SystemVerilog Assertions Handbook

… for Formal and Dynamic Verification

Published by:
VhdlCohen Publishing
P.O. 2362
Palos Verdes Peninsula CA 90274-2362
vhdlcohen@aol.com
http://www.vhdlcohen.com

Library of Congress Cataloging-in-Publication Data
A C.I.P. Catalog record for this book is available from the Library of Congress

SystemVerilog Assertions Handbook
… for Formal and Dynamic Verification
ISBN 0-9705394-7-9

Copyright © 2005 by VhdlCohen Publishing

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without the prior written permission from the author, except for the inclusion of brief quotations in a review.

Printed on acid-free paper

Printed in the United States of America

Contents

Foreword xi
Surrendra A. Dudani xi
Stuart Sutherland xiii
Harry D. Foster xv
Tarak Parikh xvii
Keith Rieken xix
Yu-Chin Hsu xxi
Alain Raynaud xxiii

Preface xxv

Acknowledgements xxix

About the authors xxxiii

Disclaimer xxxv

1 ROLE OF SYSTEMVERILOG ASSERTIONS IN A VERIFICATION METHODOLOGY 1
1.1 History of Design Verification methodologies 2
1.2 SystemVerilog Assertions in verification Strategy 5
1.2.1 Are Assertions Independent from SystemVerilog Structures? 5
1.2.2 Are Assertions Useful for the Definition and Verification of Designs? 6
1.2.2.1 Captures Designer Intent 7
1.2.2.2 Allows Protocols to be Defined and Verified 8
1.2.2.3 Reduces the Time to Market 8
1.2.2.4 Greatly Simplifies the Verification of Reusable IP 8
1.2.2.5 Facilitates Functional Coverage Metrics 9
1.2.2.6 Generates Counterexamples to Demonstrate Violation of Properties 10
1.2.3 Can/should entire functional verification task be performed using SystemVerilog Assertions? 10
1.2.4 Is SystemVerilog Assertions Solely Restricted to Applications that Use SystemVerilog? 10
1.2.4.1 VHDL Model and Testbench with SystemVerilog Assertions Module 10
1.2.4.2 VHDL Model Embedded in SystemVerilog testbench with SVA Module 11
1.3 Accellera's SystemVerilog Assertions Goals 11
1.4 SystemVerilog Assertions Language 12

2 OVERVIEW OF PROPERTIES AND ASSERTIONS 15
2.1 DEFINITIONS 15
2.1.1 Properties 15
2.1.2 Sequences 16
2.1.3 Antecedent / Consequent / Thread 16
2.1.4 Specification and Verification 18
Assertion-Based Verification 18
2.1.5 Assertion / Assumption / Verification Directive 19
2.1.6 Constraint 19
2.2 property 20
2.2.1 Named Properties 20
2.3 Assertion 21
2.3.1 Immediate assertions 22
2.3.2 Concurrent Assertion 24
Verification Directives 24
2.4 Boolean Expression 26

3 UNDERSTANDING PROPERTIES 27
3.1 Sequences Overview 28
3.1.1 Sequence Declaration 29
3.2 SystemVerilog Properties 31
3.2.1 Property Header 32
3.2.2 Property Identifier 33
3.2.3 Formal Arguments and Usage 33
3.2.4 Local Variables in Properties 34
3.2.5 Body of the Property 39
3.2.5.1 Clocking Event 39
3.2.5.2 Disabling condition 40
3.2.5.3 Property Expression 42
3.2.5.3.1 Property Operators 42

4 UNDERSTANDING SEQUENCES 47
4.1 Sequence Operators and Built-in Functions 48
4.2 Capturing Temporal Behavior in SystemVerilog Assertions 49
4.3 Implication Operators 53
4.3.1 Overlapped implication Operator |-> 54
4.3.2 Non Overlapped Implication Operator |=> 55
4.3.3 Understanding the Overlapped Implication Operator "|->" 56
4.3.4 Understanding the Non-overlapped Implication Operator " |=>" 58
4.3.5 Using "not" with Implication Operator 59
4.4 first_match Operator 60
4.5 Repetition Operators 64
4.5.1 Consecutive Repetition 66
4.5.1.1 [*n] Repetition 66
4.5.1.2 [*n:m] Repetition 66
4.5.1.3 [*0 : m] Repetitions 69
4.5.1.4 [*n : $], ##[0:$] 71
4.5.2 Sequence Non-consecutive Repetition ([=n]) 74

4.5.3 Sequence goto Repetition ([->n]) 74
4.6 Sequence Composition Operators 75
4.6.1 Sequence Fusion (##0) 76
4.6.2 Sequence Disjunction (or) 76
4.6.3 Sequence Non-Length-Matching (and) 77
4.6.4 Sequence Length-Matching (intersect) 77
4.6.5 Sequence Containment (within) 78
4.6.6 Conditions over Sequences (throughout operator) 79
4.7 Methods Supporting Sequences 80
4.7.1 Endpoint of a Single Clock Sequence "ended" 80
4.7.2 Endpoint of a Multi-Clock Sequence "matched" 83
4.7.3 Triggered Method 84
4.7.4 Sequence events 85
Exercises 86

5 Advanced Topics For Properties and Sequences 91
5.1 Data types in Properties and Sequences 91
5.2 Misuse of Assertion Overlapping 93
5.3 Multiple Threads Termination 98
5.4 Assertion Refinement Process 99
5.4.1 Relaxed, stringent assertion 100
5.5 Unbounded Range $ in Properties 100
5.6 Recursion 101
5.7 Emulating PSL-Like Constructs in SVA 104
5.7.1 whilenot 104
5.7.2 The eventually! Operator in Sequence 106
5.7.3 Emulating UNTIL with Sequences 106
5.7.4 F before G 107
5.7.5 One-Shot Assertion Using Initial blocks 108
5.7.5.1 Flag Bit Defining Start of Antecedent 108
5.7.5.2 Procedural Assertion in Initial Block 109
5.8 Assertion-Based System Functions 109
5.8.1 Sampled Valued Functions 109
5.8.1.1 Value access functions 109
5.8.1.1.1 $sampled(expression [, clocking_event]) 110
5.8.1.1.2 $past 111
5.8.1.2 Value change functions 113
5.8.1.2.1 $rose and $fell 113
5.8.1.2.2 $stable 115
5.8.2 Vector-Analysis System Functions 116
5.8.3 Severity-level System Functions 116
5.8.4 Assertion-Control System Tasks 117
5.9 Clocked Sequence and Multi-Clocking 118
5.9.1 Clock Specification for Properties and Sequences 118
5.9.2 Clock Resolution 120
5.9.2.1 Clock Resolution in Assertion and Property Directives 121
5.9.2.2 Clock Resolution in Sequences 121
5.9.3 Multiple clocked sequences 123

5.9.3.1 Rules in Using Multiple-Clocked Sequence 125
5.9.4 Multiple-clocked properties 127
5.9.5 Clock flow 128
5.9.6 Clocking Rules in Assertions 129
5.9.6.1 Single clocked assertions: 130
5.9.6.2 Sequence and Properties in Clocking Blocks 130
5.9.6.3 Multiple-clocked Assertions 131
5.10 SystemVerilog Scheduling semantics for Assertions 131
5.11 Properties in Interfaces 134
5.12 Verification Directives 135
5.12.1 assert Directive 135
5.12.1.1 Concurrent Assertion Statements Outside of Procedural Code 135
5.12.1.2 Concurrent Assertion Statements Embedded in Procedural Block 136
5.12.1.3 Immediate assertion: 137
5.12.1.4 Action-block 137
5.12.2 assume Directive 138
5.12.3 cover Directive 140
5.12.4 Expect Construct 142
5.13 Binding Properties to Scopes or Instances 143
5.14 Verifying VHDL Models with SystemVerilog Assertions 148
5.14.1 The Concept 148
5.14.2 VHDL Module in VHDL Testbench with SystemVerilog Assertions Module 148
5.14.2.1 VHDL Model 149
5.14.2.2 SystemVerilog Assertions Module 149
5.14.2.3 Connecting SystemVerilog Assertions module to VHDL design 150
5.14.2.3.1 Direct Instantiation of SVA module into VHDL Testbench 150
5.14.2.3.2 Binding of SVA Verification Module to VHDL Model 151
5.14.3 VHDL Model in a SystemVerilog Testbench with SVA Module 152

6 SystemVerilog Assertions In the Design Process 153
6.1 Traditional Design Process 154
6.2 Design Process with ABV using SVA as vehicle 154
6.2.1 System-level Assertions 155
6.2.2 Interface Assertions 161
6.2.3 Architectural Plan 161
6.2.4 Verification Plan 162
6.2.5 RTL Design 163
6.2.6 Write Testbench and Simulate 164
6.2.7 Analyze the Simulation Results and Coverage 164
6.2.8 Formal Verification (FV) 169
6.3 Case Study - Synchronous FIFO 170
6.3.1 Synchronous FIFO Requirements 170
6.3.2 Verification Plan 182
6.3.3 RTL Design 191
6.3.4 Simulation 193
6.3.5 Formal Verification 193
Exercises 195

7 FORMAL VERIFICATION USING ASSERTIONS 199
7.1 FV METHODOLOGY 200
7.1.1 Model Checking Expectations and Rules 203
7.2 Role of SystemVerilog Assertions in FV 204
7.2.1 SystemVerilog Assertions in Formal Specifications 204
7.2.2 SystemVerilog Assertions Usage in FV vs. Dynamic ABV 205
7.2.2.1 Same Inputs in Antecedent and Consequent 205
7.3 CASE STUDY - FV OF A TRAFFIC LIGHT CONTROLLER 206
7.3.1 Model 206
7.3.2 Basic requirements 209
7.3.3 SystemVerilog Assertions for Traffic Light Controller 209
7.3.4 Verification 213
7.3.5 Good Traffic Light Controller 215
7.4 FV COVERAGE METRICS 216
7.4.1 Proof Radius 216
7.4.2 Explored State-Based Coverage 217
7.4.3 Flip-flop to Property Distance 217
7.4.4 Functional Coverage Points 217
7.5 EMERGING APPLICATIONS OF SYSTEMVERILOG ASSERTIONS WITH FORMAL METHODS 217
7.5.1 SystemVerilog Assertions Based Performance Evaluation of Digital Systems 217
7.5.2 Hybrid (dynamic and formal) Verification 218
7.5.3 Directed Random Test Generation from SystemVerilog Assertions 219
7.5.4 Achieving hard-to-hit functional coverage goals using Formal Methods 219
7.6 Temporal Debugging 222
7.7 SIMULATION OR FORMAL VERIFICATION? 224
7.7.1 Arguments for Simulation with ABV 224
7.7.2 Arguments for Formal Verification 225
7.7.3 Balance 225
7.7.4 Recommendations 226
7.7.5 Validity of Formal Verification results 226

8 SystemVerilog Assertions Guidelines 229
8.1 Typographic Guidelines 230
8.1.1 Naming Convention 230
8.1.1.1 File naming 230
8.1.1.2 Object Naming 230
8.1.1.3 Naming of Assertion Constructs 231
8.1.2 Ending Statements 231
8.1.3 Constants for Modules and Interfaces 232
8.2 Use Model Guidelines 232
8.2.1 Where to Write Properties and Assertions 232
8.2.2 Assertions for Accuracy 234
8.2.2.1 Abide by Good Verilog Coding Style Rules 234
8.2.2.2 Avoid Nested System Functions 234
8.2.2.3 Beware of unsized additions in +1 versus +1'b1 235
8.2.2.4 Beware of Property Negation Operator 237
8.2.2.5 Ensure "Write before Read" while using Local Assertion Variables 238

8.2.2.6 Be Aware of Overlapping Assertions ……. 238
8.2.2.7 Beware of Metalogical Values ……. 239
8.2.2.8 Avoid Vacuous Properties ……. 239
8.2.2.9 Avoid Contradictory Properties …… 239
8.2.3 Use $sampled Function in Action Block to Display Values of Current Variables 240
8.2.4 Accessing Local Variables in Assertions …… 240
8.2.5 Style …… 240
8.2.5.1 Avoid Unbounded Ranges …… 240
8.2.5.2 Use of Default Clock …… 241
8.2.5.3 Evaluate Assertion Relative to a Clock …… 241
8.2.5.4 Handling Resets in Properties …… 241
8.2.5.5 Defining Time Unit and Time Format Specifications for Design …… 242
8.2.5.6 Direct or Implicit Declaration of Properties ……. 245
8.2.5.7 Use Formal Arguments only when Reuse is Intended …… 246
8.2.5.8 Use module ports or Registered Signals in Properties …… 246
8.2.5.9 Standardize Action Block Error Display …… 247
8.2.5.10 Use generate Construct for Assertions Conditional on Parameters …… 247
8.2.5.11 Use Pattern Format in Documenting Assertions …… 248
8.2.5.12 Review Properties and Assertions Against Requirements ……. 248
8.2.5.13 Simulate Design …… 248
8.2.5.14 Guidelines for Debugging Assertions ……. 249
8.2.6 Using SystemVerilog assertions with Verilog RTL ……. 249
8.2.7 Using Dynamic Data Types inside Properties …… 250
8.3 Methodology Guidelines ……. 251
8.3.1 Identifying Properties from Design Specifications …… 251
8.3.2 Classification of properties …….. 251
8.3.2.1 Design Centric . …… 251
8.3.2.1.1 Style in FSM properties …… 251
8.3.2.2 Assumption Centric …….. 253
8.3.2.3 Requirement / Verification Centric …… 253
8.3.2.4 Environmental Properties ……. 254
8.3.2.5 Coverage Properties …… 255
8.3.3 Process of Writing Properties and Assertions ……... 256

9 SystemVerilog Assertions Dictionary …… 261
9.1 If COND1, then COND2 ……. 262
9.2 If COND1, then at next COND2, COND3 …… 262
9.3 If COND1, then after nth COND2, COND3 ……. 263
9.4 If COND1 and first COND2, then COND3 until COND4 …….. 264
9.5 If COND1 and first COND2, then sequence …… 264
9.6 Between COND1 and COND2, Signal 1 asserted …….. 265
9.7 If COND1 and then 1 occurrence of COND2 then sequence …….. 266
9.8 If COND1 then N Occurrences of COND2 before COND3. N is value of signal …… 266
9.9 If COND1 and within n cycles y occurrences of COND2, then COND3 …….. 268
9.10 If COND1, then COND2 until COND3 …… 269
9.11 If Cond1 then Cond2 before Cond3 ……. 269
9.12 If COND1 is followed by COND2, and COND3 is not received within 64 cycles while COND2 then Error (COND5). If COND3 is received within 64 cycles then COND4 ……. 269

9.13 For every write (COND1), data transfers must alternate between odd and even entries 271
9.14 If COND1 then COND2 in N cycles unless COND3 271
9.15 Data Integrity in Memory. Data read from Memory should be same as what was last written 273
9.16 Data Integrity in QUEUES. Interface Data Written must be properly transferred to the Receiving Hardware 274
9.17 Never 2 consecutive Writes with same Address 276
9.18 Cache controller requirement: A cached address (COND1) will eventually be retired (COND2) and after that, within 2 to 7 clocks the cache copy shall be invalidated (COND3) 277
9.19 during cond1 Never COND3 after COND2. Cond2 may occur within n cycles after Cond1 278
9.20 If COND1, then next N cycles COND2. If new COND1 before end of COND2, then COND2 extended for N cycles until no COND1 278
9.21 Never two CONDs within 2 cycles Apart 280
9.22 Assume Reset low for initial N cycles 281
9.23 If COND1 and N cycle later COND2, then COND3 until COND4, unless COND5 .. 282
9.24 If Sequence COND1 followed by N non-necessarily consecutive COND2, then N consecutive COND3 until COND4 283
9.25 If COND1, COND2 doesn't change for N clocks, unless COND1 goes high again 283
9.26 If a Sequence Starts but does not Complete, then State Register must be in ERROR state 284
9.27 COND1 and COND2 are Mutually Exclusive 286
9.28 If Address Error, then eventually good address 287
9.29 Enabling a property after a trigger 288

6 Appendix A Answers to Exercises 289
A.1 Answers to Chapter 4 Exercises 289
A.2 Answers to Chapter 6 Exercise 298

Appendix B: Definitions 305

APPENDIX C: QUICK REFERENCE GUIDE 313

APPENDIX D: CLOCK RESOLUTION 317

APPENDIX E: SYSTEMVERILOG ASSERTIONS SYNTAX 321

Index 325

All code is available for download

ch1/
ch1/cookie.sv
ch1/reqack.sv
ch1/wire_explorer.sv

ch2/
ch2/handshake2_1_3.sv
ch2/handshake2_2_1.sv
ch2/handshake2_2_1b.sv
ch2/handshake2_3_2.sv
ch2/hburstmode2_1_6.sv
ch2/interface2_3_2.sv
ch2/packets2_3_1b.sv
ch2/pushpop2_3_1.sv

ch3/
ch3/cache3_2.sv
ch3/clocking3_2_5_1.sv
ch3/formal_3_2_3.sv
ch3/if3_2_5_3_1.sv
ch3/pdata3_2_4.sv
ch3/sample3_1_1.sv

ch4/
ch4/ended4_7_1.sv
ch4/event4_7_4.sv
ch4/firstmatch2.sv
ch4/match3ed4_7_2.sv
ch4/repetitions4_5.sv
ch4/repetitions4_5_1_2.sv
ch4/throughout4_6_6.sv
ch4/triggered4_7_3.sv
ch4/within4_6_5.sv

ch5/
ch5/mixed_vhdl_sv_section5_14/
ch5/mixed_vhdl_sv_section5_14/
bind.sv
ch5/mixed_vhdl_sv_section5_14/
makefile
ch5/mixed_vhdl_sv_section5_14/
reqack.vhd
ch5/mixed_vhdl_sv_section5_14/
reqack2_tb.vhd
ch5/mixed_vhdl_sv_section5_14/
reqack3_tb.sv
ch5/mixed_vhdl_sv_section5_14/
reqack_sva.sv
ch5/mixed_vhdl_sv_section5_14
reqack_tb.vhd
ch5/oneshot5_7.sv
ch5/psl_like5_7.sv
ch5/recursion5_6.sv
ch5/refinement5_4.sv
ch5/req2send_5_3.sv
ch5/sampled5_8.sv
ch5/section5_12.sv
ch5/section5_13.sv
ch5/section5_8.sv
ch5/section5_9.sv
ch5/unbounded5_5.sv
ch5/unexpectedA_5_2.sv
ch5/unexpectedB_5_2.sv

ch6/
ch6/fifo_assert_control.sv
ch6/fifo_if.sv
ch6/fifo_if_withQ.sv
ch6/fifo_pkg_include.sv
ch6/fifo_props.sv
ch6/fifo_rtl.sv
ch6/fifo_tb.sv
ch6/vpi_sect6_2_7.sv

ch7/
ch7/tlightfail/
ch7/tlightfail/tb_bc.sv
ch7/tlightfail/trafficlight080404.sv
ch7/tlightok/
ch7/tlightok/trafficlightok10avg.sv
ch7/tlightok/trafficlightok10csv.sv
ch7/tlightok/trafficlight_props.sv
ch7/tlightok/trafficlight_top.sv

ch8/
ch8/section8_2.sv
ch8/section8_2_5_10.sv
ch8/section8_2_5_4.sv
ch8/section8_2_5_5.sv
ch8/section8_2_5_9.sv

ch9/
ch9/section9_15.sv
ch9/section9_16.sv
ch9/testmodels/
ch9/testmodels/bus_xfr_data_inte
grity_check.sv
ch9/testmodels/d17.sv
ch9/testmodels/d1819.sv
ch9/testmodels/d1to16.sv
ch9/testmodels/d20.sv
ch9/testmodels/d21.sv
ch9/testmodels/d22.sv
ch9/testmodels/d23.sv
ch9/testmodels/d24.sv
ch9/testmodels/d25.sv
ch9/testmodels/d26.sv
ch9/testmodels/d27.sv
ch9/testmodels/d281.sv
ch9/testmodels/d282.sv
ch9/testmodels/d29.sv
ch9/testmodels/
memory_data_integrity_check.sv

appendixA/
appendixA/parkinglot.sv
appendixA/slave_handshake.sv
appendixA/slave_handshake_tb.sv

FOREWORD, Surrendra A. Dudani

The study of assertions has a range of applications in hardware design verification, including bug detection in simulation and emulation, formal proofs of design correctness, functional coverage of complex behaviors, and constraint-based random stimulus generation. Assertions offer improvements at every stage of design and verification process. To guide users of assertions, the authors have previously contributed to the subject of assertions, its importance in design verification and provided numerous examples illustrating its usage.

SystemVerilog Assertions Handbook introduces SystemVerilog's language of assertions from the point of view of practitioners that primarily consist of design and verification engineers. SystemVerilog language evolved from Verilog hardware description language as an industry standard language to describe hardware design as well as to write verification programs supporting the enormous task of verifying the design. The capability to encapsulate design and verification in one language is invaluable to thousands of engineers who have the formidable challenge of designing and verifying present day complex hardware systems. One of the unique features of SystemVerilog is the ability to freely express the interaction of the behavior of sequences and assertions with other features of the language that direct stimulus generation and coverage.

The past few years have brought a shift in design verification methodology, as new techniques have emerged from the work of research and academic institutions to industries that promoted them as matured products providing improvements in quality, productivity, and management of hardware design projects to successful completion in a reasonable timeframe. No longer an engineer thinks only of writing tests and checking the results of simulation as the basis of verification. The art of checking the sanity of results has been formalized into assertions, expressed in concise language form that has a mathematical foundation to also allow formal proof techniques. Despite its compelling benefits, assertion based verification has not reached the desks of many engineers due to its overwhelming expressive power in the language features and a new style of specification. The need of books, guides, and training in informing and instructing engineers with these new concepts is plainly evident.

The authors of this book, Ben Cohen, Srinivasan Venkataramanan and Ajeetha Kumari, are well experienced in the real world of design verification to present a solid introduction to assertions in a practical manner to captivate many engineers who may otherwise be reluctant to take up a new subject. Although hardware engineers are well versed in parallel behaviors interlaced with timing subtleties, assertion language features bring these together in a new and concise form of expressing. The authors have illustrated these concepts in a step-by-step manner with practical examples to relate the outcome of language constructs with the results intuitively expected by the readers. Before delving into the details of the language, a chapter is devoted to acquaint the reader with assertions, its methodology of usage and benefits. Later, the chapters are organized to bring out simpler to more advanced features of expressing sequences, properties and assertions. The depiction of design process using SystemVerilog helps the reader to step back from the language semantics and view the role of assertions in various verification tasks.

The readers of this book will benefit from the clear presentation of concepts together with practical examples and appropriate usage of the language features. Design engineers will find a wealth of easy to apply assertions as checkers to improve the quality of their day-to-day design projects. On the other hand, verification engineers will learn advanced concepts to simplify writing temporal behaviors at the system level to perform system level checking. Further, the book will also assist architects of methodology in deploying advanced verification techniques using SystemVerilog Assertions. This book is a guide much needed to fully capitalize many benefits offered by SystemVerilog Assertions.

Surrendra A. Dudani
Synopsys Scientist
http://www.synopsys.com/

FOREWORD, Stuart Sutherland

Assertion-based design and verification is an absolute necessity in today's large, complex designs. For that matter, the use of assertions applies even to the simplest of designs. *Every design engineer should be adding assertion checks to his design!* Assertions both document and check the design engineer's assumptions and expectations about the design functionality. *Every verification engineer should be taking full advantage of assertions!* Assertions can dramatically decrease the amount of effort required to define intelligent, self-checking testbenches, and, at the same time, increase the effectiveness of the testbench. As this book shows, assertions offer countless other benefits to both the design engineer and the verification engineer.

The new and exciting SystemVerilog standard adds hundreds of powerful extensions to the IEEE Verilog language standard. Prominent among these extensions is a native assertion language that is fully compatible with the existing Verilog language. SystemVerilog Assertions, abbreviated as SVA, syntactically and semantically fit into Verilog code. Engineers can directly specify assertions in their Verilog models and testbenches, without having to hide the assertions within comments, pragmas or conditional compilation directives. However, SVA is a very rich language in its own right, and is not simple to adopt. SVA has the ability to concisely describe the expected (or unexpected) results of extremely complex sequences of changes within a design. There are a number of conference papers, and even some books, that discuss SystemVerilog Assertions. These papers and books discuss the importance of SVA, and how to use an assertion based verification methodology in design projects. However, I have yet to find a paper or book that teaches how to *write* SystemVerilog Assertions.

This book fills that void. It introduces the concepts and importance of assertion-based verification, and then goes into great depth on how to write both simple and complex assertions using the SystemVerilog Assertions language. Hundreds of examples illustrate the proper usage of SVA. Many of the examples are based on real-world designs. The examples do more than just illustrate how to write an assertion. The examples serve as a cookbook of assertions that can be applied to a variety of designs.

I have been involved with the definition of the SystemVerilog standard since its inception, and am excited to see this great book on SystemVerilog Assertions. My company, Sutherland HDL, Inc., provides expert training and consulting on Verilog and SystemVerilog. We are very active in helping companies adopt SystemVerilog in their current and upcoming design projects. This book will be a valuable tool that we will make full use of in our training workshops and consulting work.

I expect that every design and verification engineer will find that this book is an essential resource in their day-to-day work.

Stuart Sutherland,
Sutherland HDL, Inc.
http://www.sutherland-hdl.com

FOREWORD, Harry D. Foster

An industry-wide effort has been underway for the past few years to extend the capabilities of the Verilog language. The fruit of that labor is SystemVerilog. The SystemVerilog extensions include enhancements in modeling as well as new features for verification. One of the key features added to the language is the ability to specify formal properties as assertion checks and as elements of a functional coverage model.

So why is this important? With the advent of design reuse and IP-based SoC design, two verification challenges have emerged: verifying that the IP complies with its specification and verifying that the IP is interoperable with other compliant devices (that is, adheres to various interface standards). Although the act of specification is fundamental to the process of functional verification, historically, the process of specification has consisted of creating a natural language description for a set of design requirements. And unfortunately, this form of specification is both ambiguous and, in many cases, unverifiable (due to the lack of a standard machine-executable representation).

As assertion and property language standards such as SystemVerilog Assertions (SVA) gain a foothold, they address the problem of ambiguities in natural language specification and reduce the time spent in verification. And as a result, IP providers are able to adopt an assertion-based verification (ABV) methodology to provide verifiable forms of IP specification.

Adopting an ABV methodology benefits both IP producers and consumers. For the IP producer, developing an assertion-based specification for the IP has a collateral benefit—the formal specification process often uncovers misconceptions about the implementer's original intent. Thus, the time invested in developing the specification is time well spent. And generally, the benefits are realized early in the design and verification cycle, before the IP producer applies any form of verification to the IP. With the assertion-based specification in hand, the IP producer is positioned to verify IP compliance and interoperability. For the IP consumer, an assertion-based specification reduces integration time by unambiguously clarifying proper IP behavior under various configurations, while providing a way to verify the SoC's interoperability with the IP. But perhaps most notably, assertion-based specification benefits both producers and consumers by unifying the verification process with a

single form of specification that can be automatically leveraged across a diverse set of verification tools, such as formal verification, simulation, emulation, and even synthesis.

Although a pool of published data confirms the benefits of adopting an ABV methodology, few guidelines exist for coding effective assertion-based specification. Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari have addressed this challenge by creating an excellent source for mastering the art of assertion-based specification. *SystemVerilog Assertions Handbook* should be a part of every RTL design and verification engineer's library.

Harry D. Foster
Chief Methodologist
Jasper Design Automation

FOREWORD, Tarak Parikh

The growing adoption of the SystemVerilog Assertions language is enabling design and verification engineers to formalize what has been done informally for many years. Assertion-based verification at its essence checks that a design behaves a certain way, and has historically been done mostly in simulation, and using formal verification with various methods and languages.

Assertion-based verification using formal model checking also has been used in the industry, but the solutions have ranged from proprietary assertion languages to standard, but difficult to use languages such as CTL and LTL. While formal model checking is a very powerful method to find corner-case bugs and to completely verify the design meets specific requirements, it has not been widely adopted in the industry until recently. The major barriers with model checking have been the difficulty in writing assertions, as well hard to use tools with inadequate debugging capabilities, not to mention the lack of a standard language.

Now, the existence of a standard assertion language suitable for use with formal model checking, integrated with a design language familiar to all design and verification community makes it much easier for EDA vendors to create tools usable by a wide audience. It also gives the engineer more options, since they are not locked into any particular proprietary solution.

Although SystemVerilog Assertions simplify design verification and makes it much easier to write powerful and portable checks, it is not a panacea for all of the verification challenges. For example, it is not well suited for higher-level data flow checks, such as verifying that a packet put into a system is eventually delivered to the right receiver with the correct data. However, when combined with the ability to run these assertions in simulation, adoption of SystemVerilog Assertions creates a significant improvement in verification productivity.

SystemVerilog Assertions are good for checking that design protocols meet specification, and that critical design functions are not violated. For example, it is simple to write an assertion to check that a bus request always is acknowledged within a certain number of clock cycles. With SystemVerilog Assertions, this sort of check may be used in simulation, or proven using formal model checking tools

such as our *@Verifier* product. The SystemVerilog Assertions also allow debugging tools to provide one unified interface for the creation, verification, and debugging of assertions, regardless if they are used in simulation or formal model checking. Our patent-pending *Assertion Studio* technology provides this sort of interface.

@HDL has strived to make assertion-based verification a usable and productive verification technique. The standardization of SystemVerilog Assertions language further enables our tools to fit seamlessly into the verification flows of our customers.

@HDL is pleased to have been able to help the authors develop this book. While powerful, SystemVerilog Assertions can be complicated. This book reviews the language elements and provides clear examples on what is legal and what is not. It is an excellent resource for the novice and experienced assertion-based verification engineer, and is useful for both learning the language and as a reference when developing SystemVerilog Assertions.

Tarak Parikh
Vice-President, Products
@HDL
http://www.athdl.com/

FOREWORD, Keith Rieken

Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari have once again provided a timely, application-oriented approach to an emerging design methodology. As design practices demand ever-increasing productivity in functional verification, the SystemVerilog language coupled with proven verification methodologies will likely be a cornerstone to improved productivity.

The 0-In Business Unit of Mentor Graphics is similarly committed to improving the overall productivity of existing functional verification processes by employing Assertion-Based Verification (ABV) methods. Having pioneered the commercialization of ABV technologies, 0-In believes SystemVerilog provides another alternative for realizing the productivity improvements associated with ABV. The standardization of a language that enables a sophisticated testbench automation solution, design construct improvements and language-based assertion specification will accelerate the adoption of ABV by the design community.

Mentor Graphics continues to support such standardization efforts through the incorporation of SystemVerilog support into its Scalable Verification solution. Through a combination of the Mentor solution and the applied approaches demonstrated in this *SystemVerilog Assertions Handbook*, teams will be able to quickly realize the benefits of a SystemVerilog ABV solution.

Keith Rieken
Director, Technical Marketing Engineering,
0-In Functional Verification Business Unit,
Mentor Graphics
http://www.mentor.com/

FOREWORD, Yu-Chin Hsu

Advanced languages and assertions are coming of age because of mounting design complexity, shrinking time-to-market, and a leveling off in productivity. The potential of SystemVerilog is not merely in its impressive host of useful modeling elements, but also lies in the benefits of a unifying coherent framework for comprehensive design, verification, and debug methodology. SVA, the assertion temporal language portion of SystemVerilog, draws on the strengths of three languages: PSL from Accellera with roots in Sugar, a language used at IBM, VERA from Synopsys, and OVA from Synopsys, also with origins stemming from the industrial setting at Sun Microsystems. The SVA syntax constructs and semantics are designed to be native to SystemVerilog to make them more easily integrated with the design and testbench, and readily approachable by designers.

Assertions can be used in a variety of roles: constraints, checkers, integration monitors, and for functional coverage. Assertions are a formal means to bridge the gap between design specification and implementation, also a way to consolidate design, verification, and debug. Assertion driven verification is perceived as the enabling methodology for early bug detection, bug source localization, and verification reuse. Given the complexity of SoC designs, and component heterogeneity challenges, Novas has developed an enhanced debug framework targeted at empowering designers to not only manage and sustain – but rather to grow – their creativity in the face of the design challenges. Assertions are a cornerstone of this development framework that revolves around automated powerful analysis and debug, and intuitive and efficient interaction with the user. We see assertions as an ideal entry, interaction, and abstraction mechanism for design and debug that boosts design productivity and understanding.

Ben Cohen and co-authors have published numerous books about design and verification of HDLs. This publication follows on the heels of the book on PSL, and contains many useful comparisons and contrasts. Readers will undoubtedly enjoy the *SystemVerilog Assertions Handbook* for it addresses the language from an application-oriented viewpoint. Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari also point out in this book many subtleties in the language and its underpinnings. With the development of OVA, PSL and now SVA, it is quite conceivable that any one user will likely encounter several of these assertion languages (even within the same project) each with its particular strengths, so a good understanding of the foundation of temporal languages and the particulars of each is well advised.

Yu-Chin Hsu
Vice President, Research and Development
Novas Software, Inc.
http://www.novas.com/

FOREWORD, Alain Raynaud

Fifteen years ago, schematics designers surely must have wondered what went wrong with their design practices when first explained the rules on latch inference for the synthesizable subset of Verilog. You are likely to experience the same puzzled feeling today when reading about the vacuous success of temporal properties. But today, just like fifteen years ago, it would be a huge mistake to dismiss the topic of assertion-based verification (ABV) as specialized or limited in scope.

Designers went from expressing the structure of the design through schematics capture, to writing register transfer level code where the key concept is to define everything that happens one clock cycle at a time. With its assertions, SystemVerilog is the first mainstream unified language to break that clock barrier and allow the expression of relationships across many cycles. This will have a tremendous impact on hardware design in the coming years. Verification is the obvious first methodology to face this new wave, and a large part of this book rightfully focuses on verification. Coverage-driven verification, in particular, is bound to see major improvements and hopefully will receive the attention it deserves. From where I sit, in the emulation side of the verification problem, I'd happily see more emulation cycles being dedicated to running actual interesting application code and fewer cycles spent on blind random regression testing.

Design practices will also evolve gradually as EDA tools start making use of the full power of assertions. We are not that far off from ABD: Assertion-Based Design; and this book touches on that subject as well, making it a must read for every designer who wants to stay ahead of the curve. When those EDA vendors come visit you a year from now to pitch their revolutionary new design tools, you'll already understand the underlying technology and you'll be able to separate the wheat from the chaff.

Ben Cohen, Srinivasan Venkataramanan and Ajeetha Kumari have written a book that will teach you more than you ever wanted to know about SystemVerilog Assertions. It can be put to practical use today and will give you an edge for tomorrow. All the right ingredients and topics are covered: from coding style to reference, from coverage to formal proof. This book not only should be part of every verification engineer's library, but designers ought to be on-board as well.

Alain Raynaud
Technical Director
EVE USA, inc.
http://www.eve-team.com/

PREFACE

The Book

SystemVerilog Assertions Handbook is a follow-up book to *Using PSL/Sugar for Formal and Dynamic Verification 2nd Edition*. It focuses on the assertions aspect of SystemVerilog, along with an explanation of the language concepts along with many examples to demonstrate how SystemVerilog Assertions (SVA) can be effectively used in an Assertion-Based Verification methodology to verify designs written in HDLs like SystemVerilog, Verilog, or VHDL. The integration of assertions in SystemVerilog proves very beneficial for the definition of a verification environment because SystemVerilog is a modern language with powerful and advanced constructs like interfaces, queues, associative array, semaphores, system functions, classes, methods, packages, safe pointers, etc.

This book presents different classes of designs, and demonstrates how SystemVerilog Assertions are used in the design process from requirements document, verification plan, design and verification using simulation and formal verification. Many of the examples use the advanced features of SystemVerilog including packages, interfaces, types, and binding. In addition, synthesizable RTL SystemVerilog code examples were synthesized to demonstrated feasibility. Other features provided in this book are a "dictionary" of English to SystemVerilog Assertions examples, guidelines in the use of SystemVerilog Assertions, and a quick reference guide of the SystemVerilog Assertions syntax. This book represents the collaboration of three authors who are experts in system engineering, architecture, and design and verification with hardware description languages (HDLs) and hardware verification languages (HVLs), along with experience in authoring books, thus bringing more synergism to this *SystemVerilog Assertions Handbook*.

The Intent

One of the reasons that we, the authors, decided to write this handbook on SystemVerilog Assertions is the positive impact that Assertion-based Verification (ABV) is providing, and we believe that SystemVerilog is setting up a new viable and effective standard in the design and verification processes. We also felt that the "assertions" aspect of SystemVerilog needed special emphasis. Thus we decided to maintain the focus of this book on SystemVerilog Assertions, with usage of many of the new features that SystemVerilog provides. We are assuming that the users are familiar with SystemVerilog, and have access to the SystemVerilog LRM and to books that address SystemVerilog language.[1]

---

[1] * *SystemVerilog Language Reference Manual* http://www.systemverilog.org/
* *SystemVerilog For Verification*, Tom Fitzpatrick, Dave Rich, Aturo Salz and Stuart Sutherland, 2005, Springer Springeronline.com
* *SystemVerilog For Design A Guide to Using SystemVerilog for Hardware Design and Modeling* Stuart Sutherland, Simon Davidmann, Peter Flake, KAP, June 2003, ISBN 1-4020-7530-8

Assertion-Based Verification is changing the traditional design process because that methodology helps to formally characterize the design intent and expected operations.[2] ABV also quickens the verification task because it provides feedback at the white-box level.[3] As a formal property specification language, SystemVerilog Assertions facilitate automation of common verification tasks that can be exploited across various verification methodologies.

As designers and consultants/trainers, we experienced many designs that were weakly specified. The RTL modeling lacked information about properties and design characteristics, and that led to difficulties and/or ambiguities in the maintenance and verification processes. A design specification is helpful in defining requirements. However, specifications are generally defined in an informal language, like English. They lack a standard machine-executable representation and cannot be dynamically simulated and/or statically processed by a formal verification tool to ensure compliance to requirements.

Assertion-Based Verification with SystemVerilog Assertions

In a manner similar to *Accellera Property Specification Language* (*PSL*)[4], the assertion aspect of SystemVerilog was developed to address these shortcomings. It gives the design architects a standard means of specifying design properties using a concise syntax with clearly defined formal semantics. Similarly, it enables the RTL designers to capture design intent and assumptions in a verifiable form, while enabling the verification engineers to validate that the implementation satisfies its specification through dynamic (i.e., simulation) and formal verification options. Furthermore, it provides a means to measure the quality of the verification process through the creation of functional coverage models built on formally specified properties. It provides a standard means for hardware designers and verification engineers to rigorously document the design specifications using a machine-executable format.

SystemVerilog with assertions improves the quality of digital designs and helps eliminate defects per the **Six Sigma methodology**[5] because assertions play an important role in a unified verification methodology ranging from requirement definitions through design and verification (see Chapter 6 for discussion on the design process with SystemVerilog Assertions). Assertions express functional design intent and can be used to express assumed input behavior, expected output behavior, or forbidden behavior. Assertions allow the architects or designers to capture the design intent and assumptions in a manner that can be verified in the implementation. Assertions are captured during the development process and are continuously verified throughout the design and verification process. Working in a unified verification methodology, assertions reduce the verification time by detecting bugs earlier, and by isolating where a bug is located (by being closer to the source of error). In addition to detection of property violations, assertions improve the efficiency in a unified methodology by improving reuse, enhancing testbench checking, and capturing coverage information. Per Lionel

---

[2] *Assertion-Based Design, Second Edition*, Harry D. Foster, Adam C. Krolnik, David J. Lacey
June 2004, ISBN 1-4020-8027-1,
The SystemVerilog Verification Methodology Manual (VMM), 2005 Springeronline.com

[3] *Writing Testbenches: Functional Verification of HDL Models*, Janick Bergeron, Kluwer Academic Publishers

[4] http://www.accellera.com
http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf

[5] http://www.isixsigma.com/sixsigma/six_sigma.asp
Six Sigma is a disciplined, data-driven approach and methodology for eliminating defects (driving towards six standard deviations between the mean and the nearest specification limit) in any process -- from manufacturing to transactional, and from product to service.

Benning's[6] experience, designers created fewer initial bugs in the RTL as an ABV methodology forced them to think more clearly and accurately about what to design. Also, properties are more accurate and less prone to misinterpretation than comments in the RTL.

When we were first exposed to SystemVerilog Assertions, we realized its strong potentials in specifying design functional specification requirements and properties in a manner easy to learn, write, and read. We particularly liked the concise syntax of the assertions, which are tightly integrated with SystemVerilog. We also appreciated the rigorously well-defined formal semantics, and expressive power of the language, permitting the formal specification (and documentation) for a large class of real world design properties. Expressing the same functionality with HDLs would require extensive coding with explicit FSM machines.

SystemVerilog Tool Support

Today, many companies are supporting SystemVerilog with assertions, and the list is growing. A list of vendors supporting SystemVerilog is shown at the site shown in the footnote.[7] During the process of writing this book, we had access to tools from Synopsys and @HDL. The intent of this book is to present the general concepts of using SystemVerilog with assertions for dynamic and formal verification in a tool independent manner.

Why ABV with SystemVerilog Assertions?

Assertion-Based Verification moved the traditional design process from an informal RTL coding approach with typically poor documentation to a process that provides the following benefits: 1) addresses and documents design decisions; 2) documents design properties and assumptions; 3) addresses solutions (e.g., interfaces, implied FSMs) to requirements prior to any RTL coding; 4) addresses verification of assertions, which items to watch out for during the design of RTL and testbench; 5) facilitates functional coverage to ensure that simulation addresses complex timing-based corner cases; 6) provides excellent basis for design and verification reviews; 7) simplifies design of testbench reference models or scoreboards, which verifies the correctness of results; 8) guides testbench vectors for conditions to be addressed.

It is important to note that SystemVerilog Assertions define the properties, and are implementation independent. It presents a different viewpoint of the design. The property definitions may imply FSMs in the implementation. However, those properties do not necessarily show any design optimizations, such as the use of don't-care conditions. As the design matures, it may be necessary to revisit the assertions, as they may be too restrictive. In addition, it may also be necessary to add assertions at the functional level. But this experience of tuning the assertions and the design is healthy because it forces users to delve into the requirements and implementation.

Our experience with the usage of SystemVerilog Assertions for front-end design definitions demonstrated that SystemVerilog Assertions are very powerful in the process of delving into design requirements, design architecture, and definition of restrictions imposed by the architecture. We found the property and assertion definitions more expressive and precise than the use of a natural language, e.g., English. The RTL design and verification tasks were greatly simplified as a result of using this assertion-based methodology because it alleviated the need to write a thorough testbench reference model prior to debugging the model. During simulation the assertions immediately alerted us of

---

[6] *Verifiable RTL Design: A Functional Coding Style Supporting Verification Processes in Verilog*, Lionel Benning and Harry Foster, Kluwer Academic Publishers

[7] http://www.synopsys.com/partners/systemverilog/systemverilog_partners.html

design and testbench errors. In fact, we strongly recommend the use of ABV with SystemVerilog on design projects. ABV is a very viable methodology for the definition and verification of designs. We must admit though that at times assertions are very frustrating because they (correctly) insisted that our designs were in error when we believed that we had all the necessary fixes!!!

More about the Book

SystemVerilog Assertions Handbook addresses the practical aspects of understanding and using assertions with SystemVerilog. This is accomplished by first defining the language, in a non-LRM manner with many examples to explain the various syntax and nuances of the language. This is then followed by explaining how SystemVerilog assertions are used in the design process through all phases of the design including system level definition, architectural and verification plans, RTL and testbench designs, dynamic and static verification. This is done by example, using a simple synchronous FIFO as a project model. Formal verification concepts and application of formal verification with SystemVerilog Assertions are then presented, along with an example of a traffic light controller to demonstrate the application of tools, and types of results typically presented by such tools. A set of language and application guidelines emanating from our experience with SystemVerilog Assertions is presented. A "dictionary" of examples demonstrates how various English requirements can be translated into SystemVerilog properties.

Book Organization

Chapter 1 provides an introduction to Assertion-Based Verification. **Chapter 2** serves as an introduction to SystemVerilog Assertions (SVA) concepts with emphasis on properties and assertions. It prepares the readers for Chapters 3, 4, and 5, which represent the "core" of SystemVerilog Assertions. **Chapter 3** delves into understanding properties. **Chapter 4** delves into the understanding and application of sequences that represent the real potential of SystemVerilog Assertions. **Chapter 5** provides a deeper appreciation of SystemVerilog Assertions by addressing advanced topics for properties and sequences. **Chapter 6** addresses the methodologies in using properties / sequences / assertions during the requirement and verification planning phases at the requirement and verification planning levels, in addition to the RTL and testbench levels. It first explains the process, and then demonstrates an application of assertions in the requirements specification and verification plan using a synchronous First-In First-Out (FIFO) as an IP (Intellectual Property) block. SystemVerilog packages, interfaces, modules, and bindings are also demonstrated. **Chapter 7** addresses the formal verification aspects of SystemVerilog Assertions. It focuses on Formal Verification (FV) methodologies for functional verification of RTL designs. It provides a case study using a traffic light controller model. **Chapter 8** provides a summary a rich set of guidelines in using SystemVerilog Assertions. These guidelines emerged from experience with usage of Assertion-Based Verification with Accellera's PSL, vendor's recommendations, code reviews, and LRM documentation. **Chapter 9** represents a "dictionary" of classes of application examples that translate English descriptions of properties to SystemVerilog properties**. Appendix A** provides the answers to the exercises asked at the end of some chapters. **Appendix B i**s a summary of terms and definitions used within this book. **Appendix C** is a SystemVerilog Assertions quick reference guide of the syntax and examples. **Appendix D** represents the SystemVerilog Assertions syntax. The book **Index** provides a page lookup for information available in this book.

Acknowledgements

SystemVerilog Assertions Handbook could not have been written without the support of Synopsys who provided us with SystemVerilog tools including *vcs* and *Magellan* and helped us in the review process.

We thank *Accellera* for granting us permission to reproduce some material from the *Accelera's* SystemVerilog 3.1a Language Reference Manual, the document that defines the rules of SystemVerilog and SystemVerilog Assertions.

Several SystemVerilog experts participated in the review process of this book. The review is a necessary step to iron out areas of disagreements, and to provide a piece of work that meets user's requirements in the use of SystemVerilog Assertions. In that endeavor, we sincerely thank the following people and organizations:

- **Stuart Sutherland**, Stuart Sutherland of Sutherland HDL, Inc., an expert Verilog and SystemVerilog instructor and consultant -- for sharing his SystemVerilog expertise, and providing valuable and constructive comments in the review of our book, and for writing a foreword.[8]
- **Surrendra Dudani,** Synopsys Scientist, for reviewing our book and writing a foreword.
- **Eduard Cerny,** Synopsys Principal Engineer, R&D, VG, for his in-depth review especially into the language part and providing feedback.
- **Badri Gopalan**, Synopsys **–** for being very supportive and encouraging throughout the process of book writing and for his feedbacks from a experienced Assertions user perspective.
- **Alessandro Fasan**, Synopsys, Product Marketing Manager, functional verification methodology and RTL formal verification, for reviewing the book and for providing constructive comments and suggestions.
- **Bruce Greene**, Synopsys technical marketing manager, for sharing his knowledge and for testing some of our models in Magellan.

[8] **Stuart Sutherland,** Sutherland HDL, Inc., Training Engineers to be Verilog and SystemVerilog Wizards.
http://www.sutherland-hdl.com

- **Cliff Cummings**, Sunburst Design, for providing technical feedback on our book.[9]
- **Harry Foster,** Chief Methodologist , Jasper Design Automation, for reviewing our book and for writing a foreword.[10]
- **Tarak Parikh,** Vice-President, Products**,** @HDL, for reviewing our book, executing some of our SystemVerilog Assertions models with @HDL *@Verifier* and *@Designer* formal verification tools, and for writing a foreword.[11]
- **Keith Rieken**, Director, Technical Marketing Engineering, 0-In Functional Verification Business Unit, Mentor Graphics, for reviewing our book and for writing a foreword.
- **Yu-Chin Hsu**, Vice President, Research and Development, Novas Software, Inc., for providing technical comments in the review of our book and for writing a foreword.[i]
- **Dr. Bassam Tabbara,** Novas Software, Inc., Architect, R&D, for providing technical comments in the review of our book
- **Alain Raynaud**, EVE USA, inc., Technical Director, for reviewing our book and for writing a foreword.[ii]

We also thank *Forte Design Systems*[12] for granting us a license of *TimingDesigner* as a tool to draw timing diagrams for use in this book.

I, Ben Cohen, especially thank my wife, Gloria Jean, for supporting again me in this endeavor.

---

[9] **Cliff Cummings,** Sunburst Design, Expert Verilog, SystemVerilog, Synthesis & Verification Training http://www.sunburst-design.com/

[10] **Harry D. Foster,** Chair of Accellera Formal Verification Technical Committee and Chief Methodologist at Jasper Design Automation,. Author of *Principles of Verifiable RTL Design,* Kluwer Academic Publishers, Second Edition, 2001, *Assertion-Based Design,* Kluwer Academic Publishers, 2003, and contributor to *Advanced Formal Verification,* Springer (http://www.springeronline.com/)

[11] **Tarak Parikh**, VP of Product Engineering, @HDL, http://www.athdl.com/

[12] *TimingDesigner* is a flexible, interactive timing analysis and diagram tool. http://www.timingdesigner.com/ http://www.forteds.com/

Sculpture Created by my Wife Gloria to
Express my Long Hours with a Laptop in the Creation of HDL Books

About the Authors

Ben Cohen is currently an HDL and Property languages (PSL, SystemVerilog Assertions) trainer and consultant. He has technical experience in digital and analog hardware design, computer architecture, ASIC design, synthesis, and use of hardware description languages for modeling of statistical simulations, instruction set descriptions, and hardware models. He applied VHDL since 1990 to model various bus functional models of computer interfaces. He authored *VHDL Coding Styles and Methodologies*, first and second editions, and *VHDL Answers to Frequently Asked Questions*, first and second editions, *Component Design by Example, Real Chip Design and Verification Using Verilog and VHDL*, and *Using PSL/SUGAR with Verilog and VHDL, Guide to Property Specification Language for ABV (1st Edition*, also translated to Japanese by Cadence*)*, and *Using PSL/Sugar for Formal and Dynamic Verification, 2nd Edition.* He was one of the pilot team members of the VHDL Synthesis Interoperability Working Group of the Design Automation Standards Committee who authored the *IEEE P1076.6 Standard for VHDL Register Transfer Level Synthesis.* He is currently a member of the *VHDL* and *Verilog Synthesis Interoperability Working Group of the Design Automation Standards Committees*, and *Accellera OVL and PSL* standardization working groups. He taught several VHDL and PSL training classes, and provided VHDL consulting services on several tasks.

VhdlCohen Publishing
Email: VhdlCohen@aol.com ben@abv-sva.org
Web: http://www.vhdlcohen.com/

Srinivasan Venkataramanan is currently employed as a Verification Solutions Engineer with Synopsys, India Private Ltd., Bangalore – India. His areas of interest are the emerging verification solutions and methodologies, such as SystemVerilog, Assertion-Based Verification, formal verification, Coverage Driven Verification (CDV), etc. He was actively involved in the verification of leading edge high-speed, multi-million gates ASIC designs. He successfully developed complex verification environments using advanced methodologies, such CDV using Verisity's *Specman*, ABV etc. Prior to joining Synopsys, he worked at Intel, Philips Semiconductors, and RealChip communications in the areas of front-end design and verification of ASICs with several HDLs and HVLs, including VHDL, Verilog, *Specman*, and *Vera*. Srini was active in several technical discussion forums, working committees of OVL, SystemVerilog, VHDL-200X Testbench & Verification, etc. Srini holds a Masters Degree from the prestigious Indian Institute of Technology (IIT), Delhi in VLSI Design, and Bachelors degree in Electrical engineering from TCE, Madurai. Srini has co-authored the book *Using PSL/Sugar for Formal and Dynamic Verification, 2nd Edition.*

Web: http://www.noveldv.com/
Email: srini@abv-sva.org

Ajeetha Kumari is currently an independent consultant in the area of front-end design and verification based in Bangalore, India. She co-authored *Using PSL/Sugar for Formal and Dynamic Verification, 2nd Edition.* Her interests include front-end design and verification methodologies, and she is actively involved in exploring new techniques and tools in this arena. She has also been involved in EDA tool evaluations. She has experience with several HDLs and HVLs including Verilog, VHDL, SystemVerilog, PSL, SystemVerilog Assertions and Vera. She currently maintains a Verification centric web site. Her interests also include mixed-signal design methodologies, which she developed as a follow-up of her research work during her Masters degree. She received her M.S. in Electrical engineering from the prestigious Indian Institute of Technology (IIT), Madras and Bachelors from TCE, Madurai.

NOVEL Design Verification
Email: ajeetha@noveldv.com ajeetha@abv-sva.org
Web : http://www.noveldv.com/

DISCLAIMER

Every attempt was made to ensure accuracy in the specifications and implementation of the languages (HDLs and SystemVerilog Assertions) and models. However, all code provided in this book and in the accompanied website is distributed with *ABSOLUTELY NO SUPPORT* and *NO WARRANTY* from the authors. Neither the authors nor any supporting vendors shall be liable for damage in connection with, or arising out of, the furnishing, performance or use of the models provided in the book and website.

Without permission, use or reproduction of the information provided in this book and on the linked website for commercial gain is strictly prohibited.

1 ROLE OF SYSTEMVERILOG ASSERTIONS IN A VERIFICATION METHODOLOGY

Recent statistics suggest that Functional Verification is one of the primary bottlenecks in a typical design cycle. To counter this problem the late 1990s saw a flurry of proprietary verification languages (aka HVLs), both procedural and declarative in nature. Though these languages enjoyed some good success at leading design houses, they are not yet as widely adopted as HDLs themselves. A recent survey revealed that more than 40% of the ASIC community do not use any of these HVLs[1]. One of the primary reasons for this is the lack of multiple EDA Vendor support for any of these languages. Accellera's SystemVerilog is being developed to address this very same issue – providing a STANDARD and powerful language that combines the best of Design and Verification capabilities in a single language. SystemVerilog also includes a powerful declarative language subset that provides a means to formally capture temporal requirements of digital systems, more popularly known as SystemVerilog Assertions. As with any new language, methodology, technology – the following questions come up immediately to a user's mind:

- Is the use of SystemVerilog Assertions a good verification strategy?
- Why should SystemVerilog Assertions be used in the definition and verification of designs?
- Can/should entire functional verification task be performed using ABV?

[1] DVCon04 Trip report http://www.deepchip.com/items/dvcon04-05.html

- What other verification strategies can be used with SystemVerilog Assertions?
- Are SystemVerilog Assertions solely restricted to applications that use SystemVerilog?

This chapter addresses these questions. It assumes that the reader has a basic understanding of HDL and SystemVerilog. This chapter also provides, for the reader's benefits, some background information on Assertion-Based Verification and the history of SystemVerilog Assertions.

1.1 HISTORY OF DESIGN VERIFICATION METHODOLOGIES

SystemVerilog – the recently ratified hardware description and verification language (HDVL) standard – is a major extension of the established IEEE 1364-2001 Verilog language, and was developed by Accellera to dramatically improve productivity in the design of large gate count, IP-based, bus-intensive chips. SystemVerilog is targeted primarily at the chip implementation and verification flow, with powerful links to the system level design flow.[2]

SystemVerilog Assertions (SVA) are an integral part of SystemVerilog and support Assertion-Based Verification (ABV) methodologies. It was developed to provide system engineers, designers, and verification engineers the means to describe complex behaviors about the design requirements and properties in a clear and concise manner. With the assertions unified syntactically with the rest of SystemVerilog, the user is able to define assertions either as externally defined design units bound to the Design Under Verification (DUV), or as embedded assertions inline with the RTL design and other verification code. These methods allow the tools to infer a great deal of information from the context of the surrounding code. Another useful feature of SystemVerilog Assertions is its testbench interaction, which unifies assertions with the design and verification language by communicating information to the testbench.[3]

Assertion-Based Verification is changing the traditional design process because that methodology helps to formally characterize the design intent and expected operations. Assertion-Based Verification also guides the verification task, quickens the verification because it provides feedback at the white-box level, and eases the design of the testbench as it transfers more of the "verifier" task onto the assertions, instead of a user-defined model. In addition, SystemVerilog Assertions allow the verification environment to be reactive, meaning that based on results of the assertions (pass or fail) different actions may be taken. Formal verification can use properties to exhaustively verify a design without a testbench, thus speeding up the verification process and increasing the quality of the design.

The concept of using assertions is not new. For ages, parents have told their children statements like "there are ten cookies in the cookie jar, and during the day, if you're hungry, you may at most take ONE cookie! At the end of the day I want to see at least 9 cookies in the jar … and I'll count the number of cookies every hour". That parenting statement is really an assertion that is verified by inspection. It can be expressed in SystemVerilog Assertions as follows:

```
property pGetAcookie;
@(posedge hour_clk) hungry && cookie_count > 9 |=> eat_1_cookie;
endproperty : pGetAcookie
apGetAcookie : assert property (pGetAcookie);
```

---

[2] http://www.systemverilog.org/

[3] http://www.eedesign.com/features/exclusive/OEG20031015S0049

This property states that "if hungry and there are more than 9 cookies, then you must eat a cookie". If a cookie is not eaten by next hour clock, then it is an error.

```
property pCookiesInJar;
 @(posedge hour_clk) not (cookie_count <9); // never less than 10 cookies in the jar
endproperty : pCookiesInJar
apCookiesInJar : assert property (pCookiesInJar) else punish_the_kid();
```

Should either of the properties *pGetAcookie* or *pCookiesInJar* be violated, the assertion would flag the error. If these were assertions for an HDL design, then simulation or formal verification would catch any design errors.

Over the years, as technology progressed and densities of designs increased, new verification methods were needed and developed. Figure 1.1-1 shows a summary of methods used over the last 35 years. The 1970's TTL and MSI technology appeared. Designs were defined with schematics on velum paper[4], and later electronically. However, verification was mainly performed with detailed paper design reviews, and through breadboard modules, which often used wire-wrap boards. During the 80's simulation tools became popular, but each vendor maintained their graphical design entry and proprietary simulation environment. Testbenches were hard to build with non-standardized tools, and verification relied on manually viewing the simulation results, with regression tests performed by comparing captured simulation runs against new updates. Any timing change would invalidate the golden copy.

In 1987, VHDL became a standard, but the use of HDLs was viewed critically because it was a major change to the "working methodologies." Some companies restricted the use of HDL to the definitions of combinational logic, which were then instantiated into the graphical schematic diagrams as modules. It took a paradigm shift, primarily caused by the acceptance of synthesis tools, to finally accept HDL as a design entry and verification vehicle. By then, government contractors started to impose HDLs as deliverables. In the 90's it became evident to designers, except "old-timers" who still view schematics as a better methodology, that the days of schematic capture being the primary method of design entry was passé.

Static verification tools also started to emerge around that time to help in the static (i.e., no simulation) analysis of the designs for compatibility to design rules (see Chapter 7 for an elaboration on this topic). As design densities increased, the use of more intelligent testbenches became the in-vogue paradigm. Self-checking testbenches with directed testing was then followed by pseudo-random, constrained-random stimulus generation and finally transaction-based. Implementation of self-checking testbenches, along with the introduction of better tools, such as coverage and better and faster simulators propelled the popularity of simulation. However, as design became even more complex, the verification effort became more difficult with the current HDLs, particularly with Verilog because the language is very limited in data typing, and cannot execute reentrant functions and tasks.[5]

---

[4] Component instantiation was performed with the pasting of transparent self-adhesive pre-printed schematics of the devices (e.g., flip-flop, multiplexer) ... that WAS technology!

[5] Some of those limitations were removed in Verilog 2001. SystemVerilog 3.1 is Accellera's Extensions to Verilog ®, and represents a set of extensions to the IEEE 1364-2001 Verilog Hardware Description Language to aid in the creation and verification of abstract architectural level models

To overcome these deficiencies, hardware verification languages (HVLs) were introduced. These HVLs were tailored for simulation and integration with HDLs. HVLs introduced the concept of functional coverage and helped in tracking the verification progress. One class of HVLs (aka Assertion-languages) also introduced the concept of assertion-based verification (ABV), where design properties could easily be specified and verified. Assertion languages were also integrated with static (a.k.a. formal) verification tools to formally and mathematically prove or disprove the assertions. Formal tools exhaustively explore all possible sequences of input values in an attempt to prove that the assertion is valid. A counterexample for the failing assertion can be demonstrated in simulation from a sequence generated by a formal tool. Essentially, the formal tool creates a mathematical model of the design and the requirements, which includes a model of transitions for the system. A set of vectors could be generated of this transition model for a failing assertion.

Assertion — An assertion is a statement that a given property is required to be true, and a directive to verification tools to verify that the property does hold. Assertions can capture requirements, both for human-readable documentation and for verification via simulation or formal verification tools. Placing assertions on the interfaces can help integrate a block into a design, because they quickly identify when the protocols, or sequence of events are incorrect.

Figure 1.1-1 Popularity of Verification Methods for Digital Designs

1.2 SYSTEMVERILOG ASSERTIONS IN VERIFICATION STRATEGY

SystemVerilog has greatly extended the capabilities of Verilog'01 to create a unified language to meet both design and verification requirements of today's complex designs. Figure 1.1 is a view of SystemVerilog capabilities, demonstrating the strong and advanced features of modern languages that also support hardware definition and verification.

Among these various features is "assertions". SystemVerilog Assertions are part of SystemVerilog. They enable the definition and verification of design properties using a concise, easier to read, and executable syntax. SystemVerilog Assertions with temporal sequences can express in just a few lines a complex FSM, whereas equivalent HDL code requires a much more complex model that is hard to follow. Because assertions are executable, they can be verified through simulation and/or static verification.

1.2.1 Are Assertions Independent from SystemVerilog Structures?

Assertions define properties that the design must meet. Many properties can be expressed strictly from variables available in the design. For example, in the typical handshake example with a *request* and an *acknowledge,* the following property and assertion can be expressed:

```
property pReqAck;
  // if request, then acknowledge should be asserted within 1 to 4 cycles.
  @ (posedge clk) request |-> ##[1:4] acknowledge;
endproperty : pReqAck
apReqAck : assert property (pReqAck);
```

While properties are extremely powerful to capture control-centric design requirements, real designs include a mix of control and data elements. Hence a closer interaction between a higher level procedural language (with abstract data types, queues, sparse arrays etc.) and a declarative, formal language is needed to verify the data aspects of the design.[6] The unified SystemVerilog standard provides exactly that. For example, queues and associative arrays can be used with SystemVerilog code as scoreboards, and they can also be used in the context of properties (though with certain restrictions). This concept is demonstrated in Chapter 6 for the definition of a synchronous FIFO requirements It is also demonstrated in Chapter 9 for the verification of data integrity in a memory module. Note that SystemVerilog Assertions allows for the definition of local variables, which can be used in many properties to verify data elements. This capability is demonstrated throughout this book. However, the use of above mentioned structures allow for the definition of more complex properties.

[6] For examples in the use of SystemVerilog procedural language support with properties see models in section 9.28 address integrity, section 9.15 data integrity in memory, and section 9.16 data integrity in interface data transfer.

Figure 1.1. SystemVerilog : Unified Language[7]

1.2.2 Are Assertions Useful for the Defintion and Verification of Designs?

SystemVerilog Assertions is an integral component of SystemVerilog to support an Assertion-Based Verification methodology. Assertion-Based Verification is a verification technique that involves instrumenting the design with assertions to provide added visibility and white-box testing into its internal state.[8] White-box functional verification using assertions enables the detection of bugs much earlier, at or near their source, rather than at outputs.

Assertions document the combinational and temporal (i.e., over cycles) properties of the design that must hold true (or never occur) for the design to be correct. They may be embedded into the design source text, or provided in separate files. Assertions can be checked either dynamically during simulation and/or statically through formal methods.

Assertions express functional design intent and can be used to express assumed input behavior, expected output behavior, interesting scenarios (for functional coverage), and forbidden behavior. For example, if a processor has a *read* and *write* interface signal, a designer might assume that "the *read* signal and the *write* signal should never be both active at the same time." With assertion-based verification (ABV), that assumption can be captured and verified. This represents an example of an interface assumption. Interface assertions are used to check the interface protocol between blocks. Other types of assertions include application assertions and structural assertions, as shown in Figure 1.5.

---

7 "SystemVerilog for VHDL Users", Tom Fitzpatrick, 2004.

8 Though in principle Assertions can be used to capture end-to-end behavior of the system, typical usage as of today is limited to lower level implementation details.

The use of ABV methodology along with SystemVerilog Assertions as the specification language provides the benefits outlined below.

Figure 1.5 Types of Assertions in a Design

1.2.2.1 Captures Designer Intent

Often, very basic assumptions made by the designer are not explicitly captured as part of the HDL model. An example of such a case that caused a failure in the *Wide-Field Infrared Explorer* spacecraft is the assumption that at start-up time, all digitals and clocks are stable and that the logic that drives the instrument pyrotechnic electronics box is in a safe mode.[9] However, the start-up time of the crystal clock oscillator was not taken into consideration, leaving the circuit in a non-deterministic state for a time sufficient for pyrotechnic actuation. Likewise, the startup characteristics of the *Actel A1020* FPGA were not considered.[10] These devices are not guaranteed to follow their "truth table" until an internal charge pump "starts" the part. These uncontrolled outputs of the pyrotechnic control logic were not blocked from the pyrotechnic devices' driver circuitry. If an Assertion-Based Verification methodology was used, the

---

[9] The Wide-Field Infrared Explorer Mission objective was to conduct a deep infrared, extra galactic science survey. The Wide-Field Infrared Explorer was launched on March 4, 1999, and was observed to be initially tumbling at a rate higher than expected during its initial pass over the Poker Flat, Alaska, ground station. After significant recovery efforts, WIRE was declared a loss on March 8, 1999. In this spacecraft, the telescope instrument cover was ejected earlier than planned and at approximately the time the WIRE pyro-electronics box was first powered on. The instrument's solid hydrogen cryogen supply started to sublimate faster than planned, causing the spacecraft to spin up to a rate of sixty revolutions per minute over the twelve hours following the opening of the secondary cryogen vent. Without any solid hydrogen remaining, the instrument could not perform its observations.
http://klabs.org/richcontent/Reports/wiremishap.htm
WIRE Mishap Investigation Board Report, June 8, 1999

[10] http://klabs.org/richcontent/Papers/Radecs95_swi.pdf
Total Dose Responses of Actel 1020B and 1280A Field Programmable Gate Arrays (FPGAs)

properties that the pyrotechnic control logic should never be in the "fire" state during start-up could have been specified. As a result, additional start-up lockout protection or independent provisions to prevent the FPGA start-up operation from propagating to the firing circuits could have been implemented. Dynamic verification with emulation of a clock with slow startup and simulation of FPGA start-up behavior could have flagged the potential catastrophic error. Below is a property that would have flagged the catastrophic error:

```
sequence qPyroFire; // sequence of a Pyro Fire
 Fire_PyroX_CMD=FIRE && // Command to fire
 power_stable && // Power is stable
 fpga_charged && // The FPGA is charged up, ready to operate correctly
 reset_cycle== DONE && // the reset cycle is completed
 armed == OK // The pyros are armed
 ##1; // delay of 1 cycle
endsequence : qPyroFire

property pPyroRelayActive;
   PyroX_relay==ACTIVATED |-> // If the relay is activated, then
        qPyroFire.ended); // the sequence to fire the pyro must have ended.
endproperty : pPyroRelayActive
```

1.2.2.2 Allows Protocols to be Defined and Verified

Temporal sequences can be used to specify and verify interface and bus protocols. Such assertions define not only the values of signals, but also the relationship between signals over time. Verification can be achieved through simulation and/or formal verification.[11]

1.2.2.3 Reduces the Time to Market

An Assertion-Based Verification methodology reduces the time to market for the following reasons:

- ABV provides early detection of errors.
- ABV facilitates debugging of failing tests as assertions can behave as monitors close to the source of error.
- When combined with formal verification, ABV can prove that the design meets the functional requirements.

This reduction in time to market is demonstrated in Figure 1.2.2.3, and was proven on many projects.

1.2.2.4 Greatly Simplifies the Verification of Reusable IP

If an IP block contains assertions that express its input constraints and interface protocols, these assertions can check whether the block is used properly in a much larger system. Assertions added to IP blocks help to ensure proper integration of the IP. Typically the verification of SoCs, built using these IP blocks, involves the verification of interfaces rather than the individual block's functionality. Hence an Assertion-Based Verification methodology fits the best, as compared to traditional verification methodologies. The common experience with IP-based designs has revealed that the majority of bugs adhere to the improper usage/configuration of an IP, rather than the functionality of the IP itself.

---

[11] Chapter 6 and Appendix A provide examples for the definition and verification of protocols, such as a handshake protocol.

Figure 1.2.2.3 Design Errors versus Time to Market Using ABV and FV[12]

1.2.2.5 Facilitates Functional Coverage Metrics

Functional coverage is a technique for measuring how much of the functionality was exercised by verification. *"Functional coverage is the ratio of exercised behaviors over the total number of specified behaviors. Trigger events and data ranges specify functional coverage points. When the trigger events occur, the testbench tool records in a database the data ranges occupied by the relevant signals".*[13] Functional coverage metrics can be specified in SystemVerilog Assertions though the definition of user-defined coverage points using the **cover** directive (see section 3.10.3). In addition, simulation vendors typically provide the collection of statistics on the SystemVerilog Assertions verification directives, thus providing feedback about the thoroughness of the simulation.[14] For example, assertions that monitor the logic controlling a FIFO queue could be used to capture if the FIFO was ever full, and whether extreme test corner cases were exercised. Functional coverage is achieved by monitoring during verification whether the assertions were exercised. If the assertion wasn't exercised, that indicates that the logic handling the queue overflow was not verified. Note that functional coverage is different from code coverage. Code coverage is a technique for measuring how much of the source code of a

[12] By permission from *An Introduction To Property Checkers For Functional Verification,* Andrew Jones and Jeremy Sonander, Saros Technology, http://www.averant.com

[13] @PhdThesis{dvc:99,"Van Campenhout, D.","Functional design verification for microprocessors by error modeling", University of Michigan, 1996

[14] EDA vendors may provide assertion coverage metrics including the number of times an assertion finished and failed. .

computer program or hardware description has been executed by a test run or simulation.[15] However, functional coverage addresses the functionality of the design. In essence, functional coverage operates at a higher-level coverage than code coverage.

1.2.2.6 Generates Counterexamples to Demonstrate Violation of Properties

Formal verification tools use SystemVerilog Assertions constraints (e.g., **assume**) and design properties to determine if the design meets those properties. If it does not, then those tools will generate counterexamples to demonstrate the conditions under which the rules were violated. This greatly facilitates the verification of designs.

1.2.3 Can/should entire functional verification task be performed using SystemVerilog Assertions?

A direct answer will be NO. However, as previously mentioned, SystemVerilog Assertions along with potential SystemVerilog constructs and supporting code will play a significant role in the definition of design requirements and in the verification process of designs. Use of SystemVerilog Assertions is strongly encouraged because the Assertion-Based Verification was demonstrated to be very beneficial. SystemVerilog Assertions does not preclude the use of other proven techniques, such as accelerators, or the use of other languages, such as SystemC. However, it can supplement those techniques during the design definition and verification process.

1.2.4 Is SystemVerilog Assertions Solely Restricted to Applications that Use SystemVerilog?

Absolutely not! SystemVerilog Assertions can be used with designs written in any HDL, VHDL or Verilog. VHDL can be used in mixed mode simulation along with SystemVerilog Assertions. Figure 1.1.4 demonstrates how mixed mode simulation can be established. Chapter 5 expands these concepts by using real examples.

Why would such an approach be beneficial? Because SystemVerilog Assertions is part of SystemVerilog, and a SystemVerilog verification module that can make use of both the powerful constructs of SystemVerilog and the flexibility and ease of use of the assertions.

1.2.4.1 VHDL Model and Testbench with SystemVerilog Assertions Module

In mixed mode simulation, a SystemVerilog module with SystemVerilog Assertions can be used to verify properties of a VHDL model. The VHDL model interfaces can be interconnected to the ports of the SystemVerilog Assertions verification module. Signals internal to the VHDL model can be exported into the SystemVerilog Assertions verification module through a variety of techniques including:

1. Copying VHDL critical signals to signals defined in a package, and routing the signals in the package to the SystemVerilog Assertions verification module.
2. Routing internal signals to output ports. Pragmas can be used to exclude synthesis from processing the routing of the critical internal signals to the output ports.
3. Use of vendor supplied PLIs to export internal signals into other modules.

---

[15] *Verification Methodology Manual for Code Coverage in HDL Designs*, Michael Stuart and David Dempster, isbn 0-9538-4820-5 Hampshire UK, www.teamwork.demon.co.uk

Figure 1.1.4 VHDL Verification with SystemVerilog Assertions

1.2.4.2 VHDL Model Embedded in SystemVerilog testbench with SVA Module

Again, in mixed mode simulation, a SystemVerilog module with SystemVerilog Assertions can be used to verify properties of a VHDL model. The VHDL model interfaces can be interconnected to the ports of the SystemVerilog Assertions verification module. Signals internal to the VHDL model can be accessed used the methods described above. Chapter 5, Section 5.14 addresses the topic of verifying VHDL modules with SystemVerilog Assertions

1.3 ACCELLERA'S SYSTEMVERILOG ASSERTIONS GOALS

Ensuring that a design's implementation satisfies its specification is the foundation of hardware verification. Key to the design and verification process is the specification. Yet historically, the process of specification has consisted of creating a natural language description of a set of design requirements. This form of specification is both ambiguous, and in many cases, unverifiable due to the lack of a standard machine-executable representation. Furthermore, ensuring that all functional aspects of the specification have been adequately verified (that is, covered) is problematic. [16]

The Accellera SystemVerilog Assertions aspect of SystemVerilog was developed to address these shortcomings. SystemVerilog Assertions has the same look and feel as the rest of SystemVerilog, thus making it easier for an engineer to learn and use SystemVerilog design and verification capabilities. The assertions are integrated in SystemVerilog with very similar syntax. No longer does an engineer need to learn (1) a design language, (2) an HVL (hardware verification language), and (3) an assertion language. They are all part of the same language. SystemVerilog Assertions give the design architect a standard means of specifying design properties using a concise syntax with clearly defined formal semantics. Similarly, it enables the RTL designer to capture design intent in a verifiable form, while enabling the verification engineer to validate that the implementation satisfies its specification through dynamic (that is, simulation) and formal verification means. Furthermore, it provides a means to measure the quality of the verification process through the creation of functional coverage models built on formally specified properties. It also provides a standard means for hardware designers and verification engineers to rigorously document the design specifications with a machine-executable language. SystemVerilog Assertions were specifically developed to fulfill the following general hardware functional specification requirements:

[16] Extracted from SystemVerilog LRM 3.1a

- Easy to learn, write, and read.
- Concise syntax.
- Rigorously well-defined formal semantics.
- Expressive power, permitting the specification for a large class of real world design properties.
- Use of known and efficient underlying algorithms in simulation, as well as formal verification.
- Integrated into SystemVerilog environment, including scheduling of assertions, system tasks, functions, and data structures.

1.4 SYSTEMVERILOG ASSERTIONS LANGUAGE

SystemVerilog Assertions formally define the syntax and grammar of an assertion language that can define design properties and constraints, including temporal (i.e., spread over several cycles) characteristics. There are several advantages in using SystemVerilog Assertions:

1. **Standard**. Accellera approved SystemVerilog 3.1a and handed the document to the IEEE's corporate standards group for IEEE standardization. The standard definition of SystemVerilog Assertions semantics is key for guiding different vendors in providing consistent support for this assertions language.

2. **Documentation**. Using short (concise) notations, SystemVerilog Assertions describe event sequences, temporal properties, and requirement implications. SystemVerilog Assertions unambiguously document the characteristics of designs, including assumptions on protocols and interfaces. SystemVerilog Assertions provide the capability to abstract higher-level verification constructs when it is combined with SystemVerilog data structures, tasks and methods, such as those supporting queues and associative arrays. Industry has adopted two methods to define requirement properties and assertions: the embedded method and the external verification program file method with binding to the design under verification (DUV). The embedded method defines the SystemVerilog properties inline within the descriptions (i.e., HDL descriptions) of designs (i.e., a module, an interface, a program, a clocking block, a package), thus directly linking the requirements to the implementation. The external file method represents separate files where assertions are defined, thus physically separating the property specifications from the design modeling. They are often used to either avoid touching the synthesizable RTL designs, or to have separate verification engineers define properties of the design and/or interfaces. External files help ensure the reuse of properties at multiple abstraction levels (e.g., behavioral, RTL, and gate-level (assuming the signals are preserved through synthesis). External files are also used in the verification of Intellectual Property Designs (IPs), commercial or home-brewed. Those files can be SystemVerilog modules or SystemVerilog interfaces bound to the design with the **bind** directive.[17] These definitions not only help reaffirm the properties of the design, but also provide a basis for the review team to critically review the RTL design against requirements.

3. **White-box verification through simulation**. Design modules supplemented with SystemVerilog Assertions facilitate the integration and reuse of those modules into higher-levels of hierarchies. During simulation or formal verification, the SystemVerilog Assertions quickly identify when protocols, or sequences of events are incorrect.

[17] See Chapter 3.

Because the assertions are local to the module, observability is increased, and it is maintained throughout the lifetime of the module, even if the module undergoes modifications. Assertions act as monitors close to the source of error.

4. **White-box verification through formal verification.** Design properties can be proven mathematically, and counterexamples that can disprove the properties can be automatically generated. Formal verification tools supporting SystemVerilog Assertions are becoming available.

5. **Black-box verification.** SystemVerilog assertions can describe the interface and behavioral requirements of a design at the black-box level. As a result of the temporal characteristics of the language, access to SystemVerilog data structures and system tasks/methods, those requirements (including data-integrity in some cases) can be specified with cycle timing tolerances (e.g., a range of cycles) to allow for design optimizations.

6. **Testbench support.** SystemVerilog Assertions were not designed to replace testbenches.[18] However, SystemVerilog Assertions can be used with traditional testbench approaches. For example, a scoreboard can be modeled using the Verification sub-set of SystemVerilog where transaction information and expected results can be written into the scoreboard. Then a SystemVerilog Assertions statement can be written to determine if the DUV is matching the expected results.

7. **System engineering support.** SystemVerilog Assertions can allow system software engineers to get involved in the verification process. For example, one can define in SystemVerilog Assertions the "system level" triggering conditions, and the expected effects to be produced at some time in the future within a latency time. For example:

```
property pFindTarget;
  @(posedge clk) disable iff (!reset_n)
  (HotSceneLoaded |=> ##[*20:50]
            duv_target_map_array==find_target_function(image_map));
endproperty : pFindTarget
```

That property states that if the scene data is ready to be processed by hardware, then within 20 to 50 cycles, the *duv_target_map_array* (DUV hardware memory) must be equal to the results of the image map (*image_map*) processed through the processing algorithm (*find_target_function*).

[18] SystemVerilog Assertions sequences/properties could be used to specify stimulus generation. This is tool dependent.

8. **Helps in identifying when verification is not done.** Identifying when verification is done is a difficult task. However, SystemVerilog Assertions can provide a greater level of assurance about the stability of a design. In simulation, functional coverage is often used to help achieve the goals of identifying when verification is done. A property or sequence should map to a testable functional requirement of the design. It should be possible to define a property that corresponds to each testable 'shall' in a functional requirements specification. Functional coverage identifies whether testing has covered those testable requirements.

Examples of designs that can benefit from the use of SystemVerilog Assertions include:

- **Controllers**. These controllers include memories, FIFOs, DMAs, bus interfaces (e.g., PCI bus, AHBA ARM bus). SystemVerilog Assertions can be used for the verification of protocols and illegal operations. This is a white-box / black-box verification process.
- **Finite State Machines** (FSMs). Properties can be defined to verify that no illegal state entry is ever entered, or that a desired sequence is accomplished, or that the correct encoding scheme (e.g., one-hot) is applied. This is a white-box verification process.
- **Black-Box.** Properties can be defined to verify that the protocols, interface constraints assumptions, and operations of a design at the black-box level are not violated. SystemVerilog assertions can be inserted at the black-box testbench level.
- **System-level requirements.** Properties can define latency requirements, processing algorithms, and mission critical requirements.

2 OVERVIEW OF PROPERTIES AND ASSERTIONS

This chapter serves as an introduction to SystemVerilog Assertions (SVA) concepts with emphasis on properties and assertions. It prepares the readers for Chapter 3 that delves into the details and applications of properties and assertions.

Properties consist of a set of Boolean expressions and operators to characterize the conditions that can occur in one or more clock cycles. SystemVerilog Assertions expressions are based on a concept similar to PSL *Sequential Extended Regular Expressions* (SEREs) to provide a more powerful method of expressing sequences.[1]

2.1 DEFINITIONS[2]

2.1.1 Properties

A property is a collection of logical and temporal relationships that in aggregate represent a set of behavior, rules, or characteristics about a design. In logic design, properties are generally classified as "functional" and "structural". Users using an assertion language such as SystemVerilog Assertions capture functional properties. Structural properties can be automatically extracted using tools and are different from functional properties. Structural properties include information about the structure of the RTL such as:

1. Deadlock states (i.e., two FSMs waiting for each other)
2. Terminal states (i.e., a state that cannot be exited)
3. Unreachable states (i.e., a state that cannot be entered)

---

[1] PSL and SystemVerilog with Assertions are property specification languages approved by Accellera and are currently in the IEEE for standardization.

[2] Appendix B provides a definition of common terms used in Assertion-Based Verification methodology.

4. Clock crossing domain instability (i.e., metastability issues)
5. Full case / parallel case issues
6. Linting (language guidelines violations)

SystemVerilog Assertions can define properties as characteristics that a hardware design should obey. These properties express temporal (i.e., behavior over a period of time) relationships among Boolean expressions, sequential expressions, and subordinate properties. Examples of properties include system level characteristics such as processing algorithms (defined in functions or tasks), system processing latencies, design and interface specifications, FSM requirements, and RTL design characteristics. Chapter 4 addresses the use of SystemVerilog Assertions in the design process.

2.1.2 Sequences

SystemVerilog Assertions provide a means to manage complex assertions spread over a long period of time using a hierarchical "divide-and-conquer" methodology. Boolean expressions represent the primitive building blocks. A set of Boolean expressions can be combined using powerful operators to build sequences. Sequences can describe common scenarios such as:

- End of reset
- Abort condition for higher level properties
- Common starting point for multiple properties (such as "write to a cached line" that should be followed by multiple parallel transactions).

2.1.3 Antecedent / Consequent / Thread

In SystemVerilog Assertions, a thread is a related sequence of events that may last for one or more cycle to represent a design property. For example, a bus request that is acknowledged within 1 to 5 cycles is a thread. A new thread may be started at every cycle, and every thread is independent from other threads that may have started at earlier cycles. Multiple threads may be concurrently active, and each of those threads may successfully terminate or may fail by the occurrence of a common event.[3] Thus, for the above example, every bus request starts a new thread that needs to be verified.

SystemVerilog Assertions checks every asserted / assumed / or covered property at every clock cycle. If the property consists of a multi-cycle temporal expression, then each start of the property is considered a thread. Many applications have a triggering event that causes the thread to start, and if the triggering event does not occur, then there is no need to further evaluate the thread (i.e., the rest of the property for that cycle), which is then considered vacuously TRUE.[4] For example, an active *bus_request* would trigger an *acknowledge* some time later, and then a set of data transfers occur until a *done* event is received to identify the completion of the handshake. However, if *bus_request* is inactive (e.g., FALSE), then there is no need to further evaluate the rest of the thread that failed to start in that cycle, and the assertion is considered to be vacuously TRUE in that cycle. This concept of thread evaluations only when needed may also be used in the optimization of the performance of the simulator. To achieve this cause-effect of thread evaluations, SystemVerilog introduces the concept of antecedent and consequent with two implication operators "|->" (overlapped, meaning that the resulting sequence starts in the same cycle as the causing sequence), and "|=>" (non-overlapped, meaning that the resulting sequence starts in the next cycle as the causing sequence). For example,

---

[3] This is explained in Chapter 3, section 3.4.3.

[4] See appendix B for definition of vacuity.

property p_handshake; [5]
@ (**posedge** clk)

request |=> acknowledge ##1 data_enable ##1 done;
endproperty : p_handshake

ap_handshake **: assert property** (p_handshake);

Property *p_handshake* states that if the antecedent sequence *request* (one-cycle sequence in this case) is TRUE then the consequent must also be TRUE, otherwise the property fails. The thread starts when the antecedent is TRUE, and completes when either the consequent succeeds or fails. A new thread may be started at every evaluation cycle. If the antecedent sequence is FALSE, then the property succeeds vacuously in that cycle. Figure 2.1.3-1 demonstrates two active threads for this property.

Figure 2.1.3-1 Two active threads for the Handshake Property
Thread T1 Completes at Time 250 ns,
Thread T2 Fails at Time 250 ns (data_enable==1'b0, expected a 1'b1)

[5] Chapter 8, section 8.1 addresses the naming convention used in this book.

2.1.4 Specification and Verification

A specification is a set of executable properties. From a SystemVerilog Assertions viewpoint, verification is the confirmation that, for a given design and a given set of constraints, a property that is required to hold in that design actually does hold under those constraints

Assertion-Based Verification

Assertion-Based Verification (ABV) is a verification and documentation methodology. It instruments system requirements, interfaces, design characteristics, and verification environments with assertions. SystemVerilog Assertions is an assertion language, tightly coupled to SystemVerilog for the definition, declaration, and verification of properties. Definition represents the syntax by which the properties are expressed. Declaration specifies where and how the properties are declared so that the verification tool can understand and exercise them. Verification is the process used by the verification tool, such as simulator or static verifier, to insure that the design complies with the declared properties. SystemVerilog Assertions provides the following advantages in the design process:

- Documentation
  - Requirements prior to any RTL code
  - Facilitates reviews
  - Unambiguous definition of interface and design properties
  - Documents design decisions
  - Documents assumptions
- Design
  - Simplifies RTL design because of better understanding of requirements
  - Properties are carried throughout the lifetime of the design
- Verification achieved through:
  - Simulation. It simplifies the testbench design because verification is performed through the SystemVerilog properties at all levels of the design hierarchy. Errors are identified by failing properties. SystemVerilog Assertions definitions guide the directed tests and functional coverage.
  - Formal verification. Exhaustive analysis of the design for adherence to the properties.

2.1.5 Assertion / Assumption / Verification Directive

An assertion (with the **assert** directive) is a statement that a given property is required to hold and a directive to verification tools to verify that it does hold. An assumption (with the **assume** directive) is a statement that the design is constrained by the given property and a directive to verification tools to consider only paths on which the given property holds. A **cover** verification directive instructs the verification tool to report on the coverage (i.e., PASS / FAIL / NOT COMPLETED) of properties and sequences.

2.1.6 Constraint

A constraint is a condition, usually on the input signals, that limits the set of behaviors to be considered. A constraint may represent real requirements such as clock cycles and resets, and legal values for inputs. Formal verification tools use constraints to limit the search space. Random stimulus generators may also use constraints to generate intelligent random vectors. Chapter 5 addresses formal verification and elaborates on the issue of constraints. An example of constraints is shown below:

```
/* these localparams define the AHB Burst Mode as specified by HBurst */
localparam SINGLE = 3'b000;
...
localparam WRAP4 = 3'b010;
localparam INCR4 = 3'b011;

// ******************* MEM_READS*******************
//| # Pattern name # MEM_READS
//|
//| # Problem # The architecture with the memory device provides the first data output in
//| the third cycle after the master asserts the address during the NONSEQ htrans cycle.
//| In addition, for burst, the memory device allows for the advance mode to increment the
//| address in a WRAP4 burst mode. Only HBurst of SINGLE and WRAP4 will be
//| accepted because of limitation of the memory device.
//|
//| # Motivation # Efficiency. Restriction for WRAP4 is set by memory device
//| selection. Other motivation is design simplicity of slave controller.
//| # Context # Reads in SINGLE or WRAP4 only.
//|
//| # Solution #
//| Reads in SINGLE or WRAP4 ONLY
// Constraint definition:
property pHBURST4IDT;
  @ (posedge hclk) (hburst==SINGLE || hburst==WRAP4);
endproperty : pHBURST4IDT

mpHBURST4IDT : assume property (pHBURST4IDT); // hburst constraint
mHsize : assume property (hsize== $random % 8); // hsize constraint
```

Patterns, explained in *Assertion-Based Verification,* provide good documentation and rationales.

NOTE: SystemVerilog has an explicit "**constraint**" construct that is part of the Verification language. In this book the terms **constraint** and **assume** are used interchangeably to refer to the **assume** construct of SystemVerilog Assertions.

2.2 PROPERTY

A property defines a behavior of the design. A property can be used for verification as an assumption, a checker, or a coverage specification. To use the behavior of a property for verification, the property must be instantiated in an **assert**, **assume**, or **cover** verification directive. A property declaration by itself does not produce any result. A property can be declared and instantiated in the following SystemVerilog declarations:

Module	Interface	Program
Clocking block	Package	Compilation-unit scope

2.2.1 Named Properties

A property is named during its declaration. The syntax of a property declaration is shown in Figure 2.2.1-1. A property can also be declared with a list of formal arguments that can be referenced within the property. A named property is instantiated with verification directives, such as the **assert** directive. A property may also be used in other properties. A property can access all variables and signals visible to its context, or made available with the **$unit** or the full object paths. A property is activated when it is instantiated with the **assert, assume, cover,** or **expect** statements. When a property with formal arguments is instantiated, the actual arguments are substituted for the formal arguments, in a manner similar to a function call. An example of a simple named property that is instantiated with an **assert** directive is shown in Figure 2.2.1-2.

```
property_declaration ::=
   property property_identifier [ ( [ list_of_formals ] ) ] ;
     { assertion_variable_declaration }
     property_spec ;
   endproperty [ : property_identifier ]

list_of_formals ::= formal_list_item { , formal_list_item }

formal_list_item ::=  formal_identifier [ = actual_arg_expr ]

formal_identifier ::= identifier

assertion_variable_declaration ::=  data_type list_of_variable_identifiers ;
```

Figure 2.2.1-1 Syntax for Property Declaration

```
// Declaration. If no reset, then if req, then ack should be active in the next cycle
// "|=>" indicates that if the antecedent is TRUE, then in the next cycle the consequent must hold.
// "|->" indicates that if the antecedent is TRUE, then in the same cycle the consequent must hold.
  property pReqAck;
        @( posedge clk) disable iff (!reset_n) req |=> ack;
  endproperty : pReqAck

// Instantiation
   apReqAck : assert property (pReqAck) else
              $display("%m time= %0t , Property pReqAck failed to hold:
                past(req)= %b, ack=%b \n",  $time,  $past(req),  $sampled(ack) );[6]
```

Figure 2.2.1-2 Simple Named Property Instantiated with the assert Directive

[6] Se section 3.6.1.1.1 on the need of **$sampled** in the action block.

Property *pReqAck* states that at every positive edge of the clock *clk*, a *req* is followed by an *ack* signal on the next clock edge, provided *reset_n*[7] is not active (i.e., *reset_n* == 1'b1). The **assert** directive instantiates the named property *pReqAck*, and should this property fail then the simulator should display the time of failure, the value of *req* in the previous cycle, and the value of the sampled *ack* in the current cycle.

If a named property has formal arguments, then actual arguments must be defined during instantiation (unless there are default values defined in their declarations), as shown in Figure 2.2.1-3.

```
// Declaration:
    property pReqAck (req, ack);
      @( posedge clk) disable iff (!reset_n)
          req |=> ack;
  endproperty : pReqAck

// Instantiation
  apReqAck : assert property (pReqAck(request, acknowledge))
    else $display("time= %0t, Property pReqAck failed to hold: past (req)= %b, ack=%b \n",
          $time, $past(request), $sampled (acknowledge) );
// If fail, then display assertion instance name, time of failure, previous value of request,
// current sampled value of acknowledge.
```

Figure 2.2.1-3 Property with Formal Arguments

2.3 ASSERTION

An assertion specifies a behavior of the system. Assertions are primarily used to validate the behavior of a design. In addition, assertions can be used to provide functional coverage and generate input stimulus for validation. There are two kinds of assertions: immediate and concurrent.

1. **Immediate assertions** follow simulation event semantics for their execution and are executed like a statement in a procedural block. Immediate assertions are primarily intended to be used with simulation, and not for formal verification. The expression is non-temporal and is interpreted the same way as an expression in the condition of a procedural **if** statement.

2. **Concurrent assertions** are based on clock semantics and use sampled values of variables. Concurrent assertions describe behavior that spans over time. Unlike immediate assertions, the evaluation model is based on a clock such that a concurrent assertion is evaluated only at the occurrence of a clock tick.

The syntax for assertion statements is shown in Figure 2.3-1.

[7] "_n" suffix is a notation to indication an active low, or negative, signal.

```
assertion_statement ::=  concurrent_assertion_statement
                       | immediate_assert_statement
procedural_assertion_statement ::=  concurrent_assertion_statement
                                  | immediate_assert_statement
immediate_assert_statement ::=  assert ( expression ) action_block

action_block ::=  statement_or_null  | [ statement ] else statement

concurrent_assertion_item ::= [ block_identifier : ] concurrent_assertion_statement

concurrent_assertion_statement ::=  assert_property_statement
                                  | assume_property_statement
                                  | cover_property_statement

assert_property_statement::=  assert property ( property_spec ) action_block

assume_property_statement::=  assume property ( property_spec ) ;

cover_property_statement::=  cover property ( property_spec ) statement_or_null
```

Figure 2.3-1 Syntax for assertions

The *action_block* specifieswhatactions are taken upon success or failure of the assertion. The statement associated with the success of the assert statement is the first statement. It is called the *pass statement* and is executed if the expression evaluates to TRUE. The pass statement can, for example, record the number of successes for a coverage log, but can be omitted altogether. If the pass statement is omitted, then no action is taken when the assert expression is TRUE. The statement associated with **else** is called a *fail statement* and is executed if the expression evaluates to FALSE. The **else** statement can also be omitted, in which case a pre-defined error message shall be displayed by the tool. The action block is executed immediately after the evaluation of the assert expression. Immediate assertions follow the simulation event semantics for their execution and are executed like a statement in a procedural block (i.e., in the active / nonblocking assignment (NBA) region). In concurrent assertions, the action block is executed immediately after the evaluation of the **assert** expression in the reactive region of the scheduling slot.[8] This has significance in the display of variables in the current time slot since those variables may have changed values in the active region, prior to the reactive region. The use of **$sampled** function provides access to the values of sampled variables used in the evaluation of the properties in concurrent assertions.

2.3.1 Immediate assertions

The immediate assertion statement is a test of an expression performed when the statement is executed in the procedural code. The expression is non-temporal and is interpreted the same way as an expression in the condition of a procedural **if** statement. That is, if the expression evaluates to X, Z or 0, then it is interpreted as being FALSE and the assertion fails. Otherwise, the expression is interpreted as TRUE and the assertion passes. Typically, the conditions that cause the assertion to be tested are within the SystemVerilog block. The immediate **assert** statement

[8] See section 3.8 on SystemVerilog scheduling semantics for assertions.

can be specified anywhere a procedural statement is specified, including **initial** and **always** blocks, tasks, and functions. Figure 2.3.1 demonstrates an example of an immediate assertion.

```
// Immediate Assertions
always @ (posedge clk) begin :Fifo_Counter
 PushPopTest: if (push && !pop) begin
  a_fifomax: assert (fifo_count < MAXCOUNT) else
    begin : Push_NoPop
      $display ("%m @ time %0t, push && ! pop && maxcount = %h", $time, fifo_count);
      // fifo_count <= fifo_count + 1'b1; ☹
    end : Push_NoPop
   fifo_count <= fifo_count + 1'b1; ☺
 end
 else if (!push && pop) begin
  a_fifomin : assert (fifo_count > 0) else
      begin :Pop_NoPush
        $display ("%m @ time %0t, !push && pop && maxcount = %h", $time, fifo_count);
        // fifo_count <= fifo_count - 1'b1; ☹
      end : Pop_NoPush
  fifo_count <= fifo_count - 1'b1; ☺
 end
end
```

Immediate assertion

Executing RTL code from the pass or fail block is not recommended as synthesis will ignore the **assert** statement completely. This will lead to mismatches between the RTL simulation and the gates produced by synthesis.

Figure 2.3.1 Immediate Assertion Example

Immediate assertions are useful when added along with the RTL code. Typically, wherever a designer would have written a comment inside a procedural block, an immediate assertion can capture that comment in an executable format. Without such immediate assertions, one needs to duplicate the logic leading to that particular point in the procedural code, in a separate concurrent assertion – a process that is both tedious and error prone. For example, consider:

```
always @(posedge clk) begin  // always
 if (!reset_n)
 // .. code
 else begin  // else
  if (sop && pkt_data = 0x8000)
  begin  // if sop
   if (CFG_NO_JUMBO_PKTS)
     a_plklen : assert (pkt_len < 9576) else
      $error ("detected a jumbo pkt while the current configuration  prevents that");
  end // if sop
 end // else
end // always
```

Immediate assertion

If not for the immediate assertions, one would need to capture the complex logic in a concurrent assertion.

2.3.2 Concurrent Assertion

A concurrent assertion statement can be specified in:

- An **always** block or **initial** block as a statement, wherever these blocks can appear. This is similar to the immediate assertion, except that temporal characteristics can be described within the properties.
- A **module**, as a concurrent block (i.e., running concurrently with other concurrent statements).
- An **interface**, as a concurrent block (i.e., running concurrently within the interface).
- A **program** block, as a concurrent block (i.e., running concurrently with other concurrent statements).

The syntax of a concurrent assertion includes an assertion directive followed by a property specification, and an optional action block, as shown below:

```
concurrent_assertion_statement ::=
   assert_property_statement
  | assume_property_statement
  | cover_property_statement

assert_property_statement::=    assert property ( property_spec ) action_block
assume_property_statement::=    assume property ( property_spec ) ;
cover_property_statement::=     cover property ( property_spec ) statement_or_null
```

NOTE: the keyword property appearing after the directives **assert**, **assume**, or **cover** identifies the concurrent assertions. The parentheses "()" around the *property_spec* is mandatory.

Verification Directives

The verification statement states the verification function to be performed on the property. The statement can be one of the following:

- **assert** to specify the property as a checker to ensure that the property holds for the design
- **assume** to specify the property as an assumption for the environment
- **cover** to monitor the property evaluation for coverage

Figure 2.3.2-1 demonstrates the application of a concurrent assertion in an **always** block. Note that the property does not specify the clocking scheme, which is defined in the **always** block.

```
module XX(...);
....
property p_handshake;
   disable iff (!reset_n)
   req |=> ## [0:4] ack ##1 !ack;
endproperty : p_handshake

always @ (posedge clk) begin : handshake_block
  // Context concurrent assertions
  ap_handshake : assert property (p_handshake);
end // always handshake block
endmodule : XX
```

[0:4] indicates that within a delay of 0 (i.e., no delay) to 4 cycles, *ack* must be TRUE.

Instantiation of property within a procedural block

Figure 2.3.2-1 Example of a Concurrent Assertion in an always Block

Figure 2.3.2-2 demonstrates the application of a concurrent assertion in a module.

```
module XX(...);
....

property p_handshake2;
   @ (posedge clk) disable iff (!reset_n) // with clocking and reset defined
      req |=> ## [0:4] ack ##1 bus_enable [*1:10] ##1!ack ##1 hold[*2];
endproperty : p_handshake2

ap_handshake2 : assert property (p_handshake2);

always @ (posedge clk) begin :handshake_block
...
end // always
endmodule : XX
```

"hold[*2]" means that *hold* is repeated 2 times. "bus_enable [*1:10] ##1!ack" means that *bus_enable* is repeated from 1 to 10 times until *!ack* is TRUE

Instantiation of property outside of a procedural block

Design

Figure 2.3.2-2 Example of a concurrent Property in a Module

The interface construct in SystemVerilog was specifically created to encapsulate the communication between blocks, allowing a smooth migration from abstract system-level design through successive refinement down to lower-level register-transfer and structural views of the design. By encapsulating the communication between blocks, the interface construct also facilitates design reuse. Assertions placed within the interface not only help in documenting properties about the interface, but can also catch errors in the inter-block connections. Those errors can be as simple as inversion mistakes, or as complex as protocol misunderstandings. Figure 2.3.2-3 demonstrates an example of properties and assertions in interfaces. [9]

```
interface handshake_if (input wire clk);
 timeunit 1ns;
 timeprecision 100ps;
 logic req, gnt, ack;
 modport slave_handshake_if (output req,
                             input  gnt,
                             input  ack);
 property pReqAck;
  @( posedge clk )
     req |=> gnt ##[1:3] ack ##1 !ack;
 endproperty : pReqAck
 apReqAck : assert property (pReqAck) ;
endinterface : handshake_if
```

See Section 8.2.5.5 Defining Time Unit and Time Format Specifications for Design for guidelines.

Figure 2.3.2-3 Properties in Interfaces

[9] See chapter 5. FIFO design and verification for detailed example of properties/assertions/tasks in interfaces.

2.4 BOOLEAN EXPRESSION

A Boolean expression is any expression that can be used as the condition of an **if** statement.
A Boolean expression can be any of the following:

- A variable, such as a *port, integer, wire, reg,* or *logic* of any width. An object need not be local, and can be accessed through a hierarchical path (e.g., *top.slave.controller.data).* If the value of an object is ZERO, the expression is FALSE. An expression will evaluate as TRUE if a vector is non-zero, with a known value (i.e., no bits set to X or Z). User may want to define a macro `true with the define, such as:

  `define true 1

  This is very useful in the definition of endpoints (explained in Chapter 3). For example, the following sequence declaration expresses a sequence of *bus_request* follwed by a don't care (TRUE) cycle:

  sequence qReqEnded;
  @(posedge clk) bus_request ##1 `true;
  endsequence : qReqEnded

  To demonstrate the concepts, that sequence can be used is a property expressions such as:

  bus_acknowledge |-> qReqEnded.**ended**;

  That property expression states that if *bus_acknowledge* is TRUE, then the endpoint of *bus_request* followed by a don't care cycle occurred in the same cycle as *bus_acknowledge*. Variations to the same statement include:

bus_request |=> bus_acknowledge; *// one cycle later, bus_acknowledge* ☺ ☺

bus_request |-> ##1 bus_acknowledge; *// one cycle later, bus_acknowledge* 😐

bus_request |-> `true[*1] bus_acknowledge; *// TRUE repeated once* ☹
// Shown here only to demonstrate legal construct.
// That style is NOT recommended though.

- Result of relational operation For example:

  (fifo_depth == MAX_SIZE) |-> fifo_full ;

  Meaning that if the FIFO depth is equal to the maximum size then, in the same cycle, the FIFO FULL signal should be asserted (i.e., to *1'b1*). MAX_SIZE is a constant.

3 UNDERSTANDING PROPERTIES

As explained in Chapter 2, properties are basically constructed from "sequences", which express a set of sequential expressions that span over one or more cycles. Properties span over multiple time intervals. They are based on clock semantics and use sampled values of signals and variables.

A property defines a behavior of the design. A property declaration by itself does not produce any result. A property must be instantiated using an **assert /assume / cover** verification directive or an **expect statement**. A property can be declared and instantiated in the following SystemVerilog declarations:

1. Module
2. Interface
3. Program
4. Clocking block
5. Package
6. Compilation-unit scope[1]

This chapter examines the definition and application of properties. It also addresses system functions related to properties. The chapter is organized as follows: Section 3.1 provides a brief overview of sequences. Since sequences are basic building blocks of properties, a quick introduction to sequences is presented. The goal is to provide the reader a basic familiarity with the SystemVerilog sequences without delving too much into every operator and its applications.

---

[1] Per LRM 18.3 *compilation-unit scope*: A scope that is local to the compilation unit.. A collection of one or more SystemVerilog source files compiled together. It contains all declarations that lie outside of any other scope.

Section 3.2 presents the SystemVerilog Properties in detail. Following this chapter, Chapter 4 presents, with examples, details in the use of sequences and sequence operators used to build complex sequences. Chapter 5 provides advanced topics in the use of assertions including system functions supporting assertions; single and multiple clocking of sequences; SystemVerilog scheduling for the support of assertions, properties in interfaces, verification directives and their significance in simulation and formal verification; and binding of designs unit with properties.

3.1 SEQUENCES OVERVIEW

A sequence in SystemVerilog can have a name (named sequence) and represents a finite series of Boolean expressions that expresses a set of behavior lasting one or more cycles. A sequence is thus a list of SystemVerilog Boolean expressions in a linear order of increasing time. The Boolean expressions must be true at those specific clock ticks for the sequence to be true over time. A Boolean expression at a point in time is a simple case of a sequence with time length of one clock cycle.

The following represents characteristics of named sequences:

- Can be declared, with optional arguments
- Can be instantiated in properties
- Can use local variables
- Can define a clocking event for the sequence
- Can be operated upon by sequence methods (e.g., **ended, matched,** or **triggered**)
- Can be identified as terminated with the **ended**, **matched**, or **triggered** methods[2]. These methods apply to named sequences only

Named sequences enhance code readability and reuse because they represent a collection of sequential events that have arguments. Named sequences ease the readability and of properties and sequences built using multiple small sequences, instead of a single large one. Methods can be applied to named sequences.

Examples of named sequences are shown in the next section. An example of an unnamed sequence is shown in the following property:

```
property pExampleOfUnamedSequences;

@ (posedge clk) (ready ##1 request |=> ack ##1 done;
endproperty : pExampleOfUnamedSequence
```

Unnamed sequence.

Unnamed sequence.

Table 3.1 summarizes the operators performed on sequences. A detailed explanation of sequence operators is provided in Chapter 4.

[2] See section 4.7

Table 3.1 Sequence Operator Summary[3]
(sequences can be named or unnamed)

Operation	Syntax	Explanation of requirement
Concatenation	seq1 ##1 seq2	*seq2* begins on the clock after *seq1* completes
Overlap	seq1 ##0 seq2	*seq2* begins on the same clock on which *seq1* completes
Ended Detection	seq1 ##1 seq2.**ended**	*seq2* completes on the clock after *seq1* completes, regardless of when *seq2* started
Repetition	seq1[*n:m]	Repeat *seq1* a minimum of *n* and maximum of *m* times. May result in multiple matching sequences
First Match Detection	**first_match**(seq1)	If *seq1* has multiple matches, use the first and ignore the rest
OR	seq1 **or** seq2	Compound sequence that matches when either *seq1* or *seq2* matches
AND	seq1 **and** seq2	Matches when one sequence maches on or after the other sequence also matches
Length-matching And	seq1 **intersect** seq2	Matches on cycles at which both *seq1* and *seq2* match
Conditional qualification	cond **throughout** seq	*cond* is TRUE for every cycle of *seq*
Within	seq1 **within** seq2	*seq1* starts on or after *seq2* and ends on or before the end of *seq2*

3.1.1 Sequence Declaration

A sequence can be declared with a name in

- a module
- an interface
- a program
- a clocking block
- a package
- a compilation-unit scope (see section 3.0 for explanation)

Once a sequence is declared, it can be either instantiated in properties or be used as level-sensitive sequence controls. Following example demonstrates these concepts.

```
sequence qAbort;
  @ (posedge clk) reset or (illegal_tx ##1 cancel);
endsequence : qAbort
```
[4]

qAbort represents a sequence indicating that at any clock cycle either a *reset* OR an illegal transaction (*illegal_tx*) followed by a *cancel* must occur.

```
parameter LENGTH = 8;
sequence qDataXfr(ack, enb, done);
    ack ##1 enb[*LENGTH] ##1 done;
endsequence : qDataXfr
```

qDataXfr represents a sequence of an *ack* followed in the cycle by 8 occurrences of *enb* followed by *done*.

---

[3] By permission http://www.synopsys.com/products/simulation/assert_sverilog_wp.pdf

[4] A naming convention used in this book is the prefixing of sequences with the letter "q" to enhance readability (*hint*: there is a "q" in "sequence")

```
property p_data_xfr (req, ack, enb, done);
@ (posedge clk) disable iff (qAbort.triggered) [5]
   $rose (req) |=>qDataXfr(ack, enb, done);
endproperty : p_data_xfr

ap_data_xfr : assert property (p_data_xfr(request, acknowledge, bus_enb, done_xfr)
   else $display ("%m data transfer error");
```

Property with two instantiations of sequences

Sequences are declared using the following syntax (please refer to Appendix E: SystemVerilog Assertions Syntax for full details).

```
sequence_declaration ::=
 sequence sequence_identifier [ ( [ list_of_formals ] ) ] ;
  [ { assertion_variable_declaration }]
  sequence_expr ;
 endsequence [ : sequence_identifier ]
```

Local variables within a sequence enable information to be stored within the thread of a sequence, and then tested at a later cycle within the sequence. For example:

```
sequence qMemData2Xbus (ready, done, mem_bus, x_bus);
 int v_data;
 (ready, v_data =mem_bus) ## [5:10] (done && x_bus== v_data);
endsequence : qMemData2Xbus
```

The sequence *qMemData2Xbus* states that if *ready* then local variable *v_data* gets assigned the value of *mem_bus*. Then within 5 to 10 cycles later, when *done* is TRUE then *x_bus* should be equal to the contents of the saved local variable *v_data*. The sequence fails if *ready* is FALSE or if following a successful *ready*, the expression "*done && x_bus== v_data*" fails to be TRUE within the range of 5 to 10 cycles after *ready*.

- Sequences may instantiate other sequences.
- Formal arguments of a sequence are untyped. The discussion of type mismatching of actual arguments and local variables for property instantiations and sequences is presented in section 3.4.1. [6]
- Local variables can be used in arithmetic manipulations of data. Thus, local variables of a sequence (or property) may be set to a value, which can be computed from a parameter or other objects (e.g., arguments, constants, objects visible by the sequence (or property)). Those variables can then be used in comparisons with other objects. However, they cannot be used to setup delays or repetition ranges, as those are required to be compile-time static.
- Actual arguments of a sequence may be a signal, sequence, a method of a sequence, a constant, a parameter, or the value returned by a function call.

[5] See section 4.7.3 for discussion of **triggered** method

[6] There is a discussion on providing in the LRM an optional type definition to formal arguments of sequences and properties.

3.2 SYSTEMVERILOG PROPERTIES

Table 3.2 provides a high level view of the property construction syntax (please refer to Appendix E: SystemVerilog Assertions Syntax for full details). Each element of this syntax is elaborated in greater details in the following subsections.

Table 3.2 Property Construct Syntax

```
property_declaration ::=
   property property_identifier [ ( [ list_of_formals ] ) ] ;
    [ {assertion_variable_declaration }]
    property_spec ;
   endproperty [ : property_identifier ]

property_spec ::=
    [clocking_event ] [ disable iff ( expression_or_dist ) ] property_expr

property_expr ::=
  sequence_expr
  | ( property_expr )
  | not property_expr
  | property_expr or property_expr
  | property_expr and property_expr
  | sequence_expr |-> property_expr
  | sequence_expr |=> property_expr
  | if ( expression_or_dist ) property_expr [ else property_expr ]
  | property_instance
  | clocking_event property_expr
```

// example
logic va, vb;
logic[31:0] vd = 0;

// example
@ (**posedge** clk) **disable iff** (!reset_n)
a |=> b

See section 3.2.5.3

It is important to note that a property is not a sequence. However, a sequence can be a property expression, as described in the syntax of a property. Thus, a sequence can be used where a property expression is needed for the definition of complex property expressions.

The property declaration consists of the following items, each of which is discussed in the following subsections:

1. Header
2. Identifier
3. Formal arguments (optional)
4. Local assertion variables (optional)
5. Property specification, i.e., the body of the property

To demonstrate the structure of a comprehensive property, consider the requirements of a cache controller in a multi-processor system. One of the requirements of such a cache controller is stated below in English and then an equivalent SystemVerilog Assertions property is shown in Figure 3.2.

In the event of a cache "read miss", the read request for the same address must be forwarded to the main memory within five clocks.

Header

Identifier

Formal argument declaration

Local assertion variables declaration

Property Specification/Body

Property instantiation with **assert**

```
property p_cache_miss_2_main_mem_xfer (
   cache_read,                  // read from cache control
   cache_miss,                  // miss control from cache controller
   rd_addr,                     // desired data address
   main_mem_rd,                 // main memory read control
   main_mem_rd_addr);           // main memory read address

 int v_rd_addr_at_cache;        // assertion variable to record and test the read address

 @(posedge clk) disable iff (!reset_n)  // abort if reset_n==0
   ((cache_read && cache_miss), v_rd_addr_at_cache =rd_addr) |=>
            ##[0:5] main_mem_rd && (main_mem_rd_addr == v_rd_addr_at_cache);
endproperty : p_cache_miss_2_main_mem_xfer

ap_cache_miss_2_main_mem_xfer : assert property // property instantiation
  (p_cache_miss_2_main_mem_xfer ( cache_rd, cache_miss, rd_addr,
                                  main_mem_rd, main_mem_rd_addr ));
```

The property is instantiated with the **assert property** statement that represents a verification directive to the verification tool to verify that the property holds, and if it fails to hold, then a failure shall be reported.

Figure 3.2 Timing Diagram for Cache Controller Property.

3.2.1 Property Header

A property declaration is enclosed within the keywords **property** and **endproperty.** The **endproperty** can be appended with the property identifier. It is a considered good documentation/coding style to append the name of the property after the **endproperty**.

3.2.2 Property Identifier

The property identifier names the property so that it can be referenced by verification directives (e.g., **assert, assume, cover**), or by other properties to construct new properties. The property identifiers should have meaningful names that clarify the functionality of the properties. Meaningful names are very useful for documentation and debugging. For example:

```
property pNeverPushOnFull;
  not(push && full && !pop);
endproperty : pNeverPushOnFull
```

3.2.3 Formal Arguments and Usage

Formal arguments to a property definition are optional, however, they enable the properties to be reusable. The formal arguments can have default values. If there is no actual argument for the defaulted argument, then the default values shall be used as the actuals. Figure 3.2.3 demonstrates the application of a reusable property with formal arguments using positional and named notations. For example, consider a property being developed to check for one-cold encoding of a bus.

```
parameter WIDTH=32;
property pASSERT_ONE_COLD (test, enable=1'b1);
   enable && (test!={ WIDTH {1'b1}}) |->
   (((~test == { WIDTH {1'b0}}) ||
   ((~test & (~test - {{ WIDTH -1{1'b0}}, 1'b1})) !=
   { WIDTH {1'b0}})));
endproperty : pASSERT_ONE_COLD [7]
```

Assume that the above property is used in a design with 2 buses declared *a_bus* and *b_bus*. One could reuse the pASSERT_ONE_COLD property described above for both these signals as follows:

```
apASSERT_ONE_COLDa_bus1 : assert property (pASSERT_ONE_COLD (
        .test(a_bus),
        .enable (enb)
);
```

Named notation, no default

```
apASSERT_ONE_COLDb_bus1 : assert property (pASSERT_ONE_COLD (
        . test(b_bus)
);
```

Named notation, with default

```
apASSERT_ONE_COLDa_bus2 : assert property (pASSERT_ONE_COLD (
        a_bus, enb)
);
```

Positional notation. no default

```
apASSERT_ONE_COLDb_bus2 : assert property (pASSERT_ONE_COLD (
        b_bus
);
```

Positional notation. with default

Figure 3.2.3 Application of default arguments to a property

[7] Algorithm extracted from Accellera OVL. http://www.eda.org/ovl/download.html

- If the formal arguments have default values, then place those arguments at the end of the list. This will create a more readable instantiation when the defaulted arguments are left out during instantiation.
- Use named notation in the instantiation of properties when the number of arguments exceed 3 or 4.
- Use %m in all assertion failure messages so that they could be easily identified and can be located for quicker debugging. To maintain consistency in the error reporting, use a common task for the error reporting (see the Guidelines chapter).
- Use formal arguments with properties when they are intended to be reused either for different assertions or other property definitions..
- Define the properties in terms of ports and registered signals when a design is targeted for synthesis and the assertions are expected to be reused during gate level simulations. The ports and registered signals are typically not optimized out during synthesis, and the object names can be traced. An example of a gate-level net can be described using escaped names, such as:
  "top.chip.\control_unit/module_1/mod_2/ .net_name".
- Do not use formal arguments with properties that are not intended to be reused or when they represent design-specific assertions being added while coding.

3.2.4 Local Variables in Properties

One of the powerful features of SystemVerilog Assertions is the ability to declare variables local to a property so that multiple threads of the same asserted property can maintain their individual copies. This feature is highly useful in pipelined designs where the consequent occurrence might be many cycles later than their corresponding antecedents. Local variables are optional and are local to properties. They can be initialized, assigned (and reassigned) a value, operated on, and compared to other expressions. The syntax of a property with a local variable is shown below.

```
property property_identifier [ ( [ list_of_formals ] ) ] ;
[{ assertion_variable_declaration }]
property_spec ;
endproperty [ : property_identifier ]
```

Asserted properties are multi-threaded, and each thread maintains its own copy of the automatic variables. Each thread and its local variables "live" until either the thread terminates (i.e., is TRUE) or it fails. The typical application of variables is the capture of data when the antecedent is valid, and have those values tested at later cycles within the same thread (typically during the consequent cycles). Since the assertion variables can also be operated on, they can be used as a

counter (to compute a repetition of a sequence) and/or as objects to be compared against arguments or signals. A simple example is shown below:

```
property p_with_local_var;
 logic [15:0] v_pkt_uid = 0;
 @(posedge clk) ($rose(sop_vld), v_pkt_uid = pkt_uid) |=>
      ##[63:1517] ($rose(eop_vld) && pkt_uid == v_pkt_uid);
endproperty : p_with_local_var
```

Declare local variable, and initialize if necessary.

Assign value to local var

Check using the local copy

- In the expression "(*b*, *v* = *e*)", if b holds, then variable *v* gets the value of *e*. Otherwise, no assignment to *v* is made
- Local variables can be updated one or more times within a thread

```
property p_req2send2;
  logic [15:0] v_data;
  @(posedge clk) ($rose(req_data), v_data =data) |=>
    ##[0:3] (ready && d==v_data, v_data=data) |=>
     ##[0:5] done && q==v_data;
  // If a new req_data, store data into local variable v_data,
  // then ready should occur within 1 to 3 cycles, at which time
  // register d should also be equal to the stored data
  //(i.e., the variable v_data). In that last cycle, local variable
  // v_data stores a new value of signal data.
  // Thereafter, from the next cycle, up to 5 cycles later
   // done should be active and register q should be equal
  // to the newly captured data stored in variable v_data..
endproperty : p_req2send2 [8]
```

- Local variables cannot influence the endpoints of temporal ranges. Thus, a local variable cannot represent a lower or upper value for a delay or a repetition range. This is because all temporal range operators are derived from explicit constant range operators, and must be known at elaboration time. Thus, the following is illegal:

```
property P;
 int v;
  ($rose(a), v=b) |=> ##[0:v] c;  // Illegal use of variable in range
endproperty : P
```

[8] `$rose` when used with a vector, returns TRUE if the least significant bit of the expression changed to 1. Otherwise, it returns FALSE.

First application example using variables [9]

A networking device requires that enqueued packets must be dequeued within 100 cyles. The dequeing order is non-deterministic and depends on target buffer availability and other factors. A problematic initial solution is shown below:

```
property pDatain2dataout_BUG;
 logic [31:0] v_data; // store incoming data
  ($rose(load), v_data=datain) |=> ##[0:100] dout && dataout==v_data;
  // when load, store input data to local variable
  // when packet is dequeued, then
  //check that the output data is same as what was stored.
endproperty : pDatain2dataout_BUG
```

The property appears to correctly define the requirements. However, an analysis of the following stream demonstrates a serious issue:

@cycle 2, load==1, datain== 1 // starts check for thread 1
@cycle 15, load==1, datain==1 // starts check for thread 2
@cycle 50, dout==1 and dataout==1
@cycle 50, thread 1 checks out OK and passes . Thread 2 is also terminated ✓

Thread 2 may complete in the same cycle as thread 1 completion when *dout* == 1 and the sampled data at load cycle is the same for both threads. Thus, uniqueness in not achieved. Figure 3.2.4-1 demonstrates the issues.

Figure 3.2.4-1 Improper Use of Variable Causing Potential Errors

A solution to this problem is to add a unique packet identification (ID) to each packet, and to check that the output value is the data that corresponds to the corresponding ID. This solution is shown below:

9 These examples address the use of variables. See section 7.15 "Data Integrity in QUEUES" for a more efficient solution that makes use of SystemVerilog queues.

```
property pDatain2dataout_OK;
logic [31:0] v_data; // store incoming data
int vpacket_id; // write packet ID  ← New
($rose(load), v_data=datain, vpacket_id= packet_id) |=> // also store the packet ID ← New
  ##[0:100] dout && packet_id==vpacket_id |-> // if match of ID, then  ← New
   dataout==v_data;  // check that data also matches
// when load, store input data to local variable,  also store the packet ID into a local variable
// when packet is dequeued, and packet ID is same as what was written, then
// check that the output data is same as what was stored.
// Note the 2 implication operators.
endproperty : pDatain2dataout_OK
```

Second application example using variables

When an acknowledge to a DMA request is received, the DMA logic transfers one data word per cycle to/from the memory until a *done* or *abort* is received. The following assertion insures that the number of memory words written/read corresponds to the value of the DMA address, which counts in an upward direction. For example, if the *dma_addr* is initially at address 10, and ends at address 15, then 5 words were read or written, as shown in Figure 3.2.4-2.

```
property pDMAcount;
 int v_count;       // word count
 logic[31:0] v_dma_addr;  // initial starting dma address
 @ (posedge clk) ($rose(dma_ack), v_count=1, v_dma_addr=dma_addr) |=>
                                                    // initialize the variables
      (`true, v_count += 1)[*0:$] ##1
      (done || abort) |-> v_count==(dma_addr - v_dma_addr);
endproperty : pDMAcount
```

Variable incremented at every repetition until *done* or *abort*

Variables initialized

Variables used in comparison

Figure 3.2.4-2 Use of Variable in a DMA Data Transfer

Third application example using variables

When a bus *request* is received, then within a user-defined number of cycles, *acknowledge* must be received. Thereafter, a transfer of data (indicated by *xfr*) must occur for *rep* number of cycles, and then *done* occurs. Figure 3.2.4-3 demonstrates a timing waveform for the property pReqAckDone with *upper* ==5, *lower*==2, and *rep*==2.

```
property pReqAckDone(req, ack, xfr, done, upper, lower, rep);
  int v;
  @ (posedge clk) (req, v = 1) |=> // If request, then initialize counter v.
  (v < upper && !ack, v += 1)[*0:$]  //@ every cycle  keep counting until v reaches
                                     // upper  range AND ack received
  ##1 ((v >= lower) && ack, v = 1) // Reset variable v to count number of data transfers
  ##1 (xfr && (v <=rep), v = v+1)[*0:$] // xfr control must be asserted vrep times
  ##1 v==rep // ends the repeat
  ##1 done;  // until done occurs. At that point, property completes.
 endproperty : pReqAckDone
```

Figure 3.2.4-3 Timing Waveform for the Property pReqAckDone with *upper* ==5, *lower*==2, and *rep*==2

3.2.5 Body of the Property

The *property_spec* portion of a property represents the body of the property, and describes its behavior. The structure of a property specification is:

property_spec ::= [clocking_event] [**disable iff** (expression_or_dist)] property_expr

It consists of three parts:

- An optional clocking event
- An optional disabling condition
- A property expression

The mnemonic **VCDP** (Very Costly Digital Processor) can be used to remember the order of statements in a property.
property name;
V // for Variable;
C // for Clock statement
D // for Disable iff() statement
P // for Property expression
// "C" is before "D" in the alphabet!
endproperty : name

3.2.5.1 Clocking Event

The *clocking_event* designates a particular event to act as the clock for the property. Typically, this expression is either the **posedge** or **negedge** of a clock signal. The timing of all the other signals specified in a given property is governed by the clocking event.[10]

The clocking event can be omitted in one of the following cases:

1. Properties defined in a clocking block
2. Properties defined with a default clocking. One clocking can be specified as the default for all cycle delay operations within a given module, interface, or program.
3. Properties built using sequences having clocked expressions within

Example:

- Properties defined in a clocking block

```
module test(input wire clk1, ...);
 logic req_data, ready;
 ...

 clocking assertions @(posedge clk1);
 // following properties use posedge clk1 as default clock
  property p_req2send;
   $rose(req_data) |=> ##[0:3] ready;
  endproperty : p_req2send
  // other property and sequence declarations
 endclocking : assertions

 // List of assertions
  ap_req2send_1: assert property (p_req2send);
endmodule : test
```

Clocking defined here, and valid for all properties and sequences defined within this clocking block. Multiple clocking blocks are allowed.

Property or sequence declaration here. NO assertion allowed.

[10] See section 5.10 for a discussion clock skew and setup time as address in assertions.

- Properties defined with a default clocking

```
module clocks (input clk1, ,...);
  clocking sysclk @(posedge clk1);
  endclocking : sysclk
...
endmodule : clocks

module processor (input clk1, input clk2, input a, output b,...);
  logic req_data, ready;
  default clocking  top.clocks.sysclk;  // clocks intantiated in top as clocks instance

  property p_reqack;
    $rose(req) |=> ##[0:3]ack;
  endproperty : p_reqack
  ap_reqack :  assert property (p_reqack);
endmodule : processor
```

Only one default clocking allowed per module, interface, or program.

- Properties built using sequences having clocked expressions within[11]

```
module cpu(..);
  sequence qS1;
      @(posedge clk1) a ##1 b ##1 c;
  endsequence : qS1
  sequence qS2;
      @(posedge clk2) d ##1 e ##1 f;
  endsequence : qS2

  property p_abcdef;
      qS1 |=> qS2;
  endproperty : p_abcdef

  ap_abcdef :  assert property (p_abcdef);
endmodule : cpu
```

Clock expression here

Clock expression here

Multi-Clock property

For a multi-clocked assertion, the clocks are explicitly specified. No default clock or inferred clock is used. In addition, multi-clocked properties are not allowed to be defined within a clocking block. A multi-clocked property **assert** statement must not be embedded in procedural code where a clock is inferred. For example, the following forms are not allowed (using *p_abcdef* as the multi-clocked property):

```
always @(clk) assert property (p_abcdef); // illegal
initial @(clk) assert property (p_abcdef);  // illegal
```

3.2.5.2 Disabling condition

A **disabling condition (disable iff** (disabling_expr) **)** clause can be attached to a *property_spec* and is typically used as a reset of the property. The **disable iff** clause allows asynchronous resets to be specified. The reset expression is tested independently for different evaluation attempts of the *property_spec*. Assertion checking is terminated when the **iff** expression is TRUE, and any

[11] See section 5.9.3 for more information on clocking sequences and multiple clocks.

future obligations of the property are also cancelled as long as the disabling expression is TRUE. When the disabling expression becomes FALSE, the property checking is then resumed. The syntax for the disable is:

disable iff (disabling_expr) property_expr

Nesting of **disable iff** clauses, explicitly or through property instantiations, is not allowed because the **disable iff** is not allowed in a property expression.

Do not use **disable iff** in properties that are likely to be used in other properties.

Restrict the **disable iff** to the higher-level properties only (higher-level properties refer to those properties that are not used as sub-expressions in any other properties).

Use **triggered** method when the disabling condition is an endpoint of a sequence or an event.[12]

Examples:

```
property pXthenY(x,y, reset);
  disable iff (reset)
  x |=> y;
endproperty : pXthenY
```

See section XX on the use of the method **triggered** in the **disable iff**

```
property pXthenYnoDisable(x,y, clr);
  x |=> ##[0:$] y;
endproperty : pXthenYnoDisable
```

```
property pIllegal;
  @(posedge clk)
  a ##1 (b || c)[*->1] |->
     pXthenY(d,e, reset);
endproperty : pIllegal
```

Cannot have **disable iff** in the body of a property

```
property pLegal;
  @(posedge clk)
  disable iff (reset)
  a ##1 (b || c)[*->1] |->
   pXthenYnoDisable (d,e)
endproperty :pLegal
```

disable iff is at higher level property only.

OK, no **disable iff** in the body of the property

```
event blast;
property P;
 disable iff (blast.triggered)
  a |=> ##[0:100] b;
endproperty : P
```

Use of **triggered** method of an event

[12] See Chapter 4, **triggered** method for coding style to enable a disable condition based on the endpoint of a sequence. The event's triggered state persists throughout the time-step (i.e., until simulation time advances).

3.2.5.3 Property Expression

A property expression represents the behaviour of the property, and per LRM represents one of following:

```
property_expr ::=
  sequence_expr
  | ( property_expr )
  | not property_expr
  | property_expr or property_expr
  | property_expr and property_expr
  | sequence_expr |-> property_expr
  | sequence_expr |=> property_expr
  | if ( expression_or_dist ) property_expr [ else property_expr ]
  | property_instance
  | clocking_event property_expr
```

In general, property expressions are built using sequences, other sub-level properties and simple Boolean expressions. These individual elements are combined using property operators; these operators are elaborated in the next sub-section.

3.2.5.3.1 Property Operators

Table 3.2.5.3.1 summarizes the property operator precedence from highest to lowest, and shows the associativity of the non-unary operators.

Table 3.2.5.3.1 Property Operator Precedence, Highest to Lowest

SystemVerilog Property Operator	Associativity	Precedence	Comments	Example
not	---	High	property is never to occur	**not** (read && write)
and	Left	↓	Both properties must be met for the property to be fulfilled	Data_xfr_prop **and** Data_no_error_prop
or	Left	↓	Either of two properties satisfies the property	Data_xfr_prop **or** cancel_prop
if ... else	right	↓	If the Boolean test expression is satisfied the property following the **if** must be met. If the test expression fails, the **else** property must be satisfied.	**if** (mode==READ) read_property **else** write_property
|-> |=>	right	Low	Implication operators. If a sequence is met, then the implied property must be met.	read |-> read_property

Definitions:[13] A left-associative operator, like subtraction, groups the leftmost elements first. A right-associative operator, like exponentiation, groups the rightmost elements first. An operator is non-associative when it doesn't make sense to use in succession (for example, the associativity of the operator "is an element of" is "non-associative").

The implication operators |-> and |=> are discussed in section 4.3.

The **not** property expression states that the property should not be TRUE; otherwise the property fails.

Use the **not** property operator with caution.

A good application of the **not** property is with a simple properties of Boolean expressions with no implication operator, meaning that the property should never occur. For example:

property pNeverReadAndWrite;
  not (mem_read && mem_write);
endproperty : pNeverReadAndWrite

See section 4.3.5 for warnings in the use of the **not** property operator with an implication operator

The property operator precedence is important because it defines the order of association, and if a different association is needed, then parentheses must be used. For example:

a ##1 b |=> c ##1 d **or** e **and** f ##1 g; // Equivalent to:
(a ##1 b) |=> ((c ##1 d) **or** e) **and** (f ##1 g);
|sequence| |property| |property| |property|
priority: 3 4 2 1

Use parentheses to clarify intent!

If a user needs to associate the ordering of the property:
(c ##1 d) **ored** with the property (e **and** f ##1 g) then parentheses are needed.[14]

a ##1 b |=> c ##1 d **or** (e **and** f ##1 g); // Equivalent to:
(a ##1 b) |=> (c ##1 d) **or** (e **and** f ##1 g);
|sequence| |property| |property| |poperty|
priority: 3 4 2 1

A property operator with the **if...else** has either the form

if (expression_or_dist) property_expr1

or the form

if (expression_or_dist) property_expr1 **else** property_expr2

expression_or_dist ::= expression [**dist** { dist_list }]

Per LRM, the property of the form "**if** (expression_or_dist) property_expr1" evaluates to TRUE if and only if either *expression_or_dist* evaluates to FALSE or *property_expr1* evaluates to TRUE. Note that this form is equivalent to "*expression_or_dist* |-> *property_expr1*". For example:

---

[13] For those who have not used these terms for a long time!

[14] A sequence is a property.

```
property pReq2send;
  if (req) ack;
endproperty : pReq2send
```

req	ack	Req2send
0	x	TRUE, vacuously
1	1	TRUE
1	0	FALSE

note: This is equivalent to
  req |-> ack;

A property of the form "**if** (expression_or_dist) property_expr1 **else** property_expr2" evaluates to TRUE if and only if either *expression_or_dist* evaluates to TRUE and *property_expr1* evaluates to TRUE, or *expression_or_dist* evaluates to FALSE and *property_expr2* evaluates to TRUE. Note that this form is equivalent to two property expressions:
"*expression_or_dist* |-> *property_expr1*".
"! *expression_or_dist* |-> *property_expr2*". For example:

```
property pReq2send2;
  if (req) ack else cancel;
endproperty : pReq2send2
```

req	ack	cancel	Req2send2
0	x	1	TRUE
1	1	x	TRUE
0	x	0	FALSE
1	0	x	FALSE

note: This is equivalent to 2 properties with
  req |-> ack; *// for one property*
  !req |-> cancel; *// for the other property*

The definition of the **if .. else** enables the expression of more complex properties that expresses multiple cases in a single property. For example, the following property expresses an allocation priority and bus acquisition sequences.

```
property p_busAbusB_allocation;
 if ($rose(req_busA) && !busyA && !busyB) // If a request for busA,
                                          // and both busA and busB are not busy
   // then busB is locked, busA gets flagged as busy, and acknowledge is asserted within 5 cycles.
   // Thereafter, doneA is asserted within 10 cycles and bus A is flagged as not busy
   ##1 lock_busB && busyA ##[0:5] ackA ##[1:10] doneA && !busyA

 else if ($rose(req_busB) && ! busyA && !busyB) // else a request for bus B
                                                // and both busA and busB are not busy
 // then bus A is locked, busB gets flagged as busy, and acknowledge is asserted within 10 cycles.
 // Thereafter, doneB is asserted with 20 cycles and bus B is flagged as not busy
   ##1 lock_busA && busyB ##[0:10] ackB ##[1:20] doneB && !busyB
 else
  // If bus A or Bus B is busy, then the busy flag must be set.
  busyA || busyB |-> buses_busy;
endproperty : p_busAbusB_allocation
```

The expression in the **if** condition can also be a distribution, which is more applicable to pseudo-random simulation or semi-formal verification. In a distribution, the expression: "x **dist** {100 := 1, 200 := 2, 300 := 5}" means *x* is equal to 100, 200, or 300 with weighted ratio of 1-2-5. An application example can be a multiplexing switch with priorities on the select control. The control *switch_select* has values of 2'b00 10% of the time, 2'b01 40% of the time, 2'b10 30% of the time, and 2'b11 20% of the time. *lock_enb* control can remain at a high level up to 10 cycle before *done* is asserted if the *switch_select* transitions from 2'b11 to 2'b00. Otherwise, *lock_enb* remains high for one to two cycles. Using the property expression:

" **if** (*expression_or_dist*) *property_expr* [**else** *property_expr*]" // expanded to:
"**if** (*expression_or_dist*)
if (*expression_or_dist*)
if (*expression_or_dist*) *property_expr* [**else** *property_expr*"

```
property p_switch_prop;
 if (switch_enb)
  if (switch_select dist{2'b00:= 1, 2'b01 := 4, 2'b10:= 3, 2'b11:=2})
   if (switch_select ==2'b00 && $past(switch_select==2'b11)) // from 11 to 00
                                        ##1 lock_enb[*1:10] ##1 done // NO ";"
   else lock_enb[*1:2] ##1 done;
 endproperty : p_switch_prop
```

The biasing feature (i.e., statistical distribution) is only useful when properties are considered as assumptions to drive random simulation. Thus, with

```
mp_switch_prop : assume property (@ (posedge clk) p_switch_prop);
```

the verification tool will have the *switch_select* with a specified distribution.

However, when a property with biasing is used in an assertion or coverage, the **dist** operator is equivalent to System Verilog's **inside** operator[15], and the weight specification is ignored. Hence with

```
ap_switch_prop: assert property (@ (posedge clk) p_switch_prop);
```

The simulator will interpret the property as:

```
property p_switch_prop;
 if (switch_enb)
   if (switch_select inside{2'b00, 2'b01, 2'b10, 2'b11} && switch_enb)
    if (switch_select ==2'b00 && $past(switch_select==2'b11)) // from 11 to 00
                                        ##1 lock_enb[*1:10] ##1 done;
   else lock_enb[*1:2] ##1 done;
 endproperty : p_switch_prop
```

[15] Refer to 7.20 Set membership of SystemVerilog LRM 3.1a version.

4 UNDERSTANDING SEQUENCES

The real power of SystemVerilog Assertions lies in the underlying strong support for sequences. In the previous chapter, section 3.1 briefly introduced the anatomy of a SystemVerilog sequence to enable the reader understand the SystemVerilog properties. In this chapter, a detailed look into the sequences with its operators is provided along with several examples.

4.1 SEQUENCE OPERATORS AND BUILT-IN FUNCTIONS

Operators provide operations on SystemVerilog code, properties, and sequences. The sequence operators and sequence built-in functions are summarized in Tables 4.1-1 and 4.1-2.

Table 4.1-1 Sequence Operators and Precedence

SystemVerilog Expression Operator	Associativity	Precedence	Comments	Example
[* n] [= n] [-> n]	---	High	Repetition, n=natural number	ack[*4] data_enb[=2] ack[-> 1]
## n	left		Cycle Delay operator, *n* is a natural number	req ##1 ack
throughout	Right		A signal throughout a sequence	enable **throughout** ack ##[1:$] done
within	Left		An inner sequence within an outer sequence	xfr_sequence **within** dma_sequence
intersect	Left		Sequences need to start and end at the same time	data_xfr_sequence **intersect** ack_done_sequence
and	Left		Sequences need to start at the same time, but may end at different times	data_xfr_sequence **and** ack_done_sequence
or	Left	Low	Either sequence satisfies the property	data_xfr_sequence **or** cancel_sequence

Table 4.1-2 elaborates on these operators and adds built-in SystemVerilog functions and methods. Each of these operators and functions and methods is then further explained in the following subsections.

Table 4.1-2 Sequence Examples and SystemVerilog Functions

Feature	Syntax	Examples and Comments
Cycle delay		
simple delay	##n	a ##1 b ##2 seq
delay range	##[n:m]	a ##[1:5] b ##[0:10] seq
open-ended	##[n:$]	a ##[1:$] b
Repetition		
fixed	[*n]	(a[*3] ##1 seq[*2] ##1 c) [*2]
empty	[*0]	a ##1 b[*0] ##1 c *//same as a ##1 c*
range	[*n:m]	a[*1:3] ##1 b[*0:2] ##1 c
open-ended	[*n:$]	a[*1:$] ##1 b[*0:$] ##1 c
non-consecutive	[= n]	a[=2] ##1 b[=3] ##1 c
goto repetition	[-> n]	a[->1] ##1 b[->2] ##1 c

Feature	Syntax	Examples and Comments
Sequence operator		
and	SEQ **and** SEQ	seq1 **and** seq2
intersect	SEQ **intersect** SEQ	seq1 **intersect** seq2
or	SEQ **or** SEQ	seq1 **or** seq2
throughout	EXP **throughout** SEQ	enable **throughout** dma_seq
within	SEQ **within** SEQ	xfr_seq **within** dma_seq
first_match	**first_match** (SEQ)	**first_match**(a ##[1:4] b ##1 c[*1:3] ##1 d)
Endpoint [1]	SEQ_Name.**ended**	
ended of a sequence		seq1.**ended** |-> ...
matched of a sequence		seq1_clk1.**matched** |-> seq2_clk2
triggered of a sequence		**disable iff** (seq1.**triggered**)
Property operators		
not property_expr	**not** PE	**not** (a ##1 b)
or	PE **or** PE	PE_Name **or** seq1[2]
and	PE **and** PE	PE_Name **and** (a ##[1:5] b)
overlapping implication	SEQ |-> PE	a ##1 ##2 b |-> c [*0:$] ##1 d
non-overlapping	SEQ |=> property_expr	a ##1 ##2 b |=> c [*0:$] ##1 d
if (EXP) PE [**else** PE]	**if** (EXP) PE [**else PE**]	**if** (req) ##[1:5] grnt **else** !grnt
Sampled value functions	**$sampled**(EXP [, clk_evnt])	**$display**("sampled(data) %0d", **$sampled**(data));
	$rose(EXP [,clk_evnt])	**$rose**(req) |=> ack;
	$fell(EXP [,clk_evnt])	**$fell**(ready) |-> !bus_enb;
	$stable(EXP [,clk_ev])	**$stable**(ack) |-> bus_enb;
	$past(EXP1 [, number_of_ticks] [, EXP2] [,clk_evnt])	ack |=> dreg==**$past**(data, 2);
Functions used within properties	**$onehot** (EXP) returns TRUE if only one bit of the expression is high.	**$onehot** (fsm_reg) ;
	$onehot0(EXP) returns TRUE if at most one bit of the expression is high.	**$onehot0**(fsm_reg);
	$isunknown (EXP) returns TRUE if any bit of the expression is X or Z.	enb |-> ! **$isunknown** (data_bus);
	$countones (EXP) count the number of 1s in a bit vector expression.	**$countones** (count_reg)== 5 |=> ..;

4.2 CAPTURING TEMPORAL BEHAVIOR IN SYSTEMVERILOG ASSERTIONS

The SystemVerilog Assertion provides the ability to capture the system behavior across a "time window". Assertions using Logical values of design elements (wires and registers) provide basic functionality to capture design intent. But any non-trivial design will have a behavior spanning

[1] See section 4.7 for elaboration of endpoints

[2] Keep in mind that a sequence is a property

across a number of clock cycles, and the window of such a span is variable. SystemVerilog provides a "Cycle Delay" operator to capture such behaviors. Cycle delays can be classified as

1. simple delay ##n *// n is a natural number (0 to $2^{\wedge 31}$ - 1) of clock cycles*
2. delay range ##[n:m] *// n and m are natural number, representing a range of cycles*
3. open-ended ##[n:$] *// n is a natural number, $ represents infinity, representing a // range n or more cycles*

A ## followed by a number or range (as shown in Table 4.1-2) specifies the delay from the current clock cycle to the beginning of the sequence that follows. For example, the delay ##1 indicates that the beginning of the sequence that follows, is one clock cycle later than the current cycle. Thus the ##1 represents the concatenation of two sequences separated by one clock tick. Example: a ##1 b;

In Verilog, a single "#" denotes timing delays typically used to specify delay within a given clock period. These delays may represent propagation or gate delays, or the period of a clock. However, two continuous pound signs "##" extend the same notion of delay, but enhance that delay to specify the number of clock cycles (or ticks).

The delay ##0 indicates that the beginning of the sequence that follows is at the same clock cycle as the current clock tick. Essentially, the ##0 represents no delay between the end of one sequence and the beginning of the next sequence. This is also known as "fusion".

When the delay is a range, such as ##[n:m], the delay ranges from n^{th} cycle up to either a maximum m^{th} cycle, or until the first element of the sequence that follows the delay range becomes TRUE.

An example of a sequence with simple delays is shown in Figure 4.2-1.

```
sequence qAB2C;
    $rose(a) ##1 b ##2 c;
endsequence : qAB2C

property pConcatenationSimpleExample;
@(posedge clk)
  qAB2C;
endproperty : pConcatenationSimpleExample
```

Most practical asserted properties include and antecedent and a consequent , e.g.,

```
property pAB2C; @ (posedge clk)
  x |-> qAB2C;
endproperty : pAB2C

apAB2C : assert property (pAB2C);
```

Properties may also use the **not** operator to express that a never sequence:

```
property pAB2Cnever; @ (posedge clk)
  not qAB2C; // sequence is a property here
endproperty : pAB2Cnever
```

apConcatenationSimpleExample: **assert property** (pConcatenationSimpleExample);

Figure 4.2-1 SEQUENCE Simple Example

This *apConcatenationSimpleExample* assertion is demonstrated with a timing model in Figure 4.2-2. This property states that at every positive edge of a clock cycle (when the property is evaluated), starting from the first cycle, the following is required to occur for the property to hold:

1. If *a* rose to TRUE, then the sequence continues, and will check for *b* ==1 at the next cycle. If *a* did not rise then the assertion **fails** in that cycle. Note that since the sequence is not used as an antecedent in the example property, a failure of a term of the sequence causes the property to fail, and not to succeed vacuously. When the sequence is used as an antecedent, then a failure of a term in the sequence causes the property to succeed vacuously.
2. If the first cycle assertion passes, then if *b* is TRUE on the second cycle, the sequence continues. Otherwise, the assertion fails.
3. If the second cycle assertion passes, then on the fourth cycle *c* == TRUE is tested. If it is, then the assertion for the sequence passes, otherwise it fails.

Figure 4.2-2 Timing Diagram for Sequence $rose(a) ##1 b ##2 c;

As mentioned previously, this type of assertion with simple concatenations within a property is rarely used because of overlapping issues, which can cause errors[3]. To avoid such errors, sequences are typically used with implication operators.

The following sequence uses a delay range from 1 to 3 cycles.

```
sequence qRangeSimpleAndRangeExample1;
  @(posedge clk) a ##1 b ##[1:3] c;
endsequence : qRangeSimpleAndRangeExample1
```

Figure 4.2-3a represents the timing diagram for this sequence. Note that this sequence requires that *c* occurs either 1, or 2, or 3 cycles after *b*.

[3] See section 5.2 for discussion on overlapping assertions.

Figure 4.2-3a Timing Diagram for Sequence "a ##1 b ##[1:3] c"

The following sequence uses a delay range from 0 to 3 cycles.

```
sequence qRangeSimpleAndRangeExample2;
  @(posedge clk)     a ##1 b ##[0:3] c;
endsequence : qRangeSimpleAndRangeExample2
```

Figure 4.2-3b the timing diagram for this sequence. Note that this sequence requires that *c* occurs either 0, 1, or 2, or 3 cycles after *b*. A zero cycle delay means that *c* can occur in the same cycle as *b*.

Figure 4.2-3b Timing Diagram for Sequence "a ##1 b ##[0:3] c"

The following demonstrates a sequence with the open range ##[0:$].

```
sequence qOpenRange;
  @(posedge clk)   a ##1 b ##[0:$] c;
endsequence : qOpenRange
```

The timing for this sequence is demonstrated in Figure 4.2-4. Note that *c* can occur after *b* at any cycle between 0 cycle (i.e., same cycle as *b*) or at any future cycle.

Figure 4.2-4 Timing for a Sequence with an Open Range

4.3 IMPLICATION OPERATORS

For many protocol related assertions, it is important to specify the sequence of events (the antecedent or cause) that must occur before checking for another sequence (the consequent or effect). This is because if the antecedent does not occur, then there is no need to perform any further verification. In hardware, this occurs because the antecedent reflects an expression that, when active, triggers something to happen. Examples include *bus_request, read,* which cause some consequent or effect to occur, such as a handshake with data transfer or a data fetch sequence. Two forms of implication operators are supported for the definition of properties

overlapped implication operator |->
non-overlapped implication operator |=>

Note that the implication operators are property operators rather than sequence operators. Also, the antecedent is a sequence, and the consequent is a property or a sequence (since a sequence is a property expression). The syntax per LRM is:

```
property_expr ::=
  sequence_expr
...
  | sequence_expr |-> property_expr
  | sequence_expr |=> property_expr
```

antecedent
consequent

Since a property expression can be a sequence, a property can be expressed as:

```
  sequence_expr |-> sequence_expr
| sequence_expr |=> sequence_expr

| sequence_expr |-> sequence_expr |-> property_expr
| sequence_expr |-> sequence_expr |-> sequence_expr |-> property_expr
```

Implication operator
|-> or |=>

- When the antecedent fails, the property holds, and is TRUE. This condition is referred to as "vacuous success". The success is vacuous because when the antecedent fails, it adds little to the meaning of the property. For example, a property expression "*request* |=> *acknowledge*" is relevant or meaningful only when *request* is TRUE since it then requires that *acknowledge* must then be TRUE. However, when *request* is FALSE, there is no required implication.
- A vacuous success does not mean that the property completes successfully. It just means that the antecedent is FALSE, hence the property did not complete. To insure full success users need to verify the coverage metrics of properties. Many tools provide automatic coverage data for the assertions exercised during a simulation run. These include for each assertion, the number of times the assertion was started, finished (or completed), and failed. If the simulation does not provide this feature, then users need to either use the **cover** verification directive, or compute these statistics with the action blocks.
- When the antecedent is TRUE, the property requires that the consequent must also be TRUE.

4.3.1 Overlapped implication Operator |->

For overlapped implication, if there is a match for the antecedent *sequence_expr*, then the end point of the match is the start point of the evaluation of the consequent *property_expr*. For example, the following property is demonstrated in Figure 4.3.1.

```
property p_test; // assuming a default clock
    a ##1 b |-> c ##1 d;
endproperty : p_test
```

. **Figure 4.3.1 Timing Diagram for Property *test***

An application example for the non-overlapping operator |-> might be a simple handshake, as in:

```
property pRequestAck;
```

```
    bus_free ##1 request |-> acknowledge ##[1:100] done;
endproperty : pRequestAck
```

A sequence of *bus_free* followed in the next cycle by a *request* must have *acknowledge* in the same cycle that *request* is TRUE. Theafter, either in the next cycle or up the 100th cycle *done* must be asserted.

4.3.2 Non-Overlapped Implication Operator |=>

For non-overlapped implication, the start point of the evaluation of the consequent *property_expr* is the clock tick after the end point of the antecedent. Therefore (and by definition):

sequence_expr "|=>" property_expr is equivalent to:
sequence_expr "##1 'true |->" property_expr

Figure 4.3.2-1 provides the timing for the following sequence expression:
a ##1 b |=> c ##1 d;

Figure 4.3.2-1 Timing Diagram for a ##1 b |=> c ##1 d;

An application example for the non-overlapping operator |=> might be a simple handshake, as in:

```
property pRequestAck2;
  bus_free ##1 request |=> acknowledge ##1 done;
endproperty : pRequestAck2
```

A sequence of *bus_free* followed by a *request* is followed at the next cycle by an *acknowledge*, and in cycle after that by *done*.

A method to easily remember the differentiation between overlapped and non-overlapped implication operators is shown in Figure 4.3.2-2.[4] When two hands overlap, they form a "single bar", and the fingers of one hand "touch" the palm of the other hand, thus creating the "overlap". This is the "|->" symbol. However, when two hands are above each other, and the tip of the

[4] This memory key was graciously donated by Cliff Cummings

fingers of one hand "touch" the wrist of the other hand, they create a "non-overlap". The "parallel "bars" of the hands above each other can be a key to remember the "=" in the non-overlapping "|=>" symbol.

Figure 4.3.2-2 Memory tip to Remember the Differentiation between Overlapping and Non-Overlapping Operators in Properties

4.3.3 Understanding the Overlapped Implication Operator "|->"

The "|->"overlapped implication operator allows *if – then* checking in a property. This operator specifies that the condition that follows the operator (i.e., the fulfilling condition or consequent) begins in the same cycle in which the endpoint of the antecedent becomes TRUE Thus, *a ##[1:3] b ##1[1:5] c|-> d ##1 e;* implies that a check for the sequence *(d ##1 e)* begins in the SAME cycle the endpoint of the sequence *(a ##[1:3] b ##1[1:5] c)* become TRUE. Figure 4.3.3-1 represents the case for (a ##1 b ##1 c |-> d ##1 e)

Figure 4.3.3-1 Timing sequence for *a ##1 b ##1 c|-> d ##1 e;*

Figure 4.3.3-2 represents a property using "|->"overlapping implication operator.

Figure 4.3.3 -2 Examples for overlapping Implication Operator |->

That specification has the following behavior, and is also shown in Figure 4.3.3 -3 as an FSM:

- If the endpoint of the sequence (a==1'b1) is TRUE, the property begins to look for the sequence (b==1'b0 ##1 c==1'b1). It starts looking for that sequence in the same cycle that endpoint of the antecedent succeeds (*a* in this case).
- If the sequence fails, then the implication succeeds vacuously and returns TRUE.
- If antecedent is TRUE and the consequent (b==1'b0 ##1 c==1'b1) also completes, the property is TRUE for that thread.
- If the consequent sequence (b==1'b0 ##1 c==1'b1) fails, the property fails.

Figure 4.3.3 -3 CONDITIONAL _SEQUENCE_EXAMPLE with |-> Overlapping Implication Operator, a |-> !b ##1 c;

A **first_match** function call to the antecedent can be used to limit the enabling condition to the first match of the antecedent sequence. [5]

[5] See section 4.4 on discussion of **first_match**

4.3.4 Understanding the Non-overlapped Implication Operator " |=>"

The "|=>" non-overlapped implication operator is equivalent to "##1 'true |->". Thus, the "|=>" starts from the cycle after the cycle(s) in which the enabling condition (or antecedent) becomes TRUE. Thus, *a ##[1:3] b ##1[1:5] c|=> d ##1 e;* implies that a check for the sequence *(d ##1 e)* begins in the cycle after the endpoint of antecedent *(a ##[1:3] b ##1[1:5] c)* becomes TRUE.

Figure 4.3.4-1 represents the case for (a ##1 b ##1 c |=> d ##1 e).

Figure 4.3.4-1 Timing Diagram for *(a ##1 b ##1 c |=> d ##1 e)*

Figure 4.3.4-2 represents another Property using "|=>", non-overlapping implication operator.

a |=> !b ##1 c;

If the endpoint of antecedent *a* is TRUE then check sequence *b==1'b0* in next cycle. If sequence in antecedent fails then the property succeeds vacuously.

Figure 4.3.4-2 Example for Non-Overlapping Implication Operator |=>

Figure 4.3.4-3 represents the FSM for the above assertion.

Figure 4.3.4-3 CONDITIONAL _SEQUENCE_EXAMPLE with |=> Non-Overlapping Implication Operator, a |=> !b ##1 c;

A **first_match** function call of the antecedent limits the enabling condition to the first match of the antecedent sequence. [6]

4.3.5 Using "not" with Implication Operator

The **not** of a property implies that the property should never occur. However, when the **not** operator is used with an implication operator, unexpected results may occur. Consider:

```
property p_new_prop;
  not orig_prop;
endproperty : p_new_prop
```

Assuming that the *orig_prop* had an antedecendent and consequent parts, if the antecedent of the implication fails, the original property succeeds vacuously and returns TRUE. The **not** of TRUE is FALSE, and the *new_prop* property fails. For example:

```
property p_test_of_not;
   @(posedge clk) not (a |-> b);
endproperty : p_test_of_not
```

That property has the following meaning:

a	b	a|-> b	**not** (a|-> b)	a |-> ! b
0	0	TRUE*	FALSE	TRUE*
0	1	TRUE*	FALSE	TRUE*
1	0	FALSE	TRUE	TRUE
1	1	TRUE	FALSE	FALSE

* = Vacuously TRUE

It is often desired to express the notion of "the antecedent should never be followed by a consequent, and if the antecedent is FALSE then the property is vacuously TRUE". However, the "**not** (antecedent *implication* consequent) does not express that desired notion. The above table shows that "**not** (a |-> b)", is equivalent to "a && ! b" and is not the same meaning as the desired "a |-> !b", which is equivalent to "!(a &&b)". The use of the implication operator with the **not** property operator gets even more confusing when there are cascading of implication operators, like "**not** (a |=> b |-> c |=>d)". Instead of this structure, restrict the use of the **not** property operator to the **not** of a property that is a sequence, without any implication.

An alternative to express that if sequence A occurs, then sequence B should never occur can be written using one of the legal syntax of properties:

```
property_expr ::=
        sequence_expr |-> property_expr
```

This can then be expanded to:

```
        sequence_expr |-> sequence_expr |-> property_expr
```

Thus,

```
SequenceA |-> sequenceB |-> `false;
```

Above states that if *SequenceA* occurs and *SequenceB* occurs, then the property expression fails.

---

[6] See section 4.4 on discussion of **first_match**

Avoid using **not** with the implication operators as this may yield unexpected results. Some tools may forbid such format.

```
not  a;
not  (a |-> b);
a |-> b |-> `false;

not  (a ##1 b [*1:4] ##1 c);
a ##1 b |-> `false;
not  (a ##1 b |=> c ##1 d);
(a ##1 b |=> c ##1 d) |-> `false;
a ##1 b  |=> (!c or  c ##1 !d);
```

4.4 FIRST_MATCH OPERATOR

By definition, a sequence can have multiple matches. For example, the sequence "a ##[1:2] b" has two possible matches: "(a ##1 b)", "(a ##2 b)" when *a* is TRUE in cycle *n*, and *b* is TRUE in cycles *n+1* and *n+2*. A more complex sequence, such as "(a ##1 b[*1:10] ## [1:8]c ##1 d)" may have many possible matches.[7] The **first_match** operator matches only the first of possibly multiple matches for an evaluation attempt of its operand sequence. This allows all subsequent matches to be discarded from consideration. In particular, when a sequence is a subsequence of a larger sequence, then applying the **first_match** operator has significant effect on the evaluation of the enclosing sequence.

sequence_expr ::= ...
 first_match (sequence_expr (, sequence_match_item))

The evaluation of a property with a sequence may behave as if an implicit **first_match** operation was applied on that sequence, depending on use. In the context of a property evalution, there are cases where a sequence behaves as if an implicit **first_match** operator was applied. These cases include:

1. A property that is a sequence (i.e., with NO implication operator in the property) behaves as if the sequence has **first_match** applied to it because as soon as a match of the sequence expression is determined, the evaluation of the property is TRUE, and the property completes for that thread.
   property pSequence; (a ##1 b ##[0:5] c); **endproperty** : pSequence
2. A property with a consequent sequence behaves as if the consequent has an implicit **first_match** applied to it. For example:
   property P2; a |=> b ##[1:2] c ##[0:4] d ; **endproperty** : P2
   // If *a* is TRUE, then some possible matches for consequent are
   // 1. b ##1 c ##1 d
   // 2. b ##2 c ##3 d
   If @ cycle t, a==1, then any first match of the consequent completes the assertion.
3. An antecedent sequence does not have an implicit **first_match** applied to it. For example:

[7] See section 3.7 for more examples of multiple matches of a sequence

property P3; b ##[1:2] c ##[0:4] d |=> a; **endproperty** : P3

The antecedent can have multiple matches, as shown in example above.

4. A sequence with an endpoint method may have multiple matches. However, a property with endpoints (e.g., **ended)** follows the rules mentioned above. Examples:

sequence qABC; a ##1 b ##[0:5] c; **endsequence** : qABC
property pSequenceEnded; **not** (qABC.**ended**); **endproperty** : pSequenceEnded

The property *pSsequenceEnded* behaves as if the endpoint has **first_match** applied to it because as soon as an endpoint of the sequence expression is determined, the evaluation of the property is TRUE, and the property completes for that thread.

property p2Ended; a |=> ##[0:3] qABC.**ended**; **endproperty** : p2Ended
The consequent endpoint of property *p2Ended* behaves as if the endpoint has an implicit **first_match** applied to it. If the antecedent is TRUE, then when the endpoint is TRUE the property completes for that thread.

property P3; qABC.**ended** |-> a; **endproperty** : P3

An antecedent endpoint may have multiple matches. When the antecedent is TRUE, the property is not necessarily completed, as it only completes if the consequent is TRUE. Thus if the antecedent qABC.**ended** has endpoints at cycle *t* and *t+1* and *a* is TRUE at cyle *t+1*, then property P3 fails at cycle *t* (qABC.**ended** is TRUE, but *a* is FALSE), and succeeds at cycle *t+1* (qABC.**ended** is TRUE, and *a* is TRUE).

Figure 4.4-1 demonstrates the evaluation of properties Z and W.

```
cycle   a   b   c   qA23B.ended   pZ = a ## [2:3] b |-> c
                    |   pZ,pW      pW = qA23B.ended |-> c
-----------------+---+----
                                 // Thread 1 2 3 4
  1     1   x   x   F   E        // property pZ and pW are in evaluation cycle
  2     0   x   x   F   E        // property pZ and pW are in evaluation cycle
  3     0   1   1   T   T        // Match in antecedent, and consequent passes (TRUE)
                                 // The endpoint for sequence "a ## [2:3] b" is TRUE
  4     0   1   1   T   T        // Match in antecedent, and consequent is evaluated
                                 // The endpoint for sequence "a ## [2:3] b" is TRUE

                                 // Thread 5 6 7 8
  5     1   x   x   F   E        // property pZ and pW are in evaluation cycle
  6     0   x   x   F   E        // property pZ and pW are in evaluation cycle
  7     0   1   1   T   T        // Match in antecedent, and consequent passes (TRUE)
                                 // The endpoint for sequence "a ## [2:3] b" is TRUE
  8     0   1   0   T   E        // Match in antecedent, and consequent fails (FALSE)
                                 // The endpoint for sequence "a ## [2:3] b" is TRUE
```

This may be unexpected, unless **first_match** is used.

E= in evaluation
T = True, completes
F = False, Fails

Figure 4.4-1 Evaluation of Properties Z and W

However, with the **first_match** operator, the following is expected:

```
property pZ_FM;
 first_match (a ## [2:3] b) |-> c;   // can have first of 2 matches, a ##2 b, or a ##3 b
endproperty : pZ

property pW_FM;
 first_match (qA23B).ended) |-> c; // Can have first of 2 matches, a ##2 b, or a ##3 b
endproperty : pW
```

Figure 4.4-2 demonstrates the evaluation of properties Z_FM and W_FM.

```
cycle  a  b  c  first_match(qA23B).ended
                 |   pZ_FM, pW_FM
-----------------+---+----
                          // Thread 1 2 3
  1    1  x  x   F   E    // property pZ_FM and pW_FM are in evaluation cycle
  2    0  x  x   F   E    // property pZ_FM and pW_FM are in evaluation cycle
  3    0  1  1   T   T    // Match in antecedent, and consequent passes (TRUE)
                          // The endpoint for sequence "a ## [2:3] b" is TRUE
  ---------------------------------------------------------------------------
  4    0  1  1   F   T    // Thread 4 holds vacuously because a == 0
                          //

cycle  a  b  c  first_match(qA23B).ended
                 |   pZ_FM, pW_FM
-----------------+---+----
                          // Thread 5, 6, 7
  5    1  x  x   F   E    // property pZ_FM and pW_FM are in evaluation cycle
  6    0  x  x   F   E    // property pZ_FM and pW_FM are in evaluation cycle
  7    0  1  1   T   T    // Match in antecedent, and consequent passes (TRUE)
                          // The endpoint for sequence "a ## [2:3] b" is TRUE
  ---------------------------------------------------------------------------
  8    0  1  0   0   T    // Thread 8 holds vacuously  because a == 0
```

Figure 4.4-2 Evaluation of Properties Z_FM and W_FM

These sequences, and the **ended** method of these sequences are shown in Figures 4.4-3 and 4.4-4. These sequences could be used as antecedents in properties. A simulation of the following property with the **ended** method confirms the unexpected failures when not using a **first_match** operation in the antecedent. The **first_match** operator is not needed in the consequent because there is an implicit **first_match** for those sequences.

Figure 4.4-3 Need of first_match in Antecedent to avoid Unexpected Failure

```
sequence qA23B;
@ (posedge clk) a ##[2:3] b;
endsequence : qA23B

property p_test_need_1stmatch;
 @ (posedge clk)
  qA23B.ended |-> c;
endproperty : p_test_need_1stmatch

ap_test_need_1stmatch : assert property (p_test_need_1stmatch);
```

Figure 4.4-4 Property Definition with no first_match in the Antecedent

Figure 4.4-5 Simulation of Property Definition with no first_match in the Antecedent

Properties without multiple matches are generally preferred due to performance considerations.

Avoid unnecessary overlapped assertions by ensuring that the triggering condition is unique. Use the "**$rose**" or "**$fell**" functions when appropriate.
If an antecedent sequence can have several matches, and it is desired to ensure a unique first match, then use the **first_match** operator.

Examples:

a ##[2:5] b |=> c ## d;

Possible unexpected errors caused by overlap of sequences.

$rose(a) ##[2:5] b |=> c ## d;

Multiple possible completion cycles, no unexpected sequence overlaps.

cy	1	2	3	4	5	6	7	8	9
a	1	0	0	0	0	0	0	0	0
b	-	-	1	1	1	1	-	-	-
c	-	-	-	1	1	1	1	-	-
d	-	-	-	-	1	1	1	1	-
assertion completes				T	T	T	T		

cy	1	2	3	4	5	6	7	8	9
a	1	0	0	0	0	0	0	0	0
b	-	-	0	1	1	0	-	-	-
c	-	-	-	-	1	1	-	-	-
d	-	-	-	-	-	1	1	-	-
assertion completes					T	T			

cy	1	2	3	4	5	6	7	8	9
a	1	0	0	0	0	0	0	0	0
b	-	-	1	1	1	1	-	-	-
c	-	-	-	1	1	0	1	-	-
d	-	-	-	-	1	1	-	0	-
assertion completes					T	T			
FAILS					F			F	

first_match($rose(a) ##[2:5] b**)** |=> c ## d;

Single completion cycle, no overlaps of sequences.

cy	1	2	3	4	5	6	7	8	9
a	1	0	0	0	0	0	0	0	0
b	-	-	1	-	-	-	-	-	-
c	-	-	-	1	-	-	-	-	-
d	-	-	-	-	1	-	-	-	-
assertion completes				T					

cy	1	2	3	4	5	6	7	8	9
a	1	0	0	0	0	0	0	0	0
b	-	-	0	1	1	0	-	-	-
c	-	-	-	-	1	-	-	-	-
d	-	-	-	-	-	1	-	-	-
assertion completes					T				

4.5 REPETITION OPERATORS

The repetition operators greatly enhance the power of SVA because they enable an expression to be repeated in different ways. There are three kinds of repetitions:

1. **SEQUENCE consecutive repetition (e.g., seq[*n], exp[*n])**, where the sequence or expression is repeated in consecutive *n* cycles. Thus, there are *n* repetitions in a path of length *n*.

2. **SEQUENCE non-consecutive repetition (e.g., seq[=n], exp[=n])**, where the sequence or expression is repeated in either consecutive or non-consecutive cycles, but the total number of repetitions is as specified. Thus, there are *n* repetitions in a path of length *m*, where *m* can be greater than *n*.

3. **SEQUENCE goto repetition, also called non-consecutive exact repetition (e.g., (seq[->n], exp[->])**, where the sequence or expression is repeated in either consecutive or non-consecutive cycles, but the last Boolean expression of the repeated sequence must occur in the cycle that immediately preceeds the expression following the **goto** repetition. Thus, the property expression " a[->2] ##1 b" is satisfied with the sequence "a ##3 a ##1 b", with last "a" immediately preceding "b".

These operators are summarized in Table 4.5, and discussed in the following subsections. A discussion of a model that demonstrates and compares all repetition operators is provided at the end of this section.

Table 4.5 Summary of Repetition Operators

Feature	Syntax	Example and Comments
Repetition		
Consecutive Fixed	[*n]	a ##1 b[*3] ##1 c // a b b b c satisfies sequence *a* holds, then starting from next cycle *b* holds for 3 cycles and then *c*
Consecutive Range	[*n:m]	a ##1 b[*1:5] ##1 c // a b b b b c, satisfies sequence *a* holds, then at next cycle *b* holds for 1 to 5 cycles until *c* is TRUE
Consecutive open-ended	[*n:$]	a ##1 b[*1:$] ##1 c // a b b ...b c, satisfies sequence *a* holds, then *b* holds until *c* is TRUE
non-consecutive	[= n]	a ##1 b[=2] ##1 c // a x x b x x x b x x c satisfies sequence *a* holds, then any 2, not necessarily consecutive, *b* occurrences, and then anytime later, *c* is TRUE . "x" is a don't care cycle. a ##1 b[=2:5] ##1 c // a x x b x b x b x x c satisfies sequence *a* holds, then any 2 to 5, not necessarily consecutive, *b* occurrences, and then anytime later, *c* is TRUE . "x" is a don't care cycle
goto repetition	[-> n]	a##1 b[->2] ##1 c // a x x b x x x b c satisfies sequence *a* holds, then any 2 not necessarily consecutive *b* occurrences, and then at next cycle, *c* is TRUE . "x" is a don't care cycle a##1 b[->2:5]##1 c // a x x b x b x b c satisfies sequence *a* holds, then any 2 to 5 not necessarily consecutive *b* occurrences, and then at next cycle, *c* is TRUE . "x" is a don't care cycle a##1 b[->1]##1 c // a x x ...b c satisfies sequence *a* holds, then any one *b* occurrence, and then at next cycle, *c* is TRUE . Same as **first_match**(a ##[0:$] ##1 b ##1 c)

4.5.1 Consecutive Repetition

To repeat a sequence, the [**range*] format is used.
A summary of the consecutive repetition operator is shown in Table 4.5-1.

Table 4.5-1 Consecutive SEQUENCE Repetition Operators

[*n]	Repeats for n cycles
[*0:$]	Repeats for zero (i.e., skipped) or any number of cycles
[*n:m]	Repeats for minimum of n to a maximum of m cycles
[*0: m]	Either skipped or repeats a maximum of m cycles
[*n:$]	Repeats for a minimum of n cycles

4.5.1.1 [*n] Repetition

SVA handles the fixed sized repetition construct "Sequence_Expression [*n]" differently than the other constructs that address ranges. With the fixed sized repetition construct [*n], one is declaring that the sequence expression must repeat exactly that number of times, with a delay of one clock tick from the end of one match to the beginning of the next, as shown in Table 4.5.1.1. The sequence expression can be a sequence of one term (e.g., *a*) or a sequence either expressed as a named sequence, or a sequence expressed in parentheses. For example the sequence (a ##1 b ##2 c) [*3] is equivalent to the sequence:

(a ##1 b ##2 c) ##1 **(a ##1 b ##2 c)** ##1 **(a ##1 b ##2 c)**

Table 4.5.1.1 Example for Fixed Sized Repetition Operator [*n]

SEQUENCE	Sequence that satisfies SEQUENCE Clock edge	1	2	3	4
a ##1 b[*2]##1 c	a	1	-	-	-
	b	-	1	1	-
	c	-	-	-	1

Entry	Meaning
-	no effect on the assertion on that clock edge
1	signal is TRUE on that clock edge

4.5.1.2 [*n:m] Repetition

The repetition range constructs specify that the SEQUENCE must be repeated within the specified range. If there is a concatenated sequence that follows the repeat construct, then there is an implicit until the first element of the concatenated sequence occurs. The following example with the overlapped implication operator "|->" clarifies the repetition construct with a range:

(a |-> (b[*1 : 3] ##1 (c ##1 d));

The [*min : max] declares that the element in the SEQUENCE must repeat a "min" times, or "min+1" times, or "min+2" times, or ... "max" times. In that example, b[*1 : 3] means that *b* may be repeated once, twice, or three times. The (b[*1 : 3] ##1 (c ##1 d)) has an implicit until because of the next concatenation expression (the *c*), meaning that *b* needs to hold a maximum of three cycles until *c* occurs. The expression *d* is only needed to complete the sequence. As a result, this construct creates a set of different sequences that can occur. The set of tables that describes the expected behavior for this assertion (up to the *c*) is shown below.

Entry	Meaning
-	no effect on the assertion on that clock edge
1	signal was TRUE on that clock edge

```
property pTEST1to3_OVERLAPPED_IMPLICATION;
@ (posedge clk) a |-> (b[*1 : 3] ##1 c);
// If b ==1 then equivalent property specification is:
// @ (posedge clk) a |-> ##[1 : 3] c;
endproperty : pTEST1to3_OVERLAPPED_IMPLICATION
```

Clock ticks

	1	2	3	4
a	1	-		
b	1	-		
c	-	1		

repetition of 1
This satisfies *a |-> (b[*1] ##1 c);*

Clock ticks

	1	2	3	4
a	1	-	-	
b	1	1	-	
c	-	0	1	

repetition of 2
This satisfies *a |-> (b[*2] ##1 c);*

Clock ticks

	1	2	3	4
a	1	-	-	-
b	1	1	1	-
c	-	0	0	1

repetition of 3
This satisfies *a |-> (b[*3] ##1 c);*

Clock ticks

	1	2	3	4
a	1	-	-	-
b	1	1	0	-
c	-	-	0	1

when *b* is FALSE, *c is* expected to be TRUE
This FAILs *a |-> (b[*1 : 3] ##1 c);*

Clock ticks

	1	2	3	4
a	1	-	-	-
b	1	1	1	-
c	-	-	-	0

c is expected to be TRUE after 3 consecutive cycles of *b*
This FAILs *a |-> (b[*1 : 3] ##1 c);*

Clock ticks

	1	2
a	1	
b	0	
c	-	

when *a* is TRUE, *b is* expected to be TRUE in same cycle
This FAILs *a |-> (b[*1 : 3] ##1 c);*

Note that in this property, if ***a*** is TRUE *b* must be TRUE on the same clock edge because the overlapped implication operator "|->" is used.

The next example is a variation of the previous assertion with the non-overlapped implication operator "|=>":

```
property pTEST1to3_NON_OVERLAPPED_IMPLICATION;
 @ (posedge clk) a |=> (b[*1 : 3] ##1 c);
 // If b ==1 then equivalent property specification is:
 // @ (posedge clk) a |=>  ##[1 : 3] c;
endproperty : pTEST1to3_NON_OVERLAPPED_IMPLICATION
```

Clock ticks

	1	2	3	4	5
a	1	-	-		
b	-	1	-		
c	-	-	1		

repetition of 1
This satisfies *a |=> (b[*1] ##1 c);*

Clock ticks

	1	2	3	4	5
a	1	-	-	-	
b	-	1	1	-	
c	-	-	0	1	

repetition of 2
This satisfies *a |=> (b[*2] ##1 c);*

Clock ticks

	1	2	3	4	5
a	1	-	-	-	-
b	-	1	1	1	-
c	-	-	0	0	1

repetition of 3
This satisfies *a |=> (b[*3] ##1 c);*

Clock ticks

	1	2	3	4	5
a	1	-	-	-	-
b	-	1	1	0	-
c	-	-	-	0	

when *b* is FALSE, *c is* expected to be TRUE
This FAILs *a |=> (b[*1 : 3] ##1 c);*

Clock ticks

	1	2	3	4	5
a	1	-	-	-	-
b	-	1	1	1	-
c	-	-	0	0	0

c is expected to be TRUE after 3 consecutive cycles of *b*
This FAILs *a |=> (b[*1 : 3] ##1 c);*

Clock ticks

	1	2
a	1	-
b	-	0
c	-	-

when *a* is TRUE, *b is* expected to be TRUE in next cycle
This FAILs *a |=> (b[*1 : 3] ##1 c);*

Note that in this property, if *a* is TRUE, *b* must be True on the next clock edge because the non-overlapped implication operator "|=>" is used.

4.5.1.3 [*0 : m] Repetitions

The [*0:m] repetition declares that the element in the SEQUENCE must repeat "0" times (i.e., can be skipped altogether) , or "1" times, or "2" times, or ... "m" times. The example, *b*[*0:3] means that *b* may either be skipped (i.e., no cycle spent on *b*), or *b* may be repeated once, twice, or three times. The sequence (b[*0:3] ##1 c) has an implicit until because of the next concatenated expression, meaning that if *c* holds at the start of this SEQUENCE, then a check on *b* is skipped altogether, as if it were not there. However, if *c* does not hold at the start of the SEQUENCE, then *b* needs to hold a maximum of three cycles until *c* occurs. As a result, this construct creates a set of different sequences that can occur. The set of tables that describes the expected behavior for this assertion is shown below.

```
property pTEST0to3_OVERLAPPED_IMPLICATION
 @ (posedge clk) a |-> (b[*0:3] ##1  c);
 // If b ==1 then equivalent property specification is:
 //  @ (posedge clk) a |->  ##[0 : 3] c;
endproperty : pTEST0to3_OVERLAPPED_IMPLICATION
     Clock ticks
    1  2  3  4
 a  1          This table is for a |-> (b[*0] ##1  c);
 b  -          When the repetition is 0, the assertion is reduced to:
 c  1                                  (a) |-> (c);
     Clock ticks
    1  2  3  4
 a  1  -
 b  1  -       This table is for a |-> (b[*1] ##1  c);
 c  0  1
     Clock ticks
    1  2  3  4
 a  1  -  -
 b  1  1  -    This table is for a |-> (b[*2] ##1  c);
 c  0  0  1
     Clock ticks
    1  2  3  4
 a  1  -  -  -
 b  1  1  1  - This table is for a |-> (b[*3] ##1  c);
 c  0  0  0  1
     Clock ticks
    1  2  3  4
 a  1  -  -  -
 b  0          This FAILs a |-> (b[*0:3] ##1  c);
 c  0          If  b is False then c should be True
     Clock ticks
    1  2  3  4
 a  1  -  -  -
 b  1  1  0    This FAILs a |-> (b[*0:3] ##1  c);
 c  -  -  0    Expected c to be True at cycle 4
```

Note that in this property, if *a* is TRUE, then *b* or *c* must be 1 on the same clock edge. This is because the overlapped implication "|->" is used.

This next example is a variation of the previous property with the non-overlapped implication operator "|=>" and clarifies the repetition construct with a range:

```
property pTEST0to3_NOV_IMPLICATION;
 @ (posedge clk) a |=> (b[*0:3] ##1 c);
 // If b ==1 then equivalent property specification is:
 // @ (posedge clk) a |=> ##[0 : 3] c;
endproperty : pTEST0to3_NOV_IMPLICATION
    Clock ticks
    1 2 3 4 5
 a  1 - -        This table is for a |=> (b[*0] ##1 c);
 b  - -          When the repetition is 0, the assertion is reduced to:
 c  - 1                                    (a) |=> (c);

    Clock ticks
    1 2 3 4 5
 a  1 - -
 b  - 1 -        This table is for a |=> (b[*1] ##1 c);
 c  - 0 1

clk 1 2 3 4 5
 a  1 - - -
 b  - 1 1 -      This table is for a |=> (b[*2] ##1 c);
 c  - 0 0 1

clk 1 2 3 4 5
 a  1 - - - -
 b  - 1 1 1 -    This table is for a |=> (b[*3] ##1 c);
 c  - 0 0 0 1

    Clock ticks
    1 2 3 4
 a  1 - - -
 b  - 0          This FAILs a |=> (b[*0:3] ##1 c):
 c  - 0          If b FALSE then expected c True

    Clock ticks
    1 2 3 4
 a  1 - - -
 b  - 1 1 0      This FAILs a |=> (b[*0:3] ##1 c);
 c  - - - 0      Expected either b True and c False, or c True
```

4.5.1.4 [*n : $], ##[0:$]

The [*n:$] constructs is similar to the [*n:m] construct, except that *m* is infinite, instead of being bounded by an integer number, and means that the repetition applies for a minimum of n cycles.

The ##[0:$] represents a delay ranging from zero (i.e., no delay) up to infinity until the sub-sequence following is met. If that sub-sequence is met, the main sequence completes.

The ##[0:$] delay or `*true*[*0:$] repeat operators take special meaning in the consequent portion of a property by causing the property to never fail, but only complete. This is very useful while capturing functional coverage using SystemVerilog Assertions. The following is a non-intuitive example that clarifies the application of the ##[*:$] or its equivalent `*true*[*0:$].

```
(a) |=> (##[0:$] ##1 b[*0:$] ##1 c);
// same as above
(a) |=> (##[0:$] `true ##1 b[*0:$] ##1 c);
(a) |=> (##[0:$] ##1 (b[*0:$] ##1 c) );
// same as above
(a) |=> (##[0:$]  (b[*0:$] ##1 c) ); // (##[0:$] ##1) is same as (##[0:$])
```

If *a*. then wait until sequence (b[*0:$] ##1 c) is true. **Property cannot fail!**

- If the antecedent is TRUE , then starting from next cycle the consequent is checked.
- In the consequent, the first element of the sequence ##[0:$]states that anything can happen for any number of samples until the remainder of that sequence (i.e., b[*0:$] ##1 c) is TRUE. If that remainder is FALSE, then the test is repeated at every cycle until it is TRUE. This means **there is no entry in the sequence following the ##[0:$] that can cause the assertion to fail.** This style of SEQUENCE is useful in applications where functional coverage is desired to determine if a sequence ever happened.
- The rest of the elements in the SEQUENCE are only used to determine when the sequence completes. Remember that a property with that sequence cannot fail because of the first ##[0:$] in the consequent.
- The second element in the consequent sequence, *b*[*0:$], states *b* may happen for any number of samples. (*b*[*0:$] ##1 *c*) sequence is satisfied if *b* is TRUE followed by *c*, or if *b* is FALSE (i.e., zero times because of the 0 in the range) followed by *c*. Thus, the 0 makes this path optional, and as a result, the assertion could be written as follows:
  (a) |=> (##[0:$] ##1 c); // same as: (a) |=> (##[0:$] ##1 b[*0:$] ##1 c);
- The third element in the sequence, *c*, will only indicate when the sequence is complete.

  Note: The use of "##[0:$]" does have implications on performance because if the antecedent succeeds once, and the consequent never succeeds, then at every cycle the simulator must test the consequent threads to verify possible success. If a thread fails, the property does not complete, but other consequent threads may succeed.

Consider a different property:

(a) |=> (##[0:$] ##1 b[*1:$] ##1 c);

- If *a* is True, then the property starts looking for the sequence specified on the right-hand side.
- On the right-hand side, the ##[0:$] element will create a SEQUENCE that cannot fail.

The sequence "*b* followed by *c*" will cause the sequence to finish. Note that *b* may occur at any time, but the sequence "*b ##1 c*" completes the property. Because of the repetition operator [*1:$], *b* must occur at least once. The sequence "*b[*1:$] ##1 c*" means that for the sequence to complete, *b* is TRUE and stays TRUE until *c* occurs one cycle after the last *b*. However, because of the ##[0:$], the property can be simplified as:

(a) |=> (##[0:$] ##1 b ##1 c); // same as: (a) |=> (##[0:$] ##1 b[*1:$] ##1 c);

Examples with explanation of the syntax and references to models are shown in Table 3.11.1.4.

Table 4.5.1.4 Examples of Consecutive SEQUENCE Operators

All signals active on rising edge.

SEQUENCE	EFFECT	EXAMPLE
expr[*n]	*expr* holds in exactly n consecutive cycles	a[*3] |=> ##4 b; If *a* holds for 3 consecutive cycles then after 4 cycles (starting count from the cycle after *a* last occurred), *b* occurs. (This is 5 cycles after last *a* occurrence because of the "|=>" operator, which counts as one cycle) Thus, the following sequence is satisfied: Cycle | | | | | | | | a 1 1 1 – – – – – b – – – – – – – 1
SEQUENCE[*n]	SEQUENCE repeats *n* times	(a ##1 b)[*2] |=> (c ##1 d)[*3]; If sequence "*a, b, a, b*" occurs, then the sequence "*c, d, c, d, c, d*" must occur, starting from the cycle after second *b* occurred.
expr [*n:m]	*expr* holds in at least *n* consecutive cycles, but in no more than m cycles.	(a) |=> (b[*1:4] ##1 c ## [*2: 3] d); If *a* holds, then starting from the cycle after *a* last occurred, *b* must hold from 1 to 4 cycles until *c* occurs. Thereafter, *d* should occur between 2 to 3 cycles after *c*. The following sequences are satisfied[8]: <-one sequence-> <other sequence> a |=> b[*2] ##1 c ##2 c ‖ a |=> b[*1] ##1 c ##2 d a 1 – – – – – ‖ 1 – – – – b – 1 1 – – – ‖ – 1 – – – c – – 0 1 – – ‖ – – 1 – – d – – – – – 1 ‖ – – – – 1

[8] The sequences do not show all permutations

expr [*0:m]	Either the cycle is skipped (i.e., ZERO times) if the next concatenated expression holds, or *expr* holds between 1 to *m* cycles until the next concatenated expression holds.	(a) |->(b[*0:4] ##1 c); IF *a* holds, then in the current cycle, *b* needs not necessarily happen if *c* occurred. If *c* does not occur, then *b* must hold for no more than 4 cycles (starting from cycle when *a* occurs) until *c* occurs. Thus the following 2 sequences are satisfied (notice the |->, instead of |=>) <-one sequence-> || <another sequence> a |-> b[*0] ##1 c || a |-> b[*3] ##1 c a 1 || 1 – – – – b – || 1 1 1 1 – c 1 || 0 0 0 0 1
expr [*n:$]	*expr* holds in at least *n* consecutive cycles. Like *expr* [*n:m] where *m* is infinity.	(a) |=> (b[*1:$] ##1 c); If *a* holds then anytime, starting from the next cycle, *b* must hold indefinitely until *c*. When *c* holds, *b* need not hold. Thus the following 2 sequences are satisfied: <-one sequence-> || <another sequence> a |=> b[*1] ##1 c || a |=> b[*2] ##1 c a 1 – – || 1 – – – b – 1 – || – 1 1 – c – – 1 || – – 0 1
expr[*0:$]	Either the cycle is skipped if the next concatenated expression holds (i.e., ZERO times), or *expr* holds until the next concatenated expression holds.	(a) |=> (b[*0:$] ##1 c); IF *a* holds, then in the next cycle, *b* need not necessarily happen if *c* occurred. If *c* does not occur, then b must hold until *c* occurs. Thus the following 2 sequences are satisfied: <-one sequence-> || <another sequence> a |=> b[*0] ##1 c || a |=> b[*2] ##1 c a 1 – || 1 – – – b – – || – 1 1 – c – 1 || – 0 0 1
expr [*1:$]	A must hold throughout the sequence until the next concatenated expression holds.	(a) |=> (b[*1:$] ##1 c); If *a* occurs, then *b* should occur starting in the next cycle for one or more cycle, until *c occurs*. Thus the following 2 sequences are satisfied: <-one sequence-> || <another sequence> a |=> b[*1] ##1 c || a |=> b[*2] ##1 c a 1 – – || 1 – – – b – 1 – || – 1 1 – c – – 1 || – – 0 1

4.5.2 Sequence Non-consecutive Repetition ([=n])

The SEQUENCE non-consecutive repetition operator ([=n]) is similar to the consecutive repetition operator, with the exception that the expression is repeated either in consecutive or non consecutive cycles, with the total number of repetitions exactly as specified by the operands of the "[=*n*]" operator. Thus, there are exactly *n* repetitions in a path of length *m*, where *m* can be greater than *n*. For example:

```
property pREPT_EQUAL;
 @ (posedge clk) (a) |=> (b[=2] ##1 c);
endproperty : pREPT_EQUAL
```

That property states that if *a* then, starting from next cycle, there should be two occurrences of *b* either consecutive or non-consecutive, and then some time later (i.e., next cycle or later) *c* occurs. The following example meets an assertion of the above property.

```
cycle    1 2 3 4 5 6 7 8 9     (a) |=> (b[=2] ##1 c));
a        0 1 - - - - - - -
b        - - 0 1 0 0 1 0 -
c        - - - - - - - 0 1
```

Note: if after the second occurrence of *b* a new *b* occurs (and NO *c*), the property FAILS. However, if after the second occurrence of *b*, *c* occurs anytime before a new *b*, then the property PASSES.

4.5.3 Sequence goto Repetition ([->n])

The SEQUENCE **goto** repetition operator ([->n]) is similar to the [=n] operator where the expression is repeated in either consecutive or non-consecutive cycles, but the Boolean expression must hold on the last cycle of the expression being repeated.

For example:

```
property pREPT_ARROW;
 @ (posedge clk) (a) |=> (b[->2] ##1 c);
endproperty : pREPT_ARROW
```

That above property states that if *a* then, starting from next cycle, there should be two occurrences of *b*, which can be consecutive or non-consecutive. However, *c* must occur in the cycle following the last occurrence of *b* (note the "[->*n*]" construct). The following example meets an assertion of the above property.

```
cycle    1 2 3 4 5 6 7 8 9     -- (a) |=> (b[->2] ##1 c));
a        0 1 0 0 0 0 0 0 0
b        - - 0 1 0 0 1 - -
c        - - - - - - - 1 -
```

By contrast, the following property states that if *a* then, starting from next cycle, there should be two consecutive occurrences of *b*, followed by an occurrence of *c*.

```
property pREPT_STAR;
 @ (posedge clk) (a) |=> (b[*2] ##1 c);
endproperty : pREPT_STAR
```

Figure 4.5.3-1 and 4.5.3-2 shows a timing diagram for the above three assertions.

Figure 4.5.3-1 Timing Diagram for Assertions with Repetition Operators

Figure 4.5.3-2 Timing Diagram for Assertions with Repetition Operators

4.6 SEQUENCE COMPOSITION OPERATORS

Sequences are one of the more powerful features of an assertion language such as SystemVerilog Assertion. Basic protocol checkers can easily be built using these sequences. However, complex protocols/scenarios require the ability to describe or specify interactions between sequences, such as choice or inclusion of sequences. Building complex sequences using only implication and cycle delay operators creates code that is difficult to express and understand. However, SystemVerilog provides sequence composition operators to combine individual sequences in a variety of ways that enhance code writing and readability.

Because this is an advanced feature of the language, a brief overview of the available sequence composition operators is provided. This is then followed by examples.

The sequence composition operators specify the relationships between sequences. These operators include:

- **##0 : sequence overlapping concatenation (aka fusion).** The ##0 constructs a sequence in which two sequences overlap by one cycle.

- **or : sequence or-ing (aka disjunction).** The sequence **or** operator constructs a composite sequence that holds TRUE if either of the two operand sequences hold at the current cycle.

- **and : non-length-matching sequence conjunction.** The binary operator **and** is used when both operands are expected to match, but the end times of the sequences can be different.

- **intersect : length-matching sequence conjunction.** The binary operator **intersect** is used when sequences start in the same cycle AND end in the same cycle.

- **within : containment of sequences.** The **within** operator defines the containment of a sequence within another sequence (see section 4.6.5 for more discussion on the **within**).

- **throughout :** The **throughout** operator is used to check for a Boolean condition to be TRUE during a sequence. Though this operator is not strictly a sequence composition operator (i.e., it doesn't operate on two operand sequences), it is described here to maintain consistency. see section 4.6.6 for more discussion on the **throughout**).

4.6.1 Sequence Fusion (##0)

The fusion "##0" operator represents an overlapping concatenation in sequences. That is, the last sub-element of the first-sequence and the first sub-element of the second sequence hold TRUE in the same cycle. This is another way of stating that the second sequence starts at the cycle in which the first sequence completes. For example, the sequence **(a ##1 b) ##0 (c ##1 d)** states that *a* occurs at cycle **t,** *b* and *c* occur at cycle **t+1** (notice the overlap), and *d* occurs at cycle **t+2** as shown in Figure 4.6.1.

Figure 4.6.1 Fusion Example for Expression (a##1 b) ##0 (c##1 d)

4.6.2 Sequence Disjunction (or)

The *sequence or* operator (**or**) constructs a sequence in which one of two alternative sequences hold at the current cycle. Thus, the sequence **(a ##1 b) or (c ##1 d)** states that either sequence *a, b* or sequence *c, d* would satisfy the assertion, as shown in Figure 4.6.2. When one of the sequences completes successfully, the whole sequence (i.e., (a ##1 b) **or** (c ##1 d)) is considered ended, and a property that uses that whole sequence is considered successfully terminated.

Figure 4.6.2. Sequence Disjunction Example for Expression (a ##1 b) or (c ##1 d)

4.6.3 Sequence Non-Length-Matching (and)

The *sequence non-length-matching* **and** operator constructs a SEQUENCE in which two sequences both hold at the current cycle, regardless of whether they complete in the same cycle or in different cycles. For example, the sequence **(a ##[0:$] b) and (c[*1:5] ##1 d)** states that both sequences must start at the same time (i.e., *a* and *c* occur), but the sequences may complete at different cycles, as shown in Figure 4.6.3.

Figure 4.6.3 Sequence Non-Length-Matching AND for Expression (a ##[0:$] b) and (c[*1: 5] ##1 d)

4.6.4 Sequence Length-Matching (intersect)

The *sequence length-matching* **intersect** operator constructs a SEQUENCE like the **and** non-length-matching operator, except that both sequences must complete in same cycle. For example, the sequence **(a ##[0:$] b) intersect (c[*1: 5] ##1 d)** states that both sequences must start at the same cycle and must complete at the same cycle, as shown in Figure 4.6.4.

Figure 4.6.4 Sequence Length-Matching intersect for Expression (a ##[0:$] b) intersect (c[*1:5] ##1 d)

4.6.5 Sequence Containment (within)

A visualization of a sequence containment composition is shown in Figure 4.6.5 wherein seq1 is contained "within" seq2.

Figure 4.6.5 Sequence Containment (within) for seq1 within seq2

The composite sequence seq1 **within** seq2 matches along a finite interval of consecutive clock ticks provided seq2 matches along the interval and seq1 matches along some sub-interval of consecutive clock ticks.

That is, the matches of seq1 and seq2 must satisfy the following:

- The start point of the match of seq1 must be no earlier than the start point of the match of seq2.
- The end point of the match of seq1 must be no later than the end point of the match of seq2.

A practical example of a DMA transfer is shown below:

```
property pWITHIN;
@ (posedge clk)
```

Users may want to constrain the range of cycles prior to the *dma_req*. e.g. ([*1:3] ##1 $rose(dma_req); ([*0:3] ##1 data_accepted)[*2]);

```
  (## [0:$] $rose(dma_req) ##1 (##[0:3] ##1 data_accepted)[*2]) within
   (go ##[1:20]  done_dma);
 endproperty : pWITHIN
ap_within_1: assert property (pWITHIN);
```

An alternate method to write the above property uses the repetition operator (`true [*0:$]), as shown below:

```
property p2within;
@ (posedge clk)
   (`true [*0:$] ##1 $rose(dma_req) ##1 (`true[*0:3] ##1 data_accepted)[*2]) within
    (go ##[1:20]  done_dma);
endproperty : p2within
p2within_2: assert property (p2within);
```

This property, as shown in Figure 4.6.5-1 and 4.6.5-2, states that within the window of time starting with a GO signal (*go*) and ending by the slave asserting the done DMA flag (*done_dma*), the DMA controller must transfer 2 words of data using the handshake sequence data ready (*dma_req*) and data accepted (*data_accepted*) from a slave unit.

Figure 4.6.5-1 DMA Transfer Property with the within Operator

Figure 4.6.5-2 DMA Transfer Property with the within Operator with Failure

4.6.6 Conditions over Sequences (throughout operator)

The **throughout** operator specifies that a signal must hold throughout a sequence. For example, in the following handshake example, a *request* is honored with an *acknowledge* sometime within the next cycle and up to 4 cycles later. The *acknowledge* is then followed at a later time by a *done* control signal. However, there is one additional requirement that throughout the time between the *acknowledge* and *done* an *enable* signal must be asserted to drive a tri-state bus. That property can be expressed as follows. Figure 4.6.6 demonstrates a timing diagram that satisfies an assertion of this property.

```
property p_handshake;
  @ (posedge clk)
  $rose(req) |-> ##[1:4] enable throughout (ack ##[1:$] done);
endproperty : p_handshake
```

Figure 4.6.6 Timing Diagram of Property Expression
"**$rose**(req) |-> ##[1:4] enable **throughout** (ack ##[1:$] done);"

4.7 METHODS SUPPORTING SEQUENCES

SystemVerilog provides a built-in package that contains system types (e.g., classes), variables, tasks and functions. To support sequences, SystemVerilog provides the following methods:

1. **ended** endpoint detection in single clock assertions
2. **matched** endpoint detection in multi-clocks assertions
3. **triggered** Level-sensitive control as a result of sequence terminations.

4.7.1 Endpoint of a Single Clock Sequence "ended"

SystemVerilog sequences can express a wide variety of complex temporal expressions through the use of cycle-range and repetition operators (e.g., "a ##1 b ##[0:4] c ##1 d[*3]"). Sequences terminate at a cycle within a window of clock ticks. It is often important to know when a sequence thread is ending. Such information can be used to build other sequences that rely on one sequence's end point.

The **ended** method applies to a single clocked instantiated sequence, and is TRUE when the instantiated sequence completes. The **ended** method shall not be used in a **disable iff** statement. It is applied to an instantiated sequence though the use of the name of the sequence, actual arguments, followed by "**.ended**". For example, consider a named sequence qWrDmaBlock256Words that writes a block of 256 words into the memory. The writes to the memory need not be consecutive, but the total number of words is 256.

```
sequence qWrDmaBlock256Words (dma_ack, rd_wrn, dma_enb, done);
  @ (posedge clk)
  $rose(dma_ack) ##1 (!rd_wrn && dma_enb)[=256] ## [0:100] done; // active low write
endsequence : qWrDmaBlock256Words
```

Now a property can be built that states that when a block of DMA transfer completes, then an interrupt should occur with 5 cycles.

```
property p_dma_block_then_interrupt;
  @ (posedge clk)
    qWrDmaBlock256Words (dma_ack, cpu_rd_wrn, dma_enb, dma_done).ended |->
      ##[1:5] cpu_interrupt[DMA]; // DMA = integer representing the DMA interrupt number
endproperty : p_dma_block_then_interrupt
```

An endpoint of a sequence, expressed with the instantiation of a sequence (i.e., name and actual arguments) post-fixed with ".**ended**", is a call to a method that returns a Boolean to identify if at that cycle the sequence has reached its completion. This is different than the instantiation of the sequence, which starts the temporal execution of the sequence. Endpoints are not a replacement for the sequence and is not a sequence, but rather a Boolean condition that represents the termination of that sequence. For example:

sequence qReqAck; req ##[0:100] ack; **endsequence** : qReqAck

property p_prop1; go |=> qReqAck ##1 done; **endproperty** : p_prop1

same as:

This is a sequence that **lasts several cycles**

property p_same; go |=> req ##[0:100] ack ##1 done; **endproperty** : p_same

This is a test that sequence has ended, and test **lasts one cycle**

But different than:

property p_diff_than_prop1; go |=> qReqAck.**ended** ##1 done;
endproperty : p_diff_than_prop1

The Boolean endpoint is expressed with the **ended** method operated on a sequence. It represents, at every cycle, information about the termination of a sequence. The end point of a sequence is reached whenever the ending clock tick of a match of the sequence is reached, regardless of the starting clock tick of the match. Figure 4.7.1-1 demonstrates two endpoint cases for the sequence *qReqAck*. It also demonstrates that an assertion of property *p_prop1* fails when *req* is not asserted after *go* (at time 300 ns), yet property *p_diff_than_prop1* successfully completes because in the cycle following *go* qReqAck.**ended** is TRUE (at time 350 ns).

Figure 4.7.1-1 Endpoint and Properties

Referring to the Past Using endpoints

Endpoints of sequences (with the **ended** method) are sometimes useful when there is a need to refer to a past event.[9] For example:

```
sequence qReq:
 $rose(req) ##3 true; // req and 3 cycles later TRUE.
endsequence : qReq
```

If *req* is high, then 3 cycles later *qReq*.**ended** is TRUE. Using this information, it becomes possible to express the notion that if *ack* is active, then *req* was active 3 cycles ago:

```
property pIfAckThenPreviousRequest;
  ack |-> qReq.ended; // equivalent to req |=> ##2 ack;
endproperty : pIfAckThenPreviousRequest
```

The timing diagram for the endpoint *qReq*.**ended** and the property *pIfAckThenPreviousRequest* is shown in Figure 4.7.1-2.

Figure 4.7.1-2 Timing Diagram for qReq.ended and property *pIfAckThenPreviousRequest*

An example application of endpoint detection can be in a write-back policy based cache coherency controller. One a write request to an address is found in the cache (i.e. a write hit), the following sequences of events should occur (possibly at varied time intervals):

- The line is marked as "dirty".
- Any other caches having a copy of this address shall mark their copies as invalid.
- Eventually, the dirty line shall be cleaned – i.e. modified data shall be written back to main memory.

---

[9] http://www.pslsugar.org/papers/sitvanit.pdf
Translated to SVA from the PSL Practitioner's Handbook: Effective Specification Techniques
Sitvanit Ruah

As it is evident, all the above 3 requirements have a common starting point – i.e. a detection of a write-hit. A good methodology would be to have the write-hit identified once and all the 3 consequent requirements start once the *qWriteHit* sequence ends.

Intuitively, one may wish to utilize the **ended** of a reset sequence to disable any valid assertions. However, the **ended** cannot be used in a **disable** block. The **ended** method has no meaning outside the narrow "observe" region and does not generate any Verilog event.[10] The **ended** can only be used in the assertion expressions for detecting the end of a sequence in any particular cycle. The **ended** method cannot be used in a clocking expression. However, SystemVerilog provides the **triggered** method to detect a persistent endpoint of a sequence that lasts until the end of a clock evaluation cycle (see section 4.7.3). The **triggered** endpoint of a sequence can be used as a level-sensitive event in the **disable iff** component of a property.

4.7.2 Endpoint of a Multi-Clock Sequence "matched"

As mentioned in the previous section, the validity of the result of the **ended** method of a sequence is a narrow window in the "observe" region. This would lead to the result of the **ended** completely being missed in a sequence involving multiple clocks. In multiple clocked sequences, the endpoint of one sub-sequence can happen in one clock domain while the start of the next sub-sequence can happen in another clock domain. In such cases, the result of the **ended** method needs to be latched and synchronized till the destination clock. SystemVerilog provides such a method, called **matched.** The **matched** method indicates the endpoint of a sequence in a multi-clock context.[11] To detect the endpoint of a sequence when the clock of the source sequence is different than the destination sequence, the method **matched** on the source sequence is used. When the endpoint of a sequence is reached, that result is stored and synchronized to the first destination clock tick after the match. For example,

```
sequence q_e1(a,b);
 @(posedge clk) $rose(a) ##[2:3] b;
endsequence : q_e1

sequence q_e2;
 @(posedge sysclk) c ##1 q_e1(a, b).matched [->1] ##1 d;  endsequence : q_e2
```

Using *sysclk* clock

Using *clk* clock

In this example, source sequence *q_e1* is evaluated at clock *clk*, while the destination sequence *q_e2* is evaluated at clock *sysclk*. In *q_e2*, the endpoint of the instance *q_e1(a, b)* is tested to occur sometime after the occurrence of *c*. Method **matched** only tests for the end point of *q_e1(a, b)* and has no bearing on the starting point of *q_e1(a, b)*. Figure 4.7.2 demonstrates the timing for sequence *q_e2* with the occurrence of *q_e1(a, b)*.**ended** and *q_e1(a, b)*.**matched**. Notice that the endpoint *q_e1(a, b)*.**ended** (clocked with clk) is synchronized to the *sysclk* to represent *q_e1(a, b)*.**matched.**

Note that a sequence may have several endpoints, depending upon how it is expressed and the value of the signals it is sensitive to. For example, consider a sequence *q*_e1 and a signal pattern defined as "a ##2 b ##1 b" e.g., *a* @ cycle 1, *b* @cycle 3, *b* @cycle 4). With that stimulus sequence, *q_e1(a,b)* would have two matches: one match being "*a ##2 b*" and the other match being "*a ##3 b*".

---

[10] See section 5.10 SystemVerilog Scheduling Semantics for Assertions

[11] See section 5.9 for discussion of multi-clocked sequences.

Figure 4.7.2 Difference between endpoints ended versus matched

An example of a property using sequences *q_e1* and *q_e2* might be:

property pE1E2;
  @ (**posedge** sysclk)
    go |=> q_e2;
endproperty : pE1E2
pE1E2: **assert property** (pE1E2);

Figure 4.7.2 only demonstrates the consequent *q_e2*.

4.7.3 Triggered Method

The built-in method **triggered** returns the current end status of an instantiated sequence. The **triggered** method provides level-sensitive controls of properties and procedural code as a result of sequence terminations. Examples of actions taken as a result of termination of sequences include:

1. Execution of procedural code delayed until termination of a sequence. This is accomplished using the level-sensitive **wait** statement in conjunction with the **triggered** method.
2. Disabling a property based on termination of a sequence.

The **triggered** sequence method evaluates to TRUE if the given sequence has reached its endpoint at that particular point in time (in the current time-step), and FALSE otherwise. The triggered status of a sequence is set during the Observe region and persists through the remainder of the time-step (i.e., until simulation time advances). Thus, the **triggered** method is an event for each declared sequence (see section 4.7.3). That event can be used as a level-sensitive event in a **wait** statement or in the **disable iff** component of a property. For example:

*// model using the **wait** statement and an immediate assertion*
module check_req_ack_done (**input** req, ack, done_xfr, clk);
sequence qReq2send;
  @(**posedge** clk) req ##[0:$] ack ##2 `true;
endsequence : qReq2send

```
always @ (posedge clk) begin
 wait(qReq2send.triggered);     // check of endpoint for sequence qReq2send
 assert_end_xfr : assert (done_xfr) else
        $display ("Incorrect data transfer, done_xfr not received");
 end
endmodule : check_req_ack_done
```

In the above example, the **always** block in module *check_req_ack_done* waits for the endpoint (**triggered**) of the sequence *qReq2send*. When the condition evaluates to TRUE the **wait** statement unblocks the process, which evaluates an immediate assertion, expecting *done_xfr* to be TRUE.

The **triggered** method can be used in a **disable iff** expression instead of the **ended** method. One of the reasons for the limitation that the expression of **disable iff** may not contain "**ended**" is that this expression is evaluated asynchronously during simulation. Since "**ended**" is not allowed and has no meaning outside of the narrow "observe" region, one cannot evaluate such an expression asynchronously. However, the **triggered** method provides an event for each declared sequence, and that can be used as a level-sensitive event in the **disable iff**. For example:

```
sequence qAbort;
  @ (posedge clk) (reset || cancel) or (dma_req ##[0:$] dma_ack);
endsequence : qAbort

property p_cpu_data_trasnsfer;
  @ (posedge clk) disable iff (qAbort.triggered)
    cpu_rd |-> ##[1:4]  mem_data_available;
endproperty : p_cpu_data_trasnsfer
ap_cpu_data_trasnsfer: assert property p_cpu_data_trasnsfer;
```

Property *cpu_data_trasnsfer* states that the verification is cancelled if either there is a reset, a cancel, or the sequence *dma_req* is eventually followed by a *dma_ack*. Otherwise, a CPU READ must be followed within four cycles by a memory data available.

4.7.4 Sequence events

A sequence instance can be used in event expressions to control the execution of procedural statements based on the successful match of the sequence. This allows the endpoint of a named sequence to trigger multiple actions in other processes. When a sequence instance is specified in an event expression, the process executing the event control is blocked until the specified sequence reaches its end-point. A process resumes execution following the Observe region in which the end point is detected. For example:

```
sequence qDoneSetup;
  @ (posedge clk) ##[0:$] load_mem ##[0:1000] done;
endsequence : qDoneSetup

initial begin
 @ qDoneSetup begin
    $display( "done memory load" );
    ready <= 1'b1;
    end
 end
```

In the example above, when the named sequence *qDoneSetup* reaches its end point, the **initial** block is unblocked. In this case, the end of the sequence acts as the trigger to unblock the event.

EXERCISES

1. Define the following two named sequences:
   a. qReq2Data: *read_req* followed within 4 cycles by *data_available*.
   b. qXmit2Done : *Start_xmit* followed after 8 cycles *done_xmit*

2. Write a property that states that if the sequence qXmit2Done should occur within 10 cycles after the sequence qReq2Data.

3. Write a property that states the following requirements:
   a. If a new *BusReq* is acknowledged (*ack*) prior to a *timeout* signal, then the sequence qXmit2Done shall be initiated. *BusReq* must be asserted until either the *ack* or the *timeout* signal is asserted. Assume that *timeout* will not be asserted if *ack* is asserted.
   b. If a new *BusReq* is not acknowledged (*ack*) prior to a *timeout* signal, then the *BusReq* shall be deasserted.

4. Write a property that states the following requirements: A buffer will be read before it is overwritten. Assume that the following signals are available:

   rd : read enable, active high
   wr : write enable, active high
   addr : write address

   This requirement states that after a WRITE to an address, that data needs to be read from that address <u>before</u> it is re-written. Thus, this property requires a "before" property. The property expresses the requirement that if a WRITE to an address occurs, then later, there should be one READ to the same address.

5. Write a property with the following requirements:

   If a synchronous counter receives a load enable (*ld_n,* active low), and the reset signal is deasserted (*reset_n*, active false), then at the next clock cycle the output of the counter will be equal to the load data input of the counter (*data_in*).

6. Explain why in defining the "cause" of an action, it is preferred to use the **$rose** or **$fell** functions for the signal in question, instead of the signal directly.

7. Provide an English description of this property:

```
property pX7;
  a |=> b[*0:4] ##1  c ##1 !c[*10] ##1 d;
endproperty : pX7
```

8. Explain why this property can never fail. Also, simplify this property to lower terms
property pX8;
a |=> ##[0:$] ##1 b[*0:4] ##1 c ##1 !c[*10] ##1 d;
endproperty : pX8

9. Explain the meaning of these property expressions:
a |=> b[*3] ##1 c;
a |=> b[->3] ##1 c;
a |=> b[=3] ##1 c;
a |=> [*3] ##1 b ##1 c;

10. Write a property that states that if there is a bus request (*bus_req)* followed by no acknowledge (*ack*) for 10 cycles, or a DMA request (*dma_req*) followed by no memory release (*mem_in_triatate*) in 5 cycles, then an error interrupt (*err_intrpt*) shall be asserted.

11. Describe the meaning of this property using text and timing description, showing the order of transactions (i.e., *send_cmd, header_enb,* etc*)* :
property pDataTransfer;
(send_cmd) |=>
##[0:10] header_enb ##1 (data_enb)[*32] ##1 crc_enb;
endproperty : pDataTransfer

12. Define a waveform for this property:

property P; @ (**posedge** clk) a |=> b |=> c; **endproperty** : P
aP : **assert property** (P);
With the following waveform, what is the value of the assertion at time 100 ns, 150 ns, 200 ns?

13. Why are properties named?

14. Write a property that expresses the following requirement:

"If an FSM (*fsm_reg*) is in IDLE state and it receives a FIRE command (*fire*, active high), then at the next clock cycle the FSM will enter the ARM state, and the cycle after that the FSM will enter the IGNITE state. The FSM will then return to the IDLE state. Assume that the states of the FSM were predefined to be IDLE, ARM, IGNITE.

15. Write a property that states that there should never be more than one *grant* (i.e., the *grant* bus can either be all ZEROs, or at most have one bit asserted).

16. Write a coverage property that states that an occurrence of an active *grant* has transpired.

17. Explain in English the meaning of the following properties:

a.
```
sequence qAck;
        ##3 ack;
  endsequence : qAck

  property pReqAck;
    req |-> ##3 ack;
  endproperty : pReqAck

  property pAckThenPrevReq;
      qAck.ended |-> $past(req, 3);
  endproperty : pAckThenPrevReq
```

b.
```
property pReq3Acks;
    req |=> ack[*3] ;
endproperty : pReq3Acks
```

c.
```
property pReqAckOnce;
    req |=> ack[->1] within `true[*3];
endproperty : pReqAckOnce
```

d.
```
property pReqAckDone;
    req ##1 ack[->2] ##0 data_xfr |=> done;
endproperty : pReqAckDone
```

e.
```
property pReqAcksDataXFrs;
    req |=> (ack && data_xfr) [=1:31] within `true[*40];
endproperty : pReqAcksDataXFrs
```

f.
```
property pReqAck100DataXfr;
    req |=> (##[0:$] data_xfr[*16]) within ack[=1:100];
endproperty : pReqAcksDataXFrs
```

18. Write assertions for the following requirements:

a. If the transmit FIFO is not empty, and no data output transmission is in progress, then a new transmission shall be started within 5 clock cycles.
Assume that the following signals are available:
fifo_size, xmit_status (0==no transmission, 1== in tranmission mode).

b. When a transmission of data is completed, as indicated but the deassertion of the *xmit*_status signal, then a CPU interrupt shall be asserted within 4 clock cycles.

5 ADVANCED TOPICS FOR PROPERTIES AND SEQUENCES

This chapter provides advanced topics in the use of assertions including system functions supporting assertions; single and multiple clocking of sequences; SystemVerilog scheduling for the support of assertions, properties in interfaces, verification directives and their significance in simulation and formal verification; and binding of designs unit with properties; mixed mode simulation with VHDL modules and SystemVerilog Assertions verification modules.

5.1 DATA TYPES IN PROPERTIES AND SEQUENCES

When a sequence or property is instantiated, the actual arguments can replace an:

- identifier
- expression
- event control expression
- upper range as $ or a static natural number

Thus, the following would be legal:

Property expression *p* would be expanded as:

```
@ (posedge clk) m |=> s ##[0:10] data=8'hFF,
```

An example of a property that uses a sequence for an actual argument is shown below.

```
sequence qA23B;
 @ (posedge clk) a ##[2:3] b; // a then within 2 to 3 cycles, b
endsequence : qA23B

sequence qDE;
 @ (posedge clk) d ##1 e[*2]; // d then e repeated twice
endsequence : qDE

property p_test(seq_end, seq, c);
 @ (posedge clk)
  seq_end |-> c ##1 seq; // if seq_end then c in same cycle, and seq at next cycle
endproperty : p_test

ap_test: assert property (p_test(qA23B.ended, qDE, c));
```

Passing as actual arguments an endpoint, a sequence, and a signal

By maintaining type independence of the formal arguments, the defined properties become "type independent", and can be reused with actual arguments of any type, as long as the operations are legal. In essence, this is a form of overloading, in that the same property can be reused with objects of any actual data type. However, even with the use of "untyped" formal arguments, properties or sequences with local variables are not truly "overloaded". This is because the local variables of properties and sequences are typed, and those variables may be compared against objects passed as actual arguments. This weak typing feature of formal arguments will yield to compilation, or elaboration errors, or warnings when the operations on the arguments and assertion variables are of incompatible types.[1] For example, given the following property declaration:

```
property p_req2send (a, b, c);
  logic [7:0] v_data ; // local assertion variable
   @ (posedge clk) // at every clock
   (a, v_data=b) |=> // if a then assertion variable v_data = b
   ##[0:3]  // within next cycle to 3 cycles later
   c==v_data); // c must be equal to data
endproperty : p_req2send
```

Local variable is an 8-bit vector

Assume that the actual arguments are defined as follows:

```
logic request;
logic [31:0] sent_data, received_data;
ap_req2send_1: assert property  (preq2send(request, sent_data, received_data));
```

Comparison against an 8-bit vector

An instantiation of the property with the above arguments will yield erroneous results because the value stored is truncated to 8 bits inside the property (using "*v_data*") and later compared against a 32-bit number. Though in the above example there is no "data type" mismatch, it highlights the issue of having untyped formal argument and typed local assertion variables. An intuitive solution to this problem would be to pass the size of the vector as another formal argument, but

[1] As of the writing date of this book, there is a potential enhancement to allow type definition on formal arguments to sequences and properties

SystemVerilog does NOT allow the use of formal arguments in places where constants are expected, and thus cannot represent sizes of arguments. Thus, the following is illegal (and ill-advised in any case):

```
property pERROR(max, seq, data, c);
 logic [max: 0] v_data;
 @ (posedge clk)
  (seq, v_data=data) |-> ##3 c;
endproperty : pERROR
```

Formal argument "max" cannot be used to parameterize sizes in a property or sequence. They must be static, known at elaboration time (e.g., a constant or a parameter). Instead, use constants or parameters to specify ranges.

The formal arguments of a property are untyped, and is also known as *polymorphic datatype*.[2] This allows a property to be reused with objects of a different type. Basically, this allows a form of "overloading" a property. **However, the local variables of a property are typed.** Thus, in instantiating properties that make use of local variables, users need to be cautious to avoid a type mismatch between the actual arguments and the local variables of the property.

When using arguments properties:

- Insure that actual arguments are compatible to the operations done inside the property description (e.g., comparison of a vector against a memory element is illegal).
- Insure that assertion variables are compatible to the operations described by the actual arguments.
- If the arguments are defaulted, use constant values, and put those arguments at the end of the list.

Note: The size of a local variable and the actual argument can be based on the same parameter constants, thus ensuring that the local variable and formal argument are compatible. The following example demonstrates the concept.

```
parameter WIDTH = 16;
logic [WIDTH-1 : 0] data;
property pOK (seq, data, c);
 logic [WIDTH-1 : 0] v_data;
 @ (posedge clk)
  (seq, v_data=data) |-> ##3 c;
endproperty : pOK
```

All sizes based on parameter constant

5.2 MISUSE OF ASSERTION OVERLAPPING

Assertion overlapping was briefly discussed in Chapter 2. When misused, assertion overlapping can lead to some unexpected behavior, and readers need to thoroughly understand the implications.

With an asserted property, the verification tool will look for the start of the assertion on every sample, even if the sequence is already started. It does not wait for a sequence to finish. This can

[2] http://en.wikipedia.org/wiki/Polymorphism_(computer_science) **polymorphism** is the idea of allowing the same code to be used with different types, resulting in more general and abstract implementations.

create overlapping assertions that are monitoring the same set of signals during the same bus cycle. While such overlapping is very powerful, these overlapping assertions can lead to unexpected assertion failures. For example, the following property represents a simple sequence for an FSM.

```
property pUNEXPECTED_ASSERTION;
@ (posedge clk)
 start |=> s_bits == STATE1 ##1 s_bits == STATE2;
endproperty : pUNEXPECTED_ASSERTION
apUNEXPECTED_ASSERTION_1: assert property (pUNEXPECTED_ASSERTION);
```

Figure 5.2-1 shows the state diagram along with timing of the signals in the design.

Figure 5.2-1 Simple State Machine

A good state machine that passes the assertions has the following behavior:

- It will transition from the *Idle* state to *State1* on the positive edge of *clk* when *start* ='1'.
- It will transition from *State1* to *State2* on the next positive edge of *clk*.

A simulation on this design may lead to an assertion failure, even though it appears that the state machine is working properly (as shown in Figure 5.2-2). This failure will occur if *start* remains high for more than one clock continuously.

The reason for the behavior is as follows:

- On the first positive edge of *clk* after *start* = 1'b1, the assertion starts looking for the sequence. This is shown in Figure 5.2-3.
- On the second clock edge, the following happens:
  - The first assertion samples *s_bits* and finds s_bits = STATE1.
  - A second, overlapping assertion is started because *start* = '1', as shown in Figure 5.2-4.
- On the third clock edge the following happens:

- The first assertion samples *s_bits* and finds *s_bits* = STATE2. This is the end of expected sequence, so the assertion completes without failure.
- The overlapping assertion samples *s_bits* and finds *s_bits* = STATE2. It was expecting *s_bits* = state1, so the assertion generates a failure.

Another overlapping assertion starts, because *start* = 1'b1. This behavior is shown in Figure 5.2-5.

- Figure 5.2-2 Simulation of "start |=> s_bits == STATE1 ##1 s_bits == STATE2;"

Figure 5.2-3 Start of the First Assertion

Figure 5.2-4: The Sta

rt of an Overlapping Assertion

Figure 5.2-5: Ov

"start |=> s_bits == S

erlapping Assertion Failure

TATE1 ##1 s_bits = STATE2;"

The assertion failure occurred because the condition that started the sequence did not describe the desired starting sequence. A better description would specify that the state machine must be in the *IDLE* state when *start* ='1', as shown in Figure 5.2-6.

```
property pEXPECTED_ASSERTION
 @ (posedge clk)  (s_bits == IDLE && start) |=> (s_bits == STATE1 ##1 s_bits==STATE2);
endproperty : pEXPECTED_ASSERTION
```

Figure 5.2

-6 Expected Assertion

Another approach is to check that the *start* signal changed from 0 to 1. This can be achieved with the use the built-in function **$rose**, as shown in Figure 5.2-7. [3]

```
property pEXPECTED_ASSERTION1:
@ (posedge clk)
 $rose (start) |=> s_bits == State1 ##1 sbits == STATE2;
endproperty : pEXPECTED_ASSERTION1
```

Figure 5.2-7 Eliminating Unwanted Overlapping in Assertions Using $rose Function

Example: Given the assertion (**$rose**(a) ##1 b ##1 c) |=> (d ##1 e), Figure 5.2-8 demonstrates a unique start of the property when *a* changes from 0 to 1 at cycle *t2*. At cycle t3, there is no overlapped start since *a* did not rise, and the property at cycle 3 succeeds vacuously. Figure 5.2-9 demonstrates a problem with a SEQUENCE (a ##1 b ##1 c) |=> (d ##1 e) at time *t6*. At cycle t3, there is an overlapped start of the property since *a* holds.

($rose(a) ##1 b ##1 c) |=> (d ##1 e)							
	t1	*t2*	*t3*	*t4*	*t5*	*t6*	
a	0	1	1	0	0	0	
b			1	1	0	0	
c				1	1	0	
d					1	0	<- OK
e						1	
		^rose(a)		^trigger			

Figure 5.2-8 Unique start when a Changes from 0 to 1 at Cycle t2.
"trigger" identifies time when LHS completes and RHS will be "checked

a ##1 b ##1 c) |=> (d ##1 e)							
	t1	*t2*	*t3*	*t4*	*t5*	*t6*	
a	0	1	1	0	0	0	
b			1	1	0	0	
c				1	1	0	
d					1	0	<- FAIL
e						1	
				^trigger	^trigger		

Figure 5.2-9 Problem if Requirement for Sensitivity to a New Occurrence of *a*.
"trigger" identifies time when LHS completes and RHS will be "checked

[3] **$rose** returns TRUE if the least significant bit of the expression changed to 1. Otherwise, it returns FALSE.
$fell returns TRUE if the least significant bit of the expression changed to 0. Otherwise, it returns FALSE

5.3 MULTIPLE THREADS TERMINATION

An asserted property is evaluated at every cycle, and every cycle can start a new thread. If a property has an antecedent that is evaluated to FALSE, then the new thread is vacuously TRUE.[4] However, with antecedents that evaluate to TRUE, it is possible for the assertion to have multiple concurrent threads.[5] These threads can be satisfied with a single termination or completion of the consequents. This is demonstrated in Figure 5.3 where two active threads are terminated with a single termination.

```
property p_req2send (a, b, c);
  @(posedge clk)    // at every clock
    $rose(a) |=> // if a then starting from next cycle
    ##[0:10] b   // either at this next cycle or up to 10 cycles later b must occur
    ##2 c; // c must then occur 2 cycles later, after b
endproperty : p_req2send
ap_req2send_1: assert property  (p_req2send(request, sent_data, received_data));
```

Figure 5.3 Multiple Assertion Threads may be Completed with a Single Termination

Note: It is also possible for multiple threads with a single termination to cause unexpected failure. For example,

```
property p_req2send_var (a, b, c, d, data, xbus);
  int v_data;  // Every thread maintains its own copy of the variable
  @(posedge clk)    // at every clock
    $rose(a) ##[1:5] (b, v_data=data) |=> // if a and later b, then variable v_data ==data
    ##[0:10] c  ##2 d && xbus==v_data; // when d, then xbus must equal to v_data
endproperty : p_req2send_var
```

If the data captured at active thread 1 is different than the data captured at thread 2, and at termination (same cycle for thread 1 and thread 2), *xbus* is equal to the *data* captured at the start of thread 1, then thread 2 termination will display an error. See section 3.2.4 for the application of a unique identification tag to avoid this problem.

---

[4] See appendix B for definition of vacuous property.

[5] An assertion evaluates to a vacuous TRUE if the antecedent is FALSE (e.g., a ##1 b |-> .. is vacuously TRUE if *a* is FALSE, or if *a* is TRUE at cycle *n*, and *b* is FALSE at cycle *n+1*).

5.4 ASSERTION REFINEMENT PROCESS

Assertions can be captured across different phases throughout the design cycle starting from specification, micro-architecture, RTL coding and Verification (See Chapter 6 – SVA in Design process). However, some of the minor timing details may not be available during specification phase, for instance. A good ABV methodology recommends a "continuous assertion refinement process". Assertion definitions can be continuously refined. It is possible to start out with an assertion that describes generic overall behavior.

```
property pIOCycle_generic;
 @ (posedge clk)
   ($rose(IOReq)) |=> ( ##[0:$] IOGrant ##[0:$] ##1 !IOReq );
 // Note: Because of the ##[0:$], above is same as:
 // ($rose(IOReq)) |=> ( ##[0:$] IOGrant ##1 !IOReq );
endproperty : pIOCycle_generic
```

This assertion states that if there is a new *IOReq*, then some time later, there will be an *IOGrant*, and then some time later the *IOReq* is deasserted. That describes the overall characteristic of an IO Cycle. This assertion can be viewed as a high level, unrefined assertion. Note that because of the " |=>(##[0:$] ## .." the assertion can never fail, but can only complete.

As a next step, the assertion can be refined to incorporate more details of the protocol, which was defined in the requirements. Adding these details make the assertion more stringent. The use of formal arguments and/or parameters can be used to create a reusable assertion.

Stringent assertion

```
// IOGrant is asserted for 2 to 5 cycle after IOReq until IOReq is deasserted.
localparam MIN_GRNT = 2;
localparam MAX_GRNT = 5;
property pIOCycle_specific (IOReq, IOGrant);
 @ (posedge clk)
   ($rose(IOReq)) |-> ##[MIN_GRNT : MAX_GRNT] IOGrant  ##1 !IOReq;
endproperty : pIOCycle_specific
```

The following example trace meets the assertion.

cycle	1	2	3	4	5	6	7
IOReq	0	1	1	1	1	1	0
IOGrant	-	-	1	1	1	1	-
		↑ *Start condition*					

$rose(*IOReq*) if followed by at least 2, but no more than 5 cycles of *IOGrant*, followed by *IOReq* ==1'b0

By adding specific requirements into the sequence definition, the assertion now requires that *IOGrant* needs to follow *IOReq* in a specific range of cycles. The rising edge of *IOReq* was used as a condition, instead of *IOReq* only to avoid the effects of overlaps in cycles 3 through 6 in the above example.

> Formal verification tools <u>may</u> require stringent and bounded assertions to evaluate the assertions over a fixed number of cycles. A bounded engine may be used either as an optimization (i.e., a quick run to find failures before checking the assertion over all time), or in cases where the unbounded engine runs into capacity problems. However, there are formal engines that will check assertions over all time, and not just over a fixed number of clocks.

5.4.1 Relaxed, stringent assertion

The design may have a condition, such as RESET or CANCEL that cancels this IO Cycle. To add this feature, one could define the following, as shown in Figure 5.4.1.

```
//IOGrant is asserted between 2 and 5 cycles after IOreq
// not IOreq must follow immediately thereafter unless IOcancel
sequence qAbort;
  (reset || IOcancel);
endsequence : qAbort

property p_iocycle;
 @ (posedge clk) disable iff (qAbort.triggered)
    $rose(IOReq) |-> ##[2:5] IOGrant ##1 !IOreq; // 6
endproperty : p_iocycle

ap_iocycle : assert property (p_iocycle);
```

property canceled when the sequence *qAbort* is triggered

Figure 5.4.1 Handshake Assertion Example

In this example the property *p_iocycle* is canceled when the sequence *qAbort* endpoint is triggered, i.e., *reset* or *IOCancel* (a one cycle sequence in this example). This assertion will not report false negatives for the cases where *qAbort* endpoint holds.

5.5 UNBOUNDED RANGE $ IN PROPERTIES

The "$" represents unbounded range specification, where the upper index can be any integer. The following example makes use of parameters in properties.

```
module test_mprops #(parameter MAX_ACK_RESPONSE = $);
 property p_reqack(req, ack, clk);
  @(posedge clk) req |=> ##[0:MAX_ACK_RESPONSE] ack;
 endproperty : p_reqack
endmodule : test_mprops

module top;
 logic request, acknowledge;
  test_mprops #(. MAX_ACK_RESPONSE(5)) test_mprops;

 ap_handshake: assert property (top.test_mprops.p_reqack(request, acknowledge);
endmodule : top
```

Bounding the unbounded range

Accessing properties defined in other modules

To detect whether a constant is unbounded (e.g., with the $), SystemVerilog provides a system function **$isunbounded**(const_expression) that returns TRUE if the expression is unbounded, i.e., left to the unbounded value $, rather than being defined with a fixed natural number. The function **$isunbounded** is also used to check the validity of the actual arguments. For example, given an interface shown below with an *enb* signal, which is two-bit wide. Only one bit of that signal can be enabled at any cycle. If the parameter *max_quiet* is zero, then one bit of the enable signal *enb* must always be set. However, if the parameter *max_quiet* is greater than zero, then

---

[6] see section 4.7.3 for more information on the use of the triggered method

when *enb* drops to zero, it must remain zero for a minimum of *min_quiet* to a maximum of *max_quiet* cycles. Using the **generate**, two properties can be defined depending upon the value of the parameter *max_quiet*.

```
interface quiet_time_if #(parameter MIN_QUIET = 0,
                          parameter MAX_QUIET = 0)
        (input logic clk, reset_n, [1:0]enb);
 generate
 begin : gen_conditional_assert
 if ( MAX_QUIET == 0) begin
   property p_quiet_time;
    @(posedge clk) reset_n |-> ($countones(enb) == 1);
   endproperty : quiet_time
   ap_maxquiet_is0: assert property (p_quiet_time);
 end

 else begin
  property p_quiet_time;
    @(posedge clk)
      (reset_n && $past(enb) != 0) && enb == 0) // detect fell to zero for a vector
      |->(enb == 0)[*MIN_QUIET : MAX_QUIET]  // enb to remain zero for a duration
          ##1 ($countones(enb) == 1); // enb has one bit active
   endproperty : p_quiet_time
   ap_maxquiet_not0: assert property (p_quiet_time)
 end

  if ((MIN_QUIET == 0) && ($isunbounded(MAX_QUIET))
  $display("MAX_QUIET is unbounded");
 end : gen_conditional_assert
 endgenerate
endinterface : quiet_time_if
```

$fell (enb) would not work because **$fell** returns TRUE if the least significant bit of the expression changed to 0.

Check if max_quiet is unbounded.

Examples of instantiations of the interface are shown below:

```
quiet_time_if #(0, 0) quiet_never (clk,1,enables);
quiet_time_if #(2, 4) quiet_in_window (clk,1,enables);
quiet_time_if #(0, $) quiet_any (clk,1,enables);
```

5.6 RECURSION

SystemVerilog allows recursive properties. A named property is recursive if its declaration involves an instantiation of itself. Recursion provides a flexible framework for coding properties to serve as ongoing assumptions, checkers, or coverage monitors. For example,

```
property prop_always(p);
  p and (1'b1 |=> prop_always(p));
endproperty : prop_always
```

This is a recursive property that says that if the formal argument property *p* holds, then recursively, in the next cycle p must also hold. In summary, *p* must hold at every cycle. One way to look at recursion is to expand the property. Thus "p **and** (1'b1 |=> prop_always(p));" expands to:

```
p and (1'b1 |=> // <------------------------- prop_always(p) ---------------------...
                p and (1'b1 |=> // <------------------------- prop_always(p) ---- ...
                                p and (1'b1 |=> // <----------------prop_always(p) ---...
                                                p and (1'b1 |=>
                                                                p and (1'b1 |=> ....);
```

The expansion of this recursive property clearly demonstrates that at every cycle *p* must hold. This concept is demonstrated graphically in Figure 5.6-1.

Figure 5.6-1 Demonstration of Recursion p and (1'b1 |=> prop_always(p));

The recursive property *prop_always(p)* is useful if there is a requirement that *p* always holds after a complicated triggering condition defined in a sequence. For example, the following can be written to specify that after a RESET sequence, the *reset* signal must remain high:

```
sequence qReset;
  !reset_n [*0:$] ##1 reset_n; // reset_n == 0 until reset_n == 1
endsequence : qReset

// Property that states that after a reset sequence, then the reset_n must remain high.
property p_ResetHighAfterActiveInitialReset;
  qReset |=> prop_always(reset_n);
endproperty : p_ResetHighAfterActiveInitialReset
```

Another example that uses recursion is the following property that requires that formal argument *p* must hold at every cycle up to but not including the first cycle at which formal argument *q* holds. Formal argument property *q* is not required to ever hold though. Note that the requirement is different than this property "p[*1:$] ##1 q", which requires *q* to eventually hold.

```
property prop_weak_until(p,q); // p until_ q
  q or (p and (1'b1 |=> prop_weak_until(p,q)));
endproperty : prop_weak_until
```

Recursively expanding this property yields the following:

```
q or (p and (1'b1 |=> // <-- prop_weak_until(p,q))); -----....
                q or (p and (1'b1 |=> // <-- prop_weak_until(p,q))); -----....
                                q or (p and (1'b1 |=> ....
```

The above clearly demonstrates that if *q* does not hold, then *p* must hold. However, if *q* holds, then *p* is not required to hold, and the property terminates.[7] This example is useful if *p* must hold

[7] This is defined in PSL as a weak **until**.

at every cycle after a complicated triggering condition encoded in sequence *s*, but the requirement on *p* holding is lifted when *q* holds. There is no failure if *q* never occurs.

```
property p2(s,p,q);
  s |=> prop_weak_until(p,q);
endproperty
p2_1: assert property (p2(go, req, ack) else
        $display ("%m  Request not holding until acknowledge, time=%0t");
```

More generally, several properties can be mutually recursive. For example, to express the notion of a property that has repeated sequences until the antecedent hold vacuously, recursive properties can be used. Below is such an example shown non-recursively to express the following requirements:

"A transaction consists of a *data request* followed by an *acknowledge*, followed by a *send* cycle and then a *done* control. The *data request / acknowledge* sequence is initiated with a *make_request* control. The *send / done* sequence is initiated with *reply2request* control. A new transaction can be started in the cycle following the *done*. However, if either *make_request* or *reply2request* is not asserted in the transaction cycle, then the transaction is considered completed". In other words, the following is expected (in non-valid notation):

```
make_request |-> req |=> ack  ##0 reply2request  |-> send |=> done  ##0
    make_request |-> req |=> ack  ##0 reply2request  |-> send |=> done  ##0
      |=> req |=> ack  |=> send |=> done
        ... // repeated forever until either
              make_request ==false    // or until
              make_request |-> req |=> ack  ##0 reply2request  == `false;
```

This type of requirement modeling yields itself to recursion, and can be expressed as follows:

```
property phase1_prop;
  req |=> ack;
endproperty : phase1_prop

property phase2_prop
  send |=> done;
endproperty : phase2_prop

property check_phase1;
 make_request |-> (phase1_prop and (1'b1 |=> check_phase2));
endproperty : check_phase1
property check_phase2;
 reply2request  |-> (phase2_prop and (1'b1 |=> check_phase1));
endproperty : check_phase2
```

Those properties can be flattened as:

```
    make_request |-> (phase1_prop and (1'b1 |=>
        reply2request  |-> (phase2_prop and (1'b1 |=>
              make_request  |-> (phase1_prop and (1'b1 |=>
                 reply2request  |-> (phase2_prop and (1'b1 |=> ....));
```

Suppose that in the 3rd recursion *make_request* is zero. Then the expanded property becomes:

```
make_request |-> (phase1_prop and (1'b1 |=>
    reply2request |-> (phase2_prop and (1'b1 |=>
        make_request ==0 |-> (phase1_prop and ...;  // ← This property holds because
                                                    //    the antecedent is FALSE
```

Thus, when *make_request* becomes FALSE in the 3rd recursion, the recursive property effectively terminates the recursion, and becomes:

make_request |-> (phase1_prop **and** (1'b1 |=> reply2request |-> (phase2_prop);

or

make_request |-> ((req |=> ack) **and** (1'b1 |=> reply2request |-> (send |=> done));

This property is shown in Figure 5.6-2.

Figure 5.6-2 Behavior of Recursive Properties check_phase1 and check_phase2

5.7 EMULATING PSL-LIKE CONSTRUCTS IN SVA

PSL and SystemVerilog with assertions are two Accellera standards that support an Assertion-Based Verification methodology. The purpose of this section is to demonstrate SVA coding styles for constructs available in PSL, with the goals of further explaining the use of SystemVerilog Assertions.

5.7.1 whilenot [8]

A *whilenot* specifies a window of time that starts when the property is triggered and lasts until the terminating condition. This is unlike the *within* that defines the window of the time in which the sequence must hold. The *whilenot* can be emulated with either of the following:

(`true[*0:$] ##1 seq1) **intersect** seq2

(##[0:$] seq1) **intersect** seq2

[8] Note: The **whilenot** operator was removed in PSL.1.1 (it existed in PSL 1.01). However, to demonstrate SVA concepts, an emlation of the **whilenot** is explained.

An example application of such *whilenot* behavior is demonstrated with a DMA sequence in Figure 5.7.1-1.

```
module whilenot;
 logic go, done_dma;
 logic dma_req, data_accepted;
 logic clk;          // synchronous clock

 property p_whilenot_go2done;
 @ (posedge clk)  $rose(go) |->
  (##[0:$] done_dma)  intersect
  (
    (##[0:$] $rose(dma_req) ##[1:4] data_accepted)[*2] ##1 `true
  );
 endproperty : p_whilenot_go2done

 a p_whilenot_go2done2_1: assert property (p_whilenot_go2done2);
endmodule : whilenot
```

Figure 5.7.1-1 Application of whilenot

This property, as demonstrated in Figure 5.7.1-2, states that within the window of time starting from the trigger condition **$rose**(go) until done DMA flag (*done_dma*), the controller must transfer 2 words of data using the handshake sequence DMA request (*dma_req*) followed within 4 cycles with data accepted (*data_accepted*). The property completes at the terminating condition *done_dma*, which must occur after the last occurrence of *data_accepted*. The assertion overlaps with every rise of *go*.

Figure 5.7.1-2 DMA Transfer Property with the whilenot Operator

5.7.2 The eventually! Operator in Sequence

SystemVerilog does not support an *eventually!* operator. However, it is possible to express a similar notion without the *eventually!* operator, as shown in Figure 5.7.2. The GRANT_AFTER_REQ4 property states that if there is a sequence *make_req* followed by *req*, then sometime in the future there may be a *grant* followed by *ok*. The *grant* may occur in the cycle when *req* is true. If *grant* followed by *ok* occurs then the assertion completes.

```
// Grant must always occur after req.
 property pGRANT_AFTER_REQ4;
  @ (posedge clk)  (make_req ##1 req) |-> ##[0:$] grant ##1 ok;
 endproperty : pGRANT_AFTER_REQ4
```

Figure 5.7.2 Application of an Equivalent Eventually! with Sequences

As mentioned before, that assertion can never fail because of the [*0:$] repetition operator, but completion will be identified. A question that is often asked is "what happens to pending obligations of such assertions in a simulation environment?" In the above example, if the antecedent is TRUE and the consequent "*grant ##1 ok*" never occurs until the simulation terminates, is that obligation a failure? The LRM is not explicit on this point, and it appears to be tool dependent. Synopsys *vcs* in SystemVerilog mode reports an incomplete evaluation attempt when an assertion evaluation that is in progress does not succeed or fail before the end of simulation. It indicates the start time of the evaluation too. It is up to the user to decide whether it should be considered a failure, i.e., interpret the semantics of satisfaction on that finite trace.[9]

The PSL *enventually!* requires that the eventual condition occurs by the end of simulation time (this is known as a *strong* operator). This *strong* operator may be emulated with sequences by putting a bounded range of reasonable size on the "[*0:$]", as shown below:

```
make_req ##1 req |-> ##[0 :100] grant ##1 ok;
```

The definition of "reasonable" is a value by which the eventual condition is expected within that bounded number of cycles, and it would be an error if it does not. Though this is not mathematically equal to the string eventuality operator, this is a reasonable approach in designs because the bounded number of cycles typically represents an expected or maximum latency. In the above example, a failure would occur if the *grant ##1 ok* sequence does not occur after 100 cycles following an occurrence of *make_req ##1 req* sequence (assuming simulation lasts for the full 100 cycles).

5.7.3 Emulating UNTIL with Sequences

SystemVerilog does not provide an *until* operator, where an expression holds until another expression holds. However, the *until* can be emulated using repetition operators. For example:

Until Expression	SVA Sequence
If *req*, then *req* must remain asserted until *ack* is asserted. *req* is not required to occur in the cycle after *ack* occurs.	req |=> req[*0:$] ##1 ack;

[9] Unbounded Assertions
http://verificationguild.com/modules.php?name=Forums&file=viewtopic&t=562

5.7.4 F before G

Another useful property is the occurrence of a sequence before another sequence.[10] With Boolean expressions, the property *FbeforeG* can be expressed as:

```
property FbeforeG(f, g);
  @ (posedge clk) (not (##[0:$] g)) or (!g[*0:$] ##1 (f && !g));
endproperty : FbeforeG
```

This property completes if *g* is FALSE until *f* is TRUE. Expression *g* must be FALSE when *f* is TRUE. The property fails if *g* is ever FALSE before *f* occurs Figure 5.7.4 demonstrates a timing diagram for this property.

Figure 3.5.4 Timing Diagram, FbeforeG

The property can be rewritten using recursion as follows, assuming a default clock,

```
property pFbeforeG(F,G);
   (not G) and (F or (1'b1 |=> pFbeforeG(F,G)));
endproperty : pFbeforeG
```

Expanding the recursion, the property can be seen as:

```
(not G) and (F or (1'b1 |=> // pFbeforeG(F,G))); ----...
                    (not G) and (F or (1'b1 |=> // pFbeforeG(F,G))); ----...
                                        (not G) and (F or (1'b1 |=> ....
```

[10] A technical complication with adding a derived operator like "before" to SystemVerilog Assertion is that in the current framework one cannot negate an instance of a recursive property, and this adds a restriction on the arguments of the derived operators. The "before" is a PSL property operator.

Thus, at every cycle, G is FALSE AND either F is TRUE or there is a repeat of the check at next cycle. This recursive property completes successfully if G is FALSE and F is TRUE. If G is FALSE and F is FALSE, then the check is repeated at the next cycle. The property fails if G is TRUE.

An application of this property is the following example: "When a read request occurs, data must be read from the memory before it is asserted onto the output port." This requirement can be expressed by using the property *FbeforeG* previously declared, and by following the definitions of a property:

```
sequence qReadMem;
  mem_read ##1 ld_data; // memory read control followed by a load of data control
endsequence : qReadMem

sequence qData2OutputPort;
  ready ##1 port_enable; // ready control followed by a port enable control
endsequence : qData2OutputPort

// When a read request occurs, data must be read from the memory before
//  that data is asserted onto the output port
property pReadMemOutput;
  read_request |=> pFbeforeG (qReadMem, qData2OutputPort);
endproperty : pReadMemOutput
```

5.7.5 One-Shot Assertion Using Initial blocks

One-shot assertions are often used to define conditions considered as assumptions during initialization. SystemVerilog syntax does not directly support one-shot assertions, as compared to PSL that defines a one-shot property as a property without the *always* or *never* operator. However, two methods can be used to emulate this behavior:

1. A flag bit activated only on the first cycle after the initialization point. That flag bit acts as an antecedent valid only during the first cycle.
2. Use of procedural assertion statements in an initial block.

5.7.5.1 Flag Bit Defining Start of Antecedent

This method uses a flag (e.g., flag *t*) initialized at declaration, and reset after the first clock cycle. The flag is used as an antecedent to an assumed property, like reset. For example:

```
logic  t = 1;
// t is reset after the first clock
initial begin @(posedge clk); t <= 0; end

// Once reset_n is high (disabled), it remains disabled
// Checked at every clock cycle.
property p_no_reset_forever;
  @(posedge clk) !reset_n |=> !reset_n;
endproperty : p_no_reset_forever
```

```
// This property is fired once. Thereafter, the property is vacuously TRUE because t==0 .
property pReset5Cycles;
 @(posedge clk) t |-> !reset_n[*5] ##1 reset_n;
endproperty : pReset5Cycles

//Verification direction for formal verification tool.
mp_no_reset_forever_1: assume property (p_no_reset_forever);
mpReset5Cycles_1: assume property (pReset5Cycles);
```

5.7.5.2 Procedural Assertion in Initial Block

This method declares the property as unclocked, but uses a verification directive (e.g., **assume** or **assert**) after a clock edge in an initial block. For example:

```
property p_reset_at_init;
  !reset_n[*5] ##1 reset_n;
endproperty : p_reset_at_init
initial @(posedge clk)
   mp_reset_at_init : assume property (p_reset_at_init);
```

The property *p_reset_at_init* is evaluated only once after the first clock cycle during start of simulation. This option is more efficient than the first one in terms of simulation performance because the previous solution requires the property *Reset5Cycles* to be evaluated in EVERY clock (and hold vacuously).

5.8 ASSERTION-BASED SYSTEM FUNCTIONS

5.8.1 Sampled Valued Functions

This section describes the system functions available for accessing sampled values of an expression. These functions include the capability to access the current sampled value, access a sampled value in the past, or detect changes in a sampled value of an expression. Two classes of functions are provided:

- Value access functions, which include:
  $sampled(expression [, clocking_event])
  $past(expression1 [, number_of_ticks] [, expression2] [, clocking_event])

- Value change functions, which include:
  $rose(expression [, clocking_event])
  $fell(expression [, clocking_event])
  $stable(expression [, clocking_event])

5.8.1.1 Value access functions

Verilog event scheduling is designed to represent event propagation for combinational and sequential logic. With Verilog, verification engineers have to be very careful that verification events do not have races with the hardware model events. However, SystemVerilog enhanced scheduling allows assertions and testbench to work together seamlessly to avoid Verilog evaluation races.[11] SystemVerilog defines the scheduling and sampling semantics to avoid the Verilog classes of ambiguities. It defines various regions for the signal updates such as Preponed, active, observe, and postponed regions. It also specifies when different constructs (such as assertions) shall be evaluated. Since real hardware has setup and hold times, it may be useful to

[11] http://www.systemverilog.org/pdf/5a_V2K_SV_compatibility.pdf

refer to the value of a signal at the sampling time. The value access functions obtain the values of signals at certain evaluation times.

5.8.1.1.1 $sampled(expression |, clocking_event|)

Function **$sampled** returns the sampled value of the expression with respect to the last occurrence of the clocking event. Section 5.10 explains the scheduling semantics for assertions, and explains that SystemVerilog scheduling has five gross basic scheduling regions: [12]

1. Preponed — Sampling of signals for evaluation of assertions
2. Active — Blocking and nonblocking signal updates (i.e., RTL processing)
3. Observed — Evaluation of properties using sampled signals
4. Reactive — Assertion blocks evaluation
5. Postponed — PLI callback for suspension of user code

Understanding the scheduling semantics is important because it impacts when signals are observed within the expression of an assertion. Consider the model of a counter shown in Figure 5.8.1.1.1

```
module counter;
 timeunit 1ns;
 timeprecision 100ps;
 logic clk=0, reset_n=0, go = 0;
 int cntr =0;
 property pCounterMaxed;
  @ (posedge clk) disable iff (!reset_n)
  go |=> (cntr <= 3);
 endproperty : pCounterMaxed

 apCounterMaxed : assert property (pCounterMaxed) else
    $display("%0t SVA_ERR: Counter exceeded 3, cntr %0d; sampled(cntr) %0d ",
         $time, cntr, $sampled(cntr));

 initial
 begin : clk_gen
  clk <= 0;  forever #10 clk <= ~clk;
 end

initial
 begin : stim
  #100;
  @ (negedge clk); // sync to neg edge to avoid data change in active clk edge
  reset_n = 1;
  go = 1;
  repeat (10) @ (negedge clk);
  $stop;
 end // stim
always @ (posedge clk)
  if (!reset_n)  cntr <= 'b0;
  else  cntr <= cntr + 1'b1;
endmodule : counter
```

Figure 5.8.1.1.1 Counter Model

[12] See section 5.10 on scheduling semantics

When the sampled value of *cntr* in the Preponed region is 4, *cntr* will increment to the value of 5 in the "active" region. The property is evaluated in the "observed" region, and the action block (the display of values) is performed in the "reactive" region. In that region, the current value of *cntr* is 5, the incremented value performed in the "active" region. However, the cause of the error for that thread is the sampled value, which had a value of 4. Thus, to display the value of *cntr* variable at the time it was sampled and used by the property, it is necessary to use the **$sampled** function. For the above model, a sample of the display reported in simulation was:

```
"../src/cntr.sv", 11: counter.apCounterMaxed: started at 70000ps failed at
90000ps    Offending '(cntr <= 3)'
```

Value of counter during evaluation of property

Sampled value used for property checking

```
90.00 ns SVA_ERR: Counter exceeded 3, cntr 5; sampled(cntr) 4
```

Note: The use of **$sampled** in assertions, although allowed, is redundant, as the LRM specifies that the assertions shall be evaluated during observe region, but the values used inside the expressions shall be those sampled at the Preponed region. However, the **$sampled** is necessary in an action block for access to signals sampled during the evaluation of the properties.

Another application of the **$sampled** function is for the definition of synchronous resets in **disable iff** within a property specification. The expression in the **disable iff** is asynchronous, but clock synchronization can be obtained by using the **$sampled** of the expression. For example:

```
property pReqAckSynchronousReset;
 @ (posedge clk) disable iff (! $sampled (reset_n))
    req |=> ##[0:3] ack;
 endproperty : pReqAckSynchronousReset
apReqAckSynchronousReset : assert property (pReqAckSynchronousReset);
```

5.8.1.1.2 $past

The **$past** function provides the value that an expression held in a previous n^{th} cycle. The syntax of the function is:

$past(expression1 [, number_of_ticks] [, expression2] [, clocking_event])

The three optional arguments define the following :

- *expression1* and *expression2* can be any expression allowed in assertions.
- *expression1* represents the expression being sought.
- *number_of_ticks* specifies the number of clock ticks in the past. *number_of_ticks* must be one or greater. If *number_of_ticks* is not specified, then it defaults to 1. If the specified clock tick in the past is before the start of simulation, the returned value from the **$past** function is a value of X.
- *expression2* is used as a gating expression for the clocking event.
- *clocking_event* specifies the clocking event for sampling expression. A clock tick is based on clocking_event.

```
regload |=> reg_data==$past(load_data);   // load_data @ previous cycle
regload |=> ##1 reg_data==$past(load_data, 2); // load_data @ previous 2 cycles
regload |=> ##1 reg_data==$past(load_data, 2, load_enable, @ (posedge clk) );
// value of load_data at previous 2nd cycle that had load_enable TRUE at clk positive edge.
```

When an explicit clocking event is specified in **$past**, it should match the clock that's used in the enclosing property.

> Always ensure that properties use the corresponding values of expressions over many cycles. Use of the **$past**[13] function may be needed to insure correspondence of an assertion that started at cycle *t*, but is tested at cycle *t+n* and needs values that were valid at time *t*.

Figure 5.8.1.1.2-1 demonstrates the issue of correspondence between signals over time. This models a CPU with an instruction register that has three fields: *instr* (i.e, instruction), *source1* (one source of data), *source2* (another source of data). At cycle *t1* there is an ADD instruction. Because of pipeline within the design, that instruction gets the results at cycle *t5*. Expressing the assertion as follows is an error because the values of *source1* and *source2* may have changed since, at cycle *t5*, the instruction register has a new instruction. However, use of the **$past** function with cycle identification solves this issue.

(go && instr==ADD) |=> (##4 dest== (rfile[source1] + rfile[source2]));

(go && instr==ADD) |=> (##4 dest==(rfile(**$past** (source1, 4) + rfile[**$past**(source2, 4));

	t1	t2	t3	t4	t5	t6	t7	
go	1	1	0	1	0			
InstructionRegister	Add(A, B)	MOV(X, Y)	XXX	SUB(C, D)	XXX			
instr	ADD	MOV	???	SUB(C, D)	???			
source1	A	X	???	C	???			
source2	B	Y	???	D	???			
dest					rfile[source1] + rfile[source2]			
$prev(source1,4					A	X	???	C
$prev(source2,4					B	Y	???	D

Figure 5.8.1.1.2-1 Example of $past Function

Another interesting application of **$past** is to find the sequence endpoints that occurred in the past. Note that **$past**(sequence_instance.**ended**) will always return FALSE as the value of **ended** is NOT persistent. The **ended** method has no meaning outside of the narrow Observe region and does not generate any event. There are two other methods to obtain the past endpoint of a sequence:

1. Using **$past** with **triggered** method: This technique uses the **triggered** method in conjunction with **$past** function. Unlike the result of **ended** method that remains valid only through the Observed narrow region, the **triggered** method is persistent from the Observe region throughout the Postponed region of a cycle. For example: **$past**(sequence_instance.**triggered**)

---

[13] SystemVerilog Assertion allows the use of local variables to store data values to be sampled during recognition of an antecedent, or at any cycle in the sequence, and then use those variables in later comparisons. However, for fixed static delays, users may rely on the **$past** function.

2. Use of the **ended** method of terminating sequence: This techniques uses the **ended** method of a terminating sequence.

Figure 5.8.1.1.2-2 represents an example that explains these two techniques. The Figure uses the following model:

```
// sequence a then b
sequence qT; a ##1 b; endsequence : qT

// sequence a then b then don't care
sequence qTended; a ##1 b ##1 `true; endsequence : qTended

// property example:
property P1; @ (posedge clk) k |-> $past(qT. triggered); endproperty : P1
property P2; @ (posedge clk) k |-> qTended.ended; endproperty : P2
```

Figure 5.8.1.1.2-2 Techniques to access previous endpoints

5.8.1.2 Value change functions

5.8.1.2.1 $rose and $fell

$rose returns `true (e.g., 1'b1) if the least significant bit of the expression changed to 1 in the current cycle (i.e., the expression was not a 1'b1 in the previous cycle (was 0, X, or Z), and is a 1'b1 in the current cycle). Otherwise, it returns 1'b0, which represents FALSE. The built-in function **$rose**() is similar (but not identical) to **posedge** that addresses clocks. **$rose** returns TRUE when a Boolean signal argument transition is TRUE at the current cycle and FALSE in the previous cycle, with respect to the clock of its context, otherwise it returns FALSE.

The **$rose** function is useful to ensure that a triggering condition is edge sensitive rather than level sensitive. For example, in the following example in Figure 5.8.1.2.1-1, the **$rose**(a) is TRUE at time 100 ns, and property *pABC_rose* is fired once every time *a* rises because the antecedent is TRUE in that cycle. This is different than the property *pABC* that fires every time *a*

is TRUE, and that case occurs at time 100 ns, 150 ns, 200 ns, and 250 ns. Property *pABC* possible yields unexpected errors at time 400 ns, 450 ns, and 500 ns because the antecedent is TRUE at time 100 ns, 150 ns, 200 ns, and 250 ns, and thus for every cycle the antecedent is TRUE the consequent must be TRUE. In this example, property *pABC_rose* successfully completes at time 300 ns. However, property pABC fails at time 400 ns, 450 ns, and 500 ns.

```
property pABC_rose;
  $rose(a) |=> ([*0:2] b ##1 c;
endproperty : pABC_rose

property pABC;
  a |=> ##[0:2]  b ##1 c;
endproperty : pABC
```

$rose and **$fell** operate on expressions, which can be:

- 1-bit signal as the expression, e.g., **$rose**(request)
- Expression on a vector or a bus, e.g., **$rose**(address==16'h0010)
- Expression on an enumeration typed object, e.g., **$rose**(light==RED)

Interpreting a vectored or enumerated signal as an expression can yield unexpected results. For example, if *addr* changes from 4'b0110 to 4'b1100 then **$rose**(address) would be 1'b0 in the cycle *addr*==4'b1100. ☹

```
property pABC_rose;
  $rose(a) |=> ([*0:2] b ##1 c;
endproperty : pABC_rose

property pABC;
  a |=> ##[0:2]  b ##1 c;
endproperty : pABC
```

Figure 5.8.1.2.1-1 Application of $rose to insure single edge of signal during assertions

The following example states that if the traffic light transitions to YELLOW, then the light must remain YELLOW until it turns to RED in two to five clock cycles.

```
typedef enum { RED, YELLOW, GREEN } tlight_enum;
tlight_enum tlight;
property p_traffic_light_YELLOW2RED;
  @ (posedge clk)
    $rose(tlight==YELLOW) |=> tlight==YELLOW[*1:4] ##1 tlight == RED;
  // If YELLOW, then remain YELLOW for 1 to 4 cycles, and then RED
  // i.e., RED in 2 to 5 cycles after YELLOW, but YELLOW until RED.
endproperty : p_traffic_light_YELLOW2RED
```

$fell is similar to **$rose**, except that **$fell** returns 1'b1 if the least significant bit of the expression changed to 0 in the current cycle (i.e., the expression was a 1'b1 in the previous cycle and is a 0, X, or Z in the current cycle). Otherwise, it returns 1'b0, which represents FALSE

5.8.1.2.2 $stable

$stable returns TRUE if the value of the expression did not change between the sampled value at the previous cycle and the sampled value at the current cycle. Otherwise, it returns FALSE. Example:

```
// Following an active low request, the active high acknowledge must remain active
// for up to 100 cycles until data valid is asserted.
// Example emphasizes use of $fell and $stable
property pReq2sendValid;
  $fell(req_n) |=> ack ##1 $stable(ack)[*0:100] ##1 data_valid;
endproperty : pReq2sendValid
```

Figure 5.8.1.2.2 represents a timing diagram for this property. Note that this property could also have been written as follows:

```
property pReq2sendValid2;
  $fell(req_n) |=> ack [*0:100] ##1 data_valid;
endproperty : pReq2sendValid2
```

Figure 5.8.1.2.2 Application of $fell and $stable

5.8.2 Vector-Analysis System Functions

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is "one-hot". The following system functions are included to facilitate such common assertion functionality:

1. **$onehot** (expression) returns *`true* (bit 1'b1) **if only one bit** of the expression is high.
2. **$onehot0**(expression) returns *`true* (bit 1'b1) **if at most one bit** of the expression is high. This is equivalent to **$onehot**(expression) || expression==0
3. **$isunknown** (expression) returns *`true* (bit 1'b1) if any bit of the expression is X or Z. This is equivalent to ^(expression) === 'bx.
4. **$countones** (<expression>) returns the number of ONEs in a bit vector expression. X and Z values are not counted towards the number of ones.

5.8.3 Severity-level System Functions

A good verification environment provides messages of varied nature such as:

- Informative messages
- Debug messages
- Warnings
- Errors
- Fatal Errors

Until recently, individual design groups implemented this in different ways. Assertions, being close to the design, can provide valuable information. But unless such information is categorized under different classes (such as listed above), it can soon become overwhelming to decipher any useful information out of such messages. For instance, a memory leak detected in a system is a FATAL error; while a packet received out of order can be a soft-error/warning. SystemVerilog provides some system functions to classify the messages generated from assertions. These are classified according to their severity levels. By default, the severity of an assertion failure is "error". The severity levels can be specified by one of the following severity system tasks in the assertion FAIL statement:

$fatal | $error | $warning | $info

- **$fatal** generates a run-time fatal assertion error, which terminates the simulation with an error code. The first argument passed to **$fatal** is consistent with the corresponding argument to the Verilog **$finish** system task, which sets the level of diagnostic information reported by the tool. Calling **$fatal** results in an implicit call to **$finish**.
  Example:
  ```
  ap_no_mem_leak : assert property (p_no_mem_leak) else
    $fatal (2, "%m @ time %0t, Memory leak detected in system.
              terminating simulation ", $time );
  ```
- **$error** represents a run-time error. Example:
  ```
  ap_handshake : assert property (p_handshake) else
    $error ("%m @ time %0t, req = %0h, ack=%0h", $time, request_bus, ack_bus);
  ```
- **$warning** is a run-time warning, which can be suppressed in a tool-specific manner. Example:
  ```
  ap_handshake : assert property (p_handshake) else
    $warning ("%m @ time %0t, req = %0h, ack=%0h", $time, request_bus, ack_bus);
  ```

- **$info** indicates that the assertion failure carries no specific severity. Example:
  ap_handshake : **assert property** (p_handshake) **else**
  $info ("%m @ time %0t, req = %0h, ack=%0h", **$time**, request_bus, ack_bus);

5.8.4 Assertion-Control System Tasks

SystemVerilog provides three system tasks to control assertions. This can be very useful in a simulation environment to disable property checking until the design under verification is in a stable initialized state. This feature eliminates the need to use the "**disable iff** (reset_n)" in every property to prevent false reporting when a user does not need to verify the assertions during initialization. The syntax for the assertion control tasks is as follows:

```
assert_control_task ::=   assert_task [ ( levels [ , list_of_modules_or_assertions ] ) ] ;
assert_task ::=
        $asserton
       | $assertoff
       | $assertkill
list_of_modules_or_assertions ::=  module_or_assertion { , module_or_assertion }
module_or_assertion ::=
        module_identifier
      | assertion_identifier
      | hierarchical_identifier
```

- **$assertoff** stops the checking of all specified assertions until a subsequent **$asserton**. An assertion that is already executing, including execution of the pass or fail statement, is not affected.
- **$assertkill** aborts execution of any currently executing specified assertions and then stops the checking of all specified assertions until a subsequent **$asserton**.
- **$asserton** re-enables the execution of all specified assertions.

levels ::= indicates the level of the hierarchy below each specified module instance is to use this assert task. Setting this argument to 0 causes the assert task to apply in the specified module and in all module instances below the specified module. The argument 0 applies only to subsequent arguments that specify module instances.

Figure 5.8.4 demonstrates the application of the **$asserton** / **$assertoff** in a testbench module that instantiates a DUV called *cpu_rtl.*

```
// separate assertion control module that controls the ON/OFF assertions for all
// modules in the design
module assert_control ();
 timeunit 1ns;
 timeprecision 100ps;
 initial begin : disable_assertions_during_reset
    $display ("%0t %m Disabling assertions during init..", $time);
    $assertoff (0, top_tb.cpu_rtl_1);
    @ (top_tb.reset_n === 1'b1);
    $display ("%0t %m Enabling assertions after init..", $time);
    $asserton (0, top_tb.cpu_rtl_1);
 end
endmodule : assert_control
```

```
module top_tb;
 timeunit 1ns;
 timeprecision 100ps;
 logic clk =1'b0, reset_n = 1'b0;
 bus_if b_if;          // Instantiation of interface
 cpu_rtl cpu_rtl_1(clk, reset_n, .*); // Instantiation of cpu module
 assert_control assert_control();  // instantiation of assertion control
 initial forever #30 clk = ~clk;
 initial #91 reset_n=1'b1;
endmodule : top_tb
```

Figure 5.8.4 Application of the $asserton / $assertoff in a Testbench

5.9 CLOCKED SEQUENCE AND MULTI-CLOCKING

5.9.1 Clock Specification for Properties and Sequences

There are a number of ways to specify a clock for a property or a sequence:

- **Sequence declaration with a clock**

  for example:

  ```
  sequence s2; // clock definition within the sequence
     @(posedge clk) a ##2 b;
  endsequence : s2
  //Instantiation of a clocked sequence within a property with no clock definition.
  property p2;
   not s2;
  endproperty : p2
  p2_1: assert property (p2);
  ```

- **Clocked concurrent property**

  For example:

  // The clock definition is within the property and need not be redefined in a sequence or in an instantiated property.

  ```
  property p3;
   @(posedge clk) (c ##2 d);
  endproperty :p3
  p3_1: assert property (p3);
  ```

- **Contextually inferred clock from a procedural block.**

  For example:

  ```
  always @(posedge clk)
      pNoAB_1: assert property (not (a ##2 b));
  ```

- **Clocking block**

  For example:

  ```
  clocking master_clk @(posedge clk);
   property p3;
    not (a ##2 b);
   endproperty : p3
   // other property and sequence declarations here
   // No concurrent assertion statement allowed within the clocking block
  endclocking : master_clk
  ```

```
// Assertions allowed outside the clocking block.
p3_1: assert property (master_clk.p3);
```

Using a default clocking declaration

The properties and assertions following a **default clocking** declaration uses that defaulted clock as the clock for the properties and sequences that follow the declaration, and do not explicitly use a clock. For example:

```
clocking master_clk
  @(posedge clk_master);
endclocking : master_clk



clocking slave_clk
  @(posedge clk_slave);
endclocking : master_clk

// Properties and sequences using master_clk as the default clock
default clocking master_clk ; // master clock as defined above
property p4;
  (a ##2 b);
endproperty : p4
p4_1: assert property (p4);
```

One **clocking** can be specified as the default for all cycle delay operations within a given **module**, **interface**, or **program**.

- **Using explicit clock(s)**

  Single clock properties and assertions

  In single properties and assertions the clock can be defined in either the property, or the sequence, or in both as long as the same clock edge is used. For example:

```
property pOneClock;
  @ (posedge clk) a |=> b ##1 c;
endproperty : pOneClock
apOneClock  : assert property (pOneClock);
apOneClock_ok  : assert property (@ (posedge clk) pOneClock);

sequence qOneClock;
  @ (posedge clk) e ##[1:5] f;
endsequence : qOneClock

property pOneClock2;
  not qOneClock;
endproperty : pOneClock2
apOneClock2  : assert property (pOneClock2);
apOneClock2_ok : assert property (@ (posedge clk) pOneClock2);

property OneClock3;
  @(posedge clk) a |=> qOneClock;
endproperty : OneClock3
apOneClock3  : assert property (pOneClock3);
apOneClock3_ok  : assert property (@ (posedge clk) pOneClock3);
```

same clock edge as in property

```
property pOneClockError;
      a |=> qOneClock; // a has no defined clock!
endproperty : pOneClock2
// There is no clock defined for "a".
apOneClockError : assert property (pOneClockError); // 💣

// There is a clock defined for "a".
apOneClockError_OK : assert property (@ (posedge clk) pOneClockError);
```

No inferred clock for antecedent "a"

Multi-clocked properties

Multi-clocked properties are addressed in section 5.9.3.

5.9.2 Clock Resolution

One of the key requirements of a structured specification language, such as SystemVerilog Assertions, is its capability to express behavior in a hierarchical manner. SystemVerilog Assertions have that hierarchical behavior definition built-in, starting from Boolean expressions that are used in simple sequences and in complex sequences composed of multiple sub-sequences. Those Boolean expressions and sequences can in turn be used to build properties, which can be instantiated with verification directives (e.g., **assert/cover**). Figure 5.9.2-1 demonstrates this hierarchy of assertion constructs.

Figure 5.9.2-1 Hierarchy of Assertion Constructs[14]

[14] "SystemVerilog AssertionsLanguage Overview, Dr. Bassam TabbaraTechnical Manager, R&D Novas Software. Inc. http://www.systemverilog.org/pdf/3a_AssertionsOverview.pdf

5.9.2.1 Clock Resolution in Assertion and Property Directives

The flexibility in the expression of assertion constructs introduces some ambiguity in the resolution of global issues such as clocks and abort conditions. SystemVerilog Assertions resolve the abort condition ambiguity by disallowing nested **disable iff** constructs directly or indirectly. A clock resolution is much more involved.

The general rule for clock resolution is that clocks expressed at the "top most layer" of the hierarchy of assertion win (referring to the "pyramid" above). Thus, a clock specified in the verification directive level precedes over any clocks specified underneath. With this consideration, any clock specified underneath, must be identical to the clock specified in the verification directive.[15] Consider the following examples:

Given the two properties:
property p_with_one_clk; @(**posedge** p_clk) a ##1 b; **endproperty** : p_with_one_clk

property p_with_no_clk; a ##1 b; **endproperty** : p_with_no_clk

Clock in property propagates to verification directive ⇨

clk MUST BE IDENTICAL to **p_clk**

a_p_with_one_clk_1 : **assert property** (p_with_one_clk); ☺

a_p_with_one_clk_2 : **assert property** (@(**posedge** clk) p_with_one_clk);
The assertion a_p_with_one_clk_2 is LEGAL if and only if **clk** is IDENTICAL to **p_clk**;

a_p_with_no_clk_1 : **assert property** (@(**posedge** clk) p_with_no_clk);
a_p_with_no_clk_2 : **assert property** (p_with_no_clk);

Error: 💣 An assertion must have a clock

5.9.2.2 Clock Resolution in Sequences

The next layer of abstraction in the pyramid model is the **sequence**. Sequences are building blocks of SystemVerilog Assertions properties. However, a sequence by itself is a property with a small but significant difference. The clock specified (if any) in a named sequence doesn't propagate to higher-level verification directive while that in a property does. Hence the following is illegal:

[15] See Appendix B for definition of "identical" clocks.

sequence q_with_clk; @(**posedge** seq_clk) a ##1 b **endsequence** : q_with_clk

a_q_with_one_clk_1 : **assert property** (q_with_clk); 💣

Clock needed with named sequence

a_q_with_one_clk_1 : **assert property** (@ (**posedge** clk) q_with_clk); ☺

Concatenating multiple sub-sequences also compose a sequence. This introduces another layer of hierarchy, and the clock resolution happens once again in a "top-down" approach. Specifically, any clock defined in the top most sequence is considered the winner, and all the sub-sequences may or may not have any clock specification in their definitions. However, if the sub-sequences have clock declarations, then those clocks must be identical to the top sequence clock. The following example illustrates this concept.

sub-sequences

//Sequence composed of two named subsequences
sequence top_seq; @(**posedge** seq_top_clk) start ##1 sub_seq_1 ##1 sub_seq_2 ##1 endtop;
endsequence : top_seq

Case 1:
Both the sub-sequences have no clocks in their definitions. E.g.

sequence sub_seq_1; a1 ##1 b1; **endsequence** : sub_seq_1
sequence sub_seq_2; c2 ##1 d2; **endsequence** : sub_seq_2

The resolved clock for *top_seq* is *seq_top_clk*, and the sequence expression is equivalent to:
@(**posedge** seq_top_clk) start ##1 a1 ##1 b1 ##1 c2 ##1 d2 ##1 endtop;

Case 2:
Both the sub-sequences have clocks in their definitions. e.g.,

sequence sub_seq_1; @(**posedge** s_clk1) a1 ##1 b1; **endsequence** : sub_seq_1
sequence sub_seq_2; @(**posedge** s_clk2) c2 ##1 d2; **endsequence** : sub_seq_2

The resolved clock for *top_seq* is *seq_top_clk*. An additional restriction is that both s_clk1 and s_clk2 must be identical to seq_top_clk, otherwise the *top_seq* is considered illegal.

The resolved clock for *top_seq* is *seq_top_clk*, and the sequence expression is equivalent to:
@(**posedge** seq_top_clk) start ##1 @(**posedge** s_clk1) a1 ##1 b1
@(**posedge** s_clk) ##1 c2 ##1 d2 ##1 endtop;

If *seq_top_clk, s_clk1, s_clk2* are **identical**, then it's a legal sequence. ☺
If *seq_top_clk, s_clk1, s_clk2* are **NOT identical**, then it's illegal sequence. 💣

Case 3:

Either of the sub-sequences has a clock in its definition. E.g.

sequence sub_seq_1; @(**posedge** s_clk1) a1 ##1 b1; **endsequence** : sub_seq_1
sequence sub_seq_2; c2 ##1 d2; **endsequence** : sub_seq_2

Just like case2, the resolved clock for *top_seq* is *seq_top_clk*. An additional restriction is that s_clk1 must be identical to seq_top_clk, otherwise the *top_seq* is considered illegal.

Once again it is worth re-iterating that the *seq_top_clk* in the sequence *top_seq* is NOT propagated to a verification directive directly enclosing it.

5.9.3 Multiple clocked sequences

Multiple clocked sequences[16] are built by concatenating singly-clocked subsequences using the single-delay concatenation operator ##1. Note that only one ##1 delay is allowed in the separation between multiple clocked sequences. This operator is non-overlapping and synchronizes between the clocks of the two sequences. The single delay indicated by ##1 is understood to be from the endpoint of the first sequence, which occurs at a tick of the first clock, to the nearest strictly subsequent tick of the second clock, where the second sequence begins.

Before getting into the rules and restrictions of expressing multiply-clocked sequences, the following example demonstrates an application of a multiple clocked subsystem. The architecture of the subsystem is shown in Figure 5.9.3-1. Subsystem 1 is clocked at *clk1* clock. Subsystem 2 is clocked at *clk2* clock. Subsystem 1 makes a transaction request *request_ck1* to subsystem 2. To avoid metastability issues, subsystem 2 reclocks signal *request_ck1* through two flip-flops using subsystem clock *clk2*. For simplicity, that request becomes *ready* after the two synchronization flip-flops. *Ready* is then clocked through one more stage of flip-flop, and becomes *acknowledge_ck2*, the acknowledge sent to subsystem 1. The *acknowledge_ck2* is also reclocked through two levels of flip-flops, and becomes *grant_ck1* in subsystem 1.

Figure 5.9.3-1 Multiple Clocks Subsystem Example

The sequence for this whole transaction through both subsystems can be expressed as follows:

[16] LRM 17.12.1 calls that topic "Multiply-clocked sequences". We feel that "multiply" is a mathematical operation, and that "multiple" better expresses the notion of multiple sequences, each clocked with different (or same) clock.

```
sequence qRequest_ck1_ck2
    (clk1, clk2, request_ck1, ready_ck2, acknowledge_ck2, grant_ck1);
  @ (posedge clk1) request_ck1 ##1          // now transitioning to next edge of clk2
  @ (posedge clk2) `true[*3:4] ## 1 ready_ck2 ##1
                   acknowledge_ck2 ## 1     // now transitioning to next edge of ck1
  @ (posedge clk1) `true[*3:4] ##1 grant_ck1;
endsequence : qRequest_ck1_ck2
```

An application of this sequence might be a property that uses an antecedent to trigger the sequence. For example:

```
property pRequestToGrant;
 @ (posedge clk1) make_handshake |=> // a make request control
        qRequest_ck1_ck2(clk1, clk2, request_ck1, ready_ck2, acknowledge_ck2, grant_ck1);
endproperty : pRequestToGrant

apRequestToGrant_1: assert property (pRequestToGrant);
```

The timing for this sequence is shown in Figure 5.9.3-2

Figure 5.9.3-2 Timing for Sequence qRequest_ck1_ck2

The analysis of this sequence is shown below:

sequence qRequest_ck1_ck2 (clk1, clk2, request_ck1, ready_ck2, acknowledge_ck2, grant_ck1); @ (**posedge** clk1) request_ck1 ##1 @ (**posedge** clk2)	→ Name of the sequence → Formal arguments → The next subsequence is tested at ***posedge*** *clk1* → *request_ck1* must hold @ (**posedge** clk1) → Because the ##1 is followed by @ (**posedge** clk2), a different clock than the one used in the previous subsequence, the ##1 moves the time to the nearest strictly subsequent **posedge** clk2. When transitioning between clocks, only the ##1 is allowed.
`true[*3:4] ## 1 ready_ck2	→ The next subsequence is now tested at ***posedge*** *clk2*. In this case, because of the metastability flip-flops, signal *ready_ck2* is expected to occur 3 to 4 clock clk2 cycles, starting the count from the first edge of clk2 following clk1. Thus, TRUE is repeated 3 to 4 cycles until *ready_ck2* occurs.
##1 acknowledge_ck2 ## 1 @ (**posedge** clk1)	→ Delay of 1 clock cycle with ck2 → Signal *acknowledge_ck2* must hold @ ***posedge*** *clk2* → Now transitioning to next edge of ck1 because of ##1 @ (**posedge** clk1)
`true[*3:4] ##1 grant_ck1;	→ The next subsequence is now tested at ***posedge*** *clk1*. In this case, because of the metastability flip-flops, signal *grant_ck1* is expected to occur 3 to 4 clock clk1 cycles, starting the count from the first edge of clk1 following clk2. Thus, TRUE is repeated 3 to 4 cycles until *grant_ck1* occurs.
endsequence : qRequest_ck1_ck2	→ end of sequence

5.9.3.1 Rules in Using Multiple-Clocked Sequence

Multiple-clocked sequences are very powerful, but they also pose the danger of introducing ambiguous behavior of sequences especially at the clock boundaries. SystemVerilog LRM rigorously defines a set of rules to avoid any such ambiguity. The following list summarizes the key rules in using sequences with different clocks.

1. **Clock domain boundary specification**

   Whenever a clock domain is being crossed in a multiple-clocked sequence, the boundary sub-sequences must be concatenated with **##1.** The rationale behind this rule is that if there are more than 1 clock ticks permitted at the boundary (e.g., N clock ticks), there will be an ambiguity as to on what clock should this number N be counted.

   A sequence that follows a clocking event (e.g., @ (**posedge** clk)) uses that clocking event for all subsequences until a ##1 new clocking event occurs. For example, with:

   @(**posedge** clk0) siga0 ##[1:5] sigb0 — siga0, sigb0 use **posedge** clk0
   ##1 @(**posedge** clk1) siga1[*1:2] ##1 sigb1; — siga1, sigb1 use **posedge** clk1

Note: If clk0 and clk1 are not identical, then the clocking event for the sequence changes after ##1. If clk0 and clk1 are identical, then the clocking event does not change after ##1 and the above sequence is equivalent to a singly-clocked sequence:

@(**posedge** clk0) siga0 ##[1:5] sigb0
##1 siga1[*1:2] ##1 sigb1;

Differently-clocked or multiple-clocked sequence operands cannot be combined with any sequence operators other than ##1. For example, if *clk1* and *clk2* are not identical, then the following statements are illegal:

@(**posedge** clk1) s1 ##0 @(**posedge** clk2) s2 // Illegal because of ##0

@(**posedge** clk1) s1 ##2 @(**posedge** clk2) s2 // Illegal because of ##2

@(**posedge** clk1) s1 **intersect** @(**posedge** clk2) s2
// Illegal because of ambiguity in ending of clocking event

Note:

@(**posedge** clk1) s1 ##4 @(**posedge** clk2) s2 // Illegal because of ##4 (need a ##1)
can be made legal as follows:
@(**posedge** clk1) s1 ##1 @(**posedge** clk2) `true[*3] ##1 s2 // legal because of ##1 ✓

2. **No empty match matches in the clock boundary.**

With repetition operators, it is possible to create "empty" matching sequences, such as s1[*0:10]. While such form of sub-sequences works well inside a single clock domain, at the cross clock domain boundary, such an empty match can lead to ambiguous results. When concatenating differently-clocked sequences, the maximal singly-clocked subsequences are required to admit only non-empty matches. Thus, if s1, s2 are sequence expressions with no clocking events, and *clk1* and *clk2* are not identical then the following is illegal:

@(**posedge** clk1) s1 ##1 @(**posedge** clk2) s2[*0:1]

Illegal because of the possibility of an empty match of s2[*0:1], which would make ambiguous whether the ending clocking event is **posedge** clk0 or **posedge** clk1.

Multiple-clocked sequence "@(**posedge** clk1) s1 ##1 @(**posedge** clk2) s2"
is legal only if neither *s1* nor *s2* can match the empty word. For example, the following is illegal:

sequence s1; b[*0:5]; **endsequence** : s1
sequence m; @(**posedge** clk1) s1 ##1 @(**posedge** clk2) c ##1 d; **endsequence** : m
// The following expanded sequence for m has an empty word:
@(**posedge** clk1) b[*0] ##1 @(**posedge** clk2) c ##1 d;

sequence s2; c[*0:2]; **endsequence** : s2
// The following expanded sequence for m has an empty word:
@(**posedge** clk1) a ##1 @(**posedge** clk2) c[*0];

Empty Word b[*0] means *b* is repeated zero times (i.e., non existent)

5.9.4 Multiple-clocked properties

A multi-clocked property evaluates to either TRUE or FALSE and has one of the following characteristics:

1. Includes one or more multiple-clocked sequences.
2. Uses the Boolean property operators (**not**, **and**, **or**) to combine singly- and multiple-clocked properties. For example:

   property p_multiple_prop; (@(**posedge** clk0) siga0 |=> ##[1:5] sigb0) **and**
   (@(**posedge** clk1) siga1 |=> ##2 sigb1 ##[1:10] sigc1);
   endproperty : p_multiple_prop

3. Uses the non-overlapping implication operator |=> to create a multiple-clocked property from an antecedent sequence and a consequent property that are differently- or multiple-clocked. For example, if s0, s1 are sequences with no clocking event, then:

   property p_mprop;
   @(**posedge** clk0)
   s0 |=> @(**posedge** clk1) s1;
   endproperty : p_mprop

   The |=> synchronizes between **posedge** clk0 and **posedge** clk1. Starting at the point at which the implication is being evaluated, for each match of s0 clocked by clk0, time is advanced from the end point of the match to the nearest strictly future occurrence of **posedge** clk1, and from that point there must exist a match of s1 clocked by clk1. This is shown graphically in Figure 5.9.4.

Figure 5.9.4 Muliply Property Timing
property p_mprop; **@(posedge clk0) s0 |=> @(posedge clk1) s1; endproperty**

4. When the overlapping implication operator is used, the clock for the end of the antecedent must be the same as the clock for the beginning of the consequent. For example, if clk0 and clk1 are not identical and s0, s1, s2 are sequences with no clocking events, then

   @(**posedge** clk0) s0 |-> @(**posedge** clk1) s1 ##1 @(**posedge** clk2) s2;
   is illegal because the endpoint of sequence s0 is clocked with *clk0*, but the start of the sequence s1, in the same cycle used clock *clk1*.

@(**posedge** clk0) s0 |-> @(**posedge** clk0) s1 ##1 @(**posedge** clk2) s2; ✓
is legal because the endpoint of sequence s0 is clocked with *clk0*, and the start of the sequence s1, in the same cycle used the same clock *clk0*. The multiple sequence s2 is synchronized to the first ***posedge*** of *clk2* following the endpoint of sequence *s1*.
NOTE: The rule for non-empty match of sequences also applies here, i.e., neither s1 nor s2 can have empty matches.

5.9.5 Clock flow

When no explicit clock event is specified in an implication operator, the clock from the endpoint of the antecedent is understood to flow across the operator. Thus, the following to property expressions are identical:

@(**posedge** clk0) s0 |-> @(**posedge** clk0) s1 ##1 @(**posedge** clk2) s2; ✓

@(**posedge** clk0) s0 |-> s1 ##1 @(**posedge** clk2) s2; ✓

Clk0 implicitly flows across the implication operator. s1 is clocked with **posedge** clk0

Similarly, a property expression of the form
if (expression_or_dist **)** property_expr [**else** property_expr]
has a restriction that the clock does not change from the Boolean condition *in the* **if** statement to the beginnings of the **if** and **else** clause properties. Thus:

@(**posedge** clk) **if** (*b*) @(**posedge** clk) w ##1 @(**posedge** clk2) *y*
else
@(**posedge** clk) *r* ##1 @(**posedge** clk2) *m;*

Same clock event for *b, w, r*

The above property expression can be written as:

@(**posedge** clk) **if** (*b*) *w* ##1 @(**posedge** clk2) *y*
else
r ##1 @(**posedge** clk2) *m;*

Same clock event, because clock is not allowed to change

The scope of a clocking event flows into parenthesized subexpressions and, if the subexpression is a sequence, also flows left-to-right across the parenthesized subexpression. However, the scope of a clocking event does not flow out of enclosing parentheses. For example, in

clk2 trapped in the parentheses

@(**posedge** clk1) *w* ##1 (*r* ##1 @(**posedge** clk2) *y*) |=> *m;*

Same clock event , *clk1*

w, *r*, *m* are clocked at ***posedge*** *clk1* and *y* is clocked at ***posedge*** *clk2*. Clock ***posedge*** *clk1* flows across ##1, across the parenthesized subsequence (*r* ##1 @(**posedge** clk2) *y*), and across the non-overlapping implication |=> operator. Clock ***posedge*** *clk1* also flows into the parenthesized subsequence, but it does not flow through @(***posedge*** *clk2*). Clock ***posedge*** *clk2* does not flow out of its enclosing parentheses.

In the following example,

w, *r*, *m*, *n* are clocked at ***posedge*** *clk1*, and *y* is clocked at ***posedge*** *clk2*. Clock *clk1* flows across the **non-overlapping** operator |=>, distributes to both operands of the property **and** operator, and flows into each of the parenthesized subexpressions. Within *(r ##1 @(**posedge** clk2) y), clk1* flows across ##1 but does not flow through *@(**posedge** clk2). Clock **posedge** clk2* does not flow out of its enclosing parentheses. Within (*m* ##1 *n*), ***posedge*** *clk1* flows across ##1.

Note that juxtaposing two clocking events nullifies the first of them. Thus,
*@(**posedge** clk2) @(**posedge** clk1) m* is equivalent to *@(**posedge** clk1) m* because the flow of clock ***posedge*** *clk2* is immediately overridden by clock **posedge** clk1.

5.9.6 Clocking Rules in Assertions

- **Rule 1:** A clocking event specified inside a SEQUENCE declaration IS NOT propagated to enclosing **verification** directive (e.g., **assert, cover, assume**).
- **Rule 2:** A clocking event specified inside a PROPERTY declarations IS propagated to enclosing **verification** directive (e.g., **assert, cover, assume**).

Assuming the given properties:

```
property p_no_clock; a |=> b; endproperty : p_no_clock
property p_with_one_clock; @ (posedge clk) a |=> b; endproperty : p_with_one_clock
sequence qWith_one_clock;  @ (posedge clk) a ##1 b; endsequence : qWith_one_clock
```

The following assertion / cover declarations ARE legal:

```
ap_with_one_clock : assert property (p_with_one_clock);          ☺ // Rule 2
cp_with_one_clock : cover property (p_with_one_clock);           ☺ // Rule 2
cq_with_one_clock_OK : cover property (@ (posedge clk) qWith_one_ck); ☺ // Rule 1
// clock declaration in property expression is needed
```

The following assertion / cover declarations ARE NOT legal:

```
ap_qWith_one_clock_BAD : assert property (qWith_one_clock);        // Rule 1
// clock declaration in the sequence does not propagate to verification directive
cp_qWith_one_clock_BAD : cover property (qWith_one_clock);         // Rule 1
// clock declaration in the sequence does not propagate to verification directive
```

5.9.6.1 Single clocked assertions:

The clock for an assertion statement is determined in the decreasing order of priority:

1) Explicitly specified clock for the assertion. For example:

```
p_no_clock_1: assert property (@ (posedge clk) p_no_clock);
p_with_one_clock_1: assert property (p_with_one_clock);
// Also allowed
p_with_one_clock_1: assert property (@ (posedge clk) p_with_one_clock);
```

Clock event can be explicit in assertion or in property . Note that the clocking event in the property expression is the same *clk* signal as in *p_with_one_clk*

2) Inferred clock from the context of the code when embedded. For example:

```
always @ (posedge clk)
```

Clocking event

```
p_no_clock_1: assert property (p_no_clock);
// Also allowed
p_with_one_clock_1: assert property (p_with_one_clock);
```

3) Default clock, if specified. For example:

```
clocking sys_clk @(posedge clk); endclocking : sys_clk
```

Clocking event

```
default clocking sys_clk ;
p_no_clock_1: assert property (p_no_clock);
// Also allowed
p_with_one_clock_1: assert property (p_with_one_clock);
```

No clock in property

5.9.6.2 Sequence and Properties in Clocking Blocks

Sequences and properties specified in clocking blocks resolve the clock by the following rules:

1) Event control of the clocking block specifies the clock. For example:

```
clocking sys_clk @(posedge clk);
   property p_no_clock; a |=> b; endproperty : p_no_clock ✓
endclocking : sys_clk
```

2) No explicit event control is allowed in any property or sequence declaration. Thus, the following is illegal:

Explicit clock

```
clocking sys_clk @(posedge clk);
   property p_with_one_clock; @ (posedge clk) a |=> b;
   endproperty : p_with_one_clock
endclocking : sys_clk
```

3) If a named sequence that is defined outside the clocking block is used, its clock, if specified, must be identical to the clocking block's clock. For example:

clocking sys_clk @(**posedge** clk);
property p_ok; **not** s_with_one_ck; **endproperty** : p_ok; ✓
property p_not_ok; **not** s_with_one_ck2; **endproperty** : p_not_ok; 💣
endclocking : sys_clk

Uses clk2 Instead of sys_clk 💣

4) Multi-clock properties are not allowed. Thus,

clocking sys_clk @(**posedge** clk);
property p_not_ok; a ##1 @ (**posedge** clk2) b ; **endproperty** : p_not_ok; 💣
endclocking : sys_clk

Uses multi-clocks (sys_clk and clk2) 💣

5.9.6.3 Multiple-clocked Assertions

For a multiple-clocked assertion, the clocks are explicitly specified and no default clock or inferred clock is used. In addition, multiple-clocked properties cannot be defined within a clocking block. For example:

property pBadMulticlock; a ##1 @ (**posedge** clk2) b ; **endproperty** : pBadMulticlock ☹
property pOkMulticlock; @ (**posedge** clk1) a ##1 @ (**posedge** clk2) b ;
endproperty : pOkMulticlock ☺

No clock for *a*

*// Attempt to use the clock in the **always** or **initial** block*

always @(**posedge** clk) **assert property** (pBadMulticlock); 💣
initial @(**posedge** clk2) **assert property** (pBadMulticlock); 💣

Clocks are not explicitly specified

//concurrent assertion

ap_multiclock : **assert property** (pOkMulticlock); ✓

Clocks are explicitly specified

A multiple-clocked property **assert** statement must not be embedded in procedural code where a clock is inferred. For example, following forms are not allowed.

always @(clk) **assert property** (pBadMulticlock); 💣
initial @(clk) **assert property** (pOkMulticlock); 💣

5.10 SYSTEMVERILOG SCHEDULING SEMANTICS FOR ASSERTIONS

SystemVerilog defines an explicit scheduling semantics to avoid race conditions between the sampling of signals used in assertions, and the update of signals by SystemVerilog code in blocking or nonblocking assignments.[17] In this section, a closer look into the scheduling semantics is presented from an assertion viewpoint. The motivation is to make readers understand when the assertions are checked, and to explain when any "reactive" actions (defined in action-blocks of assertions) are executed. SystemVerilog defines several time slot regions including:

---

[17] This explicit scheduling semantics in SystemVerilog is an important feature that is lacking when using PSL assertions with Verilog models.

1) Preponed	current time slot before any net or variable has changed state.
2) Pre-active	Support for PLI callbacks.
3) Active	current events being evaluated and can be processed in any order.
4) Inactive	events to be evaluated after all the active events are processed.
5) Pre-NBA	Support for PLI callbacks
6) NBA	nonblocking assignment region, scheduled for current or a later simulation time.
7) Post-NBA	Support for PLI callbacks
8) Observed	evaluation of the property expressions when they are triggered.
9) Post-observed	Support for PLI callbacks
10) Reactive	Action block evaluations
11) Postponed	Support for PLI callbacks

All scheduled events at a specific time form an event queue at that time slot. Simulation proceeds by executing and removing all events in the event queue in the current simulation time slot before moving on to the next non-empty time slot, in time order. Figure 5.10-1 summarizes the SystemVerilog flow of time slots and event regions using an example.

Figure 5.10-1 The SystemVerilog flow of time slots and event regions[18]

[18] "SystemVerilog Assertions Language Overview, Dr. Bassam TabbaraTechnical Manager, R&DNovas Software, Inc. http://www.systemverilog.org/pdf/3a_AssertionsOverview.pdf

SystemVerilog also allows for the specification of input and output skews. Input (or inout) signals are sampled at the designated clock event. If an input skew is specified then the signal is sampled at *skew* time units *before* the clock event. Similarly, output (or inout) signals are driven *skew* simulation time units *after* the corresponding clock event. An example of a skew definition is shown below:

```
clocking master_clock @(posedge clk);
  input #1ps address;
  input #5 output #6 data;
endclocking : master_clock
```

SystemVerilog Assertions are evaluated in the "observe" region. The signals used in these assertions must use their correct values in order for the assertions to behave as expected. In real hardware, due to setup time and clock-to-Q delay considerations, the value will be that of the previous clock cycle. SystemVerilog mimics that behavior by using the values (for signals inside assertions) sampled during the Preponed region of the time step skew time-units prior to the clocking event. The input-skew-time is typically either:

- Left unspecified (in which case the signals are sampled at the Preponed region with a default input skew of 1 step).
- Specified as a positive number (> 0, e.g. #1ps).

If the input skew time step is not an explicit #0 (i.e., the time step is either explicitly defined to non zero (e.g., ##1 ps), or it is not explicitly defined) then the value sampled corresponds to the signal value at the Postponed region of the time step skew time-units prior to the clocking event. A non-explicitly defined skew represents zero skew at the Preponed region. Figure 5.10-2 shows the sample and drive time, including skew with respect to the positive edge of the clock.

NOTE: If the input skew is explicitly specified as #0, then the signal values will correspond to that in the Observe region. Explicitly specifying an input skew of ##0 can lead to unexpected results, and should be avoided. ☹

Figure 5.10-2 — Sample and drive times including skew with respect to the positive edge of the clock

The following key points need to be clarified for a positive edge clocking system:

1. The signals are sampled prior to the rising edge of the clock in the Preponed region.
2. Blocking assignments and nonblocking assignments (NBA) are processed in their own regions in a specified order, as specified in the LRM.
3. Thereafter, the properties (i.e., assertions) are evaluated. Any action blocks are evaluated after the assertions.

5.11 PROPERTIES IN INTERFACES

The interface construct in SystemVerilog is specifically created to encapsulate the communication between blocks, allowing a smooth migration from abstract system-level design through successive refinement down to lower-level register-transfer and structural views of the design. The interface construct is explained in the LRM and in several books, and is not the scope of this book. However, properties and verification directives (e.g., **assert**) can be specified in interfaces. For example, Figure 5.11-1 demonstrates a generic interface for a common handshake, with bus request, acknowledge, data during the request, and output data during the acknowledge cycle.

Figure 5.11-1 Handshake Interface

The interface can be described as shown in Figure 5.11-2.

```
// Handshaking Interface with properties
interface req2send_if (input bit clk, reset_n);
 timeunit 1ns;   timeprecision 100ps;
 logic request, ack;  // signals belonging to interface
 logic [7:0] source_data, data_out; // signals belonging to interface
 // Define a clocking for the properties and IO ports
 clocking reqclk @(posedge clk);
   input ack, data_out,
   output request, source_data;

  // Using the clock defined in clocking, define the property
   property p_req2send (req, datain, ack, dataout);
    logic [7:0] v_data; // local assertion variable
     (req, v_data=datain) |=> // if req then assertion variable v_data = datain
        ##[0:3] (ack  && // within next cycle to 3 cycles later an ack and
        dataout ==v_data); // dataout must be equal to v_data
   endproperty : p_req2send
  endclocking : reqclk

 // Define port direction for a module that sends data
   modport sendmode_if (
    output ack, data_out,
    input request, source_data);
```

```
// Define verification directives
a p_req2send_1: assert property (p_req2send(request, source_data, , ack, data_out));
endinterface : req2send_if
```

Figure 5.11-2 Handshake Interface Code

Chapter 6 provides another application of properties and assertions in interfaces.

5.12 VERIFICATION DIRECTIVES

Verification directives inform the verification tools about the actions to be taken for the specified properties. A property on its own is never evaluated for checking an expression. A property is a declaration, and nothing else. A verification directive dictates the verification function to be performed on the property. It can be one of the following:

1. **assert** Specifies the property as a checker for verification of design.
2. **assume** Specifies the property as an assumption for the verification environment..
3. **cover** Monitors the property evaluation for functional coverage.

In addition to the above, this section addresses the **expect** construct as well. Even though the **expect** construct is not a verification directive, the **expect** construct makes use of a property.

A concurrent assertion statement is an **assert**, an **assume**, or a **cover** statement. Concurrent assertion statements outside of procedural code or embedded in a procedural block. This distinction is explained with examples in section 5.12.1.

5.12.1 assert Directive

The assert directive is used to enforce a property as a checker. The syntax for the **assert** directive is:

Concurrent assertion:

assert_property_statement::= **assert property** (property_spec) action_block
action_block ::= statement_or_null | [statement] **else** statement_or_null

5.12.1.1 Concurrent Assertion Statements Outside of Procedural Code

Assertions declared procedural code start the checking of the properties from the beginning to the end of simulation, unless disabled by assertion-control system tasks.[19] For example:

```
Examples:
module a();
logic clk, req, ack, ready;
logic [1:0] state;

property pReadyReq; // property declaration
  @ (posedge clk) ready |=> ##[0:5] req;
endproperty : pReadyReq
apReadyReq : assert property (pReadyReq); // concurrent assertion
...
endmodule : a
```

[19] See section 5.8.4 Assertion-Control System Tasks

5.12.1.2 Concurrent Assertion Statements Embedded in Procedural Block

A concurrent assertion statement can also be embedded in a procedural block. For example:

```
property p_test;
  k |=> a ##1 b ##1 c;
endproperty : p_ test

always @(posedge clk) begin
 <statements>
  p_ test: assert property (p_ test);
end

initial @(posedge clk) begin
<statements>
apNotDF : assert property (not (d &&f ));
end
```

* No clock statement unless same as block clock statement.
* the property is always monitored.
* Clock item must be first term of the event control

monitoring is performed only on the first clock tick.

Another inference made from the context is the enabling condition for a property. Such derivation takes place when a property is placed in an **if...else** block or a **case** block. The enabling condition assumed from the context is used as the antecedent of the property. For example:

```
property pReqAck;
    req |=> ##[0:10] ack;
endproperty : pReqAck

always @(posedge clk) begin
  if (enb) begin
    q <= d1;
    apReqAck : assert property (pReqAck);
  end
end
```

The above example is equivalent to:

```
property pReqAckEnb;
 @(posedge clk) enb |-> req |=> ##[0:10] ack ;
endproperty : pReqAckEnb
apReqAckEnb : assert property (pReqAckEnb);
```

Another example with the **case** statement:

```
always_ff @(posedge clk)
 begin
 case (state)
  2'b00 :  begin
    state <= 1;

    assert property ( req |=> ##[0:10] ack);
  end
  default : state <= 0;
 endcase
end // always
```

* No timing controls, e.g., @(posedge clk)
* No tasks with timing control. e.g., delay_taks(2);

Assertion is outside looping construct e.g., **for** (i=0; i<=3; i=i+1) **assert property** (req |=> ##[0:10] ack);

Above is equivalent to:

```
property pReqAckState00;
 @(posedge clk) state==2'b00 |-> req |=> ##[0:10] ack ;
endproperty : pReqAckState00
apReqAckState00 : assert property (pReqAckState00);
```

The enabling condition is inferred from procedural code inside an **always** or **initial** block, with the following restrictions:

1. There must not be a preceding statement with a timing control.
2. A preceding statement shall not invoke a task call that contains a timing control on any statement.
3. The concurrent assertion statement shall not be placed in a looping statement, immediately, or in any nested scope of the looping statement.

5.12.1.3 Immediate assertion:

immediate_assert_statement ::= **assert** (expression) action_block

The expression in non-temporal. An immediate assertion is used in an **always** or **initial** block, and is used to verify the value of an expression embedded in RTL code, such as in an FSM. For example:

```
case (state_reg)
 OFF: begin
        if (enb) next_state <= RED;
          assert (sensors==DISABLED) else begin
              t = $time;
              #5 $error("assert failed at time %0t, sensors=%h",t, sensors);
          end
        end
```

5.12.1.4 Action-block

When the property for the **assert** statement is evaluated to be true, the pass statements of the action block are executed. Otherwise, the fail statements of the *action_block* are executed. For example, given the above property *pReadyReq*:

```
apReadyReq : assert property pReadyReq  else
  begin
   $display ("%m @ time %t, pReadyReq Failure ", $time);
   $error ("Handshake ERROR ");
    hdk_err <= 1'b1; // This can cause the testbench to take a reactive action
  end;
```

Since the assertion is a statement that asserts something must be true, the failure of an assertion shall have a severity associated with it. By default, the severity of an assertion failure is **error**. Other severity levels can be specified by including one of the following severity system tasks in the fail statement (see section 3.6.3).

Note:

1. If no statement is specified for **else**, then **$error** is used as the statement when the assertion fails.
2. The *action_block* shall not include any concurrent **assert, assume**, or cover statement.
3. The *action_block* can contain immediate assertion statements.
4. The pass and fail statements are executed in the Reactive region, following the processing of the assertions.

5.12.2 assume Directive

During various design phases (e.g., micro-architecture definition, RTL coding etc.), architects and designers make a number of assumptions about the environment in which this design is expected to perform. Traditionally, these assumptions are captured using comments in the RTL code, or as text in requirement or verification planning documents. Assumptions are important because they define the limits and restrictions imposed on the design and supporting tools. In simulation, assumptions impose limits on scenarios. In formal verification, assumptions speedup the formal verification process and avoid false positives (i.e., a false error reporting because the conditions causing the error is not real). To support such features, SystemVerilog provides the **assume** directive.

The purpose of the **assume** directive is to allow properties to be considered as assumptions or constraints for formal analysis, as well as for dynamic simulation tools. The syntax for the **assume** statement is:

assume_property_statement::= **assume property** (*property_spec*) ;

Unlike the **assert** directive, the **assume** does not support an action block. When a property is assumed, the tools constrain the environment so that the property holds. The **assume** constrains the behavior of the design inputs. In other words, an asserted property is required to hold only along those paths that obey the assumption.

A constraint is a condition (on the input signals) that limits the set of behavior to be considered. A constraint may represent real requirements (e.g., clocking requirements) on the environment in which the design is used, or it may represent artificial limitations (e.g., mode settings) imposed in order to partition the verification task.

In some cases, constraints can be interpreted as properties to be proved. For example an input constraint associated with one module can also be an output property of the module driving this input. Therefore, properties and constraints may be *dual* in nature.

The exact way in which an **assume** construct is handled depends partly on the verification methodology. A SystemVerilog aware simulator might treat them as checkers to monitor the input stimuli. In case of a random test generator, the **assume** constructs can be used as constraints to direct the generator to generate only legal scenarios.

The role of the **assume** construct is more visible and valuable in formal verification. Formal verification tools consider all possible allowed input combinations when performing checks and working on a *proof*, and hence need to constrain the inputs to their *legal* behavior. Otherwise, the tool would either report *false negatives* or may not terminate at all.

Consider the following property:

```
property pGetDataWhenRequest(request, grant, grab_data);
  @ (posedge clk) $rose(request) |=> request && grab_data  && grant;
endproperty : pGetDataWhenRequest
```

The **assert** statement requires that the property succeeds, and if it does not, then it is a failure.

```
ap_GetDataWhenRequest : assert property (pGetDataWhenRequest(req, grant, ld_data));
```

In a good system, the *req* will be held high till the *grant* is received. However, since *req* is an input, without any assumption on the behavior of *req*, a formal verification tool would show a failure for this assertion because *req* can be deasserted in the cycle following the assertion of *req*. As an input, *req* can take any legal value. However, the subsystem maintains *req* until *grant*. Thus, the following values for *req* at each clock cycle is legal: 0 0 1 0 0. With this sequence of values, the assertion *ap_GetDataWhenRequest* would fail. Such failures are referred to as False Negatives.

Therefore, for formal verification, it becomes necessary to impose a set of assumptions as to the values of the inputs. This can be done using the **assume** verification directive, as shown below:

The property *pHoldRequestUntilGrant* requires that a *request* be held until a *grant*.

```
property pHoldRequestUntilGrant(request, grant);
  @ (posedge clk)  $rose(request) |=> request[*0:$] ##1 grant;
endproperty : pHoldRequestUntilGrant
mpHoldRequestUntilGrant_1 : assume property (pHoldRequestUntilGrant(req, grant));
```

This assumption constrains the formal verification tool such that if *req* is asserted, then it must remain asserted until *grant* is valid.

Notes:

1. For formal analysis, there is no obligation to verify that the assumed properties hold.
2. An assumed property can be considered as a hypothesis to prove the asserted properties.
3. For simulation, the environment must be constrained such that the properties that are assumed shall hold. Like an **assert property**, an **assumed property** must be checked and reported if it fails to hold. There is no requirement on the tools to report successes of the assumed properties. In essence, the **assume property** behaves like an **assert property** without the action block.
4. Biasing the inputs provides a way to make random choices. The biasing feature is only useful when properties are considered as assumptions to drive random simulation. When a property with biasing is used in an assertion or coverage, the **dist** operator is equivalent to **inside** operator, and the weight specification is ignored. For example:

   ```
   property p_req_distribution;
     @ (posedge clk) if (req dist {0:=40, 1:=60}) `true ; //40% req==0, 60% req==1
   endproperty : p_req_distribution

   mp_req_distribution_1: assume property (p_req_distribution);
   ```

   In simulation, the instantiation of *req_distribution_1* with *req* having a distribution takes a different meaning, and the **dist** is interpreted as **inside**. This is equivalent to:

   ```
   req_distribution_2: assert property (@ (posedge clk) if (req  inside {0, 1}) `true);
   ```

   The *req* **inside** {0, 1} states that *req* can take the values 0 and 1, with no distribution.

5. If an assumption is classified as "one-shot" (i.e., one-time) to be assumed once at some time during initialization, then insert the **assume** verification directive within an **initial** statement. For example, to assume that reset is low for 5 cycles, and then is high and stays at the high level, one can write:

```
property p_reset_at_init;
   !reset_n[*5] ##1 resetn;
endproperty : p_reset_at_init

initial
begin
 @(posedge clk)
  mp_reset_at_init : assume property (p_reset_at_init);
end
```

```
property pOnceHiStaysHi (s);
@ (posedge clk) s |=> s;
endproperty : pOnceHiStaysHi

// assumed at every clock cycle
OnceHiStaysHi_1 : assume property (OnceHiStaysHi(reset_n));
```

5.12.3 cover Directive

To monitor sequences and other behavioral aspects of the design for coverage, the same syntax as the **assert** construct is used with the **cover** statement. The syntax for the **cover** statement is:

cover_property_statement::=
cover property (property_spec) statement_or_null

Since a property specification can be a property expression or a sequence, the following cover statement represents the coverage of a sequence:
cover property (sequence_expr) statement_or_null

The tools can collect information about the evaluation and report the results at the end of simulation (this is tool dependent). An alternative to the reliance of the tool to provide such statistics is to use the action block to compute the coverage by calling a user-defined task that updates the statistics. The pass statement in the action block must not include any concurrent **assert, assume** or **cover** statement.

Coverage results are divided into two: coverage for properties, coverage for sequences. The results of coverage statement for a property contains:

- Number of times attempted
- Number of times succeeded
- Number of times failed
- Number of times succeeded because of vacuity[20]

In addition, *statement_or_null* is executed every time a property succeeds.

---

[20] Vacuity rules are applied only when implication operator is used. A property succeeds vacuously if the antecedent is FALSE. A property succeeds non-vacuously only if the consequent of the implication contributes to the success.

Results of coverage for a sequence shall include:

- Number of times attempted
- Number of times matched (each attempt can generate multiple matches)
- In addition, *statement_or_null* gets executed for every match. If there are multiple matches at the same time, the statement gets executed multiple times, one for each match. The following examples of sequences and coverage of those sequences demonstrate that different coverage statistics will be displayed, depending on the use of **first_match.**

sequence qAbcd_many; a ##[1:5] b ##1 c[*1:5] ##1 d; **endsequence** : qAbcd_many
Figure 5.12.3 shows an example of waveforms that have multiple matches for this sequence.

If a first match is needed, then the **first_match** operator needs to be used:
sequence qAbcd_first_only;
first_match(a ##[1:5] b ##1 c[*1:5] ##1 d);
endsequence : qAbcd_first_only

cp_qAbcd_many : **cover property** (@ (**posedge** clk) qAbcd_many);
cp_qAbcd_first_only: **cover property** (@ (**posedge** clk) qAbcd_first_only);

Figure 5.12.3 demonstrates that for the waveforms in the diagram, four **cover** matches are possible for the sequence expression "a ##[1:5] b ##1 c[*1:5] ##1 d". Thus, the number of times that the sequence matched may be higher than anticipated if the first match of that sequence was really intended.

Figure 5.12.3 Example of Multiple Matches for a Sequence

5.12.4 Expect Construct

The **expect** construct is not part of the "verification layer" because it does not make a statement about what should be done with a property in terms of verification. However, the **expect** statement makes use of a property. Specifically, an **expect** statement causes the executing process to block until the given property succeeds or fails. The **expect** statement starts a single thread of evaluation for the given property on the subsequent clocking event, that is, the first evaluation shall take place on the next clocking event. If the property fails at its clocking event, the optional **else** clause of the action block is executed. If the property succeeds the optional pass statement of the action block is executed. The statement following the **expect** is scheduled to execute after processing the Observe region in which the property completes its evaluation. In essence, the **expect** statement is a blocking statement that includes inline a property, and an action is executed based on the result of the evaluation of the property.

The syntax for that statement is:
expect_property_statement ::=
    expect (property_spec) action_block

Why is an **expect** statement useful? The simple answer is that it can replace the evaluation of a complex FSMs using the simpler notation of a property, and then based on the results, take an appropriate action. For example, the following program models a handshake bus request with priority. If the *ack* is not received with zero priority, then the test is retried with a higher priority. If that fails, then a status of FAIL is raised. Figure 5.12.4 demonstrates an application of the **expect** construct.

```
module tst;
 timeunit 1ns;
 timeprecision 100ps;
 logic req, ack, ready, done, retry;
 logic [3:0] priority;
 typedef enum {PASS0, PASS1, FAIL} status_t;
 status_t handshake_status;

 property p_reqack;
        @ (posedge clk)
         $rose(req) ##1 ready[*0:5] ##1 ack;
 endproperty : p_reqack

 always @(posedge clk) begin
        req <= 1'b0;
        for (int i=1; i <= 3; i=i+1)  @ (posedge clk);
        req <= 1'b1;
        priority <= 4'h0; //low priority
        expect (p_reqack) // execute property
         begin
                handshake_status <= PASS0; // pass with priority 0
                req <= 1'b0;  // do this if PASS
         end
         else  // must raise priority, restart, and see if we get an ack
```

Property used in the expect statement. A sequence is used instead of antecedent / consequent to avoid vacuous successes. Thus, if ***$rose(req)*** fails, the property fails.

Property called by the expect statement at next clock cycle, and executed inline. If no success, then change priority and try to access the bus again.

```
        begin
            req <= 1'b0;
            @ (posedge clk);
            priority <= 4'h1; // raise priority level
            req <= 1'b1;
            expect (p_reqack) // execute property
             begin
                handshake_status <= PASS1; // pass with priority 1
                req <= 1'b0;  // do this if PASS
             end
             else handshake_status <= FAIL;
           end
  end
  // ...
endmodule : tst
```

Property called by the expect statement, and executed inline at next clock cycle.

Figure 5.12.4 Sample Application of Expect

The **expect** statement can be incorporated in any procedural code, including tasks or class methods. Because it is a blocking statement, the property can refer to automatic variables as well as static variables.

- Expect is a "procedural" construct, hence only a single evaluation thread is started (unlike a concurrent assert/cover).
- Evaluation of property starts at subsequent clocking event.
- Functional Coverage on **expect** properties are significant to detect the unexpected vacuous success of properties. The **expect** means that the user expected a property to happen at this point in time (since it is a procedural call), hence it should be non-vacuously TRUE.

5.13 BINDING PROPERTIES TO SCOPES OR INSTANCES

One of the recommended methods in Verification is to separate the verification code from the design. Conventional verification methodologies consist of black-box verification approaches, which are only concerned about the interfaces between blocks. Those black-box verification models tend to be defined as separate modules. In SystemVerilog, properties of the interfaces can also be defined in interface design units, which are similar to modules. On a related issue, Assertion-Based Verification methodology also promotes white-box testing where properties and assertions are typically defined inline by the designers to trace the bugs closer to the source.

In designs that include standard protocols or highly reusable control blocks, the assertions are typically written as verification IPs (Intellectual Properties) to be easily reusable across various implementations, including minor variations of a product, or re-spins. A methodology that separates the verification code from the design allows verification teams to independently look at the specification and to develop properties for the system being designed. Since several verification models may be designed, each to address variations in the specifications or different aspects of the verification process, it becomes necessary to bind the verification designs (module, interface, or program) to the design under test (DUT). This binding can be made for specific instances of the DUT, or globally for all instances of the DUT.

The traditional binding method is to instantiate the verification module into the design-under-test (DUT), as shown in Figure 5.13-1. The issues associated with this approach are the need to modify the DUT to use different verification modules as the requirements change, or different verification modules are created.

Figure 5.13-1 Traditional Binding of Verification Module and DUT

SystemVerilog supports the binding of properties specified in modules/interfaces/programs to specific modules or instances. This provides the following benefits:

- Verification of designs with minimum changes to the design or verification code/files.
- Allows a convenient mechanism to attach verification IP to a module or an instance.

The **bind** feature in SystemVerilog is equivalent to writing properties external to a module, using hierarchical path names. With this feature, a user can bind a set of properties declared inside a module, interface, or program instance to a design module or a module instance. Figure 5.13-2 demonstrates a high-level view of such binding.

Figure 5.13-2 Binding of Property Module to Instantiation of Designs

Note:
In the statement "***bind*** *cpu cpu_props cpu_rules1(a,b,c)*;" ports (a, b,c) of the property module get bound to signals (a,b,c) of module *cpu*. Thus, the **bind** operator links a design to a module or interface, such that the bounded module behaves as if it were inserted in the design, with access to signals internal to the design. The actual arguments identify which design signals are accessed by the bounded module.

Some designs may have several verification modules performed by different verification engineers to address specific requirements or specific tests. It then becomes necessary to associate specific binding between those verification modules and the design under test. Figure 5.13-3 demonstrates the binding of several verification modules to instantiation of a design.

Figure 5.13-3 Binding of Property Modules to Instantiation of Designs

The syntax of the `bind` construct is:

```
bind_directive ::=
  bind
    hierarchical_identifier   // e.g., design_rtl module name of the DUT design
                              //   or design_rtl_1 instance name of DUT module
    constant_select           // e.g., xbus_props module
                              //   or xbus_props_if interface
                              // SystemVerilog entity containing properties and assertions
    bind_instantiation ;      // Implicit instantiation of the module or interface that
                              // contains the properties and assertions.

    bind_instantiation ::=
          program_instantiation
          | module_instantiation
          | interface_instantiation
```

An example of the binding construct is demonstrated in Figure 5.13-4

```
module test_rtl (clk, reset, request, ack);
  timeunit 1ns;  timeprecision 100ps;
  logic done;  // done is to test_rtl
  // ..
endmodule : test_rtl

module handshake_props (clk, reset, request, ack, done);
 timeunit 1ns;  timeprecision 100ps;
 ..
 property p_req_ack (a, b, c);
  ..
 endproperty : p_req_ack

 ap_req_ack : assert property  (p_req_ack(request, ack, done)) else
      $display ("%m 1 Request-acknowledge failure");

  // ...
endmodule : handshake_props


module test3_tb;
 timeunit 1ns;  timeprecision 100ps;
  logic clk, reset, request, ack; // signals local to testbench
 // perform 2 instantiations of test_rtl, using 2 styles
 test_rtl test_rtl_1(clk, reset, request, ack, done);
 test_rtl test_rtl_2(.*);

 bind test_rtl              // design under test
        handshake_props    // module/interface containing the properties
          handshake_props_1(.*); // instance of handshake_props that contains the properties.
           // Binding connection to clk, reset, request, ack of test_rtl ports , and to
           //   done internal to test_rtl
           // handshake_props_1 gets bound to the module named test_rtl.
           // Consequently, all instances of design_rtl will contain handshake_props_1.
...
endmodule : test3_tb
```

Figure 5.13-4 Binding Example

Key points to remember while using **bind:**

- When binding properties to a design module, all instances of that module get these properties. This is desirable for library modules such as a FIFO, Arbiter etc.

- When a program (containing set of properties) is bound to a specific design module instance, the program becomes part of that design instance itself (similar to `include, except that it is instance specific), and hence all the assertion related declarations become part of that instance.

5.14 VERIFYING VHDL MODELS WITH SYSTEMVERILOG ASSERTIONS

VHDL has primitive support for Assertion-Based Verification through its built-in *assert* construct. But VHDL doesn't handle temporal assertions as easily as a modern assertion language. With simulators able to handle multiple languages (HDLs: VHDL, Verilog, HVLs: Vera, E, Assertion languages: PSL, SVA, and HDVL such as SystemVerilog), users can exploit the power of an assertion language, such as SystemVerilog Assertions, regardless of the HDL being used to model the design and/or testbench. This section uses VHDL as the underlying HDL and demonstrates the verification concepts with examples. The methods used for binding the SystemVerilog Assertions module to the VHDL module include the direct instantiation or the SystemVerilog "bind" construct.

5.14.1 The Concept

In mixed mode simulation, a SystemVerilog module with SystemVerilog Assertions can be used to verify properties of a VHDL model. The VHDL model ports can be connected to the ports of the SystemVerilog Assertions verification module. Signals internal to the VHDL model can be exported into the SystemVerilog Assertions verification module through a variety of techniques including:

1. Copying VHDL critical signals to signals defined in a package, and routing the signals in the package to the SystemVerilog Assertions verification module.
2. Bringing internal signals to output ports. Pragmas can be used to exclude synthesis from processing the routing of the critical internal signals to the output ports.
3. Use of vendor supplied PLIs to export internal signals into other modules.
4. Use of SystemVerilog **bind** construct, which binds signals visible by VHDL architecture to the SystemVerilog verification module port.

Figure 5.14.1 provides a high level view of the integration of a SystemVerilog Assertions verification module to verify a VHDL model in either a VHDL or a SystemVerilog testbench. These techniques are illustrated in the following subsections.

Figure 5.14.1 VHDL Verification with SystemVerilog Assertions

5.14.2 VHDL Module in VHDL Testbench with SystemVerilog Assertions Module

This section describes the VHDL model example, the SystemVerilog Assertions module, and the VHDL testbench with binding methods of the SystemVerilog Assertions module to the VHDL module being verified. The compilation order of the models is also provided.

5.14.2.1 VHDL Model

A simple model of minimum complexity is used for demonstration purposes.

```
-- Design module
library ieee;
use ieee.std_logic_1164.all;
entity reqack is
 port (
   ack    : out std_logic;        -- acknowledge
   req    : in  std_logic;        -- request
   clk    : in  std_logic);
end entity reqack;

architecture rtl of reqack is
 signal ready : std_logic;
begin  -- architecture rtl
 reqack_proc : process (clk) is
 begin  -- process reqack_proc
  if clk'event and clk = '1' then
   if req = '1' and ready = '1' then
    ack <= '1';
   else
    ack <= '0';
   end if;
  end if;
 end process reqack_proc;
end architecture rtl;
```

5.14.2.2 SystemVerilog Assertions Module

A SystemVerilog module with two properties and assertions is used for demonstration.

```
// SystemVerilog verification module
// with SystemVerilog Assertions
module reqack_sva (input wire ack,
            input wire req,
            input wire ready,
            input wire clk) ;
 timeunit 1ns;  timeprecision 100ps;
 property p_reqack;
  @ (posedge clk)
   (req && ready) |=> ack;
 endproperty : p_reqack
 ap_reqack: assert property (p_reqack);

 property p_noack;
  @ (posedge clk)
   (!req || !ready) |=> !ack;
 endproperty : p_noack
 ap_noack : assert property (p_noack);
endmodule : reqack_sv
```

Properties used in the verification of the design. Other SystemVerilog can be used in conjunction with SystemVerilog Assertions

5.14.2.3 Connecting SystemVerilog Assertions module to VHDL design

Assuming that the testbench is developed in VHDL, the VHDL design is already instantiated in the testbench. The SystemVerilog assertions module needs to be plugged into this environment. There are two methods to make the SystemVerilog Assertions module bound to the VHDL design module:

1. Instantiation of SystemVerilog Assertions module into the VHDL testbench
2. Binding of the SystemVerilog Assertions module to the VHDL design module using the SystemVerilog "bind" directive during the compilation process.

Both of these methods are demonstrated herein.

5.14.2.3.1 Direct Instantiation of SVA module into VHDL Testbench

The VHDL testbench model with the direct instantiation of the SystemVerilog Assertions verification module is shown below. The VHDL testbench does not include the test vector generation because it is not relevant to the goals of this section.

```
-- file reqack_tb.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.all;
entity reqack_tb is
end entity reqack_tb;

architecture top of reqack_tb is
 signal ack  : std_logic;          -- acknowledge
 signal req  : std_logic;          -- request
 signal clk  : std_logic := '1';

-- Component declaration may be optional. It permits compilation of
-- of this architecture while the SystemVerilog Assertions module is being
 -- developed.
 component reqack_sva
   port (ack, req, ready, clk : in std_ulogic);
 end component;

begin -- architecture top
 clk <= not clk after 20 ns;

 -- Instantiation of VHDL module
 reqack_1 : entity work.reqack
  port map (
   ack  => ack,          -- [out std_logic] acknowledge
   req  => req,          -- [in  std_logic] request
   clk  => clk);         -- [in std_logic]
```

```
 -- Instantiation of SVA verification module
  reqack_sva_1 : reqack_sva
   port map (
    ack   => ack,          -- [in wire] acknowledge
    req   => req,          -- [in wire] request
    ready => ready,        -- [in wire] ** Ready is an internal signal!
                           -- need a copy of ready in a package.
    clk   => clk);         -- [in wire]
end architecture top;
```

The compilation order for these models is as follows:

1. reqack.vhd // *The VHDL model*
2. timescale.sv // *A SystemVerilog model to setup the timescale (see Chapter 8)*
3. reqack_sva.sv // *Tthe SystemVerilog Assertions verification model*
4. reqack_tb.vhd // *The VHDL testbench model*

5.14.2.3.2 Binding of SVA Verification Module to VHDL Model

The SystemVerilog bind can be used to bind the SystemVerilog Assertions verification module to the VHDL module. Below is the bind model.

```
//file bind.sv
// Binding of VHDL module "reqack" to
//  SVA verification module "reqack_sva"
//  with instance name "reqack_sva_1"
// All signals are those visible by VHDL module "reqack"
bind reqack  reqack_sva  reqack_sva_1(.ack(ack),
                .req(req),
                .ready(ready),
                .clk(clk));
```

The VHDL testbench does not include any statement about binding, as shown below.

```
-- file reqack2_tb.vhd
architecture top of reqack_tb is
 signal ack   : std_logic;        -- acknowledge
 signal req   : std_logic;        -- request
 signal ready : std_logic;
 signal clk   : std_logic := '1';
begin  -- architecture top
 clk <= not clk after 20 ns;

 -- Instantiation of VHDL module
 reqack_1 : entity work.reqack
  port map (
   ack   => ack,          -- [out std_logic] acknowledge
   req   => req,          -- [in  std_logic] request
   clk   => clk);         -- [in std_logic]

  -- SVA verification module done with the
  -- bind SystemVerilog directive during compilation process
end architecture top;
```

The compilation order for these models is as follows:

1. reqack.vhd // *The VHDL model*
2. timescale.sv // *A SystemVerilog model to setup the timescale (see Chapter 8)*
3. reqack_sva.sv // *The SystemVerilog Assertions verification model*
4. bind.sv // *Binding module of* SystemVerilog Assertions module to VHDL module
5. reqack2_tb.vhd // *The VHDL testbench model*

5.14.3 VHDL Model in a SystemVerilog Testbench with SVA Module

A VHDL model can be instantiated in a SystemVerilog testbench as shown below.

```
//reqack3_tb.sv
module reqack_tb;
 timeunit 1ns;  timeprecision 100ps;
 wire ack ;      // acknowledge
 reg req = 0;      // request
 reg ready = 0;
 reg clk = 1'b1;

 initial #20000 $finish;

 initial forever #50 clk = ~clk;

 // Instantiation of VHDL model
// A mixed-mode simulator will be able to resolve an otherwise unresolved module/entity
//to either VHDL or Verilog automatically.
 reqack reqack_1(.ack(ack),
            .req(req),
            .ready(ready),
            .clk(clk));

 // Instantiation of SVA module for verification
 reqack_sva reqack_sva_1(.*);

//// Optional instead of
//// reqack_sva reqack_sva_1(.*);
//// This bind is exactly same effect of instantation above
//// reqack_sva_1 is the same as reqack_sva_2
//  bind reqack  reqack_sva  reqack_sva_2(.ack(ack),
//                           .req(req),
//                           .ready(ready),
//                           .clk(clk));
endmodule : reqack_tb
```

Required at least once.
Not needed here since
it is in reqack_sva module

The compilation order for these models is as follows:

1. reqack.vhd // *The VHDL model*
2. timescale.sv // *A SystemVerilog model to setup the timescale (see Chapter 8)*
3. reqack_sva.sv // *The SystemVerilog Assertions verification model*
4. reqack3_tb.sv // *The SystemVerilog testbench model*

6 SYSTEMVERILOG ASSERTIONS IN THE DESIGN PROCESS

Assertion-based verification (ABV) with SystemVerilog Assertions is changing the traditional design process as it helps to formally characterize the design requirements at various levels of abstraction, guides the verification task, and eases the design of the testbench. These benefits are derived because ABV transfers more of the verification task onto SystemVerilog Assertions, most of which can be reused across various design phases such as micro-architecture specification, RTL development etc. Assertions can be attached directly to RTL, working in cycle-precise domain, or they can operate in transactional domain with the help of monitors/data extractors. Then, these modular checkers minimize or even eliminate the need of model checkers, which consist of HDL modules to provide the verification. Another benefit of assertions is the ease to specify functional coverage. Simulation tools compute the functional coverage, as defined by the assertions, which add a greater level of assurance that the testbench invoked the desired functions.

This chapter addresses the methodologies in using SystemVerilog properties / sequences / assertions during the requirement and verification planning phases, in addition to the RTL and testbench levels.[1] It first explains the process, and then demonstrates an application of assertions in the requirements specification and verification plan using a synchronous First-In First-Out FIFO. In addition to assertions, SystemVerilog packages, interfaces, modules, and bindings are also demonstrated. The RTL code for this model is demonstrated in section 6.3.3. Section 6.3.5 demonstrates the results of formal verification.

---

[1] This chapter makes use of work done in the book *Component Design by Example* (Ben Cohen, 2001 ISBN 0-9705394-0-1) for the definition of executable specifications.

6.1 Traditional Design Process

The traditional front-end design process consists of the following steps for each design or IP: 1) a requirement specification (with potentially some algorithms in a high level language) written in a natural language (e.g., English); 2) an architectural plan that represents the high level implementation approach; 3) a verification plan that addresses the testbench and verification approach and classes of tests; 4) the RTL design and synthesis; 5) the design verification with details about the testbench and automatic verification; and 6) and the final design documentation and delivery.

The traditional design process relies too much on a natural language such as English as a mean of communication, and that leads to several misunderstandings. In addition, that process does not use an executable, or provable method, to characterize design requirements and restrictions. .

6.2 Design Process with ABV using SVA as vehicle

Figure 6.2-1 represents a typical design process with ABV using SystemVerilog Assertions (SVA). The process consists of the steps described in the following subsections. All designs must have requirements documentation. The requirements can be classified as system-level and module-level.[2] The requirements document can be supplemented with SystemVerilog Assertions to avoid ambiguities caused by a natural language description.

Figure 6.2-1 Typical Design Process with ABV Using SVA

[2] A system consists of several subsystems. The term "system" shall loosely be used in this book to mean a conglomeration of subsystems.

6.2.1 System-level Assertions

SystemVerilog properties can be used to capture system/subsystem level requirements, and those properties can then be reused in a testbench environment later in the design cycle. Figure 6.2.1-1 is a representation of a generic system, consisting of design under consideration in an application environment. System-level requirements can have the following classifications[3]:

- **Functional**: This represents the purpose of the design.
- **Performance:** That consists of items such as baud-rate, frequency, capacity, speed, and latencies.
- **Interface:** Since the design under consideration may interface with existing busses, and connections to other subsystems, the requirements document may include software, hardware, and communications interfaces.
- **Safety:** Many designs are subject to security and safety issues to insure safe operations under certain conditions or failures (e.g., during reset, power-up, bus or unit failure).
- **Operational:** Some designs are subject to user interfaces for commands, configurations, or maintenance.
- **Resource:** Resources are requirements that are critical to a design, and include items such as power, memory, form-fit factor (i.e., size), etc.

Figure 6.2.1-1 Example of a Generic System

[3] From "Document Requirement Description", http://www.estec.esa.nl/gpqwww/paso/padocs/drds/e04.pdf

The performance, interface, and safety classifications can be expressed in SystemVerilog Assertions. It is important to note that requirements are independent of the architecture. However, the architecture is a result of requirements. Figure 6.2.1-2 represents a system view of the design. Transactions can be used to express requirements in a manner independent from the architecture. Transactions can express the attributes of requirements, as described in the following subsections. That view can then easily be translated into a testbench, as explained in section 6.2.1.4. SystemVerilog Assertions can help, via functional coverage, to determine resource usage and needs.

Figure 6.2.1-2 System View of the Design from a SVA Perspective

Figure 6.2.1-3 represents a generic subblock design verification environment with SystemVerilog Assertions.

Figure 6.2.1-3 Generic Subblock Design Verification Environment with SVA Assertions

The following sub-sections demonstrate with a few examples, how SystemVerilog Assertions can be used in the definition of system-level requirements.

6.2.1.1 Cause and Effect Class of Requirements

System-level requirements can usually be represented as cause-effect relationships. In SystemVerilog, the "cause" is called the antecedent, and the "effect" is called the consequent. For example:
Requirement: If the design is subjected to a command to fire the *pyro X* then within 5 cycles the *pyro X* relay shall be activated.

SVA property definition:

```
property pFirePyro;
  @ (posedge clk) xaction.Fire_PyroX_CMD==FIRE |->
          ## [1:5] pyro_sub.PyroX_relay==ACTIVATED;
endproperty : pFirePyro
```

With added safety, one could express the following:[4]

```
property pFirePyroSafe;
        @ (posedge clk) xaction.Fire_PyroX_CMD==FIRE &&
        pyro_sub.power_stable &&
        pyro_sub.reset_cycle== DONE &&
        pyro_sub.armed == OK |->
            ##[1:5] (pyro_sub.PyroX_relay==ACTIVATED;
endproperty : pFirePyroSafe
```

The *xaction* is a definition of the origin of the command. It may be a CPU command that can be emulated in a testbench by a *transactor* module. The Fire_PyroX_CMD is an identification of the instruction used to identify the command in a *transactor* testbench model. FIRE, DONE, and OK are states of various conditions. ACTIVATED is a state of the relay, and *pyro_sub* is the *Pyro* submodule. The *pFirePyroSafe* property states that if there is a command to fire the pyrotechnics, AND the submodule power is stable, AND the reset cycle is completed, AND the pyrotechnics hardware is armed, then within 5 cycles the pyrotechnics relay must be activated.

6.2.1.2 Latencies

Latencies are critical in a requirements document since they identify a response time. For example:
Requirement: A 256 word *AHB_bus* transfer clocked with the AHB clock shall be relayed onto the *PCI_bus* within 100 to 500 cycles of the PCI clock

---

[4] An assertion as shown may have averted a failure in the *Wide-Field Infrared Explorer* satellite because it was assumed that at start-up time, all digitals and clocks are stable and that the logic that drives the instrument pyrotechnic electronics box is in a safe mode. A review or simulation of the assertion would have challenged that assumption.
http://klabs.org/richcontent/Reports/wiremishap.htm

SVA property definition:

```
property pAhb2PCIxfr;
  @ (posedge ahb_clk) (xaction.send256to_pci) |=>
      @ (posedge pci_clk) ## [100:500] (pci_start_xfr);
endproperty : pAhb2PCIxfr
```

The above property states that after the command *send256to_pci* from an AHB bus interface synchronized to the *ahb_clk*, there must be a start of data transfer (signal *pci_start_xfr*) within 100 to 500 cycles, synchronized to the *pci_clk*.

6.2.1.3 Definition of Processing Algorithms

Processing algorithms are often specified in a requirements document with the algorithm captured as pseudo-code in a modeling language such as SystemC. For example:
Requirement: When the CPU commands a *Filter* operation, the image processing subsystem shall perform the *Filter* algorithm on the loaded image, as described by the function *Filter*, identified in the requirements document. The *Filter* operation on a frame size of 384x288 pixels shall be performed in less than 1000 cycles. Figure 6.2.1.3 is a view of a required image processing.

Figure 6.2.1.3 Sample Image Processing

SVA property definition:

```
property pFilterProcessing;
  xaction.do_Filter_cmd && image_processor.scene_loaded |=>
      ## [*0:1000] image_processor.done_Filter ;
endproperty : pFilterProcessing

property pFilterProcessingCheck;
   always image_processor.done_Filter |->
         image_processor.Filter_scene == Filter(xaction.sent_scene);
endproperty : pFilterProcessingCheck

property pHotPointResults;
  always xaction.get_Filter_hotpoints_cmd |=> ## [0:100]
              Filter_compare (results.hotpoints_array,
                              Filter_hotpoints(xaction.sent_scene));
endproperty : pHotPointResults
```

In these property descriptions, *image_processor* represents an image processor subsystem; *Filter* represents a function; *sent_scene* is an image to be submitted for processing that was sent to the memory; *Filter_scene* is the result of the Filter operation on the image; *Filter_hotpoints* is another function that identifies the compression algorithm of the detected hot points from the scene; *Filter_compare* is a function that compares array objects. Note that these functions can be written in C, as the SystemVerilog Direct Programming Interface (DPI) allows direct inter-language function calls between SystemVerilog and any foreign programming language with a C function call protocol and linking model. Functions implemented in C and made available with the **import** declarations in SystemVerilog can be called from SystemVerilog; such functions are referred to as imported functions.

The property *pFilterProcessing* states that if there is a Filter processing command, and the image scene is loaded into the image processor memory, then within 1000 cycles, the image processor completes the Filter algorithm on that image. The property *pFilterProcessingCheck* compares the actual results of the Filter operation performed on the image processor against the expected operation per the algorithm defined in the requirements document. The property *pHotPointResults* states that upon the initiation of a command to read the hot points, the results of another algorithm are compared against expected results.

6.2.1.4 Reusing System-level Properties in a Verification Environment

An interesting byproduct of the system-level properties is the ease of transitioning those properties into the verification environment. That transition makes use of a transaction-based testbench modeling style, discussed in many books and EDA vendor documentation.[5] Figure 6.2.1.4 demonstrates how the *transactor* can interface to a *server* for the translation of the high-level transaction commands to low-level interface protocols that communicate with the design. In addition, any low-level protocols received by the design under test, such as a block of data, can be captured by the *server*, and abstracted-up to a higher level of abstraction transaction through a data structure (e.g., multi-dimensional arrays).

---

[5] - *Real Chip Design and Verification Using Verilog and VHDL*, Cohen, 2002 ISBN 0-9705394-2-8

- DesignWare® Library Documentation (shown as a representative library). Transaction commands are used to drive the verification environment.
http://www.synopsys.com/products/designware/docs/

Figure 6.2.1.4 ***Transactor*** **and** ***Server*** **for the Translation of High-Level Transaction Commands to Low-Level Interface Protocols**

6.2.1.5 Analyzing Properties Prior to RTL Design

There's more to assertion-based verification than just writing assertions and running them in simulation and/or formal model checking. A technology that is emerging is the verification of properties prior to any RTL code. This methodology is explained for the verification of interface protocols in the referenced paper.[6] In this paper, the author explains how a specification defined with properties can be formally and informally verified to ensure the correctness of the protocol prior to any implementation code. This verification procedure can also check that the properties are not over-restrictive or under-restrictive when compared to what was intended. This is important because the properties are used to verify a design implementation. There are emerging tools that enable the analysis of properties with NO implementation code. This analysis provides, among other things, the generation of waveforms to witness the assertions and to demonstrate what those assertions are verifying. Such a technology helps users come up to speed on the validity of the assertions. It also helps users in the implementation of a verification environment through formal verification and simulation.

[6] Writing, Verifying, and Exploiting Formal Specifications for Hardware Designs http://chicory.stanford.edu/~kannas/publications/KS02.ps

6.2.2 Interface Assertions

Interface assertions represent in a formal manner the properties of the interfaces and black-box protocols. These assertions not only help in the documentation, understanding, and clarification of the interfaces but also provide a verification mechanism of the design at the I/O interface level. Ideally, those definitions need to be defined by a verification engineer from the requirement documents, rather than by the RTL designer. This avoids possible misunderstanding of the requirements made during the implementation to be carried through to the verification environment. However any assumptions made by RTL designer while coding his/her block should be captured as interface assumptions. While integrating the different blocks of RTL, these block level assumptions can act as "guardian" for the individual block's behavior. Another very important aspect to be considered while dealing with interfaces is the interface functional coverage. SystemVerilog properties and sequences can be used with either verification directive, **assert** or **cover**, thereby enabling interface coverage capture. An added benefit of adding interface assertions is that in a system-level simulation, they speed up debug of any failing simulations as they tend to be the closest to the block boundary where the problem originates (assuming that the problem is due to an interface protocol related issue). Section 6.5 provides a complete SystemVerilog FIFO interface example with assertions.[7]

6.2.3 Architectural Plan

Architectures represent not only a top-level basis of the design[8], but also information about the registers, interface cycle timing, design assumptions, and restrictions, along with rationale for those decisions. The architectural plan is the forefront to implementation and verification of the design.

The sequences, properties, and assertions clarify architectural issues and design assumptions. The SystemVerilog properties and assertions are very useful for reviews, and for the engineers who will implement and verify the design. Any assertion code written during the architecture planning stage can be reused during the verification phase of the design. Advanced ABV methodology recommends that any register definition shall also contain necessary functional coverage requirements on the individual fields and the cross of various fields (the individual fields may be spread across various registers).

---

[7] Also available in the downloadable files.

[8] The use of term Architecture in this section is synonymous to "Micro-Architecture" used by design teams.

6.2.4 Verification Plan

The verification plan is a specification for the verification effort. It is used to define what is first-time success, how a design is verified, and which testbenches are written.[9] A verification plan provides a strawman document that can be used by the design community to identify, early in the project, how the design will be tested. Early mistakes in the verification approach can be identified and corrected. A byproduct of the verification plan exercise is the revisit of the requirements. This enforces the process of verifying those requirements, thus helping in the identification of poorly specified or ambiguous requirements. A verification plan is a document that defines the following:

1. **The verification approach**. With ABV and SystemVerilog Assertion, the approach will consist of dynamic verification (e.g., simulation) with a simpler testbench with no (or with a minimum) explicit verifier modules. It may also consist of formal verification of the subblocks or modules with no testbenches. SystemVerilog assertions and coverage points should be included in the plan.
2. **Tests or transactions applied to the design**. These tests are used to verify the design's functional correctness as specified in the requirements specification. This includes tests at the top-level of the design as well as the subblocks. SystemVerilog Assertions can be used to some extent to specify assumptions about inputs, and transactions at the interfaces, along with expected results within a range of cycles, to allow for variations in the DUT cycle timing. Many of these transactions can be extracted from the requirements documents (system and architecture). A new trend in verification is to automatically generate stimulus/tests with SystemVerilog Assertions and assumptions as constraints.
3. **Verification environment for the design-under-test.** This includes the selection of the verification language, the structure of the testbench, and special instructions. The structure encompasses the component models (at the interface level), packages (at the declaration or higher level), and file structures (if files are used). That may include high-level modeling languages such as System-C. Another important selection is a complete tool set such as a linter, a simulator, a formal verifier, and a debugger. The verification environment will also include checkers or verifier modules. Checkers are used to insure that the actual results produced by simulation of the design meet the expected results. SystemVerilog with assertions greatly facilitates the design of checkers. Checkers will typically be a set of modules with signals, data structures (a.k.a. scoreboards), constants, and methods to support the properties and assertions.
4. **Exit criteria**. These criteria identify when verification is complete, or at least achieved the goals. They may include code and functional coverage. Code coverage may include line, branch, condition, toggle, path and FSM.[10] Functional coverage represents a user-defined model of functionalities of the design that the verification process should address. SystemVerilog has a rich set

---

[9] *Writing testbenches, Functional verification of HDL models, 2nd Edition* Janick Bergeron, Kluwer Academic Publishers 2003 ISBN 1402074018

[10] Verification Methodology Manual http://www.transeda.com/vmm/vmm1/index.html

of constructs to capture the coverage model. With SystemVerilog Assertions, coverage would also mean the definition of which assertions should be covered. There is more to assertion coverage than just saying "covered" or "uncovered" (unlike code coverage for instance). Section 6.2.7 addresses the topic of assertion coverage. Assertion coverage is a fairly new technology, and novel ideas on measuring such coverage are emerging. One concept is the static measure of the effectiveness of the assertions on the design. This is referred to as Static Structural coverage. Static structural coverage reports how well each section of the RTL code is covered by assertions. In particular, it points out code that requires more assertion monitoring to ensure sufficient verification. Assertions help to illuminate the internals of the design for thorough verification. Code with insufficient assertion monitoring represents dark corners of the design where bugs may be hiding. One measure of the Static Structural coverage is "Assertion Density". Another measure is the minimum sequential distance (MSD) from each state element of the design to assertions that monitor correct functionality. Parts of a design with high MSD scores indicate low monitoring by assertions, and are candidates for increased assertion monitoring. Design and verification engineers can use static structural coverage to ensure adequate assertion monitoring of designs.[11]

6.2.5 RTL Design

Assertions, generated during the architectural planning phases, greatly facilitate the writing of the RTL implementation because they help the designers to fully understand the architectural and interface requirements. Assertions also cause the designers to think about corner cases upfront in the process. For instance, in a FIFO controller design, a potential corner case is simultaneous Read-and-Write in which a "bypass" mode is required. Without the ABV mindset, it is unlikely that the RTL designer will think about such a case during RTL coding. The properties do contain rules and restrictions that can easily be translated into RTL.

During the coding of the RTL, it is recommended that designers add assertions as executable documentation to be checked during the verification process. Note that assertions are NOT equivalent to RTL code. They represent a higher level of abstraction for the definition of the design. Assertions represent the expected behavior per requirements, as opposed to RTL that deals with detailed FSM architecture and timing. Thus, assertions cross-correlate implementation and provide another level of design verification. Such assertions significantly help in debugging failures because embedded assertions are very close to the source of error. See Chapter 6 for guidelines on writing assertions.

[11] Metric links assertions, simulation and formal verification, providing actionable feedback to speed verification of SoC designs.
http://www.0-in.com/news/PR_030526_Structural_Coverage.html

6.2.6 Write Testbench and Simulate

With the addition of SystemVerilog Assertions throughout the design cycle, the testbench design is greatly simplified because the verification of the interface and the requirement rules are defined in an executable assertion language. That greatly eases the need and design of an automatic verifier. Errors in protocol or design can be quickly detected. The testbench consists of the instantiation of modules in the design, resets, clocks, a *transactor* to define the test scenarios through tasks to be performed (e.g., WRITEs, READs), and a *server* to translate the transactions into low-level signals. The testbench may possibly include a simpler automatic verifier. Figure 6.2.6 represents a generic testbench model with SystemVerilog Assertions.

Figure 6.2.6 Generic Testbench with SystemVerilog Assertions

6.2.7 Analyze the Simulation Results and Coverage

6.2.7.1 Functional Coverage in Verification

With the recent advances in coverage technologies, functional coverage and assertion coverage have become quite significant in the verification process. Functional coverage is recognized as a means to ensure high quality of the design to be released to manufacturing. Testimonies on the values of functional coverage can be found in many books and papers. For example, in the verification of a *Frequency-Programmable Switch Chip*, the IBM design team experienced that "the implementation of coverage models

provided a high level of confidence upon completion of the verification efforts and helped quantify the stress on the logic designs".[12]

Another interesting experience by a customer who used SystemVerilog Assertions in a battery-powered design was the benefit that coverage provided in the performance of the product. Towards the end of the project the top level testbench did not show any functional bugs using the assertions. However, functional coverage revealed that one portion of the logic was performing unnecessary read/write operations to memory. Specifically, that portion of the logic had a high number of non-vacuous completions of memory accesses as compared to the non-vacuous completions of the logic that made use of those accesses. A correction to the logic to eliminate those unnecessary memory accesses yielded a 10% power efficiency increase. Considering that this design was battery-powered, that 10% power efficiency was significant. Assertions with coverage helped in detecting the issues.

Functional coverage can extend to several coverage domains. For example, In the functional verification of the *Z990 Superscalar, Multibook Microprocessor Complex*, the IBM design team identified several coverage domains.[13] These included, among other things the following:

- Program event recording coverage to track coverage events related to a specific domain or logic function, such as branch instructions, store instructions, and exceptions.
- Instruction coverage to verify that all instructions had been executed in all valid addressing, translation, and architectural modes.
- Architected features, such as all possible values for control registers, floating-point control register, and program status word.
- Superscalar grouping that defined which instructions could be executed simultaneously.

Though functional coverage has been well adopted in the industry in the recent past, advanced users are soon realizing that having got all functionalities covered (on top of the usual code coverage) doesn't necessarily mean that all those functionalities are checked for correctness as well. It is little hard fact, but is becoming a reality and hence needs to be addressed at the earliest. Design Managers tend to think that functional coverage can find design errors; rather it is a measure of how much verification is done and not necessarily how "well" it is done. According to this referred IBM article, "the sole purpose of coverage is to measure test completeness and efficiency. Test correctness falls under functional verification". In that project, the failing test cases were all screened, defects were opened on them, and fixes were eventually provided and verified. As a result, all test cases ran successfully.

---

[12] http://www.research.ibm.com/journal/rd/483/hoppe.html
Functional verification of a frequency-programmable switch chip with asynchronous clock sections

[13] http://www.research.ibm.com/journal/rd/483/bair.html
Functional verification of the z990 superscalar, multibook microprocessor complex

While the authors believe strongly that use of functional coverage metrics is a must, they also insist that a vital link between functional coverage and correctness is even more important. One of the advantages of assertion-based coverage measurement is that the correctness is built-in into the assertions. In other words, with assertion-based coverage, functional correctness checking is the main product and functional coverage a very valuable by product. But assertions, as in use today, tend to express localized behavior rather than end-to-end functionality. Hence the authors believe that assertion coverage and functional coverage will co-exist in verification.

One of the increasingly painful problems is the extraction of meaningful information out of these functional coverage statistics, and the determination of "how much verification is done". Traditionally, such an analysis is considered a tool issue. However, with recent advances in reactive testbench techniques, it is becoming more and more important to measure the coverage progress on the fly and to react quickly to the outcome. It is also important to be tool independent and yet achieve this goal. SystemVerilog Assertions provide a set of Application Programming Interface (API) routines that can be used to access the status of assertion evaluations within the verification environment. The API use model is presented in the next section with a small example.

6.2.7.2 SystemVerilog Assertions API

While the SystemVerilog Assertions are very powerful in expressing design behavior, users may need more flexibility in the analysis and interaction of the verification environment based on results of the assertions. Examples of such applications include:

1. Grouping "functionally" related **assert/cover** directives into functional categories and analyzing them in different perspectives[14].
2. Building an advanced temporal debugger as an add-on to the SystemVerilog aware simulator.
3. Reusing internal debugging tools originally based on proprietary assertion languages, thus permitting the smooth migration to SystemVerilog, a standard language. .
4. Building more automated testbenches using custom C-routines to interact with the assertion related events. For example, in an image processing application, a custom C-routine can be called to predict the output frame when an assertion evaluation starts.

Verilog has a standard API named "Programming Language Interface" (PLI). The recent PLI version 2.0 is referred to as "Verilog Procedural Interface (VPI)". SystemVerilog 3.1a LRM defines a Direct Programming Interface (DPI) to interact with SystemVerilog objects and data structure. SystemVerilog LRM also defines a set of API routines to interact with the SystemVerilog Assertions and is called "SystemVerilog Assertions API".

[14] Synopsys supports a proprietary extension to SystemVerilog Assertions via a special attribute called "category" associated with assertions.

While the exact functional prototypes of the SystemVerilog Assertions API itself are beyond the scope of this book, this section presents a brief overview of the functions and use model. Readers are encouraged to refer to the LRM for further details.

Since SystemVerilog Assertions API is an extension above the VPI, we will hereafter refer to it as VPI. The general flow of using VPI to extract information is a two-step process:

1. Obtain a "handle" to the assertions.[15]
2. Obtain the properties of assertions.

Calls to standard VPI routines, such as **vpi_iterate** and **vpi_handle_by_name**, can derive a list of assertions in the entire design or in a specific module. An analysis of assertions statistics will involve:

- Sorting the assertions based on their static characteristics such as name, instance name, module containing the assertions etc. For debug one might also be interested in the file name, line number etc. Such information is "static" in the sense that they remain unchanged throughout the simulation phase.
- Sorting the assertions based on their dynamic characteristics such as attempt started, attempt ended, attempt aborted, attempt stopped etc.

The static information about assertions can be retrieved using a single call to the API routine **vpi_get_assertion_info()** from an assertion handle. To retrieve the dynamic characteristics, the API extends the VPI call back routines to accommodate assertions and register call-backs on specific events, such as start, stop etc.

6.2.7.2.1 Building an assertion coverage extractor using SystemVerilog Assertions API

The Assertion API capabilities are demonstrated with a simple example. The shown C-code is incomplete and represents pseudo-code to show intent.

For assertions, the coverage status property **vpiCovered** implies that the assertion has been attempted, has succeeded at least once, and has never failed. More detailed coverage information can be obtained for assertions by using different arguments to the **vpi_get** function. Table 6.2.7.2.1-1 lists various **vpi_get** arguments related to assertions.

[15] NOTES
1—As with all VPI handles, assertion handles are handles to a specific instance of a specific assertion.
2—Unnamed assertions cannot be found by name.

Table 6.2.7.2.1-1 vpi_get() Arguments and Usage

Argument to vpi_get() function	Usage
vpiAssertAttemptCovered	To extract the number of assertion attempts.
vpiAssertSuccessCovered	To extract the number of TRUE (non-vacuous) successes. When used with a sequence, it returns the number of matches.
vpiAssertVacuousSuccessCovered	To extract the number of vacuous successes.
vpiAssertFailureCovered	To extract the number of assertion failures.

Figure 6.2.7.2.1-1 represents SystemVerilog Assertions API C-code for an assertion statistics extractor.

```
#include <stdio.h>
#include <vpi_user.h>
#include <vcs_vpi_user.h>
#include <sv_vpi_user.h>
#include <vcsuser.h>
void assert_cov (int data, int reason)
{
 vpiHandle assertion_h, itr_h;
 char * a_name;
 int a_attempts, a_true_success, a_vac_success, a_failures;
 itr_h = vpi_iterate(vpiAssertion, NULL);
 vpi_printf ("SystemVerilog Assertions Statistics - demo from SVA Handbook\n");
 vpi_printf ("Assertion Name: \t \t No. of True Success \t No. of Vacuous Success \t
              No. of Failures \t No. of total attempts \n");
   while (assertion_h = vpi_scan(itr_h)) {
        a_name = vpi_get_str(vpiFullName, assertion_h);
        a_attempts = vpi_get(vpiAssertAttemptCovered, assertion_h);
        a_true_success = vpi_get(vpiAssertSuccessCovered, assertion_h);
        a_vac_success = vpi_get(vpiAssertVacuousSuccessCovered, assertion_h);
        a_failures = vpi_get(vpiAssertFailureCovered, assertion_h);
        vpi_printf ("%0s  %0d \t %0d \t %0d \t %0d \t %0d \n",
                a_name, a_true_success, a_vac_success, a_failures, a_attempts);
  } /* while */
} /* assert_cov */
```

Figure 6.2.7.2.1 SystemVerilog Assertions API C-code for Assertion Statistics Extractor.

A sample SystemVerilog model making use of the above C function is shown in Figure 6.2.7.2.1-2. This model does not show any RTL or assertions. However, types of results produced using this methodology is shown in Figure 6.2.7.2.1-3.

```
module cov_api_test ();
 timeunit 1ns;
 timeprecision 100ps;
// Design has 3 assertions named as follows:
// ap_rst_checks, ap_arm_arith_instructions, ap_arm_logical_instructions
// The definitions of those assertions are not shown here as the goal of
// this example is to demonstrate SVA API call.
event sim_ended;
// this event is emitted when simulation ends so as to enable coverage collection
// other option could be to use vpiCallBack
initial
 begin : call_api
   @ sim_ended;
   $assert_cov; // call to API C-code function
end
endmodule : cov_api_test
```

Figure 6.2.7.2.1-2 Sample SystemVerilog Example for Use of the Coverage Extractor

Table 6.2.7.2.1-2 shows arbitrary results that can be produced after simulation of the model.

Table 6.2.7.2.1-2 Sample Assertion Statistics

Assertion Name	No. of True Success	No. of Vacuous Success	No. of Failures	No. of total attempts
cov_api_test.ap_rst_checks	5	0	3	8
cov_api_test.ap_arm_arith_instructions	9	5	2	16
cov_api_test.ap_arm_logical_instructions	0	4	7	11

6.2.8 Formal Verification (FV)

If formal verification is available, the design can be exercised through the tool using the SystemVerilog assertions and assumptions already defined. However, prior to formal verification, users may opt to first exercise the model and assertions with simulation to iron out gross design errors. Chapter 7 addresses the topic of formal verification.

6.3 Case Study - Synchronous FIFO

This section provides a definition of the requirements for a synchronous FIFO used as IP. This section demonstrates by example how properties unambiguously clarify requirements.[16] The RTL design with properties is also presented. A rather simple testbench model will be used to exercise the design since the goal of this book is on using properties and assertions, rather than testbench design techniques. An explanation of how the various pieces fit together is also demonstrated.

The numbering scheme for this specification and verification plan is separate from this chapter-numbering scheme, and thus, each document starts at the number ONE.

6.3.1 Synchronous FIFO Requirements

The requirements for a synchronous FIFO are provided in the following pages.

[16] Format for this specification is from the book *Component Design by Example*, 2001 ISBN 0-9705394-0-1, Ben Cohen

REQUIREMENTS FOR A SYNCHRONOUS FIFO,
First-In First-Out Buffer

Document #: **fifo_req_001**
Release Date: __
Revision Number: ____
Revision Date: __
Originator
Name: ____
Phone: ___
email: __

Approved:
Name:
Phone:
email:

Revisions History :
Date:
Version:
Author:
Description:
Synchronous FIFO to be used as an IP. FIFO management (e.g., push, pop, error handling) is external to the FIFO.

1. SCOPE

1.1 Scope

This document establishes the requirements for an Intellectual Property (IP) that provides a synchronous First-In First-Out (FIFO) function.

The specification is primarily targeted for component developers, IP integrators, and system OEMs.

1.2 Purpose

These requirements shall apply to a synchronous FIFO with a simple interface for inclusion as a component. This requirement includes SystemVerilog assertions to further clarify the properties of the FIFO.

1.3 Classification

This document defines the requirements for a hardware design.

2. DEFINITIONS

2.1 PUSH

The action of inserting data into the FIFO buffer.

2.2 POP

The action of extracting data from the FIFO buffer

2.3 FULL

The FIFO buffer being at it maximum level.

2.4 EMPTY

The FIFO buffer with no valid data.

2.5 Read and Write Pointers

Pointers represent internal structure of the FIFO to identify where in the buffer data will be stored (write pointer, *wr_ptr*), or be read (read pointer, *rd_ptr*)

3. APPLICABLE DOCUMENTS

3.1 Government Documents

None

3.2 Non-government Documents

None

4. ARCHITECTURAL OVERVIEW

4.1 Introduction

The FIFO component shall represent a design written in SystemVerilog with SystemVerilog assertions. The FIFO shall be synchronous with a single clock that governs both reads and writes. The FIFO typically interfaces to a controller for the synchronous pushing and popping of data. Figure 4.1 represents a high level view of the interfaces.

FIFO Requirements Example (continued)

Figure 4.1 High Level View of the FIFO Interfaces

The FIFO shall include the following features:
1. Parameterized storage space for data buffers
2. Parameterized data widths for the data.
3. Flag information for FULL, EMPTY, ALMOST FULL at the ¾ level, ALMOST EMPTY at the ¼ level.
4. A synchronous RESET capability.

4.2 System Application

The FIFO can be applied in a variety of system configurations. Figure 4.2-1 demonstrates one such configuration where the FIFO interfaces on one side to a bus controller, and on the other side to a different controller. All buses use the same system clock. It is the responsibility of the enqueue/dequeue controller to manage the integrity of quantity of data transferred into and extracted out of the FIFO.

Figure 4.2-2 Hardwired Application of a FIFO

5. PHYSICAL LAYER

The physical hardware interfaces shall be as shown in Figure 5.0

Figure 5.0 Interfaces of the FIFO

A SystemVerilog description of the interface is shown in Figure 5.1.

```
// PACKAGE for type and parameter definitions
// ch4/fifo_queue/fifo_pkg.sv
package fifo_pkg;
  timeunit 1ns;
  timeprecision 100ps;
  localparam BIT_DEPTH = 4; // 2**BIT_DEPTH = depth of fifo
  localparam FULL = int'(2** BIT_DEPTH -1);
  localparam ALMOST_FULL = int'(3*FULL / 4);
  localparam ALMOST_EMPTY = int'(FULL/4);
  localparam WIDTH = 32;
  typedef logic [WIDTH-1 : 0] word_t;
  typedef word_t [0 : 2**BIT_DEPTH-1] buffer_t;
  // Other types for testbench support can be added here
endpackage : fifo_pkg
//[17]
```

FIFO Requirements Example (continued)

[17] The complete SystemVerilog interface with assertions is in file ch4/fifo_if.sv

```
// INTERFACE of FIFO
interface fifo_if(input clk, reset_n);
 import fifo_pkg:: *; // access to package
 logic push; // push data into the fifo
 logic pop; // pop data from the fifo
 logic almost_full; // fifo is at 3/4 maximum level
 logic almost_empty; // fifo is at 1/4 maximum level
 logic full; // fifo is at maximum level
 logic empty; // fifo is at the zero level (no data)
 logic error; // fifo push or pop error
 word_t data_in;
 word_t data_out;

// FIFO DUV, FIFO Slave interface
 modport fslave_if (
            output empty,
            output almost_empty,
            output almost_full,
            output full,
            output data_out,
            input data_in,
            input push,
            input pop);

 // FIFO driver, FIFO Driver interface
 modport fdrvr_if (
            output data_in,
            output push,
            output pop,
            input empty,
            input almost_empty,
            input almost_full,
            input full,
            input data_out);
// tasks / sequences / properties / assertions shall be added here (see Section 5.1)
endinterface : fifo_if
```

Interface description with modports clarifies the use of the ports. Maintain ordering convention: outputs first, inputs last.

Used by application interfaced to the FIFO

Figure 5.1 SystemVerilog FIFO Interface

5.1 Interface Port Description

The individual port elements in the interface in figure 5.1 are described in this section with requirements on them captured as assertions. Since some of the ports describe data intensive portion of the system (such as the data being popped from the FIFO), some of the SystemVerilog testbench features such as queues and tasks are used to capture their requirements. Since these tasks and queues are meant solely for the purpose of specification and verification, and do not have a direct correlation to the hardware implementation of the FIFO, they are

FIFO Requirements Example (continued)

declared in the interface itself:

```
// Data queue for verification.
// Queue. maximum size is 2**BIT_DEPTH
  word_t dataQ [$:2**BIT_DEPTH-1];
// Data read from queue
  word_t data_fromQ;

// Push and Pop tasks
 task pop_task;
   begin
    data_in <= 'X; // unsized Xs
    pop  <= 1'b1;
    data_fromQ <= dataQ.pop_front();
    @ (posedge clk);
   end
 endtask : pop_task

 task push_task (word_t data);
    begin
      $display ("%0t %m Push data %0h ", $time, data);
      data_in <= data; //data to be written
       push <= 1'b1;
       dataQ.push_back(data); // push to dataQ
       @ (posedge clk);
    end
 endtask : push_task

 task idle_task(int num_idle_cyles);
  begin
   push <= 1'b0;
   pop <= 1'b0;
   data_in <= 'X;
   assert (num_idle_cycles < 10000) else
    $warning ("%0t %0m idle_task is invoked with LARGE number of idle
cycles %0d ", num_idle_cycles);
   repeat (num_idle_cycles) @ (posedge clk);
  end
 endtask : idle_task
```

Data Queue

Input data stored into FIFO buffer 1 cycle following the push

Use immediate assertion for simple, local checks.

FIFO Requirements Example (continued)

5.1.1 Data Input/Output

Figure 5.1.1 provides a timing diagram of the interface.

Figure 5.1.1 FIFO Interface Timing Diagram

5.1.1.1 Data_In

Direction: Input, Peripheral -> FIFO;
Size: Determined by WIDTH parameter; Active level: High
Data sent from a peripheral device to the FIFO under the control of the *push* control.

5.1.1.2 Data_Out

Direction: Output, FIFO -> Peripheral;
Size: Determined by WIDTH parameter; Active level: *High*
FIFO data sent to a peripheral device under the control of *pop* signal.

5.1.2 Push / Pop

5.1.2.1 push

Direction: Input, Peripheral -> FIFO; Size: 1 bit, Active level: high
When *push* is active, *data_in* shall be stored into the FIFO buffer at the next clock cycle. It is an error if a push with no pop control occurs on a full FIFO. The following property characterizes these requirements:

```
// never a push and full and no pop
 sequence q_push_error;
  !(push && full && !pop);
 endsequence : q_push_error
 ap_push_error : assert property (@(posedge clk) q_push_error);
```

5.1.2.2 pop

Direction: Input, Peripheral -> FIFO; Size: 1 bit, Active level: high
When *pop* is active, *data_out* shall carry the data that was first stored into the FIFO, but was not yet popped. The *data_out* shall be asserted in the same cycle of *pop* control. It is an error if a pop control occurs on an empty FIFO. The following properties and task characterize these requirements:

```
// Data out timing and data integrity
```

FIFO Requirements Example (continued)

```
property p_pop_data;
   @ (posedge clk)
        pop |-> data_out == data_fromQ;
        // from 5.1 pop_task
endproperty : p_pop_data
ap_pop_data : assert property (p_pop_data);

// never a pop on empty
 sequence q_pop_error;
     ! (pop && empty && !push);
endsequence : q_pop_error
ap_pop_error : assert property (@ (posedge clk) q_pop_error);
```

5.1.2.3 Push-Pop Data Sequencing

Data entered into the FIFO buffer shall be outputted in the same order that it is entered. The *push_task* and *pop_task* tasks, and the properties characterized in sections 5.1.2.1 and 5.1.2.2 define the ordering sequence. Specifically, data pushed in the back of the FIFO buffer is extracted from the front of the buffer in a first-in, first-out manner.

5.1.3 Status Flags

5.1.3.1 Full

Direction: Output, FIFO -> Peripheral ; Size: 1 bit, Active level: high

When the FIFO reaches the maximum depth of the buffer, as defined by the parameter BIT_DEPTH, then the *full* flag shall be active. The following sequence and property characterize this requirement:

```
sequence qFull;
 @ (posedge clk)
  dataQsize == BIT_DEPTH;
endsequence : qFull

property p_fifo_full;
   @ (posedge clk)  qFull |-> full;
endproperty : p_fifo_full
ap_fifo_full : assert property (p_fifo_full);
```

```
int dataQsize; // queue size
assign dataQsize=dataQ.size;
// See section 8.2.7 for
// guidelines on using dynamic
// data types inside properties.
```

5.1.3.2 Almost Full

Direction: Output, FIFO -> Peripheral ; Size: 1 bit, Active level: high

When the number of entries in the FIFO reaches or is greater than the predefined value of ¾ of the maximum depth of the buffer, as defined by the parameter ALMOST_FULL, then the *almost_full* flag shall be active. The following sequence and property characterizes this requirement:

```
sequence qAlmost_full;
 @ (posedge clk)
  dataQsize >= ALMOST_FULL;
endsequence : qAlmost_full
```

FIFO Requirements Example (continued)

```
property p_fifo_almost_full;
   @ (posedge clk)  qAlmost_full |-> almost_full;
endproperty : p_fifo_almost_full
ap_fifo_almost_full : assert property (p_fifo_almost_full);
```

5.1.3.3 Empty

Direction: Output, FIFO -> Peripheral ; Size: 1 bit, Active level: high

When all the enqueued data has been dequeued, then the *empty* flag shall be active. A reset shall cause the empty flag to be active. The following sequence and properties characterize these requirements:

```
// sequence definition, use in cover for empty
sequence qEmpty;
 @ (posedge clk)
   dataQsize==0;
endsequence : qEmpty

property p_fifo_empty;
 @ (posedge clk)  qEmpty |-> empty;
endproperty : p_fifo_empty
ap_fifo_empty : assert property (p_fifo_empty);
```

The property for the flags at reset time is defined in section 5.1.4.

5.1.3.4 Almost Empty

Direction: Output, FIFO -> Peripheral ; Size: 1 bit, Active level: high

When the number of entries in the FIFO reaches or is less the predefined value of ¼ of the maximum depth of the buffer, as defined by the parameter ALMOST_EMPTY, then the *almost_empty* flag shall be active. The following sequence and property characterize this requirement:

```
sequence qAlmost_empty;
 @ (posedge clk)  dataQsize <= ALMOST_EMPTY;
endsequence : qAlmost_empty

property p_fifo_almost_empty;
 @ (posedge clk) qAlmost_empty |-> almost_empty;
endproperty : p_fifo_almost_empty
ap_fifo_almost_empty : assert property (p_fifo_almost_empty);
```

5.1.4 Reset

Direction: Input, Peripheral -> FIFO ; Size: 1 bit, Active level: low

The *reset_n* is an active low reset control that clears the pointers and the status flags. The *reset_n* is asynchronous to the system clock *clk*. See properties defined in section 5.1.3.3 for the behavior of the empty flag when *reset_n* is asserted in the FIFO.

```
property p_fifo_ptrs_flags_at_reset;
 @ (posedge clk)
   !reset_n |-> ##[0:1] !almost_empty && ! full && !almost_full && empty;
endproperty : p_fifo_ptrs_flags_at_reset
ap_fifo_ptrs_flags_at_reset : assert property (p_fifo_ptrs_flags_at_reset);
```

FIFO Requirements Example (continued)

5.15 Clock

Direction: Input, Peripheral -> FIFO ; Size: 1 bit, Active edge: rising edge

The *clk* clock is the synchronous system clock for the FIFO for both the read and write transactions, active on the positive edge of the clock. The clock shall be at 50% duty cycle.

5.16. Error

Direction: Output, FIFO -> Peripheral; Size: 1 bit, Active level: high

When either an overflow (push on full) or underflow (pop on empty) error has occurred, the error flag shall be asserted. The following properties characterize the *error* output.

```
// Reusing the q_push_error and q_pop_error definitions,
property p_error_flag;
  @ (posedge clk)
        q_push_error or q_pop_error |=> error;
endproperty : p_error_flag
ap_error_flag : assert property (p_error_flag);
```

6. PROTOCOL LAYER

The FIFO operates on single word writes (*push*) or single word reads (*pop*).

7. ROBUSTNESS

7.1 Error Detection

The FIFO shall lump all overflow (push on full) or underflow (pop on empty) errors as a single error output. See section 5.16 for details.

8. HARDWARE AND SOFTWARE

8.1 Fixed Parameterization

The FIFO shall provide the following parameters used for the definition of the implemented hardware during hardware build:

BIT_DEPTH where 2**BIT_DEPTH represents the depth of FIFO.
WIDTH represents the data width.
ALMOST_FULL (0.75 * (2 ** BIT_DEPTH))
ALMOST_EMPTY (0.25 * (2 ** BIT_DEPTH))

8.2 Software Interfaces

The FIFO shall enter input data (*data_in*) into the FIFO buffer when the *push* control is active. It shall provide data from the buffer upon an activation of the *pop* control. See section 5.1.2 Push / Pop for definition of the properties that characterize these controls. The FIFO contains no internal registers that can be configured.

This section typically contains the internal registers that the software can access and configure.

9. PERFORMANCE

9.1 Frequency

The FIFO shall support a maximum rate of 25 MHz.

FIFO Requirements Example (continued)

9.2 Power Dissipation
The power shall be less than 0.01 watt at 25 MHz.

9.3 Environmental
Does not apply.

9.4 Technology
The design shall be adaptable to any technology because the design shall be portable and defined in SystemVerilog RTL.

10. TESTABILITY
None required.

11. MECHANICAL
Does not apply.

12. Backup Information
A copy of the FIFO interface model and supporting package is included in the download files.

FIFO Requirements Example (continued)

6.3.2 Verification Plan

The following demonstrates the application of assertions in a verification plan to clarify the verification goals and milestones.

Header page

VERIFICATION PLAN FOR SYNCHRONOUS FIFO, First-In First-Out Buffer

Pertinent logistics data about the requirements.

Document #: fifo_ver_plan_001
Release Date: __/__/__
Revision Number: ____
Revision Date: __/__/__
Originator
Name:
Phone:
email:

Conform to company policies and style

Approved:
Name:
Phone:
email:

Revisions History:
Date:
Version:
Author:
Description:
Describes verification approaches for the FIFO buffer.

...

FIFO Verification Plan Example

1. SCOPE

1.1 Scope

This document establishes the verification plan for the FIFO design specified in the requirements specification. It identifies the features to be tested, the test cases, the expected responses, and the methods of test case application and verification. SystemVerilog Assertions properties specify characterizations that the design must meet, and test sequences that must be covered.

The verification plan is primarily targeted for component developers, IP integrators, and system OEMs.

1.2 Purpose

The verification plan provides a definition of the testbench, verification properties, test environment, coverage sequences, application of test cases, and verification approaches for the FIFO design as specified in the requirement specification number *fifo_req_001*, and in the implementation document number *fifo_dsgn_001*.[18]

The goals of this plan is not only to provide an outline on how the component will be tested, but also to provide a strawman document that can be scrutinized by other design and system engineers to refine the verification approach.

1.3 Classification

This document defines the test methods for a hardware design.

2 DEFINITIONS

2.1 BFM

A Bus Functional Model is a model that emulates the operation of an interface (i.e., the bus), but not necessarily the internal operation of the interface.

2.2 Transaction

Tasks that need to be executed to verify the device under test. An example of a transaction would be a push with specified DATA along with a simultaneous pop.

3. APPLICABLE DOCUMENTS

3.1 Government Documents

None.

3.2 Non-government Documents

Document #: *fifo_req_001*, Requirement Specification for a Synchronous FIFO.

FIFO Verification Plan Example (continued)

[18] The implementation document is not supplied because it is not within the scope of this book, which focuses on SystemVerilog Assertions rather than RTL design.

3.3 Executable specifications

Interface verification properties written in SystemVerilog, file *ch4/fifo_if.sv*.

3.4 Reference Sources

SystemVerilog 3.1a LRM[19].

4. COMPLIANCE PLAN

SystemVerilog with assertions along with simulation will be used as the verification language because it is an open language that provides good constructs and verification features. This plan consists of the following:

- Feature extraction and test strategy
- Test application approach for the FIFO
- Test verification approach

4.1 Feature Extractions and Test Strategy

The design features are extracted from the requirements specification. For each feature of the design, a test strategy is recognized. The strategy consists of directed and pseudo-random tests. A verification criterion for each of the design feature is documented. This feature definition, test strategy, test sequence, and verification criteria forms the basis of the functional verification plan. Table 4.1 summarizes the feature extraction and verification criteria for the functional requirements.

For corner testing, pseudo-random **push** and **pop** transactions will be simulated to mimic a FIFO in a system environment. The environment will perform the following transactions at pseudo-random intervals:

1. Create push requests
2. Create pop requests
3. Force resets

The properties specified in section 5 of the specification document will be used. Properties are also used to clarify the test sequences.

FIFO Verification Plan Example (continued)

[19] http://www.eda.org/sv/SystemVerilog_3.1a.pdf

Table 4.1 Feature Extraction and Verification Criteria

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE / STRATEGY	VERIFICATION CRITERIA Interface verification SVA properties (*fifo_props.sv*) + the following
1	**Fixed Parameterization** - Bit_depth (FIFO buffer size) - data width	8.1, 5.1	1	**Configuration Setup** Buffer depth == 2**4, 2**8 Width==16, 32 Pseudo-random push and pop transactions. Unique data patterns using random values.	Design compiles and elaborates correctly. Simulation + monitoring of properties and coverage. Verification of output sequence (i.e., first-in is first-out)

2	**RESET**
	. Reset applied when fill state of FIFO is at different levels.
		5.1.4	1	**property** p_t1_full; @ (**posedge** clk)
				full |=> !reset_n;
				endproperty : p_t1_full
				
				property p_t2_afull; @ (**posedge** clk)
				almost_full |=> !reset_n;
				endproperty : p_t2_afull
				
				property p_t3_empty; @ (**posedge** clk)
				empty |=> !reset_n;
				endproperty : p_t3_empty
				
				property p_t4_a empty; @ (**posedge** clk)
				almost_empty |=> !reset_n;
				endproperty : p_t4_a empty
					Simulation + monitoring of properties and coverage.

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE	VERIFICATION CRITERIA Interface verification SVA properties (*fifo_props.sv*) + the following
3	**COVERAGE** Using the properties and sequences defined in 4.1		1	**cover property** (p_push_pop_sequencing); **cover property** (qFull); **cover property** (qEmpty); **cover property** (qAlmost_empty); **cover property** (qAlmost_full); **cover property** (qOffFull); **cover property** (qOffEmpty); **cover property** (qOffAlmost_empty); **cover property** (qOffAlmost_full);	Sequences must all have a coverage count of at least one.

Page intentionally left blank.

4.2 Testbench Architecture

Several architectural elements must be considered in the definition of the testbench environment, including the following:

- Reusability / ease of use / portability / verification language
- Number of BFMs to emulate the separate busses
- Synchronization methods between BFMs
- Transactions definition and sequencing methods
- Transactions driving methods
- Verification strategies for design and its subblocks

Figure 4.2.1-1 represents the testbench architecture. The testbench makes use of the FIFO interface definition, FIFO package, and the FIFO property module. The testbench includes a *transactor* block to generate transactions such as **reset, push, pop**, and **idle** cycles. A set of server tasks provides the low level protocols to execute the transactions.

Figure 4.2.1-1 FIFO Testbench Architecture

SystemVerilog will be used for this design because it is a standard language, and is portable across tools. A reusable design style will be applied. A SystemVerilog package captures the common parameters for this design and is shown in Figure 4.2.1-2. A property module file is defined in Figure 4.2.1-2, for binding to the FIFO from within the testbench. An outline of the FIFO testbench that demonstrates the module instantiations and binding is shown in Figure 4.2.1-3

```
// PACKAGE for type and parameter definitions
package fifo_pkg;
  timeunit 1ns;
  timeprecision 100ps;
  localparameter BIT_DEPTH = 4; // 2**BIT_DEPTH = depth of fifo
  localparameter FULL = int'(2** BIT_DEPTH -1);
  localparameter ALMOST_FULL = int'(3*FULL / 4);
  localparameter ALMOST_EMPTY = int'(FULL/4);
  localparameter WIDTH = 32;
  typedef logic [WIDTH-1 : 0] word_t;
  typedef word_t [0 : 2**BIT_DEPTH-1] buffer_t;

  // types supporting the testbench
   endpackage : fifo_pkg
```

Figure 4.2.1-2. Supporting Package (/ch4/fifo_queue/fifo_pkg.sv)

```
// ------------------------------------------------
// PROPERTY MODULE for FIFO
// This module is used for verification of the FIFO, and is
// intended to be bound (with the SystemVerilog "bind") to the DUV
module fifo_props (input clk, input reset_n, fifo_if fifo_if);
  import fifo_pkg::*;

  // Coverage points based on value of fifo fullness
 // As specified in the Verification Plan, Table 4.1
  property p_t1_full;  @ (posedge clk)
    fifo_if.full |=> reset_n==0;
  endproperty : p_t1_full
  cp_t1_full_1: cover property (p_t1_full);

  property p_t2_afull;  @ (posedge clk)
    fifo_if.almost_full |=> reset_n==0;
  endproperty : p_t2_afull
  cp_t2_afull_1: cover property (p_t2_afull);

  property p_t3_empty;  @ (posedge clk)
    fifo_if.empty |=> reset_n==0;
  endproperty : p_t3_empty
  cp_t3_empty_1: cover property (p_t3_empty);

  property p_t4_aempty;  @ (posedge clk)
   fifo_if.almost_empty |=>  reset_n==0;
  endproperty : p_t4_aempty
  cp_t4_aempty_1 : cover property (p_t4_aempty);

  property p_push_pop_sequencing; @ (posedge clk)
    fifo_if.push => ##[0:$] fifo_if.pop;
  endproperty : p_push_pop_sequencing
```

FIFO Verification Plan Example (continued)

```
// coverage of sequences
// As specified in the Verification Plan, Table 4.1

 cp_push_pop_sequencing : cover property (p_push_pop_sequencing);
 c_qFull :            cover property ( @ (posedge clk) fifo_if.qFull);
 c_qEmpty :           cover property (@ (posedge clk) fifo_if.qEmpty);
 c_qAlmost_empty :    cover property (@ (posedge clk) fifo_if.qAlmost_empty);
 c_qAlmost_full :     cover property (@ (posedge clk) fifo_if.qAlmost_full);
endmodule : fifo_props
```

Figure 4.2.1-3 Property File for Inclusion with the Bind Construct (/ch4/fifo_queue/fifo_props.sv)

```
// FIFO testbench Outline
      module fifo_tb;
       timeunit 1ns;
       timeprecision 100ps;
        logic clk = 1'b0;  // system clock
        logic reset_n = 1'b0;

        import fifo_pkg::*; // Access to package information

        fifo_if b_if(.*);       // instantiation of fifo interface
        fifo_rtl fifo_rtl_1(.*);  // instantiation of fifo DUV

        // bind the fifo_rtl model to an implicit instantiation (fifo_props_1)
        // of property module fifo_props
        bind fifo fifo_props fifo_props_1(clk, reset_n, b_if);
        task reset_task(int num_rst_cycles);
        begin
         $display("%0t Resetting DUT for %0d cycles ", $time, num_rst_cycles);
          reset_n = 1'b0;
          repeat (num_rst_cycles) begin
             {b_if.push, b_if.pop} = $random % 2;
              b_if.data_in = $random;
             @ (posedge clk);
          end // repeat
          reset_n = 1'b1;
          b_if.push = 1'b0;
          b_if.pop = 1'b0;
       end
       endtask : reset_task

       // testbench code
       initial forever #50 clk = ~clk;
```

```
initial
    begin : client
    // directed tests
     reset_task(5);
    // 3 pushes
    for (int i=0; i<= 3; i++) begin
     // push_task($random % WIDTH);
     b_if.push_task($random );
     b_if.idle_task($random % 5);
     b_if.push_task(11 );
    end

    // 3 pop
    for (int i=0; i<= 3; i++) begin
     b_if.pop_task;
     b_if.idle_task($random % 5);
    end

   // push/pop random
    for (int i=0; i<= 5; i++) begin
     if ($random %2) begin
      b_if.push_task($random % WIDTH);
      b_if.idle_task($random % 3);
     end
     else begin
      b_if.pop_task;
      b_if.idle_task($random % 4);
     end
    end
    $stop;
    end // block: client
endmodule : fifo_tb
```

FIFO Verification Plan Example (continued)

Figure 4.2.1-4 FIFO Testbench Outline (fifo_tb.sv)

5.0 Design Tools

This is beyond the scope of this book. This section typically defines the names of the linting, simulation, debugging, formal verification, and any other tool used in the verification process.

FIFO Verification Plan Example (continued)

6.3.3 RTL Design

The RTL design using the FIFO interface is shown in Figure 6.7-1.

```
module fifo (input clk, input reset_n, fifo_if.fslave_if b_if);
  timeunit 1ns;
  timeprecision 100ps;
 // import fifo_pkg::*;
 logic [BIT_DEPTH-1 : 0] wr_addr; // write fifo address
 logic [BIT_DEPTH-1 : 0] rd_addr; // read fifo address
 buffer_t buffer; // fifo storage
 parameter shiftVal = int'(2**BIT_DEPTH) ;

 // Push on full with no pop error
 property p_push_error;
  @ (posedge clk)
    not (b_if.push && b_if.full && !b_if.pop);
 endproperty : p_push_error
 ap_push_error_1 : assert property (push_error);
 // mp_push_error : assume property (p_push_error);

 // No pop on empty
 property p_pop_error;
  @ (posedge clk)
    not (b_if.pop && b_if.empty);
 endproperty : p_pop_error
 ap_pop_error_1 : assert property (p_pop_error);
// mp_pop_error : assume property (p_pop_error);

// RTL Detailed design
logic [1:0]        push_pop;
// Defining control for maintenance of FIFO word count
assign        push_pop = {b_if.push , b_if.pop};   // temporary concatenation
// providing READ data to output
assign b_if.data_out = buffer[rd_addr];

// purpose: writes data from FIFO and controls
//        words within the FIFO.  It also provides the READ and
//        WRITE Fifo Addresses.
 always @ (posedge clk or negedge reset_n)
  begin
  if (!reset_n)
     begin  // asynchronous reset (active low)
      rd_addr       <= 0;
      wr_addr       <= 0;
     end
    else
     begin
      case (push_pop)
     2'b00 :
     ;          // no push, no pop
```

```
        2'b01 :
          begin              // no push, pop for READ
           rd_addr       <= (rd_addr+1) % shiftVal;
          end
         2'b10 :             // push for WRITE, no pop
          begin
           wr_addr       <= (wr_addr + 1) % shiftVal;
           buffer[wr_addr] <= b_if.data_in;
          end
         2'b11 :             // push for WRITE, pop for READ
          begin
           rd_addr <= (rd_addr+1) % shiftVal;
           wr_addr <= (wr_addr + 1) % shiftVal;
           buffer[wr_addr] <= b_if.data_in;
          end
        default :
        begin
          a_illegal_fifo_cmd_1 : assert (1'b0) else
            $warning ("%0t %0m Meta value detected in FIFO command, push_pop %2b ",
                  $time, push_pop);
        end

        endcase
      end //else
    end //always

  // Reporting of flags
  // properties used as guide for this design since they define
  // the requirements
  always @ (posedge clk)
  begin
    b_if.error <= (b_if.pop && b_if.empty) || (b_if.push && b_if.full && !b_if.pop);
  end // always

  assign b_if.full = (wr_addr - rd_addr) == FULL;
  assign b_if.empty = (wr_addr == rd_addr);
  assign b_if.almost_full = (wr_addr - rd_addr) >= ALMOST_FULL;
  assign b_if.almost_empty = (wr_addr - rd_addr) <= ALMOST_EMPTY;

endmodule : fifo
```

Figure 6.7-1 FIFO with Interfaces ***/ch4/fifo_rtl.sv.***

6.3.4 Simulation

The model includes the following designs and files. Please refer to the files for details. Model was successfully compiled and executed in *vcs* simulator.

Deign Name	File Name ch6	Comments
fifo_pkg	fifo_pkg_include.sv	// PACKAGE for type and parameter definitions
fifo_props	fifo_props.sv	Property file used to demonstrate binding
fifo_rtl	fifo_rtl.sv	RTL design of FIFO with assertions
fifo_assert_control	fifo_assert_control.sv	Disable assertions during_reset
fifo_tb	fifo_tb.sv	Testbench

6.3.5

Formal Verification

The RTL FIFO model was formally verified using *@Verifier* formal verification tool.[20] The analysis reported that aside from two errors related to the potential application of the device, the model met all functional requirements. The failed assertions are related to a *push* on a full FIFO (*push_error*), and a *pop* on an empty FIFO (*pop_error*). These errors are demonstrated in the output of the formal verification tool in Figure 6.3.5-1 and 6.3.5-2.

@Verifier Window

Module: All Scope: Type: All Status: All Display Instance File: Reload MC Engine

Module: fifo_rtl num instances (1) total rows = 7

Name	Type	Status	Line #	Engine	#Flops + #Gates	Debug	Explorer		Message #	Message
fifo_almost_empty	SVA	passed	122	Full Proof	13 + 141	-	✓	✓	191	passed - at top level module fifo_rtl
fifo_almost_full	SVA	passed	101	Full Proof	13 + 166	-	✓	✓	191	passed - at top level module fifo_rtl
fifo_empty	SVA	passed	108	Full Proof	13 + 74	-	✓	✓	191	passed - at top level module fifo_rtl
fifo_full	SVA	passed	95	Full Proof	13 + 151	-	✓	✓	191	passed - at top level module fifo_rtl
fifo_ptrs_flags_at_reset	SVA	passed	116	Full Proof	13 + 251	-	✓	✓	191	passed - at top level module fifo_rtl
pop_error	SVA	failed	68	Full Proof	9 + 33		✓	✓	191	failed - at top level module fifo_rtl
push_error	SVA	failed	55	Full Proof	9 + 105		✓	✓	191	failed - at top level module fifo_rtl

Figure 6.3.5-1 Formal Verification Results for FIFO RTL (@HDL @Verifier)

[20] http://www.athdl.com/
@Verifier is a family of assertion-based verification product, supporting both PSL and SystemVerilog that also supports multiple clock domain verification and analysis.

Figure 6.3.5-2. counterexample demonstrating case for the push error (@HDL @verifier)

The analysis of the two errors demonstrate that the RTL logic provides no interlock logic to prevent the situations causing the errors. However, the design meets the requirements "6.2 System Application" that states that *it is the responsibility of the enqueue / dequeue controller to manage the integrity of quantity of data transferred into and extracted out of the FIFO.* Since these errors are acceptable, two **assume** verification directives can be added to the formal verification tool to instruct the verification tool to assume that the critical conditions will not occur. Specifically:

m_OK_Full_NoPush : **assume property** (full |-> !push);
m_OK_Empty_NoPop: **assume property** (empty |-> ! pop);

EXERCISES

1. Parking structure controller

Write properties that define the following requirements. A package and an interface are provided below.

/* Problem Statement[21]

The example FSM controls a FIFO Queue but to be practical, consider that we want to design a simple digital controller that monitors cars entering and leaving a parking lot. The operation specs are as follows.

* The controller maintains the current COUNT of the total number of cars parked in the lot.
* The lot has a limited number of parking spots, for simplicity sixteen (16).
* There are two input sensors at the entrance and exit of the lot. The sensors provide input signals to the controller as cars pass by and the controller increments/decrements the current COUNT. What if two cars enter and exit simultaneously?
* If COUNT=16, then the controller issues a FULL output signal, otherwise the OPEN output signal is on.
* There is a manual input for the lot attendant to override the COUNT for special vehicles.
* A START/STOP input is also available to the attendant, with the STOP input causing a CLOSED output signal.

*/

Figure 6.ex1 is a pictorial view of the parking lot.

Figure 6.ex1 Parking Lot overview

[21] Requirements extracted from http://bear.ces.cwru.edu/eecs_316/
Instructors: Prof. Chris Papachristou, Francis Wolff

```
package parking_pkg;
  timeunit 1ns;
  timeprecision 100ps;
// Type defintions
  typedef enum bit[1:0] {NEUTRAL, STOP, START} gate_override_t;
  typedef enum bit {CLOSED, OPEN} gate_t;
  typedef enum bit {ENABLE_COUNTER, BYPASS_COUNTER} counter_cntrl_t;
endpackage : parking_pkg

interface parking_structure_if (input wire clk, reset_n);
  import parking_pkg:: *;
  bit          parking_full;
  bit          entrance_sensor;
  bit          exit_sensor;
  gate_override_t  gate_override;
  counter_cntrl_t  counter_cntrl;

  modport parking_cntlr_if (
  output gate_t          gate_status,
  output bit             parking_full,
  input  bit             entrance_sensor,
  input  bit             exit_sensor,
  input  gate_override_t  gate_override,
  input  counter_cntrl_t  counter_cntrl);
endinterface : parking_structure_if
```

2. Handshake Model

Write properties that define the following requirements for a slave unit that acquires a bus through a handshake protocol. The protocol requirements can be summarized as follows:

1. The slave issues a bus request (*req*), and holds that request until master asserts the acknowledge (*ack*) signal, at which time the slave will deassert the bus request.
2. The master replies to bus request with a single acknowledge cycle, asserted within four cycles from the bus request.
3. In response to *ack* signal, the slave issues
   a. *Start* for 1 cycle, used for synchronization purposes by master.
   b. *Valid* for 4 cycles, used for data envelope. The *valid* signal occurs in the same cycle as the *start*. Data is not within the design of this slave.
   c. *Done* for 1 cycle, used for synchronization purposes by master. The *done* signal occurs 1 cycle after the last cycle of *valid*.
4. An active *reset* signal aborts any handshake operation.
5. All operations are synchronous to the system clock.

Figure 6.ex2 demonstrates a typical timing for the handshake protocol.

Figure 6.ex2 Handshake Protocol Timing Example

7 FORMAL VERIFICATION USING ASSERTIONS

As previously stated, the formal nature of the SystemVerilog Assertions language makes it suitable for both unambiguous documentation and computer-aided verification through simulation-based and formal property checking. This chapter addresses the formal verification aspects of SystemVerilog Assertions. It focuses on **Formal Verification** (FV) methodologies for functional verification of RTL designs. Section 7.1 provides a general introduction to Formal Verification (FV) methodology. Section 7.2 addresses the role of SystemVerilog Assertions in FV. Section 7.3 provides a case study using a traffic light controller model and formal verification of the same. Section 7.4 details coverage metrics in FV. Section 7.5 discusses emerging applications of SystemVerilog Assertions with Formal Methods. Section 7.6 addresses temporal debugging. Section 7.7 discusses the issue of simulation versus formal verification.

An additional reference in the latest developments in the verification domain from the perspectives of the user and the developer can be found in the book *Advanced Formal Verification.*[1]

---

[1] *Advanced Formal Verification*, edited by Rolf Drechsler, Kluwer Academic Publishers, Boston, ISBN 1-4020-7721-1, January 2004 http://www.wkap.nl/prod/b/1-4020-7721-1

Formal verification of digital designs (and subblocks) embellished with SystemVerilog Assertions is very useful for the following reasons:

1. Very quick verification of design iterations, without a testbench, to arrive at a working model. This enables the designers to validate their designs at the block-level. Verification engineers can focus on specifying the design properties and creating larger level simulation-based verification environments (e.g., for cluster, full chip, system level). The block-level assertions can also be re-used at higher levels as additional checkers/monitors.

2. Not only is each design-iteration fast with formal verification, but also the analysis is complete, covers complex test sequences, and provides counterexamples that demonstrate functional errors. A review of the failed properties that define the requirements, and an analysis of the counterexample waveforms help in identifying the sources of errors. Those errors can be an RTL error, a misunderstanding of the requirements, or property specification errors. In any case, the engineer is challenged to fully understand the design and requirements.

3. Corrections of these errors can be quickly verified with another run of the formal verification tools. This methodology is analogous to the edit/compilation/synthesis process used in RTL design and synthesis to eliminate early mortality (or gross errors) in the design.

4. Many vendors incorporate various static tools into their formal verification toolset, such as linting for compliance to language, synthesis and user's rules, and detection of unreachable code. They may also include FSM analysis of terminal states (i.e., no way out of a state under any condition). These embedded tools significantly help in debugging models.

5. Formal verification coverage analysis capability helps in providing a deeper understanding of the model. For example, formal verification can be used to get a witness, or a trace showing one scenario on how an assertion passed. As explained in Chapter 4, Section 4.2.1.5, the witnessing of assertions can also be done without any RTL code, using the assertions only.[2]

7.1 FV METHODOLOGY

Conventional design verification relies heavily on test vector-based simulation and on the automatic or manual analysis of the results. While this approach is by far the most understood technology and is widely used, the amount of man-hours it takes has been ever increasing as design complexity and density keeps increasing. In addition, conventional verification methodologies suffer from the infamous question of "have all corner cases been tested?" This question cannot be answered with confidence primarily because these simulation methodologies specify directed tests (i.e., inputs under normal scenarios) and expand the scenarios for different corner cases using pseudo-random tests. Recent advances in CDV (Coverage-Driven Verification) have helped specify and track these corner cases. However, the fundamental problem still remains largely unsolved, and has shifted from coding all scenarios to coding all coverage points. Note that hitting these coverage points might again require coding them as directed tests.

[2] This capability is tool dependent.

Formal Verification (FV) is an alternate method for verifying designs as the technique exhaustively explores all possible sequences of input values in an attempt to prove that each assertion is valid. For each failing assertion, the formal tool can demonstrate in simulation a counterexample sequence that results in the failure. Essentially, the formal tool creates a mathematical model of the design and the requirements, which includes a model of transitions for the system. .

A typical testbench-based methodology relies on the application of test vectors, and on the observation of results as compared to expected behavior. That observation can be manual, but is typically performed with a verifier model that understands the protocols, transactions, and expected results. FV works in almost the opposite fashion: it tries to "break" the system (or violate the expectations) by applying all possible legal input combinations. While this approach sounds GREAT, it has its own set of issues: the necessity to eliminate illegal input combinations, and the need to specify the "expected behavior" in a formal manner that can be interpreted by the tools. This chapter addresses some of these issues. FV has been under research for quite some time, and has matured aggressively. There are three major classes of FV.

1. Equivalence Checking
2. Model Checking
3. Theorem Proving

Among these, the equivalence checking technology has been widely explored by the EDA industry and mature tools are available today to perform Equivalence Checking between two representations of the same system, such as RTL and Gate-level netlists. Out of the other two classes, Theorem Proving involves heavy mathematics and has been very difficult for the average engineer/user (i.e., it requires expertise). Model Checking has been well understood in the academic media, and recently EDA tools are emerging to take advantage of Model Checking, particularly for verification of digital designs. The focus of this chapter is on Model Checking. The terms FV, Static Verification and Model Checking are used interchangeably throughout this section.

In Model Checking, properties (written in a property language, like SystemVerilog Assertions) are used to capture the design intent. A Formal Verifier reads in an RTL Design (HDL implementation) and builds a state transition graph for the system in a manner similar to an FSM. The Verifier then reads in the properties. Typically these properties represent the design specification as temporal sequences (such as request to grant protocols, illegal transitions), and they specify whether these properties hold forever or during a finite time. A Model Checker then checks whether the design implementation obeys these properties by searching the entire state space of the design, and produces counterexamples if the design fails to obey the properties.

By definition, FV is a complete proof, and hence is the ultimate verification methodology – if applicable. However, FV suffers heavily from the state space explosion problem for larger designs because it needs to search a very large number of states. An acceptable methodology is to limit the FV to blocks of manageable size (see discussion in 7.2.1). Given the complexity of today's designs, hierarchical verification is quite common, and the task of building those block-level testbenches can be avoided with block-level FV. Higher-level verification can still use testbench-based verification methodology, and many of the lower-level assertions (coded in SystemVerilog Assertions) can be used as monitors (during simulations, i.e., Dynamic ABV) that are closer to the source of any error. Figure 7.1 represents a typical design process using FV with SystemVerilog Assertions. Note that the formal verification process does not rely on a testbench, or on the verification of simulation results. Instead, the effort is shifted into specifying the design properties, and assumptions or constraints, and in the analysis of the FV results. The results and

debugging methods of FV are tool dependent. Counterexamples demonstrate cases for assertion violations. These are shown graphically from a database generated by the FV tool. That format may be VCD, for display in a vendor's tools. Note that assertions do not represent the design, but rather the characteristics of the design, and do not necessarily directly correlate to the RTL design.[3] This is advantageous since the properties present a different viewpoint of the design, rather than the RTL implementation.

The question that arises is: How many properties and assertions are required to fully characterize the design? The answer is, enough properties to clarify the design and to avoid ambiguities. Too few assertions represent an incomplete specification, which can cause a static verification tool to either approve the design as meeting requirements, or can yield false negatives about the operation of the design. Too many assertions may not necessarily meet the criteria for an acceptable specification. It may represent a lot of unnecessary work, and could even cause static verification tools to hang. There should be a good balance of properties and assertions to fully characterize the design.[4] There are evolving metrics to help in this aspect. Those metrics are discussed in section 7.4. Since formal verification is a fairly new concept, the authors envision that there will be several iterations between design and property development and FV tool use. Specifically at block-level (where FV is better suited as of today), FV tools typically takes a few minutes to come up with a counter-example, and hence the iteration loop doesn't take long to close (obviously that is a function of the complexity of the design as well).

Figure 7.1-1 Typical Formal Verification Design Process with SystemVerilog Assertions

[3] An assertion is a "statement about a design's intended behavior" (From *Assertion-Based Design*, Foster). A property is a characteristic of the design. The distinction between the two terms is marginal, and the two terms are often used interchangeably

[4] Readers are encouraged to read Chapter 8 that addresses guidelines in writing properties.

7.1.1 Model Checking Expectations and Rules

A first and foremost requirement for Model Checking is that designs need to be synthesizable and cycle-based. Model Checking tries to extract mathematical equations from the design and the properties and tries to prove/disprove the equations. A natural limitation of such an approach would be that designs that contain elements such as phase-locked loops (PLLs), behavioral memories, etc. cannot be converted into logic equations, thus they can't be proven using the formal engines. Typically a model checker treats non-synthesizable blocks as black-boxed, making the outputs of those blocks free ranging variables.

In formal verification, clocks are not handled in the same manner as in simulation, which is typically event driven. Gated clocks are assumed to always be active and the actual gating logic is ignored. Assertions with multiple clocks that are integer ratio of each other may present a proof issue. For example an assertion described with one signal being sampled at 66 MHz clock and another signal at 100 MHz clock may not prove the assertion multiple times because of the way the clocks overlap (this process is tool dependent).

Another important aspect in Model Checking is the size of designs that it can handle. The size of a design is a function of state-levels, RTL gates, number of assertions, class of formal verification engine, and interestingly enough the tool algorithms. Formal verification engine approach can be "bounded" (Bounded Model Checking, BMC) in the number of cycles the engine tries to use to check for assertion violations. They can also be "unbounded" (UBMC), where the number of cycles is not bounded (see section 7.4.1).

There is a debate as to which type of engine (BMC, UBMC) can handle large designs. Proponents of BMC claim that bounded engines can run much larger assertions than unbounded. For example, bounded model checking has been successfully used on non-trivial designs (with reasonably complex assertions. 5000 flop, 80k gates) for 200 clock cycles. However, such a model may not complete when using an unbounded engine. The drawback of BMC is that the logic is only evaluated over a specific number of clock cycles. If a design is run using a bounded engine over 200 clock cycles and the assertion passes, there is no guarantee that the assertion will not fail at cycle 250. Proponents of UBMC claim that an unbounded engine is capable of incorporating more powerful reductions than a bounded engine. In fact, they have demonstrated that unbounded engines significantly outperform bounded model checking on a testbench. But in most of the real examples, BMC has been a better success. Another interesting observation has been that a hybrid formal verification yields better results than a pure FV as the hybrid approach doesn't suffer from the state space explosion problem that limits the usage of pure FV.[5]

With formal verification, the size of the driving logic for each respective assertion may play a significant role in the performance of the tool, and reductions may be applied that do not scale in a monotonic fashion with the size of the driving logic. Some vendors may extract the cone of logic associated with the assertion, and that could define the size limit. Other formal tools may read the entire design into memory at once and try and build a state space to be explored by the formal engine; those tools will indeed be limited by design size, not assertion size. The size limit is also more related to the number of flip-flops (i.e., the state-space) than the number of gates. In addition, there are other techniques that some formal verification vendors adopt to minimize the runtime. These techniques may include hierarchical, incremental, and distributed model checking. Some of the earlier generation model checkers used BDD (Binary Decision Diagrams)

[5] NVIDIA Results with MG
http://biz.yahoo.com/prnews/041027/sfw068_1.html

as a basis to build the state space, and that puts a tremendous size limitation on the number of state elements that could be handled. As the technology matured, Ordered, Reduced Ordered BDDs are being used, which helps in the handling of larger designs.

The types of assertion failures that a user can expect from formal verification generally do not run for many clock cycles. However, that issue is debatable as there are vendors that claim support of assertions verified over very long number of clock cycles. If a formal tool does not support a failure trace for thousands of clock cycles, then such failures are better debugged with simulation-based techniques. Another verification approach would be to split such long assertions into multiple small assertions.

Note: Depending on the design, formal tools often return the shortest possible failure trace. Thus, instead of returning a failure track of 1000 clock cycles, which can be very difficult to follow, the tools return the shortest trace. An assertion that requires a large number of clock cycles to fail can be troublesome for a bounded engine. However, formal verification tools (this is very design and vendor dependent) were successful in generating several failure traces over 1000 clock cycles.

7.2 Role of SystemVerilog Assertions in FV

This section correlates SystemVerilog Assertions with the FV process, and addresses the similarities and differences between the application of SystemVerilog Assertions in Dynamic ABV and FV. It also presents some practical applications of SystemVerilog Assertions + FV as a combined methodology to tackle some of the niche areas of verification and design in general.

7.2.1 SystemVerilog Assertions in Formal Specifications

FV, as a technology, has been around for at least half-a-decade and has been used by some of the large semiconductor companies for verification with reasonable success.[6] One of the main hurdles for FV to become accepted in main stream FPGA and ASIC Design Verification has been the lack of standardization in "specifying the behavior" of the system under verification. Almost every FV implementation used its own proprietary language to specify the design behavior. However, the standardization of property languages like PSL and SystemVerilog Assertions helped FV tool vendors to support those languages as their main interface. A FV tool will require at least two inputs: the design implementation (RTL), and a set of properties (e.g., SystemVerilog Assertions). It may also require information about design constraints. As shown in Figure 7.1-1, SystemVerilog Assertions properties can be part of a specification, and it is becoming common to have formal, machine-executable specifications defined for the protocols.[7] A formal specification provides an unambiguous golden reference model of the properties and requirements of a design, independent of the implementation. Refer to Chapter 4 for examples of such specifications.

---

[6] http://www.cl.cam.ac.uk/users/jrh/papers/fparith.pdf
A machine-checked theory of Floating point arithmetic, John Harrison, Intel Corporation

[7] Verification of the Futurebus+ cache coherence protocol, by E.M.Clarke et. al.
http://www-2.cs.cmu.edu/~emc/ Presents a formalization of an IEEE standard protocol and demonstrates that such formalization helps in finding errors in the Specification itself. Though this particular citing doesn't use SVA as their language, the concept still holds.

7.2.2 SystemVerilog Assertions Usage in FV vs. Dynamic ABV

Though the SystemVerilog Assertions definition is independent of the underlying verification methodology (i.e., FV vs. Dynamic ABV), some constructs within SystemVerilog Assertions have more sense/relevance to FV than to Dynamic ABV. Since FV tries to explore all possible combinations of inputs/states, it is very important to exclude illegal combinations of inputs. SystemVerilog Assertions supports the **assume** construct for constraining inputs. Typically the **assume** construct is used to describe timing relationships among inputs and to constrain their values.

An important point to consider in using SystemVerilog Assertions with FV is the handling of the *reset* input. Since, by definition, FV should explore all possibilities, the tool should theoretically toggle the *reset* as well. However, toggling *reset* is not a very frequent occurrence, and will not serve much to the functional verification task. Some formal verification tools automatically detect *reset* and clocks, and assume that the proof starts right after *reset* is deasserted. It is also possible to have the tool provide a simulation dump and simulation time to get a starting condition for the proof.

However, some other tools require the user to restrict the *reset* behavior during FV runs. Considering an active LOW reset signal as *reset_n*, the following property can be assumed.

```
property p_rst_low_first_cycle_then_high;
  !reset_n |=> reset_n;
property p_rst_low_first_cycle_then_high
mp_rst_low_first_cycle_then_high : assume property (p_rst_low_first_cycle_then_high);

property p_rst_high_after_first_cycle;
  reset_n |=> reset_n;
property p_rst_high_after_first_cycle
mp_rst_high_after_first_cycle : assume property (p_rst_high_after_first_cycle);
```

The above properties direct a formal verifier to assume that:

- At the beginning of verification, reset_n = 1'b0, meaning that the system was under reset condition (active LOW reset).
- Thereafter, and throughout the verification process reset_n = 1'b1, i.e., the reset be treated as inactive.

There are some special scenarios in which the user may want to intentionally toggle the **reset** signal during a session. This may be needed to check conditions such as an FSM deadlock/recovery after a **reset**, or an instruction pipeline flushing due to **reset**. Thus, it is recommended to keep the **reset** constraints in separate modules bound to the design using the **bind** directive. When checking for the aforementioned special properties, one of modules with the **reset** constraint omitted can be used in the binding.

7.2.2.1 Same Inputs in Antecedent and Consequent

Inputs are sometimes used in both the antecedent and consequent of a property. They are often useful in simulation to ensure compliance to protocol by either the higher-level design or by the

testbench. For example, the property pACK_START_VALID_DONE states that both the trigger condition (the antecedent) and the EFFECT sequence (consequent) rely on the input *ack_in*.

```
sequence qACTIVE_REQUEST; req && !ack_in;
property pACK_START_VALID_DONE;
 @ (posedge clk) disable iff (!reset_n)
 qACTIVE_REQUEST ##1 ack_in |=>
    start && !ack_in && valid ##1
    (!start && !ack_in && valid)[*3] ##1
    (!start && !valid && done).
 endproperty : pACK_START_VALID_DONE
apACK_START_VALID_DONE : assert property (pACK_START_VALID_DONE);
```

ack_in used in antecedent

input *ack_in* also used in consequent

When the same input signals are used in the antecedent and consequent side of a property, FALSE NEGATIVE errors in formal verification can be created because the formal verification tool can assume inputs to have any value at any cycle. This is not a design fault, but rather an error in the verification environment or assumptions of the design being verified.

Techniques to avoiding false negative cause by input terms in consequent

For formal verification the user must specify properties to define the assumed values of inputs used in antecedent and consequent portions of properties. For example, an assumption for the above handshake property can be specified as follows:

```
property pCONSTRAINT_ACK_START_VALID_DONE ;
 @ (posedge clk) disable iff (!reset_n)
 qACTIVE_REQUEST; ack_in |=>
    !ack_in ##1
    (!ack_in)[*3];

 endproperty : pCONSTRAINT_ACK_START_VALID_DONE
 mpCONSTRAINT_ACK_START_VALID_DONE : assume property
                                (pCONSTRAINT_ACK_START_VALID_DONE);
```

constraint on input *ack_in*

7.3 CASE STUDY - FV OF A TRAFFIC LIGHT CONTROLLER

7.3.1 Model

This design represents a simple traffic light controller for a North-South and East-West intersection. The North-South is the main road, and is given the GREEN light unless a sensor on the East-West Street is activated. When that occurs, and the North-South light was GREEN for sufficient time, then the light will change to give way to the East-West traffic. The design also takes into account emergency vehicles that can activate an emergency sensor. When the emergency sensor is activated, then the North-South and East-West lights will turn RED, and will stay RED for a minimum period of 3 cycles. Figure 7.3.1-1 is a view of the intersection. The model has the following interfaces:

The model makes use of the following type definition:

```
typedef enum { OFF,   // power off
               RED, // red state
               YELLOW,      // yellow state
               GREEN,       // green state
               PRE_GREEN}   // state before green
               lights_t;
```

```
module trafficlight
(
 output lights_t ns_light,  // North/South light status, Main road
 output lights_t ew_light,  // East/West light status
 input           ew_sensor, // East/West sensor for new car
 input           emgcy_sensor, // emergency sensor
 input           reset_n,   // synchronous reset
 input           clk        // master clock
);
 timeunit 1ns;
 timeprecision 100ps;
 parameter FAIL   = 1'b0;
```

Figure 7.3.1-1 North-South and East-West Intersection.

Since the traffic lights are timed, two timers are used in this first implementation:

```
logic [1:0]   ns_green_timer;  // timer for NS light in Green
logic [1:0]   ew_green_timer;  // timer for EW light in GREEN
```

A high-level view of the traffic light controller is shown in Figure 7.3.1-2. The architecture includes two FSMs and the timers. The operation of the machine is very simple:

1. North-South remains GREEN unless one of the sensors is activated.
2. If the North-South light is RED and the North-South GREEN timer is 3, then the light will switch to GREEN.
3. If the North-South light is YELLOW, it will switch to RED.

4. If the North-South light is GREEN, and the emergency sensor is activated, the light will switch to YELLOW. Also, if the North-South timer is 3, and the East-West sensor is activated, the North-South light will switch to YELLOW.
5. The North-South GREEN timer is reset to zero at reset, or when the North-South light is YELLOW. Otherwise, it increments at every clock until it reaches the maximum count of 3.
6. The East-West light switches from RED to a PRE_GREEN state (to allow the North-South light to go to YELLOW) if the North-South timer is 3 and the East-West sensor is activated.
7. If the East-West is PRE_GREEN, it will switch to GREEN at the next clock.
8. If the East-West light is YELLOW, it will switch to RED.
9. If the East-West is GREEN and either the emergency sensor is activated or the East-West timer reaches a count of 3 then it switches to YELLOW.
10. The East-West GREEN timer is reset to zero at reset or when the East-West light is YELLOW. Otherwise, it increments at every clock until it reaches the maximum count of 3.

Figure 7.3.1-2 High-Level View of Traffic Light Controller

Advance Notification:

On the surface, this architecture looks acceptable. However, it is flawed, as there are illegal behaviors, as demonstrated by formal verification tools. For example, it is possible for both the North-South light and the East-West light to be GREEN in the same cycle.

Even though the implementation of this synthesizable model is flawed, it is used in this book to demonstrate the value and power of ABV with SystemVerilog Assertions in formal verification. Later on in this chapter, a good traffic light controller is modeled and verified.

7.3.2 Basic requirements

Using the pattern approach to document the ABV requirements, the following requirements can be defined for this traffic light controller example.

```
//| # Pattern name # NorthSouth
//|
//| # Problem # The NorthSouth light is the main street light. It must remain
//| GREEN for ns_green_timer == 3 before it can switch. Timer ns_green_timer
//| will count to 3, and remain at 3 until light changes. If ew_sensor is activated
//| (new car), then light will switch for the East-West light when the minimum
//| time for ns_light is satisfied (i.e., 3 cycles). The ew_green_timer will count to 3,
//| at which time, the ns_light will regain control of GREEN. This gives the
//| North-South traffic a higher priority than the East-West traffic.
//| An emgcy_sensor shall turn all lights to RED. The lights shall remain RED for a
//| minimum of 3 cycles.
//| Going from GREEN to RED requires a YELLOW for 1 cycle.
//|
//| # Motivation # Simplicity and conformity.
//|
//| # Context #
//|
//| # Solution #
//|   2 timers ns_green_timer, ew_green_timer
//|   Light cycle: GREEN (3 cycle min), YELLOW (1 cycle), RED (until GREEN)
```

7.3.3 SystemVerilog Assertions for Traffic Light Controller

Based upon the requirements, several assertions can be expressed prior to writing the RTL. Below (in file *trafficlight.v*[8]) are the assertions written for this design.

```
`define true 1
typedef enum {OFF, RED, YELLOW, GREEN, PRE_GREEN} lights_t;
module trafficlight
 (
  output lights_t ns_light, // North/South light status, Main road
  output lights_t ew_light, // East/West light status
  input          ew_sensor, // East/West sensor for new car
  input          emgcy_sensor, // emergency sensor
  input          reset_n,  // synchronous reset
  input          clk       // master clock
 );
 timeunit 1ns;
 timeprecision 100ps;
 parameter FAIL  = 1'b0;

 //| # Pattern name # NorthSouth
 //| # Problem # The NorthSouth light is the main street light. It must remain
 //| GREEN for ns_green_timer == 3 before it can switch. Timer ns_green_timer
```

[8] Please refer to file model trafficlight.sv for the complete RTL code.

```
//| will count to 3, and remain at 3 until light changes. If ew_sensor is activated
//| (new car), then light will switch for the East-West light when the minimum
//| time for ns_light is satisfied (i.e., 3 cycles). The ew_green_timer will count to 3,
//| at which time, the ns_green_timer will regain control of GREEN. This gives the
//| North-South traffic a higher priority than the East-West traffic.
//| An emgcy_sensor shall turn all lights to RED.
//| Going from GREEN to RED requires a YELLOW for 1 cycle.
//| # Motivation # Simplicity and conformity.
//| # Context #
//| # Solution #
//|   2 timers ns_green_timer, ew_green_timer
//|   Light cycle: GREEN (3 cycle min), YELLOW (1 cycle), RED (until GREEN).
//|
//| # Considerations #

// Based upon the requirements, several assertions can be expressed prior to
// writing the RTL. Below are the assertions written for this design.

// sequence defintion
sequence qEmgcy;
  @ (posedge clk) emgcy_sensor;
endsequence : qEmgcy

// ***************************************************
// Safety property
property Never_NS_EW_ALL_GREEN;
  @ (posedge clk) disable iff (!reset_n)
   not (ns_light==GREEN && ew_light==GREEN);
endproperty : Never_NS_EW_ALL_GREEN
Never_NS_EW_ALL_GREEN_1 : assert property (Never_NS_EW_ALL_GREEN);
// ***************************************************
// State of lights at reset
property nsLightAtReset;
  @ (posedge clk) disable iff (!reset_n)
   reset_n==1'b0 |=> ns_light==OFF;
endproperty : nsLightAtReset
nsLightAtReset_1 : assert property (nsLightAtReset);
//
property ewLightAtReset;
  @ (posedge clk) disable iff (!reset_n)
  reset_n==1'b0 |=> ew_light==OFF;
endproperty : ewLightAtReset
ewLightAtReset_1 : assert property (ewLightAtReset);
// ***************************************************
```

```
// State of lights during emergency
 // Lights switch from GREEN to YELLOW to RED
 property NsLightsWhenEmergency;
  @ (posedge clk) disable iff (!reset_n)
  emgcy_sensor |=> `true ##1  ns_light==RED;
 endproperty : NsLightsWhenEmergency
NsLightsWhenEmergency_1 : assert property (NsLightsWhenEmergency);
```

```
//
property EwLightsWhenEmergency;
  @ (posedge clk) disable iff (!reset_n)
   emgcy_sensor |=> `true ##1 ew_light==RED;
endproperty : EwLightsWhenEmergency
EwLightsWhenEmergency_1 : assert property (EwLightsWhenEmergency);
// ************************************************
// Safety, GREEN to RED is illegal.  Need YELLOW
property NsNeverFromGreenToRed;
  @ (posedge clk) disable iff (!reset_n)
   ns_light==GREEN ##1 (ns_light==RED) |-> FAIL;
endproperty : NsNeverFromGreenToRed
NsNeverFromGreenToRed_1 : assert property (NsNeverFromGreenToRed);
```

```
property EwNeverFromGreenToRed;
  @ (posedge clk) disable iff (!reset_n)
   (ew_light==GREEN) ##1 (ew_light==RI
endproperty : EwNeverFromGreenToRed
EwNeverFromGreenToRed_1 : assert pro
// ********************************
// The NorthSouth light is the main street li
// It must remain GREEN for ns_green_tin
// Timer ns_green_timer will count to 3, ar
 property NsGreenForMin3Ccyles;
  @ (posedge clk) disable iff (!reset_n || e
   $rose(ns_light==GREEN) |=>
       ns_light==GREEN [*2]; // abort em
endproperty : NsGreenForMin3Ccyles
NsGreenForMin3Ccyles_1 : assert prope
```

```
// ********************************
// East-West North-South Lights with Eas
// If ew_sensor is activated (new car), the
// when minimum time for ns_light is sati
// at which time, the ns_green_timer will
 property EwNewSensorActivation;
  @ (posedge clk) disable iff (!reset_n || e
   ( (ew_sensor==1'b1) && ns_green_tim
       ns_light==YELLOW ##1 ew_light=
endproperty : EwNewSensorActivation
EwNewSensorActivation_1 : assert pro

//********************************
// North-South light to GREEN
 property NsLightToGreen;
  @ (posedge clk) disable iff (!reset_n
   (ew_green_timer==2'b11) |=>
     `true ##1 ns_light==GREEN;
endproperty : NsLightToGreen
NsLightToGreen_1 : assert property
// ********************************
```

```
ED) |-> FAIL;

perty (EwNeverFromGreenToRed);
*****************
ght.
ner == 3 before it can switch.
d remain at 3 until light changes.

ngcy_sensor)

ngcy_sensor);

rty (NsGreenForMin3Ccyles);
```

```
*****************
t-West sensor
n light will switch for the ew_light
sfied. ew_green_timer will count to 3,
regain control of GREEN.

emgcy_sensor)
er==2'b11 ) |=>
==GREEN; // abort emgcy_sensor==1'b1);

perty (EwNewSensorActivation);

*****************


emgcy_sensor)


NsLightToGreen);
*****************
```

```
// East-West light to RED
  property EwLightToRed;
   @ (posedge clk) disable iff (!reset_n || emgcy_sensor)
   (ew_green_timer==2'b11) |=>
     `true ##1 ew_light==RED;
  endproperty : EwLightToRed
  EwLightToRed_1 : assert property (EwLightToRed);

  reg [1:0]   ns_green_timer;  // timer for NS light in Green
  reg [1:0]   ew_green_timer;  // timer for EW light in GREEN

  lights_t ns_state  ;   // register that holds the current state of the FSM
  lights_t next_ns_state ; // register that holds the next state of the FSM
  lights_t    ew_state  ;   // register that holds the current state of the FSM
  lights_t    next_ew_state ; // register that holds the next state of the FSM

// ....

endmodule : trafficlight
```

7.3.4 Verification

It is generally recommended to use a light weight simulation prior to applying FV on a give block to iron out any issues such as:

- Clock not being connected properly.
- Polarity of reset being wrong.
- General connectivity check across modules etc.

While FV may be able to point those errors, the real power of FV lies in proving and/or falsifying properties than to locate such errors. The above model was verified with Synopsys Magellan and it revealed six critical failures in the design, as shown in Figure 7.3.5-2

Figure 7.3.5-2 Summary of Formal Verification Results on Traffic Light Controller Design with SystemVerilog Assertions, Synopsys Magellan

A study of the counterexamples demonstrates that certain sequences of inputs can cause properties to fail. Some of the sequences are fairly complex, and would have required extensive simulation with pseudo-random patterns to arrive at the failed situation. The East West light to RED (*EwLightToRed_1*) assertion failure is demonstrated with a counterexample shown in Figure 7.3.5-3

Figure 7.3.5-3 Counterexample for East-West light not to RED, Synopsys Magellan

7.3.5 Good Traffic Light Controller

The traffic light controller demonstrated in the previously section represents a model where the East-West and North-South FSMs are loosely tied, and there lies the source of errors such as both East-West and North-South lights turning GREEN in the same cycle. Of course, that erroneous condition only happens under specific sequences, as demonstrated by formal verification.

To resolve this issue, a centralized architecture will be used. The new design relies on the North-South FSM being the master controller. The East-West slave FSM makes an *ew_green_request* whenever it wants access to the light. That request is granted with the *ew_green_grant* handshake provided by the North-South FSM when the North-South goes YELLOW. In addition, to maintain this centralized control, the North-South FSM will inform the East-West FSM to go RED with the *ew_to_red_cmd* command. That command is a function of the value of the emergency sensor, the value of the East-West sensor, and the length of time that East-West light stayed GREEN. A centralized North-South GREEN timer, instead of two independent timers, controls that time. Figure 7.3.5-1 represents a high-level view of this architecture.

Figure 7.3.6-1 Good Traffic Light Controller Architecture

The SystemVerilog model for this new architecture is in file ch5/trafficlightok10.sv. Some minor changes to the SystemVerilog Assertions code were made to reflect additional knowledge about the design.

Reflections:
Some reflections on the derivation of this model are in order:

1. It took about 10 iterations to arrive at a working model.

2. Each iteration went fairly fast because formal verification was used without a testbench. As the design matured, and more analysis was done. it took longer between iterations (early mortality effect). This methodology is analogous to the edit-compilation/synthesis process used in RTL design and synthesis to eliminate the gross errors.

3. Linting of the model significantly helped in debugging the model prior to performing formal verification. Linting uncovered unreachable code.

4. Formal verification quickly arrived at errors in the design. along with counterexamples that demonstrated the problems. Corrections of these errors were quickly verified with another run of the formal verification tool.

5. Formal verification also demonstrated that some properties needed modifications since they were not properly expressed. These incorrectly stated properties challenged the author of the properties in his understanding of the requirements. That cycle caused a better understanding of the operation of the machine and the requirements.

6. Formal verification also demonstrated some false negatives for some assertions. These were analyzed, and were considered acceptable.

7. Coverage analysis (explained later) using formal verification helped in providing a deeper understanding of the model.

7.4 FV COVERAGE METRICS

As with any verification methodology, metrics to measure the quality of verification are indispensable. While simulation-based verification and its associated metrics have evolved over the years, formal verification is fairly new, and hence its metrics are still evolving. This section presents few of the known metrics.

7.4.1 Proof Radius

As mentioned earlier, formal verification tries to prove the properties across the entire state space and over several number of clock cycles. As the design size grows, the state space grows and formal verification tools get into the infamous state space explosion problem. One way to solve this problem is to limit the number of clock cycles over which each property is verified. This is referred to as **Proof Radius**. A design being verified with a proof radius of 10 means that within 10 clocks the design doesn't violate the properties for any combination of inputs, but doesn't provide any guarantee that it won't violate afterwards. Typically the micro-architect and designer should be able to provide an idea of what is the optimum proof radius for a given design and properties under verification. An extension of this Proof Radius concept is "Bounded Model Checking", which is "Model Checking" bound to a certain number of clocks after the initial state of all the properties are verified.

7.4.2 Explored State-Based Coverage

A FV tool converts the design implementation into a state transition graph, and then proves the properties across this state space. One metric that could be used to measure the quality of the properties (and hence the verification process) is to keep track of how many states the verifier examines. This is somewhat similar to code coverage in traditional simulation, but is rather more closely related to the formal verifier. Static proofs of reachability properties are complimentary to simulation coverage in terms of exposing unreachable areas. This can apply to both structural coverage (i.e., FSM arc transition) and functional coverage (i.e., coverage of assertion triggering condition in a handshake). Recently Averant, a model checking EDA tool vendor, was granted a patent on Static Coverage Technology [9], which is more closely related to the question on whether there are enough assertions (similar to 0-In [7.4.3]).

7.4.3 Flip-flop to Property Distance

A simple metric of property-based verification could be the **property to a flip-flop ratio**, i.e., how many state elements are being monitored by the properties. This is more like a lint metric, but it can identify some parts of the design not covered by any property. This is a modified form of what 0-in calls Static Structural coverage or MSD – Minimum Sequential Distance between a FF and a nearby assertion[10].

7.4.4 Functional Coverage Points

SystemVerilog Assertions supports constructs to code functional coverage points (FCP) that represent the interesting scenarios that a verification engineer would like to test.[11] This is similar to the FCP supported in a modern HVL (such as Verisity's e[12], Synopsys's VERA[13] etc.) and the OVL's coverage monitors.[14] FCPs are a great form of verification metric as they represent "functionality" of the design rather than just "a piece of code".

7.5 EMERGING APPLICATIONS OF SYSTEMVERILOG ASSERTIONS WITH FORMAL METHODS

This section discusses some of the novel applications of SystemVerilog Assertions with formal methods, such as performance evaluation of digital systems, automatic stimuli generation etc.

7.5.1 SystemVerilog Assertions Based Performance Evaluation of Digital Systems

This sub-section describes a novel application of SystemVerilog Assertions for performance analysis and/or evaluation of digital systems using SystemVerilog Assertions with Formal methods. Consider the case of the simple round-robin arbiter that serves 4 requesters. Depending on the implementation, the maximum request-to-grant delay can vary. If the architecture requires that the maximum allowed request to grant delay is 20 clocks (arbitrary number assumed for the sake of discussion), it is not always possible to verify this using vector-based simulation alone. However, a carefully designed testcase can verify this requirement, but when the implementation changes, even slightly, it may call for a modification

[9] http://www.averant.com/patent.html

[10] http://www.0-in.com/news/PR_030526_Structural_Coverage.html

[11] Chapter 3 addresses SVA functional coverage. Also, see section 4.2.3.1 and 6.1.2.

[12] Http://www.verisity.com

[13] Http://www.synopsys.com

[14] Http://www.verificationlib.org

to the test and careful observation of the results. Instead, a SystemVerilog Assertions property can be specified with a range as:

```
property pGrantLessThan20Cycles;
  req_bus[0] == 1'b1 |-> [*0 :19] ##1 gnt_bus[0] == 1'b1;
endproperty : pGrantLessThan20Cycles
aGrantLessThan20Cycles : assert property (GrantLessThan20Cycles);
```

A formal verifier (Model Checking tool) can analyze the design implementation and can exhaustively prove that this property is invariant. It can also provide a counterexample if there is one. This is almost opposite to traditional, vector-based performance testing in the sense that the verification starts from an infinite possibility and eliminates any unexpected behavior (of inputs).

Though the above example may sound trivial, this idea can be very effectively used in complex designs, such as an ATM Switch where the switch must process a packet every four clocks. Another application could be to verify that a processor pipeline never gets stalled. Though the idea of writing such properties/assertions is not new, the way it is being interpreted/applied is different and can be of high value for performance evaluation.

7.5.2 Hybrid (dynamic and formal) Verification

Given the present day limitations of pure formal verification, a new breed of ideas is being introduced in the verification arena, which essentially tries to extract the best of both worlds, "Simulation and Formal Verification". This is often referred to as Semi-formal Verification. One of the issues with pure formal verification is that the tool selects the initial state of all state elements (FFs and latches). This may lead to unintended modes of operation and may report failures (known as False Negatives). As the design complexity increases, the number of such elements increases, and that essentially requires the verification engineer to spend considerable time in setting up the initial states (through **assume** directives within SystemVerilog Assertions). This is often viewed as a duplication of a conventional testbench development effort.

Semi-formal verification tries to solve this problem. It monitors the design throughout a traditional simulation method, and then stores the values of all state elements when a "checkpoint" is reached. The stimulus vectors may be a set of pseudo-random values on the inputs. Thereafter, a FV engine is started with its initial states saved from simulation checkpoint. The users can specify the number of clocks of exhaustive analysis that needs to be performed (known as Proof Radius), and the FV engine tries to prove the properties within the given limits. While this may sound tool specific, the concept is generic and can be extended to derive SystemVerilog Assertions assumptions and constraints from a previous simulation run. A FV tool can read in such constraints and perform exhaustive analysis from a well-defined initial state. The concern with this approach is that the success relies heavily on the choice of the "checkpoints" and on "the quality of the testbench".[15] Such class of semi-formal tools utilizes the simulation to provide a good starting point, and then performs a pure Formal Verification. This class is susceptible to false negative/positives due to incomplete/over constraints. The primary reason for such false results is that once a FV is initiated, the constraints are ignored.

Another class of semi-formal tools utilizes the constraints to generate "intelligent random vectors." It also uses formal techniques (bounded model checking, specifically) to guide the

[15] 0-In Search is a well known example for this class of tools (http://www.0-in.com)

random vectors to target the vectors to make the assertions fail.[16] An additional benefit of this approach is that it is easier to look at the generated waveforms to see if the constraints are behaving as intended.

7.5.3 Directed Random Test Generation from SystemVerilog Assertions

A new trend is to use the constraints to derive intelligent random stimulus, which can be used by traditional simulation methodologies. For simple blocks (where the number of possible modes of operation is not very high (refer to previous sub-section 7.5.2) a testbench can be automatically generated for use with an HDL simulator.[1] Since this technology is fairly new, it is advisable to use a code coverage tool to see the effectiveness of such tests and fine-tune the constraints as required.

7.5.4 Achieving hard-to-hit functional coverage goals using Formal Methods

With CDV (Coverage Driven Verification) gaining acceptance, the class and complexity of coverage points being defined is becoming more mature. FCPs (Functional Coverage Points) are "functional scenarios" envisioned by the user, and represent "the likelihood of errors in the design implementation". FCPs do not necessarily define an "error location", but rather its "likelihood". The real motivation behind such FCPs is that if such scenarios can be covered through pseudo-random tests, then it is highly likely that the scenarios "in and around" that coverage point would also be exercised. FCPs are LEGAL and VALID scenarios that a user may want to test, but yet do NOT represent a failure. Instead, they represent a "corner case" or a "performance scenario" such as back-to-back requests; an ATM switch switching at "wire-speed"; a set of continuous high priority interrupts leading to very long service latencies for lower priority interrupts, etc. FCPs may also represent "any interesting scenario" (not necessarily as "stressful" as defined above). They essentially mirror the traditional "test plan" in a more trackable and formal manner. Traditional test plans include sections as:

1. Targeted and desired (i.e., what a user intends to test) test Scenario sets – with CDV these will be translated to FCPs
2. How to go about achieving them.
3. What's the expected behavior of the DUT.

With modern verification languages offering solid FCP provisions, "intention of a test" is relatively easy to capture as a coverage point. With self-checking testbenches, the modeling of the expected behavior in every test is no longer needed (except in some special cases). However, the challenge is in the method used to arrive at such coverage scenarios. One solution is to use pseudo-random verification techniques and let the automation do more of the work to reach such coverage points/goals. Under CDV, the initial results appear positive, as the target seems reachable. However, it soon becomes evident that the functional coverage stalls, and may reach an 80 % limit. The remaining 20 % coverage is difficult to attain, as per the well-known 20/80 rule (20% of verification task consumes 80 % of the project time). Unfortunately, in the interest of meeting deadlines, verification engineers tend to write focus tests just to cover the coverage holes. This approach means that coverage is achieved by testing only specific scenarios, but NOT "around these scenarios". This defeats the original purpose of FCPs altogether. The verification engineers are not the only ones to be blamed, as the scenarios could be very complex, and may depend on the understanding of the verification environment and the design. Such an approach also requires a long time to code and maintain specific tests (an error fix may cause the test not to achieve its goal).

---

[16] @HDL's *@Verifier* has a component of this class and is termed as "DCS" – Directed Constrain Solver.

Though hard-to-hit coverage points represent LEGAL scenarios, the creation of such scenarios in simulation is a daunting task. Emulation represents an alternative, as low cost emulators with added debug capabilities are emerging, and a huge number of cycles can be exercised in a reasonable amount of time. The addition of synthesized assertions to the emulation would guarantee that the many cycles being emulated are scrutinized for correctness just as much as in a standard simulation. Assertions combined with emulation would therefore make it easier to reach this initial goal of 80% coverage, with the added benefit of being able to actually measure the achieved coverage. This was not the case with older emulators where one was running almost blind, without the added benefit of assertion verification.

Recently, formal verification techniques are being applied to help solve this problem of hard-to-attain functional coverage points. As mentioned earlier, FV on a large design gets into "state space explosion" problem. However, given a "specific" scenario, it is relatively easier for a formal method to trace a sequence of input combinations to arrive at such FCPs because the search space in such a case is relatively small compared to the entire breadth and depth of the design. To be able to use FV in such a scenario, one needs to "challenge" a FV tool to compute a possible trace that would lead to the FCP scenario. This could be achieved by defining a "property/assertion" that asserts that the defined scenario in the FCP will NEVER occur. It is very important to understand that such a property is NOT an assertion that the design is expected to obey. Note that some FV tools offer the capability to "witness" assertions by asking the tool to get a witness, i.e., a trace showing one scenario on how the assertion passed.

As previously stated, using simulation to find a sequence that meets the coverage requirement is a difficult task, particularly for complex designs. However, instead of simulation, a formal verification can be employed to get a possible trace for specific testcases. The approach is to write a non-requirement property (i.e., fictitious) that will assert a "never" of the coverage point. This is an assertion that specifies that such a case is NOT possible, even though the requirements and the design dictates that such a condition is possible. In fact, the purpose of this assertion is to have the formal verification FAIL, and come up with a case that demonstrates such a complex LEGAL case.

NOTE: Inverting a passing assertion to show a specific scenario is useful when the tool does not offer the capability to "witness" an assertion.

Witnessing of an assertion is defined as a provision of a tool to demonstrate an example where the assertion passes. This is similar to counterexamples that demonstrate the sequence of event that causes an assertion fails.

Tools may also convert these witnesses and counterexamples into testbench sequences for subsystem verification.

1. If a tool does not support witnessing of assertions, then write a property that addresses a **non-occurrence** of expected coverage points. This is an assertion that specifies that there is no case (i.e., **never**) that can be created such that the coverage of keyed legal sequence is satisfied. Since the requirements and the design meet the coverage sequence, the formal verification tool will create such a case.

2. Write replicated properties (if the tool supports it), or individual properties to let the formal verification compute the latencies of critical timing.

To further elaborate this concept, consider the traffic light example discussed in Section 7.3. Assume that there is a need in the testplan for a scenario to test the case where the East-West light transitions to GREEN twice in 9 cycles as shown in Figure 7.5.4-1. In addition, there is a need to know the minimum latency between a fall of the East-West light from GREEN to a rise of the East-West light to GREEN.

Cycle	-1	0	1	2	3	4	5	6	7	8	9
Ew_state	RED		X	X	X	X	X	X	X	X	

Figure 7.5.4-1 Example of a hard-to-achieve coverage scenario

Using SystemVerilog Assertions, one could code this condition as follows:

```
sequence qRose2RoseEwGreen8;
        $rose(ew_is_green) ## 9  $rose(ew_is_green);
endsequence : qRose2RoseEwGreen8

property pCov_Rose2RoseEwGreen8;
  not qRose2RoseEwGreen8;
endproperty : pCov_Rose2RoseEwGreen8
a pCov_Rose2RoseEwGreen8 : assert property  (pCov_Rose2RoseEwGreen8);
```

In a good model, this assertion would fail as expected because this is a legal sequence, and the formal verification tool would demonstrate a counterexample. Similarly, to test for a latency of 4 between a fall of the East-West light from GREEN to a rise of the East-West light to GREEN one could write:

```
sequence qFell2RoseEwGreen4 = {$fell(ew_is_green); [*4]; $rose(ew_is_green)};
property Cov_Fell2RoseEwGreen4 = not qFell2RoseEwGreen4;
a_Cov_Fell2RoseEwGreen4 : assert Cov_Fell2RoseEwGreen4;
```

A set of such properties can be defined to determine the minimum case where the assertion still fails. Thus, the sequence can be tested with [*2], [*3], etc.

7.5.5 Functional Coverage Points Generation from SystemVerilog Assertions + FV

With CDV (Coverage Driven Verification) the concept of a huge regression suite is slowly fading away making way to "intelligent and smart" pseudo-random tests, which when run with multiple seeds, will achieve the verification goals (as defined by the FCPs). A possible flip side of this

methodology is that since the number of tests are reduced and the seeds are random, one is not quite sure whether any of the previously found (and fixed) errors don't re-appear in some other fashion. Advanced CDV recommends that new coverage points be added when an error is found (in case no other coverage point covered that scenario). Now, with the introduction of FV, it is likely that we find many corner case errors early in the design cycle, and one may wish to track such corner cases at higher levels of verification, typically performed through simulation, emulation etc. One approach to handle the tracking of critical corner cases across verification levels is to produce a coverage point specification (wherever possible) when a FV tool uncovers an error. A FV tool can be tuned to automatically output a coverage point code for the behavior under consideration, and such coverage points could be added to the regular set of FCPs. This technique will help in implementing the proposed "Verification Fragmentation" methodology, which in essence tries to employ a variety of methodologies to tackle the daunting verification problem.[17]

7.6 Temporal Debugging

Hardware debugging is an area yet to be fully researched, and most of current day debug tools borrow ideas from the software domain – Flow/Algorithm based debug. An algorithm based debugger lets the users walk through the source code (HDL in this case) and allows break points, watch dogs etc. However, mere source code based debug is insufficient in hardware systems and a better representation would be the actual logic implied by the HDL source code. A synthesized model of the design can let the users visualize the design through the schematic diagrams with the logic being interpreted as MUX, adders FFs etc. Annotating the values of various nodes within the design, as obtained through a simulation run for instance, on top of the schematics would give a comprehensive description of the state of the system under debug.[18] Such debugging is popularly known as "structure or connectivity based debugging". With the introduction of ABV, the debugging requirement would expand to a "time window" from just a "snapshot in time" as many complex assertions would span across multiple clock cycles, and hence it is important to follow the assertion from the "start" to "finish" (the "finish" could end up with FAIL or SUCCESS). This warrants a "temporal debugger" that can expand the basic structure based debug to span across a time window. An important advantage of using ABV is that it localizes the source of error, so the debugging can be focused to the few paths where the suspected errors lie. When an assertion fails, the temporal behavior can be extracted using a two-step process as:[19]

- Starting from the node of concern (e.g., the output signal where the assertion failed), build a "fan-in" cone that contains all the signals that contributed to a value change to this node. The cone starts from the node of concern and extends till every contributing signal is traced to a register or a primary input. Typically the users will be able to choose the levels of "registers" (essentially the depth of the cone).
- Annotate the values of various nodes in the cone. These values can be derived from a wave dump. The values of registers should be annotated along with the timing relative to the chosen signal's time (a.k.a reference time). During this process, the active and inactive elements in the fan-in cone can be identified, and inactive elements can be ignored for ease of debug.

---

[17] Fragmentation of the IC Verification Process. http://www.edat.com/NEA21.htm

[18] Many commercial EDA tools support this kind of sophisticated debugging environment

[19] We sincerely thank Bassam Tabbara of NOVAS for his valuable inputs to this section. We recommend the readers to refer to Bassam's paper on "Assertion based Hardware debugging" that appeared in DVCon 2003 conference. NOVAS's VERDI is a commercial temporal debugger.

Some tools can extract the "cone" of a failure and can support value annotation on to the cone. Some tools may even be able to display the "value annotation across time window".

Figure 7.7-1 Temporal debugging of an assertion from the Traffic Light example

To demonstrate the concepts explained above, the figure 7.7-1 was drawn for an assertion from the traffic light controller example. This figure was derived partly from tools and partly from the concepts explained above. Readers are recommended to check with their tool vendors for the availability of temporal debugging capabilities.

7.7 SIMULATION OR FORMAL VEF RIFICATION?

Which is better, simulation or formal verificati on? Specifically, if a design is characterized with RTL implementation assertions and is verifie d with formal verification tools, will that design operate in the system? There are mixed opinic ons on this question, as elaborated in an EE Times article *Formal tools won't replace simulation, c xperts say.*[20]

Realistically, the two are complimentary techn ologies for finding errors: formal verification and simulation. Using SystemVerilog Assertions ABV along with formal verification tools is more applicable for design and verification productiv ity. With formal verification, designs can quickly be evolved and verified, particularly at the sut oblock level where typically designers do not use subblock testbenches and rely on higher-leve l simulation for verification. The use of formal verification definitely helps in finding bugs, e ven though it is possible to eventually find those bugs in simulation. User-written assertions are also good for finding corner cases.

A testbench of the subsystem can increase the confidence of a design and quickly find any major functional problems. Formal verification will g et users to good code faster at the block level, and will help find corner cases. The more intere sting question is not "which is better, formal or simulation", but "how do I create an effect ive methodology that uses both simulation and formal?"

7.7.1 Arguments for Simulation with ABV

Industry is currently focusing exclusively o n synthesizable RTL implementation assertions. There is a concern in specifying implementati on assertions at only the RTL level because those lower-level assertions do not characterize the e nd-to-end behavior of a block or design at the sub-system level. In fact, it is possible to formall y prove all the lower-level (RTL implementation) assertions, and a block could still be broken fr om a black-box perspective. Simulation is a well-understood process, and testbenches designs g enerally address the black-box level by exposing the DUT with a deluge of interface waveform s to verify the design. RTL level assertions will detect implementation errors, while simulatior will detect higher level "system errors". Using ABV in the testbench will also help in detc ecting "system level, black-box" errors, and can identify the degree of functional coverage.

From referenced EE Times article, "What m akes formal verification difficult for ASICs are capacity limits, complex latching structures ar nd the lack of specialists on ASIC design teams". Another concern is the level of HDL and ABV support provided by the verification tool. Since this methodology is evolving and maturing, not all features of ABV and HDL (e.g., loops, configuration, generate) are necessarily fully supported by tool vendors implementing formal verification (this is vendor and version depend ent). In addition, complex data structures, such as large memories, are not well supported in form al verification. However, ABV is better supported in simulation. These restrictions surface the is sues of portability and validation (i.e., can the tool really formally prove the assertions).

[20] By Richard Goering, EE Times, June 6, 2003 (12:11 p.m. ET)
http://www.eetimes.com/story/OEG20030606S0017

7.7.2 Arguments for Formal Verification

Formal verification can verify the module's behavior against all possible inputs and input sequences. It can verify the module using corner-cases that a testbench author may not have considered. It can examine sequences that may not be reached in a necessarily finite amount of exhaustive testing.

From referenced EE Times article, "formal verification has two possible applications—finding bugs in RTL code, and gaining assurance of zero bugs prior to tapeout. What we've found at Intel, although we do find bugs, is that the real value of formal verification is the assurance". "Formal tools can find tough bugs that simulators miss. To successfully use formal tools, designers need to choose the right problem, have a clear specification, and know which formal tool fits best—equivalence checking, Model Checking, symbolic simulation or Theorem Proving".

From referenced EE Times article, "we are very close now to a time when formal is a technology everyone will be using, but they might not be aware they're using it." Harry Foster, chief methodologist at formal verification startup Jasper Design Automation, was the strongest defender of a pure formal approach. "Dynamic verification will not solve the verification challenge," he said. "Pure formal verification can go a lot further, without relying on simulation. The effort to create a quality formal testbench is no worse than the effort to create a quality simulation testbench, and the quality of results is better." What's needed now, said Foster, is design for verification, such that engineers develop code with both synthesis and verification in mind. He called for greater use of block-level verification and for proving "end-to-end behavior" as well as isolated assertions.

7.7.3 Balance

People who have written enough assertions for formal (high or low-level assertions) experienced that formal verification may require as much or more engineering input than does simulation. This is because, starting with a verification testplan, in simulation one must make sure that the testbench provides sufficient checkers to satisfy the testplan. Similarly, from a formal perspective, one must write enough assertions (possibly a one-for-one match to the simulation checkers) to satisfy the testplan. The real issue comes down to "can one verify these higher-level assertions in a formal engine?" This is the challenge, and many vendors answer YES to that question.

From referenced EE Times article, "There is no contradiction between simulation, emulation and formal verification. The challenge is to find the right combination to make them all cost effective. Formal technology is here and the business model is ready—and what's missing is the 'packaging' that will insert it into the design flow. Formal methods have been held back by a lack of standards, but Accellera's SystemVerilog and Property Specification Language will help".

7.7.4 Recommendations

ABV (e.g., with SystemVerilog Assertions) needs to be added to RTL, interface, and system levels. Formal verification tools can be used to verify subblocks and subsystems, provided properties at all levels of abstractions are incorporated into the verification environment. In the current state of the art, the authors of this book believe that formal verification would be very beneficial at the subblock design level. However, designs still need to be verified through simulation, particularly when integrated with other partitions. ABV will greatly simplify the testbench since it provides another level of verification at the white box level. When ABV is used at the higher levels of abstractions, it provides a greater degree of assurance that the design is conformant to the requirements (i.e., interfaces and operations).

How much assertions are enough to characterize a design? This question relates to the requirements and the testplan. How well are those documents written? As described in chapter 6 and in this chapter, ABV can be used at those levels, and those properties can then be migrated into the verification environment. With poor requirements and a poor testplan, it is difficult to define assertions, and neither simulation nor formal verification will do good justice to the verification process.

Separate verification teams should independently write assertions for architectural, system level, and black-box designs. However, the lower level implementation assertions should be applied inline with the RTL design (i.e., as comments) by the designers who can best document their code with assertions. This effort captures all assumptions and documents areas of concerns, which then become targets for formal and dynamic verification. Many users have experienced success using this methodology. When non-designers write assertions, they tend to miss out on the white-box aspect of the ABV methodology that requires a deep knowledge of the design. These two models are not mutually exclusive, but complimentary, with different teams using ABV differently at different points of the design cycle.

Which is better, simulation or formal verification? Specifically, if a design is characterized with RTL implementation assertions and is verified with formal verification tools, will that design operate in the system? There are mixed opinions on this question, as elaborated in an EE Times article *Formal tools won't replace simulation, experts say*.[21]

7.7.5 Validity of Formal Verification results

Finally, there is the question of validation. It has taken years for certain HDL simulators in the marketplace to achieve "sign-off" status. That status indicates that the simulator is now trusted enough that it is fully expected, by both the designer and the silicon provider. The simulation results will indeed predict exactly how the hardware will behave. The same questions are now asked of the property simulation and formal verification tools. Are the results dependable? How can one be sure that a "pass" grade from a tool vendor will in fact guarantee results in silicon that satisfy the given properties?

Even though there is no approved standard test suite for SystemVerilog Assertions, the validation of SystemVerilog Assertions for simulation is usually addressed by tool vendors who not only verify the algorithms, but also use their internal test suites to ensure accuracy. For formal verification, the issues are different. Validation of properties for formal verification is performed

[21] By Richard Goering, EE Times, June 6, 2003 (12:11 p.m. ET)
http://www.eetimes.com/story/OEG20030606S0017

with verification algorithms developed and researched over a decade by several universities and commercial companies. A verification algorithm may start assuming that each specified property that needs verification will fail, and will look for evidence that it will pass. This approach is conservative, but may generate a false negative (i.e., state that the property fails, when in reality it does not). However, it will never (or at least should not) generate a false positive (i.e., state that the property passes when in reality it is failing). Another formal verification approach is to assume that each specified property will pass, and will look for evidence that it will fail. This technique can generate false positives, but not false negatives. Tool vendors strive to avoid false results.

For properties that formal verification flags as unsatisfied or failing, the questions can be answered easily. Most tools provide counterexamples to prove the existence of a path to an illegal state in the circuit. A certain number of false-negatives is acceptable to most design methodologies. While the time inspecting a false counter-example may be considered wasted, if the tool does catch and report real problems, most design groups would be happy to pay the price. If half of all counter-examples led to real design problems, most designers would consider the tool a success. However, developers of formal verification tools strive at reducing the number of false negatives that are reported by their tool.

A more difficult issue is a positive result. How can one be assured that a "pass" represents good hardware and not a poorly formed property or an error by the verification tool? What kinds of processes will it take for formal verification tools to prove themselves in the real world? Poorly formed or stated properties are the GIGO[22] problem, and can be circumvented with good design practices, such as good documentation and design reviews. Errors in the verification tool can be minimized by using a reputable vendor, and by running testcases on the desired subset of SystemVerilog Assertions to fully understand the implications of the constructs, and to ensure tool compliance to this subset.

[22] Garbage In; Garbage Out,

8 SYSTEMVERILOG ASSERTIONS GUIDELINES

This chapter provides a rich set of guidelines in using SystemVerilog Assertions. These guidelines emerged from experience with usage of Assertion-Based Verification with Accellera's PSL, vendor's recommendations, code reviews, and SystemVerilog LRM documentation.[1]

It is recommended that when Assertion-Based Verification is used in a project, all designers and verification engineers involved in that project use that methodology. Sporadic and partial usage of assertions in a project defuses the benefits of ABV.

---

[1] *Using PSL/SUGAR for Formal and Dynamic Verification 2nd Edition*. 2004, Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, isbn 0-9705394-6-0

8.1 TYPOGRAPHIC GUIDELINES

8.1.1 Naming Convention

A naming convention provides consistency in a design project for the clarification of how objects in the design are used and classified. Users may not agree with our proposed naming convention, and may prefer an optional alternative. Our position is that as long as a consistent naming convention is explained and used across a project, the choice of the convention is not critical.

8.1.1.1 File naming

1. Each SystemVerilog file should use the ".sv" suffix to differentiate its HDL class (e.g., Verilog (.v), VHDL (.vhd)).
2. Each design name should include the unit type appended as a suffix to identify its use. The file name should match the design name. The following table summarizes the design unit naming convention. This approach helps clarify the content and intent of a file and design.

Table 8.1.1 Design Unit Naming Convention

Type of file	Suffix	Example	File name
RTL	_rtl	**module** cpu_rtl (...);	cpu_rtl.sv
Interface	_if	**interface** usb_bus_if (..); **modport** pci_master_if (..);	usb_bus_if.sv
Program	_pgm	**program** test_pgm;	test_pgm.sv
Package	_pkg	**package** cpu_pkg;	cpu_pkg.sv
Behavior	_beh	**module** cpu_beh(...);	cpu_beh.sv
Properties in - a module - an interface - program - package	_props	 **module** cpu_props(..); **interface** usb_bus_props_if (..); **program** cpu_props_pgm (..); **package** cpu_props_pkg;	 cpu_props.sv usb_bus_props_if.sv cpu_props_pgm.sv cpu_props_pkg.sv
Testbench	_tb	**module** top_tb	top_tb.sv

8.1.1.2 Object Naming

The following table summarizes the object naming convention.

Table 8.1.1.2 Object Naming Convention

Type of object	Suffix	Example
Type	_t	**typedef logic** [WIDTH-1 : 0] word_t; **typedef enum** {OFF, RED, YELLOW, GREEN} lights_t;
Interface	_if	fifo_if fifoip_if; // instantiation of FIFO bus
Active low variable	_n	**logic** reset_n // active low reset
Constant	Upper Case	**parameter** WIDTH=16; **localparam** DEPTH=256;

8.1.1.3 Naming of Assertion Constructs

A naming convention for the assertion constructs helps identify the construct, and becomes meaningful during the display of failed assertions or the access of instantiated assertions using the SystemVerilog Assertions Application Programming Interface (API). Table 8.1.1.3 provides a summary of recommended notation. Note that the underscore '_' character is optional, but recommended when the first letter of the object name is in lower case, as it enhances readability.

Table 8.1.1.3 Recommended Prefix Named Notation for Assertion Constructs

Type of object	Pre-fix	Example
sequence	**q**	**sequence** q_req; @ (**posedge** clk) **$rose**(ready) ##[0:4] req; **endsequence** : q_req **sequence** qReq; **$rose**(ready) ##[0:4] req; **endsequence** : qReq
property	**p**	**property** p_reqack; @ (**posedge** clk) **$rose**(req) \|=> ack; **endproperty** : p_reqack **property** pReqAck; **$rose**(req) |=> ack; **endproperty** : pReqAck

variable	**v**	**property** p_req;
		logic [31:0] v_data; // local variable for data
		logic [1:0] v_vie; // local variable for vie
		(**$rose**(req), v_data=data, v_vie=vie) |=>
		ack && buff=v_data && vie!=v_vie;
		endproperty : p_req

assert	**a**	ap_reqack : **assert property** (p_reqack); apReqAck : **assert property** (pReqAck); *// using default clock*
cover	**c**	cq_req : **cover property** (@ (**posedge** clk) q_req); cpReqAck : **cover property** (pReqAck); *// using default clock* cp_reqack : **cover property** (p_reqack);
assume	**m**	mp_reqack : **assume property** (p_reqack); mpReqAck : **assume property** (pReqAck); *// using default clock* mq_req : **assume property** (@ (**posedge** clk) q_req);

8.1.2 Ending Statements

End all statements with the ": end_identifier name". This applies to properties, sequences, modules, packages, etc. For example:

```
module fifo_rtl;
  sequence qReqack;
    @ (posedge clk) req ##[1:10] ack;
  endsequence : qReqack
  ...
endmodule : fifo_rtl
```

Identifier name

This convention was used throughout this book because it enhances readability.

8.1.3 Constants for Modules and Interfaces

Constants are named data variables that never change du ring the simulation, synthesis, or formal verification process. All constants should be written i n upper case. These constants can be declared with the keywords **parameter**, **localparam**, ar d **const**, all of which can be initialized during declaration. For example:

```
parameter int DELAY = 10;
localparam WIDTH = 32;
const logic READY_FLAG = 1 ;
```

A **const** form of constant differs from a **localparam** cons tant in that the **localparam** must be set during elaboration, whereas a **const** can be set during sim ulation, such as in an automatic task.

```
task automatic increment;
  const ONE=1;
  sum=a + ONE;
endtask : increment
```

ONE is set during simulation

8.2 USE MODEL GUIDELINES

8.2.1 Where to Write Properties and Assertions

Properties can be defined in several design units, as sl own in Figure 8.2.1-1. Figure 8.2.1-2 demonstrates which design units allow the assertion of properties. Chapter 3 stated that properties could be written in the following SystemVerilog declarations:

1. Compilation-unit scope, representing decla rations outside other design units (e.g., modules, packages).
2. Module, such as DUT, property definition, o r testbench.
3. Interface, representing busses and collection of ports.
4. Program, used in testbench designs
5. Clocking block, representing blocks with common clocking schemes. Clocking blocks can be defined within modules, interfaces, programs, and packages.
6. Package, representing a collection of declara tions.

Figure 8.2.1-1 Design Units where Properties could be Declared

Figure 8.2.1-2 Design Units Supporting Verification Directives

Where should properties be declared and assert ed?

The simple answer is where it is appropriate with in the scope of the design. This is elaborated below.

1. **Embedded assertions in RTL synthesiza ble code (i.e., in DUT module)**

   This technique is best suited when writte n by RTL designers to document properties of the design, and to support a white-box ver ification methodology.

2. **Property module bound to RTL synthes izable code**

   This is best suited when written by a ver ification engineer or by the RTL designer who want to separate the properties from the RTL code. The binding is done with the "bind" directive. The separation of the RTL design from the properties permits the synthesis (or other processing) of the RTL code written in Verilog with tools that not support SystemVerilog. However, that technique also permits the verification of the design with tools that support SystemVerilog.

3. **Packages**
   Packages offer the advatage of reusability . Only declarations are allowed in packages.

4. **Embedded assertion in testbench ve rification or system requirement modules.**
   This is best suited when a separate engi neer writes the system level requirements or the testbench for verification of system leve l properties. A separate module can also act as verification IP for the DUT.

5. **Interfaces**
   Defining and asserting properties in inter faces creates a close bond between the interfaces and the characteristics along with verific ation of those interfaces in application code. A recommended style is to also define with in the interface, tasks that are related to interface operations, such as READ and WRITE.

8.2.2 Assertions for Accuracy

8.2.2.1 Abide by Good Verilog Coding Sty le Rules

SystemVerilog is a rich set of extensions to Ver ilog 2001, and hence basic coding guidelines for Verilog hold for SystemVerilog based designs as well. For instance, the guidelines for proper usage of blocking and nonblocking assi gnments are still essential. Knowledgeable Verilog/SystemVerilog instructors teach these g uidelines. An excellent source for why, how, and where to use blocking and nonblocking assignm ents is available at the footnoted link.[2]

8.2.2.2 Avoid Nested System Functions

Avoid nested assertion system tasks because the y may represent difficulties in interpretation, such as in multi-clock cases. An example of an illeg al nested assertion is "$past($rose (variable))". Section 8.3.2.1.1 addresses an alternative to the use of these nested system tasks.

[2] http://www.sunburst-design.com/papers/ Cliff Cu mmings
Verilog Nonblocking Assignments With Delays. M yths & Mysteries

8.2.2.3 Beware of unsized additions in +1 versus +1'b1, Use Size Casting if Necessary

Since Verilog is loosely-typed, it allows un-sized expressions in addition operation. This can lead to unexpected errors if not used carefully. This problem exists both in design and assertion constructs. The following pseudo-code highlights one such example:

```
logic clk;
logic [3:0] count, prevcount;
always @ (posedge clk) begin
  if (count == (prevcount + 1)     $display("case 1");
  if (count == (prevcount + 1'b1) $display("case 2");
  if (count == 4'(prevcount + 1)  $display("case 3");
end
```

Consider the case where count ==4'b0 and prevcount==4'b1111.

Case	Expression	Value	Comparison Result Count==Expression
1	(prevcount + 1)	5'b10000	FALSE
2	(prevcount + 1'b1)	4'b0000	TRUE
3	4'(prevcount + 1)	4'b0000	TRUE

Analysis

The expression *(prevcount+1)* is evaluated as follows:

- 1 is a 32-bit integer
- *prevcount* is a 4-bit value, by definition
- Addition result expression is a 32-bit value because "1" is an integer.
  For example: if count is 4'b1111, *prevcount* + 1 is evaluated as follows:

```
count                                   1111
Integer 1          00000000000000000000000000000001
-------------------------------------------------
Expression         00000000000000000000000000010000
(32-bit integer)
```

However, The expression *(prevcount + 1'b1)* is evaluated as follows:

- 1'b1 is a 1-bit value
- *prevcount* is a 4-bit value, by definition
- Addition result expression is a 4-bit value because addition is between values that are a 4-bit and a 1-bit in widths. For example: if *prevcount* is 4'b1111
  count+ 1 is evaluated as follows:

```
count                    1111
1'b1                        1
-----------------------------
expression               0000
```

When *(prevcount +1)* is assigned to a 4-bit variable, it gets truncated to the least 4-bit value. However, when performing comparisons, the resulting expression *(prevcount +1)* retains its 32-bit results. In the example model:

These types of comparisons are often performed in the definition of properties as either conditions to tests as antecedent, or conditions to verify as consequents. Figure 8.2.2.2 demonstrates a simple assertion test case where such comparisons can occur, and different results are obtained. Note that the style of assertions in this example resembles RTL code, and is generally not recommended. However, the goal of this model is to demonstrate the issue.

```
module bit_width;
 timeunit 1ns;
 timeprecision 100ps;
 logic clk = 0;
 logic load = 1'b1;
 logic [3:0] count = 4'h0;
 logic [3:0] prevcount;
 wire eq_ok1, eq_ok2, eq32;
 initial forever #20 clk = ~clk;

 property OK1; @ (posedge clk)
   load |=>  count == $past(count) + 1'b1; endproperty : OK1

 property OK2; @ (posedge clk)
  load |=>  count == 4'($past(count) + 1); endproperty : OK2

 property BAD;
  @ (posedge clk)  load |=>  count == $past(count) + 1;  endproperty : BAD // ☹

 OK1_1 : assert property (OK1);
 OK2_1 : assert property (OK2);
 BAD_1 : assert property (BAD);

 always @(posedge clk)  begin
    prevcount <= count;
    count <= count + 1;
 end
 assign eq_ok1  = (count == (prevcount + 1'b1));
 assign eq_ok2  = (count == 4'(prevcount + 1));
 assign eq32 = (count == (prevcount + 1));
endmodule : bit_width
```

Different results

Figure 8.2.2.2 Assertion Model with Correct and Incorrect use of Comparisons

A simulation result of this model revealed the following error:

```
"bit_width.sv", 26: bit_width.BAD_1: started at 620000ps failed at 660000ps
        Offending '(count == ($past(count) + 1))'
```

8.2.2.4 Beware of Property Negation Operator

Apply the "**not**" negation operator to a property that is a Boolean expressions or a sequence. Do not apply it to a property that includes an implication operator (e.g., |->, |=>). Though allowed in the language, the negation property is more likely to create unexpected failures. It is a common requirement in designs to verify that "sequence B" does NOT follow "sequence A". A novice user might be tempted to use the negation operator to express this behavior. Such a usage might lead to unexpected failures. For instance, consider the following example:

Example	Property Example	
1	**property P; not** (!a && b); **endproperty** : P	✓
2	**property Y; not** (a ##[1:5] b); **endproperty** : Y	✓
3	**property** BAD; **not** (a ##1 b |-> c ##1 d); **endproperty** : BAD	💣
4	**property** GOOD; **not** (a ##1 b ##0 c ##1 d); **endproperty** : GOOD	✓
5	**property** OK2; a ##1 b |-> c ##1 d |-> `false; **endproperty** : OK2	✓

If the "antecedent" sequence in property BAD were not to hold true (e.g. @ t0 a = 1, @t1 b = 0), then the sequence expression "a ##1 b |-> c; d" succeeds vacuously, but the negation operator surrounding the sequence will report a failure. If a property is required to specify that an antecedent followed by a consequent should never occur, either create the property expression as a simple sequence (see example 4 above) or follow the consequent with the overlapping operator "|->" followed by "`false"(see example 5 and property *pReqAckNoError2* below). Another alternative is to express the expected behavior, instead of the erroneous one. For example, to express that no *error_flag* (i.e., *!error_flag*) is expected when a *request* is followed by an *acknowledge* and a transmission of 128 words, the property can be written in one of the following ways:

Expected behavior

```
property pReqAckNoErrorl;   @ (posedge clk)
    request |=> acknowledge ##1 data_enb[*128] ##1 !error_flag;
endproperty : pReqAckNoError1
```

Description of a failed behavior

However, if the error flag is TRUE, this is an erroneous

```
property pReqAckNoError2;   @ (posedge clk)
    request |=> acknowledge ##1 data_enb[*128] ##1 error_flag |-> `false;
endproperty : pReqAckNoError2
```

Notes:

1. Property *pReqAckNoErrorl* can complete successfully. However, property *pReqAckNoError2* can either succeed vacuously or fail.
2. Since the **not** operator applies only to properties, it cannot be applied to antecedents. Antecedents must be sequences, but consequent are properties (which encompass sequences). Per LRM: Property expression is (among other things):
   not property_expr
   | sequence_expr |-> property_expr
   | sequence_expr |=> property_expr

8.2.2.5 Ensure "Write before Read" while using Local Assertion Variables

Some properties can create unsafe references to variables. Properties described using sequence composition operators (such as **or**, **intersect**) can start their evaluation under multiple (independent) conditions. When a local assertion variable is being used in such properties, make sure that the variable is updated in all possible starting points. For example:

```
property p_unsafe1;
 int v_data;
 @ (posedge clk)
   ((req, v_data=data) or go)  ##1 ack && xbus==v_data;
 endproperty : p_unsafe1
```

Unsafe because if *go* without *req*, then *v_data* will not be assigned a value.

```
property p_unsafe2;
 int v_data;
 @ (posedge clk)
   ##[0:$](req, v_data=data1) intersect  ##[0:$] (ready, v_data=data2)
                                          ##[1:$] ack && xbus==v_data;
 endproperty : p_unsafe2
```

Unsafe because *v_data* may have the value of *data1* or *data2* depending upon the cycle in which *req* or *ready* occurs. The comparison of *xbus* to *v_data* is not deterministic.

Those properties can be made safe as follows:

```
property p_safe1;
 int vdata;
 @ (posedge clk)
   ((req, vdata=data) or (go, vdata=data))  ##1 ack && xbus==vdata;
 endproperty : p_safe1

property p_safe2;
 int vdata1, vdata2;
 @ (posedge clk)
   ##[0:$](req, vdata1=data1) intersect
        ##[0:$] (ready, vdata2=data2) ##[1:$] ack && xbus==vdata1 && ybus== vdata2;
 endproperty : p_safe2
```

8.2.2.6 Be Aware of Overlapping Assertions

Overlapping assertions can yield unexpected errors. Use **$rose** (signal) when the property is dependent on the rising edge of that signal.

$rose**(req) \|-> ##[1:5] ##1 ack; *The **$rose of req guarantees a single trigger for the verification of the RHS, even if req was held asserted for multiple cycles.*	👍👍
(req) |-> ##[1:5] ##1 ack; *For every cycle where req is asserted, the RHS is triggered for verification.* *A single occurrence of ack may satisfy multiple overlapped assertions.*	👎👎

If the antecedent has a range, use the **first_match** of the antecedent sequence to avoid unexpected results due to multiple matches in the antecedent.

8.2.2.7 Beware of Metalogical Values

Like Verilog, SystemVerilog defines **False** as the single bit value 1'b0, 1'bx, 1'bz. **True** is defined as the single bit value 1'b1. The negation of a metalogical value (x, z) is a metalogical value. Considering the property expression "a |-> ! b", the following should hold in SystemVerilog Assertions:

a	b	a |-> ! b	Comments
1	0	**PASS**	
1	1	**FAIL**	
1	x, z	**FAIL**	Meta-logical

If a property relies on the value of Z or X, then use the *case equality* and *case inequality* operators(=== and !==) instead of the *logical equality* and *logical inequality* operators (== and !=).

8.2.2.8 Avoid Vacuous Properties

A property is vacuous if within the correctly reachable state space, the antecedent is never true at some point during execution. This can happen if the antecedent is set to FALSE as a result of a constant, or if a disable clause forbids the antecedent from ever being TRUE. For example:

```
property p_nsLightAtReset;
  @ (posedge clk) disable iff (!reset_n)
    !reset_n |=> ns_light==OFF; // "disable iff" forbids antecedent from becoming TRUE
endproperty : p_nsLightAtReset
```

NOTE: This is slightly different from "vacuous success" of certain evaluations of a property. In these examples, the property is vacuous by "construction" and hence should be avoided.

8.2.2.9 Avoid Contraditory Properties

Avoid properties that are contradictory. For example:

```
property pStopOnRed;
  not (light== RED && drive_thru); // Can never go thru on RED
endproperty : pStopOnRed

property pGoOnOverride;
  light==RED && override |-> drive_thru; // Can go thru on RED with override
endproperty : pGoOnOverride
```

Those two properties are contradictory because *pGoOnOverride* allows for a condition *drive_thru* to be TRUE when *light==RED*. However, *pStopOnRed* disallows the condition where both *light==RED* and *drive_thru* are both TRUE.

In practicality, a slight change in specification during a later part of design cycle might introduce such conflicting requirements. To avoid the spurious errors in such cases, a user may lower the assertion severity levels for some assertions. For example:

```
ap_pStopOnRed : assert property (pStopOnRed) else
                $warning("drive_thru is TRUE when ns_light is RED");
```

From a verification standpoint, it is important to obtain coverage information about certain properties. If a tool does not automatically provide coverage for properties, then use *cover* verification directive with such properties to obtain these statistics.

8.2.3 Use $sampled Function in Action Block to Display Values of Current Variables

The action block is executed immediately after the evaluation of the **assert** expression in the reactive region of the scheduling slot.[3] This has significance in the display of variables in the current time slot since those variables may have changed values in the active region, prior to the reactive region. The use of **$sampled** function accesses the values of sampled variables used in the evaluation of the properties.

```
apReqAck : assert property (pReqAck(request, acknowledge))
   else $display("time= %t, $past(req)= %b, ack=%b \n",
        $time, $past(request), $sampled (acknowledge) );
```

8.2.4 Accessing Local Variables in Assertions

While the local assertion variables add good expressiveness to assertions, they also slightly complicate the debug of such assertions. The issue is that each thread maintains its own copy and the local assertion variables cannot be easily displayed in a waveform viewer or displayed using common print tasks. The real issue lies with the fact that local variables can't be accessed from outside the property. Specifically, the LRM states, "Hierarchical references to a local variable are not allowed". However, a solution would be to display the value of the local variables within a property, which upon a condition that represents a failure displays the information. For example:

```
property pDisplayReqAckError;
 int vdata;
  @(posedge clk)
   (req, v_data=data) |=> ##[1:5] ack && vdata != data
     |-> (`true, $display($time,"ack = %b data = %0h expected_data = %0h\n",
                 ack, data, v_data));
endproperty : pDisplayReqAckError
apDisplayReqAckError : assert property (pDisplayReqAckError);
```

Users need to be aware that these debug assertions represent an "additional overhead", and hence should be used with caution. One can use compiler directives to hide them by default and enable them in debug mode.

8.2.5 Style

This section addresses guidelines in the definition of properties and assertions. A general guideline in the use of assertions is to ensure a common feel and look throughout the design. This includes among other things, naming convention, directory structure, error reporting, initialization, synchronization or properties, use of default clock, defining resets in properties, the declaration of properties and tasks in packages, interfaces, and/or modules. A consistency in the verification environment will speed up the design and debug process.

8.2.5.1 Avoid Unbounded Ranges

An unbounded range is the use of '$' in a range, such as "##[0:$]", "a [*1:$]". The guideline is to avoid, if possible, such unbounded ranges because it impacts simulation performance. In formal verification a property may be proved to be "completely" satisfied with such unbounded assertions (i.e., when all possible states in a state space domain are visited). But in simulation, it

[3] See section 5.10 on SystemVerilog scheduling semantics for assertions. Also see section 5.8.1.1.1 on the use of **$sampled**.

is the stimulus that drives the assertion checking process, and hence the "completeness" of the unbounded boundary is not visible to a simulator. This leads to a problem with such properties when used with simulators. Some simulators treat the "end of simulation" as the final bound and report any active, unfinished assertion with unbounded range as failures.[4] Also, in a good design and a well architected verification environment, many properties should be bound to a maximum range (there could be few exceptions such as a user's interaction to a vending machine controller). Hence a recommended style is to define a constant as the maximum limit. A verification engineer can tune this constant until a maximum limit is satisfied.[5] For example,

```
parameter MAX_ACK_CYCLES =100;
property pReqAck;
 @ (posedge clk) req |=> [0 : MAX_ACK_CYCLES] ack;
endproperty : pReqAck
```

8.2.5.2 Use of Default Clock

In synchronous designs with one clock, the clocking block allows for the definition of a default clock, thus eliminating the need to always add the clocking event in every property. The general recommendation for a design with a single clock domain is to either define the property declarations within clocking blocks, or to use a default clocking statement (see 3.2.5.1). However, when dealing with multiple clocks, explicitly specify the clocks within their expressions. These declarations must be outside clocking blocks (see section 5.9).

8.2.5.3 Evaluate Assertion Relative to a Clock

Combinational assertions, which are not checked at a clock edge, are illegal in SystemVerilog. Use immediate assertions instead. For example:

```
// unclocked property
property pNeverAckWithNoReady;
    not (ack && !ready);
endproperty : pNeverAckWithNoReady

always @ (*) begin // Combinational block with no clocks
  ack = req && ready;
  apNeverAckWithNoReadyERROR : assert property (pNeverAckWithNoReady);  // 💣
  apNeverAckWithNoReadyOK : assert (not (ack && !ready)); // immediate assertion ☺
end
```

No clock edge

8.2.5.4 Handling Resets in Properties

Most of the properties describe the design behavior under "normal working conditions" – i.e., when reset is inactive (except for a few tests that check for proper reset sequence, default values after reset etc.). SystemVerilog Assertions provide a way to disable the assertion checking through **disable iff** construct (Refer to section 3.2.5.2). However, this would mean that most of the properties would need to include this construct – a process that users find very cumbersome, and if the reset sequence were to change even slightly, all such properties will require modifications

---

[4] VCS supports "-assert verbose" flag to enable such reporting.

[5] Refer to "Assertion refinement process" in Section 5.4

A possible solution is to use the system functions **$asserton**, **$assertoff**, and **$assertkill** to control the checking of assertions. Those functions are explained in section 5.8.4 (Assertion-Control System Tasks), but are further demonstrated with a guideline in this section.

The use of assertion-control system tasks provide the following advantages:

1. They allow for less code in the property definitions because the **disable iff** clause is not necessary.
2. They allow long reset cycles, possibly handled though a scanning approach, instead of a simple asynchronous or synchronous reset signal.
3. Any modifications to the reset sequence can be handled very easily.

Since the assertion-control is typically performed by the testbench at initialization on all designs under test, the recommendation is to use a separate file that defines a module to specify and initiate the assertion-controls. This file is compiled along with the design and testbench. An example of this assertion-control module is shown in Figure 8.2.5.4.

```
module assert_control ();
  timeunit 1ns;
  timeprecision 100ps;
  initial begin : disable_assertions_during_reset
  $display ("%0t %m Disabling assertions during init..", $time);
  $assertoff (0, fifo_tb.fifo_rtl_1);
  @ (fifo_tb.reset_n === 1'b1);
    #1; // To avoid any race conditions
    $display ("%0t %m Enabling assertions after init..", $time);
    $asserton (0, fifo_tb.fifo_rtl_1);
  end
endmodule : assert_control
```

Figure 8.2.5.4 Example of an Assertion-Control Module

8.2.5.5 Defining Time Unit and Time Format Specifications for Design[6]

It is the designer's responsibility to avoid--or at least control—file order dependencies when using the Verilog *`timescale* compiler directive. File order dependency results from the fact that Verilog source code compilers read in multiple files as if they are one long stream of text. A file that does not contain a *`timescale* directive is dependent on the order of the files read in previously in the source stream. Compiling the same design twice, with the files listed in a different order, can result in very different simulation results.

The new SystemVerilog **timeunit** and **timeprecision** declarations remove the ambiguity of compilation file order. This is done by making the declaration of time units and time precision part of the language, rather than being a directive to the compiler. The **timeunit** and **timeprecision** can—and should--be declared inside each module, interface or test program in the SystemVerilog source code. The declarations must come immediately after the port list, before any other declarations in the model. For example,

---

[6] Guidelines provided by permission from Stuart Sutherland, Sutherland HDL, Inc., Training Engineers to be Verilog and SystemVerilog Wizards.
http://www.sutherland-hdl.com

```
module my_module (input wire [31:0] a,
          input wire      clk,
          output logic [31:0] b);

 timeunit 1ns;
 timeprecision 100ps;

 logic k; // local variable used inside this module

 property pTest; @ (posedge clk) k |=>a==b; endproperty : pTest

 apTest  : assert property (pTest) else
          $display ("%0t %m a=, b= %0h ", $time, a, b);

 initial $timeformat(-9, 2, " ns",12); // ns, 2 digit precision, ns display, 12 bits total

 intial begin
  #1000; // wait for 1000 timeunits.  In this case, wait for 1000 ns
  @ (posedge clk)
     k <= 1'b1;
     b <= 32'hFF00AABB;
  @ (posedge clk) k <= 1'b0;
 end

 ... // additional code with the model functionality
endmodule : my_module
```

SystemVerilog also has packages, which provide a mechanism for declarations that can be shared my multiple modules, interfaces or test programs. The **timeunit** and **timeprecision** declarations can be made inside a package, and that package can be imported in each module, interface or test program. Using packages is a convenient way to ensure that all design blocks in all of, or a section of, a design use the same time units and precision.

```
package common_declarations_pkg;
 timeunit 1ns;
 timeprecision 100ps;
 typedef enum {OFF, RED, YELLOW, GREEN, PRE_GREEN} lights_t;
 ... // additional declarations shared by multiple design blocks
endpackage : common_declarations_pkg

module my_module2 (input wire [31:0] a,
          input wire      clk,
          output logic [31:0] b);
 import common_declarations_pkg ::*; // import declarations from the package
                                     <-- Seen here
 lights_t lights;
 logic k; // local variable used inside this module
 property pTest; @ (posedge clk) (lights==YELLOW) |=>a==b; endproperty : pTest
 apTest  : assert property (pTest);
 ... // additional code with the model functionality
endmodule : my_module2
```

SystemVerilog also defines an implicit package, called **$unit**. Any declarations made outside of modules, interfaces, test programs, or packages are considered to be part of the implicit **$unit** package. This implicit package is implicitly imported into modules, interfaces and test programs that are compiled together with the external declarations. A **timeunit** or **timeprecision** declaration made external to a module, interface or test program can be used to define the time units and precision of several modules, interfaces or test programs.

SystemVerilog defines a precedence for determining what time unit and precision are applied to a module, interface or test program. The precedence rules ensure full backward compatibility with existing Verilog code that uses the old *`timescale* compiler directive. The **timeunit** and **timeprecision** declarations ***inside*** a module, interface or test program have the highest precedence. Next, a **timeunit** and **timeprecision** statement declared in a package that has been imported into the module, interface or test program is second in the precedence order. If not defined inside the module or imported into the module (or interface or test program), then the last *`timescale* directive that was read in the source code text stream is used. If no *`timescale* has been defined up to that point in the compilation stream, then a **timeunit** and **timeprecision** declaration in the **$unit** implicit package will be used. Finally, if no **timeunit** or **timeprecision** has been defined at all, a software tool may either apply its own default **timeunit** and **timeprecision**, or the tool may consider the time value to be a simple integer number that does not represent any number of time units.

It is legal to have an entire design with no time units or precision specified any where, but it is an error to have a mix of modules, interfaces or test programs where some blocks have time units specified and some have no time units specified. The latter situation can occur if some modules, interfaces or test programs were read in by the compiler before any time units were specified, and then the compiler encounters a time unit specification (either a **timeunit** keyword or a *`timescale* directive).

These precedence rules ensure that older Verilog code that uses the *`timescale* directive can be simulated together with SystemVerilog code that uses the new **timeunit** and **timeprecision** keywords. It is important to understand that preserving backward compatibility with older Verilog code also means that the ambiguity of file order dependency is preserved. There is no ambiguity when the new SystemVerilog **timeunit** and **timeprecision** declarations are made ***inside*** a module, interface or test program, or if the **timeunit** and **timeprecision** declarations are imported from a named package. The time units and precision have been explicitly defined for that design block, and the order in which source code is read in has no effect. The time units and precisions of following example, however, will be ambiguous, even though it appears that the time units and precision are explicitly specified.

```
timeunit 1ns;
timeprecision 100ps;
module my_module_bad_timeunit_use (input wire [31:0] a,
        input wire      clk,
        output logic [31:0] b);
 logic k; // local variable used inside this module
 property pTest; @ (posedge clk) k |=>a==b; endproperty : pTest
 apTest : assert property (pTest);
```

```
intial begin
 #1000; // wait for 1000 timeunits.  In this case, wait for 100 ns
  @ (posedge clk)
    k <= 1'b1;
   b <= 32'hFF00AABB;
   @ (posedge clk) k <= 1'b0;
end
```

A previous definition of `timescale 10 ns / 1 ns has higher precedence than the **timeunit/timeprecision** declared in the **$unit** implicit package, thus altering time unit and precision for this module.

```
... // additional code with the model functionality
endmodule : my_module_bad_timeunit_use
```

In this example, the **timeunit** and **timeprecision** declarations come before the module, instead of ***inside*** the module. This places these declarations in the **$unit** implicit package. As noted in the precedence order described earlier, the *`timescale* directive has a higher precedence than **timeunit/timeprecision** declarations in the **$unit** implicit package space. Therefore, any files read in by a compiler as part of its source code text stream could alter the time units and precision of this module.

To avoid the ambiguity of compilation file order, designers and verification engineers should ensure that every module, interface or test program either explicitly declare the time units and precision within that design block, or explicitly import a package containing the **timeunit** and **timeprecision** declarations.

Note: In this book, most examples do not show the **timeunit / timeprecision** constructs because the emphasis is on assertions. However, the downloadable examples include those constructs.

8.2.5.6 Direct or Implicit Declaration of Properties

SystemVerilog allows both named properties being asserted using a verification directive and a property specification inline with a verification directive (such as **assert**). A question that is often asked is whether it is best to first declare properties and then instantiate them, or simply define the property within the instantiation with the verification directive (unnamed property expression).

Arguments FOR declaring properties before instantiation

1. Reusability
   a. When declared with formal arguments, the named properties are reusable with different actual arguments.
   b. Named properties can be reused with other verification directives (i.e., **assert, assume, cover**) and with the **expect** directive.
2. Documentation
   a. Named properties can be declared in requirement and verification documents, where assertions are not required, and properties are defined.

3. Separation of declaration and instantiation
   a. Named properties can be declared in modules, packages, interfaces, and in the compilation unit scope, and can then be asserted in modules or interfaces.
   b. Named properties can be placed in clocking blocks, and asserted in modules or interfaces.

```
//Example: Named property
property pReqAck (req, ack) ; req |=> ##[0:5] ack; endproperty : pReqAck;
apReqAck : assert property (apReqAck(request, acknowledge));
```

Arguments FOR direct instantiation of properties with no declarations

1. Less code since declaration is not needed
2. Useful for non-reusable assertions, typically used once.
3. More suitable for inline assertions added while doing RTL coding.

```
// Example: Direct instantiation of property with no declaration
aReqAck2: assert property (req |=> ##[0:5] ack) ;
```

In this book, we used the explicit declaration of properties in all our examples because we feel that it is generally a better coding style.

8.2.5.7 Use Formal Arguments only when Reuse is Intended

Not every property needs to be declared with formal arguments since it may never be reused, and is always applicable to a specific module or protocol. Declaring with formal arguments adds more code since it requires formal arguments in the definitions and an instantiations of the property/sequence/endpoint. However, if re-use of the sequences or properties is desired, then it is worth the effort.

8.2.5.8 Use module ports or Registered Signals in Properties

When writing properties, use ports and registered signals (i.e., inputs/outputs of design and output of registers), instead of signals that represent combinational logic. This facilitates the reuse of properties during gate-level simulation because synthesis generally maintains those names in a recognizable manner. Combinational signals tend to get optimized away, while most of the sequential signals are retained (though often renamed through synthesis naming rules). If the assertions are to be reused in Gate Level simulation, then it is a good idea to have the signal names defined using text macro than the actual name of the signal. For example:

```
`define REQ_SIG top.dut.controller_1.arbiter_1.req
`define ACK_SIG top.dut.controller_1.arbiter_1.ack
`define CLK_SIG top.dut.controller_1.arbiter_1.clk

property p_req2ack;
 @ (posedge `CLK_SIG) `REQ_SIG |=> [1:5] `ACK_SIG;
endproperty : p_req2ack
```

It is also recommended to keep the macro definitions in a separate file, such as "*assertion_defines.sv*". During gate level simulation, the properties can be reused without any modification by simply replacing the RTL "*assertion_defines.sv*" with an equivalent gate level file. An example gate level "*assertion_defines.sv*" file using escaped signal names is shown below:

```
`define REQ_SIG top.\dut/controller_1/arbiter_1 .req
`define ACK_SIG top.\dut/controller_1/arbiter_1 .ack
`define CLK_SIG top.\dut/controller_1/arbiter_1 .clk
```

8.2.5.9 Standardize Action Block Error Display

Standardize the display and collection of total violations using a task in a package. The action block provides the task call. For example:

```
package action_pkg;
  timeunit 1ns;
  timeprecision 100ps;
  int total_sva_violations = 0;
  task report_sva_violation (string user_defined_err_msg );
    $display ("%0t SVA_VIOLATION: %m %s", $time, user_defined_err_msg);
    total_sva_violations ++; // A global status counter to count number of SVA violations.
  endtask : report_sva_violation
endpackage : action_pkg

module m;
  import action_pkg :: *;   // Import of package is visible by module only
  // code and property declarations
  logic reset_n, clk, push1, pop1, fifofull, fifocount_err;
  property pPushNoPop_FifoCount(rst_n, clk, push, pop, fifofull);
          @ (posedge clk) disable iff (!rst_n)
           not (push && !pop && fifofull);
  endproperty : pPushNoPop_FifoCount

  apPushNoPop_FifoCount: assert property
      (pPushNoPop_FifoCount(reset_n, clk, push1, pop1, fifofull))
  else
   begin
     report_sva_violation("Fifo push & no pop error");
     fifocount_err <= 1'b1; // This can cause the testbench to take a reactive action
   end
endmodule : m
```

8.2.5.10 Use generate Construct for Assertions Conditional on Parameters

When using a parameter to choose between different properties that are mutually exclusive, use the **generate** construct to select the desired properties. Avoid using the parameter to create a vacuous properties as a differentiator. For example,

```
parameter WIDTH = 16;
module e_props (  // verification module
   input logic[WIDTH-1:0] q,
   input logic[WIDTH-1:0] d,
   input logic load,
   input logic reset_n,
   input logic clk,
   input logic[WIDTH-1:0] d_r );
 timeunit 1ns;
 timeprecision 100ps;
 parameter MODEL_ID=1; // ID==1 then model with architecture RTL1
                 // else ID==2 then model with architecture RTL2
```

```
  // Example of poor use of property definition to create a vacuous assertion
  // when MODEL_ID == 2
  // property pD_r1; // for architecture RTL1
  //   @ (posedge clk) disable iff (!reset_n)
  //     MODEL_ID==1 |=> d_r== $past(d);
  // endproperty : pD_r1
  // apD_r1 : assert property (pD_r1);
generate begin : gen_conditional_assert
  if (MODEL_ID==1) begin
    property pD_r1;
      @ (posedge clk) disable iff (!reset_n)
        d_r== $past(d);
    endproperty : pD_r1
    apD_r1 : assert property (pD_r1);
  end // if (MODEL_ID==1)
  else begin
    property pD_r2;
      @ (posedge clk) disable iff (!reset_n)
        load |=> d_r== $past(d);
    endproperty : pD_r2
    apD_r2 : assert property (pD_r2);
  end // else
end : gen_conditional_assert
endgenerate
endmodule : e_props
```

☹ Poor style because of possibility of vacuity. The **generate** creates only needed assertions instead.

8.2.5.11 Use Pattern Format in Documenting Assertions

It is recommended that the assertion description uses the "patterns format" described in the book *Assertion-Based Design*[7] to document design insights, assumptions and decisions. In this context, a pattern is "a system for documenting (in a consistent form) and describing commonly occurring assertions found in today's RTL designs". An example of this format is shown in Section 2.1.6.

8.2.5.12 Review Properties and Assertions Against Requirements

Review of the properties and assertions is very beneficial at all stages of the design because the review process often reveals deficiencies in the understanding of the requirements, properties, and design.

8.2.5.13 Simulate Design

Simulation with assertions is highly recommended as a guideline for the following reasons:

1. Simulation offers another level of assurance that the design meets the requirements. In other words, a user should not solely rely on formal verification because a user may miss the "complete" set of assertions required to formally prove the design under all conditions. However, formal verification adds a level of confidence about design compliance. Thus, the two methods are complementary.

---

[7] Assertion-Based Design, Harry Foster, Adam Krolnik, David Lacey, 2003, Kluwer Academic Publishers, ISBN 1-4020-7498-0

2. Simulation allows the use of complex data structures for verification, which are not tolerable in formal verification.[8]
3. Simulation allows the use of a greater set of SystemVerilog features than those supported by formal verification. This will change in time, as tools mature.

8.2.5.14 Guidelines for Debugging Assertions

The following guidelines help in affirming that the SystemVerilog aware tools recognize properties:

1. **Understand tool capabilities:** Learn and use simulator's assertion debugging capabilities to make sure the sequences are properly coded and are covered. Some tools provide an assertion browser for assertion statistics (e.g., pass and fail count). In addition, the waveform viewer may display the status of assertions as "Started, finished, failed" etc.
2. **Add messages and signal information in action blocks:** The action block can provide a wealth of useful information upon a failure. This can include a text message, along with values of signals that could have caused the failure.[9]
3. **Use constraints to avoid false negatives:** Practice usage of **assume** directives in the description to avoid false negatives. Note that tools may use different algorithms and methods to constrain the design, and users need to understand the tool restrictions

8.2.6 Using SystemVerilog assertions with Verilog RTL

A question that often comes up is how to use SystemVerilog assertions with Verilog RTL code. Users understand that the addition of assertions means that the tool handling the RTL code is a SystemVerilog tool. However, not all tools support SystemVerilog in a similar time frame. Users may desire to maintain the current Verilog design flow with older Verilog design tools that do not recognize SystemVerilog assertions. But they also may want to provide the assertion capability for use with other tools, such as simulation or formal verification. The following provide potential solutions:

1. **Binding**. With this scheme, the assertions are in separate modules, and the "bind" SystemVerilog directive binds the property modules to the RTL modules. This approach offers a clear separation between the RTL modules and the assertion modules. However, because of the separation between the RTL and the assertions, this approach does not directly, inline with the RTL code, inform the reader of the assumptions, and expectations the code makes. Hiding the assertions through a binding directive does not help the reader or developer.
2. **Complier Directives** The **`ifdef** and **`ifndef** compiler directives are defined in the Verilog standard (IEEE Std 1364-2001). They can be used to enable or disable a block of property and verification directives within the RTL code. Specifically, the **`ifdef** compiler directive checks for the existence of a macro name that was defined using `define. If the macro name is defined, then the lines following the **`ifdef** directive are included. If the macro name is not defined and an **else** directive exists, then this source is compiled". In his book, *Assertion-Based Design*[7], Adam Krolnik recommends from *Principles of Verifiable RTL*[i] the use of **`ifdef** ASSERT_ON to allow one to make the assertions visible or hidden based on the tool being used.

---

[8] For examples, see Section 9.15, 9.16 on the use of associative array for data integrity verification, and section 6.3.1 (subsection 5.1) on the use of queues in the definition of a FIFO.

[9] Be careful to use **$sampled** to read the correct value of signals at the display cycle.

3. **Pragma**. A pragma is a directive to the tool to accept or reject a line or block of code. It is similar to the `ifdef, except that it does not rely on the definition of a macro name. Historically, pragmas are vendor specific, and can lead to interoperability problems. One of the SystemVerilog goals is to get rid of the bad coding practice of using pragmas. Thus, if tools and models are to be compliant with the SystemVerilog standard, they need to recognize SystemVerilog Assertions.

4. **Generate.** A generate statement can be used to allow selective instantiation of assertions. For example:

```
generate
  if (ENABLE_XBUS_ASSERTIONS) begin
    apX1: assert property (pX1);
    apX2: assert property (pX2);
  end
endgenerate
```

From the above potential solutions, the viable approaches are the binding and/or the use of **ifdef** directive, depending whether the user wants the assertions in a separate file, or inline with the code. We, the authors of this book, recommend the binding method for use of assertions with legacy code where the RTL cannot be modified. For new designs, we recommend the use of the **ifdef** compiler directive.

8.2.7 Using Dynamic Data Types inside Properties

SystemVerilog has dynamic data objects such as classes, queues, dynamic arrays etc. However, LRM 17.4.1 explicitly prevents the use of the following data types in properties:

- non-integer types (shortreal, real and realtime)
- string, event, chandle, class
- associative arrays
- dynamic arrays and queues

The significance of this limitation is that such objects should be avoided inside properties or sequences. For example, the following is illegal:

```
int dataQ [$]; // queue, to store/read incoming data
// Never a READ with no data received
property p_never_read_on_empty;
  @(posedge clk_rcv25)
    not (dataQ.size == 0 && rd_25);
endproperty : p_never_read_on_empty
```

A solution is to declare a temporary placeholder object of static data type and use that inside the properties. For the above example, the following can be expressed:

```
int dataQsize; // queue size.
assign dataQsize= dataQ.size;
property p_never_read_on_empty;
  @(posedge clk_rcv25)
    not (dataQsize == 0 && rd_25);
endproperty : p_never_read_on_empty
```

8.3 METHODOLOGY GUIDELINES

8.3.1 Identifying Properties from Design Specifications

The success of assertion-based verification methodology relies heavily on the quality of properties describing the intended behavior of the design. Since it is fairly a new methodology, verification and design engineers are often faced with a question of "how to identify good properties" for a design. It may be tempting to write properties that closely resemble the implementation. For example, "if there is a *dma_load* then the data available in signal *data_in* will be latched into a register *reg_out*". Experience with users has shown that such properties are indeed "useless" from a verification perspective, even though they are real "properties" of the design. An example of a useless property is shown below:

```
property p_useless_reg_load;
  @ (posedge clk) dma_load |=> reg_out == $past(data_in);
endproperty : p_useless_reg_load
```

From verification perspective, that property is an inefficient use of assertions and is meaningless because it just emulates RTL. A more meaningful property would define one at a higher level of abstraction that would embrace the subsystem requirements. For the above example, the following property would be of greater verification value:

```
property p_verification_minded_reg_out;
  logic [31:0] v_data;
  @ (posedge clk) $rose(data_req) ##1 (data_ack, v_data==data_bus) |=>
      ##[0:2] reg_out==v_data;
endproperty : p_verification_minded_reg_out
```

That property addresses the subsystem viewpoint, rather than the RTL viewpoint, and states that upon a new handshake the output should have certain timing and data relationships to the subsystem.

8.3.2 Classification of properties

Properties can be classified as:

1. Design centric
2. Assumption centric
3. Requirement / verification centric
4. Environmental
5. Coverage

8.3.2.1 Design Centric

Design centric properties represent assertions added by the RTL designers to characterize white-box design attributes (i.e., relevant to variables local to the RTL module). These typically address properties of FSMs, local memories, clock synchronization logic, and other hardware related designs. The design should conform to the expectations of these properties. They should not however emulate or repeat RTL code.

8.3.2.1.1 Style in FSM properties

When defining properties for an FSM, define the transitions from one state to the same or to another state upon what is observed, as opposed to an RTL viewpoint that represents the causes of the transitions.

In RTL, FSMs express the next state based on the current state and inputs or other states. A similar approach can be done in defining properties, but that just emulates the RTL, and is not useful from a verification perspective. A better style in defining such properties is to consider a reactive strategy. Specifically, the properties assert that if the FSM is in a state, then they identify the possible causes that forced the FSM to be in that state. The following example demonstrates the concepts of these styles.

Requirements: If the FSM is in IDLE state and the CPU becomes ready for data (*ready4data* signal), then the FSM shall assert at the next cycle the *bus_request* signal. If the FSM is in a PAUSE state, and the CPU releases the *hold* signal, then the FSM shall assert at the next cycle the *bus_request* signal.

```
// Style that emulates RTL, and not very useful from a verification perspective
property pRTLStyle;
 @ (posedge clk) disable iff (!reset_n)
     (fsm_state_reg == IDLE && $rose(ready4data)) ||
     (fsm_state_reg == PAUSE && $fell(hold)) |=> bus_request;
endproperty : pRTLStyle
```

```
// Reactive style that clarifies that if an effect occurs, it could have been one of several causes.
property pReactiveStyle;
 @ (posedge clk) disable iff (!reset_n)
  bus_request |-> $past(fsm_state_reg ==IDLE) && $past(ready4data) or
                  $past(fsm_state_reg ==PAUSE) && $past(hold);
endproperty : pReactiveStyle
```

These two examples may appear identical. However, the RTL style of expressing properties closely resembles the RTL code, and offers very little value to the verification process of the design. The recommended reactive style addresses a different strategy because it states the possible conditions (i.e., antecedents) that caused the effect (i.e., consequent). This approach distances the properties from the RTL code, thus offering value to the verification process. Note that the reactive style cannot be expressed when cycle ranges are required because the ticks in the **$past** function must be static, known at elaboration time. Also note that "**$past($rose** (variable)) is illegal. If such a test is desired, an option would be to use the **ended** method of a named sequence. That style enables the use of ranges. For the above example, the following can be written:

```
sequence qPast1Ready4Data;
 @ (posedge clk) $rose (ready4data) ##1 `true;
endsequence : qPast1Ready4Data

sequence qPast1Hold;
  @ (posedge clk)  $fell(hold) ##1 `true;
endsequence : qPast1Hold

// Reactive style that clarifies that if an effect occurs, it could have been one of several causes.
property pReactiveStyle2;
 @ (posedge clk) disable iff (!reset_n)
  bus_request |-> ($past(fsm_state_reg ==IDLE) && qPast1Ready4Data.ended) or
                  ($past(fsm_state_reg ==PAUSE) && qPast1Hold.ended);
endproperty : pReactiveStyle2
```

8.3.2.2 Assumption Centric

Assumption centric properties represent assumptions about the design environment. They are used in formal verification to specify assumptions about the inputs. In simulation, assumption centric properties are used to verify coverage of the assumed properties (i.e., that those conditions were exercised). For example, if inputs *bus1_request* and *bus2_request* are mutually exclusive in an application, then the following property can be specified:

```
property pNeverBus1Bus2Requests; @ (posedge clk)
  not (bus1_request && bus2_req);
endproperty : pNeverBus1Bus2Requests
```

The **assume** directive can inform a verification tool to assume such a condition during the analysis.

```
mpNeverBus1Bus2Requests: assume property (pNeverBus1Bus2Requests);
```

8.3.2.3 Requirement / Verification Centric

Many requirement properties are best described using a "cause-to-effect" style that specifies what should happen under a given condition. For example, a handshake can be described as "if request is asserted, then acknowledge should be asserted within 5 cycles".

```
property p_handshake;
  @ (posedge clk) req |=> ##[0:4] ack;
endproperty : p_handshake
```

However, a reverse way of looking at the design can lead to what are called "Verification Centric Properties". Such properties can specify what should NOT happen under given conditions. Other forms of Verification centric properties could be to look for "effect-to-cause" reverse relationship. The purpose of this approach is to verify conditions that are often not checked. For example, to detect errors in the pointers of a FIFO (i.e., the READ pointer and the WRITE pointer) a cause (e.g., a POP and no PUSH) should reflect no change to the WRITE pointer. A FIFO FULL implies no increment to the FIFO counter[10], and no increment to the WRITE pointer. Table 8.2.2.3 lists a few Verification centric properties for a FIFO Controller. These properties refer to read and write pointers that are typical of a FIFO controller.

[10] This is implementation dependent. For demonstration purposes, a FIFO counter is assumed here.

Table 8.2.2.3 Verification Centric Properties for a FIFO Controller

1. If POP, no change to the WRITE pointer.
   pop |=> **$stable**(wr_ptr); *// wr_pntr is a vector pointing to where new data will be written*

2. If PUSH, no change to READ pointer.
   push |=> **$stable**(rd_ptr) *// rr_pntr is a vector pointing to where data will be read*

3. If an increment in *fifo_count* is observed, then in the previous clock, the condition PUSH && !POP should have been TRUE.
   fifo_count == **$past**(fifo_count) + 1'b1 |->
   (**$past**(push) && ! **$past**(pop));

4. If decrement in *fifo_count* is observed, then in the previous clock, the condition POP && !PUSH should have been TRUE.
   fifo_count == **$past**(fifo_count) - 1'b1 |->
   (**$past**(pop) && ! **$past**(push));

5. A stable *fifo_count* would mean either:
   a. A simultaneous PUSH & POP or
   b. No operation.
   $stable(fifo_count) |->
   ((**$past**(push) && **$past**(pop)) ||
   (! **$past**(push) && ! **$past**(pop))
)

6. If *fifo_full*, no increment to *fifo_count*, no increment to WRITE pointer.
   fifo_full |=> (fifo_count != (**$past**(fifo_count) + 1'b1) &&
   wr_ptr != (**$past**(wr_ptr) + 1'b1));

7. If *fifo_empty*, no decrement to *fifo_count*, no increment to READ pointer.
   fifo_empty |=> (fifo_count != (**$past**(fifo_count) – 1'b1) &&
   rd_ptr != (**$past**(rd_ptr) + 1'b1));

While these properties may look strange, they are "necessary" conditions for the operation of the FIFO. They also represent viewpoints that are far away from the RTL implementation. Such specifications tend to be more generic and are applicable in formal verification.

8.3.2.4 Environmental Properties

Another class of properties that one should consider adding to a design is "environmental properties", i.e., properties that are required for a proper functionality of the design when it is embedded in a larger design. These properties are not strictly of this design itself, rather assumptions about the environment where this design shall be integrated. If these were to be violated, the design doesn't guarantee proper functionality, and it is the responsibility of the environment surrounding this design instance to prevent such events. Without Assertion-Based Verification, the verification of environmental properties is transferred to the outer environment of the black-box, meaning that the check of environmental characteristics is not embedded in the black-box design. With that approach, any such environmental violations might go undetected for many clock cycles, and depending on how solid the black-box checkers are, may lead to data

integrity at the primary outputs. Recent experiences with SoC designs using lot of third party IPs reveal that the biggest contributors to design errors are due to incorrect usage of an IP block. For example, the IP block wasn't intended to be used in a certain mode, and such assumptions were mute. Table 8.2.2.4 lists sample environmental properties for a FIFO Controller.

Table 8.2.2.4 Environmental Properties for a FIFO Controller

1. Never PUSH while the *fifo_full* is asserted.
2. Never POP while the *fifo_empty* is asserted.

8.3.2.5 Coverage Properties

Another important application of ABV is the creation of functional coverage monitors at a level close to the RTL, so that the efficiency of white-box verification can be measured. With the emergence of hardware verification languages (HVL), functional coverage has become an accepted methodology to capture design features at a higher level. Functional coverage typically tends to be at a very high level such as execution of different instructions, sequences of instructions etc. Conventional code coverage presents the structural coverage of the HDL code. While the combination of these two metrics (high-level functional and structural) has been beneficial, the tend is to address levels of abstraction that may go untracked, such as the functional coverage of design elements. For example, Section 6.2.7.1 addressed the case where, in a battery-powered design, functional coverage revealed that one portion of the logic was performing unnecessary read/write operations to memory. Specifically, that portion of the logic had a high number of non-vacuous completions of memory accesses as compared to the non-vacuous completions of the logic that made use of those accesses. A correction to the logic to eliminate those unnecessary memory accesses yielded a 10% power efficiency increase.

In principle, functional coverage can capture such requirements. Functional coverage properties are typically defined by architects and verification engineers, and hence tend to miss implementation level details (or white-box functional coverage). Using ABV, RTL designers can capture implementation specific coverage requirements. This could mean monitoring and profiling interesting events/corner cases.[11] In traditional verification methodologies, these items were captured in a testplan and generally executed using directed tests. With recent advances in the Coverage Driven Random Verification methodology, the directed tests are substituted with constrained random tests. These generate scenarios "in-and-around" interesting scenarios so that the verification doesn't limit itself to the identified "hot spots", but rather spans across the design space. Table 8.2.2.5 lists some of the coverage properties for a FIFO Controller.

[11] For example, in FIFO designs, regression runs to see statistics such as:

- How often the FIFO was filled above a threshold.
- How often the FIFO was freed below a threshold etc.

Table 8.2.2.5 Coverage Points for a FIFO Controller

1. Almost Full & Push
   cAlmostFullPush : **cover property** (almost_full && full);

2. Almost Empty & Pop
   cAlmostEmptyPop : **cover property** (almost_empty && pop);

3. Read and Write Pointers both at same level
   cRdPtr_eq_WrPtr : **cover property** (rd_pntr == wr_pntr);

4. Read and Write Pointers at their "edge" values, i.e., 0, MAX, Middle
   cRdWrPntrEdge :
   cover property (rd_pntr==0 || rd_pntr=MAX || wr_pntr==0 || wrpntr== MAX);

5. Push and Pop at the same time
   cPushPop : **cover property** (push && pop);

6. Sequence coverage: Almost full, then PUSH followed by POP, followed by PUSH, PUSH, POP in consecutive cycles.
   cAfullPushPopPushPushPop :
   cover property (almost_full |=> push ##1 pop ##1 push ##1 push ##1 pop);
7. Sequence coverage: Almost empty, then POP followed by PUSH, followed by POP, POP, PUSH in consecutive cycles.
   cAemptyPopPushPopPopPush :
   cover property (almost_empty |=> pop ##1 push ##1 pop ##1 pop ##1 push);

While these are legal and valid scenarios to which the design is supposed to properly respond, the verification engineer needs to make sure that such events have been exercised during their simulations.

Another classical application of Assertion-Based Verification during the design phase is the possible value addition of ABV to design optimization. Consider two design blocks being developed in parallel by different designers. One design uses a defensive style that includes buffers at both the output of *BlockA* and input of *BlockB* to ensure no loss of data. With assertions, these buffers can be "profiled" (also known as Watermarking technique). A buffer watermarking technique can monitor the buffer usage levels during regression runs. The output of such profiling may demonstrate that the buffer usage level in *BlockB* is very low, whereas *BlockA* has already buffered the data. Such a feedback can be used to optimize the design and/or micro-architecture, thereby saving area during synthesis. A similar profiling concept with ABV can be used to measure a FIFO depth using a system-level simulation.

8.3.3 Process of Writing Properties and Assertions

Chapter 1 addressed the notion that **assertions** document the combinational and temporal properties that must hold true (or never occur) for the design to be correct. These definitions, along with logic design experience, point to the following process in writing properties:

1. **Identify the characteristics of the design**. Depending upon which design element (e.g., specification, RTL) is being described, the characteristics are defined either in a document (such as a specification), or are implicitly known or specified by the designer. The most common format for these design characteristics is of the form "antecedent and

consequent" (or cause and effect). The antecedent represents a signal or a sequence or an endpoint that causes a reaction, or effect, to occur. For example, a direct memory access (DMA) request or a transaction for a memory dump represents the antecedent. The consequent to that antecedent might be a sequence where the unit currently holding the memory bus releases its hold of the bus within 5 cycles (i.e., drivers into high-impedance), and then provides a DMA acknowledge until the DMA request is released. That property can be expressed:

```
property pDMA_CYCLE;
  @ (posedge clk) $rose(dma_request) |=>
      ##[*0:4] databus_is_intristate ##1 dma_ack ##[0:$] $fell(dma_request);
endproperty : pDMA_CYCLE
```

Another example of a sequence as an antecedent might be a configuration of the transmit registers, followed by a load of transmit data into the local buffer, and then a *transmit_go* command. The effect could be the serial transmission, in 32-bit blocks, of 128 32-bit words of data from the local buffer. A block consists of a 2-bit header, and a one-bit trailer for synchronization. That property can be expressed:

```
property p_serial_xmission;
  @ (posedge clk) $rose(done_xmit_config) ##1 done_xmit_data_load ##1 xmit_go |=>
      (header[*2] ##1 data[*32] ##1 trailer) [*128];
endproperty : p_serial_xmission
```

2. **Write sequences as named sequences.** The examples in item 1 did not use named sequences. However, named sequences, particularly if they are complex, improve readability and help in code reviews because they name a sequence of events, associate that sequence to a grouped function, and separate that sequence from the associated properties. Writing named sequences is not a requirement, but a recommendation. Other advantages in using named sequences are that endpoints can be defined from instantiated sequences, and more complex sequences can be built using named sequences. In addition, parameterized sequences offer the advantage of re-use. Considering the previous example the antecedent can be expressed as sequence for:
- Configuration of all transmit registers,
- Load transmit data into the local buffer and the *transmit_go* command.

```
sequence qCONFIG_XMIT1; // Configuration of one transmit register
  write ==1'b1 && addr==2'b10; endsequence : qCONFIG_XMIT1

sequence qCONFIG_XMIT2; // Configuration of another transmit register
  write==1'b1 && addr=2'b11; endsequence : qCONFIG_XMIT2

// Complete antecedent
sequence qCONFIG_DATA_GO;
   qCONFIG_XMIT1 ##1  qCONFIG_XMIT2 ##1
   data_buffer_loaded==1'b1 ##1  transmit_go==1'b1;
endsequence : qCONFIG_DATA_GO
```

The consequent of the previous example can be expressed as:
- Serial transmission of 128 32-bit words from the local buffer.

```
sequence qFRAME_DATA; //Data framing is taken into account
   header[*2]; data[*32]; trailer;
```

Frame to be repeated 128 times

Above sequence states that it consists of a 2-bit header, 32 bits of data, and a trailer. Signal *header*, *data*, and *trailer* represent envelopes when header, data, and trailer are asserted in the serial data. These signals are used by the RTL to control the serial stream.

3. **Write the properties and assertions using the defined sequences.** Using the defined sequences, the properties can be easily defined. For the previous example, the following properties and assertion can be written as

```
property pCPU_TO_DATA;   @ (posedge clk) disable iff (!reset_n)
      qCONFIG_DATA_GO |=> qFRAME_DATA [*128];
endproperty : pCPU_TO_DATA

apCPU_TO_DATA_1 : assert property (CPU_TO_DATA);
```

The above property states that following a configuration of the data, as defined by the sequence qCONFIG_DATA_GO, there should be 128 consecutive serial data blocks, as identified by the sequence qFRAME_DATA. The content of the serial data is ignored in this assertion.

4. **Use Endpoints to Improve Readability**

Endpoints are not sequences, but specify the termination of a sequence. The conversion of a sequence into a Boolean provides two advantages:

- Can keep a sequence/property simple by referring to the endpoint of any sub-sequence sampled on the same or a different clock.
- Make it easier to align the completion of a sub-sequence within a larger sequence context.. For example: **property** pCPU_TO_DATA;

```
@ (posedge clk) disable iff (!reset_n)
qCONFIG_DATA_GO.ended |=> qFRAME_DATA [*128]; endproperty : pCPU_TO_DATA
```

Note: The use of "qFRAME_DATA.**ended** [*128]" as a consequent is incorrect because it states that the consequent must 128 consecutive ended sequences in 128 cycles, rather than the intended meaning of 128 sequences, each lasting 35 cycles (header, data, trailer).

5. **With "disable iff" use triggered method of sequence when condition is synchronous**

The typical condition in a **disable iff** expression is an asynchronous reset. Guideline rules:

- Simple asynchronous condition: If the disabling expression represents an asynchronous condition use the expression as the condition (e.g., **disable iff** (!reset_n)).
- Synchronous signal: If the disabling expression represents a synchronous condition that is used as an asynchronous condition, such as a synchronous reset used driving the asynchronous ports of flip-flops, then use the expression as the condition. However, if the disabling expression represents a synchronous condition that is used as a synchronous condition then first create a named sequence for that expression, and use the **triggered** method of the named sequence as the disabling condition. For example:

```
sequence qSoftReset;
  @ (posedge clk) soft_hardware_reset_command;
endsequence : qSoftReset
property pTest;
  @ (posedge clk) disable iff (qSoftReset.triggered)
   a |=> b;
endproperty : pTest
```

- Disabling sequence: If the condition is a sequence of events, create a named sequence, and then use the endpoint of that named sequence with **triggered** method. For example:

```
sequence qSoftReset2;
  @ (posedge clk) $rose(soft_hardware_reset_command) ##1 ack;
endsequence : qSoftReset2

property pTest;
 @ (posedge clk) disable iff (qSoftReset2.triggered)
  a |=> b;
endproperty : pTest
```

6. **Define Reusable Libraries of Verification Units**

To speed up the verification process, a library of verification units can be defined and accessed. This library may include the following:

- Commonly used assertions. These include properties similar to those defined in the Accellera Open Verification Library. [12]
- Verification IP. In this context, a verification IP is a module that consists of a collection of related assertions and data structures, along with some SystemVerilog code, to verify the proper operation of the interfaces and data integrity of a device. Examples of verification IPs are provided in Section 9.14 for a memory data integrity check, Section 9.15 for a bus data transfer with data integrity check, and in file *ch6/fifo_if.sv*. Some vendors, such as 0-In, provide a wide range of verification IPs. [13]

7. **Consider scoreboards for computation of data centric properties.**

Many designs require the verification of data integrity throughout the design. For example, the "first-in-first-out" behavior of a FIFO needs to be verified. The authors believe that such a check is better done using a SystemVerilog mailbox or queue than SystemVerilog Assertions. It is important to understand that this recommendation is NOT because SystemVerilog Assertions can't handle this; in fact it can in some instances through the use of local variables. However, SystemVerilog Assertions should be used sensibly and a behavioral checker (such as FIFO behavior) is a "data centric" problem, and hence is better handled using a regular, high level "programming language feature" than a "control centric" ABV methodology. Assertions can however be used to support and simplify the verification with the behavioral checker, thus providing synergy between SystemVerilog programming language (e.g., with scoreboards, and scoreboard management) and assertions.

---

[12] http://www.eda.org/ovl/download.html

[13] http://www.0-in.com/

Examples of verification IPs: PC-Interconnect (e.g., AGP, LPC, PCI, USB), Processor (e.g., AMBA, HyperTransport, Rapid IO), Switch Fabric (e.g., InfiniBand Link Layer, PCI Express), Memory (e.g., DDR SRAM, DDR SDRAM), Networking (e.g., CSIX L1, POS-PHY L2, UTOPIA L1), Storage (e.g., SAS).

9 SYSTEMVERILOG ASSERTIONS DICTIONARY

This chapter provides a series of classes of application examples that translate English descriptions of properties to SystemVerilog properties. Many of those English properties were downloaded from "http://www.eda.org/vfv/". That site does not provide SystemVerilog Assertions equivalents. The SystemVerilog Assertions properties for these examples are not necessarily unique, as the language allows different methods to express the same or similar connotations. The purpose of these examples is to provide a type of dictionary that readers can use to quickly lookup a common design requirement and to demonstrate how SystemVerilog Assertions can be used to express requirements. Since natural language descriptions are often imprecise or vague, some assumptions and clarifications were made while expressing them in SystemVerilog Assertions. Users may disagree with some of the translations. That disagreement is "healthy" since it demonstrates the point that natural language descriptions are imprecise, and can be clarified with an assertion language like SystemVerilog Assertions.

The phrase **COND** is an abbreviation for **CONDITION** in this chapter. In every example a set of potential SystemVerilog Assertions constructs that could be used to describe the requirement is provided. The rationale being, the users can choose the construct that is more appropriate for the problem in hand. The following two constructs (and their hierarchical derivatives) are applicable to most examples, and hence will be assumed to be potential candidates for all the examples:

sequence_expr |-> property_expr	sequence_expr |-> sequence_expr |-> property_expr
sequence_expr |=> property_expr	sequence_expr |=> sequence_expr |=> property_expr

Per guidelines in 8.2.5.1, "Avoid Unbounded Ranges", the following constant shall be used within many properties in this chapter:

```
parameter MAX_CYCLES =100;
```

9.1 IF *COND1*, THEN *CON*

)2

If a snoop hits a modified line i
transaction at next clock cycle must

in the cache (*snoop && hit_modified* (COND1)), then the
be a snoop writeback (*writeback* (COND2)).

```
property pCacheSnoopWriteback:
 @ (posedge clk) (snoop && hit_r
endproperty : pCacheSnoopWrit
```

```
nodified)|=> (writeback);
eback
```

snoop && hit_modified	0
writeback	-

	0	1	-
	-	-	1

If a reset condition is needed, the p
example, with an asynchronous re
property pCacheSnoopWriteback
@ (posedge clk) disable iff (!res
(snoop && hit_modified)|=
endproperty : pCacheSnoopWri

property can be enhanced with a **disable iff** construct. For
set, active low:
2;
et_n) *// disable as long as reset_n==0*
(writeback);
teback2

Note: As mentioned in section 3.
then 1) any pending threads (and
current thread is also vacuously
waiting to either be TRUE or FAI

2.5.2, wherever the disabling condition (e.g., *reset_n*) is TRUE,
there could be many) will then become vacuously TRUE, and 2)
TRUE. A property is either in a TRUE state or is in progress,
SE.

If a synchronous reset or a disabl
and future obligations of an asser

e sequence endpoint causes an immediate termination of current
tion, then the "**triggered**" method of a sequence can be used.[1]

9.2 IF *COND1*, THEN AT NEXT *COND2*, *COND3*

If a snoop hits a modified line in
trans_start (COND2) a snoop
hierarchical use of the following

the cache (*snoop && hit_modified* (COND1)), then at the next
writeback (COND3) occurs. This is a good application of the
property :

sequence_expr |=> property_exp

r sequence_expr |-> property_expr

```
property pCacheSnoopWriteb
 @ (posedge clk) (snoop && h
        ##[0:$] $rose(trans
endproperty : pCacheSnoopW
```

```
ack4;
it_modified) |=>
start) |-> writeback;
riteback4
```

The ##0 may also used instead
a small difference in results:

```
property pCacheSnoopWriteb
 @ (posedge clk) (snoop && h
        ##[0:$] $rose(trans
endproperty : pCacheSnoopW
```

f the overlapping implication operator, as shown below, but with

```
ack_not_same;
it_modified) |=>
tart) ##0 writeback;
riteback_not_same
```

[1] See section 8.3.3 in "Process of Writing Properties and Assertions" on discussion of when to use the **triggered** method. Also see 4.7.3 on the application of the **triggered** method.

Note the use of **$rose**(trans_start) to reflect the next occurrence of *trans_start*.

Also, note that the above two properties express different notions. Property *pCacheSnoopWriteback_not_same* cannot fail because if the consequent fails, the delay range ##[0:$] waits for another cycle until it can succeed. However, property *pCacheSnoopWriteback* can fail because if consequent "##[0:$] **$rose**(trans_start)" holds, then that consequent becomes an antecedent to the next expression. The property then requires the consequent *writeback* to occur if antecedent "##[0:$] **$rose**(trans_start)" holds.

9.3 IF *COND1*, THEN AFTER NTH *COND2, COND3*

If signal *enable* rises (COND1), then anytime after the fourth data transfer (*xfr* (4th occurrence of COND2)), signal *pending* (COND3) must rise.

```
property pEnable2Pending;
  @ (posedge clk) $rose(enable) |-> xfr[=4] ##1 $rose(pending);
endproperty : pEnable2Pending
```

Slight variant example: If the above spe
(instead of any cycle after) the 4th occurre
be written with a simple replacement
repetition". Note the "[->4]" instead of "[

```
property pEnable2Pending1;
  @ (posedge clk) $rose(enable) |-> xfr[
endproperty : pEnable2Pending1
```

9.4 IF *COND1* AND FIRST *CON*

If signal *req4* (COND1) is asserted, then
(*ack==1'b1* and *id!=4*, COND2), then re
acquire signal (COND4) is asserted (incl

```
property pReqAck4 ;
  @ (posedge clk) $rose(req4) |-> ##[0
  // Not same, but another option
  // @ (posedge clk) $rose(req4) |-> ##
endproperty : pReqAck4
```

9.5 IF *COND1* AND FIRST C

If signal *hit* is active and signal
becomes active (COND2). signal

```
property pHitPendingSel5,
  @ (posedge clk) $rose(hit && !p
endproperty : pHitPendingSel5
```

Slight variant example: If sig
time signal "pending" becom

```
property pHitPendingSel5b
  @ (posedge clk) ($rose(hit &&
endproperty : pHitPendingSel5
```

ification required that "if *COND1* then the **cycle after**
ce of *xfr*, *COND3* must occur, then the property could
of "GOTO" repetition instead of "Non-Consecutive
=4]".

```
->4] ##1 $rose(pending);
```

)2, THEN *COND3* UNTIL *COND4*

if the first acknowledge identification is not equal to 4
quest *req4* must stay deasserted (COND3) until the
usive of *acquire* cycle).

```
:$] (ack && id != 4) |-> (!req4 [*0:$] ##1 acquire);

[0:$] (ack && id != 4) ##0 (!req4 [*0:$] ##1 acquire);
```

'OND2, THEN *SEQUENCE*

pending is inactive (COND1), then the next time *pending*
'5 must be active.

```
ending) ##1 pending[->1] |-> (sel5);
```

'hit" is active and signal "pending" is inactive, then the next
ive, signal "sel5" must be active in the next cycle.

```
!pending) ##1 pending[->1]) |=> (sel5);
```

9.6 BETWEEN *COND1* AND *COND2*, *SIGNAL 1* ASSERTED

Between a request *req* (COND1) and its acknowledge *ack* (COND2), the *busy* signal (signal 1) must remain asserted.

```
property pReqAckBusy;
  @ (posedge clk) $rose(req) |-> (busy[*0:$] ##1 ack);
endproperty : pReqAckBusy
```

The property does not insure that *req* remains asserted until the *ack* is received. If that is a requirement, the property can be modified as follows:

```
property pReqAckBusy2;
  @ (posedge clk) $rose(req) |-> (busy&& req) [*0:$] ##1  ack;
endproperty : pReqAckBusy2
```

9.7 IF *COND1* AND THEN 1 OC CURRENCE OF *COND2* THEN *SEQUENCE*

If a data packet of any size starts a one bit envelope called *pkt_data_vld* (COND1) and eventually gets a *lastbit* (COND2), th en next data packet *pkt_data_vld* must have the *first* bit asserted (sequence).

Option 1:

```
property pPacketLastFirst1;
 @ (posedge clk)
   $rose(pkt_data_vld) |-> ##[*0:$](las
                       |-> ##[0:$] ($rc
 endproperty : pPacketLastFirst1
```

Option 2:

```
property pPacketLAstFirst2;
 @ (posedge clk) $rose(pkt_data_vld
                 |=> (##[0:$] $rose(pkt_dat
endproperty : pPacketLAstFirst2
```

Note: Option 1 or option 2 properties can never fail, but must complete. A user may want to put a time limit on the latency of the packets, as dictated by the architecture.

```
property pPacketLastFirst3;
 @ (posedge clk) $rose(pkt_data_vld) ##1 lastbit[->1]
                 |=> (##[0: MAX_CYCLES] $rose(pkt_data_vld) && first);
endproperty : pPacketLastFirst3
```

9.8 IF *COND1* THEN N OCCURRENCES OF *COND2* BEFORE *COND3*. N IS VALUE OF SIGNAL

If a WRITE command *cmd* (COND1) starts and *size*==N (where N equals 1 through 8), then N assertions of signal *gx_start* (COND2) should occur before the LASTbit *data_last* (COND3) becomes active.

The following property may seem intuitive because it dynamically computes the replication operator. However, a dynamic repetition range is illegal in SystemVerilog Assertions.

ILLEGAL

```
property pWriteNstart_ERROR; //
  @ (posedge clk)
    $rose(cmd=WRITE) && size >=1 && size <=8) |=>( gx_start[=size] ##1  data_last);
endproperty : pWriteNstart_ERROR
```

Illegal because value is not static

However, the following construct can be used:

```
if ( expression_or_dist ) property_expr [ else property_expr ]
sequence_expr |-> property_expr
| sequence_expr |=> property_expr
```

Note that the repetition of *gx_start* is static, rather than dynamic (which is illegal).

```
property pWriteNstart; //   OK
  @ (posedge clk)
  if ($rose(cmd==WRITE) && size ==1) gx_start[=1 ] ##1  data_last else
  if ($rose(cmd==WRITE) && size ==2) gx_start[= 2] ##1  data_last else
  if ($rose(cmd==WRITE) && size ==3) gx_start[= 3] ##1  data_last else
  if ($rose(cmd==WRITE) && size ==4) gx_start[= 4] ##1  data_last else
  if ($rose(cmd==WRITE) && size ==5) gx_start[= 5] ##1  data_last else
  if ($rose(cmd==WRITE) && size ==6) gx_start[= 6] ##1  data_last else
  if ($rose(cmd==WRITE) && size ==7) gx_start[= 7] ##1  data_last else
  if ($rose(cmd==WRITE) && size ==8) gx_start[= 8] ##1  data_last ;
endproperty : pWriteNstart
```

The following figure demonstrates one case for this property.

Property pWriteNstart for size == 3

9.9 IF *COND1* AND WITHIN N CYCL ES Y OCCURRENCES OF *COND2*, THEN *COND3*

If *dgrant* is asserted (COND1), then as soon as *dbusy_n* (COND2) is high for two clocks (Y occurrences), then the data bus should contain valid data (indicated by the assertion of *dbus_enb* signal) (COND3). Assume that a message has a maximum length of 256 cycles (N cycles).

```
property pDataGrantDbusy;
  @ (posedge clk) $rose(dgrant) ##[0:255] dbus
endproperty : pDataGrantDbusy
```

That property states that if rose of *dgrant*, a y_n[*2] |=> dbus_enb;
consecutive cycles, then the next cycle should h nd within 256 cycles *dbusy* is asserted for two ave valid data (i.e., *dbus_enb*).

Another way to express a similar, but not identi al property is a follows:

```
property pDataGrantDbusy2;
   @ (posedge clk) $rose(dgrant) |-> ##[1:MA X_CYCLES] (dbusy_n) [=2] ##1 dbus_enb;
endproperty : pDataGrantDbusy2
```

That property states that if *dgrant* is asserted. th en anytime after the 2nd occurrence of *dbsusy_n*, then *dbus_enb* becomes newly asserted . Note that in this case, the occurrences of *dbusy_n* need not be consecutive, and *dbus_enb* can occur any t ime after the last occurrence of *dbusy_n*.

A third alternative is to use the **goto** repetitio n operator to express a similar, but not identical property is as follows: Note that in this case, th e occurrences of *dbusy_n* need not be consecutive, and *dbus_enb* must occur in the cycle after the l ast occurrence of *dbusy_n*.

```
property pDataGrantDbusy3;
   @ (posedge clk)  $rose(dgrant) |-> ##[1:$] d busy_n [->2] ##1 $rose(dbus_enb);
endproperty : pDataGrantDbusy3
```

9.10 IF *COND1*, THEN *COND2* UNTIL *COND3*

Between a *req* (COND1) and its *ack* (COND3) the *busy* (COND2) signal must remain asserted.

```
property pBusRequest;
  @ (posedge clk) $rose(req) |-> (busy[*0:$]  ##1 ack);
endproperty : pBusRequest
```

9.11 IF COND1 THEN COND2 BEFORE COND3

Following a *bus_enable* (COND1), the header *header_enable* (COND21) must be asserted before the data framed with *data_enable* (COND3). The following property states that following a *bus_enable*, there should never be a condition where *header_enable* is not asserted, but *data_enable* is asserted.

```
property pFaultyEnbHeaderBeforeData;
  @ (posedge clk) disable iff (!reset_n)
    bus_enable |=> not (!header_enable[*0:$] ##1 data_enable)
endproperty : pFaultyEnbHeaderBeforeData
```

A different method to express the property expression is to define the expected conditions. The following property expression states that following a *bus_enable*, 16 cycles of *header_enable* occur and then *data_enable* occurs. Note that this is different than the previous property expression that defines the faulty condition instead of the expected condition.

```
property pEnbHeaderBeforeData;
  @ (posedge clk) disable iff (!reset_n)
    bus_enable |=> ##[0:$] header_enable[*16] ##1 $rose (data_enable);
endproperty : pEnbHeaderBeforeData
```

9.12 IF *COND1* IS FOLLOWED BY *COND2*, AND *COND3* IS NOT RECEIVED WITHIN 64 CYCLES WHILE *COND2* THEN ERROR (*COND5*). IF *COND3* IS RECEIVED WITHIN 64 CYCLES THEN *COND4*

If a request *req* (COND1) is granted with signal *grant* (COND2) and signal *frame* (COND3) is not received within 64 cycles while *grant* (COND2) remains asserted, then signal *time_out* (COND5) shall be asserted. If *frame* (COND3) is asserted within 64 cycles, then signal *acquired* (COND4) shall be asserted. Such a complex specification would typically require a complex FSM to be coded in either HDL or a HVL to check for its compliance, while SystemVerilog

Assertions describes it in a few lines. Two properties shall be used to express the above requirements.

```
property pReqGrantTimeout;
  @ (posedge clk) (req && $rose(grant) ##1 (!frame && grant )[*63]) |-> (time_out);
endproperty : pReqGrantTimeout
```

```
property pReqGrantAcquired;
  @ (posedge clk) (req && $rose(grant) ##1 (!frame && grant )[*0:62] ##1 frame) |-> (aquired);
endproperty : pReqGrantAcquired
```

9.13 FOR EVERY *WRITE* (COND1), DATA TRANSFERS MUST ALTERNATE BETWEEN ODD AND EVEN ENTRIES

For every *write* (COND1), data transfers must alternate between odd and even entries (signal 1). In other words, if there is a *write* (COND1), then as long as we are transferring data belonging to this *write*, consecutive data transfers must alternate (signal 1 toggles) between even and odd addresses. The ***within*** construct can be used for this requirement.

sequence_expr ::= sequence_expr **within** sequence_expr

```
property pWriteOddEven;
  @ (posedge clk)
   ( (! write_en[*0:$] ##1 write_en && !addr[0] ##1
       ! write_en[*0:$] ##1 write_en && addr[0])[*0:$]
    ) within  ($rose(write) ##[1:$] $fell(write));
endproperty : pWriteOddEven
```

Note the repetition of *write_en* and toggling of LSB of *addr* within a rise and fall of the *write* signal.

9.14 IF *COND1* THEN *COND2* IN N CYCLES UNLESS *COND3*

It is not possible to perform a simultaneous *read* and *write* operation. Doing so results in undefined behavior. When a *read* or *write* request is scheduled (COND1), it should be served (COND2) within 5 clock cycles (signal *rd_served* for *read*, or *wr_served* for write is asserted) unless an *interrupt* (COND3) occurs. The following constructs can be used for this example:

not property_expr
sequence_expr |-> property_expr
property_expr **or** property_expr
sequence method **triggered**

```
property pNeverReadWrite;
  @ (posedge clk) not (read && write);
endproperty : pNeverReadWrite
```

```
property pReadSchedule;
    @ (posedge clk)  $rose(read) |-> (##[1:5]rd_served) or (##[1:5] interrupt);
endproperty : pReadSchedule
```

```
property pWriteSchedule;
    @ (posedge clk)  $rose(write) |-> (##[1:5]wr_served) or (##[1:5] interrupt);
endproperty : pWriteSchedule
```

An alternative to the ORing of the properties would be to use the **diable iff** with the **.triggered** of a sequence.

```
sequence  qNewInterrupt;
 @ (posedge clk) ##[0:$] $rose(interrupt); // the use of the clocking event is optional
endsequence : qNewInterrupt

property pReadSchedule;
  @ (posedge clk) disable iff (qNewInterrupt.triggered )
     $rose(read) |-> (##[1:5]rd_served);
endproperty : pReadSchedule
```

```
property pReadWriteIntrpt ;
 @ (posedge clk) disable iff (qNewInterrupt.triggered)
  ($rose(read) || $rose(write)) |-> (##[1:5] (rd_served || wr_served ));
endproperty : pReadWriteIntrpt
```

Note that multiple occurrences of new *read* requests may complete the assertion by a single *rd_served*.

9.15 DATA INTEGRITY IN MEMORY. DATA READ FROM MEMORY SHOULD BE SAME AS WHAT WAS LAST WRITTEN

The data read (with *rd* control) from a memory is the last data that was written (with *wr* control) to that address. A READ must occur anytime after a WRITE. A READ is not required to occur. Any re-READ to the same address prior to another WRITE at that address should return the same data that was initially written.

The simplest solution to express this assertion is to use an associative array to store the incoming data with the memory address as the index to that array. Within a property, the **exist** function verifies upon a *read* request that the memory address was previously accessed for a write. If it was accessed, then the data within the associative array (AA) should be equal to the data read. The associative array acts as a scoreboard, and the property with the assertion acts as a verifier.

```
// This module monitors the memory transactions and verifies
// data integrity of the external memory
module memory_data_integrity_check (
   input write, // memory write
   input read,  // memory read
   input int wdata, // data written to memory
   input int rdata, // data read from memory
   input int addr;  // memory address
   input reset_n, // active low reset
   input clk);   // clock
 timeunit 1ns;
 timeprecision 100ps;
 int mem_aarray[*]; // associative array to be used by property
 wire mem_aaray_exists;  // exists at specified address
 // note: LRM 17.4.1 Operand types does not allow use
 // of methods on associative array in properties.
 assign mem_aarray_exists = mem_aarray.exists(addr);

 always@ (posedge clk)
  if (reset_n==1'b0) mem_aarray.delete; // Clears AA elements
  else if (write) mem_aarray[addr] <= wdata; // store data

 assign mem_array_addr = mem_aarray[addr];

 property p_read_after_write;
  @(posedge clk)
  (read && mem_aarray_exists) |-> mem_array_addr==rdata;
 endproperty : p_read_after_write
 ap_read_after_write : assert property (p_read_after_write);

 property p_read_before_write;
  @(posedge clk)
   not (read && !mem_aarray_exists);
 endproperty : p_read_before_write
 ap_read_after_write : assert property (p_read_before_write);
endmodule : memory_data_integrity_check
```

9.16 DATA INTEGRITY IN QUEUES. INTERFACE DATA WRITTEN MUST BE PROPERLY TRANSFERRED TO THE RECEIVING HARDWARE

The data received from an interface (with *wr* control) must be properly transferred to the receiving hardware (with *rd* control). The data is sourced at a 33 MHz rate and is extracted at a 25 MHz rate. The receiver uses a *ready4data* signal to throttle the transmission of one Kilo-words of data block. The data extracted by the receiver (*rdata*) is in the same order that it was transmitted. The following Figure shows a timing diagram of the interface.

1. A wr_33 stores wdata into dataQ. Hardware stores data into local buffer.
2. A rd_25 pops data from dataQ into rdata_from_q bus.
   Hardware stores buffer data into rdata
3. At cycle after rd_25, verification property compares rdata with rdata_from_q

The simplest solution to express this verification is to create a verification module, which can be inserted in a testbench. In this model, a SystemVerilog queue is used to store data at the 33 MHz rate up upon a *wr_33* signal. The declaration "**int** dataQ [$];" declares an unbounded queue. Data is stored into the queue with the **push_front** method. A *rd_25* synchronous to the 25 MHz clock causes data to be popped into a variable *rdata_from_q* from the queue, using the **pop_back** method. In this model, it is understood that the data is extracted by the receiver (*rdata*) in the cycle following the *rd_25* control. With the use of the queue, and queue management code, the verification logic needs not be concerned with the synchronization between the two clocks. That simplifies the definition of the verification properties. In fact, one property is needed to verify the data integrity:

```
// Data integrity check
 property p_read_data_integrity;
  @(posedge clk_rcv25)
   rd_25 |=> rdata_from_q==rdata;
 endproperty : p_read_data_integrity
 ap_read_data_integrity : assert property (p_read_data_integrity);
```

The throttling of data is the responsibility of the hardware design and is not verified directly. However, the effect of the throttling is verified with a simple property to insure that data cannot be read (with the *rd_25*) if no data was written, or if all the data that was written was extracted by the hardware.

```
// Never a READ with no data received
 int dataQsize; // queue size.
  // note: LRM 17.4.1 Operand types does not allow use
 // of methods on queues or associative arrays in properties.
 assign dataQsize= dataQ.size;
 property p_never_read_on_empty
  @(posedge clk_rcv25)
   not (dataQsize == 0 && rd_25);
 endproperty : p_never_read_on_empty
 ap_never_read_on_empty : assert property (p_never_read_on_empty);
```

If the hardware had a buffer size of 1024 words, and the interface is restricted from writing data into the design when the buffer is full, one could do the following:

1. Restrict the size of the verification queue in the declaration of the queue:
   int dataQ [$:1024]; *// queue, to store/read incoming data, restricted to 1K*
   This is not a requirement in this design because of definition of property *p_write_on_max_buf*
2. Add a property that states that there should never be a write if the hardware internal buffer is 1024. This can be done with information from the queue with no knowledge of the hardware structure.

```
property p_write_on_max_buff;
 @(posedge clk_xt33)
  not (dataQsize == MAX_BUFF_SIZE && wr_33);
endproperty : p_write_on_max_buff
ap_write_on_max_buff : assert property (p_write_on_max_buff);
```

The definition of the verification module is shown below.

```
// The data received from an interface (with wr control) must be properly
// transferred to the receiving hardware (with rd control).
// The data is sourced at a 33 MHz rate and is extracted at a 25 MHz rate.
// The receiver uses a ready4data signal to throttle the transmission
// of one Kilo-words of data block. The data extracted by the receiver
// is in the same order that it was transmitted.
module bus_xfr_data_integrity_check (
  input wr_33, // Transmitter write signal @33MHz
  input rd_25, // Receiver  read signal @25 MHz
  input int wdata, // write data from Transmitter
  input int rdata, // read data from Receiver
  input reset_n, // active low reset
  input clk_xt33,  // clock transmitter @33 MHz
  input clk_rcv25); // clock Receiver @25 MHz
 timeunit 1ns;
 timeprecision 100ps;
 parameter MAX_BUFF_SIZE=1024;
 int waddr; // write address for storage of incoming data
```

```
 int raddr; // read address for reading of received data
 int rdata_from_q; // read data from queue to compare to Receiver
 int dataQ [$]; // queue, to store/read incoming data
 int dataQsize; // queue size.
 // note: LRM 17.4.1 Operand types does not allow use
 // of methods on queues or associative arrays in properties.
 assign dataQsize= dataQ.size;

 // Storage of data into queue
 always @ (posedge clk_xt33)
  if (reset_n==1'b0)
    dataQ = {}; // clear the queue contents
  else  if (wr_33) dataQ.push_front(wdata);  // store data

 // Collect data from the Q
 always @ (posedge clk_rcv25)
  if (rd_25)  rdata_from_q <= dataQ.pop_back();

 // Data integrity check
 property p_read_data_integrity;
  @(posedge clk_rcv25)
   rd_25 |=>  rdata_from_q==rdata;
 endproperty : p_read_data_integrity
 ap_read_data_integrity : assert property (p_read_data_integrity);

 // Never a READ with no data received
 property p_never_read_on_empty;
  @(posedge clk_rcv25)
   not (dataQsize == 0 && rd_25);
 endproperty : p_never_read_on_empty
 ap_never_read_on_empty : assert property (p_never_read_on_empty);

 // never a write on a full buffer
 property p_write_on_max_buff;
  @(posedge clk_xt33)
   not (dataQsize == MAX_BUFF_SIZE && wr_33);
 endproperty : p_write_on_max_buff
 ap_write_on_max_buff : assert property (p_write_on_max_buff);
endmodule : bus_xfr_data_integrity_check
```

9.17 NEVER 2 CONSECUTIVE WRITES WITH SAME ADDRESS

Two consecutive writes cannot be done to the same address. Address appears one cycle after the *write_valid.*

```
property pConsecutiveWrites;
  @ (posedge clk) (wr_valid[*2] ) |-> (addr  !=$past(addr));
endproperty : pConsecutiveWrites
```

wr_valid1	wr_valid2	
	addr1	addr2

addr2 must be different from to addr1

9.18 CACHE CONTROLLER REQUIREMENT: A CACHED ADDRESS (COND1) WILL EVENTUALLY BE RETIRED (COND2) AND AFTER THAT, WITHIN 2 TO 7 CLOCKS THE CACHE COPY SHALL BE INVALIDATED (COND3)

If the cache of an address was set valid (*cache[address]*, COND1) then the next time the *retire* signal (COND2) is asserted for this address, the cache for this address will be invalidated (COND3) 3 to 8 clock cycles later. Assume a 1K memory. INVALID == 0, VALID == 1.

```
property pValidRetire;
 int vaddress;
 @ (posedge clk) disable iff (!reset_n)
  ($rose(address) && cache[address]==VALID, vaddress= address) |->
            (##[0:MAX_CYCLES] retire==vaddress ##[2:7] cache[vaddress]==INVALID);
endproperty : pValidRetire

apValidRetire : assert property (pValidRetire)
                    else report_sva_violation("Cache Retire Error");
```

9.19 DURING *COND1* NEVER *COND3* AFTER *COND2*. COND2 MAY OCCUR WITHIN N CYCLES AFTER COND1

During read transactions (*read_xctn*) (COND1), no target may assert *trdy* (COND3) in the first cycle following the assertion of *frame* (COND2), which may occur within 11 cycles after the start of the transaction.

```
property pReadTrdyFrame;
  @ (posedge clk) not ($rose(read_xctn) ##1 !frame[*0:10] ##1 frame ##1 trdy);
endproperty : pReadTrdyFrame
```

Another method to express the same concept is to use the overlapping implication operator with `false (which equals to 0).

```
property pReadTrdyFrame2;
  @ (posedge clk) $rose(read_xctn) ##1 !frame[*0:10] ##1 frame ##1 trdy |-> `false;
endproperty : pReadTrdyFrame2

property pReadTrdyFrame3;
  @ (posedge clk) $rose(read_xctn) ##1 !frame[*0:10] ##1 frame |=> trdy |-> `false;
endproperty : pReadTrdyFrame3
```

9.20 IF *COND1*, THEN NEXT N CYCLES *COND2*. IF NEW *COND1* BEFORE END OF *COND2*, THEN *COND2* EXTENDED FOR N CYCLES UNTIL NO *COND1*

Whenever a *read* signal is asserted (COND1), *busy* (COND2) has to be asserted for 3 cycles and then de-asserted. If an additional *read* arrives before *busy* has been de-asserted, then *busy* has to stay high for 3 cycles from the last *read*. The value of *busy* at the same cycle in which *read* is asserted is not important.

That description can be described with two properties. and the sequence method **triggered** to force a disable or abort of an ongoing property. The following diagram is used to explain the properties

1. The first requirement states "Whenever a *read* signal is asserted (COND1), *busy* (COND2) has to be asserted for 3 cycles and then de-asserted". This can be accomplished with the property expression "(read) |-> (busy[*3])". Referring to the diagram, *read* is TRUE at time 100ns, thus this thread '1' states that *busy* must be TRUE at time 100ns, 150ns, and 200ns.
2. The second requirement states "If an additional *read* arrives before *busy* has been de-asserted, then *busy* has to stay high for 3 cycles from the last *read*". This can be accomplished with the same property expression "(read) |-> (busy[*3])". At time 150ns, *read* is also TRUE, thus *busy* must be TRUE at time 150ns, 200ns, and 250ns. This starts a second thread '2'.
3. There is a need to verify when *read* is FALSE. The use of the property expression "(read) |-> (busy[*3] ##1 !busy)" would not satisfy the requirements because a new *read* must extend the *busy*, and all except the last thread would fail.
4. One solution is to cancel, or make all older thread vacuously TRUE is *read* fell. This can be accomplished with the **disable iff** (qFallRead.**triggered**), labeled *qFRT* in the diagram. The aborted verification threads are thread 1 and 2, as identified with filled circles in the diagram. At time 200ns, *read* falls and thread 1 and 2 become vacuously TRUE.
5. At time 200ns, with the fall of the read, there is a need to extend the *busy* for 2 more cycle. This can be expressed with the property expression "(**$fell**(read && busy) |-> (busy[*2] ##1 !busy);"
6. However, there still is a need to met the requirement "If an additional *read* arrives before *busy* has been de-asserted, then *busy* has to stay high for 3 cycles from the last *read*. " Thus, another rise of *read* before the property terminates should abort the verification of the consequent "(busy[*2] ##1 !busy)".
7. This abort of property verification can be done with the "**disable iff** (qRiseRead.**triggered**).

The sequences and properties for this model are as follows:

There is a need to cause a trigger for the sequence of a fall of a *read*.

```
sequence qFallRead;
  @ (posedge clk) ##[0:$] $fell(read);
endsequence : qFallRead
```

There is a need to cause a trigger for the sequence of a rise of a *read*.

```
sequence qRiseRead;
 @ (posedge clk) ##[0:$] $rose(read);
endproperty : qRiseRead
```

The first property handles the extension of the *busy* unless *read* falls. This is described below:

```
property pReadBusyRise;
 @ (posedge clk) disable iff (qFallRead.triggered )
  (read) |-> (busy[*3]) ;
endproperty : pReadBusyRise
```

Another property takes care of the extension of busy after fall of *busy* and *read:*

```
property pReadBusyFell;
 @ (posedge clk) disable iff (qRiseRead.triggered )
  ($fell(read && busy) |-> (busy[*2] ##1 !busy);
endproperty : pReadBusyFell
```

9.21 NEVER TWO *CONDS* WITHIN 2 CYCLES APART

The conditions are represented as bits in a vector. The bit vector "v[7:0]" is not allowed to contains more than one bit asserted. Moreover, if bit p (0 <= p <= 7) of v is asserted at time N and bit q of v is asserted at time M, then |M-N| >=3.

```
property pMustBeOnehot(v); // v must be one-hot
 @ (posedge clk) $onehot(v); //
endproperty : pMustBeOnehot
```

As soon as *v* is changed in value (! **$stable**(v)), then the property states that it must be stable for 2 more cycles:

```
property MustBeStableFor3Cycles (v);
   @ (posedge clk)   ! $stable(v) |=> $stable(v) [*2];
endproperty : MustBeStableFor3Cycles
```

Another approach is to detect a change in *v*, and then any other change within two cycles is an error.

```
property pNewVin3Cyles(v);
    @ (posedge clk) !$stable(v) ##[1:2] !$stable(v) |-> `false;
endproperty : pNewVin3Cyles
```

9.22 ASSUME RESET LOW FOR INITIAL N CYCLES

Put an assumption on the environment such that the design is reset for 5 clock cycles (*reset_n=0)* followed by *reset_n* staying high forever thereafter to enable normal functionality. Two assumptions used:

An assumption on reset sequence is important for formal verification. For example, a design might require that the reset be asserted for 5 clocks and then stay de-asserted. Any concurrent assertion gets evaluated on every sampling event, hence a one-shot effect is not achievable intuitively. However, SystemVerilog Assertions support procedural assertions, and hence one-shot effect can be achieved by using a procedural assertion in an **initial** block. This method declares the property as unclocked, but uses a verification directive (e.g., **assume** or **assert**) after a clock edge in an initial block. For example:

```
property p_preset_at_init;
  !reset_n[*5] ##1 reset_n;
endproperty : p_preset_at_init

initial @(posedge clk) mp_preset_at_init assume property (p_preset_at_init);
```

The property *p_preset_at_init* is evaluated only once after the first clock cycle of simulation.

```
// Once reset_n is high (deasserted), it remains deasserted
// Checked at every clock cycle.
property p_no_reset_forever;
 @(posedge clk) !reset_n |=> !reset_n;
endproperty : p+no_reset_forever
mp_no_reset_forever : assume property (p_no_reset_forever);
```

9.23 IF *COND1* AND N CYCLE LATER *COND2*, THEN *COND3* UNTIL *COND4*, UNLESS *COND5*

Every request (*req*, COND1) that is not retried (*retry*, COND2) (a retry may happen two cycles after assertion of signal *req*) must be followed by a sequence in which *busy* (COND3) is asserted until *endtx* (COND4) is asserted. If *endtx* is never asserted, then *busy* must stay asserted forever. A request that is honored with an acknowledge (*ack*, COND5) does not receive the *busy*.

```
sequence qAck;
 @ (posedge clk) ## $rose (req) ##[0:$] ack;
endsequence : qAck

property pReqRetryEnd;
 @ (posedge clk) disable iff (qAck.triggered)
  ($rose(req) ##2 !retry) |=> (busy[*0:$] ##1 endtx);
endproperty : pReqRetryEnd
```

The property *pReqRetryEnd* states that a rise of *req* followed by no *retry* in the 2nd cycle must be followed by *busy* until *endtx*. However, a request that is acknowledged cancels the requirement that the property *pReqRetryEnd* holds, and any pending threads hold TRUE vacuously.

9.24 IF SEQUENCE *COND1* FOLLOWED BY *N* NON-NECESSARILY CONSECUTIVE *COND2*, THEN *N* CONSECUTIVE *COND3* UNTIL *COND4*

A sequence beginning with the assertion of signal *go* (COND1), containing three not necessarily consecutive assertions of signal *get*, during which *kill* is not asserted (COND2), must be followed by a sequence of 3 consecutive assertions of signal *put* (COND3) before signal *endtx* (COND4) can be asserted. The following property could be used for this description:

```
property pGoGetKill2;
  @ (posedge clk) ($rose(go) ##1 (get && !kill)[=3]) |=>
  ((!endtx&& put)[*3] ##[0:$] endtx);
endproperty : pGoGetKill2
```

If *endtx* is never asserted, the property is in the "active", but not in terminated state.

9.25 IF COND1, COND2 DOESN'T CHANGE FOR N CLOCKS, UNLESS COND1 GOES HIGH AGAIN

"If signal *foo* goes high, signal *bar* doesn't change for 32 clocks, unless signal *foo* goes high again". The **triggered** method can be used to disable the property upon a new occurrence of *foo*. However, users need to carefully write the property such then the verification is not cancelled. For example, an incorrect use of a disable is shown below:

```
sequence qRoseFoo_baduse;                    ☹
  @ (posedge clk) ##[0:$] $rose (foo);
endsequence : qRoseFoo_baduse

property pBadUseOfDisable;                   ☹
  @ (posedge clk) disable iff (qRoseFoo_baduse.triggered)
 $rose(foo) |=>
   ($past(bar) == bar)[*32]);   // N is 32 in this example
endproperty : pBadUseOfDisable
```

Every time *foo* goes high, the obligation of "($**rose**(foo)) |=> ($**past**(bar) == bar)[*32])" gets cancelled, thus causing the property to never be verified. However, a signal *bar_steady* can be created in the verification model to identify when *foo* is expected to be steady. With this information, the following property can be written:

```
sequence qRoseFoo;
 @ (posedge clk) ##[0:$] $rose (foo) && bar_steady;
endsequence : qRoseFoo

property pGoodUseOfDisable;
 @ (posedge clk) disable iff (qRoseFoo.triggered)
  $rose(foo) && !bar_steady |=>
   ($past(bar) == bar)[*32];
endproperty : pGoodUseOfDisable
```

Timing Diagram for Property pGoodUseOfDisable

9.26 IF A SEQUENCE STARTS BUT DOES NOT COMPLETE, THEN STATE REGISTER MUST BE IN ERROR STATE

If a bus request *req* is not followed by an acknowledge *ack* within 5 cycles, or if the acknowledge *ack* is not followed by a *done* within 100 cycles, then the status register must flag an error (*status_reg==1)*.

This class of problem states that if an expected sequence starts but it does NOT finish, then a signal must have a certain value. This requirement is in general contradiction to the classical model that states that if a sequence starts and completes, then another sequence or signal value is expected.

- Traditional model for property expression (with *n* and *m* as an integer numbers)
  SequenceA_starts |-> [n:m] SequenceB

- Desired model to express requirement of "if no completion then another sequence:"
  SequenceA_starts |-> [n:m] SequenceB_timeout_with_no_success |-> SequenceC

Two solutions are examined.

1. If trigger condition starts then one of two outcomes ☺

An analysis of the requirement demonstrates that if the desired sequence starts, then either it completes or it fails. The trigger point is "**$rose**(req)". Based on this, one can check for one of two acceptable outcomes, which can be either a successful sequence, or a failed sequence.

```
// sequence needed to determine a successful completion with no errors
// ack within 5 cycles, followed within 100 cycle by rose of done, and then no error flag
sequence qDoneSuccessful;
  ##[1:5] ack ##[1:100] $rose(done) ##0 (status_reg == 1'b0);
endsequence : qDoneSuccessful

// sequence needed to define an error flag
// no ack for 5 cycles, and no rose of done for 100 cycles, and then error flag
sequence qError;
  !ack[*1:5]  ##1 !$rose(done)[*1:100] ##0 (status_reg == 1'b1);
endsequence : qError

// If req, then either successful completion or error flag is set
property pReqAckDoneFailOK;
  @(posedge clk)
    $rose(req) |-> qDoneSuccessful or qError;
endproperty : pReqAckDoneFailOK
```

☺

2. If assertion fails, then set new trigger for further properties 😐

This method makes use of the action block of an assertion. This is a more complex solution, however it can offer more flexibility in some cases. The idea is to cause the triggering of a signal when the assertion fails. That triggering signal is then used as an antecedent to another property to check that the expected condition (or subsequent sequence) succeeds. Conceptually, the following is defined:

```
property pReqAckDone;
  @ (posedge clk) req |-> ##[1:5] ack ##[1:100] done;
endproperty: pReqAckDone
```

Expected property without the error flag check

```
apReqAckDone : assert property (pReqAckDone) else begin
   ReqAckDoneFail <= 1'b1; // generate one pulse on signal ReqAckDoneFail
   @ (posedge clk) ReqAckDoneFail <= 1'b0;
   end

property pStatusRegOnError;
  @ (posedge clk)
   ReqAckDoneFail |=> status_reg == 1'b1;
endproperty : pStatusRegOnError

apStatusRegOnError : assert property (pStatusRegOnError);
```

Generate a new trigger to be used as an antecedent of another property

This consequent could be another sequence

Used in this property

No *done*, *status_reg* Set t

ONE

27 COND1 AND COND2 ARE MUTUALLY E

XCLUSIVE

the BUS REQUEST sequence occurs, then the DMA sec
onversely, if the DMA sequence occurs, then the BUS RI

quence should NOT occur.
EQUEST sequence should NOT occur.

BUS REQUEST sequence is *bus_req* followed within
nsfer of 1 to 128 words (*bus_enb*) followed by *done*.

5 cycles by *bus_ack* followed by a

DMA sequence is *dma_req* followed within 3 cycles by
8 data words (*mem_enb*).

dma_ack followed by the transfer of

vo mutually exclusive sequences imply an exclusive OR
us_Tranfer_Sequence **and not** DMA_Transfer_Sequenc
ot Bus_Tranfer_Sequence **and** DMA_Transfer_Sequenc

function, such as:
e) **or**
e)

e above definition cannot be directly expressed as a pr
a property, and does not apply to a sequence. Inst
itten. The sequence expression: "expression_or_dist t
press that the endpoint of a sequence should never
ogress.

perty because the **not** operator appli
ead, several distinct properties can
hroughout sequence_expr" is used
occur while the other sequence is

us transfer sequence: within 5 cycles bus_ack is follow
of 1 to 128 words (bus_enb) followed by done.
quence qBusTransfer;
##[1:5] bus_ack ##1 bus_enb [*1:128] ##1 done;
dsequence : qBusTransfer

d by a bus transfer

DMA transfer sequence: within 3 cycles dma_ack is fol
he transfer of 128 data words (mem_enb).
quence qDmaTransfer;
##[1:3] dma_ack ##1 mem_enb[*128];
dsequence : qDmaTransfer

owed by

```
// Never a trigger for both DMA and Bus transfers
property pBusXfr_XOR_DmaXfr;
  not ($rose(bus_req) or $rose(dma_req));
endproperty : pBusXfr_XOR_DmaXfr

// If a bus transfer, then no DMA sequence endpoint throughout the bus transfer
property pIfBusXfr_NoDmaXfr;
   $rose(bus_req) and !$rose(dma_req)
        |-> !qDmaTransfer.ended throughout qBusTransfer;
endproperty : pIfBusXfr_NoDmaXfr

// If a DMA transfer, then no bus sequence endpoint throughout the DMA transfer
property pIfDmaXfr_NoBusXfr;
   $rose(dma_req) and !$rose(bus_req)
       |-> !qBusTransfer.ended throughout qDmaTransfer;
endproperty : pIfDmaXfr_NoBusXfr
```

9.28 IF ADDRESS ERROR, THEN EVENTUALLY GOOD ADDRESS

If there is a single bit error on an address, them the same address eventually gets written to correct the error.

Preferred approach

```
property pAddrErrorRewrite;
 int v_addr; // local variable for addr
  (write && addr_error, v_addr=addr) |=> // if error, store address in variable
   ##[0:$] write && !addr_error && addr==v_addr;
assert property (pAddrErrorRewrite);
```

Another approach is to use an associative array, demonstrated below because it may have applications for other classes of properties.

Supporting code

```
int mem_aarray[*]; // associative array to be used by property
logic mem_aarray_exists; // To be used by property
// Methods are not allowed in properties
assign mem_aarray_exists = mem_aarray.exists(addr);

always@ (posedge clk)
  if (reset_n==1'b0) mem_aarray.delete;
  else if (addr_error) mem_aarray[addr] <= addr;

property pAddrErrorRewrite2;
  @(posedge clk)
  write && addr_error |=>
        ##[0:$] write && !addr_error && mem_aarray_exists &&
                addr==mem_aarray[addr];
endproperty : pAddrErrorRewrite2

a pAddrErrorRewrite2 : assert property (pAddrErrorRewrite2);
```

9.29 ENABLING A PROPERTY AFTER A TRIGGER

The EMPTY flag implies that there was a POP transaction in the previous cycle, which resulted in the empty flag.

At first glance, one may express this property as follows:

```
property p_empty_no_pop;
  @(posedge clk) empty |-> $past(pop);
endproperty : p_empty_no_pop
ap_empty_no_pop  : assert property (p_empty_no_pop);
```

However, the above assertion will fail after a reset because there was no previous *pop*. That condition can be is TRUE for several cycles if the FIFO is never exercised (i.e., no *push* or *pop* after reset). To address this concern, one could enable the assertion only after a reset. For example, if *reset* occurs only once at initialization, the following can be expressed:

```
initial
 begin : turn_checks_on_after_fst_push
  $assertoff (0, ap_empty_no_pop); // Using the assertion identifier
  @ (push === 1);
  $display("%0t %m Turning empty_2_pop check on ");
  $asserton (0, ap_empty_no_pop); // Using the assertion identifier
end
```

APPENDIX A
ANSWERS TO EXERCISES

A.1 ANSWERS TO CHAPTER 4 EXERCISES

1. **Define the following two named sequences:**
   a. qReq2Data: *read_req* followed within 4 cycles by *data_available.*
   b. qXmit2Done : *Start_xmit* followed after 8 cycles *done_xmit*

sequence qReq2Data; **$rose**(read_req) ##[1:4] data_available; **endsequence :** qReq2Data
sequence qXmit2Done; start_xmit ##8 done_xmit; **endsequence :** qXmit2Done

2. **Write a property that states that if the sequence qXmit2Done should occur within 10 cycles after the sequence qReq2Data.**

property Req2Done;
qReq2Data) |-> ##[1:10] qXmit2Done;
// equivalent to
*// **$rose**(read_req) ##[1:4] data_available|-> ##[1:10] start_xmit ##8 done_xmit;*
endproperty : Req2Done

3. **Write properties that states the following requirements:**
   a. If a new *BusReq* is acknowledged (*ack*) prior to a *timeout* signal, then the sequence *qXmit2Done* shall be initiated. *BusReq* must be asserted until either the *ack* or the *timeout* signal is asserted. Assume that *timeout* will not be asserted if *ack* is asserted.

b. If a new *BusReq* is not acknowledged (*ack*) prior to a *timeout* signal, then the *BusReq* shall be deasserted.

```
property Req2Done2 ;
    $rose(BusReq) |=:> (BusReq && ! timeout)[*0:$] ##1 ack) |-> qXmit2Done;
endproperty : Req2Done2

sequence qAck; ack; endsequence : qAck

property Req2NoAck;
 disable iff (qAck.triggered)
    $rose(BusReq) ##1 BusReq[*0:$] ##1 timeout |-> !BusReq;
endproperty : Req2NoAck
```

4. **Write a property that states the following requirements**: A buffer will be read before it is overwritten. Assume that the following signals are available:

rd	: read enable, active high
wr	: write enable, active high
addr	: write address

This requirement states that after a WRITE to an address, that data needs to be read from that address before it is re-written. Thus, this property requires a "before" property. SVA does not support the before operator. However, as mentioned in chapter 5 a reusable before property can be used instead.

```
property AbeforeB(a, b);
 (not (##[0:$] b)) or (!b[*0:$] ##1 (a && !b));
endproperty : AbeforeB

property FbeforeG(F,G);
   (not G) and (F or (1'b1 |=> FbeforeG(F,G)));
endproperty
```

The following property expresses the requirement that if a WRITE to an address occurs, then later, there should be one READ to the same address.

```
property read_after_write (write, read, addr) ;
  logic [31:0] v_waddr; // address to be stored at a write
  @(posedge clk)
  (write, v_waddr= addr)   // Found a write, indicate we started
       |=> FbeforeG (read && addr== v_waddr,
                          write && addr == v_waddr);
endproperty : read_after_write


assert property (read_after_write (wr, rd, addr))
          else $display ("Bad READ after WRITE");
```

5. **Write a property with the following requirements:**
   If a synchronous counter receives a load enable *(ld_n*, active low), and the reset signal is deasserted *(reset_n*, active false), then at the next clock cycle *the output of the counter will be equal to the load data input of the counter* (data_in).

```
property CounterLoad; // one option
  @ (posedge clk) !ld_n && reset_n |=> counter==$past(data_in);
endproperty : CounterLoad

property CounterLoad2; // Another option
 int v_counter;
  @ (posedge clk)
  (!ld_n && reset_n, v_counter=data_in) |=> counter==v_counter;
endproperty : CounterLoad2
```

6. **Explain why in defining the "antecedent" of an action, it is preferred to use the $rose or $fell functions for the signal in question, instead of the signal directly.**

Using the **$rose** or **$fell** prevents the generation of assertion errors caused by overlapping of the properties being tested at every cycle when the triggering condition is a level, rather than a unique condition. For example:
"a |=> (b ##1 c ##1 d);" would be started at every cycle whenever "a" is true.

However, "**rose**(a) |=> (b ##1 c ##1 d);" would be started at every cycle rising edge of "a", and is not sensitive to the level of "a". Thus, is "a" is true at cycle 1, 2, and 3 then the property would start only once at cycle 1, and would be vacuously TRUE at cycles 2 and 3.

7. **Provide an English description of this Property** :
   always $rose(a) |=> (b[*0:4] ##1 c ##1 !c[*10] ##1 d);

   If new *a* then starting from the next cycle, *b* is either omitted or may be asserted up to 4 cycles until *c*. *c* may occur in the cycle after *a*. Thus, *b* may either be omitted (i.e., repeated zero times), or repeated up to 4 times. After *c*, not c is asserted for 10 cycles, and then *d* is asserted. Thus, the following sequences represent examples that satisfy the property:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
a	b	b	c	!c	!c	!c	!c	!c	!c	!c	!c	!c	!c	d		
a	c	!c	!c	!c	!c	!c	!c	!c	!c	!c	!c	d				
a	b	b	b	b	c	!c	!c	!c	!c	!c	!c	!c	!c	!c	!c	d

8. **Explain why this Property can never fail. Also, simplify this property to lower terms.**

```
property X8;
  a |=> ##[0:$] ##1 b[*0:4] ##1  c ##1 !c[*10] ##1 d;
endproperty : X8
```

The property can never fail because if the condition following the ##[0:$] fails, the property remains active, waiting for the sequence to the right of the ##[0:$] to complete. Property can only terminate, which can provide functional coverage. The property expression can be simplified to the following because *b* need not necessarily repeat (the "[*0" expresses that).

a |=> ##[0:$] c ##1 !c[*10] ##1 d;

9. **Explain the meaning of these assertions:**
a |=> (b[*3] ##1 c);
Satisfied with sequence "a ##1 **b** ##1 **b** ##1 **b** ##1 c"

a |=> (b[->3] ##1 c);
Satisfied with sequence "a ##1 **b** ##1!b ##1 **b** ##1!b ##1 !b ##1 **b** ##1 c"

a |=> (b[=3] ##1 c);
Satisfied with sequence "a **##1 b** ##1!b ##1**b** ##1!b ##1 !b ##1 **b ##1** !b ##1 **c**"

a |=> (`true[*3] ##1 b ##1c));
Satisfied with sequence "a **##1** `true ##1 `true ##1 `true **##1 b ##1 c**"
// equivalent to:
a |=> ##4 b ##1c;

10. Write a property that states that if there is a bus request (*bus_req)* followed by no acknowledge (*ack*) for 10 cycles, or a DMA request (*dma_req*) followed by no memory release (*mem_in_tristate*) in 5 cycles, then an error interrupt (*err_intrpt*) shall be asserted.

```
sequence qBreqNoAck; $rose(bus_req) ##1 !ack[*10]; endsequence : qBreqNoAck
sequence qDmaReqNoTriState; $rose(dma_req) ##1 !mem_in_tristate[*5];
endsequence : qDmaReqNoTriState
property BreqDmaReq2Error;
    (qBreqNoAck | qDmaReqNoTriState) |-> (##[0:$] err_intrpt);
```

11. **Describe the meaning of this property using text and timing description, showing the order of transactions** (i.e., *send_cmd, header_enb,* etc*)* **:**

```
property DataTransfer;
 (send_cmd) |=>
     ##[0:10] header_enb ##1 (data_enb)[*32] ##1 crc_enb;
endproperty : DataTransfer
```

If a send command *send_cmd* is received, then either at the next cycle or up to 10 cycles later *header_enb* is asserted. This is then followed by 10 consecutive assertions of *data_enb* and then *crc_enb*. Basically, the timing for the data looks as follows:

	Send	Header	Data	CRC
	send_cmd	*header_enb*	*data_enb*	*crc_enb*
Cycle Duration	1	1	32	1

12. **Define a waveform for this property**
With the following waveform, what is the value of the assertion at time 100 ns, 150 ns, 200 ns?
property P; @ (**posedge** clk) a |=> b |=> c; **endproperty** : P
assert property (P);

Considering the following property expression: sequence_expr |=> property_expr

That expression states that if *sequence_expr* is TRUE then *property_expr* must be TRUE for the property to hold. Now, consider the following property:

property XthenY(x, y); x |=> y; **endproperty**
property P2; a |=> XthenY(b, c); **endproperty**

//Note that property *P2* is identical to property *P*

// **property** P; @ (**posedge** clk) a |=> b |=> c; **endproperty** : P
assert property (P2);

With the test waveform, *a==1* at time 100 ns. Thus at time 100 ns, the antecedent is TRUE and the assertion is in the "active" state. Therefore, for the assertion to pass, property *XthenY(b, c)* must be TRUE.

At the next cycle, time 150 ns, *b==0*. With *b==0*, the property *XthenY(b, c)* is vacuously TRUE since the antecedent of *XthenY(b, c)* is zero. Since the consequent of property *P2* is vacuously TRUE, at time 150 ns, then property *P2* is vacuously TRUE.

At time 200 ns, *a==0*. Since the antecedent of *P2* is FALSE, property *P2* is vacuously TRUE.

13. **Why are properties named?**
A property may be applicable in more than one part of the design. Naming a property allows for the definition of the property once. A named property can also be parameterized for reuse. A named property can then be referred wherever the property needs to be asserted , assumed, or instantiated within another property. In addition, named properties are easier to debug as their names appear in the debug trace, assertion browser etc. For example:

```
property pNeverAllGreen;
  @ (posedge clk) not (main_light==GREEN && side_light==GREEN);
endproperty : pNeverAllGreen
```

Label assertions to help in debug

```
apNeverAllGreen : assert property NeverAllGreen;
```

14. Write a property that expresses the following requirement:

"If an FSM (*fsm_reg*) is in IDLE state and it receives a FIRE command (*fire*, active high), then at the next clock cycle the FSM will enter the ARM state, and the cycle after that the FSM will enter the IGNITE state. The FSM will then return to the IDLE state. Assume that the states of the FSM were predefined to be IDLE, ARM, IGNITE.

```
// RTL style of writing properties
// generally not a recommended style because it emulates RTL
property pFSM_Pyro;
  @ (posedge clk)   fsm_reg==IDLE && fire |=> fsm_reg==ARM
            |=> fsm_reg==IGNITE |=> fsm_reg==IDLE;
endproperty : pFSM_Pyro

Another way to express the same concept can be written as follows with the cycle delays:
property pFSM_Pyro2;
   @ (posedge clk) fsm_reg==IDLE && fire ##1  fsm_reg==ARM  ##1
            fsm_reg==IGNITE |=> fsm_reg==IDLE;
endproperty : pFSM_Pyro2

// Better style because if defines that given a state, it identifies possibilities that
//  led to that state.
property p_event_led_to_Ignite;
  @ (posedge clk) fsm_reg==IGNITE |->
      $past(fsm_reg) == ARM &&
      $past(fsm_reg, 2)== IDLE &&
      $past(fire, 2);
endproperty : p_event_led_to_Ignite

//In a review of the above example, another possibility that can cause the IGNITE could
// be brought up.  For example, an emergency ignite_override signal may be necessary
// for the design.  The above property can then be rewritten as:
property p_event_led_to_Ignite2;
  @ (posedge clk) fsm_reg==IGNITE |->
      ( $past(fsm_reg) == ARM &&   $past(fsm_reg, 2)== IDLE &&
      $past(fire, 2))  or  $past(ignite_override);
endproperty : p_event_led_to_Ignite2
```

Another strategy would be to use sequences and methods. Specifically:

```
sequence qIdle2Arm;
  @ (posedge clk) fsm_reg==IDLE && fire ##1 fsm_reg==ARM ##1 `true;
endsequence : qIdle2Arm

property p_event_led_to_Ignite3;
  @ (posedge clk) fsm_reg==IGNITE |->
      qIdle2Arm.ended or
      $past(ignite_override);
endproperty : p_event_led_to_Ignite3
```

Write a property that states that there should never be more than one *grant* (i.e., the *grant* bus can either be all ZEROs, or at most have one bit asserted).

```
property pGrantValues;  // option 1
    @ (posedge clk) grant==8'b0 || $countones(grant)==1;
endproperty : pGrantValues

property pGrantValues2  // option 2  ☺
// $onehot0(<expression>) returns true if at most one bit of the expression is high.
  @ (posedge clk) $onehot0 (grant);  //  expected condition at every cycle
endproperty : pGrantValues
```

15. **Write a coverage property that states that an occurrence of an active *grant* has transpired.**

```
property pGrant_coverage;
  @ (posedge clk) grant;
endproperty : pGrant_coverage

cpGrant_coverage : cover property (pGrant_coverage);  // option 1
cGrant_coverage cover property (@ (posedge clk) grant);  // option 2
```

16. Explain in English the meaning of the following properties:

```
sequence qAck; ##3 ack; endsequence : qAck
property pReqAck;  req |-> ##3 ack; endproperty : pReqAck
property pAckThenPrevReq;
  qAck.ended |-> $past(req, 3);
endproperty : pAckThenPrevReq
```

If *req* then at the 3rd cycle, *ack* is asserted. Thus:

req	1			
ack	-	-	-	1

Property *pAckThenPrevReq* states that if a sequence of 3 cycles of anything, followed by *ack*, then a *req* was asserted in the past 3 cycles.

b. **property** pReq3Acks; req |=> ack[*3] ; **endproperty** : pReq3Acks

If *req* then *ack* holds from next cycle through next 3 cycles.

req	1			
ack		1	1	1

c. **property** pReqAckOnce;
req |=> ack[->1] **within** `true[*3];
endproperty : pReqAckOnce

If *req* then *ack* holds at least once from next cycle until next 3 cycles.

req	1			
ack		-	1	-

d. **property** pReqAckDone;
req ##1 ack[->2] ##0 data_xfr |=> done;
endproperty : pReqAckDone

If *req* followed by 2nd occurrence of *ack*, then *data_xfr* followed by *done*

req	1						
ack			1	0	1		
data_xfr	-	-	-	-	1		
done						1	

e. **property** pReqAcksDataXFrs;
req |=> (ack && data_xfr) [=1:31] **within** `true[*40];
endproperty : pReqAcksDataXFrs

If *req* then anytime between 1 to 31 occurrences of *ack && data_xfr* within the next 40 cycles.

cycle	1	2	3	...	30	31
req	1					
ack		0	1	-	1	0
data_xfr	-	-	1	-	1	-

f. **property** pReqAck100DataXfr;
req |=> (##[0:$] data_xfr[*16]) **within** ack[=1:100];
endproperty : pReqAcksDataXFrs

If *req* then anytime between 1 to 100 occurrences of *ack*, there is at least one occurrence of a block of 16 data transfers (*data_xfr*).

cycle	1	2	3	4	...	18	...100
req	1						
ack		0	1	-	-	-	0
data_xfr	-	-	1	1	1	1	-

17. Write assertions for the following requirements:

a) If the transmit FIFO is not empty, and no data output transmission is in progress, then a new transmission shall be started within 5 clock cycles. Assume that the following signals are available:
fifo_size, xmit_status (0==no transmission, 1== in transmission mode).

b) When a transmission of data is completed, as indicated but the deassertion of the *xmit*_status signal, then a CPU interrupt shall be asserted within 4 clock cycles.

```
property pNewXmissionOnMt;
   (fifo_size!=0 && !xmit_status) |-> ##[1:5] xmit_status;
endproperty : pNewXmissionOnMt
```

note: Multiple overlaps of the antecedent (the left-hand side) can be satisfied by a single occurrence on *xmit_status*.

```
property pDataDoneInterrupt;
   $fell(xmit_status) |-> ##[1:4] interrupt;
endproperty : pDataDoneInterrupt
```

A.2 ANSWERS TO CHAPTER 6 EXERCISE

1. Parking structure controller

Write properties that define the following requirements. A package and an interface are provided below.

```
/* Problem Statement[1]
The example FSM controls a FIFO Queue but to be practical, consider that we
want to design a simple digital controller that monitors cars entering and
leaving a parking lot. The operation specs are as follows.
  * The controller maintains the current COUNT of the total number of cars
    parked in the lot.
  * The lot has a limited number of parking spots, for simplicity sixteen (16).
  * There are two input sensors at the entrance and exit of the lot.
    The sensors provide input signals to the controller as cars pass by and
    the controller increments/decrements the current COUNT. What if two cars
    enter and exit simultaneously?
  * If COUNT=16, then the controller issues a FULL output signal,
    otherwise the OPEN output signal is on.
  * There is a manual input for the lot attendant to override the COUNT
    for special vehicles.
  * A START/STOP input is also available to the attendant, with the STOP
    input causing a CLOSED output signal.
*/
```

Figure A2-1 is a pictorial view of the parking lot.

Figure A2-1 Parking Lot overview

[1] Requirements extracted from http://bear.ces.cwru.edu/eecs_316/
Instructors: Prof. Chris Papachristou, Francis Wolff

```
// The steps involved in translating these requirements to SVA include the following:
// 1.    Define objects and types for these objects
// 2.    Define interfaces for module.
// 3.    Map English requirements to SVA.  Method used for this application is first copying each
// English requirement into the module as comments, and then mapping the requirements to SVA.
// // Define objects and types for these objects
package parking_pkg;
  timeunit 1ns;
  timeprecision 100ps;
  // Type definitions
  typedef enum bit[1:0] {NEUTRAL, STOP, START} gate_override_t;
  typedef enum bit {CLOSED, OPEN} gate_t;
  typedef enum bit {ENABLE_COUNTER, BYPASS_COUNTER} counter_cntrl_t;
endpackage : parking_pkg

// Define interfaces for module.
// compilation-unit scope
import parking_pkg:: *;
interface parking_structure_if (input wire clk, reset_n);
  timeunit 1ns;
  timeprecision 100ps;
  gate_t            gate_status;
  bit               parking_full;
  bit               entrance_sensor;
  bit               exit_sensor;
  gate_override_t   gate_override;
  counter_cntrl_t   counter_cntrl;

  modport parking_cntlr_if (
   output gate_t            gate_status,
   output bit               parking_full,
   input  bit               entrance_sensor,
   input  bit               exit_sensor,
   input  gate_override_t   gate_override,
   input  counter_cntrl_t   counter_cntrl);
endinterface : parking_structure_if

// compilation-unit scope
import parking_pkg::*;
//Module definition
module parking_structure (input clk,
                          input reset_n,
                          parking_structure_if.parking_cntlr_if park_if);
  timeunit 1ns;
  timeprecision 100ps;
// Object definition that relate to requirements
//  * The controller maintains the current COUNT of the total number of cars
//    parked in the lot.
//  * The lot has a limited number of parking spots, for simplicity sixteen (16).
  logic [4:0] car_counter; // counter of parked cars
  parameter MAX_SPOTS = 16;
```

```
// A START/STOP input is also available to the attendant,
// with the STOP input causing a CLOSED output signal.
// use the gate_override
// * There are two input sensors at the entrance and exit of the lot.
//   The sensors provide input signals to the controller as cars pass by and
//   the controller increments/decrements the current COUNT. What if two cars
//   enter and exit simultaneously?
// * If COUNT=16, then the controller issues a FULL output signal,
//   otherwise the OPEN output signal is on.
//What if two cars enter and exit simultaneously?
 property p_parking_full_to_count_chk;
  @ (posedge clk) car_counter == MAX_SPOTS |-> park_if.parking_full;
 endproperty : p_parking_full_to_count_chk
 ap_parking_full_to_count_chk : assert property p_parking_full_to_count_chk;

// From Spec: Otherwise the OPEN output signal is on.
// * There is a manual input for the lot attendant to override the COUNT
//   for special vehicles.
// * A START/STOP input is also available to the attendant, with the STOP
//   input causing a CLOSED output signal.
 property p_gate_open_status_chk;
  @ (posedge clk) p_park_if.gate_status == OPEN
  |-> (car_counter < MAX_SPOTS || $past(park_if.gate_override) == START);
 endproperty : gate_open_status_chk
 ap_gate_open_status_chk : assert property p_gate_open_status_chk;

 property p_gate_closed_status_chk;
  @ (posedge clk) park_if.gate_status == CLOSED
  |-> (car_counter == MAX_SPOTS || $past(park_if.gate_override) == STOP);
 endproperty : p_gate_closed_status_chk
 ap_gate_closed_status_chk : assert property p_gate_closed_status_chk;

// Value change in the car counter
 property p_counter_val _change_chk;
        @ (posedge clk) (car_counter != $past(car_counter))
     |-> park_if.counter_cntrl==ENABLE_COUNTER && // control is enabled
      !($past(park_if.entrance_sensor) && $past(park_if.exit_sensor))
            //No simulaneous entrance and exit
      ($past(park_if.entrance_sensor) || $past(park_if.exit_sensor));
                                                  // either an entrance or an exit
 endproperty : p_counter_val_change_chk
 ap_counter_val_change_chk : assert property p_counter_val_change_chk;
```

```
// Counter has no change
 property p_counter_val_no_change_chk;
  @ (posedge clk) (car_counter == $past(car_counter)) |->
   !($past(park_if.entrance_sensor) or $past(park_if.exit_sensor)) or // no entrance or exit
    ($past(park_if.entrance_sensor) && $past(park_if.exit_sensor)) or
                                                    // simulaneous entrance and exit
    park_if.counter_cntrl==BYPASS_COUNTER; // in BYPASS mode
 endproperty p_counter_val_no_change_chk;
 ap_counter_val_no_change_chk : assert property p_counter_val_no_change_chk;

 // Overflow detection
 property p_counter_overflow;
   @ (posedge clk) not(car_counter == MAX_SPOTS + 1);
 endproperty : p_counter_overflow
 ap_counter_overflow_1 : assert property p_counter_overflow else
        $display("\n %m counter overflew Prev. value %d Cur Value %d ",
                $past(car_counter), car_counter);
endmodule // parking_structure
```

Handshake Model

Write properties that define the following requirements for a slave unit that acquires a bus through a handshake protocol. The protocol requirements can be summarized as follows:

1. The slave issues a bus request *(req)*, and holds that request until master asserts the acknowledge *(ack)* signal, at which time the slave will deassert the bus request.
2. The master replies to bus request with a single acknowledge cycle, asserted within four cycles from the bus request.
3. In response to *ack* signal, the slave issues
   a. *Start* for 1 cycle, used for synchronization purposes by master.
   b. *Valid* for 4 cycles, used for data envelope. The *valid* signal occurs in the same cycle as the *start.* Data is not within the design of this slave.
   c. *Done* for 1 cycle, used for synchronization purposes by master. The *done* signal occurs 1 cycle after the last cycle of *valid.*
4. An active *reset* signal aborts any handshake operation.
5. All operations are synchronous to the system clock.

Figure A2-2 demonstrates a typical timing for the handshake protocol.

Figure A2-2 Handshake Protocol Timing Example

```
module slave_handshake
  (
  output logic     req    ,     // bus request
  output logic     start  ,     // start of data cycle
  output logic     valid  ,     // doing work
  output logic     done   ,     // to arbiter
  input            ack    ,     // acknowledge
  input            reset_n ,     // reset active low
  input            clk    );
   timeunit 1ns;
   timeprecision 100ps;

  typedef enum {IDLE, REQUEST, ACQUISITION} hmode_t; // handshake modes
  parameter ERROR=1;
  parameter FAIL=0;
  `define true 1

hmode_t state_r ;          // State reg
logic [0:10] counter;      // count # valid cycles

//*************************************************
// Protocol Requirements
// 1. slave issues a req, and holds request until ack
// 2. Master to reply with 1 cycle ack within 3 cycles
// 3. In response to ack, slave issues
//    start for 1 cycle, used for sync purposes
//    valid for 4 cycles, used for data envelope
//    done for 1 cycle, used for sync
// 4. a reset or a cancel aborts any operation
//*************************************************
//        ASSERTIONS

property p_req_ack_eventually;
  disable iff (!reset_n)  req && !ack |-> ## [0:$] ack;
endproperty : p_req_ack_eventually

always @(posedge clk) // always SystemVerilog block
```

```
// Demontration if a different style to assert a property
  ap_req_ack_eventually_1: assert property (p_req_ack_eventually);

sequence qActive_request;(req && !ack); endsequence

property p_req_ack_eventually_in_4_cycles;
 @ (posedge clk) disable iff (!reset_n)  qActive_request |-> ##[0:3] ack;
endproperty : p_req_ack_eventually_in_4_cycles
ap_req_ack_eventually_in_4_cycles: assert property (p_req_ack_eventually_in_4_cycles);

// req cannot be asserted in a cycle after the ack.
// In addition, there can only be one single acknowledge.
// There cannot be 2 sequential ack
  property p_no_req_after_ack;
   @ (posedge clk) disable iff (!reset_n) not (qActive_request ##1 ack ##1 (req || ack));
  endproperty : p_no_req_after_ack
  ap_no_req_after_ack : assert property (p_no_req_after_ack);

property p_ack_start_valid_done;
@ (posedge clk) disable iff (!reset_n)
    qActive_request ##1 ack |=>
      start && !ack && valid ##1
     (!start && !ack && valid)[*3] ##1
     !start && !valid && done;
endproperty : p_ack_start_valid_done
ap_ack_start_valid_done : assert property (p_ack_start_valid_done);

sequence qValid; valid [*4]; endsequence : qValid
 // If in IDLE, then previous cycle must have been
 //      either a reset or the end of a valid sequence.
 // If in REQUEST, then previous cycle must have been a request with state in IDLE.
 // If in ACQUISITION, then previous cycle must have been
 //      either in REQUEST state and assertion of acknowledge
 //      or in ACQUISITION state and the end of the valid sequence was reached.
  //**** TEST OF CLOCKING
property p_handshake;
  @ (posedge clk) disable iff (!reset_n)
  if (state_r==IDLE) |-> $past(reset_n) || $past(qValid.ended);
  else if (state_r==REQUEST) |-> $past(req) && $past(state_r==IDLE);
  else if (state_r==ACQUISITION) |-> ($past(state_r==REQUEST) && $past(ack)) or
                          $past(state_r==ACQUISITION) && !$past(qValid.ended);
 endproperty : p_handshake
 ap_handshake :  assert property (p_handshake);

// Request to acknowledge coverage, NO failure
property p_req_ack_coverage;
  @ (posedge clk)  disable iff (!reset_n)
 qActive_request ## [0:3] ##1 ack |-> `true;
endproperty : p_req_ack_coverage
cp_req_ack_coverage : cover property (p_req_ack_coverage);
```

```
  // Occurrence of a start and valid
  property p_start_valid_coverage;
   @ (posedge clk) disable iff (!reset_n)
     qActive_request |=> ##[0:$] start ##1 valid;
  endproperty : p_start_valid_coverage
  cp_start_valid_coverage : cover property(p_start_valid_coverage);

always @ (posedge clk, negedge reset_n) begin : ack_prc
   if (reset_n == 1'b0) begin          // asynchronous reset (active low)
      state_r <= IDLE;
      counter <= 0;
      req   <= 1'b0;
      start <= 1'b0;
      valid <= 1'b0;
      done  <= 1'b0;
   end
   else  // rising clock edge
      case (state_r)
        IDLE: begin
          req   <= 1'b1;
          state_r <= REQUEST;
          done  <= 1'b0;
          counter <= 0;
        end
        REQUEST: begin
          if (ack == 1'b1) begin
           req   <= 1'b0;
           start <= 1'b1;
           valid <= 1'b1;
           state_r <= ACQUISITION;
          end
        end
       ACQUISITION : begin
           counter <= (counter+1) % 4;
           req   <= 1'b0;
           start <= 1'b0;
           valid <= 1'b1;
           if (counter == (3 - ERROR)) begin
            done  <= 1'b1;
            valid <= 1'b0;
            state_r <= IDLE;
           end
          end
      endcase
 end // ack_prc
endmodule // slave_handshake
```

Appendix B: Definitions[1]

assertion: A statement that a given property is required to hold and a directive to verification tools to verify that it does hold. Assertions allow for automated checking that the specified property is true, and can generate automatic error messages if the property is not true. An assertion is a "statement about a design's intended behavior" (From *Assertion-Based Design*, Foster). A property is a characteristic of the design. The distinction between the two terms is marginal, and the two terms are often used interchangeably

Assertion-Based Verification: A verification technique that involves instrumenting the design with assertions to provide added visibility and white-box testing into its internal state.

assumption: A statement that the design is constrained by the given property and a directive to verification tools to consider only paths on which the given property holds.

BFM : A Bus Functional Model that emulates the operation of an interface (i.e., the bus), but not necessarily the internal operation of the interface.

bind: Links a program (or module or interface) to a module or an instance. The linked program (that may include properties) becomes part of the bound object.

behavior: A path.

Boolean: A Boolean expression, for example a==4'b1101. Variables of length one bit are processed by SystemVerilog Assertions as Boolean object (e.g., b |=> c ##[0:4] d).

Boolean expression: An expression that yields a logical value. The expressions used in sequences are evaluated over sampled values of the variables that appear in the expressions. The outcome of the evaluation of an expression is Boolean and is interpreted the same way as an expression is interpreted in the condition of a procedural **if** statement. That is, if the expression evaluates to X, Z, or 0, then it is interpreted as being FALSE. Otherwise, it is TRUE.

checker: An auxiliary process (usually constructed as a finite state machine) that monitors simulation of a design and reports errors when asserted properties do not hold. A checker may be represented in the same HDL code as the design or in some other form that can be linked with a simulation of the design.

completes: A sequential expression (or property) completes at the last cycle of any design behavior described by that sequential expression (or property).

---

[1] Many of these definitions were extracted from the Accelera PSL and SystemVerilog LRMs.

compilation-unit scope: A scope that is local to the compilation unit.. A collection of one or more SystemVerilog source files compiled together. It contains all declarations that lie outside of any other scope.

computation path: A succession of states of the design, such that the design can actually transition from each state on the path to its successor.

constraint: A condition (usually on the input signals) that limits the set of behavior to be considered. A constraint may represent real requirements (e.g., clocking requirements) on the environment in which the design is used, or it may represent artificial limitations (e.g., mode settings) imposed in order to partition the verification task.

Constant — There are two types of constants in SystemVerilog: elaboration and run-time constants. Parameters and local parameters are elaboration constants. Their values are calculated before elaboration is complete. Elaboration constants can be used to set the range of array types. The other form of constant is a run-time constant. These are variables that can only be set in an initialization expression using the **const** qualifier. A run-time constant cannot be used in properties for cycle delays or repeat values.

count: A number or range.

coverage: The percentage of verification objectives that have been met. It is used as a metric for evaluating the progress of a verification project in order to reduce the number of simulation cycles spent in verifying a design. A measure of the occurrence of certain behavior during (typically dynamic) verification and, therefore, a measure of the completeness of the (dynamic) verification process. (Also see functional coverage and test coverage).

cycle: An evaluation cycle.

deadlock : Two (or more) FSMs waiting for each other to do something.

describes: A Boolean expression, sequential expression, or property describes the set of behavior for which the Boolean expression, sequential expression, or property holds.

design: A model of a piece of hardware, described in some hardware description language (HDL). A design typically involves a collection of inputs, outputs, state elements, and combinational functions that compute next state and outputs from current state and inputs.

design behavior: A computation path for a given design.

DUT: Design Under Test.

DUV: Design Under Verification.

dynamic verification: A verification process in which a property is checked over individual, finite design behavior that are typically obtained by dynamically exercising the design through a finite number of evaluation cycles. Generally, dynamic verification supports no inference about whether the property holds for a behavior over which the property has not yet been checked.

equivalence checking : Verification that two representations of the same design are equivalent.

evaluation: The process of exercising a design by iteratively applying values to its inputs, computing its next state and output values, advancing time, and assigning to the state variables and outputs their next values.

evaluation cycle: One iteration of the evaluation process. At an evaluation cycle, the state of the design is recomputed (and may change).

extension: An extension of a path is a path that starts with precisely the succession of states in the given path.

false: An interpretation of certain values.

- The single bit value 1'b0 is interpreted as the logical value FALSE.
- The result of a Boolean comparison that fails
- A vector has all of its bits with a value of zero or has bits set to X or Z.

finite range: A range with a finite high bound (e.g., [0:4]). This contradicts an infinite range with the high bound set to $ (e.g., [0:$])

formal verification: A verification process in which analysis of a design yields a logical inference about whether the properties hold for all behavior of the design.

- If a property is declared TRUE by a formal verification tool, no simulation can show it to be FALSE.
- If the property does not hold for all behavior, then the formal verification process should provide a specific counterexample to the property, if possible.

Functional coverage: A user-defined metric that measures how much of the design specification, as enumerated by features in the test plan, has been exercised. It can be used to measure whether interesting scenarios, corner cases, specification invariants, or other applicable design conditions have been observed, validated and tested.

holds: A term used to talk about the meaning of a Boolean expression, sequential expression or property. Loosely speaking, a Boolean expression, sequential expression, or property holds in the first cycle of a path if and only if the path exhibits the behavior described by the Boolean expression, sequential expression, or property.

holds tightly: A term used to talk about the meaning of a sequential expression (SERE). Sequential expressions are evaluated over finite paths (behavior). Loosely speaking, a sequential expression holds tightly along a finite path if and only if the path exhibits the behavior described by the sequential expression.

identical clocks: Clocks are considered identical if they are syntactically identical. Thus, their waveforms are exactly the same under all circumstances.

Examples of identical clocks include:

- The same clock signal, whether referenced by the local name or a hierarchical path name.
- Clock names aliased to each other via the net alias statement.

Examples of non-identical clocks caused by post-synthesis timing include:
These examples are possibly caused by post-synthesis timing of clocks where added delays would make the clocks different.

- Clock names where one clock is derived from the other clock by passing through a primitive (such as *buf* or *tran*).
- Clocks names that are connected together via module ports.
- Clock names where one clock is continuously assigned the value of another clock.

interface: An interface encapsulates the communication between blocks of a design, allowing a smooth migration from abstract system-level design through successive refinement down to lower-level register-transfer and structural views of the design. By encapsulating the communication between blocks, the interface construct of SystemVerilog also facilitates design reuse. Properties can be defined in interfaces.

IP: An Intellectual Property, meaning either a hardware device (such as a PCI interface design, or a module), or a piece of software.

liveness property: A property that specifies an eventuality that is unbounded in time. Loosely speaking, a liveness property claims that "something good" eventually happens. More formally, a liveness property is a property for which any finite path can be extended to a path satisfying the property. For example, the property "whenever signal *req* is asserted, signal *ack* is asserted some time in the future" is a liveness property.

logic type: An HDL data type that includes values that are interpreted as logical values. A logic type may include both logical values and metalogical values. Such a logic type usually represents a multi-valued logic.

logical value: A value in the set {TRUE, FALSE}.

LRM — LRM is an abbreviation for Language Reference Manual, such as the *SystemVerilog LRM*..[2] The *Verilog LRM* refers to the IEEE manual *1364-2001 IEEE Standard for Verilog Hardware Description Language 2001*.

metalogical value: A value of a (multi-valued) logic type that is not interpreted as a logical value, such as X, Z.

model checking: A type of formal verification.

monitor: *See:* checker.

natural: A non-negative integer value that includes zero.

number: A non-negative integer value, and a statically computable expression yielding such a value.

occurs, occurrence: A Boolean expression is said to "occur" in a cycle if it holds in that cycle. For example, "the next occurrence of the Boolean expression" refers to the next cycle in which the Boolean expression holds.

[2] http://www.systemverilog.org/home.html

path: A succession of states of the design, whether or not the design can actually transition from one state on the path to its successor.

positive count: A positive number or a positive range.

positive number: A number that is greater than zero (0), as opposed to a **natural** number that includes zero.

positive range: A range with a low bound that is greater than zero (0).

prefix: A prefix of a given path is a path of which the given path is an extension.

property: A collection of logical and temporal relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behavior. A property is a characteristic of the design. An assertion is a "statement about a design's intended behavior" (From *Assertion-Based Design*, Foster). The distinction between the two terms is marginal, and the two terms are often used interchangeably

range: A series of consecutive numbers, from a low bound to a high bound, inclusive, such that the low bound is less than or equal to the high bound. In particular, this includes the case in which the low bound is equal to the high bound. Also, a pair of statically computable integer expressions specifying such a series of consecutive numbers, where the left expression specifies the low bound of the series, and the right expression specifies the high bound of the series.

required (to hold): A property is required to hold if the design is expected to exhibit behavior that is within the set of behavior described by the property.

restriction: A statement that the design is constrained by the given sequential expression and a directive to verification tools to consider only paths on which the given sequential expression holds.

safety property: A property that specifies an invariant over the states in a design. The invariant is not necessarily limited to a single cycle, but it is bounded in time. Loosely speaking, a safety property claims that "something bad" does not happen. More formally, a safety property is a property for which any path violating the property has a finite prefix such that every extension of the prefix violates the property. For example, the property, "whenever signal *req* is asserted, signal *ack* is asserted within 3 cycles" is a safety property.

scoreboard: In verification, a scoreboard is a data structure, independent from the design under test, used to store expected results for later comparisons against actual results. Scoreboards can be records (e.g., **struct**), queues, associative arrays, dynamic arrays. Assertions can access scoreboards in the verification process.

scoreboarding: The process of using a scoreboard.

sequence: A linear sequence is a finite list of SystemVerilog Boolean expressions in a linear order of increasing time. The linear sequence is said to match along a finite interval of consecutive clock ticks provided the first Boolean expression evaluates to true at the first clock tick, the second Boolean expression evaluates to true at the second clock tick, and so forth, up to and including the last Boolean expression evaluating to true at the last clock tick. A single Boolean expression is an example of a simple linear sequence, and it matches at a single clock tick provided the Boolean expression evaluates to true at

that clock tick. More complex sequential behaviors are described by SystemVerilog sequences. A sequence is a regular expression over the SystemVerilog Boolean expressions that concisely specifies a set of zero, finitely many, or infinitely many linear sequences. If at least one of the linear sequences from this set matches along a finite interval of consecutive clock ticks, then the sequence is said to match along that interval.

sequential expression: A finite series of terms that represent a set of behavior.

SERE: A sequential expression (usually addressed in PSL).

signal A signal is an informal term, usually meaning either a variable or net. The context where it is used may imply further restrictions on allowed types.

simulation: A type of dynamic verification.

Static variable: A variable whose value is known at elaboration time, and cannot be modified during simulation.

static verification: A process to analyze a design statically without simulation. Static verification may include topological verification including structural and language (e.g., linting) or formal verification.

starts: A sequential expression starts at the first cycle of any behavior for which it holds. In addition, a sequential expression starts at the first cycle of any behavior that is the prefix of a behavior for which it holds. For example, if *a* holds at cycle 7 and *b* holds in every cycle from 8 onward, then the sequential expression {a; b[*] ;c} starts at cycle 7.

strictly before: Before, and not in the same cycle as.

strong operator: A temporal operator, the (non-negated) use of which creates a liveness property.

terminal state : A condition where there is no way out of the state under any condition,

terminating condition: A Boolean expression, the occurrence of which causes a property to complete.

terminating property: A property that, when it holds, causes another property to complete.

test coverage: A measure of manufacturing defects (that will be detected on the manufactured devices). This differs from functional coverage or code coverage that is considered in the "Design Verification" phase.

thread: A related sequence of events that may last for one or more cycle to represent a design property. For example, a bus request that is acknowledged within 1 to 5 cycles is a thread. A new thread may be started at every cycle, and every thread is independent from other threads that may have started at earlier cycles.

transaction : Tasks and parameters that need to be executed.

true: An interpretation of certain values of certain data types.

- The single bit value 1'b1 is interpreted as the logical value TRUE.
- The result of a Boolean comparison that succeeds.
- A vector that is non-zero, with a known value (i.e., no bits set to X or Z).

vacuity: A property is vacuous if it lacks serious purpose. Vacuity rules are usually applied when implication operator is used. A property succeeds vacuously if the antecedent is FALSE. A property succeeds non-vacuously only if the consequent of the implication contributes to its success. Many applications have a triggering event that causes the thread to start (in the antecedent), and if the triggering event does not occur, then there is no need to further evaluate the thread (i.e., the rest of the property for that cycle), which is then considered vacuously TRUE.

verification: The process of confirming that, for a given design and a given set of constraints, a property that is required to hold in that design actually does hold under those constraints. Kropf defines "validation" as the "process of gaining confidence in the specification by examining the behavior of the implementation."[3] Recently there was discussion on the subject of "verification versus. validation" in the on-line Verification Guild.[4] Many views were presented regarding the difference. One view was that "validation ensures it is the right design, while verification ensures that the design is right." Another view was "verification means pre-silicon testing (Verilog/VHDL simulations) while validation is post-silicon testing (testing silicon on boards in the lab)."

weak operator: A temporal operator, the (non-negated) use of which does not create a liveness property.

Acronyms and abbreviations

BNF	extended Backus-Naur Form
CTL	computation tree logic
EDA	electronic design automation
FSM	finite state machine
HDL	hardware description language
iff	if and only if
LTL	linear-time temporal logic
PSL	Property Specification Language
RTL	Register Transfer Level
SERE	Sequential Extended Regular Expression
SVA	SystemVerilog Assertions
VHDL	VHSIC Hardware Description Language

[3] Kropf, T.: Formal hardware verification: Methods and systems in comparison. Springer, 1997
[4] http://www.verificationguild.com/

APPENDIX C: QUICK REFERENCE GUIDE

Declarations	Example
property property_identifier [([list_of_formals])] ; assertion_variable_declaration property_spec ; **endproperty** [: property_identifier]	**property** pReadyAck(clk, reset_n, ready, ack, data, data_reg); **logic** [15:0] v_data; @ (**posedge** clk) **disable iff** (!reset_n) (**$rose**(ready), v_data=data) |=> ##[0:3] ack && data_reg=v_data; **endproperty** : pReadyAck
sequence sequence_identifier [([list_of_formals])] ; [assertion_variable_declaration] sequence_expr ; **endsequence** [: sequence_identifier]	**sequence** qReqReady (clk, req, data, mbus, ready); **logic** [0:31] v_data @ (**posedge** clk) (req, v_data=data) ##[0:$] ready && mbus==v_data; **endsequence** : qReqReady
property_spec ::= [clocking_event] [**disable iff** (expression_or_dist)] property_expr	**sequence** qCancel; interrupt ##1; **endsequence** : qCancel **event** blast_event; **property** pReqAck; @ (posedge clk) **disable iff** (!reset_n || qCancel.**triggered** || blast_event.**triggered**) req |=> ##[0:20] ack; **endproperty** : pReqAck
property_expr ::= sequence_expr | (property_expr) | **not** property_expr | property_expr **or** property_expr | property_expr **and** property_expr | sequence_expr |-> property_expr | sequence_expr |=> property_expr | **if** (expression_or_dist) property_expr [**else** property_expr] | property_instance | clocking_event property_expr	**property** pP1; qReqReady (ck, req, dta, mbus, rdy); **endproperty** : pP1 **property** pP2; (a |=> ##[0:20] b); **endproperty** : pP2 **property** pP3; **not** (push && pop); **endproperty** : pP3 **property** pP4; (a ##[0:4] b) **or** (c ##[0:20] d) ; **endproperty** : pP4 **property** pP5; (a ##[0:4] b) **and** (c ##[0:20] d) ; **endproperty** : pP5 **property** pP6; y |-> pP4; **endproperty** : pP6 **property** pP7; y |=> pP5 ; **endproperty** : pP **property** pP8; **if** (req) pP4 **else** pP5; **endproperty** : pP8 **property** pP9; @ (posedge clk) pP8; **endproperty** : pP **property** pP10; @ (posedge clk1) req |=> @ (posedge clk2) ##[0:2] ack; **endproperty** : pP
sequence_expr ::= cycle_delay_range sequence_expr { cycle_delay_range sequence_expr } | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr} | expression_or_dist [boolean_abbrev]	**sequence** qS1; ##1 a ##[0:3] b; **endsequence** : qS1 **sequence** qS2; a ##1 b ##[0:$] ##1 qS1; **endsequence** : qS1 **sequence** qS3; a[*3] ##1 b[=2] ##1 c[->1] ##1 d[*1:3] ##1 e; **endsequence** : qS3
| (expression_or_dist { , sequence_match_item }) [boolean_abbrev] | sequence_instance [sequence_abbrev] | (sequence_expr { , sequence_match_item }) [sequence_abbrev] | sequence_expr **and** sequence_expr | sequence_expr **intersect** sequence_expr | sequence_expr **or** sequence_expr | **first_match** (sequence_expr { , sequence_match_item})[1] | expression_or_dist **throughout** sequence_expr | sequence_expr **within** sequence_expr | clocking_event sequence_expr	**sequence** qS4; **int** v; (a, v=b) [*1:3] c ##[1:4] d==v; **endsequence** : qS4 **sequence** qS5; qS1[*2] ##1 qS2[->1]; **endsequence** : qS5 **sequence** qS6; **logic**[0:31] v; (a, v=b)[->1] c && v== d; **endsequence** : qS6 **sequence** qS7; qS1 **and** c ##[0:$] d; **endsequence** : qS7 **sequence** qS8; qS1 **intersect** c ##[0:$] d; **endsequence** : qS8 **sequence** qS9; qS1 **or** c ##[0:$] d; **endsequence** : qS9 **sequence** qS10; **int** v; **first_match**(a ##[0:3] b, v=b); **endsequence** : qS10 **sequence** qS11; a **throughout** b ##[0:4] c; **endsequence** : qS11 **sequence** qS12; qS1 **within** c ##[0:$] d; **endsequence** : qS12 **sequence** qS13; @ (posedge clk2) qS1; **endsequence** : qS13 **sequence** qS13b @ (posedge clk1) c ##1 qS13; **endsequence** : qS13b

[1] **First_match** is typically used in antecedents, such as:
property P; @ (posedge clk) **first_match**(a ##b[*1:2] ##1 c) |=> d; **endproperty** : P

Built-in Functions	Example
$sampled(EXP [, clk_evnt])	**$display**("sampled(data) %0d", **$sampled**(data));
$rose(EXP [,clk_evnt])	**$rose**(req) |=> ack;
$fell(EXP [,clk_evnt])	**$fell**(ready) |-> !bus_enb;
$stable(EXP [,clk_ev])	**$stable**(ack) |-> bus_enb;
$past(EXP1 [, number_of_ticks] [, EXP2] [,clk_evnt])	ack |=>ready ##1 dreg==**$past**(data, 2);
$onehot (EXP)	**$onehot** (fsm_reg) ;
$onehot0(EXP)	! **$onehot0**(fsm_reg) |=> error_flag;
$isunknown (EXP)	enb |-> ! **$isunknown** (data_bus);
$countones (EXP)	**$countones** (count_reg)== 5 |=> b;
$fatal	**$fatal** (2, ("%m @ time %0t, terminating simulation ", **$time**));
$error	**$error** ("%m @ time %0t, ack=%0h", **$time**, ack_bus);
$warning	**$warning** ("%m @ time %0t, ack=%0h", **$time**, ack_bus);
$info	**$info** ("%m @ time %0t, ack=%0h", **$time**, ack_bus);

Endpoints	Example
ended *// endpoint of a sequence, valid in Observed region*	**sequence** qABclk1(a, b): @ (posedge clk1) a ##1 b; **endsequence** : qABclk1 **property** pPe1; @ (posedge clk1) c |=> qABclk1**.ended;** **endproperty :** pPe1
matched *// endpoint synchronized to new clock*	**property** p_multi_clock; @(**posedge** sysclk) c ##1 qABclk1(a, b)**.matched** [->1] ##1 d; *// After c @ sysclk, 1 occurrence of qABclk1 endpoint checked @ clk1* *// and synchronized to sysclk, then d @ sysclk* **endproperty** : p_multi_clock
triggered *// endpoint lasting from observed to preponed region to end of cycle.* *// named event's triggered state persisted throughout the time-step*	**property** p_test; @ (posedge clk1) **disable iff** (!reset_n || qABclk1**.triggered**) c |=> d ##[1:7] e; **endproperty** : p_test

Verification directives	Example
Concurrent assertion: **assert property** (property_spec) action_block action_block ::= statement_or_null | **[** statement **]** **else** statement_or_null	ap_test: **assert property** p_test **else begin** **$display** ("%m @ time %0t, Failure, datain= %0h, dataout=%0h", **$time**, datain, dataout); **$info** ("FIFO ERROR – data out != data in"); fifo_err <= 1'b1; *// take a reactive action* **end**;
Immediate assertion **assert** (expression) action_block	**always** @ (posedge clk) **if** (a) **assert** (!(c && d);
assume property (property_spec);	**property** pHiStaysHi (s); @ (posedge clk) s |=> s; **endproperty** mpIfHiStaysHi :**assume property** (pHiStaysHi(rst_n));
cover property (property_spec) statement_or_null;	cp_datain2dataout : **cover property** (pdatain2dataout);
expect (property_spec) action_block	**property** pReqAck; @ (posedge clk) **$rose**(req) ##1 ready[*0:5] ##1 ack; **endproperty** : pReqAck **always** @(posedge clk) **begin** // ... **expect** p_reqack *// execute property* handshake_status <= PASS1 **end**

Sequence Operators	Example
##1 Temporal concatenation	**$rose**(go) ##1 req |=> !req && ack ##1data_transfer;
a ##1 b ##1 c // *a* at cycle t, *b* at t+1, *c* at t+2	
##0 Temporal overlap	req ##1 rdy ##0 xfr |=> ##[0:$] done;
	// rdy & xfr at same cycle
##[0:m] delay 0 (overlap) to m cycles until sequence	a |-> ##[0:3] ##1 b; //if *a* then 0 to 3 cycles *b*
##[n:m] delay n to m cycles until sequence	a |-> ##[1:3] ##1 b; //if *a* then 1 to 3 cycles *b*
##[0:$] delay 0 to infinity until sequence	a |-> ##[0:$] ##1 b; //if *a* then eventually *b*
[*] Consecutive repetition	
seq[*n] Sequence Repeats for n cycles	a |=> b[*2] ##1 c; // if *a*, then *b b c*
seq[*0:$] Repeats for zero or any number of cycles	a |=> b[*0:$] ##1 c; // if *a*, then *c* | b b b … c
seq[*n:m] Repeats for min of n to max of m cycles	a |=> b[*1:3] ##1 c; //if *a* then *b c* | *b b c* | *b b b c*
seq[*n:$] Repeats for a minimum of n cycles	a |=> b[*0:$] ##1 c; // if *a* then *c* | *b c* | *b b c* | *b b …b c*
seq[*0: m] Either skipped or repeats max m cycles	a |=> b[*0:3] ##1 c; // if *a* then *c* | *b c* | *b b c* | *b b b c*
[=count] Non-consecutive repetition	a |=> b[=2] ##1 c; // If *a* then b - .. - b - - .. - *c*;
[->count] GOTO repetition	a |=> b[->2]##1 c; // If *a* then b - - b *c*;
count is a number or a range	a |=> b[->2:3]##1 c; // If *a* then b - - b *c*;

Sequence composition operators

##0 sequence fusion. Two sequences overlap by one cycle

or sequence disjunction. One of two alternative sequences hold at the current cycle

and non-length-matching sequence conjunction. Two sequences both hold at the current cycle, regardless of whether they complete in same cycle or in different cycles.

intersect length-matching sequence conjunction. Two sequences both hold at the current cycle, and both complete in the same cycle.

↑Fusion (a ##1 b) ##0 (c ##1 d)

↑Sequence Disjuction (a ##1 b) **or** (c ##1 d)

Non-Length-Matching a ##[1:10] b **and** c[*1:5] ##1 d ↑

Length-Matching a ##1 [1:10] b **intersect** c ##[1:5] d ↑

APPENDIX E: SYSTEMVERILOG ASSERTIONS SYNTAX

The full SystemVerilog 3.1 hyperlinked BNF is available at:
http://www.eda.org/sv-ac/annexA_bnf_3_1_final.html
Users are encouraged to download this complete BNF, as the hyperlinked characteristics of this file make it very convenient to quickly find and search the desired syntax. This appendix only addresses the assertion aspect of SystemVerilog.

(NORMATIVE)

The formal syntax of SystemVerilog is described using Backus-Naur Form (BNF). The conventions used are:

- Keywords and punctuation are in *bold* text.
- Syntactic categories are named in non-bold text.
- A vertical bar (|) separates alternatives.
- Square brackets (*[ ]*) enclose optional items.
- Braces (*{ }*) enclose items which can be repeated zero or more times.

The full syntax and semantics of Verilog and SystemVerilog are not described solely using BNF. The normative text description contained within the chapters of the IEEE 1364-2001 Verilog standard and this SystemVerilog document provide additional details on the syntax and semantics described in this BNF.

ASSERTION DECLARATIONS

concurrent_assertion_item ::= [block_identifier :] concurrent_assertion_statement

concurrent_assertion_statement ::=
    assert_property_statement
    | assume_property_statement
    | cover_property_statement

assert_property_statement::=
    assert property (property_spec *)* action_block

assume_property_statement::=
    assume property (property_spec *) ;*

cover_property_statement::=
    cover property (property_spec *)* statement_or_null

expect_property_statement ::=
    expect (property_spec *)* action_block

property_instance ::=
    ps_property_identifier [*(* [actual_arg_list] *)*]

concurrent_assertion_item_declaration ::=
    property_declaration
    | sequence_declaration

```
property_declaration ::=
        property property_identifier [ ( [ list_of_formals ] ) ] ;
        { assertion_variable_declaration }
        property_spec ;
        endproperty [ : property_identifier ]

property_spec ::=
        [clocking_event ] [ disable iff ( expression_or_dist ) ] property_expr

property_expr ::=
        sequence_expr
        | ( property_expr )
        | not property_expr
        | property_expr or property_expr
        | property_expr and property_expr
        | sequence_expr |-> property_expr
        | sequence_expr |=> property_expr
        | if ( expression_or_dist ) property_expr [ else property_expr ]
        | property_instance
        | clocking_event property_expr

sequence_declaration ::=
        sequence sequence_identifier [ ( [ list_of_formals ] ) ] ;
        { assertion_variable_declaration }
        sequence_expr ;
        endsequence [ : sequence_identifier ]

sequence_expr ::=
        cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
        | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr}
        | expression_or_dist [ boolean_abbrev ]
        | ( expression_or_dist { , sequence_match_item } ) [ boolean_abbrev ]
        | sequence_instance [ sequence_abbrev ]
        | ( sequence_expr { , sequence_match_item } ) [ sequence_abbrev ]
        | sequence_expr and sequence_expr
        | sequence_expr intersect sequence_expr
        | sequence_expr or sequence_expr
        | first_match ( sequence_expr { , sequence_match_item} )
        | expression_or_dist throughout sequence_expr
        | sequence_expr within sequence_expr
        | clocking_event sequence_expr

cycle_delay_range ::=
        ## integral_number
        | ## identifier
        | ## ( constant_expression )
        | ## [ cycle_delay_const_range_expression ]

sequence_method_call ::=
        sequence_instance . method_identifier
```

sequence_match_item ::=
operator_assignment
| inc_or_dec_expression
| subroutine_call

sequence_instance ::=
ps_sequence_identifier [([actual_arg_list])]

formal_list_item ::=
formal_identifier [= actual_arg_expr]

list_of_formals ::= formal_list_item { , formal_list_item }

actual_arg_list ::=
actual_arg_expr { , actual_arg_expr }
| . formal_identifier (actual_arg_expr) { , . formal_identifier (actual_arg_expr) }

actual_arg_expr ::=
event_expression
| $

boolean_abbrev ::=
consecutive_repetition
| non_consecutive_repetition
| goto_repetition

sequence_abbrev ::= consecutive_repetition

consecutive_repetition ::= [* const_or_range_expression]

non_consecutive_repetition ::= [= const_or_range_expression]

goto_repetition ::= [-> const_or_range_expression]

const_or_range_expression ::=
constant_expression
| cycle_delay_const_range_expression

cycle_delay_const_range_expression ::=
constant_expression : constant_expression
| constant_expression : $

expression_or_dist ::= expression [dist { dist_list }]

assertion_variable_declaration ::=
data_type list_of_variable_identifiers ;

APPENDIX E: RESERVED KEYWORDS

alias‡
always
always_comb†
always_ff†
always_latch†
and
assert†
assign
assume*
automatic
before‡
begin
bind‡
bins*
binsof*
bit†
break†
buf
bufif0
bufif1
byte†
case
casex
casez
cell
chandle‡
class‡
clocking‡
cmos
config
const†
constraint‡
context‡
continue†
cover‡
covergroup*
coverpoint*
cross*
deassign
default
defparam
design
disable
dist‡
do†
edge
else
end
endcase
endclass‡
endclocking‡
endconfig
endfunction
endgenerate
endgroup*
endinterface†
endmodule
endpackage*
endprimitive
endprogram‡
endproperty‡
endspecify
endsequence‡
endtable
endtask
enum†
event
expect*
export†
extends‡
extern†
final‡
first_match‡
for
force
foreach*
forever
fork
forkjoin†
function
generate
genvar
highz0
highz1
if
iff†
ifnone
ignore_bins*
illegal_bins*
import†
incdir
include
initial
inout
input
inside‡
instance
int†
integer
interface†
intersect‡
join
join_any‡
join_none‡
large
liblist
library
local‡
localparam
logic†
longint†
macromodule
matches*
medium
modport†
module
nand
negedge
new‡
nmos
nor
noshowcancelled
not
notif0
notif1
null‡
or
output
package*
packed†
parameter
pmos
posedge
primitive
priority†
program‡
property‡
protected‡
pull0
pull1
pulldown
pullup
pulsestyle_onevent
pulsestyle_ondetec
t
pure‡
rand‡
randc‡
randcase*
randsequence*
rcmos
real
realtime
ref‡
reg
release
repeat
return†
rnmos
rpmos
rtran
rtranif0
rtranif1
scalared
sequence‡
shortint†
shortreal†
showcancelled
signed
small
solve‡
specify
specparam
static†
string‡
strong0
strong1
struct†
super‡
supply0
supply1
table
tagged*
task
this‡
throughout‡
time
timeprecision†
timeunit†
tran
tranif0
tranif1
tri
tri0
tri1
triand
trior
trireg
type†
typedef†
union†
unique†
unsigned
use
var‡
vectored
virtual‡
void†
wait
wait_order‡
wand
weak0
weak1
while
wildcard*
wire
with‡
within‡
wor
xnor
xor

Legend:
— † keywords added to the IEEE 1364 Verilog-2001 standard as part of SystemVerilog 3.0
— ‡ keywords added to the IEEE 1364 Verilog-2001 standard as part of SystemVerilog 3.1
— * keywords added to the IEEE 1364 Verilog-2001 standard as part of SystemVerilog 3.1a

INDEX

SYMBOLS

$assertkill 117, 242
$assertoff 117, 241
$asserton 117, 241
$countones 49, 116
$error 116
$fatal 116
$fell 49, 109,113,115
$info 116,117
$isunbounded 100
$isunknown 49,116
$onehot 49,116
$onehot0 49,116
$past 49,109,111-113
$rose 49,109,113
$sampled 49,109-111, 240
$stable 49,109,115
$unit 20
$warning 116
##n ##[n:m] ##[n:$] 48
|-> implication 53,54
|=> implication 53,55

A

ABV
  Applications 18
  Time to market 9
  design process 154

Accellera
  SVA Goals 11

API
  assertion 166-169
  in RTL 163

action block 22 137
  display guideline 247

active region 133

antecedent 16

API 166-169

architectural plan 161

assert
  directive 135-138

assertion
  algorithms in 158
  API 166-169
  applications 6, 12
  classification of 155
  clocking rules 129-131
  concurrent 21, 24
  conditional / generate 247-248
  constructs naming 232
  definition 4 11
  direct vs implicit 245-246
  display assertion variable
    in action block 240
  guideline
    writing 256
  hierarchy of constructs 120
  immediate 21 22 137,
  multi-clocked 131
  overlapping 93-97
  procedural block 136
  system-level 155-161
  variable 34, 238
  with Verilog RTL 249

assertion control
  $assertkill 117
  $assertoff 117
  $asserton 117

assume directive 45, 138-140

B

benefits
  assertion 6

BFM 183
  in testbench 187

before
  emulation 107-108

Binding 143-147

Boolean expression 26

C

:lassification
assertions 155

:lock
default
guidelines 241
identical
definition 307

'locking 39

lock flow in sequences 128-129

lock resolution 120-123, 317-320

locked sequence 118-120

ocking
default 119, 241

ocking
:locking rules
assertions 129-131

mpilation unit scope 27

ncurrent assertion 24
rocedural block 136

nstraint
efinition, example 19

nsequent 16

ver
irective 140-141

verage
etrics 216
roperties 255-259

le delay
n ##[n:m] ##[n:$] 48

data
seq
pro

defi

defa

dela
cyc

defir
Ass
AB

Desi
defi

direc
asse
assi
cov
veri

disju

disab
con
nest
guic

displ
in

dist

emul
until
while
eve

ev fp
enab
mon
trigg

D

types
uence 91
perty 91

nitions 305-311

ult clocking 39, 241

y
le ##n ##[n:m] ##[n:$] 48

nition
sertion 4 11
V 6

gn process 154
ault clocking 119

tives
ert 19, 135
xme 19, 138-140
er 19, 140-141
fication directives 135-142

nction 76

ble iff
dition 40
ing 121
lelines 241-242

ay assertion variable
action block 240, 247

45

E

ation
l 106
lenot 104
ntually 106

oint
ed 49, 80-85
ched 49, 80-85
gered 49, 80-85

event
clocking 39
sequence 85

Expert
construct 142-143

Expression
Boolean 26

F

$fell 113

FIFO
verification 162, 182-190
requirements plan for FIFO 171-181
RTL design 191-192
formal verification 193-194

first_match 49 60-64

formal arguments
property 33

formal verification
bounded/unbounded
model checking 203
case study
traffic light contoller 206-216
FIFO 193-194
metrics
proof radius 216
reflections
process 216
semi-formal 218
temporal debugging 222
topic 199-206
traffic light controller
good 215
bad 206

functions
see system functions 109

functional coverage
defintion 9
metrics 216
points 221

fusion 76

G

generate
property 101

Goto [-> 65

guidelines 229-259
$assertkill 242
$assertoff 242
$asserton 242
$sampled 240
action block
display guideline 247
assertion constructs naming 232
clock default 241
disable iff
guidelines 241-242
file naming 230
metalogical 239
methodology properties 251-255
named sequence 257
object naming 230
process writing 256-259
reset 241
timeformat 242-245
timeunit 242-245
unsized additions 235
variables 238

H

handshake model 197, 301-304

history
ABV, SystemVerilog 2 11

hierarchy
assertion constructs 120

HVL 4 11

I

immediate
assertion 21, 22, 137

interface
assertion 161
example 175
properties example 101

interface
property 134

M

matched
endpoint 49, 80-85

metalogical 239

methods
endpoint
ended 49, 80-85
matched 49, 80-85
triggered 49, 80-85

mixed mode
VHDL 10, 148-152

model
data integrity 274-275
FIFO 176
handshake 197, 301-304
memory data integrity 273
parking structure 195, 298-301
traffic light controller
good 215
bad 206

multiple-clock
rules 125

multiple-clocked
assertions 131
properties 127
sequences 123, 131

multi-clocking 118-120

N

named property 20

nesting
disable iff 121

non-consecutive
repetition 64

non-overlapped
implication operator 53 55

not
property operator 43
with implication 59

O

observe
region 133

one-shot assertion 108

operator
|-> implication 53 54
|=> implication 53 55
[*n] 65-77
[*n:m] 65-77
[*n:$] 65-77
[=n] 65-77
[->] 65-77
and 48 49 76 77
first_match 49 60-64
goto 64
if else 44, 49
intersect 49 77
not 49, 237
property 42
or 48 49 76
repetition 64-75
sequence 29
sequence composition
throughout 48 76 79
within 48 49 76 78

operator emulation
eventualally
SERE 106
whilenot 104

P

package
example 174

parking structure
model 195, 298-301

Pattern
in property 19

plan
architectural 161
verification 162, 182-190
requirements plan for FIFO 171-181

precedence
property operator 42
sequence operator 48

procedural block
assertion 136

process
design with SVA 153-197

property
binding 143
conditional / generate 247
construct syntax 31
coverage 255-256
data types 91
definition 20
dynamic data in properties 250
expression 42
formal arguments 33
interface 134
local variable 34, 238,
named 20
operators precedence 42
process writing 256-259
where declared 27

PSL like constructs
before 107-108
eventualally! 106
whilenot 104-105
SVA 104

Q

quueue
data integrity model 274-275
FIFO model 176

R

range
unbound 100
delay ##[n:m] 48

recursion 101
before emulation 107

repetition
operators 64-77
non-consecutive [=n] 74
goto [->n] 74
repetition [*] 66-73
[*n] 65-77
[*n:m] 65-77
[*n:$] 65-77
[=n] 65-77
[->n] 65-77

regions & scheduling
SystemVerilog 131-132

Requirements
Architectural Overview 172
Hardware and Software 180
Plan example for FIFO
System Application 173 194

requirements plan
FIFO 171-181

reset 241

reserved words 324

RTL
design and assertion 161

S

scheduling 131-133

sequence
data types 91

overview 16 28
operators 29, 76-80
precedence 48

sequence composition
operator 75

sequence event 85

severity levels
$error 116
$fatal 116
$info 116
$warning 116

system functions
$fell 109
$past 109, 111-113
$rose 109
$sampled 49, 109-111, 240
$stable 109

system-level
assertions 155-161

SystemVerilog
regions 132
scheduling 131-133
Unified Language 6
assertion usage 12

T

testbench 187,
assertions 164

time slots 132

timeunit 242-245

timeformat 242-245

time slots 131-133

Thread
definition 16
termination 98

triggered 41, 80-85
disable iff 41
endpoint49

U

unbound
range 100

until
emulation 106

V

vacuity definition 311

vacuous 44, 51, 54, 57-62, 239

variable
property 34

vector 35

verification
dynamic 12
plan 162
feature extraction 185
compliance plan 184
static 12

verification directives 135-142

verification IP 259

Verilog RTL
with SVA 249

VHDL
with SVA 10, 148-152

W

within operator 78

whilenot emulation 104