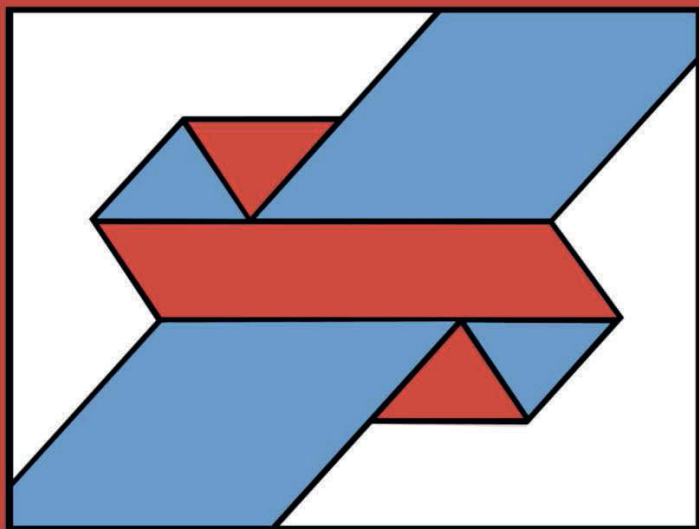




Hardware Verification with SystemVerilog

An Object-Oriented Framework

Mike Mintz
Robert Ekendahl

 Springer

Hardware Verification with SystemVerilog

An Object-Oriented Framework

Mike Mintz
Robert Ekendahl

Hardware Verification with SystemVerilog

An Object-Oriented Framework

Cover art from the original painting “Dementia #10” by John E.
Bannon, johnebannon.com



Mike Mintz
Harvard, MA
USA

Robert Ekendahl
Somerville, MA
USA

Library of Congress Control Number: 2007923923

ISBN 0-387-71738-2
ISBN 978-0-387-71738-8

e-ISBN 0-387-71740-4
e-ISBN 978-0-387-71740-1

Printed on acid-free paper.

© 2007 Springer Science+Business Media, LLC

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, LLC, 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.

9 8 7 6 5 4 3 2 1

springer.com

*For Joan, Alan, and Brian.
Thanks again for your patience.*

Mike

*For Chantal.
Thanks again for your understanding,
love, and active support.*

*And to Newton—and now Darwin.
For many more missed walks.*

Robert

Contents

Preface xix
Acknowledgments xxi

Chapter 1: Introduction 1
 Background 3
 What is Functional Verification?. 4
 Why Focus on SystemVerilog? 5
 A Tour of the Handbook 5
 For Further Reading 6

Part I:
SystemVerilog and Verification
(The Why and How) 7

Chapter 2: Why SystemVerilog? 9
 Overview 10
 SystemVerilog as a Verification Language 11
 Main Benefits of Using SystemVerilog 13
 Drawbacks of Using SystemVerilog 13
 SystemVerilog Traps and Pitfalls 14
 SystemVerilog is not Verilog 14
 Errors and run-time crashes 15

Contents

Five languages in one!	15
The assertions language	15
The constraint language	16
The coverage language	18
SystemVerilog features not discussed	19
Summary	20
For Further Reading	20
 Chapter 3: OOP and SystemVerilog	23
Overview	24
The Evolution of OOP and SystemVerilog	25
Assembly programming: The early days	25
Procedural languages: The next big step	25
OOP: Inheritance for functionality	26
OOP: Inheritance for interface	28
A word or two about “interface”	28
The Evolution of Functional Verification	29
Verification through inspection	29
Verification through randomness	29
The emergence of hardware verification languages	30
OOP: A current trend in verification	31
OOP: A possible next step	31
OOP and SystemVerilog	32
Data abstraction through classes	32
A DMA descriptor example	32
Access control	33
Constructors	34
Member methods and variables	35
Inheritance for functionality	36
Inheritance for code interface	37
What’s a header file?.	39
Packages	40
Separating HDL and testbench code	42

Wiggling wires: the interface concept.	42
Building and using interfaces	44
Summary	46
For Further Reading	46

Chapter 4: A Layered Approach 47

Overview.	48
A Whiteboard Drawing	50
An “ends-in” approach.	51
Refining the whiteboard blocks	52
The “Common-Currency” Components	52
The Component Layer in Detail	53
The connection layer.	54
The agent layer	56
The transaction layer	57
The Top-Layer Components	58
What is a Test?	60
The Test Component	62
The Test Irritator	64
A Complete Test	65
Summary	67
For Further Reading	67

Part II: An Open-Source Environment with SystemVerilog69

Chapter 5: Teal Basics 71

Overview.	72
Teal’s Main Components.	72
Using Teal	74
A simple test	74
Logging Output.	74
Using Test Parameters	77
Accessing Memory	79

Contents

A memory example	80
Constrained Random Numbers	84
Required initialization	84
Using random numbers.	85
Working with Simulation Events	86
Summary	87

Chapter 6: Truss: A Standard Verification Framework 89

Overview	90
General Considerations	91
SystemVerilog considerations	91
Keeping it simple	92
Major Classes and Their Roles	93
Key test algorithm: The “dance”	94
The verification_component Virtual Base Class	97
Detailed Responsibilities of the Major Components	98
The testbench class	99
Watchdog timer	101
Test class	102
Test Component and Irritator Classes	106
The test component virtual base class	106
An AHB example	108
Test-component housekeeping functionality	109
The irritator virtual base class	110
Using the irritator.	112
Summary	113

Chapter 7: Truss Flow 115

Overview.	116
About truss_verification_top.sv	116
The Test Component Dance	119
The Irritator Dance	121
Compiling and Running Tests.	122
The truss run script.	123
Switches	124
Using “-f” files.	125
The First Test: A Directed Test	125
The Second Test:	
Adding Channels and Random Parameters	127
The channel pseudo-templated classes	128
Building the second test.	129
Building the second test’s test_component	131
Adjusting the second test’s parameters	132
The Remaining Tests:	
Mix-and-Match Test Components	135
Summary.	136

Chapter 8: Truss Example 137

Overview.	138
Directory Structure	138
Theory of Operation	140
Running the Simple ALU Example.	142
Points of Interest	142
Power-on Reset	143
Driver and Monitor Protocol	144
The alu_test_component	145
Checking the Chip.	146
Completing the Test	147
Summary.	149

Part III: Using OOP for Verification (Best Practices). 151

Chapter 9: Thinking OOP 153

Overview.	154
Sources of Complexity	155
Essential complexity vs. implementation complexity	155
Flexibility vs. complexity	156
Apparent simplicity vs. hiding inherent complexity.	159
Example: How hiding complexity can create confusion.	159
Example: How apparent simplicity leads to later problems	160
Team dynamics	162
Team roles	162
Using a “code buddy”	163
Creating Adaptable Code	163
Achieving adaptability.	163
Why is adaptability tricky?	164
Architectural Considerations to Maximize Adaptability	165
Changes are easy—or just plain impossible.	166
Where is adaptation likely to happen?	167
Separating Interface from Implementation	168
Code Interface, Implementation, and Base Classes	169
Summary.	170
For Further Reading	171

Chapter 10: Designing with OOP 173

Overview	174
Keeping the Abstraction Level Consistent	174
Using “Correct by Construction”	176
The Value of Packages	178
Data Duplication—A Necessary Evil	180
Designing Well, Optimizing Only When Necessary	181
Using the Protocol, Only the Protocol	182
Verification Close to the Programming Model	183
The Three Parts of Checking	184
Separating the Test from the Testbench	186
Summary	187
For Further Reading	188

Chapter 11: OOP Classes 189

Overview	190
Defining Classes	191
How Much Electricity?	191
Classes	192
Packages	192
Pointers and virtual functions	192
Global Services	193
Package it up!	193
Static methods	194
Singletons—A Special Case of Static Methods	194
Packages or static methods?	195
Other considerations	196
Class Instance Identifiers	197
Strings as identifiers	197
Static integers as identifiers	197
Combination identifiers	198
Class Inheritance for Reuse	198
A BFM base-class example	199

Contents

A BFM agent class	200
Reusing the BFM class	200
Class Inheritance for Code Interfaces	201
Inheritance for a verification component	201
Inheritance for a payload code interface.	202
Summary	203
For Further Reading	204
 Chapter 12: OOP Connections	205
Overview	206
How Tight a Connection?	207
Types of Connections.	209
Peer-to-peer connections.	209
Master-to-slave and push-vs.-pull connections	209
Two Tight Connection Techniques	211
Using pointers	211
Using inheritance	212
Threads and Connections	214
Events—explicit blocking interconnects.	214
Hiding the thread block in a method	216
Fancier Connections	217
Listener or callback connections	218
Channel connections	219
Action object connections	220
Summary	221
For Further Reading	222

Chapter 13: Coding OOP	223
Overview	224
“If” Tests—A Necessary Evil	224
“If” tests and abstraction levels	225
“If” tests and code structure	226
Repeated “if” expressions	227
“If” tests and factory functions	228
A factory function example	229
Coding Tricks	232
Coding only what you need to know	232
Reservable resources	233
The register: an int by any other name	234
Using data members carefully	234
Coding Idioms	236
The singleton idiom	237
Public nonvirtual methods:	
Virtual protected methods	238
Enumeration for Data, Integer for Code Interface	240
What’s in a Name?	241
Keeping class name the same as file name	241
Keeping class and instance names related	241
Coding with Style	242
Proceeding with caution	243
General syntax conventions	243
Identifying local and protected members	244
Summary	245
For Further Reading	246

Part IV:	
Examples	
(Putting It All Together)	247
 Chapter 14: Block-Level Testing	 249
Overview	250
Theory of Operation	251
Verification environment	252
Verification IP	253
UART VIPs	253
Wishbone VIP	254
The verification dance	255
Running the UART Example	255
Points of Interest	256
Configuration	256
VIP UART package	257
VIP UART configuration class	258
Randomization of parameters	258
UART 16550 configuration class	260
Configuring the Chip	261
Register access	262
The wishbone_memory_bank and wishbone_driver	263
Traffic Generation	265
The generator_agent and uart_bfm_agent classes	265
The Checker	267
Checking the data	268
Connecting It All Together	270
The testbench	270
Building the channels	271
Building the configuration and interface port	271
Building the component-layer objects	273
The wishbone objects	274
The test component	275

The uart_basic_test_component::do_randomize() method.	277
The basic data test	278
More Tests.	280
Summary.	280

Chapter 15: Chip-Level Testing 281

Overview.	282
Theory of Operation	282
Verification environment	283
Running the UART Example.	284
The quad_uart_test_components Test	284
The quad_uart_irritators Test	286
UART irritator class.	286
The test	288
The quad_uart_vectors Test	292
The block_uart Test	293
Summary.	293

Chapter 16: Things to Remember 295

Part I: Use SystemVerilog and Layers!	296
Part II: An Open-Source Approach	296
Part III: OOP—Best Practices	297
Part IV: Examples—Copy and Adapt!	298
Conclusion to the Conclusion.	298

Index 301

Preface

This is the second of our books designed to help the professional verifier manage complexity. This time, we have responded to a growing interest not only in object-oriented programming but also in SystemVerilog. The writing of this second handbook has been just another step in an ongoing masochistic endeavor to make your professional lives as painfree as possible.

The authors are not special people. We have worked in several companies, large and small, made mistakes, and generally muddled through our work. There are many people in the industry who are smarter than we are, and many coworkers who are more experienced. However, we have a strong desire to help.

We have been in the lab when we bring up the chips fresh from the fab, with customers and sales breathing down our necks. We’ve been through software bring-up and worked on drivers that had to work around bugs¹ in production chips.

What we feel makes us unique is our combined broad experience from both the software and hardware worlds. Mike has over 20 years of experience from the software world that he applies in this book to hardware verification. Robert has over 12 years of experience with hardware verification, with a focus on environments and methodology.

What we bring to the task of functional verification is over three decades of combined experience, from design, verification, software development, and management. It is our experiences that speak in this handbook. It is our desire that others might learn and benefit from these experiences.

We have had heated discussions over each line of code in this book and in our open-source libraries. We rarely agree at first, but by having to argue our cases we arrive at what we feel are smart, efficient, flexible, and simple solutions. Most of these we have “borrowed” from the software industry but have applied to the field of verification.

We believe that the verification industry can benefit from the lessons learned from the software domain. By using industry-standard languages, the verification domain can adapt techniques and code from over twenty calendar years

¹. Features.



of software effort, the scope of which is nothing short of stunning. Many brilliant people have paved the way in the software field. Although the field of verification is much younger, we could benefit greatly from listening, learning, and adapting mature programming techniques to the production of products of the highest quality.

So why do we provide open-source software at our website, www.trusster.com? Open-source software is a key to uniting and increasing the productivity of our industry. There is almost no successful closed-source (“hard macro”) intellectual property (IP), for a good reason. Without the ability to look at the source and edit as necessary, the task is much more difficult and the chances for success are slim.

We hope that you enjoy this book—and better yet, find its principles increasingly useful in daily practice. We look forward to your comments. Please keep in touch with us at www.trusster.com.

Mike Mintz

Robert Ekendahl

Cambridge, Massachusetts, USA

March 2007

Acknowledgments

It takes a village to raise a child, and it takes a village to create a book. There is a core family, and a few relatives, and a whole lot of helpful neighbors and friends. Once again, the authors would like to bow humbly to our village—in particular, to the global verification village.

This, our second book, shares many of the same reviewers and adds some new ones. They provided great comments on almost every chapter, both detailed and “big picture,” helping to improve many sections substantially.

Michael Meyer was once again our main technical editor, turning our gibberish into English and making clear where we were unclear. This book would not have been readable without him.

We are truly grateful for all the reviewers, their time, and their suggestions during both the early and near final stages of the book. In particular, we thank Ed Arthur, Oswaldo Cadenas, Jesse Craig, Simon Curry, Thomas Franco, John Hoglund, Mark Goodnature, Tom Jones, James Keithan, Ajeetha Kumari, David Long, Bryan Morris, Nancy Pratt, Joe Pizzi, Dave Rich, Henrik Scheuer, Chris Spear, Peter Teng, Thomas Tessier, Greg Tierney, Igor Tsapenko, Gerry Ventura, Stephanie Waters, and Andrew Zoneball.

We are also grateful for the support and encouragement of the producers of the HDL simulators. In particular, we thank the following simulator companies—Cadence, Mentor Graphics, and Synopsys—for providing licenses to their products, so we could confirm that the examples in this handbook work.

However, we are getting ahead of ourselves, so before we dive into the practical problem of verification, let's look at how SystemVerilog expresses OOP techniques.

OOP and SystemVerilog

This section shows how SystemVerilog expresses the OOP concepts described above. It describes some of the techniques we use to build a successful verification environment in later chapters. For engineers experienced with other OOP languages such as C++, Vera or “e,” this chapter can serve as a way to map concepts from one language to another.

Data abstraction through classes

Using classes to express data abstraction is an important technique in building large verification systems. Data abstraction, by grouping the data and the operations together, allows engineers to reason about the code.

We will look at a direct memory access (DMA) descriptor class to show how a class can be constructed, then evolved by means of inheritance.

A DMA descriptor example

DMA is a common hardware feature for transferring data from one memory location to another without putting a load on the CPU. In this example, we verify a DMA chip that accepts DMA descriptors, puts them into an on-chip memory array, and then executes them. Each descriptor has a source and destination memory address, as well as the number of bytes (called “length”) to transfer.

In the verification environment, a DMA descriptor could be represented by a small class. The DMA generator is then responsible for building, or *instantiating*, DMA descriptors and “pushing” them to the chip and to the checker.

The following code describes the DMA descriptor class:

```
class descriptor;
    //Constructor
    extern function new (int src, int dest,
                        int length, int status);
    //Code (or Public) Interface:
    extern virtual function void print();
    int source_address_; //public as an example!
    extern virtual function bit equal(descriptor d);
    //Implementation (or local and protected) interface:
    extern local virtual function int unique_id ();
    protected int destination_address_;
    protected int length_;
    protected int status_;
    protected int verif_id_;
endclass
```

The descriptor class is divided into the code (or public) interface and the implementation (or local and protected) interface, as shown by the *access control* level on each line. The next section will explain what access control labels are for.

Access control

The keywords `protected` and `local`, as used in the descriptor class, are SystemVerilog *access control* labels. They indicate how methods and variables following the statements can be used. The absence of a label indicates that the methods and variables that follow are publicly accessible by any code that has access to an instance of the class.¹ A `local` label indicates that only the code inside the class itself can access the variable or method.

The keyword `protected` indicates a private variable or method that can be modified through inheritance. Public, local, and protected can be used to express and enforce the intent of the class quite clearly.

¹. This is an unfortunate default, as a vast majority of classes will have far fewer public methods and variables compared to the local and protected code used to implement the class. Be prepared to type “local” and “protected” a lot.