Verilog and SystemVerilog Gotchas

101 Common Coding Errors and How to Avoid Them

Stuart Sutherland
Don Mills

Verilog and SystemVerilog Gotchas

101 Common Coding Errors and How to Avoid Them

Springer

Stuart Sutherland
Sutherland HDL, Inc.
Tualatin, OR
USA

Don Mills
LCDM Engineering
Chandler, AZ
USA

Library of Congress Control Number: 2007926706

ISBN 978-0-387-71714-2 e-ISBN 978-0-387-71715-9

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 know 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

Dedication

To my wonderful wife, LeeAnn, and my children, Ammon, Tamara, Hannah, Seth and Samuel — thank you for your patience during the many long hours and late nights you tolerated while this book was being written.

Stu Sutherland
Portland, Oregon

To my wife and sweetheart Geri Jean, and my children, Sara, Kirsten, Adam, Alex, Dillan, Donnelle, Grant and Gina — thanks to each of you for the patience you have had with me as I have dealt with debugging many of these gotchas on designs over the years.

Don Mills
Chandler, Arizona

About the Authors

Mr. Stuart Sutherland is a member of the IEEE 1800 working group that oversees both the Verilog and SystemVerilog standards. He has been involved with the definition of the Verilog standard since its inception in 1993, and the SystemVerilog standard since work began in 2001. In addition, Stuart is the technical editor of the official IEEE Verilog and SystemVerilog Language Reference Manuals (LRMs). Stuart is an independent Verilog consultant, specializing in providing comprehensive expert training on the Verilog HDL, SystemVerilog and PLI. Stuart is a co-author of the books "*SystemVerilog for Design*", "*Verilog-2001: A Guide to the New Features in the Verilog Hardware Description Language*" and is the author of "*The Verilog PLI Handbook*", as well as the popular "*Verilog HDL Quick Reference Guide*" and "*Verilog PLI Quick Reference Guide*". He has also authored a number of technical papers on Verilog and SystemVerilog, which are available at www.sutherland-hdl.com/papers. You can contact Stuart at stuart@sutherland-hdl.com.

visit the author's web page at www.sutherland-hdl.com

Mr. Don Mills has been involved in ASIC design since 1986. During that time, he has worked on more than 30 ASIC projects. Don started using top-down design methodology in 1991 (Synopsys Design Compiler 1.2). Don has developed and implemented top-down ASIC design flows at several companies. His specialty is integrating tools and automating the flow. Don works for Microchip Technology Inc. as an internal SystemVerilog and Verilog consultant. Don is a member of the IEEE Verilog and System Verilog committees that are working on language issues and enhancements. Don has authored and co-authored numerous papers, such as "SystemVerilog Assertions are for Design Engineers Too!" and "RTL Coding Styles that Yield Simulation and Synthesis Mismatches". Copies of these papers can be found at www.lcdm-eng.com. Mr. Mills can be reached at mills@lcdm-eng.com or don.mills@microchip.com.

visit the author's web page at www.lcdm-eng.com

Acknowledgments

The authors express their sincere appreciation to the contributions of several Verilog and SystemVerilog experts.

Chris Spear of Synopsys, Inc. suggested several of the verification related gotchas, provided the general descriptions of these gotchas, and ran countless tests for us.

Shalom Bresticker of Intel also suggested several gotchas.

Jonathan Bromley of Doulos, Ltd., ***Clifford Cummings*** of Sunburst Design, ***Tom Fitzpatrick*** of Mentor Graphics, ***Steve Golson*** of Trilobyte Systems, ***Gregg Lahti*** of Microchip Technology, Inc. and ***Chris Spear*** of Synopsys, Inc. provided thorough technical reviews of this book, and offered invaluable comments on how to improve the gotcha descriptions.

Steve Golson of Trilobyte Systems provided a wonderful foreword to this book

Lastly, we acknowledge and express our gratitude to our wives, ***LeeAnn Sutherland*** and ***Geri Jean Mills***, for meticulously reviewing this book for grammar and punctuation. If any such errata remain in the book, it could only be due to changes we made after their reviews.

Table of Contents

List of Gotchas xv

Foreword
by Steve Golson 1

Chapter 1:
Introduction,
What Is A Gotcha? 3

Chapter 2:
Declaration and Literal Number Gotchas 7

Gotcha 1: Case sensitivity 7
Gotcha 2: Implicit net declarations 10
Gotcha 3: Default of 1-bit internal nets 13
Gotcha 4: Single file versus multi-file compilation of $unit declarations 15
Gotcha 5: Local variable declarations 17
Gotcha 6: Escaped names in hierarchical paths 19
Gotcha 7: Hierarchical references to automatic variables 22
Gotcha 8: Hierarchical references to variables in unnamed blocks 25
Gotcha 9: Hierarchical references to imported package items 27
Gotcha 10: Importing enumerated types from packages 28
Gotcha 11: Importing from multiple packages 29
Gotcha 12: Default base of literal integers 30
Gotcha 13: Signedness of literal integers 32
Gotcha 14: Signed literal integers zero extend to their specified size 33
Gotcha 15: Literal integer size mismatch in assignments 35
Gotcha 16: Filling vectors with all ones 37
Gotcha 17: Array literals versus concatenations 38
Gotcha 18: Port connection rules 39
Gotcha 19: Back-driven ports 43

Gotcha 20: Passing real (floating point) numbers through ports 46

Chapter 3:
RTL Modeling Gotchas 49

Gotcha 21: Combinational logic sensitivity lists with function calls 49
Gotcha 22: Arrays in sensitivity lists 52
Gotcha 23: Vectors in sequential logic sensitivity lists 54
Gotcha 24: Operations in sensitivity lists 56
Gotcha 25: Sequential logic blocks with begin...end groups 57
Gotcha 26: Sequential logic blocks with resets 59
Gotcha 27: Asynchronous set/reset flip-flop for simulation and synthesis 60
Gotcha 28: Blocking assignments in sequential procedural blocks 62
Gotcha 29: Sequential logic that requires blocking assignments 64
Gotcha 30: Nonblocking assignments in combinational logic 66
Gotcha 31: Combinational logic assignments in the wrong order 70
Gotcha 32: Casez/casex masks in case expressions 72
Gotcha 33: Incomplete decision statements 74
Gotcha 34: Overlapped decision statements 77
Gotcha 35: Inappropriate use of unique case statements 79
Gotcha 36: Resetting 2-state models 82
Gotcha 37: Locked state machines modeled with enumerated types 84
Gotcha 38: Hidden design problems with 4-state logic 86
Gotcha 39: Hidden design problems using 2-state types 88
Gotcha 40: Hidden problems with out-of-bounds array access 90
Gotcha 41: Out-of-bounds assignments to enumerated types 92
Gotcha 42: Undetected shared variables in modules 94
Gotcha 43: Undetected shared variables in interfaces and packages 96

Chapter 4:
Operator Gotchas 99

Gotcha 44: Assignments in expressions 99
Gotcha 45: Self-determined versus context-determined operators 101
Gotcha 46: Operation size and sign extension in assignment statements 105
Gotcha 47: Signed arithmetic rules 108

Gotcha 48: Bit-select and part-select operations 111
Gotcha 49: Increment, decrement and assignment operators 112
Gotcha 50: Pre-increment versus post-increment operations 113
Gotcha 51: Modifying a variable multiple times in one statement 115
Gotcha 52: Operator evaluation short circuiting 116
Gotcha 53: The not operator (!) versus the invert operator (~) 118
Gotcha 54: Array method operations 119
Gotcha 55: Array method operations on an array subset 121

Chapter 5:
General Programming Gotchas 123
Gotcha 56: Verifying asynchronous and synchronous reset at time zero 123
Gotcha 57: Nested if...else blocks 128
Gotcha 58: Evaluation of equality with 4-state values 129
Gotcha 59: Event trigger race conditions 131
Gotcha 60: Using semaphores for synchronization 134
Gotcha 61: Using mailboxes for synchronization 137
Gotcha 62: Triggering on clocking blocks 139
Gotcha 63: Misplaced semicolons after decision statements 140
Gotcha 64: Misplaced semicolons in for loops 142
Gotcha 65: Infinite for loops 144
Gotcha 66: Locked simulation due to concurrent for loops 145
Gotcha 67: Referencing for loop control variables 147
Gotcha 68: Default function return size 148
Gotcha 69: Task/function arguments with default values 150
Gotcha 70: Continuous assignments with delays cancel glitches 151

Chapter 6:
Object Oriented and Multi-Threaded Programming Gotchas 153
Gotcha 71: Programming statements in a class 153
Gotcha 72: Using interfaces with object-oriented testbenches 155
Gotcha 73: All objects in mailbox come out with the same values 157
Gotcha 74: Passing handles to methods using input versus ref arguments 158
Gotcha 75: Constructing an array of objects 159

Gotcha 76: Static tasks and functions are not re-entrant 160
Gotcha 77: Static versus automatic variable initialization 162
Gotcha 78: Forked programming threads need automatic variables 164
Gotcha 79: Disable fork kills too many threads 166
Gotcha 80: Disabling a statement block stops more than intended 168
Gotcha 81: Simulation exits prematurely, before tests complete 171

Chapter 7:
Randomization, Coverage and Assertion Gotchas 173
Gotcha 82: Variables declared with rand are not getting randomized 173
Gotcha 83: Undetected randomization failures 175
Gotcha 84: $assertoff could disable randomization 177
Gotcha 85: Boolean constraints on more than two random variables 179
Gotcha 86: Unwanted negative values in random values 181
Gotcha 87: Coverage reports default to groups, not bins 182
Gotcha 88: Coverage is always reported as 0% 184
Gotcha 89: The coverage report lumps all instances together 186
Gotcha 90: Covergroup argument directions are sticky 187
Gotcha 91: Assertion pass statements execute with a vacuous success 188
Gotcha 92: Concurrent assertions in procedural blocks 190
Gotcha 93: Mismatch in assert...else statements 192
Gotcha 94: Assertions that cannot fail 193

Chapter 8:
Tool Compatibility Gotchas 195
Gotcha 95: Default simulation time units and precision 195
Gotcha 96: Package chaining 198
Gotcha 97: Random number generator is not consistent across tools 200
Gotcha 98: Loading memories modeled with always_latch/always_ff 202
Gotcha 99: Non-standard language extensions 204
Gotcha 100:Array literals versus concatenations 206
Gotcha 101:Module ports that pass floating point values (real types) 208

Index **209**

List of Gotchas

Gotcha 1: *7*
The names in my code look correct and worked in my VHDL models, but Verilog/SystemVerilog gets errors about "undeclared identifiers".

Gotcha 2: *10*
A typo in my design connections was not caught by the compiler, and only showed up as a functional problem in simulation.

Gotcha 3: *13*
In my netlist, only bit zero of my vector ports get connected.

Gotcha 4: *15*
My models compile OK, and the models from another group compile OK; but when compiled together, I get errors about multiple declarations.

Gotcha 5: *17*
I get compilation errors on my local variable declarations, but the declaration syntax is correct.

Gotcha 6: *19*
I get weird compiler errors when I try to reference a design signal with an escaped name from my testbench.

Gotcha 7: *22*
I get compilation errors when my testbench tries to print out some signals in my design, but other signals can be printed without a problem.

Gotcha 8: *25*
With Verilog, my testbench could print out local variables in a begin...end block, but with SystemVerilog I get compilation errors.

Gotcha 9: *27*
My design can use imported package items just fine, but my testbench cannot access the items for verification.

Gotcha 10: *28*
I imported an enumerated type from a package, but I cannot access the labels defined by the enumerated type.

Gotcha 11: *29*
I get errors when I try to wildcard import multiple packages, but I can wildcard import each package separately without any errors.

List of Gotchas

Gotcha 12: *30*
Some branches of my case statement are never selected, even with the correct input values.

Gotcha 13: *32*
My incrementor model sometimes gets incorrect values when I increment using a literal 1'b1.

Gotcha 14: *33*
When I specify a signed, sized literal integer with a negative value, it does not sign extend.

Gotcha 15: *35*
When I assign a 4-bit negative value to an 8-bit signed variable, it is not sign extended.

Gotcha 16: *37*
I can use a literal integer to set all bits to Z on a vector of any size, but when I use the same syntax to set all bits to 1, I get a decimal 1 instead.

Gotcha 17: *38*
The wrong values are stored when I assign a list of values to a packed array or structure.

Gotcha 18: *39*
My design doesn't work correctly when I connect all the modules together, but each module works correctly by itself.

Gotcha 19: *43*
I declared my port as an input, and software tools let me accidentally use the port as an output, without any errors or warnings.

Gotcha 20: *46*
I cannot find a way to pass real values from one module to another using either Verilog or SystemVerilog.

Gotcha 21: *49*
My combinational logic seemed to simulate OK, but after synthesis, the gate-level simulation does not match the RTL simulation.

Gotcha 22: *52*
I need my combinational logic block to be sensitive to all elements of a RAM array, but the sensitivity list won't trigger at the correct times.

Gotcha 23: *54*
My always block is supposed to trigger on any positive edge in a vector, but it misses most edges.

List of Gotchas

Gotcha 24: *56*
My sensitivity list should trigger on any edge of a or b, but it misses some changes.

Gotcha 25: *57*
The clocked logic in my sequential block gets updated, even when no clock occurred.

Gotcha 26: *59*
Some of the outputs of my sequential logic do not get reset.

Gotcha 27: *60*
When I code an asynchronous set/reset D-type flip-flop following synthesis coding rules, my simulation results are sometimes wrong.

Gotcha 28: *62*
My shift register sometimes does a double shift in one clock cycle.

Gotcha 29: *64*
I'm following the recommendations for using nonblocking assignments in sequential logic, but I still have race conditions in simulation.

Gotcha 30: *66*
My RTL simulation locks up and time stops advancing.

Gotcha 31: *70*
Simulation of my gate-level combinational logic does not match RTL simulation.

Gotcha 32: *72*
My casex statement is taking the wrong branch when there is an error in the case expression.

Gotcha 33: *74*
My full_case, parallel_case decision statement simulated as I expected, but the chip does not work.

Gotcha 34: *77*
One of my decision branches never gets executed.

Gotcha 35: *79*
I am using unique case everywhere to help trap design bugs but my synthesis results are not what I expected.

Gotcha 36: *82*
My design fails to reset the first time in RTL simulation.

Gotcha 37: *84*
My state machine model locks up in its start-up state.

List of Gotchas

Gotcha 38: *86*
There was a problem deep inside the logic of my design, but it never propagated to module boundaries.

Gotcha 39: *88*
Some major functional bugs in my design did not show up until after synthesis, when I ran gate-level simulations.

Gotcha 40: *90*
A design bug caused references to nonexistent memory addresses, but there was no indication of a problem in RTL simulation.

Gotcha 41: *92*
My enumerated state machine variables have values that don't exist in the enumerated definition.

Gotcha 42: *94*
My RTL model output changes values when it shouldn't, and to unexpected values.

Gotcha 43: *96*
Variables in my package keep changing at unexpected times and to unexpected values.

Gotcha 44: *99*
I need to do an assignment as part of an if condition, but cannot get my code to compile.

Gotcha 45: *101*
In some operations, my data is sign extended and in other operations it is not sign extended, and in yet other operations it is not extended at all.

Gotcha 46: *105*
I declared my outputs as signed types, but my design is still doing unsigned operations.

Gotcha 47: *108*
My signed adder model worked perfectly until I added a carry-in input, and now it only does unsigned addition.

Gotcha 48: *111*
All my data types are declared as signed, and I am referencing the entire signed vectors in my operations, yet I still get unsigned results.

Gotcha 49: *112*
I'm using the ++ operator for my counter; the counter value is correct, but other code that reads the counter sees the wrong value.

List of Gotchas

Gotcha 50: *113*
My while loop is supposed to execute 16 times, but it exits after 15 times, even though the loop control variable has a value of 16.

Gotcha 51: *115*
When I have multiple operations on a variable in a single statement, I get different results from different simulators.

Gotcha 52: *116*
I am calling a function twice in a statement, but sometimes only one of the calls is executed.

Gotcha 53: *118*
My if statement with a not-true condition did not execute when I was expecting it to.

Gotcha 54: *119*
I get the wrong result when I sum all the values of an array using the built-in .sum method.

Gotcha 55: *121*
I get the wrong answer when I sum specific array elements in an array.

Gotcha 56: *123*
Sometimes my design resets correctly at time zero, and sometimes it fails to reset.

Gotcha 57: *128*
My else branch is pairing up with the wrong if statement.

Gotcha 58: *129*
My testbench completely misses problems on design outputs, even though it is testing the outputs.

Gotcha 59: *131*
I'm using the event data type to synchronize processes, but sometimes when I trigger an event, the sensing process does not activate.

Gotcha 60: *134*
My processes are not synchronizing the way I expected using semaphores. Even when there are waiting processes, some other process gets to run ahead of them.

Gotcha 61: *137*
My mailbox works at first, and then starts getting errors during simulation.

Gotcha 62: *139*
I cannot get my test program to wait for a clocking block edge.

Gotcha 63: *140*
Statements in my if() decision execute, even when the condition is not true.

Gotcha 64: *142*
My for loop only executes one time.

Gotcha 65: *144*
My for loop never exits. When the loop variable reaches the exit value, the loop just starts over again.

Gotcha 66: *145*
When I run simulation, my for loops lock up or do strange things.

Gotcha 67: *147*
My Verilog code no longer compiles after I convert my Verilog-style for loops to a SystemVerilog style.

Gotcha 68: *148*
My function only returns the least significant bit of the return value.

Gotcha 69: *150*
I get a syntax error when I try to assign my task/function input arguments a default value.

Gotcha 70: *151*
Some delayed outputs show up with continuous assignments and others do not.

Gotcha 71: *153*
Some programming code in an initial procedure compiles OK, but when I move the code to a class definition, I get compilation errors.

Gotcha 72: *155*
I get a compilation error when I try to use a class object to create test values when the testbench connects to the design using an interface.

Gotcha 73: *157*
My code creates random object values and puts them into a mailbox, but all the objects coming out of the mailbox have the same value.

Gotcha 74: *158*
My method constructs and initializes an object, but I can never see the object's value.

Gotcha 75: *159*
I declared an array of objects, but get a syntax error when I try to construct the array.

Gotcha 76: *160*
My task works OK sometimes, but gets bogus results other times.

Foreword by Steve Golson

Some people collect baseball cards, old car magazines, or maybe rubber duckies.

I collect Verilog books.

It started back in 1989 with a looseleaf copy of "Gateway VERILOG-XL Reference Manual Version 1.5a" in a three-ring binder. Verilog was a bit simpler back then—it's hard to believe we actually designed chips using only one type of procedural assignment (nonblocking assigns were not part of the language yet). And we ran our simulations on a VAX, or maybe a fancy Apollo workstation.

Since then I've bought pretty much every Verilog book that came along. I've got a few synthesis books, and I'll pick up an occasional VHDL reference or maybe a text on the history of hardware description languages, but mostly it's Verilog. Dozens and dozens of books about Verilog.

There's a funny thing about most of these books though. After I leaf through them a few times, they sit on the shelf. I admit that it looks pretty impressive once you have an entire bookcase filled with Verilog books, but the discerning visitor will notice how fresh and new they all are. Unused. Unread. Useless.

I'm often disappointed to find very little information which is useful for the practicing engineer. What I'm looking for is a book I can use every day, a book that will help me get my chip out the door, on time and working.

Stu and Don have written such a book. I've known these guys for many years, and they have probably forgotten more Verilog than I've ever known. They have distilled their collective knowledge into this helpful and extremely useful book. Read it and you won't be disappointed.

If you are an old hand at Verilog try to pick out all the Gotchas that you have found the hard way. Smile and say to yourself "Oh yeah, I remember getting caught by that one!"

Those of you who are new to Verilog and SystemVerilog, welcome aboard! Here's your chance to learn from two of the leading experts in the field. And if you ever have a chance to take a training class from either of these gentlemen, don't hesitate to sign up. I guarantee you won't regret it.

Oh by the way, my favorite Gotcha is "Gotcha 65: Infinite for loops". Why? Well, I built a chip with that bug in it. Believe me, when a modeling error causes you to have broken silicon, you never forget why it happened. Back then I didn't have this book to help me, but you do! Keep this book close at hand, refer to it often, and may all your models compile and all your loops terminate.

Steve Golson
Trilobyte Systems
http://www.trilobyte.com

Chapter 1

Introduction, What Is A Gotcha?

This chapter defines what a "*gotcha*" is, and why programming languages allow gotchas. For the curious, the chapter also provides a brief history of the Verilog and SystemVerilog standards. The topics presented in this chapter include:

- What are Verilog and SystemVerilog
- The definition of a gotcha
- A brief description of the Verilog and SystemVerilog standards

What are Verilog and SystemVerilog?

The terms "Verilog" and "SystemVerilog" are sometimes a source of confusion because the terms are not used consistently in the industry. For the purposes of this book, "Verilog" and SystemVerilog are used as follows:

Verilog is a Hardware Description Language (HDL). It is a specialized programming language used to model digital hardware designs and, to a limited extent, to write test programs to exercise these models.

SystemVerilog is a substantial set of extensions to the Verilog HDL. A primary goal of these extensions is to enable modeling and verifying larger designs with more compact code. By itself, SystemVerilog is not a complete language; it is just a set of additions to the base Verilog language.

What is a Gotcha?

A programming "gotcha" is a language feature, which, if misused, causes unexpected—and, in hardware design, potentially disastrous—behavior. The classic example in the C language is having an assignment within a conditional expression, such as:

```
if (day=15)        /* GOTCHA! assigns value of 15 to day, then */
  do_mid_month_payroll;  /* if day is non-zero, do a payroll */
```

Most likely, what the programmer intended to code is `if (a==b)` instead of `if (a=b)`. The results are very different! This classic C programming Gotcha is not a syntax error; the code is perfectly legal. However, the code probably does not produce the intended results. If the coding error is not detected before a product is shipped, a simple bug like this could lead to serious ramifications in a product.

Just like any programming language, Verilog, and the SystemVerilog extensions to Verilog, have gotchas. There are constructs in Verilog and SystemVerilog that can be used in ways that are syntactically correct, but yield unexpected or undesirable results. Some of the primary reasons Verilog and SystemVerilog have gotchas are:

- Inheritance of C and C++ gotchas

  Verilog and SystemVerilog leverage the general syntax and semantics of the C and C++ languages. Verilog and SystemVerilog inherit the strengths of these powerful programming languages, but they also inherit many of the gotchas of C and C++. (Which raises the question, can the common C coding error such as `if (day=15)` be made in Verilog/SystemVerilog? The answer can be found in Gotcha 44 on page 99.)

- Loosely typed operations

  Verilog and SystemVerilog are *loosely typed* languages. As such, operations can be performed on any data type, and underlying language rules take care of how operations should be performed. If a design or verification engineer does not understand these underlying language rules, then unexpected results can occur.

- Allowance to model good and bad designs

  An underlying philosophy of Verilog and SystemVerilog is that engineers should be allowed to model and prove both what works correctly in hardware, and what will not work in hardware. In order to legally model hardware that does not work, the language must also permit unintentional modeling errors when the intent is to model designs that work correctly.

The Verilog and SystemVerilog standards

Verilog is an international standard Hardware Description Language. The official standard is **IEEE Std 1364-2005 Verilog Language Reference Manual** (LRM), commonly referred to as ***"Verilog-2005"***. The Verilog standard defines a rich set of programming and modeling constructs specific to representing the behavior of digital logic. The Verilog Hardware Description Language was first created in 1984. Verilog was designed to meet the needs of engineering in the mid 1980s, when a typical design was under 50,000 gates and ICs were based on 3 micron technology. As digital design size and technologies changed, Verilog evolved to meet new design requirements. Verilog was first standardized by the IEEE in 1995 (IEEE Std 1364-1995). In 2001, The IEEE released the Verilog-2001 standard (IEEE Std 1364-2001) which enhanced Verilog in several ways, such as synthesizable signed arithmetic on any vector size and re-entrant tasks and functions. The IEEE updated the Verilog standard in 2005, but no major modeling enhancements were added in this version. Instead, all enhancements to Verilog were documented in a separate standard, SystemVerilog.

SystemVerilog is a standard set of extensions to the Verilog-2005 Standard. These extensions are documented in a separate standard, **IEEE Std 1800-2005 SystemVerilog Language Reference Manual**, commonly referred to as ***"SystemVerilog-2005"***. The SystemVerilog extensions enable writing synthesizable models that are continuously increasing in size and complexity, as well as verifying these multi-million gate designs. SystemVerilog adds to Verilog features from the SUPERLOG, VERA C, C++, and VHDL languages, along with OVA and PSL assertions. SystemVerilog was first developed by Accellera, a consortium of companies that do electronic design and companies that provide Electronic Design Automation (EDA) tools. Accellera released a preliminary version of the extensions to Verilog in 2002, called ***SystemVerilog 3.0*** (3.0 to show that SystemVerilog was the next generation of Verilog, where Verilog-1995 was the first generation and Verilog 2001 was the second generation). In 2003, Accellera released ***SystemVerilog 3.1*** and in 2004 ***SystemVerilog 3.1a***. This latter Accellera standard was then submitted to the IEEE for full standardization.

The original intent was for the IEEE to fold the Accellera SystemVerilog extensions into the Verilog standard. At the insistence of EDA companies, however, the IEEE made the decision to temporarily keep the SystemVerilog extensions in a separate document to make it easier for EDA companies to implement the extensive set of new features in their Verilog tools.