilog is easy and elegant. When an exception is detected, all you need to do is disable all the blocks in the model using the *disable* statement. The *always* blocks restart their execution from the top. Note that, as described in “Disabled Scheduled Values” on page 187, pending values assigned using a nonblocking assignment may remain in the event queue and clobber the reset state of a variable in some SystemVerilog simulator.

Sample 7-7. Transaction-level blocks to be reset

```
initial count = 0;
always
begin
   strobe <= 1'b0;
   wait (go);
   while (go) begin
      count++;
      @sync;
   end
   strobe <= 1'b1;
   #10;
   strobe <= 1'b0;
   wait (ack);
   count = 0;
end
```

Use a *forever* loop in *initial* blocks.

Only the *initial* blocks present a difficulty. Since they only run once in a simulation, they cannot be disabled since they are no longer active. If they are still active, disabling them would simply make them inactive immediately. They cannot be replaced by *always* blocks because *program* can only contain *initial* blocks. To include *initial* blocks in the reset handler, simply turn their body into a *forever* loop with an infinite *wait* statement at the bottom. Sample 7-7 shows an original transaction-level model. Sample 7-8 shows the same model, this time with the proper handling of reset exceptions using the *disable* statement.

Encapsulate the *disable* statements in a task.

It is good practice to encapsulate all *disable* statements into a single task to perform a reset of a transaction-level model. Multiple reset sources and exception detection can call this task to perform the reset operation. This technique also reduces maintenance to a single location when *always* blocks are added or removed. The *reset* task can also be called using a hierarchical reference when a higher-level module in a complex transaction-level model needs to reset all its lower-level components. This approach is more efficient than having to create and assert a synthetic reset signal broadcast through the pins of all interfaces in the model. Sample 7-9 shows the reset handler of Sample 7-8 modified to use a task to disable all of the blocks.

Writing Good Transaction-Level Models

Many attempts to write transaction-level models fail.

I have seen and heard of many projects where the use of transaction-level models was attempted, but without producing much benefit over RTL models. Often, the transaction-level model was

Sample 7-8. Transaction-level model with reset detection and handling

```
initial
forever begin: init
   count = 0;
   wait (0);
end

always
begin: main
   strobe <= 1'b0;
   wait (go);
   while (go) begin
      count = count + 1;
      @ sync;
   end
   strobe <= 1'b1;
   #10;
   strobe <= 1'b0;
   wait (ack);
end

always
begin
   // Detect reset exception
   ...
   disable init;
   disable main;
end
```

Sample 7-9. Encapsulating the *disable* statements in a task

```
task reset;
   disable init;
   disable main;
endtask

always
begin
   // Detect reset exception
   ...
   reset;
end
```

abandoned in favor of the RTL model as soon as the latter became available. The transaction-level model failed to exhibit any of the benefits outlined in "The Benefits of Transaction-Level Models" on page 349.

Writing a good transaction-level model requires specialized skills.

Further investigation into those failed attempts usually reveals that the transaction-level model was written by experienced hardware designers. Unfortunately, their valuable skills were not appropriate to writing good transaction-level models. Their level of thinking was still too close to the implementation and they had difficulty thinking in terms of higher levels of abstraction. Very often, there was the implicit intent of refining the transaction-level model into a synthesizable model. This is a fatal mistake as it is conducive to low-level thinking that yields not a behavioral model, but an "RTL++" model.

Focus on the relevant functional details.

All the techniques illustrated in this chapter, as well as in Chapter 4, can be used and still yield a poor transaction-level model. A good transaction-level model captures the details that are functionally relevant and does not include implementation artifices. For example, the *latency* of a design—the number of clock cycles necessary for an input to be transformed into an output—is usually not functionally relevant[1]. If you insist on writing a model that is clock-cycle accurate with the actual implementation, you may be spending a lot of maintenance effort and adding a lot of complexity for a characteristic that may not be important functionally.

At first glance, latency seems a significant characteristic.

To many, saying that latency may not be a relevant functional detail and should not be modeled sounds like a recipe for disaster. But if you take a step back from your design, ignoring its implementation details, does it *really* matter whether a particular output comes exactly N cycles after the corresponding input was sampled? As long as the *order* of these outputs is the same, is the time at which they come out significant?

Consider the speech synthesizer design illustrated in Figure 3-4 on page 95. To produce audible speech, coefficients must be modified at regular intervals to produce the different sequences of sounds that compose normal speech.

For example, to say "cat", the coefficients would be modified to create the sequence of sounds "k", "a", "a", "a", "t", "t". From these coefficients, a digitized sound waveform should come out at a 8 KHz sample rate. The delay between the time the coefficients are

1. But if it *is* relevant, then it should be modeled.

set and the corresponding sound is synthesized is irrelevant, as long as it is under the limit of perception by the user. A similar argument can be made for packet routers: It does not really matter how long it takes for packets to transit through a routing node; what matters is that they eventually come out in the same order.

In some cases, latency *is* significant.

The only time where a "detail" like latency is significant is when the design under verification does not have complete visibility and control over all elements of a system-level transaction. A system-level transaction is the smallest amount of data than can be processed by the system: an atomic operation. For example, a packet router's system-level transaction is an entire packet. In a speech synthesizer, it is a phoneme. In a hardware tester, it is a complete vector with input and expected output values. If the design under verification only processes a portion of the system-level transaction, it is important that the latencies in the reconvergent paths are identical so the system-level transaction is reassembled properly.

For example, the input formatter in a hardware tester, as illustrated in Figure 7-2, only processes the input value. For the corresponding expected output value to be checked at the proper time, it must have the exact same latency as the Expect Delay design.[1] In a packet router, as illustrated in Figure 7-3, if the packet is dismembered to be routed by a different switching node, each node must have an identical latency for the packet to be put back together properly. If you mix behavioral and RTL models in a system-level verification, and each has a different latency, the system-level simulation would become a very effective packet scrambler!

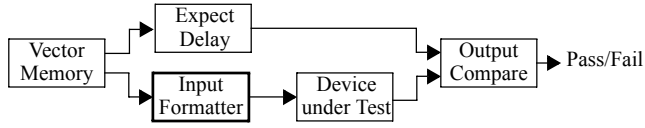


Figure 7-2. Reconvergent paths in a hardware tester

1. Actually, since the latter is easier to design, its latency is made to match that of the input formatter, whatever it may be.

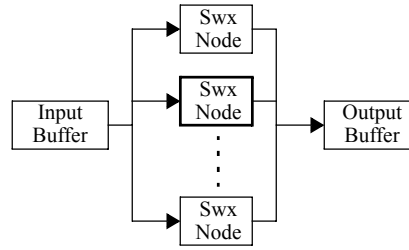


Figure 7-3. Reconvergent paths in a packet router

Do not let the testbench dictate what is functionally relevant.

The reason most often cited for making a transaction-level model clock-cycle accurate with the implementation is to be able to pass the same cycle-oriented testbenches. If the testbenches enforce a specific latency, they are verifying a specific implementation, not a specification.[1] I hope I have explained successfully how to write testbenches that are independent of the latency of the design under verification in Chapters 5 and 6. If your testbenches do not expect a specific latency, then you need not model it.

Details relevant at the system-level can be back-annotated.

An implementation "detail", such as latency, may not be relevant to the functionality of the *stand-alone* design under verification. However, it may be critical for the proper operation of the system-level design. If that is the case, such as the example designs shown in Figure 7-2 and Figure 7-3, the behavioral model still may be modeled as if the latency was not important and perform its transformation in zero-time. At appropriate points in the input or output paths, programmable delay pipelines can be introduced so the exact latency of the implementation can be *back-annotated* into the transaction-level model. The transaction-level model would then model the functionality of the synthesizable model at a clock-accurate level. Sample 7-10 shows a configurable delay pipeline to adjust the latency of a transaction-level model.

Specify the functionality, not the implementation.

Another big obstacle to writing good and efficient transaction-level models is the level of the specification for the design. If it is written at a very low level, it becomes difficult to abstract significant functionality and discard irrelevant implementation details. I once had to write a transaction-level model for a customer whose functional

1. Unless of course a specific latency is required, in which case it should be specified in the specification document. And if something is specified, it should be modeled and verified.

Sample 7-10. Configurable delay pipeline

```
initial
begin
   const int unsigned delay = 10;
   transaction pipeline[$];

   repeat (delay) pipeline.push_back(new);
   forever begin
      always @ (posedge clk);
      actual_ouput = pipeline.pop_front();
      pipeline.push_back(output);
   end
end
```

specification was done using technology-independent schematics using a general-purpose drawing tool. Each block was specified independently with no description of the overall functionality. Not only did it make the job of writing RTL code that met timing requirements difficult, it made writing a transaction-level model impossible. After 10 weeks, I had a model that was barely faster than the RTL model. But after those 10 weeks, I was able to piece the entire design together in my mind and understand the intended functionality. I scrapped the first model and rewrote it *entirely* in under two weeks. That newer model outperformed the RTL model. Had the specification been written at an appropriate level in the first place, a more effective behavioral model could have been written from the start.

Transaction-Level Models Are Faster

They are faster to write.

As shown in “High-Level versus RTL Thinking” on page 113, a high-level model is much faster to write simply because the functionality is described using significantly fewer statements than an RTL model. Furthermore, transaction-level models do not need to meet physical timing or other implementation constraints. They are written with the sole purpose to describe the functionality of a design.

They are faster to debug.

The fewer statements, the fewer bugs. Bugs are easier to identify because of the simpler descriptions. The code is written based on a functional description. The code is not cluttered by directives aimed at a synthesis tool or twisted to be synthesized into specific hardware structures. Transaction-level models also tend to use fewer parallel constructs, instead preferring large sequential descriptions

in a few blocks. Sequential code is much easier to debug than parallel code, since it does not involve synchronization or data exchange intricacies.

They are faster to simulate.

Less code used to describe a function should naturally simulate faster. But the greatest contributor to the increase in simulation speed of a transaction-level model over a synthesizable model is avoiding the use of the synthesizable subset itself. Look at all the *always* blocks used to infer registers. Each and every one of them is sensitive to the clock. If you remember the discussion on event-driven simulation in "The Parallel Simulation Engine" on page 159, you know that this sensitivity causes all of these threads to be scheduled for execution after each event on the clock signal, whether their internal state needs to change or not.

In a typical ASIC, activity levels are well below 40%. This means that well over 60% of the *always* blocks are evaluated for no reason. A transaction-level model only executes when there is useful work to be done. The load it puts on the simulator is much lower. In the small example illustrated in "Contrasting the Approaches" on page 115, the activity in the high-level model is estimated to be 10 times lower than in the equivalent RTL model.

They are faster to bring to "market".

Being faster to write and debug, a transaction-level model takes significantly less time to develop to a level where it can be used in a system-level model. With transaction-level models, you are able to start system-level simulations sooner. Because they also simulate faster, you are able to run more of them, on less expensive hardware.

The Cost of Transaction-Level Models

Transaction-level models require additional authoring effort.

Someone has to write these transaction-level models. If you use your existing resources, it means that the coding of the RTL model will be delayed. If you do not want to affect the schedule of the synthesizable model, you will have to hire additional resources to write the transaction-level model. But being a completely separate model, it is a task that is easy to parallelize with the implementation effort. And writing a transaction-level model is not as costly as writing an RTL model. A transaction-level model that is sufficient to start simulating and debugging the testbenches should not take more than two person-weeks to produce. A complete model with all

of the functionality of the design under verification should not take more than five percent of the effort required to write an equivalent RTL model.

The maintenance requires additional efforts.

When was the last time you were involved in a design project where the functional specification did not change? Whenever a functional or architectural change is made, the transaction-level model needs to be modified. Often, these modifications are dictated by the RTL model because the technology cannot implement the original design and still meet timing requirements. Some of these implementation-driven changes can be planned for and made easy to modify, such as the latency. More significant changes may require rewriting a significant portion of the transaction-level model. Toward the end of a project, when schedule pressure is at its greatest, it often leads to the decision of abandoning the transaction-level model in favor of focusing on the RTL.[1] However, most of the modifications to an RTL model are made to meet timing goals and do not affect the functionality of the design and thus should not require modification of the transaction-level model.

The Benefits of Transaction-Level Models

Audit the specification.

Most specification reviews I have attended focus on high-level functions and on the spelling and grammatical errors in the document. The missing functional details were often left to be discovered during RTL coding. Decisions regarding these functional details were usually then made according to the ease of implementation. There is nothing like writing a model to make you thoroughly read a specification document.

For example, after you've coded a particular function that occurs under some condition, you've come to the *else* part of the *if* statement. What should be done when the condition does *not* occur? Flip, flip, flip through the specification document. Not a word. You've just found a case of incomplete specification! Since you are writing the transaction-level model faster than the RTL model, you'll reach that section of the specification earlier than the RTL designers. By the time the RTL model incorporates this functional-

1. An error in my opinion. See the next section titled "The Benefits of Transaction-Level Models".

ity, it will have been specified. A similar process occurs with inconsistencies in the specification. When the RTL model is written, there are fewer problems in the specification, and thus it takes less time to write.

Develop and debug the testbenches in parallel with the RTL coding.

Testbenches are implemented using code, just as RTL models are. If the RTL model requires debugging, so do the testbenches. Since a transaction-level model is available much earlier than the RTL code, you are able to debug the testbenches earlier as well. You are debugging the transaction-level model and the testbenches effectively while the RTL is being written. And because the transaction-level model simulates faster than the RTL model, the testbench debug cycles are much shorter.

Once the RTL is available, you will have a whole series of debugged testbenches. Whenever an error is detected, it likely will be due to an error in the RTL model. If you decide to abandon the maintenance of the transaction-level model after the RTL is available, debugging the testbenches (which will also need to be modified whenever the RTL is modified significantly) will take much longer. It is important to maintain the transaction-level model to keep reaping its benefits for the entire duration of the project.

System verification can start earlier.

Figure 7-4 shows a design process that uses transaction-level models for developing the testbenches and the functional verification of the system. Figure 7-5 shows a comparative timeline for a design and verification process with and without transaction-level models. The design process is somewhat shortened by using a transaction-level model because the testbenches are already debugged. But the greatest saving comes from system verification. The transaction-level model is available sooner than the RTL model, so functional verification can start much earlier. Because a transaction-level model is much smaller and simulates more efficiently than the equivalent RTL model, you are able to create models of larger systems, execute longer testcases and run on ordinary hardware platform configurations. If the transaction-level model is demonstrated to be equivalent to the RTL model, the latter never needs to be brought into the system-level verification. For systems incorporating very large designs, a transaction-level model may be that which makes system verification even possible.

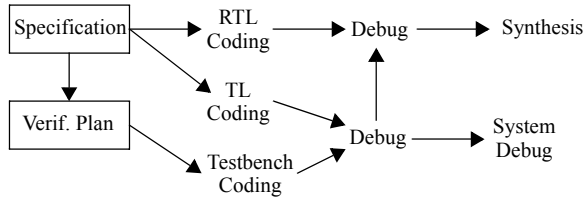


Figure 7-4. Design process including behavioral model

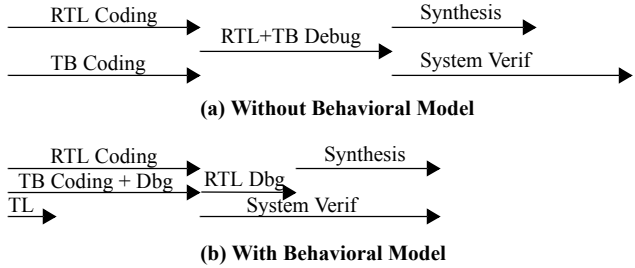


Figure 7-5. The effect of behavioral models on a project timeline

It can be used as an evaluation and integration tool by your customers.

If your design is to be available as reusable intellectual property or a chipset, a transaction-level model can be a powerful marketing tool. Since it only describes functionality, not implementation, and it is far from being synthesizable, the transaction-level model should not convey intellectual property information.[1] A customer could start using the transaction-level model while the legal issues with licensing the RTL model are being resolved. The system-level models could be used as application notes. The transaction-level model could be used to start the integration of your design into your customer's design. Since reusing intellectual property is about time-to-market, a behavioral model can be an effective tool to help your customers improve the odds that they will meet their market window.

Demonstrating Equivalence

The RTL and TL models must be equivalent.

The greatest benefit from creating a transaction-level model comes from system verification. To use it instead of the RTL model in a simulation or as a marketing tool, you have to demonstrate that both are an equivalent representation of the design. I use the term *dem-*

1. Unless the intellectual property is in the function itself, such as a DSP algorithm.

onstrate because I do not think it will ever be possible to *prove* mathematically that they are equivalent.

Equivalence checking can prove that an RTL model is equivalent to a gate-level model or to another RTL model because they are structurally very similar. A properly written transaction-level model would use a completely different modeling approach that would be very difficult to correlate mathematically with the equivalent RTL model.

Demonstrate equivalence by using the same test suite.

The only way to demonstrate that the transaction-level and the RTL models are equivalent is to verify both of them using the same verification environment. If both models pass the same testcases, from a system-level perspective, it should not matter which one you are using. For a testcase to be executable on both models, it must not depend on a specific implementation. Based on the testcase taxonomy described in "Functional Verification Approaches" on page 11, only black- and grey-box testcases can be used to demonstrate equivalence. Both are executed through the same physical interface. Both do not depend on a particular implementation of the design under verification. The grey-box testcases may not be very relevant to the transaction-level model as they are designed to test a particular implementation-specific feature in the RTL model, but should nonetheless execute successfully.

PASS OR FAIL?

This section describes how the ultimate failure or success of a self-checking testbench is determined.

The absence of errors is not a sufficient condition.

The goal of a testbench is to determine if the design under verification passes or fails a simulation. But how do you determine if the design passed the simulation? Is it by the absence of error messages? What if the simulation never ran at all? It could be caused by a lack of licenses, or a runtime error such as running out of memory or experiencing a power failure, or a simple syntax error in your source code. You need positive proof that the simulation ran to completion successfully.

Produce and look for a termination message.

Do not rely on a time bomb to terminate your simulation. Nor should you attempt to have the simulation terminate by itself through event starvation. Each simulation should be terminated

intentionally. Upon termination, it should produce a message that the simulation was terminated normally. If that message is not present, you must assume that the simulation did not run to completion and failed. To terminate a simulation from within the testbench, use the *$finish* statement. Sample 7-11 illustrates the use of an explicit termination statement.

Sample 7-11. Terminating a simulation

```
program test;
harness th = new;

initial
begin
   ...
   $write("Simulation terminated normally...\n");
   $finish;
end
endprogram
```

An error in the testbench could prevent error detection.

What if there is a functional problem in your testbench? That error could prevent the testbench from detecting any errors at all. This would clearly be a *false-positive* situation. You should always ensure that your testbench is functionally correct as part of your testcases. Error detection can be verified by injecting errors deliberately in the design under verification. These errors can be introduced by simply misconfiguring the design for the expected output. For example, a UART could be configured with the wrong parity setting to verify that the output monitor detects the bad parity.

Provide consistent error message formats.

The final pass or fail judgment could be made by a script parsing the simulation output log file, counting all error messages from all sources. To facilitate the implementation of such a script, use a consistent error format. This style is best accomplished by using a message log package that produces consistent headers, as shown in Sample 7-12.

Keep track of success or failure in the log service.

By using a single message log service as shown in Sample 7-13, it is possible for the simulation to keep track of its own success or failure by checking that no error messages were issued. By including a simulation termination function, the final pass or fail indica-

Sample 7-12. Simulation log service.

```
class syslog;
static integer n_errs = 0;

task warning;
   $write("WARNING at %t: ");
endtask

task error;
   n_errs++;
   $write("*ERROR* at %t: ");
endtask

endclass
```

tion can be determined by the simulation, without using a script to parse the output log.

Sample 7-13. Determining pass or fail in the simulation log package

```
module log;

integer n_errs;

task warning;
   $write("WARNING at %t: ");
endtask

task error;
   n_errs = n_errs + 1;
   $write("*ERROR* at %t: ");
endtask

task terminate;
begin
   $write("Simulation %0sED\n",
          (n_errs) ? "FAIL" : "PASS");
   $finish;
end
endtask

endmodule
```

Assertions have predefined error reporting system tasks.

SystemVerilog has a set of predefined error message system tasks that are usually used for specifying user-defined messages in assertions. If any error message is issued through one of the predefined system tasks, the simulator will issue and format the messages, not a user-defined message log service. The message service will thus be unable to properly declare a failure. However, if these pre-

defined system tasks are used only when assertions fail, the simulator will be able to report a failure.

Using a script to parse the simulation output log is still a good idea.

Using a message log service is not sufficient to determine if a testcase is successful. Other errors could have been generated before the simulation started, when the log service is not available. Other error messages generated by the simulator itself—such as default assertion failure messages and solver failures—would not be issued by the log service. You still need to confirm the presence of the termination message to verify that the testcase was executed properly in its entirety. Errors could also have been produced by some verification IP using its own log service. The output log parsing script can also detect the presence of errors or warnings issued by the simulation management tools, linting tools, syntax errors, elaboration warnings and other possible error conditions not visible to the testbench log service.

See *vmm_log*.

The section titled “Message Service” starting on page 134 of the *Verification Methodology Manual for SystemVerilog* specifies a powerful message service and guidelines for using it.

MANAGING SIMULATIONS

Are you simulating the right model?

You’ve defined your verification task through a verification plan. You have a verification harness with many bus-functional models and utilities. Several testcases using that verification harness have been written and you can choose between the RTL and transaction-level model to simulate them. How do you bring all of these components together in a single simulation? How can you reproduce a simulation? And more importantly, how do you make sure that what you simulate is what will be built?

Configuration Management

A configuration is the set of models used in a simulation.

Configuration management is different from source management. Source management, as described in “Revision Control” on page 61, deals with changes to source files and the set of source files making up a particular release. Configuration management deals with the particular set of models you decide to use in a particular simulation. For a specific design, a single configuration would be composed of a specific test function, the verification harness used by that test function and the model of the design to be exer-

cised by the testbench. In a system-level simulation, the configuration would also include that particular mix of models used to populate the system model.

It must be easy to specify a particular configuration.

The only information required to define a particular configuration is the identity of the test function, the verification harness and the model of the design under verification. The problem is that each configuration component is composed potentially of several source files and design units. Many individuals contribute to the creation of these source files and design units. Their number and names may change throughout the project. It is not realistic to expect every engineer who needs to run a simulation to know exactly what makes up a particular component of the desired simulation. Just as bus-functional models abstract the data from the physical implementation level, configuration management abstracts the details of the structure of a model and the files that describe it.

Use a script to create a configuration.

The most efficient way to abstract the configuration details from the user is to provide a script that expands a test name and an abstraction level for the design under verification into their respective simulation components. Different scripts have to be used for different designs, file system structures and simulators. To simplify the user interface and minimize the amount of repeated information, scripts infer pathnames and expect particular set-up files.

For example, Sample 7-14 shows the command line of a hypothetical script named *sim_design* used to simulate a configuration composed of the test named "*basic_tx*" on the transaction-level model. It is followed by a configuration composed of the testcase named "*overflow_rx*" on the RTL model.

Sample 7-14. Configuration script command line

```
% sim_design -t basic_tx
% sim_design -r overflow_rx
```

There are many ways of specifying files.

There are six different ways to include a source file into a SystemVerilog simulation:

1. Specify the filename on the command line.
2. Specify the name of a file containing a list of filenames, using the *-f* option.

3. Specify a directory to search for files likely to contain the definition of a missing module, using the *-y* option. The files used in the simulation depend on the *+libext* command-line option.
4. Specify the name of a file that may contain the definition of missing modules, using the *-v* option.
5. Include a source file inside another using the *`include* directive. The actual file included in the simulation depends on the *+incdir* command-line option.
6. Locate files in virtual libraries specified in a library search order in a *configuration*.

Use *'include* or the *configuration*.

Of all the mechanisms for specifying input source files in SystemVerilog, only the *'include* and *configuration* mechanisms can be source-controlled and reproduced reliably. They are also the only mechanisms that are defined formally as part of the language and not left to the tool implementation.

Use a *configuration* for each model of the design.

Each available model of the design should be specified using its own *configuration*. For example, if you have a transaction-level model, two versions of the RTL model (one for FPGAs, the other for the final ASIC) and two gate-level models (one mapped to FPGA gates, the other mapped to ASIC gates), there should be five different *configurations* available.

Use *'include* for the verification harness and testcases.

The *configuration* is designed for the static structure of SystemVerilog models: module, interface, program and package instances. It is not designed for class instances. It is not possible, using a *configuration* to specify which version of a class to instantiate in the verification harness or testcases. The verification harness should implicitly support all possible configurations of the design. All the files required by the verification harness and testcases should be included using a *'include* directive. A suitable *'ifndef/'define/'endif* structure should protect all bus-functional model and harness component files against multiple inclusion, as shown in Sample 7-15.

Include files at *$root* level.

When a file is included by another file using the *'include* directive, it is included as-is within the scope where the *'include* directive is specified. Files should be coded to be included in the *$root* scope, i.e. outside of any other scope, at the primary file level. This will ensure that files are compilable stand-alone and do not require some additional context to be used. It will also ensure that files are included in a known and consistent context.

Used files should report name and version.

For additional confidence and a positive confirmation of the files and version of the files used in a simulation, you should have each file report its name and revision number. Sample 7-15 shows how to use a *module* containing a single *initial* block and RCS keywords to perform this task. All modules included in the compilation are included in the simulation, even if they are not instantiated. If each file contains a suitably uniquely named module, it can be used to display the filename and revision information. This mechanism requires that all source files be designed to be included or compiled in the *$root* scope, as recommended in the previous paragraph..

Sample 7-15. File reporting its filename and revision

```
`ifndef ATM_CELL__SV
`define ATM_CELL__SV

module file_atm_cell_sv;
initial $write("Configuration: $Header$\n");
endmodule

class atm_cell;
   ...
endclass: atm_cell

`endif
```

Avoiding Recompilation or SDF Re-Annotation

This section may not be applicable.

With some simulators, compiling the entire design under verification and the verification harness for each testcase may take a long time. To minimize the amount of time spent recompiling code that does not change from testcase to testcase, it may be necessary to use a compile-once, run many times strategy. Other simulators may provide negligible compilation times or incremental compilation technology that does not present this problem.

Back-annotation is a process used only with gate-level models. Due to their large size, they are excruciatingly painful to simulate in terms of performance and resource requirements. The purpose of gate-level simulation is to verify that the synthesis tool has synthesized the RTL description correctly without modifying the functional behavior. The purpose of gate-level simulation is also to verify that there are no timing violations. In most circumstances, these checks are better performed using an equivalence checking static timing analysis tool (see “Equivalence Checking” on page 8).

SDF files are used to model accurate delays.

In a gate-level model, each gate is modeled using delays estimated from average output load conditions. However, in a real gate-level netlist, each gate is subject to different output loads: The gates drive different numbers of inputs, and the length of the wires connecting the output of the gate to the driven inputs are different. Each contributes to the load on the output of the gate, producing different loads for different instances of the same gate.

To be more accurate, gate-level simulations are back-annotated with delay values calculated from the physical netlist or the layout geometries. These more accurate delay values are stored in a *Standard Delay File*. The *SDF* file is read by the simulator and each delay value replaces the average delay estimate for each instance. Thus, each gate instance can have a different delay value. The delay between an output pin and each of its driven input pins also can be different.

SDF annotation can take a long time.

Gate-level netlists can contain a few million gates and several million pin-to-pin nets or connections. Each must be annotated with a new delay value. This process can be very time consuming and should be minimized whenever possible. If you have to recompile your model for each testcase, you have to perform the back-annotation each time as well.

Use compiled back-annotation whenever possible.

Compiled simulators usually offer compile-time back-annotation of a gate-level model. In that mode, the back-annotation of the delay values is performed once at compile time. Different testcases can be configured to run on the design in separate simulations without requiring that the back-annotation process be repeated.

Concatenate testcases to minimize back-annotation.

Reducing compilation time can only be accomplished by minimizing the number of times the simulation is compiled. To compile the simulation only once for multiple testcases, you need to concatenate each testcase into a single simulation. The simulation is then invoked multiple times, to execute each testcase separately. It may also be possible to sequence separate testcases into a single simulation. However, concatenating testcases at run-time will create reproducibility challenges: what if a bug is found in a test when it runs after a hundred other tests? How can the bug be efficiently reproduced and debugged by running just that one testcase?.

Encapsulate testcases.

For testcases to be compilable into the same simulation, they must not interfere with each other. Therefore, any testcase-specific decla-

rations must be localized to that testcase only. This can be accomplished by encapsulating testcases into their own class. Within that class, they are free to make local declarations without affecting other testcases. Simply pass a reference to the verification harness to the test class either at construction time, as shown in Sample 7-16. The test procedure itself is then encapsulated in a *run()* method. If a testcase was originally implemented using multiple *initial* blocks, they should be forked from within the *run()* method. Deriving all testcase classes from a common base class is a good idea to facilitate some of the tasks that will be discussed in the following paragraphs.

Sample 7-16. Encapsulated testcase

```
class test1 extends testcase;
   harness th;

   function new(harness th);
      this.th = th;
   endfunction

   virtual task run();
      ... // Testcase procedure
   endtask: run
endclass: test1
```

Reset the state of the verification harness.

Testcases must be able to execute, whether they are the first or last testcase to be run in the sequence. Therefore, the initial state of the verification harness must be the same at the start of a testcase, regardless of the execution order of that testcase. After completion, a testcase should reset the state of the verification harness to the same state it found it at the start. Resetting a verification harness involves more than simply resetting the execution threads in the bus-functional models. It involves removing any callbacks introduced by the testcase. It also involves returning random generators to their default, unconstrained state by restoring the default randomized instances that were replaced by constrained extensions. If random stability is required between consecutive testcases, the state of the random generators must also be restored.

The design need not be reset.

Whether the design itself is reset and reconfigured is up to the testcase. The response checking or purpose of a testcase may depend on a specific configuration and initial state of the design—especially directed testcases. But some testcases may be able to

explore additional corner cases if they can and are allowed to execute from an arbitrary design state.

Instantiate and run all known testcases.

A simulation must be able to run all known tests included in the compilation. This can be done by instantiating all known testcases in an array or queue of tests, as shown in Sample 7-17. The array can then be traversed to identify and run the required tests.

Sample 7-17. Instantiating and running known testcases.

```
program tests;

harness th = new;
testcase known_tests[$];

initial
begin
   test1 tc = new(th);
   known_tests.push_back(tc);
   ...
   foreach (known_tests[i]) begin
      known_tests[i].run();
   end
   $finish;
end
endprogram: tests
```

Identify testcases using user-defined options.

The *initial* block invokes the *run()* method of all known testcases. To control which testcases are run and which ones are not, each *run()* method contains an *if* statement that tests for a user-defined command-line option. That way, you might run only a subset of testcases instead of all of them. Sample 7-17 shows how all testcases are created and run. Sample 7-18 shows how the *run()* method of each testcase checks if it is selected based on a user-defined command-line option. The *+all_testcases* user-defined option can be used to run all testcases. Notice how the use of a testcase base class and the object-oriented programming model greatly simplifies the implementation of each test selection. Sample 7-19 shows an example of each usage with a VCS-compiled simulation. Notice how it was unnecessary to recompile the model to execute different testcases.

Output File Management

Simulations produce output files.

A simulation usually creates at least one output file. For example, VCS simulations generate a copy of the output messages in a file

Sample 7-18. Testcase selection mechanism.

```
class testcase;
   function bit is_selected(string name);
      is_selected =
         $test$plusargs("all_testcases") ||
         $test$plusargs({"run_", name});
   endfunction: is_selected
endclass: testcase

class test1 extends testcase
   ...
   virtual task run();
      if (!super.is_selected(“test1”)) return;
      ... // Testcase procedure
   endtask: run
end
endmodule
```

Sample 7-19. Running different testcases

```
% vcs -f all_tests.f -f gate/design.f \
      -f phy/sdf.f
% ./simv +run_testcase3 +run_testcase7
% ./simv +all_testcases
```

named *vcs.log* by default. Another frequently produced output file is the file containing the signal trace information for a waveform viewer. These output files are valuable. They are used to determine if the simulation was successful. They should be saved after each simulation run and parsed or post-processed to determine success or failure.

Multiple simulations can clobber each other’s files.

When you run only one simulation at a time, you can save them by renaming them after the completion of the simulation. That way, you can keep a history of testcases that were run on the design under verification. However, if you run multiple simulations in parallel, usually on different machines, the output files from one simulation can clobber those of another. If you rely on default or hardcoded filenames, you will not be able to run simulations in parallel. You must be able to name files differently for different testcases.

Specify output filenames on the command line in your simulation run script.

A few default output filenames can be changed from the command line. For example, the *-l* option can usually be used to change the name of the output *log* file. In “Configuration Management” on page 355, I recommended that you use a script to help manage the configuration of a simulation. That same script can also manage the naming of the output files according to the name of the testcase.

Sample 7-20 shows how a perl script can use the name of the testcase specified on the command line to rename the output log file.

Sample 7-20. Simulation run script

```
require "getopts.pl";
&usage if &getopts("hr") || $opt_h || !@ARGV;

sub usage {
   print <<USAGE;
Usage: $0 [-r] {testcase}
  -r   Use the RTL model instead of behavioral
USAGE
   exit (1);
}

$design = ($opt_r) ? "rtl" : "beh";
$prefix  = "vcs -R -f $design/design.f ";

foreach $test (@ARGV) {
   $command = "$prefix -f tests/$test.f";
   $command .= " -l logs/$test.log";
   system($command);
}
```

Name created files after the testcase.

If files are created by the verification harness, they should be named according to the testcase that caused them to be created. This can be accomplished by simply using an argument to the verification harness constructor instead of a hard-coded name and pass the testcase name to the verification harness as a constructor argument. String variables can be concatenated using the usual concatenation operator to create unique filenames. Sample 7-21 and Sample 7-22 show an example.

Sample 7-21. Generating unique file names

```
class harness;
   ...
   local string testname
   int results;

   function new(string testname);
      this.testname = testname;
      results = $fopen({testname, ".out"});
      ...
   endfunction: new
endclass
```

Sample 7-22. Specifying unique file names

```
program test1;
harness th = new("test1");

initial
begin
   ...
end
endprogram: test1
```

Seed Management

Seeds contribute random stability.

The main concern with random stimulus is reproducing a simulation that detected a functional error. Random stability allows the same input sequence to be generated if the same initial seed is used, even in the presence of some changes in the source code. Any instance that is not affected by randomization-related source code changes—such as additional constraints or random objects—will always produce the same pseudo-random sequence in two different simulations if the same seed is used, even if other instances are affected by such changes. Random stability involves more than using the same seed. A C++ model using the *random*() function is not randomly stable as any change in randomization-related code will affect all subsequent calls to the function in the entire model. With random stability, the effects are localized in the modified instance.

Don't always use the default seed.

SystemVerilog uses a default seed unless a different seed is specified. Most people keep using the default seed over and over until the simulation runs error free, then they consider their job done. Using the same seed will generate the same input sequence. You will not be verifying or debugging your environment under different conditions. Before declaring your environment or bus-functional model "done", verify it with different seeds.

Pick random seeds.

Your SystemVerilog simulator may be able to pick a random seed automatically using a specific command-line option. If not, it must have a command-line option to manually specify a specific seed. That value can be a random value generated by a suitable function in your simulation run script or the output of the simple C program shown in Sample 7-23. Do not generate a random seed based on the current system time from within SystemVerilog because, by the time the seed is set, some constructors may already have been

invoked, initializing the local random source for those instances from the default seed.

Sample 7-23. Generating a random seed

```
#include <stdlib.h>
#include <time.h>
main() {
  srandom(time(NULL));
  printf("%d\n", random());
  exit(0);
}
```

Seed used is displayed.

Whatever seed is being used in a simulation, its value must be known so it can be reused. Simulation scripts should display the value of the seed used at the beginning of a simulation, as shown in Sample 7-24. This display makes it possible to associate the results in an output log file with a particular seed.

Sample 7-24. Random seed display in simulation script

```
...
$seed = `random`;
print "Random seed is $seed\n";
`simv +ntb_random_seed=$seed ...`
```

Associate output files with seeds.

Simulation results are the product of a simulation run with a specific seed. Performing another simulation run, using the exact same code, but with a different seed will yield different results. It is therefore important to associate results with a specific seed. Once this association exists, results can be reproduced. They also can be graded to identify which seeds contribute most toward the final verification objective.

Include the seed in all output file names.

If the same output filename is used by two simulations of the same code but using different seeds, the results from the first simulation will be lost. You should include the seed value in all output pathnames. This technique can be done by putting all output files in a seed-specific directory or by including the seed value in the filename itself.

REGRESSION

A regression ensures backward compatibility.

A regression suite ensures that modifications to the design remain backward compatible with previously verified functionality. Many times, a change in the design made to fix a problem detected by a

testcase, will break functionality that was verified previously. Once a testbench is written and debugged to simulate successfully, you must make sure that it continues to be successful throughout the duration of the design project.

Running Regressions

Regressions are run at regular intervals.

As individual self-checking testbenches are completed, they are added to a master list of testbenches included in the regression simulation. This regression simulation is run at regular intervals, usually nightly. For directed testcases, simulations are run one after another. For random-based testbenches, simulations are run repeatedly using different seeds. As the number of testbenches or the size of the functional coverage space grows, it may not be possible to complete a full regression simulation overnight.

Divide directed testcases into two groups.

Directed testcases can then be classified into two groups: One group is run every night, while the second group is included only in regressions run over a weekend. Testcases selected for the first group should be the ones that verify the basic functionality of the design.

Rank seeds.

With random-based testbenches, rank seeds based on their incremental contribution to the overall functional coverage goal. Select the seeds that provide the greatest contribution and start the regression simulation with those. If there is still time left in the regression period, continue with additional, randomly selected seeds.

Testbenches may have a fast mode to speed up regressions.

Another approach is to provide a *fast mode* to testcases where only a subset of the functionality is verified during overnight regression simulations, or simulations are run for shorter periods of time with the same seed. The full-length simulation would be performed only during individual simulations or regression simulations over a weekend. The fast mode could be turned on using a user-defined command-line option, as shown in Sample 7-25. .

Use a script to run regressions.

A regression script could invoke each testbench in the regression test suite, for a specific number of seeds, using the simulation configuration script used to invoke individual simulations, as discussed in “Configuration Management” on page 355. If the number and duration of testbenches in the regression suite make it impossible to run a regression simulation in the allotted time, you will want to consider parallel simulations. If you do, it is necessary that test-

Sample 7-25. Implementing a fast mode

```
% simv ... +fastmode

program testcase;
...
initial
begin
   // Repeat only 4 times in fast mode
   repeat (($test$plusarg("+fastmode"))?4:256)
      begin
         ...
      end
   syslog.terminate;
end
endprogram
```

benches be designed to produce results independently from each other, as discussed in "Output File Management" on page 361. Parallel simulations can be managed using readily available utilities, such as *pmake* or *LSF*.

Regression Management

Check out a fresh view with local copies.

Not all source files are suitable for regression runs. If you are using your revision control system properly, you should be checking in files at times convenient for you, not convenient for the regression run. The latest version of a file might contain code that was not verified at all or that might even have syntax errors. You do not want to waste a regression simulation on files that were not debugged properly. Before running a regression, you should checkout a complete view of the source control database, populated with local copies whose revisions are tagged as being suitable for regression testing. This tag is applied by verification and design engineers once they have confidence in the basic functionality of the code and are ready to submit that particular revision of the testbench or the design to regression. Sample 7-26 shows an example of tagging a particular view of a file system, then checkout the particular files associated with a tag at a later time using CVS.

Sample 7-26. Tagging and retrieving a particular revision of a view

```
% cvs tag -R regress
...
% cvs update -dA -rregress
```

Put a time bomb in all simulations.

One of the greatest killers of regression simulations, second only to the infinite loop, is the simulation that never terminates. A simulation will run forever if a condition you are waiting for never occurs. The clock generator keeps the simulation alive by generating events continuously. Time advances until the maximum value is reached, which, in modern simulators using 64-bit time values, will take a long time! To prevent a testcase from hanging a regression simulation, include a time bomb in all simulations. This time bomb should go off after a delay long enough to allow the normal operations of the testcase to complete without interruption. Sample 7-27 shows a time bomb, included in the verification harness in Sample 7-28. The time bomb could be modified to include a virtual interface that, when observed with an event, would reset the fuse.

Sample 7-27. Time bomb class

```
class timebomb;
   function new(time fuse);
      #(fuse);
      $write("Boom!\n");
      $finish;
   endfunction
endclass: timebomb;
```

Sample 7-28. Using the time bomb

```
class harness;
   timebomb bomb = new(12ms);
   ...
endclass: harness
```

Do not rely on a time bomb for normal termination.

The time bomb should be used only to prevent runaway simulations from running forever. It should not be used to terminate a testcase under normal conditions. It would be impossible to distinguish between a successful completion of the testcase and a deadlock condition. Furthermore, the time bomb would require fine tuning every time the testbench or design is modified to avoid the testcase from being interrupted prematurely or wasting simulation cycles by running for too long.

Automatically generate a report after each regression run.

Once the regression simulations are completed, the success or failure of each testcase in the regression suite should be checked using the output log scan script (see “Pass or Fail?” on page 352.) The results are then summarized into a single regression report outlining which particular testbench and seed was successful or unsuccessful. It is a good idea to have the regression script mail the report to all the engineers in the design team to ensure that the design remains

backward compatible at all times. Reviewing this report also should be the first item on the agenda in any design team meeting.

SUMMARY

Write a transaction-level model to help debug your verification environment sooner and faster.

Transaction-level models are not the same as RTL models but must pass the same verification suite.

Transaction-level models enable system-level verification.

Carefully model exceptions in transaction-level models.

Use a common error reporting mechanism.

Use a script to look for the absence of error messages and the presence of the termination message to declare if a simulation completed successfully.

Manage your models and the components of the verification environment using configuration techniques.

Have simulations report the filename and version number in the output log file.

Have a mechanism for reporting—and later specifying—a seed used for a particular simulation run.

Separate output files for each testbench and each seed used to simulate each testbench.

Run regression simulations at regular intervals, using a tagged version of the design and verification environment.

Include a time bomb in all simulations to prevent a single testbench from blocking an entire regression run.

APPENDIX A CODING GUIDELINES

There have been many sets of coding guidelines published for Verilog, but historically they have focused on the synthesizable subset and the target hardware structures. Writing testbenches involves writing a lot of code and also requires coding guidelines. These guidelines are designed to enhance code maintainability and readability, as well as to prevent common or obscure mistakes.

Guidelines can be customized.

The specifics of a guideline may not be important. It is the fact that it is specified and that everyone does it the same way that is important. Many of the guidelines presented here can be customized to your liking. If you already have coding guidelines, keep using them. Simply augment them with the guidelines shown here that are not present in your own.

Define guidelines as a group, then follow them.

Coding guidelines have no functional benefits. Their primary contribution is toward creating a readable and maintainable design. Having common design guidelines makes code familiar to anyone familiar with the implied style, regardless of who wrote it. The primary obstacle to coding guidelines are personal preferences. It is important that the obstacle be recognized for what it is: personal taste. There is no intrinsic value to a particular set of guidelines. The value is in the fact that these guidelines are shared by the entire group. If even one individual does not follow them, the entire group is diminished.

Enforce them!

Guidelines are just that: guidelines. The functionality of a design or testbench will not be compromised if they are not followed. What

will be compromised is their maintainability. The benefit of following guidelines is not immediately obvious to the author. It is therefore a natural tendency to ignore them where inconvenient. Coding guidelines should be enforced by using a linting tool or code reviews.

See the *Verification Methodology Manual for SystemVerilog.*

The guidelines presented in this appendix are simple coding guidelines. For methodology implementation guidelines, refer to the *Verification Methodology Manual for SystemVerilog*. That book is entirely written as a series of guidelines to implement a state of the art verification methodology using SystemVerilog.

FILE STRUCTURE

Use an identical directory structure for every project.

Using a common directory structure makes it easier to locate design components and to write scripts that are portable from one engineer's environment to another. Reusable components and bus-functional models that were designed using a similar structure will be more easily inserted into a new project.

Example project-level structure:

```
.../bin/      Project-wide scripts/commands
    doc/      System-level specification documents
    SoCs/     Data for SoCs/ASICs/FPGA designs
    boards/   Data for board designs
    systems/  Data for system designs
    mech/     Data for mechanical designs
    shared/   Data for shared components
```

At the project level, there are directories that contain data for all design and testbench components for the project. Components shared, unmodified, among SoC/ASIC/FPGA, board and system designs and testbenches are located in a separate directory to indicate that they impact more than a single design. At the project level, shared components are usually verification and interface models.

Some “system” designs may not have a physical correspondence and may be a collection of other designs (SoCs, ASICs, FPGAs and

boards) artificially connected together to verify a subset of the system-level functionality.

Each design in the project has a similar structure. Example of a design structure for an SoC:

```
SoCs/name/          Data for ASIC named "name"
          doc/      Specification documents
          bin/      Scripts specific to this design
          tl/       Transaction-level model
          rtl/      Synthesizable model
          syn/      Synthesis scripts & logs
          phy/      Physical model and SDF data
          verif/    Verif env and simulation logs
SoCs/shared/        Data for shared ASIC components
```

Components shared, unmodified, between SoC designs are located in a separate directory to indicate that they impact more than a single design. At the SoC level, shared components include bus-functional models, processor cores, soft and hard IP and internally reused blocks.

Use relative pathnames.

Using absolute pathnames requires that future use of a bus-functional model, component or design be installed at the same location. Absolute pathnames also require that all workstations involved in the design have the design structure mounted at the same point in their file systems. The name used may no longer be meaningful, and the proper mount point may not be available.

If full pathnames are required, use preprocessing or environment variables.

Put a Makefile with a default 'all' target in every source directory.

Makefiles facilitate the compilation, maintenance, and installation of a design or model. With a Makefile the user need not know how to build or compile a design to invoke “make all”. Top-level makefiles should invoke *make* on lower level directories.

Example “all” makefile rule:

```
all: subdirs ...

SUBDIRS = ...
subdirs:
     for subdir in $(SUBDIRS); do \
        (cd $$subdir; make); \
     done
```

Use a single module, interface, program or package in a file.

A file should contain a single compilation unit. This will minimize the amount of recompilation in incremental compilation simulators. It will also simplify identifying the location of components in the file system.

Specify files required by the current file using the `*include* directive.

Whether you include a file using the *‘include* directive or by naming it on the command line, the result is the same. A SystemVerilog compilation is the simple concatenation of all of its input files. If each file includes all of the file is depends on to compile successfully, you only need to specify one file on the command line: the top-level file.

Surround source files with *‘ifndef*, *‘define* and *‘endif* directives.

It is very likely that more than one file would depend on the same file. If each file includes all of the file it depends on, the file would included more than once, causing compilation errors. By surrounding source files with conditional compilation directives, it will be compiled only once, even if it is included multiple times.

```
‘ifndef DESIGN__SV
‘define DESIGN__SV

module design(...);
...
endmodule
‘endif
```

Filenames

Name files containing SystemVerilog source code using the *.sv suffix. Name files containing Verilog-2001 source code using the *.v suffix.

SystemVerilog introduced new reserved words that may have been used as user identifiers in Verilog-2001 source code. Tools must have a way to differentiate between the two versions of Verilog.

Use filename extensions that indicate the content of the file.

Tools often switch to the proper language-sensitive function based on the filename extension. Use a postfix on the filename itself if different (but related) contents in the same language are provided. Using postfixes with identical root names causes all related files to show up next to each other when looking up the content of a directory.

Example of poor file naming:

```
design.svt          Testbench
design.svb          Transaction-level model
design.svr          RTL model
design.vg           Gate-level model
```

Example of good file naming:

```
design_tb.sv        Testbench
design_tl.sv        Transaction-level model
design_rtl.sv       RTL model
design_gate.v       Gate-level model
```

STYLE GUIDELINES

Comments

Put the following information into a header comment for each source file: copyright notice, brief description, revision number and maintainer name and contact data.

Example (under RCS or CVS):

```
//
// (c) Copyright MMCC, Company Name
// All rights reserved.
//
// This file contains proprietary and confidential
// information. The content or any derivative work
// can be used only by or distributed to licensed
// users or owners.
//
// Description:
//    This script parses the output of a set of
//    simulation log files and produces a
//    regression report.
//
// Maintainer: John Q. Doe <jdoe@domain.com>
// Revision  : $Revision$
```

Use a trailer comment describing revision history for each source file.

The revision history should be maintained automatically by the source management software. Because these can become quite lengthy, put revision history at the bottom of the file. This location eliminates the need to wade through dozens of lines before seeing the actual code.

Example (under RCS or CVS):

```
//
// History:
//
// $Log$
//
```

Use comments to describe the intent or functionality of the code, not its behavior.

Comments should be written with the target audience in mind: A junior engineer who knows the language, but is not familiar with the design, and must maintain and modify the design without the benefit of the original designer's help.

Example of a useless comment:

```
// Increment i
i++;
```

Example of a useful comment:

```
// Move the read pointer to the next input element
i++;
```

Preface each major section with a comment describing what it does, why it exists, how it works and assumptions on the environment.

A major section could be an *always* or *initial* block, an *interface*, a *clocking* block, a *task*, a *function*, a *program*, a *class* or a *package*.

It should be possible to understand how a piece of code works by looking at the comments only and by stripping out the source code itself. Ideally, it should be possible to understand the purpose and structure of an implementation with the source code stripped from the file, leaving only the comments.

Describe each argument in individual comments.

Describe the purpose, expected valid range, and effects of each module, interface, program, function or task arguments and return value. Whenever possible, show a typical usage.

Example:

```
//
// Function to determine if a testcase
// has been selected to be executed
//
// Example: if (!is_selected("test1")) return;
//
function bit         // TRUE if selected
```

```
is_selected(
   string name);  // Name of the testcase
```

Delete bad code; do not comment-out bad code.

Commented-out code begs to be reintroduced. Use a proper revision control system to maintain a track record of changes.

Layout

Lay out code for maximum readability and maintainability.

Saving a few lines or characters only saves the time it takes to type it. Any cost incurred by saving lines and characters will be paid every time the code has to be understood or modified.

Use a minimum of three spaces for each indentation level.

An indentation that is too narrow (such as 2), does not allow for easily identifying nested scopes. An indentation level that is too wide (such as 8), quickly causes the source code to reach the right margin.

Write only one statement per line.

The human eye is trained to look for sequences in a top-down fashion, not down-and-sideways. This layout also gives a better opportunity for comments.

Limit line length to 72 characters. If you must break a line, break it at a convenient location with the continuation statement and align the line properly within the context of the first token.

Printing devices are still limited to 80 characters in width. If a fixed-width font is used, most text windows are configured to display up to 80 characters on each line. Relying on the automatic line wrapping of the display device may yield unpredictable results and unreadable code.

Example of poor code layout:

```
#10
expect = $realtobits((coefficient * datum)
```

```
            + 0.5);
```

Example of good code layout:

```
#10 expect = $realtobits((coefficient * datum)
                         + 0.5);
```

Use a tabular layout for lexical elements in consecutive declarations, with a single declaration per line.

A tabular layout makes it easier to scan the list of declarations quickly, identifying their types, classes, initial values, etc. If you use a single declaration per line, it is easier to locate a particular declaration. A tabular layout also facilitates adding and removing a declaration.

Example of poor declaration layout:

```
int unsigned counta, countb;
float c = 0.0;
bit [31:0] sum;
logic [6:0] z;
```

Example of good declaration layout:

```
int unsigned counta;
int unsigned countb;
float        c       = 0.0;
bit   [31:0] sum;
logic [ 6:0] z;
```

Declare ports and arguments in logical order according to purpose or functionality; do not declare ports and arguments according to direction.

Group port declarations that belong to the same interface. Grouping port declarations facilitates locating and changing them to a new interface. Do not order declarations output first, data input second, and control signals last because it scatters related declarations.

Use named ports when calling tasks and functions or instantiating modules, interfaces and programs. Use a tabular layout for lexical elements in consecutive associations, with a single association per line.

Using named ports is more robust than using port order. Named ports do not require any change when the argument list of a subroutine or subunit is modified or reordered. Furthermore, named ports provide for self-documenting code as it is unnecessary to refer to another section of the code to identify what value is being passed to which argument. A tabular layout makes it easier to scan the list of arguments being passed to a subprogram quickly. If you use one named port per line, it is easier to locate a particular association. A tabular layout also facilitates adding and removing arguments.

Example of poor association layout:

```
fifo in_buffer(voice_sample_retimed,
               valid_voice_sample, overflow, ,
               voice_sample, 1'b1, clk_8kHz,
               clk_20MHz);
```

Example of good association layout:

```
fifo in_buffer(.data_in   (voice_sample),
               .valid_in  (1'b1),
               .clk_in    (clk_8kHz),
               .data_out  (voice_sample_retimed),
               .valid_out (valid_voice_sample),
               .clk_out   (clk_20MHz),
               .full      (overflow),
               .empty     ());
```

Structure

Encapsulate repeatedly used operations or statements in subroutines.

By using tasks or functions, maintenance is reduced significantly. Code only needs to be commented once and bugs only need to be fixed once. Using subprograms also reduces code volume.

Example of poor expression usage:

```
// sign-extend both operands from 8 to 16 bits
operand1 = {{8 {ls_byte[7]}}, ls_byte};
operand2 = {{8 {ms_byte[7]}}, ms_byte};
```

Example of proper use of subprogram:

```
// sign-extend an 8-bit value to a 16-bit value
function [15:0] sign_extend;
   input [7:0] value;
   sign_extend = {{8 {value[7]}}, value};
endfunction

// sign-extend both operands from 8 to 16 bits
operand1 = sign_extend(ls_byte);
operand2 = sign_extend(ms_byte);
```

Use a maximum of 50 consecutive sequential statements in any statement block.

Too many statements in a block create many different possible paths. This layout makes it difficult to grasp all of the possible implications. It may be difficult to use a code coverage tool with a large statement block. A large block may be broken down using subprograms.

Use no more than three nesting levels of flow-control statements.

Understanding the possible paths through several levels of flow control becomes difficult exponentially. Too many levels of decision making may be an indication of a poor choice in processing sequence or algorithm. Break up complex decision structures into separate subprograms.

Example of poor flow-control structure:

```
if (a == 1 && b == 0) begin
   case (val)
   4:
   5: while (!done) begin
         if (val % 2) begin
            odd = 1;
            if (choice == val) begin
               for (j = 0; j < val; j++) begin
```

```
                        select[j] = 1;
                    end
                    done = 1;
                end
            end
            else begin
                odd = 0;
            end
        end
    0: for (i = 0; i < 7; i++) begin
            select[j] = 0;
        end
    default:
        z = 0;
    endcase
end
```

Example of good flow-control structure:

```
function void
process_selection(int val);
   odd = 0;
   while (!done) begin
      if (val % 2) begin
         odd = 1;
      end
      if (odd && choice == val) begin
         for (j = 0; j < val; j++) begin
            select[j] = 1;
         end
         done = 1;
      end
   end
endfunction: process_selection

if (a == 1 && b == 0) begin
   case (val)
   0: for (i = 0; i < 7; i++) begin
         select[j] = 0;
      end
   4:
   5: process_selection(val);
   default:
       z = 0;
   endcase
end
```

Debugging

Include a mechanism to exclude all debug statements automatically.

Debug information should be excluded by default and should be enabled automatically via a control file or command-line options. Do not comment out debug statements and then uncomment them when debugging. This approach requires significant editing. When available, use a preprocessor to achieve better runtime performance.

Example of poor debug statement exclusion:

```
// $write("Address = %h, Data = %d\n",
//         address, data);
```

Example of proper debug statement exclusion:

```
`ifdef DEBUG
   $write("Address = %h, Data = %d\n",
           address, data);
`endif
```

NAMING GUIDELINES

These guidelines suggest how to select names for various user-defined objects and declarations. Additional restrictions on naming can be introduced by more specific requirements.

Capitalization

Use lowercase letters for all user-defined identifiers.

Using lowercase letters reduces fatigue and stress from constantly holding down the Shift key. Reserve uppercase letters for identifiers representing special functions.

Do not rely on case mix for uniqueness of user-defined identifiers.

The source code may be processed eventually by a case-insensitive tool. The identifiers would then lose their uniqueness. Use naming to differentiate identifiers.

Example of bad choice for identifier:

```
typedef struct {
   byte red;
   byte green;
   byte blue;
} RGB;
...
begin
   RGB rgb;
   ...
end
```

Example of better choice for identifier:

```
typedef struct {
   byte red;
   byte green;
   byte blue;
} rgb_typ;
...
begin
   rgb_typ rgb;
   ...
end
```

Use uppercase letters for constant identifiers (runtime or compile-time).

The case differentiates between a symbolic literal value and a variable.

Example:

```
module block(...);
...
`define DEBUG
parameter WIDTH = 4;
typedef enum {RESET_ST, RUN_ST, ...} state_typ;
...
endmodule
```

Separate words using an underscore; do not separate words by mixing upper-case and lowercase letters

It can be difficult to read variables that use case to separate word boundaries. Using spacing between words is more natural. In a case-insensitive language or if the code is processed through a case-insensitive tool, the case convention cannot be enforced by the compiler.

Example of poor word separation:

```
int readIndexInTable = 0;
```

Example of proper word separation:

```
int read_index_in_table = 0;
```

Identifiers

Do not use reserved words of popular languages or languages used in the design process as user-defined identifiers.

Not using reserved words as identifiers avoids having to rename an object to a synthetic, often meaningless, name when translating or generating a design into another language. Popular languages to consider are C, C++, VHDL, PERL, OpenVera and *e*.

Use meaningful names for user-defined identifiers, and use a minimum of five characters.

Avoid acronyms or meaningless names. Using at least five characters increases the likelihood of using full words.

Example of poor identifier naming:

```
if (e = 1) begin
   c = c + 1;
end
```

Example of good identifier naming:

```
if (enable = 1) begin
   count = count + 1;
```

```
end
```

Name objects according to function or purpose; avoid naming objects according to type or implementation.

This naming convention produces more meaningful names and automatically differentiates between similar objects with different purposes.

Example of poor identifier naming:

```
count8 <= count8 + 8'h01;
```

Example of good identifier naming:

```
addr_count <= addr_count + 8'h01;
```

Do not use identifiers that are overloaded or hidden by identical declarations in a different scope.

If the same identifier is reused in different scopes, it may become difficult to understand which object is being referred to.

Example of identifier overloading:

```
reg [7:0] address;

begin: decode
   integer address;

   address = 0;
   ...
end
```

Example of good identifier naming:

```
reg [7:0] address;

begin: decode
   integer decoded_address;
```

```
   decoded_address = 0;
   ...
end
```

Use *this.* when referring to data members.

Explicitly using *this.* documents that you are referring to a data members instead of a variable currently in scope. Variables in scope are usually declared nearby whereas data members can be inherited and their declarations located in different files. Furthermore, it avoid having to come up with artificially different names for the same thing in method arguments.

Example of data member reference:

```
class bfm;
   virtual interf sigs;
   function new(virtual interf sigs);
      this.sigs = sigs;
   endfunction: new
endclass: bfm
```

Use suffixes to differentiate related identifiers semantically.

The suffix could indicate object kind such as: type, constant, signal, variable, flip-flop etc., or the suffix could indicate pipeline processing stage or clock domains.

Name all *begin* blocks.

Declarations inside an unnamed block cannot be accessed using hierarchical references. Naming a block makes it possible to be explicitly disabled. If a block is not named, some features in debugging tools may not be available. Labeling also provides for an additional opportunity to document the code.

Example of a named block:

```
foreach (data[i]) begin: scan_bits_lp
   ...
end
```

Label closing “end” keywords.

The start and end of a block may be separated by hundreds of lines. Labeling matching end keywords facilitates recognizing the end of a particular construct.

Example:

```
module FIFO(...);
...
endmodule: FIFO
```

Constants

Use symbolic constants instead of “magic” hard-coded numeric values.

Numeric values have no meaning in and of themselves. Symbolic constants add meaning and are easier to change globally. This result is especially true if several constants have an identical value but a different meaning. Parameters, enumerals and *‘define* symbols.

Example of poor constant usage:

```
int table[256];

for (i = 0; i <= 255; i++) ...
```

Example of good constant usage:

```
parameter TABLE_LENGTH = 256;

int table[TABLE_LENGTH];

for (i = 0; i < TABLE_LENGTH; i++) ...
```

Number multi-bit objects using the range N:0.

Using this numbering range avoids accidental truncation of the top bits when assigned to a smaller object. This convention also provides for a consistent way of accessing bits from a given direction. If the object carries an integer value, the bit number represents the power-of-2 for which this bit corresponds.

Example:

```
parameter width = 16;

reg [      7:0] byte;
reg [     31:0] dword;
reg [width-1:0] data;
```

Do not specify a bit range when referring to a complete vector.

If the range of a vector is modified, all references would need to be changed to reflect the new size of the vector. Using bit ranges implicitly means that you are referring to a subset of a vector. If you want to refer to the entire vector, do not specify a bit range.

Example of poor vector reference:

```
bit [15:0] count;
...
count[15:0] <= count[15:0] + 1;
carry <= count[15];
```

Example of proper vector reference:

```
bit [15:0] count;
...
count <= count + 1;
carry <= count[15];
```

Preserve names across hierarchical boundaries.

Preserving names across hierarchical boundaries facilitates tracing signals up and down a complex design hierarchy. This naming convention is not applicable when a unit is instantiated more than once or when the unit was not designed originally within the context of the current design.

It will also enable the use of the implicit port connection capability of SystemVerilog.

PORTABILITY GUIDELINES

Start every module with a `*resetall* directive.

Compiler directives remain active across file boundaries. A module inherits the directives defined in earlier files. This inheritance may create compilation-order dependencies in your model. Using the *`resetall* directive ensures that your module is not affected by previously defined compiler directives and will be self-contained properly.

Avoid using `*define* symbols.

`define* symbols are global to the compilation and may interfere with other symbols defined in another source file. For constant values, use parameters. If *`define symbols must be used, undefine them by using *`undef* when they are no longer needed.

Example of poor style using `*define* symbols:

```
`define CYCLE 100
`define ns    * 1
always
begin
   #(`CYCLE/2 `ns);
   clk = ~clk;
end
```

Example of good style avoiding `*define* symbols:

```
parameter CYCLE = 100;
`define ns * 1
always
begin
   #(CYCLE/2 `ns);
   clk = ~clk;
end
`undef ns
```

Minimize identifiers in shared name spaces.

A shared name space is shared among all of the components implemented using the same language. When components define the

same identifier in a shared name space, a collision will occur when they are integrated in the same simulation. Minimize your consumption of shared name spaces.

SystemVerilog has two shared name space: *$root* (module names, program names, interface names, primitive names, package names, global class names, global task and function names, etc...) and *`define* symbols.

Use prefixes to differentiate identifiers in shared space.

When declaring an identifier in a shared name space, prefix it with a unique prefix that will ensure it will not collide with a similar identifier declared in another component. The suffix used has to be unique to the author or the authoring group or organization.

Example of poor shared identifier naming:

```
`define DEBUG
```

Example of good shared identifier naming:

```
`define MII_DEBUG
```

Use a nonblocking assignment for variables used outside the *always* or *initial* block where the variable was assigned.

Using nonblocking assignments prevents race conditions between blocks that read the current value of the variable and the block that updates the variable value. This assignment guarantees that simulation results will be the same across simulators or with different command-line options.

Example of coding creating race conditions:

```
always @ (s)
begin
   if (s) q = q + 1;
end

always @ (s)
begin
   $write("Q = %b\n", q);
end
```

Example of good portable code:

```
always @ (s)
begin
   if (s) q <= q + 1;
end

always @ (s)
begin
   $write("Q = %b\n", q);
end
```

Assign variables from a single *always* or *initial* block.

Assigning variables from a single block prevents race conditions between blocks that may be setting a variable to different values at the same time. This assignment convention guarantees that simulation results will be the same across simulators or with different command-line options.

Example of coding that creates race conditions:

```
always @ (s)
begin
   if (s) q <= 1;
end

always @ (r)
begin
   if (r) q <= 0;
end
```

Example of good portable code:

```
always @ (s or r)
begin
   if (s)      q <= 1;
   else if (r) q <= 0;
end
```

Do not disable tasks with output or inout arguments.

The return value of output or inout arguments of a task that is disabled is not specified in the SystemVerilog standard. Use the *return* statement or disable an inner begin/end block instead. This technique guarantees that simulation results will be the same across simulators or with different command-line options.

Example of coding with unspecified behavior:

```
task cpu_read(output [15:0] rdat);
   ...
   if (data_rdy) begin
      rdat = data;
      disable cpu_read;
   end
   ...
endtask
```

Example of good portable code:

```
task cpu_read(output [15:0] rdat);
   ...
   if (data_rdy) begin
      rdat = data;
      return;
   end
   ...
endtask
```

Do not disable blocks containing nonblocking assignments with delays.

What happens to pending nonblocking assignments performed in a disabled block is not specified in the SystemVerilog standard. Not disabling this type of block guarantees that simulation results will be the same across simulators or with different command-line options.

Example of coding with unspecified behavior:

```
begin: drive
   addr <= #10 16'hZZZZ;
   ...
```

```
end

always @ (rst)
begin
   if (rst) disable drive;
end
```

Do not read a wire after updating a register in the right-hand side of a continuous assignment, after a delay equal to the delay of the continuous assignment.

The Verilog standard does not specify the order of execution when the right-hand side of a continuous assignment is updated. The continuous assignment may be evaluated at the same time as the assignment or in the next delta cycle.

If you read the driven wire after a delay equal to the delay of the continuous assignment, a race condition will occur. The wire may or may not have been updated.

Example creating a race condition:

```
assign qb = ~q;
assign #5 qq = q;
initial
begin
   q = 1'b0;
   $write("Qb = %b\n", qb);
   #5;
   $write("QQ = %b\n", qq);
end
```

Do not use the bitwise operators in a Boolean context.

Bitwise operators are for operating on bits. Boolean operators are for operating on Boolean values. They are not always interchangeable and may yield different results. Use the bitwise operators to indicate that you are operating on bits, not for making a decision based on the value of particular bits.

Some code coverage tools cannot interpret a bitwise operator as a logical operator and will not provide coverage on the various com-

ponents of the conditions that caused the execution to take a particular branch.

Example of poor use of bitwise operator:

```
reg [3:0] BYTE;
reg VALID
if (BYTE & VALID) begin
   ...
end
```

Example of good use of Boolean operator:

```
reg [3:0] BYTE;
reg VALID
if (BYTE != 4'b0000 && VALID) begin
   ...
end
```

APPENDIX B GLOSSARY

ASIC	Application-specific integrated circuit.
ATM	Asynchronous Transfer Mode.
ATPG	Automatic test pattern generation.
BFM	Bus-functional model.
CAD	Computed aided design.
CPU	Central processing unit.
CRC	Cyclic redundancy check.
CTS	Clear to send.
DFT	Design for test.
DFV	Design for verification.
DSP	Digital signal processing.
DTR	Data terminal ready.
EDA	Electronic design automation.

EPROM	Erasable programmable read-only memory.
FAA	Federal aviation agency (US government).
FCS	Frame check sequence (ethernet).
FIFO	First in, first out.
FPGA	Field-programmable gate array.
FSM	Finite state machine.
GB	Gigabytes.
Gb	Gigabits.
GMII	Gigabit Medium-Independent Interface (ethernet)
ID	Identification.
HDL	Hardware description language
HEC	Header error check (ATM).
HVL	Hardware verification language.
IEEE	Institute of electrical and electronic engineers.
IP	Internet protocol, intellectual property
LAN	Local area network.
LFSR	Linear feedback shift register.
LLC	Link layer control (ethernet).
MAC	Media access control (ethernet).
MII	Media independent interface (ethernet).
MPEG	Moving picture expert group.

NASA	National aeronautic and space agency (US government).
NNI	Network-network interface (ATM).
OO	Object-oriented.
OOP	Object-oriented programming.
OVL	Open verification library.
PC	Personal computer.
PCI	PC component interface.
PLL	Phase-locked loop.
RAM	Random access memory.
RGB	Red, green and blue (video).
ROM	Read-only memory.
RMII	Reduced Medium-Independent Interface (ethernet)
RTL	Register transfer level.
SDF	Standard delay file.
SDH	Synchronous digitial hierarchy (european SONET).
SMII	Serial Medium-Independent Interface (ethernet)
SNAP	(ethernet).
SOC	System on a chip.
SOP	Subject-oriented programming.
SONET	Synchronous optical network (north-american SDH).
UART	Universal asynchronous receiver transmitter.

UFO	Unidentified flying object.
UNI	User-network interface (ATM).
VHDL	VHSIC hardware description language.
VHSIC	Very high-speed integrated circuit
VLAN	Virtual local area network (ethernet).
VMM	Verification Methodology Manual for SystemVerilog (published by Springer).
VPI	Virtual path identifier (ATM).
XAUI	Ten gigabit Adaptor Universal Interface (ethernet)
XGMII	Ten Ggabit Medium-Independent Interface (ethernet)

INDEX

S

W

Z