PREFACE

If you survey hardware design groups, you will learn that between 60% and 80% of their effort is now dedicated to verification. Unlike synthesizeable 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 absence of constraints and historical lack of available expertise and references in verification has resulted in ad hoc approaches. The consequences of an informal 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

Take a survey of the books about Verilog or VHDL currently available. You will notice that the majority of the pages are devoted to explaining the details of the languages. In addition, several chapters are focused on the synthesizeable—or RTL—coding style replete with examples. Some books are even devoted entirely to the subject of RTL coding.

When verification is addressed, only one or two chapters are dedicated to the topic. And often, the primary focus is to introduce more language constructs. Verification is usually presented in a very rudimentary fashion, using simple, non-scalable techniques that become tedious in large-scale, real-life designs.

The first edition of this book was the first book specifically devoted to functional verification techniques for hardware models. Since then, several other verification-only books have appeared. Major conferences now include verification tracks. Universities, in collaboration with industry, are now offering verification courses in their engineering curriculum. Pure verification EDA companies are now offering new tools to improve productivity and the overall design quality. 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 this second edition, I will present the latest verification techniques that are successfully being used to produce fully functional first-silicon ASICs, systems-on-a-chip (SoC), boards and entire systems. It builds on the content of the first edition—transaction-level self-checking testbenches—to introduce a revolution in functional verification: coverage-driven constrainable random testbenches.

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 VHDL and Verilog language semantics that are often overlooked or oversimplified in textbooks intent on describing the synthesizeable subset. These unfamiliar semantics become important in understanding what makes a well-implemented and robust testbench and in providing the necessary control and monitor features. Once these new semantics are understood in a familiar language, the same semantics are presented in new verification-oriented languages.

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()* function in whatever language is used to implement them. 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.

Behavioral 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. For many, behavioral modeling is synonymous with synthesizeable or RTL modeling. In this book, the term “behavioral” is used to describe any model that adequately emulates the functionality of a design, usually using non-synthesizeable constructs and coding style.

WHAT PRIOR KNOWLEDGE YOU SHOULD HAVE

This book focuses on the functional verification of hardware designs using VHDL, Verilog, *e* or OpenVera. 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 as a reference along with this book[1]. I do not describe a synthesizeable subset, nor limit the implementation of the verification techniques to using that subset. Verification is a complex task: The power of a 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.

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.

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 are an experienced Verilog or VHDL user, you may wish to skip Chapter 4—but read it anyway, just to make sure your definition of "experienced" matches mine.

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.

1. For Verilog, I recommend *The Verilog Hardware Description Language* by Thomas & Moorby, 3rd edition or later (Kluwer Academic Publisher).

   For VHDL, I recommend *VHDL Coding Styles and Methodologies* by Ben Cohen (Kluwer Academic Publisher).

   For OpenVera, the *OpenVera Language Reference Manual* is available at http://Open-Vera.com. Vera users will find the *Vera Users Manual* available under $VERA_HOME/doc/vum.

   For *e*, Specman Elite users will find the *e Language Reference Manual* under the HELP menu. It can also be found at https://verificationvault.com.

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

CHOOSING A LANGUAGE

The first decision a design group is often faced with is deciding which language to use. As the author of this book, I faced the same dilemma. In many cases, the choice does not exist as the company has selected a language over others and has invested heavily into supporting that language in terms of licenses, training and intellectual property. But for small companies, companies in transition or companies without a central CAD group, the answer is usually dictated by the decision maker's own knowledge or personal preference.

VHDL vs. Verilog

In my opinion, VHDL and Verilog are inadequate by themselves, especially for verification. They are both equally poor for synthesizeable description. Some things are easier to accomplish in one language than in the other. For a *specific* model, one language is better than the other: One language has features that map better to the functionality to be modeled. However, as a general rule, neither is better than the other.

Some sections are Verilog only. 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 learning curve is not as steep, it is true. However, all languages provide similar concepts: sequential statements, parallel constructs, structural constructs and the illusion of parallelism.

For all languages, 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. Both 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.

Hardware Verification Languages

Hardware verification languages (HVLs) are languages that were specifically designed to implement testbenches efficiently and productively. As I write this book, there are several to choose from. Commercial solutions include *e* from Verisity, OpenVera from Synopsys and RAVE from Forte Design. 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. Verification extensions to the Verilog language are also being added in SystemVerilog. Not all support a coverage-driven constrainable random verification strategy (see “Coverage-Driven Random-Based Approach” on page 109) equally well. Many are still better suited to a directed test strategy (see “Directed Testbenches Approach” on page 104).

Switching from Verilog or VHDL to an HVL involves more than simply learning a new syntax. Although one can continue to use an HDL-like directed methodology with an HVL, using an HVL requires a shift in the way verification is approached and testbenches are implemented. The directed verification strategy used with Verilog and VHDL is the schematic capture of verification. Using an HVL with a constraint-driven random verification strategy is the synthesis of verification. When used properly, HVLs are an incredible productivity boost (see Figure 2-17 on page 63).

If this book had been written from scratch, I would not have bothered including Verilog or VHDL examples. Because they were already there and they can be useful as a foundation for understanding the new concepts provided by HVLs, I’ve decided to keep these examples. I also took advantage of this second edition to update the Verilog content to reflect the new Verilog-2001 standard.

And the Winner Is...

I know VHDL, Verilog, C++, *e* and OpenVera equally well. I work using all of them. I teach all of them. When asked which one I prefer, I usually answer that I was asked the wrong question. The right question should be, “Which one do I hate the least?” And the answer to that question is, “The one I’m not currently working with.” When working in one language, you do not notice the things that are simple to describe or achieve in that language. Instead, you notice the frustrations and how it would be easy to do it if only you were using the *other* language.

Verification techniques transcend the language used. VHDL, Verilog, *e* and OpenVera are only implementation vehicles. All are used throughout the book, but examples are typically shown in only one language. I trust that a monolingual reader will be able to understand the example in the other languages, even though the syntax is slightly different. In areas where each language requires different approaches or methodologies, I will present each individually. However, I will make no attempt to cover all of the features of each language. This is a methodology book, not a language book.

I have selected *e* and OpenVera because they are the languages I know best and, at the time of writing, are the HVLs that best support a coverage-driven constrainable random verification process. The book was not written as a medium for comparing language features or the number of lines required to implement various functionality. The decision to use one language over another involves more than a mere side-by-side comparison of features and syntax.

The code syntax in all examples, any mentioned language or tool limitations and any example or discussion of tool output or feature is up-to-date and factually correct[2] at the time of writing. For VHDL, VHDL-93 and ModelSim™ *5.5.e* were used. For Verilog, VCSi 6.1 and ModelSim 5.5.e were used. For OpenVera, Vera™ 6.0.0 was used. For *e,* Specman Elite™ 4.1 was used.

2. I welcome correction of any factual errors in the book via email at `janick@bergeron.com`. These corrections will be posted in the *errata* section of the book website.

A common complaint I received about the first edition was the lack of complete examples. You'll notice that in this edition, like the first, code samples are still provided only as excerpts. I fundamentally 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. Instead, the entire source code for all examples can now be found in the *source* section at the following URL:

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

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.[3]

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. Chris Macionski, Andrew Piziali, Chris Spear, Ben Cohen and Grant Martin, my technical reviewers, gave it light from its stubborn darkness. And FrameMaker, my word processing software, once more reliably went where no Word had gone before!

I also thank Mentor Graphics for supplying licenses for their ModelSim™ HDL simulator, Verisity Design Ltd. for supplying

3. 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.

licenses for their Specman Elite™ environment, and Synopsys Inc. for supplying licenses for their VCS™ and Vera™ simulators.