
SYSTEMVERILOG FOR VERIFICATION

A Guide to Learning the Testbench Language Features

SYSTEMVERILOG FOR VERIFICATION

A Guide to Learning the Testbench Language Features

CHRIS SPEAR
Synopsys, Inc.

 Springer

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

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

Library of Congress Control Number: 2006926262

ISBN-10: 0-387-27036-1
ISBN-13: 9780387270364

e-ISBN-10: 0-387-27038-8
e-ISBN-13: 9780387270388

Printed on acid-free paper.

© 2006 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.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

springer.com

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

Contents

List of Examples	xi
List of Figures	xxi
List of Tables	xxiii
Foreword	xxv
Preface	xxvii
Acknowledgments	xxxiii
1. VERIFICATION GUIDELINES	1
1.1 Introduction	1
1.2 The Verification Process	2
1.3 The Verification Plan	4
1.4 The Verification Methodology Manual	4
1.5 Basic Testbench Functionality	5
1.6 Directed Testing	5
1.7 Methodology Basics	7
1.8 Constrained-Random Stimulus	8
1.9 What Should You Randomize?	10
1.10 Functional Coverage	13
1.11 Testbench Components	15
1.12 Layered Testbench	16
1.13 Building a Layered Testbench	22
1.14 Simulation Environment Phases	23
1.15 Maximum Code Reuse	24
1.16 Testbench Performance	24
1.17 Conclusion	25
2. DATA TYPES	27
2.1 Introduction	27
2.2 Built-in Data Types	27

2.3	Fixed-Size Arrays	29
2.4	Dynamic Arrays	34
2.5	Queues	36
2.6	Associative Arrays	37
2.7	Linked Lists	39
2.8	Array Methods	40
2.9	Choosing a Storage Type	42
2.10	Creating New Types with typedef	45
2.11	Creating User-Defined Structures	46
2.12	Enumerated Types	47
2.13	Constants	51
2.14	Strings	51
2.15	Expression Width	52
2.16	Net Types	53
2.17	Conclusion	53
3.	PROCEDURAL STATEMENTS AND ROUTINES	55
3.1	Introduction	55
3.2	Procedural Statements	55
3.3	Tasks, Functions, and Void Functions	56
3.4	Task and Function Overview	57
3.5	Routine Arguments	57
3.6	Returning from a Routine	62
3.7	Local Data Storage	62
3.8	Time Values	64
3.9	Conclusion	65
4.	BASIC OOP	67
4.1	Introduction	67
4.2	Think of Nouns, not Verbs	67
4.3	Your First Class	68
4.4	Where to Define a Class	69
4.5	OOP Terminology	69
4.6	Creating New Objects	70
4.7	Object Deallocation	74
4.8	Using Objects	76
4.9	Static Variables vs. Global Variables	76
4.10	Class Routines	78
4.11	Defining Routines Outside of the Class	79
4.12	Scoping Rules	81
4.13	Using One Class Inside Another	85
4.14	Understanding Dynamic Objects	87
4.15	Copying Objects	91
4.16	Public vs. Private	95

4.17	Straying Off Course	96
4.18	Building a Testbench	96
4.19	Conclusion	97
5.	CONNECTING THE TESTBENCH AND DESIGN	99
5.1	Introduction	99
5.2	Separating the Testbench and Design	99
5.3	The Interface Construct	102
5.4	Stimulus Timing	108
5.5	Interface Driving and Sampling	114
5.6	Connecting It All Together	121
5.7	Top-Level Scope	121
5.8	Program – Module Interactions	123
5.9	SystemVerilog Assertions	124
5.10	The Four-Port ATM Router	126
5.11	Conclusion	134
6.	RANDOMIZATION	135
6.1	Introduction	135
6.2	What to Randomize	136
6.3	Randomization in SystemVerilog	138
6.4	Constraint Details	141
6.5	Solution Probabilities	149
6.6	Controlling Multiple Constraint Blocks	154
6.7	Valid Constraints	154
6.8	In-line Constraints	155
6.9	The pre_randomize and post_randomize Functions	156
6.10	Constraints Tips and Techniques	158
6.11	Common Randomization Problems	164
6.12	Iterative and Array Constraints	165
6.13	Atomic Stimulus Generation vs. Scenario Generation	172
6.14	Random Control	175
6.15	Random Generators	177
6.16	Random Device Configuration	180
6.17	Conclusion	182
7.	THREADS AND INTERPROCESS COMMUNICATION	183
7.1	Introduction	183
7.2	Working with Threads	184
7.3	Interprocess Communication	194
7.4	Events	195
7.5	Semaphores	199
7.6	Mailboxes	201
7.7	Building a Testbench with Threads and IPC	210

7.8	Conclusion	214
8.	ADVANCED OOP AND GUIDELINES	215
8.1	Introduction	215
8.2	Introduction to Inheritance	216
8.3	Factory Patterns	221
8.4	Type Casting and Virtual Methods	225
8.5	Composition, Inheritance, and Alternatives	228
8.6	Copying an Object	233
8.7	Callbacks	236
8.8	Conclusion	240
9.	FUNCTIONAL COVERAGE	241
9.1	Introduction	241
9.2	Coverage Types	243
9.3	Functional Coverage Strategies	246
9.4	Simple Functional Coverage Example	248
9.5	Anatomy of a Cover Group	251
9.6	Triggering a Cover Group	253
9.7	Data Sampling	256
9.8	Cross Coverage	265
9.9	Coverage Options	272
9.10	Parameterized Cover Groups	274
9.11	Analyzing Coverage Data	275
9.12	Measuring Coverage Statistics During Simulation	276
9.13	Conclusion	277
10.	ADVANCED INTERFACES	279
10.1	Introduction	279
10.2	Virtual Interfaces with the ATM Router	279
10.3	Connecting to Multiple Design Configurations	284
10.4	Procedural Code in an Interface	290
10.5	Conclusion	294
	References	295
	Index	297

List of Examples

Example 1-1	Driving the APB pins	17
Example 1-2	A task to drive the APB pins	18
Example 1-3	Low-level Verilog test	18
Example 1-4	Basic transactor code	22
Example 2-1	Using the logic type	28
Example 2-2	Signed data types	28
Example 2-3	Checking for four-state values	29
Example 2-4	Declaring fixed-size arrays	29
Example 2-5	Declaring and using multidimensional arrays	29
Example 2-6	Unpacked array declarations	30
Example 2-7	Initializing an array	30
Example 2-8	Using arrays with for and foreach loops	31
Example 2-9	Initialize and step through a multidimensional array	31
Example 2-10	Output from printing multidimensional array values	31
Example 2-11	Array copy and compare operations	32
Example 2-12	Using word and bit subscripts together	33
Example 2-13	Packed array declaration and usage	33
Example 2-14	Declaration for mixed packed/unpacked array	34
Example 2-15	Using dynamic arrays	35
Example 2-16	Using a dynamic array for an uncounted list	35
Example 2-17	Queue operations	36
Example 2-18	Declaring, initializing, and using associative arrays	38
Example 2-19	Using an associative array with a string index	39
Example 2-20	Creating the sum of an array	40
Example 2-21	Array locator methods: min, max, unique	41
Example 2-22	Array locator methods: find	41

Example 2-23	Array locator methods	42
Example 2-24	User-defined type-macro in Verilog	45
Example 2-25	User-defined type in SystemVerilog	45
Example 2-26	Definition of uint	45
Example 2-27	Creating a single pixel type	46
Example 2-28	The pixel struct	46
Example 2-29	Using typedef to create a union	47
Example 2-30	Packed structure	47
Example 2-31	A simple enumerated type	48
Example 2-32	Enumerated types	48
Example 2-33	Specifying enumerated values	48
Example 2-34	Incorrectly specifying enumerated values	49
Example 2-35	Correctly specifying enumerated values	49
Example 2-36	Stepping through all enumerated members	50
Example 2-37	Assignments between integers and enumerated types	50
Example 2-38	Declaring a const variable	51
Example 2-39	String methods	52
Example 2-40	Expression width depends on context	53
Example 2-41	Disabling implicit nets with ‘default_nettype none	53
Example 3-1	New procedural statements and operators	55
Example 3-2	Using break and continue while reading a file	56
Example 3-3	Ignoring a function’s return value	56
Example 3-4	Void function for debug	57
Example 3-5	Simple task without begin...end	57
Example 3-6	Verilog-1995 routine arguments	58
Example 3-7	C-style routine arguments	58
Example 3-8	Verbose Verilog-style routine arguments	58
Example 3-9	Routine arguments with sticky types	58
Example 3-10	Passing arrays using ref and const	59
Example 3-11	Using ref across threads	60
Example 3-12	Function with default argument values	61
Example 3-13	Using default argument values	61
Example 3-14	Original task header	61
Example 3-15	Task header with additional array argument	61
Example 3-16	Task header with additional array argument	62
Example 3-17	Return in a task	62
Example 3-18	Return in a function	62
Example 3-19	Specifying automatic storage in program blocks	63
Example 3-20	Static initialization bug	64

Example 3-21	Static initialization fix: use automatic	64
Example 3-22	Time literals and \$timeformat	65
Example 4-1	Simple BusTran class	69
Example 4-2	Declaring and using a handle	71
Example 4-3	Simple use-defined new function	72
Example 4-4	A new function with arguments	72
Example 4-5	Calling the right new function	73
Example 4-6	Allocating multiple objects	74
Example 4-7	Creating multiple objects	75
Example 4-8	Using variables and routines in an object	76
Example 4-9	Class with a static variable	77
Example 4-10	Initializing a static variable in a task	78
Example 4-11	Routines in the class	79
Example 4-12	Out-of-block routine declarations	80
Example 4-13	Out-of-body task missing class name	81
Example 4-14	Name scope	82
Example 4-15	Class uses wrong variable	83
Example 4-16	Using this to refer to class variable	83
Example 4-17	Bug using shared program variable	84
Example 4-18	Statistics class declaration	85
Example 4-19	Encapsulating the Statistics class	86
Example 4-20	Using a typedef class statement	87
Example 4-21	Passing objects	88
Example 4-22	Bad packet creator task, missing ref on handle	89
Example 4-23	Good packet creator task with ref on handle	89
Example 4-24	Bad generator creates only one object	90
Example 4-25	Good generator creates many objects	90
Example 4-26	Using an array of handles	91
Example 4-27	Copying a simple class with new	92
Example 4-28	Copying a complex class with new	92
Example 4-29	Simple class with copy function	93
Example 4-30	Using copy function	94
Example 4-31	Complex class with deep copy function	94
Example 4-32	Basic Transactor	97
Example 5-1	Arbiter model using ports	101
Example 5-2	Testbench using ports	101
Example 5-3	Top-level netlist without an interface	102
Example 5-4	Simple interface for arbiter	103
Example 5-5	Top module using a simple arbiter interface	103

Example 5-6	Testbench using a simple arbiter interface	104
Example 5-7	Arbiter using a simple interface	104
Example 5-8	Connecting an interface to a module that uses ports	105
Example 5-9	Interface with modports	105
Example 5-10	Arbiter model with interface using modports	106
Example 5-11	Testbench with interface using modports	106
Example 5-12	Arbiter model with interface using modports	107
Example 5-13	Interface with a clocking block	109
Example 5-14	Race condition between testbench and design	111
Example 5-15	Testbench using interface with clocking block	113
Example 5-16	Signal synchronization	115
Example 5-17	Synchronous interface sample and module drive	115
Example 5-18	Testbench using interface with clocking block	116
Example 5-19	Interface signal drive	117
Example 5-20	Driving a synchronous interface	117
Example 5-21	Interface signal drive	118
Example 5-22	Bidirectional signals in a program and interface	119
Example 5-23	Bad clock generator in program block	120
Example 5-24	Good clock generator in module	121
Example 5-25	Top module using a simple arbiter interface	121
Example 5-26	Top-level scope for arbiter design	122
Example 5-27	Cross-module references with \$root	123
Example 5-28	Checking a signal with an if-statement	124
Example 5-29	Simple procedural assertion	124
Example 5-30	Error from failed procedural assertion	125
Example 5-31	Creating a custom error message in a procedural assertion	125
Example 5-32	Error from failed procedural assertion	125
Example 5-33	Creating a custom error message	126
Example 5-34	Concurrent assertion to check for X/Z	126
Example 5-35	ATM router model header without an interface	128
Example 5-36	Top-level netlist without an interface	129
Example 5-37	Testbench using ports	130
Example 5-38	Rx interface	132
Example 5-39	Tx interface	132
Example 5-40	ATM router model with interface using modports	133
Example 5-41	Top-level netlist with interface	133
Example 5-42	Testbench using interface with clocking block	134
Example 6-1	Simple random class	139
Example 6-2	Constraint without random variables	141

Example 6-3	Constrained-random class	142
Example 6-4	Constrain variables to be in a fixed order	142
Example 6-5	Random sets of values	143
Example 6-6	Inverted random set constraint	143
Example 6-7	Inverted random set constraint	143
Example 6-8	Choosing from an array of possible values	144
Example 6-9	Using randc to chose array values in random order	145
Example 6-10	Weighted random distribution with dist	146
Example 6-11	Dynamically changing distribution weights	146
Example 6-12	Bidirectional constraint	147
Example 6-13	Constraint block with implication operator	148
Example 6-14	Constraint block with if-else operator	148
Example 6-15	Expensive constraint with mod and unsized variable	149
Example 6-16	Efficient constraint with bit extract	149
Example 6-17	Class Unconstrained	149
Example 6-18	Class with implication	150
Example 6-19	Class with implication and constraint	151
Example 6-20	Class with implication and solve...before	152
Example 6-21	Using constraint_mode	154
Example 6-22	Checking write length with a valid constraint	155
Example 6-23	The randomize() with statement	156
Example 6-24	Building a bathtub distribution	157
Example 6-25	Constraint with a variable bound	159
Example 6-26	dist constraint with variable weights	159
Example 6-27	rand_mode disables randomization of variables	160
Example 6-28	Using the implication constraint as a case statement	161
Example 6-29	Turning constraints on and off with constraint_mode	162
Example 6-30	Class with an external constraint	163
Example 6-31	Program defining external constraint	163
Example 6-32	Signed variables cause randomization problems	164
Example 6-33	Randomizing unsigned 32-bit variables	164
Example 6-34	Randomizing unsigned 8-bit variables	165
Example 6-35	Constraining dynamic array size	165
Example 6-36	Random strobe pattern class	166
Example 6-37	Using random strobe pattern class	167
Example 6-38	First attempt at sum constraint: bad_sum1	167
Example 6-39	Program to try constraint with array sum	168
Example 6-40	Output from bad_sum1	168
Example 6-41	Second attempt at sum constraint: bad_sum2	168

Example 6-42	Output from bad_sum2	168
Example 6-43	Third attempt at sum constraint: bad_sum3	169
Example 6-44	Output from bad_sum3	169
Example 6-45	Fourth attempt at sum_constraint: bad_sum4	169
Example 6-46	Output from bad_sum4	169
Example 6-47	Simple foreach constraint: good_sum5	170
Example 6-48	Output from good_sum5	170
Example 6-49	Creating ascending array values with foreach	170
Example 6-50	UniqueArray class	171
Example 6-51	Unique value generator	172
Example 6-52	Using the UniqueArray class	172
Example 6-53	Command generator using randsequence	173
Example 6-54	Random control with randcase and \$random_range	175
Example 6-55	Equivalent constrained class	176
Example 6-56	Creating a decision tree with randcase	177
Example 6-57	Simple pseudorandom number generator	178
Example 6-58	Ethernet switch configuration class	180
Example 6-59	Building environment with random configuration	181
Example 6-60	Simple test using random configuration	182
Example 6-61	Simple test that overrides random configuration	182
Example 7-1	Interaction of begin...end and fork...join	185
Example 7-2	Output from begin...end and fork...join	185
Example 7-3	Fork...join_none code	186
Example 7-4	Fork...join_none output	186
Example 7-5	Fork...join_any code	187
Example 7-6	Output from fork...join_any	187
Example 7-7	Generator class with a run task	188
Example 7-8	Dynamic thread creation	189
Example 7-9	Bad fork...join_none inside a loop	190
Example 7-10	Execution of bad fork...join_none inside a loop	190
Example 7-11	Automatic variables in a fork...join_none	191
Example 7-12	Steps in executing automatic variable code	191
Example 7-13	Disabling a thread	192
Example 7-14	Limiting the scope of a disable fork	193
Example 7-15	Using disable label to stop threads	194
Example 7-16	Using wait fork to wait for child threads	194
Example 7-17	Blocking on an event in Verilog	195
Example 7-18	Output from blocking on an event	196
Example 7-19	Waiting for an event	196

Example 7-20	Output from waiting for an event	196
Example 7-21	Passing an event into a constructor	197
Example 7-22	Waiting for multiple threads with wait fork	198
Example 7-23	Waiting for multiple threads by counting triggers	198
Example 7-24	Waiting for multiple threads using a thread count	199
Example 7-25	Semaphores controlling access to hardware resource	200
Example 7-26	Exchanging objects using a mailbox: the Generator class	203
Example 7-27	Bounded mailbox	204
Example 7-28	Output from bounded mailbox	205
Example 7-29	Producer–consumer without synchronization, part 1	205
Example 7-30	Producer–consumer without synchronization, continued	206
Example 7-31	Producer–consumer without synchronization output	206
Example 7-32	Producer–consumer synchronized with an event	207
Example 7-33	Producer–consumer synchronized with an event, continued	208
Example 7-34	Output from producer–consumer with event	208
Example 7-35	Producer–consumer synchronized with a mailbox	209
Example 7-36	Output from producer–consumer with mailbox	210
Example 7-37	Basic Transactor	211
Example 7-38	Environment class	212
Example 7-39	Basic test program	213
Example 8-1	Base Transaction class	216
Example 8-2	Extended Transaction class	217
Example 8-3	Constructor with argument in an extended class	219
Example 8-4	Driver class	219
Example 8-5	Generator class	220
Example 8-6	Generator class using factory pattern	222
Example 8-7	Environment class	223
Example 8-8	Simple test program using environment defaults	224
Example 8-9	Injecting extended transaction from test	224
Example 8-10	Base and extended class	225
Example 8-11	Copying extended handle to base handle	226
Example 8-12	Copying a base handle to an extended handle	226
Example 8-13	Using \$cast to copy handles	226
Example 8-14	Transaction and BadTr classes	227
Example 8-15	Calling class methods	227
Example 8-16	Building an Ethernet frame with composition	230
Example 8-17	Building an Ethernet frame with inheritance	231
Example 8-18	Building a flat Ethernet frame	232
Example 8-19	Base transaction class with a virtual copy function	233

Example 8-20	Extended transaction class with virtual copy method	234
Example 8-21	Base transaction class with copy_data function	234
Example 8-22	Extended transaction class with copy_data function	235
Example 8-23	Base transaction class with copy_data function	235
Example 8-24	Base callback class	237
Example 8-25	Driver class with callbacks	237
Example 8-26	Test using a callback for error injection	238
Example 8-27	Test using callback for scoreboard	239
Example 9-1	Incomplete D-flip flop model missing a path	244
Example 9-2	Functional coverage of a simple object	249
Example 9-3	Coverage report for a simple object	250
Example 9-4	Coverage report for a simple object, 100% coverage	251
Example 9-5	Functional coverage inside a class	253
Example 9-6	Test using functional coverage callback	254
Example 9-7	Callback for functional coverage	255
Example 9-8	Cover group with a trigger	255
Example 9-9	Module with SystemVerilog Assertion	255
Example 9-10	Triggering a cover group with an SVA	256
Example 9-11	Using auto_bin_max set to 2	257
Example 9-12	Report with auto_bin_max set to 2	258
Example 9-13	Using auto_bin_max for all cover points	258
Example 9-14	Using an expression in a cover point	259
Example 9-15	Defining bins for transaction length	259
Example 9-16	Coverage report for transaction length	260
Example 9-17	Specifying bin names	261
Example 9-18	Report showing bin names	261
Example 9-19	Conditional coverage — disable during reset	262
Example 9-20	Using stop and start functions	262
Example 9-21	Functional coverage for an enumerated type	262
Example 9-22	Report with auto_bin_max set to 2	263
Example 9-23	Specifying transitions for a cover point	263
Example 9-24	Wildcard bins for a cover point	264
Example 9-25	Cover point with ignore_bins	264
Example 9-26	Cover point with auto_bin_max and ignore_bins	264
Example 9-27	Cover point with illegal_bins	265
Example 9-28	Basic cross coverage	266
Example 9-29	Coverage summary report for basic cross coverage	267
Example 9-30	Specifying cross coverage bin names	268
Example 9-31	Cross coverage report with labeled bins	268

Example 9-32	Excluding bins from cross coverage	269
Example 9-33	Specifying cross coverage weight	270
Example 9-34	Cross coverage with bin names	271
Example 9-35	Cross coverage with binsof	271
Example 9-36	Mimicking cross coverage with concatenation	272
Example 9-37	Specifying comments	272
Example 9-38	Specifying per-instance coverage	273
Example 9-39	Report all bins including empty ones	273
Example 9-40	Specifying the coverage goal	274
Example 9-41	Simple parameter	274
Example 9-42	Pass-by-reference	275
Example 9-43	Original class for transaction length	275
Example 9-44	solve...before constraint for transaction length	276
Example 10-1	Interface with clocking block	280
Example 10-2	Testbench using physical interfaces	281
Example 10-3	Testbench using virtual interfaces	282
Example 10-4	Testbench using virtual interfaces	283
Example 10-5	Interface for 8-bit counter	285
Example 10-6	Counter model using X_if interface	285
Example 10-7	Testbench using an array of virtual interfaces	286
Example 10-8	Counter testbench using virtual interfaces	287
Example 10-9	Driver class using virtual interfaces	288
Example 10-10	Testbench using a typedef for virtual interfaces	289
Example 10-11	Driver using a typedef for virtual interfaces	289
Example 10-12	Testbench using an array of virtual interfaces	290
Example 10-13	Testbench passing virtual interfaces with a port	290
Example 10-14	Interface with tasks for parallel protocol	292
Example 10-15	Interface with tasks for serial protocol	293

List of Figures

Figure 1-1	Directed test progress	6
Figure 1-2	Directed test coverage	6
Figure 1-3	Constrained-random test progress	8
Figure 1-4	Constrained-random test coverage	9
Figure 1-5	Coverage convergence	9
Figure 1-6	Test progress with and without feedback	14
Figure 1-7	The testbench — design environment	15
Figure 1-8	Testbench components	16
Figure 1-9	Signal and command layers	19
Figure 1-10	Testbench with functional layer	19
Figure 1-11	Testbench with scenario layer	20
Figure 1-12	Full testbench with all layers	21
Figure 1-13	Connections for the driver	22
Figure 2-1	Unpacked array storage	30
Figure 2-2	Packed array layout	33
Figure 2-3	Packed arrays	34
Figure 2-4	Associative array	37
Figure 4-1	Handles and objects	74
Figure 4-2	Static variables in a class	77
Figure 4-3	Contained objects	85
Figure 4-4	Handles and objects across routines	88
Figure 4-5	Objects and handles before copy with new	93
Figure 4-6	Objects and handles after copy with new	93
Figure 4-7	Objects and handles after deep copy	95
Figure 4-8	Layered testbench	96
Figure 5-1	The testbench – design environment	99

Figure 5-2	Testbench – Arbiter without interfaces	100
Figure 5-3	An interface straddles two modules	103
Figure 5-4	Main regions inside a SystemVerilog time step	112
Figure 5-5	A clocking block synchronizes the DUT and testbench	114
Figure 5-6	Sampling a synchronous interface	116
Figure 5-7	Driving a synchronous interface	118
Figure 5-8	Testbench – ATM router diagram without interfaces	127
Figure 5-9	Testbench - router diagram with interfaces	131
Figure 6-1	Building a bathtub distribution	157
Figure 6-2	Random strobe waveforms	166
Figure 6-3	Sharing a single random generator	178
Figure 6-4	First generator uses additional values	179
Figure 6-5	Separate random generators per object	179
Figure 7-1	Testbench environment blocks	183
Figure 7-2	Fork...join blocks	184
Figure 7-3	Fork...join block	185
Figure 7-4	Fork...join block diagram	193
Figure 7-5	A mailbox connecting two transactors	202
Figure 8-1	Simplified layered testbench	216
Figure 8-2	Base Transaction class diagram	217
Figure 8-3	Extended Transaction class diagram	218
Figure 8-4	Factory pattern generator	221
Figure 8-5	Factory generator with new pattern	222
Figure 8-6	Simplified extended transaction	225
Figure 8-7	Multiple inheritance problem	232
Figure 8-8	Callback flow	236
Figure 9-1	Coverage convergence	241
Figure 9-2	Coverage flow	242
Figure 9-3	Bug rate during a project	245
Figure 9-4	Coverage comparison	248
Figure 9-5	Uneven probability for transaction length	276
Figure 9-6	Even probability for transaction length with solve...before	276

List of Tables

Table 1.	Book icons	xxxi
Table 5-1.	Primary SystemVerilog scheduling regions	112
Table 6-1.	Solutions for bidirectional constraints	147
Table 6-2.	Solutions for Unconstrained class	150
Table 6-3.	Solutions for Imp1 class	151
Table 6-4.	Solutions for Imp2 class	152
Table 6-5.	Solutions for solve x before y constraint	153
Table 6-6.	Solutions for solve y before x constraint	153
Table 8-1.	Comparing inheritance to composition	229

Foreword

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

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

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

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

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

PHIL MOORBY
New England, 2006

Preface

What is this book about?

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

Who should read this book?

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

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

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

Why was SystemVerilog created?

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

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

Importance of a unified language

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

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

Importance of methodology

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

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

Comparing SystemVerilog and SystemC for high-level design

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

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

Overview of the book

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

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

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

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

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

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

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

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

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



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

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

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

Icons used in this book

Table 1. Book icons

	<p>Shows verification methodology to guide your usage of SystemVerilog testbench features</p>
	<p>Shows common coding mistakes</p>

Final comments

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

<http://chris.spear.net/systemverilog>

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

CHRIS SPEAR
Synopsys, Inc.

Acknowledgments

Few books are the creation of a single person. I want to thank all the people who spent countless hours helping me learn SystemVerilog and reviewing the book that you now hold in your hand. I especially would like to thank all the people at Synopsys for their help, including all my patient managers.

Janick Bergeron provided inspiration, innumerable verification techniques, and top-quality reviews. Without his guidance, this book would not exist. But the mistakes are all mine!

Alex Potapov and the VCS R&D team always showed patience with my questions and provided valuable insight on SystemVerilog features.

Mike Barnaby, Bob Beckwith, Quinn Canfield, James Chang, Cliff Cummings, Al Czamara, John Girard, Alex Lorgus, Mike Mintz, Brad Pierce, Arturo Salz, and **Kripa Sundar** reviewed some very rough drafts and inspired many improvements.

Hans van der Schoot gave me the confidence to write that one last chapter on functional coverage, and the detailed feedback to make it useful. **Benjamin Chin, Paul Graykowski, David Lee,** and **Chris Thompson** originated many of the ideas that evolved into the functional coverage chapter.

Dan McGinley and **Sam Starfas** patiently helped lift me from the depths of Word up to the heights of FrameMaker.

Ann K. Farmer — Arrigato gozaimasu! You brought sense to my scribblings.

Will Sherwood inspired me to become a verification engineer, and taught me new ways to break things.

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

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

All trademarks and copyrights are the property of their respective owners.

Chapter 1

Verification Guidelines

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

1.1 Introduction

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

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

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

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