SystemVerilog For Design

Second Edition

A Guide to Using SystemVerilog
for Hardware Design and Modeling

SystemVerilog For Design

Second Edition

A Guide to Using SystemVerilog for Hardware Design and Modeling

by

Stuart Sutherland

Simon Davidmann

Peter Flake

Foreword by Phil Moorby

Springer

Stuart Sutherland
Sutherland DHL, Inc.
22805 SW 92nd Place
Tualatin, OR 97062
USA

Simon Davidmann
The Old Vicerage
Priest End
Thame, Oxfordshire 0X9 3AB
United Kingdom

Peter Flake
Imperas, Ltd.
Imperas Buildings, North Weston
Thame, Oxfordshire 0X9 2HA
United Kingdom

SystemVerilog for Design, Second Edition
A Guide to Using SystemVerilog for Hardware Design and Modeling

Library of Congress Control Number: 2006928944

ISBN-10: 0-387-33399-1   e-ISBN-10: 0-387-36495-1
ISBN-13: 9780387333991   e-ISBN-13: 9780387364957

Printed on acid-free paper.

Printed in the United States of America.

9 8 7 6 5 4 3 2

springer.com

Dedications

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

Stuart Sutherland
Portland, Oregon

To all of the staff of Co-Design and the many EDA colleagues that worked with me over the years — thank you for helping to evolve Verilog and make its extension and evolution a reality. And to Penny, Emma and Charles — thank you for allowing me the time to indulge in language design (and in cars and guitars...).

Simon Davidmann
Santa Clara, California

To my wife Monique, for supporting me when I was not working, and when I was working too much.

Peter Flake
Thame, UK

Table of Contents

Foreword xxi

Preface xxiii

Target audience xxiii
Topics covered xxiv
About the examples in this book xxv
Obtaining copies of the examples xxvi
Example testing xxvi
Other sources of information xxvii
Acknowledgements xxx

Chapter 1: Introduction to SystemVerilog 1

1.1 SystemVerilog origins 1
  1.1.1 Generations of the SystemVerilog standard 2
  1.1.2 Donations to SystemVerilog 4
1.2 Key SystemVerilog enhancements for hardware design 5
1.3 Summary 6

Chapter 2: SystemVerilog Declaration Spaces 7

2.1 Packages 8
  2.1.1 Package definitions 9
  2.1.2 Referencing package contents 10
  2.1.3 Synthesis guidelines 14
2.2 $unit compilation-unit declarations 14
  2.2.1 Coding guidelines 17
  2.2.2 SystemVerilog identifier search rules 17
  2.2.3 Source code order 17
  2.2.4 Coding guidelines for importing packages into $unit 19
  2.2.5 Synthesis guidelines 25
2.3 Declarations in unnamed statement blocks 26
  2.3.1 Local variables in unnamed blocks 27
2.4 Simulation time units and precision 28
  2.4.1 Verilog's timescale directive 28
  2.4.2 Time values with time units 30
  2.4.3 Scope-level time unit and precision 31

2.4.4 Compilation-unit time units and precision 32
2.5 Summary 34

Chapter 3: SystemVerilog Literal Values and Built-in Data Types....................37
3.1 Enhanced literal value assignments..38
3.2 ‘define enhancements..39
3.2.1 Macro argument substitution within strings..39
3.2.2 Constructing identifier names from macros...41
3.3 SystemVerilog variables...42
3.3.1 Object types and data types...42
3.3.2 SystemVerilog 4-state variables...43
3.3.3 SystemVerilog 2-state variables...44
3.3.4 Explicit and implicit variable and net data types 47
3.3.5 Synthesis guidelines 48
3.4 Using 2-state types in RTL models 48
3.4.1 2-state type characteristics 49
3.4.2 2-state types versus 2-state simulation..49
3.4.3 Using 2-state types with case statements 51
3.5 Relaxation of type rules..52
3.6 Signed and unsigned modifiers 55
3.7 Static and automatic variables...56
3.7.1 Static and automatic variable initialization 59
3.7.2 Synthesis guidelines for automatic variables 60
3.7.3 Guidelines for using static and automatic variables.......................................61
3.8 Deterministic variable initialization 61
3.8.1 Initialization determinism 61
3.8.2 Initializing sequential logic asynchronous inputs 65
3.9 Type casting 67
3.9.1 Static (compile time) casting..67
3.9.2 Dynamic casting...69
3.9.3 Synthesis guidelines 70
3.10 Constants 71
3.11 Summary 72

Chapter 4: SystemVerilog User-Defined and Enumerated Types 75
4.1 User-defined types...75
4.1.1 Local typedef definitions..76
4.1.2 Shared typedef definitions..76
4.1.3 Naming convention for user-defined types 78
4.2 Enumerated types 79
4.2.1 Enumerated type label sequences..83

4.2.2 Enumerated type label scope 83
4.2.3 Enumerated type values 84
4.2.4 Base type of enumerated types 85
4.2.5 Typed and anonymous enumerations 86
4.2.6 Strong typing on enumerated type operations 86
4.2.7 Casting expressions to enumerated types 88
4.2.8 Special system tasks and methods for enumerated types 89
4.2.9 Printing enumerated types 92
4.3 Summary 93

Chapter 5: SystemVerilog Arrays, Structures and Unions 95

5.1 Structures 96
5.1.1 Structure declarations 97
5.1.2 Assigning values to structures 98
5.1.3 Packed and unpacked structures 101
5.1.4 Passing structures through ports 104
5.1.5 Passing structures as arguments to tasks and functions 105
5.1.6 Synthesis guidelines 105
5.2 Unions 105
5.2.1 Unpacked unions 106
5.2.2 Tagged unions 108
5.2.3 Packed unions 109
5.2.4 Synthesis guidelines 111
5.2.5 An example of using structures and unions 111
5.3 Arrays 113
5.3.1 Unpacked arrays 113
5.3.2 Packed arrays 116
5.3.3 Using packed and unpacked arrays 118
5.3.4 Initializing arrays at declaration 119
5.3.5 Assigning values to arrays 121
5.3.6 Copying arrays 123
5.3.7 Copying arrays and structures using bit-stream casting 124
5.3.8 Arrays of arrays 125
5.3.9 Using user-defined types with arrays 126
5.3.10 Passing arrays through ports and to tasks and functions 127
5.3.11 Arrays of structures and unions 128
5.3.12 Arrays in structures and unions 128
5.3.13 Synthesis guidelines 128
5.3.14 An example of using arrays 129
5.4 The foreach array looping construct 130

5.5 Array querying system functions 132
5.6 The $bits "sizeof" system function 134
5.7 Dynamic arrays, associative arrays, sparse arrays and strings 135
5.8 Summary 136

Chapter 6: SystemVerilog Procedural Blocks, Tasks and Functions 137

6.1 Verilog general purpose always procedural block 138
6.2 SystemVerilog specialized procedural blocks 142
  6.2.1 Combinational logic procedural blocks 142
  6.2.2 Latched logic procedural blocks 150
  6.2.3 Sequential logic procedural blocks 152
  6.2.4 Synthesis guidelines 152
6.3 Enhancements to tasks and functions 153
  6.3.1 Implicit task and function statement grouping 153
  6.3.2 Returning function values 153
  6.3.3 Returning before the end of tasks and functions 154
  6.3.4 Void functions 155
  6.3.5 Passing task/function arguments by name 156
  6.3.6 Enhanced function formal arguments 157
  6.3.7 Functions with no formal arguments 158
  6.3.8 Default formal argument direction and type 158
  6.3.9 Default formal argument values 159
  6.3.10 Arrays, structures and unions as formal arguments 160
  6.3.11 Passing argument values by reference instead of copy 161
  6.3.12 Named task and function ends 165
  6.3.13 Empty tasks and functions 166
6.4 Summary 166

Chapter 7: SystemVerilog Procedural Statements 169

7.1 New operators 170
  7.1.1 Increment and decrement operators 170
  7.1.2 Assignment operators 173
  7.1.3 Equality operators with don't care wildcards 176
  7.1.4 Set membership operator — inside 178
7.2 Operand enhancements 180
  7.2.1 Operations on 2-state and 4-state types 180
  7.2.2 Type casting 180
  7.2.3 Size casting 181
  7.2.4 Sign casting 182
7.3 Enhanced for loops 182
  7.3.1 Local variables within for loop declarations 183

7.3.2 Multiple for loop assignments....185
7.3.3 Hierarchically referencing variables declared in for loops....185
7.3.4 Synthesis guidelines....186
7.4 Bottom testing do...while loop....186
7.4.1 Synthesis guidelines....188
7.5 The foreach array looping construct....188
7.6 New jump statements — break, continue, return....188
7.6.1 The continue statement....190
7.6.2 The break statement....190
7.6.3 The return statement....191
7.6.4 Synthesis guidelines....192
7.7 Enhanced block names....192
7.8 Statement labels....194
7.9 Enhanced case statements....195
7.9.1 Unique case decisions....196
7.9.2 Priority case statements....199
7.9.3 Unique and priority versus parallel_case and full_case....201
7.10 Enhanced if...else decisions....203
7.10.1 Unique if...else decisions....203
7.10.2 Priority if decisions....205
7.11 Summary....206

Chapter 8: Modeling Finite State Machines with SystemVerilog....207
8.1 Modeling state machines with enumerated types....208
8.1.1 Representing state encoding with enumerated types....210
8.1.2 Reversed case statements with enumerated types....211
8.1.3 Enumerated types and unique case statements....213
8.1.4 Specifying unused state values....214
8.1.5 Assigning state values to enumerated type variables....216
8.1.6 Performing operations on enumerated type variables....218
8.2 Using 2-state types in FSM models....219
8.2.1 Resetting FSMs with 2-state and enumerated types....219
8.3 Summary....221

Chapter 9: SystemVerilog Design Hierarchy....223
9.1 Module prototypes....224
9.1.1 Prototype and actual definition....225
9.1.2 Avoiding port declaration redundancy....225
9.2 Named ending statements....226
9.2.1 Named module ends....226
9.2.2 Named code block ends....226

9.3 Nested (local) module declarations 227
9.3.1 Nested module name visibility 230
9.3.2 Instantiating nested modules 231
9.3.3 Nested module name search rules 232
9.4 Simplified netlists of module instances 233
9.4.1 Implicit .name port connections 238
9.4.2 Implicit .* port connection 242
9.5 Net aliasing 244
9.5.1 Alias rules 245
9.5.2 Implicit net declarations 246
9.5.3 Using aliases with .name and .* 247
9.6 Passing values through module ports 251
9.6.1 All types can be passed through ports 251
9.6.2 Module port restrictions in SystemVerilog 252
9.7 Reference ports 255
9.7.1 Reference ports as shared variables 256
9.7.2 Synthesis guidelines 256
9.8 Enhanced port declarations 257
9.8.1 Verilog-1995 port declarations 257
9.8.2 Verilog-2001 port declarations 257
9.8.3 SystemVerilog port declarations 258
9.9 Parameterized types 260
9.10 Summary 261

Chapter 10: SystemVerilog Interfaces 263
10.1 Interface concepts 264
10.1.1 Disadvantages of Verilog's module ports 268
10.1.2 Advantages of SystemVerilog interfaces 269
10.1.3 SystemVerilog interface contents 273
10.1.4 Differences between modules and interfaces 273
10.2 Interface declarations 274
10.2.1 Source code declaration order 276
10.2.2 Global and local interface definitions 276
10.3 Using interfaces as module ports 277
10.3.1 Explicitly named interface ports 277
10.3.2 Generic interface ports 278
10.3.3 Synthesis guidelines 278
10.4 Instantiating and connecting interfaces 278
10.5 Referencing signals within an interface 279
10.6 Interface modports 281

10.6.1 Specifying which modport view to use 282
10.6.2 Using modports to define different sets of connections 286
10.7 Using tasks and functions in interfaces 288
10.7.1 Interface methods 289
10.7.2 Importing interface methods 289
10.7.3 Synthesis guidelines for interface methods 292
10.7.4 Exporting tasks and functions 293
10.8 Using procedural blocks in interfaces 296
10.9 Reconfigurable interfaces 296
10.10 Verification with interfaces 298
10.11 Summary 299

Chapter 11: A Complete Design Modeled with SystemVerilog..........................301
11.1 SystemVerilog ATM example...301
11.2 Data abstraction 302
11.3 Interface encapsulation 305
11.4 Design top level: squat 308
11.5 Receivers and transmitters...315
11.5.1 Receiver state machine...315
11.5.2 Transmitter state machine 318
11.6 Testbench...321
11.7 Summary 327

Chapter 12: Behavioral and Transaction Level Modeling 329
12.1 Behavioral modeling 330
12.2 What is a transaction?..330
12.3 Transaction level modeling in SystemVerilog 332
12.3.1 Memory subsystem example...333
12.4 Transaction level models via interfaces 335
12.5 Bus arbitration 337
12.6 Transactors, adapters, and bus functional models 341
12.6.1 Master adapter as module..341
12.6.2 Adapter in an interface 348
12.7 More complex transactions 353
12.8 Summary 354

Appendix A: The SystemVerilog Formal Definition (BNF) 355

Appendix B: Verilog and SystemVerilog Reserved Keywords..........................395

Appendix C: A History of SUPERLOG, the Beginning of SystemVerilog 401

Index 415

About the Authors

Stuart Sutherland provides expert instruction on using SystemVerilog and Verilog. He has been involved in defining the Verilog language since the beginning of IEEE standardization work in 1993, and is a member of both the IEEE Verilog standards committee (where he has served as the chair and co-chair of the Verilog PLI task force), and the IEEE SystemVerilog standards committee (where he has served as the editor for the SystemVerilog Language Reference Manual). Stuart has more than 20 years of experience in hardware design, and over 17 years of experience with Verilog. He is the founder of *Sutherland HDL Inc.*, which specializes in providing expert HDL training services. He holds a Bachelors degree in Computer Science, with an emphasis in Electronic Engineering Technology. He has also authored *"The Verilog PLI Handbook"* and *"Verilog-2001: A Guide to the New Features of the Verilog HDL"*.

Simon Davidmann has been involved with HDLs since 1978. He was a member of the HILO team at Brunel University in the UK. In 1984 he became an ASIC designer and embedded software developer of real time professional musical instruments for Simmons Percussion. In 1988, he became involved with Verilog as the first European employee of Gateway Design Automation. He founded Chronologic Simulation in Europe, the European office of Virtual Chips (inSilicon), and then the European operations of Ambit Design. In 1998, Mr. Davidmann co-founded Co-Design Automation, and was co-creator of SUPERLOG. As CEO of Co-Design, he was instrumental in transitioning SUPERLOG into Accellera as the beginning of SystemVerilog. Mr. Davidmann is a member of the Accellera SystemVerilog and IEEE 1364 Verilog committees. He is a consultant to, and board member of, several technology and EDA companies, and is Visiting Professor of Digital Systems at Queen Mary, University of London. In 2005 Mr. Davidmann founded Imperas, Inc where he is President & CEO.

Peter Flake was a co-founder and Chief Technical Officer at Co-Design Automation and was the main architect of the SUPERLOG language. With the acquisition of Co-Design by Synopsys in 2002, he became a Scientist at Synopsys. His EDA career spans more than 30 years: he was the language architect and project leader of the HILO development effort while at Brunel University in Uxbridge, U.K., and at GenRad. HILO was the first commercial HDL-based simulation, fault simulation and timing analysis system of the early/mid 1980s. In 2005 he became Chief Scientist at Imperas. He holds a Master of Arts degree from Cambridge University in the U.K. and has made many conference presentations on the subject of HDLs.

List of Examples

This book contains a number of examples that illustrate the proper usage of SystemVerilog constructs. A summary of the major code examples is listed in this section. In addition to these examples, each chapter contains many code fragments that illustrate specific features of SystemVerilog. The source code for these full examples, as well as many of the smaller code snippets, can be downloaded from ***http://www.sutherland-hdl.com***. Navigate the links to ***"SystemVerilog Book Examples"***.

Page xxv of the Preface provides more details on the code examples in this book.

Chapter 1: Introduction to SystemVerilog

Chapter 2: SystemVerilog Declaration Spaces

Example 2-1: A package definition 9
Example 2-2: Explicit package references using the :: scope resolution operator 10
Example 2-3: Importing specific package items into a module 11
Example 2-4: Using a package wildcard import 13
Example 2-5: External declarations in the compilation-unit scope (not synthesizable) 15
Example 2-6: Package with conditional compilation (file name: definitions.pkg) 21
Example 2-7: A design file that includes the conditionally-compiled package file 23
Example 2-8: A testbench file that includes the conditionally-compiled package file 23
Example 2-9: Mixed declarations of time units and precision (not synthesizable) 34

Chapter 3: SystemVerilog Literal Values and Built-in Data Types

Example 3-1: Relaxed usage of variables 53
Example 3-2: Illegal use of variables 54
Example 3-3: Applying reset at simulation time zero with 2-state types 65

Chapter 4: SystemVerilog User-Defined and Enumerated Types

Example 4-1: Directly referencing typedef definitions from a package 77
Example 4-2: Importing package typedef definitions into $unit 78
Example 4-3: State machine modeled with Verilog ‘define and parameter constants 79
Example 4-4: State machine modeled with enumerated types 81
Example 4-5: Using special methods to iterate through enumerated type lists 91
Example 4-6: Printing enumerated types by value and by name 92

Chapter 5: SystemVerilog Arrays, Structures and Unions

Example 5-1: Using structures and unions 112
Example 5-2: Using arrays of structures to model an instruction register 129

Chapter 6: SystemVerilog Procedural Blocks, Tasks and Functions

Example 6-1: A state machine modeled with **always** procedural blocks 145
Example 6-2: A state machine modeled with **always_comb** procedural blocks 147
Example 6-3: Latched input pulse using an **always_latch** procedural block 151

Chapter 7: SystemVerilog Procedural Statements

Example 7-1: Using SystemVerilog assignment operators 175
Example 7-2: Code snippet with unnamed nested **begin**...**end** blocks 192
Example 7-3: Code snippet with named **begin** and named **end** blocks 193

Chapter 8: Modeling Finite State Machines with SystemVerilog

Example 8-1: A finite state machine modeled with enumerated types (poor style) 208
Example 8-2: Specifying one-hot encoding with enumerated types 210
Example 8-3: One-hot encoding with reversed case statement style 212
Example 8-4: Code snippet with illegal assignments to enumerated types 216

Chapter 9: SystemVerilog Design Hierarchy

Example 9-1: Nested module declarations 228
Example 9-2: Hierarchy trees with nested modules 231
Example 9-3: Simple netlist using Verilog's named port connections 235
Example 9-4: Simple netlist using SystemVerilog's .name port connections 239
Example 9-5: Simple netlist using SystemVerilog's .* port connections 243
Example 9-6: Netlist using SystemVerilog's .* port connections without aliases 248
Example 9-7: Netlist using SystemVerilog's .* connections along with net aliases 249
Example 9-8: Passing structures and arrays through module ports 252
Example 9-9: Passing a reference to an array through a module ref port 255
Example 9-10: Polymorphic adder using parameterized variable types 261

Chapter 10: SystemVerilog Interfaces

Example 10-1: Verilog module interconnections for a simple design 264
Example 10-2: SystemVerilog module interconnections using interfaces 270
Example 10-3: The interface definition for `main_bus`, with external inputs 274
Example 10-4: Using interfaces with .* connections to simplify complex netlists 275
Example 10-5: Referencing signals within an interface 280
Example 10-6: Selecting which modport to use at the module instance 283
Example 10-7: Selecting which modport to use at the module definition 284

Example 10-8: A simple design using an interface with modports 287
Example 10-9: Using modports to select alternate methods within an interface 291
Example 10-10:Exporting a function from a module through an interface modport 294
Example 10-11:Exporting a function from a module into an interface 294
Example 10-12:Using parameters in an interface 297

Chapter 11: A Complete Design Modeled with SystemVerilog

Example 11-1: Utopia ATM interface, modeled as a SystemVerilog interface 306
Example 11-2: Cell rewriting and forwarding configuration 307
Example 11-3: ATM squat top-level module 309
Example 11-4: Utopia ATM receiver 315
Example 11-5: Utopia ATM transmitter 318
Example 11-6: UtopiaMethod interface for encapsulating test methods 321
Example 11-7: CPUMethod interface for encapsulating test methods 322
Example 11-8: Utopia ATM testbench 323

Chapter 12: Behavioral and Transaction Level Modeling

Example 12-1: Simple memory subsystem with read and write tasks 333
Example 12-2: Two memory subsystems connected by an interface 335
Example 12-3: TLM model with bus arbitration using semaphores 338
Example 12-4: Adapter modeled as a module 341
Example 12-5: Simplified Intel Multibus with multiple masters and slaves 342
Example 12-6: Simple Multibus TLM example with master adapter as a module 343
Example 12-7: Simple Multibus TLM example with master adapter as an interface 348

Foreword

by Phil Moorby
The creator of the Verilog language

When Verilog was created in the mid-1980s, the typical design size was of the order of five to ten thousand gates, the typical design creation method was that of using graphical schematic entry tools, and simulation was beginning to be an essential gate level verification tool. Verilog addressed the problems of the day, but also included capabilities that enabled a new generation of EDA technology to evolve, namely synthesis from RTL. Verilog thus became the mainstay language of IC designers.

Throughout the 1990s, the Verilog language continued to evolve with technology, and the IEEE ratified new extensions to the standard in 2001. Most of the new capabilities in the 2001 standard that users were eagerly waiting for were relatively minor feature refinements as found in other HDLs, such as multidimensional arrays, automatic variables and the generate statement. Today many EDA tools support these Verilog-2001 enhancements, and thus provide users with access to these new capabilities.

SystemVerilog is a significant new enhancement to Verilog and includes major extensions into abstract design, testbench, formal, and C-based APIs. SystemVerilog also defines new layers in the Verilog simulation strata. These extensions provide significant new capabilities to the designer, verification engineer and architect, allowing better teamwork and co-ordination between different project members. As was the case with the original Verilog, teams who adopt SystemVerilog based tools will be more productive and produce better quality designs in shorter periods.

A strong guiding requirement for SystemVerilog is that it should be a true superset of Verilog, and as new tools become available, I believe all Verilog users, and many users of other HDLs, will naturally adopt it.

When I developed the original Verilog LRM and simulator, I had an expectation of maybe a 10-15 year life-span, and during this time I have kept involved with its evolution. When Co-Design Automation was formed by two of the authors, Peter Flake and Simon Davidmann, to develop SUPERLOG and evolve Verilog, I was invited to join its Technical Advisory Board and, later, I joined the company and chaired its SUPERLOG Working Group. More recently, SUPERLOG was adopted by Accellera and has become the basis of SystemVerilog. I did not expect Verilog to be as successful as it has been and, with the extensions in SystemVerilog, I believe that it will now become the dominant HDL and provide significant benefits to the current and future generation of hardware designers, architects and verification engineers, as they endeavor to create smaller, better, faster, cheaper products.

If you are a designer or architect building digital systems, or a verification engineer searching for bugs in these designs, then SystemVerilog will provide you with significant benefits, and this book is a great place to start to learn SystemVerilog and the future of Hardware Design and Verification Languages.

Phil Moorby,
New England, 2003

Preface

SystemVerilog, officially the **IEEE Std 1800-2005™** standard, is a set of extensions to the **IEEE Std 1364-2005™ Verilog Standard** (commonly referred to as ***"Verilog-2005")***. These extensions provide new and powerful language constructs for modeling and verifying the behavior of designs that are ever increasing in size and complexity. The SystemVerilog extensions to Verilog can be generalized to two primary categories:

- Enhancements primarily addressing the needs of hardware modeling, both in terms of overall efficiency and abstraction levels.
- Verification enhancements and assertions for writing efficient, race-free testbenches for very large, complex designs.

Accordingly, the discussion of SystemVerilog is divided into two books. This book, ***SystemVerilog for Design***, addresses the first category, using SystemVerilog for modeling hardware designs at the RTL and system levels of abstraction. Most of the examples in this book can be realized in hardware, and are synthesizable. A companion book, ***SystemVerilog for Verification***[1], covers the second purpose of SystemVerilog, that of verifying correct functionality of large, complex designs.

Target audience

NOTE: This book assumes the reader is already familiar with the Verilog Hardware Description Language.

This book is intended to help users of the Verilog language understand the capabilities of the SystemVerilog enhancements to Verilog. The book presents SystemVerilog in the context of examples, with an emphasis on correct usage of SystemVerilog constructs. These examples include a mix of standard Verilog code along with SystemVerilog the enhancements. The explanations in the book focus on these SystemVerilog enhancements, with an assumption that the reader will understand the Verilog portions of the examples.

Additional references on SystemVerilog and Verilog are listed on page xxvii.

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

Topics covered

This book focusses on the portion of SystemVerilog that is intended for representing hardware designs in a manner that is both simulatable and synthesizable.

Chapter 1 presents a brief overview of SystemVerilog and the key enhancements that it adds to the Verilog language.

Chapter 2 discusses the enhancements SystemVerilog provides on where design data can be declared. Packages, $unit, shared variables and other important topics regarding declarations are covered.

Chapter 3 goes into detail on the many new data types SystemVerilog adds to Verilog. The chapter covers the intended and proper usage of these new data types.

Chapter 4 presents user-defined data types, a powerful enhancement to Verilog. The topics include how to create new data type definitions using **`typedef`** and defining enumerated type variables.

Chapter 5 looks at using structures and unions in hardware models. The chapter also presents a number of enhancements to arrays, together with suggestions as to how they can be used as abstract, yet synthesizable, hardware modeling constructs.

Chapter 6 presents the specialized procedural blocks, coding blocks and enhanced task and function definitions in SystemVerilog, and how these enhancements will help create models that are correct by design.

Chapter 7 shows how to use the enhancements to Verilog operators and procedural statements to code accurate and deterministic hardware models, using fewer lines of code compared to standard Verilog.

Chapter 8 provides guidelines on how to use enumerated types and specialized procedural blocks for modeling Finite State Machine (FSM) designs. This chapter also presents a number of guidelines on modeling hardware using 2-state logic.

Chapter 9 examines the enhancements to design hierarchy that SystemVerilog provides. Significant constructs are presented, including nested module declarations and simplified module instance declarations.

Chapter 10 discusses the powerful interface construct that SystemVerilog adds to Verilog. Interfaces greatly simplify the representation of complex busses and enable the creation of more intelligent, easier to use IP (intellectual property) models.

Chapter 11 ties together the concepts from all the previous chapters by applying them to a much more extensive example. The example shows a complete model of an ATM switch design, modeled in SystemVerilog.

Chapter 12 provides another complete example of using SystemVerilog. This chapter covers the usage of SystemVerilog to represent models at a much higher level of abstraction, using transactions.

Appendix A lists the formal syntax of SystemVerilog using the Backus-Naur Form (BNF). The SystemVerilog BNF includes the full Verilog-2005 BNF, with the SystemVerilog extensions integrated into the BNF.

Appendix B lists the set of reserved keywords in the Verilog and SystemVerilog standards. The appendix also shows how to mix Verilog models and SystemVerilog models in the same design, and maintain compatibility between the different keyword lists.

Appendix C presents an informative history of hardware description languages and Verilog. It covers the development of the SUPERLOG language, which became the basis for much of the synthesizable modeling constructs in SystemVerilog.

About the examples in this book

The examples in this book are intended to illustrate specific SystemVerilog constructs in a realistic but brief context. To maintain that focus, many of the examples are relatively small, and often do not reflect the full context of a complete model. However, the examples serve to show the proper usage of SystemVerilog constructs. To show the power of SystemVerilog in a more complete context, Chapter 11 contains the full source code of a more extensive example.

The examples contained in the book use the convention of showing all Verilog and SystemVerilog keywords in bold, as illustrated below:

Example: SystemVerilog code sample

```
module uart (output logic [7:0] data,
             output logic       data_rdy,
             input              serial_in);

  enum {WAITE, LOAD, READY} State, NextState;
  logic [2:0] bit_cnt;
  logic       cntr_rst, shift_en;
```

```
  always_ff @(posedge clock, negedge resetN) begin: shifter
    if (!resetN)
      data <= 8'h0;                     //reset (active low)
    else if (shift_en)
      data <= {serial_in, data[7:1]};  //shift right
  end: shifter
endmodule
```

Longer examples in this book list the code between double horizontal lines, as shown above. There are also many shorter examples in each chapter that are embedded in the body of the text, without the use of horizontal lines to set them apart. For both styles of examples, the full source code is not always included in the book. This was done in order to focus on specific aspects of SystemVerilog constructs without excessive clutter from surrounding code.

NOTE The examples do not distinguish standard Verilog constructs and keywords from SystemVerilog constructs and keywords. It is expected that the reader is already familiar with the Verilog HDL, and will recognize standard Verilog versus the new constructs and keywords added with SystemVerilog.

Obtaining copies of the examples

The complete code for all the examples listed in this book are available for personal, non-commercial use. They can be downloaded from ***http://www.sutherland-hdl.com***. Navigate the links to ***"SystemVerilog Book Examples"***.

Example testing

Most examples in this book have been tested using the *Synopsys* **VCS**® simulator, version 2005.06-SP1, and the *Mentor Graphics* **Questa™** simulator, version 6.2. Most models in this book are synthesizable, and have been tested using the *Synopsys* **DC Compiler™** synthesis compiler, version 2005.12.[1]

1. All company names and product names mentioned in this book are the trademark or registered trademark names of their respective companies.

Other sources of information

This book only explains the SystemVerilog enhancements for modeling hardware designs. The book does not go into detail on the SystemVerilog enhancements for verification, and does not cover the Verilog standard. Some other resources which can serve as excellent companions to this book are:

SystemVerilog for Verification—A Guide to Learning the Testbench Language Features by Chris Spear.

Copyright 2006, Springer, Norwalk, Massachusetts. ISBN 0-387-27036-1.

A companion to this book, with a focus on verification methodology using the SystemVerilog assertion and testbench enhancements to Verilog. This book presents the numerous verification constructs in SystemVerilog, which are not covered in this book. Together, the two books provide a comprehensive look at the extensive set of extensions that SystemVerilog adds to the Verilog language. For more information, refer to the publisher's web site: *www.springer.com/sgw/cda/frontpage/0,11855,4-40109-22-107949012-0,00.html*.

IEEE Std 1800-2005, SystemVerilog Language Reference Manual LRM)—IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language.

Copyright 2005, IEEE, Inc., New York, NY. ISBN 0-7381-4811-3. Electronic PDF form, (also available in soft cover).

This is the official SystemVerilog standard. The book is a syntax and semantics reference, not a tutorial for learning SystemVerilog. For information on ordering, visit the web site: *http://shop.ieee.org/store* and search for SystemVerilog.

IEEE Std 1364-2005, Verilog Language Reference Manual LRM)—IEEE Standard for Verilog Hardware Description Language.

Copyright 2005, IEEE, Inc., New York, NY. ISBN 0-7381-4851-2. Electronic PDF form, (also available in soft cover).

This is the official Verilog HDL and PLI standard. The book is a syntax and semantics reference, not a tutorial for learning Verilog. For information on ordering, visit the web site: *http://shop.ieee.org/store* and search for Verilog.

1364.1-2002 IEEE Standard for Verilog Register Transfer Level Synthesis 2002—Standard syntax and semantics for Verilog HDL-based RTL synthesis.

Copyright 2002, IEEE, Inc., New York, NY. ISBN 0-7381-3501-1. Softcover, 106 pages (also available as a downloadable PDF file).

This is the official synthesizable subset of the Verilog language. For information on ordering, visit the web site: *http://shop.ieee.org/store* and search for Verilog.

Writing Testbenches Using SystemVerilog by Janick Bergeron

Copyright 2006, Springer, Norwell Massachusetts.
ISBN: 0-387-29221-7. Hardcover, 412 pages.

Provides an explanation of the many testbench extensions that SystemVerilog adds for verification, and how to use those extensions for efficient verification. For more information, refer to the publisher's web site: *www.springer.com/sgw/cda/frontpage/0,11855,4-40109-22-104242164-0,00.html.*

The Verification Methodology Manual for SystemVerilog (VMM) by Janick Bergeron, Eduard Cerny, Alan Hunter, Andrew Nightingale

Copyright 2005, Springer, Norwell Massachusetts.
ISBN: 0-387-25538-9. Hardcover, 510 pages.

A methodology book on how to use SystemVerilog for advanced verification techniques. This is an advanced-level book; It is not a tutorial for learning SystemVerilog. For more information, refer to the publisher's web site: *www.springer.com/sgw/cda/frontpage/0,11855,4-40109-22-52495600-0,00.html*.

A Practical Guide for SystemVerilog Assertions, by Srikanth Vijayaraghavan, and Meyyappan Ramanathan

Copyright 2005, Springer, Norwell Massachusetts.
ISBN: 0-387-26049-8. Hardcover, 334 pages.

Specifically covers the SystemVerilog Assertions portion of the SystemVerilog standard. For more information, refer to the publisher's web site: *www.springer.com/sgw/cda/frontpage/0,11855,4-40109-22-50493024-0,00.html*.

SystemVerilog Assertions Handbook, Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari

Copyright 2004, VhdlCohen, Palos Verdes Peninsula, California.
ISBN: 0-9705394-7-9. Softcover, 330 pages.

Presents Assertion-Based Verification techniques using the SystemVerilog Assertions portion of the SystemVerilog standard. For more information, refer to the publisher's web site: *www.abv-sva.org/#svah*.

Assertions-Based Design, Second Edition, Harry Foster, Adam Krolnik, and David Lacey

Copyright 2004, Springer, Norwell Massachusetts.
ISBN: 1-4020-8027-1. Hardcover, 414 pages.

Presents how assertions are used in the design and verification process, and illustrates the usage of OVL, PSL and SystemVerilog assertions. For more information, refer to the publisher's web site: *www.springer.com/sgw/cda/frontpage/0,11855,4-102-22-33837980-0,00.html*.

The Verilog Hardware Description Language, 5th Edition by Donald E. Thomas and Philip R. Moorby.

Copyright 2002, Kluwer Academic Publishers, Norwell MA.
ISBN: 1-4020-7089-6. Hardcover, 408 pages.

A complete book on Verilog, covering RTL modeling, behavioral modeling and gate level modeling. The book has more detail on the gate, switch and strength level aspects of Verilog than many other books. For more information, refer to the web site *www.wkap.nl/prod/b/1-4020-7089-6*.

Verilog Quickstart, A Practical Guide to Simulation and Synthesis, 3rd Edition by James M. Lee.

Copyright 2002, Kluwer Academic Publishers, Norwell MA.
ISBN: 0-7923-7672-2. Hardcover, 384 pages.

An excellent book for learning the Verilog HDL. The book teaches the basics of Verilog modeling, without getting bogged down with the more obscure aspects of the Verilog language. For more information, refer to the web site *www.wkap.nl/prod/b/0-7923-7672-2*.

Verilog 2001: A Guide to the New Features of the Verilog Hardware Description Language by Stuart Sutherland.

Copyright 2002, Kluwer Academic Publishers, Norwell MA.
ISBN: 0-7923-7568-8. Hardcover, 136 pages.

An overview of the many enhancements added as part of the IEEE 1364-2001 standard. For more information, refer to the web site *www.wkap.nl/book.htm/0-7923-7568-8*.

Acknowledgements

The authors would like to express their gratitude to all those who have helped with this book. A number of SystemVerilog experts have taken the time to review all or part of the text and examples, and provided invaluable feedback on how to make the book useful and accurate.

We would like to specifically thank those that provided invaluable feedback by reviewing this book. These reviewers of the first edition include (listed alphabetically) **Clifford E. Cummings**., **Tom Fitzpatrick**, **Dave Kelf**, **James Kenney**, **Matthew Hall**, **Monique L'Huillier**, **Phil Moorby**, **Lee Moore**, **Karen L. Pieper**, **Dave Rich**, **LeeAnn Sutherland** and **David W. Smith**. The updates made for the second edition were reviewed by **Shalom Bresticker** and **LeeAnn Sutherland**.

We also want to acknowledge the significant contribution of **Lee Moore**, who converted the *Verification Guild* ATM model shown in Chapter 11 from behavioral Verilog into synthesizable SystemVerilog. The authors also express their appreciation to **Janick Bergeron**, moderator of the *Verification Guild* on-line newsletter, for granting permission to use this ATM switch example.

Chapter 1

Introduction to SystemVerilog

This chapter provides an overview of SystemVerilog. The topics presented in this chapter include:

- The origins of SystemVerilog
- Technical donations that went into SystemVerilog
- Highlights of key SystemVerilog features

1.1 SystemVerilog origins

SystemVerilog extends Verilog

SystemVerilog is a standard set of extensions to the **IEEE 1364-2005 Verilog Standard** (commonly referred to as ***"Verilog-2005"***). The SystemVerilog extensions to the Verilog HDL that are described in this book are targeted at design and writing synthesizable models. These extensions integrate many of the features of the SUPERLOG and C languages. SystemVerilog also contains many extensions for the verification of large designs, integrating features from the SUPERLOG, VERA C, C++, and VHDL languages, along with OVA and PSL assertions. These verification assertions are in a companion book, *SystemVerilog for Verification*[1].

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

This integrated whole created by SystemVerilog greatly exceeds the sum of its individual components, creating a new type of engineering language, a **H**ardware **D**escription *and* **V**erification **L**anguage or **HDVL**. Using a single, unified language enables engineers to model large, complex designs, and verify that these designs are functionally correct.

The Accellera standards organization

SystemVerilog started as an Accellera standard

The specification of the SystemVerilog enhancements to Verilog began with a standards group under the auspices of the ***Accellera Standards Organization***, rather than directly by the IEEE. Accellera is a non-profit organization with the goal of supporting the development and use of Electronic Design Automation (EDA) languages. Accellera is the combined VHDL International and Open Verilog International organizations. Accellera helps sponsor the IEEE 1076 VHDL and IEEE 1364 Verilog standards groups. In addition, Accellera sponsors a number of committees doing research on future languages. SystemVerilog is the result of one of those Accellera committees. Accellera itself receives its funding from member companies. These companies comprise several major EDA software vendors and several major electronic design corporations. More information on Accellera, its members, and its current projects can be found at *www.accellera.org*.

SystemVerilog is based on proven technology

Accellera based the SystemVerilog enhancements to Verilog on proven technologies. Various companies have donated technology to Accellera, which has then been carefully reviewed and integrated into SystemVerilog. A major benefit of using donations of technologies is that the SystemVerilog enhancements have already been proven to work and accomplish the objective of modeling and verifying much larger designs.

1.1.1 Generations of the SystemVerilog standard

Accellera SystemVerilog 3.0 extended modeling capability

A major portion of SystemVerilog was released as an Accellera standard in June of 2002 under the title of ***SystemVerilog 3.0***. This initial release of the SystemVerilog standard allowed EDA companies to begin adding the SystemVerilog extensions to existing simulators, synthesis compilers and other engineering tools. The focus of this first release of the SystemVerilog standard was to extend the synthesizable constructs of Verilog, and to enable modeling hardware at a higher level of abstraction. These are the constructs that are addressed in this book.

SystemVerilog is the third generation of Verilog

SystemVerilog began with a version number of 3.0 to show that SystemVerilog is the third major generation of the Verilog language. Verilog-1995 is the first generation, which represents the standardization of the original Verilog language defined by Phil Moorby in the early 1980s. Verilog-2001 is the second major generation of Verilog, and SystemVerilog is the third major generation. Appendix C of this book contains more details on the history of hardware descriptions languages, and the evolution of Verilog that led up to SystemVerilog.

Accellera SystemVerilog 3.1 extends verification capability

A major update to the SystemVerilog set of extensions was released in May of 2003. This release was referred to as ***SystemVerilog 3.1***, and added a substantial number of verification capabilities to SystemVerilog. These testbench enhancements are covered in the companion book, *SystemVerilog for Verification*[1].

Accellera SystemVerilog 3.1a was donated to the IEEE

Accellera continued to refine the SystemVerilog 3.1 standard by working closely with major Electronic Design Automation (EDA) companies to ensure that the SystemVerilog specification could be implemented as intended. A few additional modeling and verification constructs were also defined. In May of 2004, a final Accellera SystemVerilog draft was ratified by Accellera, and called ***SystemVerilog 3.1a***.

SystemVerilog 3.1a was donated to the IEEE

In June of 2004, right after SystemVerilog 3.1a was ratified, Accellera donated the SystemVerilog standard to the IEEE Standards Association (IEEE-SA), which oversees the Verilog 1364 standard. Accellera worked with the IEEE to form a new standards request, to review and standardize the SystemVerilog extensions to Verilog. The project number assigned to SystemVerilog was P1800 (the “P” in IEEE standards numbers stands for “proposed”, and is dropped once the IEEE has officially approved of the standard).

IEEE 1800-2005 is the official SystemVerilog standard

The IEEE-SA formed a P1800 Working Group to review the SystemVerilog 3.1a documentation and prepare it for full IEEE standardization. The working group formed several focused committees, which met on a very aggressive schedule for the next several months. The P1800 Working Group completed its work in

1. Spear, Chris *“SystemVerilog for Verification”*, Norwell, MA: Springer 2006, 0-387-27036-1.

March of 2005, and released a ballot draft of the P1800 standard for voting on by corporate members of the IEEE-SA. The balloting and final IEEE approval process were completed in October 2005, and, in November of 2005, the official IEEE 1800-2005 standard was released to the public. See page xxvii of the Preface for information on obtaining the IEEE 1800-2005 SystemVerilog Reference Manual (LRM).

IEEE 1364-2005 is the base language for SystemVerilog 1800-2005

Prior to the donation of SystemVerilog 3.1a to the IEEE, the IEEE-SA had already begun work on the next revision of the IEEE 1364 Verilog standard. At the encouragement of Accellera, the IEEE-SA organization decided not to immediately add the SystemVerilog extensions to work already in progress for extending Verilog 1364. Instead, it was decided to keep the SystemVerilog extensions as a separate document. To ensure that the reference manual for the base Verilog language and the reference manual for the SystemVerilog extensions to Verilog remained synchronized, the IEEE-SA dissolved the 1364 Working Group and made the 1364 Verilog reference manual part of the responsibility of the 1800 SystemVerilog Working Group. The 1800 Working Group formed a subcommittee to update the 1364 Verilog standard in parallel with the specification of the P1800 SystemVerilog reference manual. For the most part, the work done on the 1364 revisions was limited to errata corrections and clarifications. Most extensions to Verilog were specified in the P1800 standard. The 1800 SystemVerilog Working Group released a ballot draft for an updated Verilog P1364 standard at the same time as the ballot draft for the new P1800 SystemVerilog standard. Both standards were approved at the same time. The 1364-2005 Verilog Language Reference Manual is the official base language for SystemVerilog 1800-2005.

1.1.2 Donations to SystemVerilog

The primary technology donations that make up SystemVerilog include:

SystemVerilog comes from several donations

- The SUPERLOG Extended Synthesizable Subset (SUPERLOG ESS), from Co-Design Automation
- The OpenVERA™ verification language from Synopsys
- PSL assertions (which began as a donation of Sugar assertions from IBM)
- OpenVERA Assertions (OVA) from Synopsys

- The DirectC and coverage Application Programming Interfaces (APIs) from Synopsys
- Separate compilation and $readmem extensions from Mentor Graphics
- Tagged unions and high-level language features from BlueSpec

SUPERLOG was donated by Co-Design

In 2001, Co-Design Automation (which was acquired by Synopsys in 2002) donated to Accellera the SUPERLOG Extended Synthesizable Subset in June of 2001. This donation makes up the majority of the hardware modeling enhancements in SystemVerilog. Accellera then organized the Verilog++ committee, which was later renamed the SystemVerilog committee, to review this donation, and create a standard set of enhancements for the Verilog HDL. Appendix C contains a more complete history of the SUPERLOG language.

OpenVERA and DirectC were donated by Synopsys

In 2002, Synopsys donated OpenVERA testbench, OpenVERA Assertions (OVA), and DirectC to Accellera, as a complement to the SUPERLOG ESS donation. These donations significantly extend the verification capabilities of the Verilog language.

The Accellera SystemVerilog committee also specified additional design and verification enhancements to the Verilog language that were not part of these core donations.

SystemVerilog is backward compatible with Verilog

Two major goals of the SystemVerilog committee within Accellera were to maintain full backward compatibility with the existing Verilog HDL, and to maintain the general look and feel of the Verilog HDL.

1.2 Key SystemVerilog enhancements for hardware design

The following list highlights some of the more significant enhancements SystemVerilog adds to the Verilog HDL for the design and verification of hardware: This list is not intended to be all inclusive of every enhancement to Verilog that is in SystemVerilog. This list just highlights a few key features that aid in writing synthesizable hardware models.

- Interfaces to encapsulate communication and protocol checking within a design

- C like data types, such as **`int`**
- User-defined types, using **`typedef`**
- Enumerated types
- Type casting
- Structures and unions
- Packages for definitions shared by multiple design blocks
- External compilation-unit scope declarations
- ++, --, += and other assignment operators
- Explicit procedural blocks
- Priority and unique decision modifiers
- Programming statement enhancements
- Pass by reference to tasks, functions and modules

1.3 Summary

SystemVerilog unifies several proven hardware design and verification languages, in the form of extensions to the Verilog HDL. These extensions provide powerful new capabilities for modeling hardware at the RTL, system and architectural levels, along with a rich set of features for verifying model functionality.

Chapter 2

SystemVerilog Declaration Spaces

*V*erilog only has limited places in which designers can declare variables and other design information. SystemVerilog extends Verilog's declaration spaces in several ways. These extensions make it much easier to model complex design data, and reduce the risk of hard-to-find coding errors. SystemVerilog also enhances how simulation time units are defined.

The topics discussed in this chapter include:

- Packages definitions and importing definitions from packages
- $unit compilation declaration space
- Declarations in unnamed blocks
- Enhanced time unit definitions

Before examining in detail the many new data types that SystemVerilog offers, it is important to know <u>*where*</u> designers can define important information that is used in a design. To illustrate these new declaration spaces, this chapter will use several SystemVerilog data types that are not discussed until the following chapters. In brief, some of the new types used in this chapter are:

> **`logic`** — a 1-bit 4-state variable, like the Verilog **`reg`** type; can be declared as any vector size (discussed in Chapter 3).
>
> **`enum`** — an enumerated net or variable with a labeled set of values; similar to the C enum type, but with additional syntax and semantics for modeling hardware (discussed in Chapter 4).

`typedef` — a user-defined data type, constructed from built-in types or other user-defined types, similar to the C typedef (discussed in Chapter 4).

`struct` — a collection of variables that can be referred to individually or collectively, similar to the C struct type (discussed in Chapter 5).

2.1 Packages

Verilog requires local declarations

In Verilog, declarations of variables, nets, tasks and functions must be declared within a module, between the **`module`**...**`endmodule`** keywords. The objects declared within a module are local to the module. For modeling purposes, these objects should be referenced within the module in which they are declared. Verilog also allows hierarchical references to these objects from other modules for verification purposes, but these cross-module references do not represent hardware behavior, and are not synthesizable. Verilog also allows local variables to be defined in named blocks (formed with **`begin`**...**`end`** or **`fork`**...**`join`**), tasks and functions. These declarations are still defined within a module, however, and, for synthesis purposes, only accessible within the module.

Verilog does not have a place to make global declarations, such as global functions. A declaration that is used in multiple design blocks must be declared in each block. This not only requires redundant declarations, but it can also lead to errors if a declaration, such as a function, is changed in one design block, but not in another design block that is supposed to have the same function. Many designers use include files and other coding tricks to work around this shortcoming, but that, too, can lead to coding errors and design maintenance problems.

SystemVerilog adds user-defined types to Verilog

SystemVerilog adds user-defined types, using **`typedef`**. It is often desirable to use the definition of user-defined types in multiple modules. Using Verilog rules, where declarations are always local to a module, it would be necessary to duplicate a user-defined type definition in each and every module in which the definition is used. Redundant local definitions would not be desirable for user-defined types.

2.1.1 Package definitions

SystemVerilog adds packages to Verilog

To enable sharing a user-defined type definition across multiple modules, SystemVerilog adds ***packages*** to the Verilog language. The concept of packages is leveraged from the VHDL language. SystemVerilog packages are defined between the keywords **`package`** and **`endpackage`**.

The synthesizable constructs that a packages can contain are:

- **`parameter`** and **`localparam`** constant definitions
- **`const`** variable definitions
- **`typedef`** user-defined types
- Fully automatic **`task`** and **`function`** definitions
- **`import`** statements from other packages
- Operator overload definitions

Packages can also contain global variable declarations, static task definitions and static function definitions. These are not synthesizable, however, and are not covered in this book.

package definitions are independent of modules

A package is a separate declaration space. It is not embedded within a Verilog module. A simple example of a package definition is:

Example 2-1: A package definition

```
package definitions;

  parameter VERSION = "1.1";

  typedef enum {ADD, SUB, MUL} opcodes_t;

  typedef struct {
    logic [31:0] a, b;
    opcodes_t    opcode;
  } instruction_t;

  function automatic [31:0] multiplier (input [31:0] a, b);
    // code for a custom 32-bit multiplier goes here
    return a * b;  // abstract multiplier (no error detection)
  endfunction
endpackage
```

parameters in packages cannot be redefined

Packages can contain **parameter**, **localparam** and **const** constant declarations. The **parameter** and **localparam** constants are Verilog constructs. A **const** constant is a SystemVerilog constant, which is discussed in section 3.10 on page 71. In Verilog, a **parameter** constant can be redefined for each instance of a module, whereas a **localparam** cannot be directly redefined. In a package, however, a **parameter** constant cannot be redefined, since it is not part of a module instance. In a package, **parameter** and **localparam** are synonymous.

2.1.2 Referencing package contents

Modules and interfaces can reference the definitions and declarations in a package four ways:

- Direct reference using a scope resolution operator
- Import specific package items into the module or interface
- Wildcard import package items into the module or interface
- Import package items into the `$unit` declaration space

The first three methods are discussed in this section. Importing into `$unit` is discussed later in this chapter, in section 2.2 on page 14.

Package references using the scope resolution operator

`::` is used to reference items in packages

SystemVerilog adds a `::` "*scope resolution operator*" to Verilog. This operator allows directly referencing a package by the package name, and then selecting a specific definition or declaration within the package. The package name and package item name are separated by double colons (`::`). For example, a SystemVerilog module port can be defined as an `instruction_t` type, where `instruction_t` is defined in the package `definitions`, illustrated in example 2-1 on page 9.

Example 2-2: Explicit package references using the `::` scope resolution operator

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
  always_ff @(posedge clock) begin
```

```
    case (IW.opcode)
      definitions::ADD : result = IW.a + IW.b;
      definitions::SUB : result = IW.a - IW.b;
      definitions::MUL : result = definitions::
                                    multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

Explicit package reference help document source code

Explicitly referencing package contents can help to document the design source code. In example 2-2, above, the use of the package name makes it is very obvious where the definitions for `instruction_t`, `ADD`, `SUB`, `MUL` and `multiplier` can be found. However, when a package item, or items, needs to be referenced many times in a module, explicitly referencing the package name each time may be too verbose. In this case, it may be desirable to import package items into the design block.

Importing specific package items

import statements make package items visible locally

SystemVerilog allows specific package items to be imported into a module, using an **`import`** statement. When a package definition or declaration is imported into a module or interface, that item becomes visible within the module or interface, as if it were a locally defined name within that module or interface. It is no longer necessary to explicitly reference the package name each time that package item is referenced.

Importing a package definition or declaration can simplify the code within a module. Example 2-2 is modified below as example 2-3, using import statements to make the enumerated type labels local names within the module. The case statement can then reference these names without having to explicitly name the package each time.

Example 2-3: Importing specific package items into a module

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
```

```
  import definitions::ADD;
  import definitions::SUB;
  import definitions::MUL;
  import definitions::multiplier;

  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

NOTE Importing an enumerated type definition does not import the labels used within that definition.

In example 2-3, above, the following import statement would not work:

```
import definitions::opcode_t;
```

enumerated labels must be imported in order to reference locally

This import statement would make the user-defined type, `opcode_t`, visible in the module. However, it would not make the enumerated labels used within `opcode_t` visible. Each enumerated label must be explicitly imported, in order for the labels to become visible as local names within the module. When there are many items to import from a package, using a wildcard import may be more practical.

Wildcard import of package items

all items in a package can be made visible using a wildcard

SystemVerilog allows package items to be imported using a wildcard, instead of naming specific package items. The wildcard token is an asterisk (`*`). For example:

```
import definitions::*;  // wildcard import
```

NOTE A wildcard import does not automatically import all package contents.

wildcard imports do not automatically import the entire package

When package items are imported using a wildcard, only items actually used in the module or interface are actually imported. Definitions and declarations in the package that are not referenced are not imported.

Local definitions and declarations within a module or interface take precedence over a wildcard import. An import that specifically names package items also takes precedence over a wildcard import. From a designer's point of view, a wildcard import simply adds the package to the search rules for an identifier. Software tools will search for local declarations first (following Verilog search rules for within a module), and then search in any packages that were imported using a wildcard. Finally, tools will search in SystemVerilog's `$unit` declaration space. The `$unit` space is discussed in section 2.2 on page 14 of this chapter.

Example 2-4, below, uses a wildcard import statement. This effectively adds the package to the identifier search path. When the case statement references the enumerated labels of ADD, SUB, and MUL, as well as the function `multiplier`, it will find the definitions of these names in the `definitions` package.

Example 2-4: Using a package wildcard import

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
  import definitions::*;  // wildcard import

  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

In examples 2-3, and 2-4, for the IW module port, the package name must still be explicitly referenced. It is not possible to add an **`import`** statement between the module keyword and the module port definitions. There is a way to avoid having to explicitly reference the package name in a port list, however, using the `$unit` declaration space. The `$unit` space is discussed in 2.2.

2.1.3 Synthesis guidelines

for synthesis, package tasks and functions must be automatic

When a module references a task or function that is defined in a package, synthesis will duplicate the task or function functionality and treat it as if it had been defined within the module. To be synthesizable, tasks and functions defined in a package must be declared as **`automatic`**, and cannot contain static variables. This is because storage for an automatic task or function is effectively allocated each time it is called. Thus, each module that references an automatic task or function in a package sees a unique copy of the task or function storage that is not shared by any other module. This ensures that the simulation behavior of the pre-synthesis reference to the package task or function will be the same as post-synthesis behavior, where the functionality of the task or function has been implemented within one or more modules.

For similar reasons, synthesis does not support variables declarations in packages. In simulation, a package variable will be shared by all modules that import the variable. One module can write to the variable, and another module will see the new value. This type of inter-module communication without passing values through module ports is not synthesizable.

2.2 $unit compilation-unit declarations

SystemVerilog has compilation units

SystemVerilog adds a concept called a ***compilation unit*** to Verilog. A compilation unit is all source files that are compiled at the same time. Compilation units provide a means for software tools to separately compile sub-blocks of an overall design. A sub-block might comprise a single module or multiple modules. The modules might be contained in a single file or in multiple files. A sub-block of a design might also contain interface blocks (presented in Chapter 10) and testbench program blocks (covered in the companion book, *SystemVerilog for Verification*[1]).

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

compilation-unit scopes contain external declarations

SystemVerilog extends Verilog's declaration space by allowing declarations to be made outside of package, module, interface and program block boundaries. These external declarations are in a ***compilation-unit scope***, and are visible to all modules that are compiled at the same time.

The compilation-unit scope can contain:

- Time unit and precision declarations (see 2.4 on page 28)
- Variable declarations
- Net declarations
- Constant declarations
- User-defined data types, using **`typedef`**, **`enum`** or **`class`**
- Task and function definitions

The following example illustrates external declarations of a constant, a variable, a user-defined type, and a function.

Example 2-5: External declarations in the compilation-unit scope (not synthesizable)

```
/****************** External declarations ******************/
parameter VERSION = "1.2a";   // external constant

reg resetN = 1;               // external variable (active low)

typedef struct packed {       // external user-defined type
  reg [31:0] address;
  reg [31:0] data;
  reg [ 7:0] opcode;
} instruction_word_t;

function automatic int log2 (input int n);  // external function
  if (n <=1) return(1);
  log2 = 0;
  while (n > 1) begin
    n = n/2;
    log2++;
  end
  return(log2);
endfunction

/******************** module definition *******************/
// external declaration is used to define port types
module register (output instruction_word_t  q,
                 input  instruction_word_t  d,
```

```
                input  wire                clock );

  always @(posedge clock, negedge resetN)
    if (!resetN) q <= 0;  // use external reset
    else q <= d;
endmodule
```

NOTE External compilation-unit scope declarations are not global

A declaration in the compilation-unit scope is not the same as a global declaration. A true global declaration, such as a global variable or function, would be shared by all modules that make up a design, regardless of whether or not source files are compiled separately or at the same time.

SystemVerilog's compilation-scope only exists for source files that are compiled at the same time. Each time source files are compiled, a compilation-unit scope is created that is unique to just that compilation. For example, if module `CPU` and module `controller` both reference an externally declared variable called `reset`, then two possible scenarios exist:

- If the two modules are compiled at the same time, there will be a single compilation-unit scope. The externally declared `reset` variable will be common to both modules.
- If each module were compiled separately, then there would be two compilation-unit scopes, possibly with two different `reset` variables.

In the latter scenario, the compilation that included the external declaration of `reset` would appear to compile OK. The other file, when compiled separately, would have its own, unique $unit compilation space, and would not see the declaration of `reset` from the previous compilation. Depending on the context of how `reset` is used, the second compilation might fail, due to an undeclared variable, or it might compile OK, making `reset` an implicit net. *This is a dangerous possibility!* If the second compilation succeeds by making `reset` an implicit net, there will now be two signals called `reset`, one in each compilation. The two different `reset` signals would not be connected in any way.

2.2.1 Coding guidelines

$unit should only be used for importing packages

1. Do not make any declarations in the $unit space! All declarations should be made in named packages.
2. When necessary, packages can be imported into $unit. This is useful when a module or interface contains multiple ports that are of user-defined types, and the type definitions are in a package.

Directly declaring objects in the $unit compilation-unit space can lead to design errors when files are compiled separately. It can also lead to spaghetti code if the declarations are scattered in multiple files that can be difficult to maintain, re-use, or to debug declaration errors.

2.2.2 SystemVerilog identifier search rules

Declarations in the compilation-unit scope can be referenced anywhere in the hierarchy of modules that are part of the compilation unit.

the compilation-unit scope is third in the search order

SystemVerilog defines a simple and intuitive search rule for when referencing an identifier:

1. First, search for local declarations, as defined in the IEEE 1364 Verilog standard.
2. Second, search for declarations in packages which have been wildcard imported into the current scope.
3. Third, search for declarations in the compilation-unit scope.
4. Fourth, search for declarations within the design hierarchy, following IEEE 1364 Verilog search rules.

The SystemVerilog search rules ensure that SystemVerilog is fully backward compatible with Verilog.

2.2.3 Source code order

NOTE Data identifiers and type definitions must be declared before being referenced.

Variables and nets in the compilation-unit scope

undeclared identifiers have an implicit net type

There is an important consideration when using external declarations. Verilog supports implicit type declarations, where, in specific contexts, an undeclared identifier is assumed to be a net type (typically a **wire** type). Verilog requires the type of identifiers to be explicitly declared before the identifier is referenced when the context will not infer an implicit type, or when a type other than the default net type is desired.

external declarations must be defined before use

This implicit type declaration rule affects the declaration of variables and nets in the compilation-unit scope. Software tools must encounter the external declaration before an identifier is referenced. If not, the name will be treated as an undeclared identifier, and follow the Verilog rules for implicit types.

The following example illustrates how source code order can affect the usage of a declaration external to the module. This example will not generate any type of compilation or elaboration error. For module `parity_gen`, software tools will automatically infer `parity` as an implicit net type local to the module, since the reference to `parity` comes before the external declaration for the signal. On the other hand, module `parity_check` comes after the external declaration of `parity` in the source code order. Therefore, the `parity_check` module will use the external variable declaration.

```
module parity_gen (input wire [63:0] data );
  assign parity = ^data; // parity is an
endmodule               // implicit local net


reg parity;  // external declaration is not
             // used by module parity_gen
             // because the declaration comes
             // after it has been referenced


module parity_check (input  wire [63:0] data,
                     output logic       err);
  assign err = (^data != parity); // parity is
                                  // the $unit
endmodule                         // variable
```

2.2.4 Coding guidelines for importing packages into $unit

SystemVerilog allows module ports to be declared as user-defined types. The coding style recommended in this book is to place those definitions in one or more packages. Example 2-2 on page 10, listed earlier in this chapter, illustrates this usage of packages. An excerpt of this example is repeated below.

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
```

Explicitly referencing the package as shown above can be tedious and redundant when many module ports are of user-defined types. An alternative style is to import a package into the $unit compilation-unit scope, prior to the module declaration. This makes the user-defined type definitions visible in the SystemVerilog search order. For example:

```
// import specific package items into $unit
import definitions::instruction_t;

module ALU
(input  instruction_t  IW,
 input  logic          clock,
 output logic [31:0]   result
);
```

A package can also be imported into the $unit space using a wildcard import. Keep in mind that a wildcard import does not actually import all package items. It simply adds the package to the SystemVerilog source path. The following code fragment illustrates this style.

```
// wildcard import package items into $unit
import definitions::*;

module ALU
(input  instruction_t  IW,
 input  logic          clock,
 output logic [31:0]   result
);
```

Importing packages into $unit with separate compilation

The same care must be observed when importing packages into the $unit space as with making declarations and definitions in the $unit space. When using $unit, file order dependencies can be an issue, and multiple $units can be an issue.

file order compilation dependencies

When items are imported from a package (either with specific package item imports or with a wildcard import), the import statement must occur before the package items are referenced. If the package import statements are in a different file than the module or interface that references the package items, then the file with the import statements must be listed first in the file compilation order. If the file order is not correct, then the compilation of the module or interface will either fail, or will incorrectly infer implicit nets instead of seeing the package items.

multiple file compilation versus single file compilation

Synthesis compilers, lint checkers, some simulators, and possibly other tools that can read in Verilog and SystemVerilog source code can often compile one file at a time or multiple files at a time. When multiple files are compiled as single compilation, there is a single $unit space. An import of a package (either specific package items or a wildcard import) into $unit space makes the package items visible to all modules and interfaces read in after the import statement. However, if files are compiled separately, then there will be multiple separate $unit compilation units. A package import in one $unit will not be visible in another $unit.

using import statements in every file

A solution to both of these problems with importing package items into the $unit compilation-unit space is to place the import statements in every file, before the module or interface definition. This solution works great when each file is compiled separately. However, care must still be taken when multiple files are compiled as a single compilation. It is illegal to import the same package items more than once into the same $unit space (The same as it is illegal to declare the same variable name twice in the same name space).

conditional compilation with $unit package imports

A common C programming trick can be used to make it possible to import package items into the $unit space with both single file compilation and multiple file compilation. The trick is to use conditional compilation to include the import statements the first time the statements are compiled into $unit, and not include the statements if they are encountered again in the same compilation. In order to tell if the import statements have already been compiled in the current

$unit space, a **`define** flag is set the first time the import statements are compiled.

In the following example, the `definitions` package is contained in a separate file, called `definitions.pkg` (Any file name and file extension could be used). After the endpackage keyword, the package is wildcard imported into the $unit compilation-unit space. In this way, when the package is compiled, the definitions within the package are automatically made visible in the current $unit space.

Within the `definitions.pkg` file, a flag is set to indicate when this file has been compiled. Conditional compilation surrounds the entire file contents. If the flag has not been set, then the package will be compiled and imported into $unit. If the flag is already set (indicating the package has already been compiled and imported into the current $unit space), then the contents of the file are ignored.

Example 2-6: Package with conditional compilation (file name: `definitions.pkg`)

```
`ifndef DEFS_DONE  // if the already-compiled flag is not set...
  `define DEFS_DONE  // set the flag
  package definitions;

    parameter VERSION = "1.1";

    typedef enum {ADD, SUB, MUL} opcodes_t;

    typedef struct {
      logic [31:0] a, b;
      opcodes_t    opcode;
    } instruction_t;

    function automatic [31:0] multiplier (input [31:0] a, b);
      // code for a custom 32-bit multiplier goes here
      return a * b;  // abstract multiplier (no error detection)
    endfunction
  endpackage

  import definitions::*;  // import package into $unit

`endif
```

The line:

```
`include "definitions.pkg"
```

should be placed at the beginning of every design or testbench file that needs the definitions in the package. When the design or testbench file is compiled, it will include in its compilation the package and import statement. The conditional compilation in the `definitions.pkg` file will ensure that if the package has not already been compiled and imported, it will be done. If the package has already been compiled and imported into the current $unit space, then the compilation of that file is skipped over.

NOTE For this coding style, the package file should be passed to the software tool compiler indirectly, using a **`` `include ``** compiler directive.

package compilation should be indirect, using `` `include ``

This conditional compilation style uses the Verilog **`` `include ``** directive to compile the `definitions.pkg` file as part of the compilation of some other file. This is done in order to ensure that the import statement at the end of the `definitions.pkg` file will import the package into the same $unit space being used by the compilation of the design or testbench file. If the `definitions.pkg` file were to be passed to the software tool compiler directly on that tool's command line, then the package and import statement could be compiled into a different $unit space than what the design or testbench block is using.

The file name for the example listed in 2-6 does not end with the common convention of `.v` (for Verilog source code files) or `.sv` (for SystemVerilog source code files). A file extension of `.pkg` was used to make it obvious that the file is not a design or testbench block, and therefore is not a file that should be listed on the simulator, synthesis compiler or other software tool command line. The `.pkg` extension is an arbitrary name used for this book. The extension could be other names, as well.

Examples 2-7 and 2-8 illustrate a design file and a testbench file that include the entire file in the current compilation. The items within the package are then conditionally included in the current $unit compilation-unit space using a wildcard import. This makes the package items visible throughout the module that follows, including in the module port lists.

Example 2-7: A design file that includes the conditionally-compiled package file

```
`include "definitions.pkg" // compile the package file

module ALU
(input  instruction_t  IW,
 input  logic          clock,
 output logic [31:0]   result
);
  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

Example 2-8: A testbench file that includes the conditionally-compiled package file

```
`include "definitions.pkg" // compile the package file

module test;
  instruction_t test_word;
  logic [31:0]  alu_out;
  logic         clock = 0;

  ALU dut (.IW(test_word), .result(alu_out), .clock(clock));

  always #10 clock = ~clock;

  initial begin
    @(negedge clock)
    test_word.a = 5;
    test_word.b = 7;
    test_word.opcode = ADD;
    @(negedge clock)
    $display("alu_out = %d (expected 12)", alu_out);
    $finish;
  end
endmodule
```

`include works with both sinlge-file and multi-file compilation

In a single file compilation, the package will be compiled and imported into each $unit compilation-unit. This ensures that each $unit sees the same package items. Since each $unit is unique, there will not be a name conflict from compiling the package more than once.

In a multiple file compilation, the conditional compilation ensures that the package is only compiled and imported once into the common $unit compilation space that is shared by all modules. Whichever design or testbench file is compiled first will import the package, ensuring that the package items are visible for all subsequent files.

NOTE The conditional compilation style shown in this section does not work with global variables, static tasks, and static functions.

package variables are shared variables (not synthesizable)

Packages can contain variable declarations. A package variable is shared by all design blocks (and test blocks) that import the variable. The behavior of package variables will be radically different between single file compilations and multiple file compilations. In multiple file compilations, the package is imported into a single $unit compilation space. Every design block or test block will see the same package variables. A value written to a package variable by one block will be visible to all other blocks. In single file compilations, each $unit space will have a unique variable that happens to have the same name as a variable in a different $unit space. Values written to a package variable by one design or test block will not be visible to other design or test blocks.

static tasks and functions in packages are not synthesizable

Static tasks and functions, or automatic tasks and functions with static storage, have the same potential problem. In multiple file compilations, there is a single $unit space, which will import one instance of the task or function. The static storage within the task or function is visible to all design and verification blocks. In single file compilations, each separate $unit will import a unique instance of the task or function. The static storage of the task or function will not be shared between design and test blocks.

This limitation on conditionally compiling import statements into $unit should not be a problem in models that are written for synthesis, because synthesis does not support variable declarations in packages, or static tasks and functions in packages (see section 2.1.3 on page 14).

2.2.5 Synthesis guidelines

The synthesizable constructs that can be declared within the compilation-unit scope (external to all module and interface definitions) are:

- **`typedef`** user-defined type definitions
- Automatic functions
- Automatic tasks
- **`parameter`** and **`localparam`** constants
- Package imports

using packages instead of $unit is a better coding style

While not a recommended style, user-defined types defined in the compilation-unit scope are synthesizable. A better style is to place the definitions of user-defined types in named packages. Using packages reduces the risk of spaghetti code and file order dependencies.

external tasks and functions must be automatic

Declarations of tasks and functions in the $unit compilation-unit space is also not a recommended coding style. However, tasks and functions defined in $unit are synthesizable. When a module references a task or function that is defined in the compilation-unit scope, synthesis will duplicate the task or function code and treat it as if it had been defined within the module. To be synthesizable, tasks and functions defined in the compilation-unit scope must be declared as **`automatic`**, and cannot contain static variables. This is because storage for an automatic task or function is effectively allocated each time it is called. Thus, each module that references an automatic task or function in the compilation-unit scope sees a unique copy of the task or function storage that is not shared by any other module. This ensures that the simulation behavior of the pre-synthesis reference to the compilation-unit scope task or function will be the same as post-synthesis behavior, where the functionality of the task or function has been implemented within the module.

A **`parameter`** constant defined within the compilation-unit scope cannot be redefined, since it is not part of a module instance. Synthesis treats constants declared in the compilation-unit scope as literal values. Declaring parameters in the $unit space is not a good modeling style, as the constants will not be visible to modules that are compiled separately from the file that contains the constant declarations.

2.3 Declarations in unnamed statement blocks

local variables in named blocks

Verilog allows local variables to be declared in named **begin**...**end** or **fork**...**join** blocks. A common usage of local variable declarations is to declare a temporary variable for controlling a loop. The local variable prevents the inadvertent access to a variable at the module level of the same name, but with a different usage. The following code fragment has declarations for two variables, both named i. The **for** loop in the named **begin** block will use the local variable i that is declared in that named block, and not touch the variable named i declared at the module level.

```
module chip (input clock);
  integer i;  // declaration at module level

  always @(posedge clock)
    begin: loop       // named block
      integer i;      // local variable
      for (i=0; i<=127; i=i+1) begin
        ...
      end
    end
endmodule
```

hierarchical references to local variables

A variable declared in a named block can be referenced with a hierarchical path name that includes the name of the block. Typically, only a testbench or other verification routine would reference a variable using a hierarchical path. Hierarchical references are not synthesizable, and do not represent hardware behavior. The hierarchy path to the variable within the named block can also be used by VCD (Value Change Dump) files, proprietary waveform displays, or other debug tools, in order to reference the locally declared variable. The following testbench fragment uses hierarchy paths to print the value of both the variables named i in the preceding example:

```
module test;
  reg clock;
  chip chip (.clock(clock));

  always #5 clock = ~clock;

  initial begin
    clock = 0;
    repeat (5) @(negedge clock) ;
    $display("chip.i = %0d", chip.i);
    $display("chip.loop.i = %0d", chip.loop.i);
```

```
    $finish;
  end
endmodule
```

2.3.1 Local variables in unnamed blocks

local variables in unnamed blocks

SystemVerilog extends Verilog to allow local variables to be declared in unnamed blocks. The syntax is identical to declarations in named blocks, as illustrated below:

```
module chip (input clock);
  integer i;  // declaration at module level

  always @(posedge clock)
    begin             // unnamed block
      integer i;      // local variable
      for (i=0; i<=127; i=i+1) begin
        ...
      end
    end
endmodule
```

Hierarchal references to variables in unnamed blocks

local variables in unnamed blocks have no hierarchy path

Since there is no name to the block, local variables in an unnamed block cannot be referenced hierarchically. A testbench or a VCD file cannot reference the local variable, because there is no hierarchy path to the variable.

named blocks protect local variables

Declaring variables in unnamed blocks can serve as a means of protecting the local variables from external, cross-module references. Without a hierarchy path, the local variable cannot be referenced from anywhere outside of the local scope.

inferred hierarchy paths fro debugging

This extension of allowing a variable to be declared in an unnamed scope is not unique to SystemVerilog. The Verilog language has a similar situation. User-defined primitives (UDPs) can have a variable declared internally, but the Verilog language does not require that an instance name be assigned to primitive instances. This also creates a variable in an unnamed scope. Software tools will infer an instance name in this situation, in order to allow the variable within the UDP to be referenced in the tool's debug utilities. Software tools may also assign an inferred name to an unnamed block, in order to allow the tool's waveform display or debug utilities to reference the local variables in that unnamed block. The SystemVerilog standard neither requires nor prohibits a tool inferring a scope name for unnamed blocks, just as the Verilog standard neither requires nor prohibits the inference of instance names for unnamed primitive instances.

Section 7.7 on page 192 also discusses named blocks; and section 7.8 on page 194 introduces statement names, which can also be used to provide a scope name for local variables.

2.4 Simulation time units and precision

The Verilog language does not specify time units as part of time values. Time values are simply relative to each other. A delay of 3 is larger than a delay of 1, and smaller than a delay of 10. Without time units, the following statement, a simple clock oscillator that might be used in a testbench, is somewhat ambiguous:

```
forever #5 clock = ~clock;
```

What is the period of this clock? Is it 10 picoseconds? 10 nanoseconds? 10 milliseconds? There is no information in the statement itself to answer this question. One must look elsewhere in the Verilog source code to determine what units of time the `#5` represents.

2.4.1 Verilog's timescale directive

Verilog specifies time units to the software tool

Instead of specifying the units of time with the time value, Verilog specifies time units as a command to the software tool, using a **`` `timescale ``** compiler directive. This directive has two components: the time units, and the time precision to be used. The precision component tells the software tool how many decimal places of accuracy to use.

In the following example,

```
`timescale 1ns / 10ps
```

the software tool is instructed to use time units of 1 nanosecond, and a precision of 10 picoseconds, which is 2 decimal places, relative to 1 nanosecond.

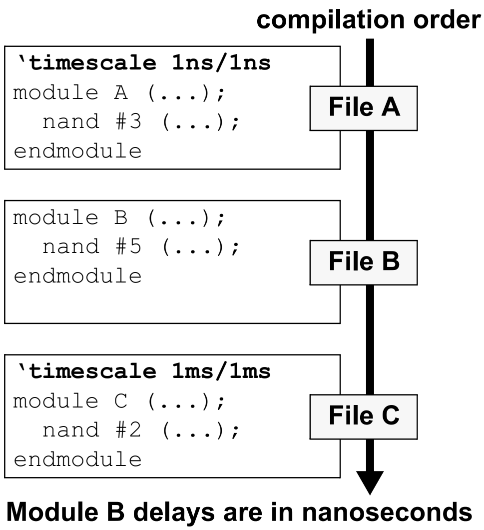
multiple 'timescale directives

The **`timescale** directive can be defined in none, one or more Verilog source files. Directives with different values can be specified for different regions of a design. When this occurs, the software tool must resolve the differences by finding a common denominator in all the time units specified, and then scaling all the delays in each region of the design to the common denominator.

the 'timescale directive is file order dependent

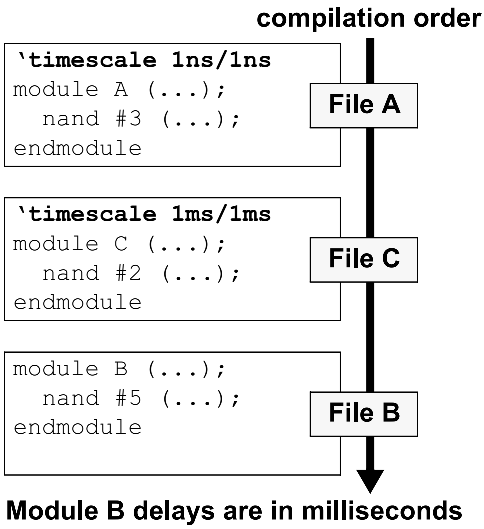
A problem with the **`timescale** directive is that the command is not bound to specific modules, or to specific files. The directive is a command to the software tool, and remains in effect until a new **`timescale** command is encountered. This creates a dependency on which order the Verilog source files are read by the software tool. Source files without a **`timescale** directive are dependent on the order in which the file is read relative to previous files.

In the following illustration, files A and C contain **`timescale** directives that set the software tool's time units and time precision for the code that follows the directives. File B, however, does not contain a **`timescale** directive.



If the source files are read in the order of File A then B and then C, the **`timescale** directive that is in effect when module B is compiled is 1 nanosecond units with 1 nanosecond precision. Therefore, the delay of 5 in module B represents a delay of 5 nanoseconds.

If the source files are read in by a compiler in a different order, however, the effects of the compiler directives could be different. The illustration below shows the file order as A then C and then B.



In this case, the **`timescale** directive in effect when module B is compiled is 1 millisecond units with 1 millisecond precision. Therefore, the delay of 5 represents 5 milliseconds. The simulation results from this second file order will be very different than the results of the first file order.

2.4.2 Time values with time units

time units specified as part of the time value

SystemVerilog extends the Verilog language by allowing time units to be specified as part of the time value.

```
forever #5ns clock = ~clock;
```

Specifying the time units as part of the time value removes all ambiguity as to what the delay represents. The preceding example is a 10 nanoseconds oscillator (5 ns high, 5 ns low).

The time units that are allowed are listed in the following table.

Table 2-1: SystemVerilog time units

Unit	Description
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	femtoseconds
step	the smallest unit of time being used by the software tool (used in SystemVerilog testbench clocking blocks)

NOTE No space is allowed between the time value and the time unit.

When specifying a time unit as part of the time value, there can be no white space between the value and time unit.

```
#3.2ps      // legal
#4.1 ps     // illegal: no space allowed
```

2.4.3 Scope-level time unit and precision

SystemVerilog allows the time units and time precision of time values to be specified locally, as part of a module, interface or program block, instead of as commands to the software tool (interfaces are discussed in Chapter 10 of this book, and program blocks are presented in the companion book, *SystemVerilog for Verification*[1]).

timeunit and timeprecision as part of module definition

In SystemVerilog, the specification of time units is further enhanced with the keywords **timeunit** and **timeprecision**. These keywords are used to specify the time unit and precision information within a module, as part of the module definition.

```
module chip (...);
  timeunit 1ns;
  timeprecision 10ps;
```

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

```
   ...
endmodule
```

The **timeunit** and **timeprecision** keywords allow binding the unit and precision information directly to a a module, interface or program block, instead of being commands to the software tool. This resolves the ambiguity and file order dependency that exist with Verilog's **`timescale** directive.

The units that can be specified with the **timeunit** and **timeprecision** keywords are the same as the units and precision that are allowed with Verilog's **`timescale** directive. These are the units that are listed in table 2-1 on page 31, except that the special **step** unit is not allowed. As with the **`timescale** directive, the units can be specified in multiples of 1, 10 or 100.

NOTE The **timeunit** and **timeprecision** statements must be specified immediately after the module, interface, or program declaration, before any other declarations or statements.

timeunit and timeprecision must be first

The specification of a module, interface or program **timeunit** and **timeprecision** must be the first statements within a module, appearing immediately after the port list, and before any other declarations or statements. Note that Verilog allows declarations within the port list. This does not affect the placement of the **timeunit** and **timeprecision** statements. These statements must still come immediately after the module declaration. For example:

```
module adder (input  wire [63:0] a, b,
              output reg  [63:0] sum,
              output reg         carry);
  timeunit 1ns;
  timeprecision 10ps;
  ...
endmodule
```

2.4.4 Compilation-unit time units and precision

external timeunit and timeprecision

The **timeunit** and/or the **timeprecision** declaration can be specified in the compilation-unit scope (described earlier in this chapter, in section 2.2 on page 14). The declarations must come before any other declarations. A **timeunit** or **timeprecision** declaration in the compilation-unit scope applies to all modules, program blocks and interfaces that do not have a local **`timeunit`** or **`timeprecision`** declaration, and which were not compiled with the Verilog **`'timescale`** directive in effect.

At most, one **`timeunit`** value and one **`timeprecision`** value can be specified in the compilation-unit scope. There can be more than one **`timeunit`** or **`timeprecision`** statements in the compilation-unit scope, as long as all statements have the same value.

Time unit and precision search order

time unit and precision search order

With SystemVerilog, the time unit and precision of a time value can be specified in multiple places. SystemVerilog defines a specific search order to determine a time value's time unit and precision:

- If specified, use the time unit specified as part of the time value.
- Else, if specified, use the local time unit and precision specified in the module, interface or program block.
- Else, if the module or interface declaration is nested within another module or interface, use the time unit and precision in use by the parent module or interface. Nested module declarations are discussed in Chapter 9 and interfaces are discussed in Chapter 10.
- Else, if specified, use the **`` `timescale ``** time unit and precision in effect when the module was compiled.
- Else, if specified, use the time unit and precision defined in the compilation-unit scope.
- Else, use the simulator's default time unit and precision.

backward compatibility

This search order allows models using the SystemVerilog extensions to be fully backward compatible with models written for Verilog.

The following example illustrates a mixture of delays with time units, **`timeunit`** and **`timeprecision`** declarations at both the module and compilation-unit scope levels, and **`'timescale`** compiler directives. The comments indicate which declaration takes precedence.

Example 2-9: Mixed declarations of time units and precision (not synthesizable)

```
timeunit 1ns;          // external time unit and precision
timeprecision 1ns;

module my_chip ( ... );

  timeprecision 1ps; // local precision (priority over external)

  always @(posedge data_request) begin
    #2.5 send_packet;  // uses external units & local precision
    #3.75ns check_crc; // specific units take precedence
  end

  task send_packet();
    ...
  endtask

  task check_crc();
  ...
  endtask
endmodule

`timescale 1ps/1ps  // directive takes precedence over external
module FSM ( ... );
  timeunit 1ns;     // local units take priority over directive

  always @(State) begin
    #1.2 case (State) // uses local units & timescale precision
      WAITE: #20ps ...;  // specific units take precedence
    ...
  end
endmodule
```

2.5 Summary

This chapter has introduced SystemVerilog packages and the $unit declaration space. Packages provide a well-defined declaration space where user-defined types, tasks, functions and constants can be defined. The definitions in a package can be imported into any number of design blocks. Specific package items can be imported, or the package definitions can be added to a design block's search path using a wildcard import.

The $unit declaration space provides a quasi global declaration space. Any definitions not contained within a design block, testbench block or package falls into the $unit compilation-unit space. Care must be taken when using $unit to avoid file order dependencies and differences between separate file compilation and multifile compilation. This chapter provided coding guidelines for the proper usage of the $unit compilation-unit space.

SystemVerilog also allows local variables to be defined in unnamed **`begin`**...**`end`** blocks. This simplifies declaring local variables, and also hides the local variable from outside the block. Local variables in unnamed blocks are protected from being read or modified from code that is not part of the block.

SystemVerilog also enhances how simulation time units and precision are specified. These enhancements eliminate the file order dependencies of Verilog's **`` `timescale ``** directive.

Chapter 3

SystemVerilog Literal Values and Built-in Data Types

SystemVerilog extends Verilog's built-in variable types, and enhances how literal values can be specified. This chapter explains these enhancements and offers recommendations on proper usage. A number of small examples illustrate these enhancements in context. Subsequent chapters contain other examples that utilize SystemVerilog's enhanced variable types and literal values. The next chapter covers another important enhancement to variable types, user-defined types.

The enhancements presented in this chapter include:

- Enhanced literal values
- `` `define `` text substitution enhancements
- Time values
- New variable types
- Signed and unsigned types
- Variable initialization
- Static and automatic variables
- Casting
- Constants

3.1 Enhanced literal value assignments

filling a vector with a literal value

In the Verilog language, a vector can be easily filled with all zeros, all Xs (unknown), or all Zs (high-impedance).

```
parameter SIZE = 64;
reg [SIZE-1:0] data;

data = 0;   // fills all bits of data with zero
data = 'bz; // fills all bits of data with Z
data = 'bx; // fills all bits of data with X
```

Each of the assignments in the example above is scalable. If the `SIZE` parameter is redefined, perhaps to 128, the assignments will automatically expand to fill the new size of `data`. However, Verilog does not provide a convenient mechanism to fill a vector with all ones. To specify a literal value with all bits set to one, a fixed size must be specified. For example:

```
data=64'hFFFFFFFFFFFFFFFF;
```

This last example is *not* scalable. If the `SIZE` constant is redefined to a larger size, such as 128, the literal value must be manually changed to reflect the new bit size of `data`. In order to make an assignment of all ones scalable, Verilog designers have had to learn coding tricks, such as using some type of operation to fill a vector with all ones, instead of specifying a literal value. The next two examples illustrate using a ones complement operator and a two's complement operator to fill a vector with all ones:

```
data = ~0;  // one's complement operation

data = -1;  // two's complement operation
```

special literal value for filling a vector

SystemVerilog enhances assignments of a literal value in two ways. First, a simpler syntax is added, that allows specifying the fill value without having to specify a radix of binary, octal or hexadecimal. Secondly, the fill value can also be a logic 1. The syntax is to specify the value with which to fill each bit, preceded by an apostrophe (`'`), which is sometimes referred to as a "*tick*". Thus:

- `'0` fills all bits on the left-hand side with 0
- `'1` fills all bits on the left-hand side with 1
- `'z` or `'Z` fills all bits on the left-hand side with z

- `'x` or `'X` fills all bits on the left-hand side with x

Note that the apostrophe character (') is not the same as the grave accent (`), which is sometimes referred to as a "*back tick*".

Using SystemVerilog, a vector of any width can be filled with all ones without hard coding the width of the value to be assigned, or using operations.

```
data = '1; // fills all bits of data with 1
```

literal values scale with the size of the left-hand side vector

This enhancement to the Verilog language simplifies writing models that work with very large vector sizes. The enhancement also makes it possible to code models that automatically scale to new vector sizes without having to modify the logic of the model. This automatic scaling is especially useful when using initializing variables that have parameterized vector widths.

3.2 `define enhancements

SystemVerilog extends the ability of Verilog's **`define** text substitution macro by allowing the macro text to include certain special characters.

3.2.1 Macro argument substitution within strings

Verilog allows the quotation mark (") to be used in a **`define** macro, but the text within the quotation marks became a literal string. This means that in Verilog, it is not possible to create a string using text substitution macros where the string contains embedded macro arguments.

In Verilog, the following example will not work as intended:

```
`define print(v) \
   $display("variable v = %h", v)

`print(data);
```

In this example, the macro `` `print() `` will expand to:

```
$display("variable v = %h", data);
```

The intent of this text substitution example is that all occurrences of the macro argument `v` will be substituted with the actual argument value, `data`. However, since the first occurrence of `v` is within quotes in the macro definition, Verilog does not substitute the first occurrence of `v` with `data`.

`" allows macro argument substitution within strings

SystemVerilog allows argument substitution inside a macro text string by preceding the quotation marks that form the string with a grave accent (` ). The example below defines a text substitution macro that represents a complete `$display` statement. The string to be printed contains a `%h` format argument. The substituted text will contain a text string that prints a message, including the name and logic value of the argument to the macro. The %h within the string will be correctly interpreted as a format argument.

```
`define print(v) \
  $display(`"variable v = %h`", v)

`print(data);
```

In this example, the macro `` `print() `` will expand to:

```
$display("variable data = %h", data);
```

In Verilog, quotation marks embedded within a string must be escaped using `\"` so as to not affect the quotation marks of the outer string. The following Verilog example embeds quotation marks within a print message.

```
$display("variable \"data\" = %h", data);
```

`\`" allows an escaped quote in a macro text string containing argument substitution

When a string is part of a text substitution macro that contains variable substitution, it is not enough to use `\"` to escape the embedded quotation marks. Instead, `` `\`" `` must be used. For example:

```
`define print(v) \
  $display(`"variable `\`"v`\`" = %h`", v)

`print(data);
```

In this example, the macro `` `print() `` will expand to:

```
$display("variable \"data\" = %h", data);
```

3.2.2 Constructing identifier names from macros

Using Verilog **`define**, it is not possible to construct an identifier name by concatenating two or more text macros together. The problem is that there will always be a white space between each portion of the constructed identifier name.

`` serves as a delimiter without a space in the macro text

SystemVerilog provides a way to delimit an identifier name without introducing a white space, using two consecutive grave accent marks, i.e. ``. This allows two or more names to be concatenated together to form a new name.

One application for `` is to simplify creating source code where a set of similar names are needed several times, and an array cannot be used. In the following example, a 2-state **bit** variable and a **wand** net need to be defined with similar names, and a continuous assignment of the variable to the net. The variable allows local procedural assignments, and the net allows wired logic assignments from multiple drivers, where one of the drivers is the 2-state variable: The **bit** type is discussed in more detail later in this chapter. In brief, the **bit** type is similar to the Verilog **reg** type, but **bit** variables only store 2-state values, whereas **reg** stores 4-state values.

In source code without text substitution, these declarations might be:

```
bit d00_bit;  wand d00_net = d00_bit;
bit d01_bit;  wand d01_net = d01_bit;

... // repeat 60 more times, for each bit

bit d62_bit;  wand d62_net = d62_bit;
bit d63_bit;  wand d63_net = d63_bit;
```

Using the SystemVerilog enhancements to **`define**, these declarations can be simplified as:

```
`define TWO_STATE_NET(name) bit name``_bit; \
  wand name``_net = name``_bit;

`TWO_STATE_NET(d00)
`TWO_STATE_NET(d01)
...
`TWO_STATE_NET(d62)
`TWO_STATE_NET(d63)
```

3.3 SystemVerilog variables

3.3.1 Object types and data types

Verilog data types

Verilog's hardware types

The Verilog language has hardware-centric variable types and net types. These types have special simulation and synthesis semantics to represent the behavior of actual connections in a chip or system.

- The Verilog **`reg`**, **`integer`** and **`time`** variables have 4 logic values for each bit: 0, 1, Z and X.
- The Verilog **`wire`**, **`wor`**, **`wand`**, and other net types have 120 values for each bit (4-state logic plus multiple strength levels) and special wired logic resolution functions.

SystemVerilog data types

data declarations have a type and a data type

Verilog does not clearly distinguish between signal types, and the value set the signals can store or transfer. In Verilog, all nets and variables use 4-state values, so a clear distinction is not necessary. To provide more flexibility in variable and net types and the values that these types can store or transfer, the SystemVerilog standard defines that signals in a design have both a ***type*** and a ***data type***.

"type" defines if data is a net or variable

Type indicates if the signal is a net or variable. SystemVerilog uses all the Verilog variable types, such as **`reg`** and **`integer`**, plus adds several more variable types, such as **`byte`** and **`int`**. SystemVerilog does not add any extensions to the Verilog net types.

"data type" defines if data is 2-state or 4-state

Data type indicates the value system of the net or variable, which is 0 or 1 for 2-state data types, and 0, 1, Z or X for 4-state data types. The SystemVerilog keyword **`bit`** defines that an object is a 2-state data type. The SystemVerilog keyword **`logic`** defines that an object is a 4-state data type. In the SystemVerilog-2005 standard, variable types can be either 2-state or 4-state data types, where as net types can only be 4-state data types.

3.3.2 SystemVerilog 4-state variables

The 4-state logic type

the Verilog reg type

The Verilog language uses the **reg** type as a general purpose variable for modeling hardware behavior in **initial** and **always** procedural blocks. The keyword **reg** is a misnomer that is often confusing to new users of the Verilog language. The term "*reg*" would seem to imply a hardware "*register*", built with some form of sequential logic flip-flops. In actuality, there is no correlation whatsoever between using a **reg** variable and the hardware that will be inferred. It is the context in which the **reg** variable is used that determines if the hardware represented is combinational logic or sequential logic.

the logic variable type replaces reg

SystemVerilog uses the more intuitive **logic** keyword to represent a general purpose, hardware-centric data type. Some example declarations using the **logic** type are:

```
logic resetN; // a 1-bit wide 4-state variable

logic [63:0] data; // a 64-bit wide variable

logic [0:7] array [0:255]; // an array of 8-bit
                                   variables
```

the logic keyword is a data type

The keyword **logic** is not actually a variable type, it is a *data type*, indicating the signal can have 4-state values. However, when the **logic** keyword is used by itself, a variable is implied. A 4-state variable can be explicitly declared using the keyword pair **var logic**. For example:

```
var logic [63:0] addr; // a 64-bit wide variable
```

A Verilog net type defaults to being a 4-state logic data type. A net can also be explicitly declared as a 4-state data type using the logic keyword. For example:

```
wire logic [63:0] data; // a 64-bit wide net
```

Explicitly declaring the data type of nets and variables is discussed in more depth in section 3.3.4 on page 47.

Semantically, a variable of the **logic** data type is identical to the Verilog **reg** type. The two keywords are synonyms, and can be used interchangeably (except that the **reg** keyword cannot be paired with net type keywords, as discussed in section 3.3.4 on page 47). Like the Verilog **reg** variable type, a variable of the **logic** data type can store 4-state logic values (0, 1, Z and X), and can be defined as a vector of any width.

Because the keyword **logic** does not convey a false implication of the type of hardware represented, **logic** is a more intuitive keyword choice for describing hardware when 4-state logic is required. In the subsequent examples in this book, the **logic** type is used in place of the Verilog **reg** type (except when the example illustrates pure Verilog code, with no SystemVerilog enhancements).

3.3.3 SystemVerilog 2-state variables

SystemVerilog's 2-state types

SystemVerilog adds several new 2-state types, suitable for modeling at more abstract levels than RTL, such as system level and transaction level. These types include:

- **bit** — a 1-bit 2-state integer
- **byte** — an 8-bit 2-state integer, similar to a C **char**
- **shortint** — a 16-bit 2-state integer, similar to a C **short**
- **int** — a 32-bit 2-state integer, similar to a C **int**
- **longint** — a 64-bit 2-state integer, similar to a C **longlong**

Using the 2-state bit type

Abstract modeling levels do not need 4-state values

Variables of the **reg** or **logic** data types are used for modeling hardware behavior in procedural blocks. These types store 4-state logic values, 0, 1, Z and X. 4-state types are the preferred types for synthesizable RTL hardware models. The Z value is used to represent unconnected or tri-state design logic. The X value helps detect and isolate design errors. At higher levels of modeling, such as the system and transaction levels, logic values of Z and X are seldom required.

a 2-state bit variable can be used in place of reg or logic

SystemVerilog allows variables to be declared as a **bit** data type. Syntactically, a **bit** variable can be used any place **reg** or **logic** variables can be used. However, the **bit** data type is semantically different, in that it only stores 2-state values of 0 and 1. The **bit** data type can be useful for modeling hardware at higher levels of abstraction.

Variables of the **bit** data type can be declared in the same way as **reg** and **logic** types. Declarations can be any vector width, from 1-bit wide to the maximum size supported by the software tool (the IEEE 1364 Verilog standard defines that all compliant software tools should support vector widths of at least 2^{16} bits wide).

```
bit resetN; // a 1-bit wide 2-state variable

bit [63:0] data; // a 64-bit 2-state variable

bit [0:7] array [0:255]; // an array of 8-bit
                             2-state variables
```

the bit keyword is a data type

The keyword **bit** is not actually a variable type, it is a *data type*, indicating the variable can have 2-state values. However, when the **bit** keyword is used by itself, a variable is implied. A 2-state variable can also be explicitly declared using the keyword pair **var bit**. For example:

```
var bit [63:0] addr; // a 64-bit wide variable
```

Explicitly declaring the data type of variables is discussed in more depth in section 3.3.4 on page 47.

Using the C-like types

2-state types can be used to interface to C and C++ models

A primary usage for the C-like 2-state types, such as **int** and **byte**, is for modeling more abstract bus-functional models. At this level, it is not necessary for the model to represent detailed hardware such as tri-state busses and hardware resolution that can result in logic X values. Another key usage of these C-like types is for interfacing Verilog models to C or C++ models using SystemVerilog's Direct Programming Interface (DPI). Using types that have a common representation in both languages makes it simple and efficient to pass data back and forth between the languages.

the int type can be used as a for-loop control variable

Another common usage of the **int** type can be as the loop-control variable in **for** loops. In synthesizable RTL models, the loop control variable is typically just a temporary variable that disappears in the synthesized gate-level representation of a design. As such, loop control variables do not need 4-state values. The **int** type works well as the control variable in **`for`** loops for both abstract models and synthesizable RTL models.

2-state simulation semantics

4-state types begin simulation with a logic X

The 4-state variables, such as **`reg`**, **`logic`**, and **`integer`**, default to beginning simulation with all bits at a logic X. These variables are considered uninitialized, and, therefore, at an unknown value until a first value is assigned to the variable (for example, by the design reset logic). 4-state variables can be defined to begin simulation with some other value using in-line initialization, but this is not synthesizable. In-line initialization is discussed more in section 3.8.

2-state types begin simulation with a logic 0

All 2-state date types begin simulation with a logic 0. Since 2-state types do not store an X value, they cannot represent an unitialized state. This is one of the reasons that it is preferable to use 4-state types to represent synthesizable RTL models.

X and Z values are mapped to 0 in 2-state types

It is legal to assign 4-state values to 2-state variables. For example, the value of a 4-state input to a model can be assigned to a 2-state **`bit`** type within the module. Any bits that have an X or Z value in the 4-state type will be translated to a logic 0 in the matching bit position of the 2-state variable.

Other abstract types

a void type represents no storage

SystemVerilog adds a **`void`** type that indicates no storage. The **`void`** type can be used in tagged unions (see Chapter 5) and to define functions that do not return a value (see Chapter 6).

shortreal is equivalent to a C float

SystemVerilog also adds a **`shortreal`** variable type that compliments Verilog's **`real`** type. **`shortreal`** stores a 32-bit single-precision floating point, the same as a C **`float`**, whereas the Verilog **`real`** stores a double-precision variable, the same as a C **`double`**. The **`real`** and **`shortreal`** types are not synthesizable, but can be useful in abstract hardware models and in testbenches.

The verification enhancements in SystemVerilog add classes and other dynamic types for use in high-level testbenches. These types are not covered in this book.

3.3.4 Explicit and implicit variable and net data types

SystemVerilog has net and variable types, and 2-state or 4-state data types

In SystemVerilog terminology, variables and nets are *types* which can have either a 2-state or 4-state *data type* (In the 2005 SystemVerilog standard, nets can only have a 4-state data type). A 4-state data type is represented with the keyword **logic**. A 2-state data type is represented with the keyword **bit**. When these 4-state or 2-state data types are used without explicitly specifying that the data type is a variable or net, an implicit *variable* is inferred.

```
logic [7:0] busA;  // infers a variable that is
                   // a 4-state data type

bit [31:0] busB;  // infers a variable that is
                  // a 2-state data type
```

The Verilog keywords **integer** and **time** are variables that are 4-state data types with predefined vector sizes. The SystemVerilog keywords **int**, **byte**, **shortint** and **longint** are variables that are 2-state data types with predefined vector sizes.

SystemVerilog allows an optional **var** keyword to be specified before any of the data types. For example:

```
var logic [7:0] a;   // 4-state 8-bit variable

var bit [31:0] b;    // 2-state 32-bit variable

var int i;           // 2-state 32-bit variable
```

"var" is short for "variable"

The **var** keyword (short for "*variable*") documents that the object is a variable. The **var** keyword does not affect how a variable behaves in simulation or synthesis. Its usage is to help make code more self-documenting. This explicit documentation can help make code more readable and maintainable when variables are created from user-defined types. For example:

```
typedef enum bit {FALSE, TRUE} bool_t;
var bool_t c;  // variable of user-defined type
```

A variable can also be declared using **var** without an explicit data type. In this case, the variable is assumed to be of the **logic** data type.

```
var [7:0] d;         // 4-state 8-bit variable
```

All Verilog *net types* (**wire**, **uwire**, **wand**, **wor**, **tri**, **triand**, **trior**, **tri0**, **tri1**, **trireg**, **supply0** and **supply1**) are implicitly of a 4-state logic data type. There are no 2-state net types.

```
wire [31:0] busB; // declares a net type
                  // that is implicitly a
                  // a 4-state logic data type
```

Optionally, a net can be declared using both the net type and the **logic** data type:

```
wire logic [31:0] busC;
```

To prevent confusing combinations of keywords, SystemVerilog does not allow the keyword **reg** to be directly paired with any of the net type keywords.

```
wire reg [31:0] busD; // ILLEGAL keyword pair
```

3.3.5 Synthesis guidelines

2-state types synthesize the same as 4-state types

The 4-state **logic** type and the 2-state **bit**, **byte**, **shortint**, **int**, and **longint** types are synthesizable. Synthesis compilers treat 2-state and 4-state types the same way. The use of 2-state types primarily affects simulation.

synthesis ignores the default initial value of 2-state types

2-state types begin simulation with a default value of logic value of 0. Synthesis ignores this default initial value. The post-synthesis design realized by synthesis is not guaranteed to power up with zeros in the same way that pre-synthesis models using 2-state types will appear to power up.

Section 8.2 on page 219 presents additional modeling considerations regarding the default initial value of 2-state types.

3.4 Using 2-state types in RTL models

2-state types simulate differently than 4-state types. The initial value of 2-state types at simulation time 0 is different than 4-state types, and the propagation of ambiguous or faulty logic (typically indicated by a logic X in simulation) is different. This section discusses some of the considerations designers should be aware of when 2-state types are used in RTL hardware models.

3.4.1 2-state type characteristics

SystemVerilog adds 2-state types

SystemVerilog adds several 2-state types to the Verilog language: **`bit`** (1-bit wide), **`byte`** (8-bits wide), **`shortint`** (16-bits wide), **`int`** (32-bits wide) and **`longint`** (64-bits wide). These 2-state types allow modeling designs at an abstract level, where tri-state values are seldom required, and where circuit conditions that can lead to unknown or unpredictable values—represented by a logic X— cannot occur.

mapping 4-state values to 2-state

SystemVerilog allows freely mixing 2-state and 4-state types within a module. Verilog is a loosely-typed language, and this characteristic is also true for SystemVerilog's 2-state types. Thus, it is possible to assign a 4-state value to a 2-state type. When this occurs, the 4-state value is mapped to a 2-state value as shown in the following table:

Table 3-1: Conversion of 4-state values to 2-state values

4-state Value	Converts To
0	0
1	1
Z	0
X	0

3.4.2 2-state types versus 2-state simulation

tool-specific 2-state modes

Some software tools, simulators in particular, offer a 2-state mode for when the design models do not require the use of logic Z or X. These 2-state modes allow simulators to optimize simulation data structures and algorithms and can achieve faster simulation run times. SystemVerilog's 2-state types permit software tools to make the same types of optimizations. However, SystemVerilog's 2-state types have important advantages over 2-state simulation modes.

SystemVerilog standardizes mixing 2-state and 4-state types

The software tools that provide 2-state modes typically use an invocation option to specify using the 2-state mode algorithms. Invocation options are often globally applied to all files listed in the invocation command. This makes it difficult to have a mix of 2-state logic and 4-state logic. Some software tools provide a more flexible control, by allowing some modules to be compiled in 2-state mode, and others in the normal 4-state mode. These tools may also use tool-specific pragmas or other proprietary mechanisms to allow specific variables within a module to be specified as using 2-state or 4-state modes. All of these proprietary mechanisms are tool-specific, and differ from one software tool to another. SystemVerilog's 2-state types give the designer a standard way to specify which parts of a model should use 2-state logic and which parts should use 4-state logic.

SystemVerilog 2-state to 4-state mapping is standardized

With 2-state simulation modes, the algorithm for how to map a logic Z or logic X value to a 2-state value is proprietary to the software tool, and is not standardized. Different simulators can, and do, map values differently. For example, some commercial simulators will map a logic X to a 0, while others map a logic X to a 1. The different algorithms used by different software tools means that the simulation results of the same model may not be the same. SystemVerilog's 2-state types have a standard mapping algorithm, providing consistent results from all software tools.

SystemVerilog 2-state initialization is standardized

Another difference between 2-state modes and 2-state types involves the initialization of a variable to its 2-state value. The IEEE 1364 Verilog standard specifies that 4-state variables begin simulation with a logic X, indicating the variable has not been initialized. The first time the 4-state variable is initialized to a 0 or 1 will cause a simulation event, which can trigger other activity in the design. Whether or not the event propagates to other parts of the design depends in part on nondeterministic event ordering. Most of the proprietary 2-state mode algorithms will change the initial value of 4-state variables to be a logic 0 instead of a logic X, but there is no standard on when the initialization occurs. Some simulators with 2-state modes will set the initial value of the variable without causing a simulation event. Other simulators will cause a simulation event at time zero as the initial value is changed from X to 0, which may propagate to other constructs sensitive to negative edge transitions. The differences in these proprietary 2-state mode algorithms can lead to differences in simulation results between different software tools. The SystemVerilog 2-state variables are specifically defined to begin simulation with a logic value of 0 without causing a simulation event. This standard rule ensures consistent behavior in all software tools.

SystemVerilog 2-state is standardized

The Verilog **casez** and **casex** decision statements can be affected by 2-state simulation modes. The **casez** statement treats a logic Z as a don't care value instead of high-impedance. The **casex** statement treats both a logic X and a logic Z as don't care. When a proprietary 2-state mode algorithm is used, there is no standard to define how **casez** and **casex** statements will be affected. Furthermore, since these simulation modes only change the 4-state behavior within one particular tool, some other tool that might not have a 2-state mode might interpret the behavior of the same model differently. SystemVerilog's standard 2-state types have defined semantics that provide deterministic behavior with all software tools.

3.4.3 Using 2-state types with case statements

At the abstract RTL level of modeling, logic X is often used as a flag within a model to show an unexpected condition. For example, a common modeling style with Verilog **case** statements is to make the default branch assign outputs to a logic X, as illustrated in the following code fragment:

```
case (State)
  RESET:   Next = WAITE;
  WAITE:   Next = LOAD;
  LOAD:    Next = DONE;
  DONE:    Next = WAITE;
  default: Next = 4'bx; // unknown state
endcase
```

The default assignment of a logic X serves two purposes. Synthesis treats the default logic X assignment as a special flag, indicating that, for any condition not covered by the other case selection items, the output value is "don't care". Synthesis will optimize the decode logic for the case selection items, without concern for what is decoded for case expression values that would fall into the default branch. This can provide better optimizations for the explicitly defined case selection items, but at the expense of indeterminate results, should an undefined case expression value occur.

Within simulation, the default assignment of logic X serves as an obvious run-time error, should an unexpected case expression value occur. This can help trap design errors in the RTL models. However, this advantage is lost after synthesis, as the post-synthesis model will not output logic X values for unexpected case expression values.

Assigning a logic X to a 2-state variable is legal. However, the assignment of a logic X to a variable will result in the variable having a value of 0 instead of an X. If the `State` or `Next` variables are 2-state types, and if a value of 0 is a legitimate value for `State` or `Next`, then the advantage of using an X assignment to trap design errors at the RTL level is lost. The default X assignment will still allow synthesis compilers to optimize the decode logic for the case selection items. This means that the post-synthesis behavior of the design will not be the same, because the optimized decoding will probably not result in a 0 for undefined case expression values.

3.5 Relaxation of type rules

Verilog restricts usage of variables and nets

In Verilog, there are strict semantic restrictions regarding where variable types such as **`reg`** can be used, and where net types such as **`wire`** can be used. When to use **`reg`** and when to use **`wire`** is based entirely on the context of how the signal is used within the model. The general rule of thumb is that a variable must be used when modeling using **`initial`** and **`always`** procedural blocks, and a net must be used when modeling using continuous assignments, module instances or primitive instances.

These restrictions on type usage are often frustrating to engineers who are first learning the Verilog language. The restrictions also make it difficult to evolve a model from abstract system level to RTL to gate level because, as the context of the model changes, the type declarations may also have to be changed.

SystemVerilog relaxes restrictions on using variables

SystemVerilog greatly simplifies determining the proper type to use in a model, by relaxing the rules of where variables can be used. With SystemVerilog, a variable can receive a value in any one of the following ways, but no more than one of the following ways:

- Be assigned a value from any number of **`initial`** or **`always`** procedural blocks (the same rule as in Verilog).
- Be assigned a value from a single **`always_comb`**, **`always_ff`** or **`always_latch`** procedural block. These SystemVerilog procedural blocks are discussed in Chapter 6.
- Be assigned a value from a single continuous assignment statement.
- Receive a value from a single module or primitive output or inout

port.

most signals can be declared as logic or bit

These relaxed rules for using variables allow most signals in a model to be declared as a variable. It is not necessary to first determine the context in which that signal will be used. The type of the signal does not need to be changed as the model evolves from system level to RTL to gate level.

The following simple example illustrates the use of variables under these relaxed type rules.

Example 3-1: Relaxed usage of variables

```
module compare (output logic        lt, eq, gt,
                input  logic [63:0] a, b );

  always @(a, b)
    if (a < b) lt = 1'b1;     // procedural assignments
  else         lt = 1'b0;

  assign gt = (a > b);        // continuous assignments

  comparator u1 (eq, a, b);   // module instance

endmodule

module comparator (output logic  eq,
                   input  [63:0] a, b);
  always @(a, b)
    eq = (a==b);
endmodule
```

Restrictions on variables can prevent design errors

NOTE Variables cannot be driven by multiple sources.

SystemVerilog restrictions on using variables

It is important to note that though SystemVerilog allows variables to be used in places where Verilog does not, SystemVerilog does still have some restrictions on the usage of variables.

SystemVerilog makes it an error to have multiple output ports or multiple continuous assignments write to the same variable, or to combine procedural assignments with continuous assignments or output drivers on the same variable.

The reason for these restrictions is that variables do not have built-in resolution functionality to resolve a final value when two or more devices drive the same output. Only the Verilog net types, such as **`wire`**, **`wand`** (wire-and) and **`wor`** (wire-or), have built-in resolution functions to resolve multi-driver logic. (The Verilog-2005 standard also has a **`uwire`** net type, which restricts its usage to a single driver, the same as with variables.)

Example 3-2: Illegal use of variables

```
module add_and_increment (output logic [63:0] sum,
                          output logic        carry,
                          input  logic [63:0] a, b );
  always @(a, b)
    sum = a + b;         // procedural assignment to sum

  assign sum = sum + 1;  // ERROR! sum is already being
                         // assigned a value

  look_ahead i1 (carry, a, b);  // module instance drives carry

  overflow_check i2 (carry, a, b); // ERROR! 2nd driver of carry
endmodule

module look_ahead (output wire        carry,
                   input  logic [63:0] a, b);
  ...
endmodule

module overflow_check (output wire        carry,
                       input  logic [63:0] a, b);
  ...
endmodule
```

TIP

Use variables for single-driver logic, and use nets for multi-driver logic.

SystemVerilog's restriction that variables cannot receive values from multiple sources can help prevent design errors. Wherever a signal in a design should only have a single source, a variable can be used. The single source can be procedural block assignments, a single continuous assignment, or a single output/inout port of a module or primitive. Should a second source inadvertently be connected to the same signal, it will be detected as an error, because each variable can only have a single source.

SystemVerilog does permit a variable to be written to by multiple **`always`** procedural blocks, which can be considered a form of multiple sources. This condition must be allowed for backward compatibility with the Verilog language. Chapter 6 introduces three new types of procedural blocks: **`always_comb`**, **`always_latch`** and **`always_ff`**. These new procedural blocks have the restriction that a variable can only be assigned from one procedural block. This further enforces the checking that a signal declared as a variable only has a single source.

Only nets can have multiple sources, such as multiple continuous assignments and/or connections to multiple output ports of module or primitive instances. Therefore, a signal in a design such as a data bus or address bus that can be driven from several devices should be declared as a Verilog net type, such as **`wire`**. Bi-directional module ports, which can be used as both an input and an output, must also be declared as a net type.

It is also illegal to write to an automatic variable from a continuous assignment or a module output. Only static variables can be continuously assigned or connected to an output port. Static variables are required because the variable must be present throughout simulation in order to continuously write to it. Automatic variables do not necessarily exist the entire time simulation is running.

3.6 Signed and unsigned modifiers

Verilog-1995 signed types

The first IEEE Verilog standard, Verilog-1995, had just one signed type, declared with the keyword **`integer`**. This type has a fixed size of 32 bits in most, if not all, software tools that support Verilog. Because of this, and some limitations of literal numbers, Verilog-1995 was limited to doing signed operations on just 32-bit wide vectors. Signed operations could be performed on other vector sizes by manually testing and manipulating a sign bit (the way it is done in actual hardware), but this required many lines of extra code, and could introduce coding errors that are difficult to detect.

Verilog signed types

The IEEE Verilog-2001 standard added several significant enhancements to allow signed arithmetic operations on any type and with any vector size. The enhancement that affects types is the ability to declare any type as **signed**. This modifier overrides the default definition of unsigned types in Verilog. For example:

```
reg [63:0] u; // unsigned 64-bit variable
reg signed [63:0] s; // signed 64-bit variable
```

SystemVerilog signed and unsigned types

SystemVerilog adds new types that are signed by default. These signed types are: **byte**, **shortint**, **int**, and **longint**. SystemVerilog provides a mechanism to explicitly override the signed behavior of these new types, using the **unsigned** keyword.

```
int s_int; // signed 32-bit variable
int unsigned u_int; // unsigned 32-bit variable
```

NOTE SystemVerilog's signed declaration is not the same as C's.

The C language allows the **signed** or **unsigned** keyword to be specified before or after the type keyword.

```
unsigned int u1;  /* legal C declaration */

int unsigned u2;  /* legal C declaration */
```

Verilog places the **signed** keyword (Verilog does not have an **unsigned** keyword) after the type declaration, as in:

```
reg signed [31:0] s;  // Verilog declaration
```

SystemVerilog also only allows the **signed** or **unsigned** keyword to be specified after the type keyword. This is consistent with Verilog, but different than C.

```
int unsigned u;  // SystemVerilog declaration
```

3.7 Static and automatic variables

Verilog-1995 variables are static

In the Verilog-1995 standard, all variables are static, with the expectation that these variables are for modeling hardware, which is also static in nature.

Verilog-2001 has automatic variables in tasks and functions

The Verilog-2001 standard added the ability to define variables in a task or function as **`automatic`**, meaning that the variable storage is dynamically allocated by the software tool when required, and deallocated when no longer needed. Automatic variables—also referred to as dynamic variables—are primarily intended for representing verification routines in a testbench, or in abstract system-level, transaction-level or bus-functional models. One usage of automatic variables is for coding a re-entrant task, so that the task can be called while a previous call of the task is still running.

Automatic variables also allow coding recursive function calls, where a function calls itself. Each time a task or function with automatic variables is called, new variable storage is created. When the call exits, the storage is destroyed. The following example illustrates a balance adder that adds the elements of an array together. The low address and high address of the array elements to be added are passed in as arguments. The function then recursively calls itself to add the array elements. In this example, the arguments `lo` and `hi` are automatic, as well as the internal variable `mid`. Therefore, each recursive call allocates new variables for that specific call.

```
function automatic int b_add (int lo, hi);
  int mid = (lo + hi + 1) >> 1;
  if (lo + 1 != hi)
    return(b_add(lo,(mid-1)) + b_add(mid,hi));
  else
    return(array[lo] + array[hi]);
endfunction
```

In Verilog, automatic variables are declared by declaring the entire task or function as automatic. All variables in an automatic task or function are dynamic.

SystemVerilog adds static and automatic variable declarations

SystemVerilog extends the ability to declare static and automatic variables. SystemVerilog adds a **`static`** keyword, and allows any variable to be explicitly declared as either **`static`** or **`automatic`**. This declaration is part of the variable declaration, and can appear within tasks, functions, **`begin`**...**`end`** blocks, or **`fork`**...**`join`** blocks. Note that variables declared at the module level cannot be explicitly declared as **`static`** or **`automatic`**. At the module level, all variables are static.

The following code fragment illustrates explicit automatic declarations in a static function:

```
function int count_ones (input [31:0] data);
  automatic logic [31:0] count = 0;
  automatic logic [31:0] temp = data;

  for (int i=0; i<=32; i++) begin
    if (temp[0]) count++;
    temp >>= 1;
  end
  return count;
endfunction
```

The next example illustrates an explicit static variable in an automatic task. Automatic tasks are often used in verification to allow test code to call a task while a previous call to the task is still executing. This example checks a value for errors, and increments an error count each time an error is detected. If the `error_count` variable were automatic as is the rest of the task, it would be recreated each time the task was called, and only hold the error count for that call of the task. As a static variable, however, `error_count` retains its value from one call of the task to the next, and can thereby keep a running total of all errors.

```
typedef struct packed {...} packet_t;
task automatic check_results
  (input  packet_t sent, received);
   output int      total_errors);
  static int error_count;
  ...
  if (sent !== received) error_count++;
  total_errors = error_count;
endtask
```

backward compatibility

The defaults for storage in SystemVerilog are backward compatible with Verilog. In modules, **begin**...**end** blocks, **fork**...**join** blocks, and non-automatic tasks and functions, all storage defaults to static, unless explicitly declared as automatic. This default behavior is the same as the static storage in Verilog modules, **begin**...**end** or **fork**...**join** blocks and non-automatic tasks and functions. If a task or function is declared as **automatic**, the default storage for all variables will be automatic, unless explicitly declared as static. This default behavior is the same as with Verilog, where all storage in an automatic task or function is automatic.

3.7.1 Static and automatic variable initialization

Verilog variable in-line variable initialization

Verilog only permits in-line variable initialization for variables declared at the module level. Variables declared in tasks, functions and **begin**...**end** or **fork**...**join** blocks cannot have an initial value specified as part of the variable declaration.

SystemVerilog in-line variable initialization

initializing automatic variables

SystemVerilog extends Verilog to allow variables declared within tasks and functions to be declared with in-line initial values.

A variable declared in a non-automatic task or function will be static by default. An in-line initial value will be assigned one time, before the start of simulation. Calls to the task or function will not re-initialize the variable.

NOTE Initializing static variables in a task or function is not considered synthesizable, and may not be supported in some tools.

static variables are only initialized once

The following example will not work correctly. The `count_ones` function is static, and therefore all storage within the function is also static, unless expressly declared as **`automatic`**. In this example, the variable `count` will have an initial value of 0 the first time the function is called. However, it will not be re-initialized the next time it is called. Instead, the static variable will retain its value from the previous call, resulting in an erroneous count. The static variable `temp` will have a value of 0 the first time the function is called, rather than the value of `data`. This is because in-line initialization takes place prior to time zero, and not when the function is called.

```
function int count_ones (input [31:0] data);
  logic [31:0] count = 0;   // initialized once
  logic [31:0] temp = data; // initialized once

  for (int i=0; i<=32; i++) begin
    if (temp[0]) count++;
    temp >>= 1;
  end
  return(count);
endfunction
```

automatic variables are initialized each call

A variable explicitly declared as **automatic** in a non-automatic task or function will be dynamically created each time the task or function is entered, and only exists until the task or function exits. An in-line initial value will be assigned each time the task or function is called. The following version of the count_ones function will work correctly, because the automatic variables count and temp are initialized each time the function is called.

```
function int count_ones (input [31:0] data);
  automatic logic [31:0] count = 0;
  automatic logic [31:0] temp = data;

  for (int i=0; i<=32; i++) begin
    if (temp[0]) count++;
    temp >>= 1;
  end
  return(count);
endfunction
```

A variable declared in an automatic task or function will be automatic by default. Storage for the variable will be dynamically created each time the task or function is entered, and destroyed each time the task or function exits. An in-line initial value will be assigned each time the task or function is entered and new storage is created.

3.7.2 Synthesis guidelines for automatic variables

The dynamic storage of automatic variables can be used both in verification testbenches and to represent hardware models. To be synthesized in a hardware model, the automatic variables should only be used to represent temporary storage that does not propagate outside of the task, function or procedural block.

NOTE Static variable initialization is not synthesizable. Automatic variable initialization is synthesizable.

Initialization of static variables is not synthesizable, and should be reserved for usage in testbench code and abstract bus functional models.

In-line initialization of automatic variables is synthesizable. The count_ones function example listed earlier in this chapter, in section 3.7, meets these synthesis criteria. The automatic variables `count` and `temp` are only used within the function, and the values of the variables are only used by the current call to the function.

In-line initialization of variables declared with the **`const`** qualifier is also synthesizable. Section 3.10 on page 71 covers **`const`** declarations.

3.7.3 Guidelines for using static and automatic variables

The following guidelines will aid in the decision on when to use static variables and when to use automatic variables.

- In an **`always`** or **`initial`** block, use static variables if there is no in-line initialization, and automatic variables if there is an in-line initialization. Using automatic variables with in-line initialization will give the most intuitive behavior, because the variable will be re-initialized each time the block is re-executed.
- If a task or function is to be re-entrant, it should be automatic. The variables also ought to be automatic, unless there is a specific reason for keeping the value from one call to the next. As a simple example, a variable that keeps a count of the number of times an automatic task or function is called would need to be static.
- If a task or function represents the behavior of a single piece of hardware, and therefore is not re-entrant, then it should be declared as static, and all variables within the task or function should be static.

3.8 Deterministic variable initialization

3.8.1 Initialization determinism

Verilog-1995 variable initialization

In the original Verilog language, which was standardized in 1995, variables could not be initialized at the time of declaration, as can be done in C. Instead, a separate **`initial`** procedural block was required to set the initial value of variables. For example:

```
integer i;   // declare a variable named i
integer j;   // declare a variable named j

initial
  i = 5;     // initialize i to 5
initial
  j = i;     // initialize j to the value of i
```

Verilog-1995 initialization can be nondeterministic

The Verilog standard explicitly states that the order in which a software tool executes multiple **initial** procedural blocks is nondeterministic. Thus, in the preceding example it cannot be determined whether j will be assigned the value of i before i is initialized to 5 or after i is initialized. If, in the preceding example, the intent is that i is assigned a value of 5 first, and then j is assigned the value of i, the only deterministic way to model the initialization is to group both assignments into a single **initial** procedural block with a **begin**...**end** block. Statements within **begin**...**end** blocks execute in sequence, giving the user control the order in which the statements are executed.

```
integer i;   // declare a variable named i
integer j;   // declare a variable named j

initial begin
  i = 5;   // initialize i to 5
  j = i;   // initialize j to the value of i
end
```

Verilog-2001 variable initialization

The Verilog-2001 standard added a convenient short cut for initializing variables, following the C language syntax of specifying a variable's initial value as part of the variable declaration. Using Verilog, the preceding example can be shortened to:

```
integer i = 5;   // declare and initialize i
integer j = i;   // declare and initialize j
```

Verilog initialization is nondeterministic

Verilog defines the semantics for in-line variable initialization to be exactly the same as if the initial value had been assigned in an **initial** procedural block. This means that in-line initialization will occur in a nondeterministic order, in conjunction with the execution of events in other **initial** procedural blocks and **always** procedural blocks that execute at simulation time zero.

This nondeterministic behavior can lead to simulation results that might not be expected when reading the Verilog code, as in the following example:

```
integer i = 5;    // declare and initialize i
integer j;        // declare a variable named j

initial
  j = i;     // initialize j to the value of i
```

In this example, it would seem intuitive to expect that `i` would be initialized first, and so `j` would be initialized to a value of 5. The nondeterministic event ordering specified in the Verilog standard, however, does not guarantee this. It is within the specification of the Verilog standard for `j` to be assigned the value of `i` before `i` has been initialized, which would mean `j` would receive a value of X instead of 5.

SystemVerilog initialization order

SystemVerilog in-line initialization is before time zero

The SystemVerilog standard enhances the semantics for in-line variable initialization. SystemVerilog defines that all in-line initial values will be evaluated prior to the execution of any events at the start of simulation time zero. This guarantees that when **`initial`** or **`always`** procedural blocks read variables with in-line initialization, the initialized value will be read. This deterministic behavior removes the ambiguity that can arise in the Verilog standard.

NOTE SystemVerilog in-line variable initialization does not cause a simulation event.

Verilog in-line initialization may cause an event

There is an important difference between Verilog semantics and SystemVerilog semantics for in-line variable initialization. Under Verilog semantic rules, in-line variable initialization will be executed during simulation time zero. This means a simulation event will occur if the initial value assigned to the variable is different than its current value. Note, however, that the current value of the variable cannot be known with certainty, because the in-line initialization occurs in a nondeterministic order with other initial assignments—in-line or procedural—that are executed at time zero. Thus, with Verilog semantics, in-line variable initialization may or may not cause in-line initialization simulation events to propagate at simulation time zero.

SystemVerilog initialization does not cause an event

SystemVerilog semantics change the behavior of in-line variable initialization. With SystemVerilog, in-line variable initialization occurs prior to simulation time zero. Therefore, the initialization will never cause a simulation event within simulation.

SystemVerilog initialization is backward compatible

The simulation results using the enhanced SystemVerilog semantics are entirely within the allowed, but nondeterministic, results of the Verilog initialization semantics. Consider the following example:

```
logic resetN = 0; // declare & initialize reset

always @(posedge clock, negedge resetN)
  if (!resetN) count <= 0;  // active low reset
  else count <= count + 1;
```

Verilog in-line initialization is nondeterministic

Using the Verilog nondeterministic semantics for in-line variable initialization, two different simulation results can occur:

- A simulator could activate the **always** procedural block first, prior to initializing the `resetN` variable. The **always** procedural block will then be actively watching for the *next* positive transition event on `clock` or negative transition event on `resetN`. Then, still at simulation time zero, when `resetN` is initialized to 0, which results in an X to 0 transition, the activated **always** procedural block will sense the event, and reset the counter at simulation time zero.
- Alternatively, under Verilog semantics, a simulator could execute the initialization of `resetN` before the **always** procedural block is activated. Then, still at simulation time zero, when the **always** procedural block is activated, it will become sensitive to the *next* positive transition event on `clock` or negative transition event on `resetN`. Since the initialization of `resetN` has already occurred in the event ordering, the counter will not trigger at time zero, but instead wait until the next positive edge of `clock` or negative edge of `resetN`.

SystemVerilog in-line initialization is deterministic

The in-line initialization rules defined in the Verilog standard permit either of the two event orders described above. SystemVerilog removes this non-determinism. SystemVerilog ensures that in-line initialization will occur first, meaning only the second scenario can occur for the example shown above. This behavior is fully backward compatible with the Verilog standard, but is deterministic instead of nondeterministic.

3.8.2 Initializing sequential logic asynchronous inputs

Verilog's nondeterministic order for variable initialization can result in nondeterministic simulation behavior for asynchronous reset or preset logic in sequential models. This nondeterminism can affect resets or presets that are applied at the beginning of simulation.

Example 3-3: Applying reset at simulation time zero with 2-state types

```
module counter (input  wire         clock, resetN,
                output logic [15:0] count);

  always @(posedge clock, negedge resetN)
    if (!resetN) count <= 0;  // active low reset
    else count <= count + 1;
endmodule

module test;
  wire [15:0] count;
  bit         clock;
  bit         resetN = 1;  // initialize reset to inactive value

  counter dut (clock, resetN, count);

  always #10 clock = ~clock;

  initial begin
    resetN = 0;      // assert active-low reset at time 0
    #2 resetN = 1;  // de-assert reset before posedge of clock
    $display("\n count=%0d (expect 0)\n", count);
    #1 $finish;
  end
endmodule
```

In the example above, the counter has an asynchronous reset input. The reset is active low, meaning the counter should reset the moment `resetN` transitions to 0. In order to reset the counter at simulation time zero, the `resetN` input must transition to logic 0. If `resetN` is declared as a 2-state type such as **`bit`**, as in the testbench example above, its initial value by default is a logic 0. The first test in the testbench is to assert reset by setting `resetN` to 0. However, since `resetN` is a 2-state data type, its default initial value is 0. The first test will not cause a simulation event on `resetN`, and therefore the counter model sensitivity list will not sense a change on `resetN` and trigger the procedural block to reset the counter.

To ensure that a change on `resetN` occurs when `resetN` is set to 0, `resetN` is declared with an in-line initialization to logic 1, the inactive state of reset.

```
bit  resetN = 1;   // initialize reset
```

Following Verilog semantic rules, this in-line initialization is executed during simulation time zero, in a nondeterministic order with other assignments executed at time zero. In the preceding example, two event orders are possible:

- The in-line initialization could execute first, setting `resetN` to 1, followed by the procedural assignment setting `resetN` to 0. A transition to 0 will occur, and at the end of time step 0, `resetN` will be 0.
- The procedural assignment could execute first, setting `resetN` to 0 (a 2-state type is already a 0), followed by the in-line initialization setting `resetN` to 1. No transition to 0 will occur, and at the end of time step 0, `resetN` will be 1.

SystemVerilog removes this non-determinism. With SystemVerilog, in-line initialization will take place before simulation time zero. In the example shown above, `resetN` will always be initialized to 1 first, and then the procedural assignment will execute, setting `resetN` to 0. A transition from 1 to 0 will occur every time, in every software tool. At the end of time step 0, `resetN` will be 0.

TIP

Testbenches should initialize variables to their inactive state.

ensuring events at time zero

The deterministic behavior of SystemVerilog in-line variable initialization makes it possible to guarantee the generation of events at simulation time zero. If the variable is initialized using in-line initialization to its inactive state, and then set to its active state using an **`initial`** or **`always`** procedural block, SystemVerilog semantics ensure that the in-line initialization will occur first, followed by the procedural initial assignment.

In the preceding example, the declaration and initialization of `resetN` would likely be part of a testbench, and the **`always`** procedural block representing a counter would be part of an RTL model. Whether in the same module or in separate modules, SystemVerilog's deterministic behavior for in-line variable initialization ensures that a simulation event will occur at time zero, if a variable is initialized to its inactive state using in-line initialization, and then changed to its active level at time zero using a procedural assignment. Verilog's nondeterministic ordering of in-line initialization versus procedural initialization does not guarantee that the desired events will occur at simulation time zero.

3.9 Type casting

Verilog is loosely typed

Verilog is a loosely typed language that allows a value of one type to be assigned to a variable or net of a different type. When the assignment is made, the value is converted to the new type, following rules defined as part of the Verilog standard.

casting is different than loosely typed

SystemVerilog adds the ability to cast a value to a different type. Type casting is different than converting a value during an assignment. With type casting, a value can be converted to a new type within an expression, without any assignment being made.

Verilog does not have type casting

The Verilog 1995 standard did not provide a way to cast a value to a different type. Verilog-2001 added a limited cast capability that can convert signed values to unsigned, and unsigned values to signed. This conversion is done using the system functions **`$signed`** and **`$unsigned`**.

3.9.1 Static (compile time) casting

SystemVerilog adds a cast operator

SystemVerilog adds a cast operator to the Verilog language. This operator can be used to cast a value from one type to another, similar to the C language. SystemVerilog's cast operator goes beyond C, however, in that a vector can be cast to a different size, and signed values can be cast to unsigned or vice versa.

To be compatible with the existing Verilog language, the syntax of SystemVerilog's cast operator is different than C's.

type casting

`<type>'(<expression>)` — casts a value to any type, including user-defined types. For example:

```
7+ int'(2.0 * 3.0);  // cast result of
                     // (2.0 * 3.0) to int,
                     //   then add to 7
```

size casting

<size>’(<expression>) — casts a value to any vector size. For example:

```
logic [15:0] a, b, y;

y = a + b**16'(2);  // cast literal value 2
                    //   to be 16 bits wide
```

sign casting

<sign>’(<expression>) — casts a value to signed or unsigned. For example:

```
shortint a, b;
int      y;

y = y - signed'({a,b}); // cast concatenation
                        //   result to a signed
                        //   value
```

Static casting and error checking

static casting does not have run-time checking

The static cast operation is a compile-time cast. The expression to be cast will always be converted during run time, without any checking that the expression to be cast falls within the legal range of the type to which the value is cast. In the following example, a static cast is used to increment the value of an enumerated variable by 1. The static cast operator does not check that the result of `state + 1` is a legal value for the `next_state` enumerated type. Assigning an out of range value to `next_state` using a static cast will not result in a compile-time or run-time error. Therefore, care must be taken not to cause an illegal value to be assigned to the `next_state` variable.

```
typedef enum {S1, S2, S3} states_t;
states_t state, next_state;

always_comb begin
  if (state != S3)
    next_state = states_t'(state + 1);
  else
    next_state = S1;
end
```

3.9.2 Dynamic casting

compile-time versus dynamic casting

The static cast operation described above is a compile-time cast. The cast will always be performed, without checking the validity of the result. When stronger checking is desired, SystemVerilog provides a new system function, **$cast**, that performs dynamic, run-time checking on the value to be cast.

$cast system function

The **$cast** system function takes two arguments, a destination variable and a source variable. The syntax is:

```
$cast( dest_var, source_exp );
```

For example:

```
int radius, area;

always @(posedge clock)
  $cast(area, 3.154 * radius ** 2);
    // result of cast operation is cast to
    // the type of area
```

invalid casts

$cast attempts to assign the source expression to the destination variable. If the assignment is invalid, a run-time error is reported, and the destination variable is left unchanged. Some examples that would result in an invalid cast are:

- Casting a **real** to an **int**, when the value of the real number is too large to be represented as an **int** (as in the example, above).
- Casting a value to an enumerated type, when the value does not exist in the legal set of values in the enumerated type list, as in the example, that follows.

```
typedef enum {S1, S2, S3} states_t;
states_t state, next_state;

always_latch begin
  $cast(next_state, state + 1);
end
```

$cast can be called as a task

$cast can be called as a task as in the example above. When called as a task, a runtime error is reported if the cast fails, and the destination variable is not changed. In the example above, not changing the next_state variable will result in latched functionality.

$cast can return a status flag

$cast can be called as a system function. The function returns a status flag indicating whether or not the cast was successful. If the cast is successful, **$cast** returns 1. If the cast fails, the **$cast** function returns 0, and does not change the destination variable. When called as a function, no runtime error is reported.

```
typedef enum {S1, S2, S3} states_t;
states_t state, next_state;

int status;

always_comb begin
  status = $cast(next_state, state + 1);
  if (status == 0) // if cast did not succeed...
    next_state = S1;
end
```

Note that the **$cast** function cannot be used with operators that directly modify the source expression, such as ++ or +=.

```
  $cast(next_state, ++state);  // ILLEGAL
```

A primary usage for **$cast** is to assign expression results to enumerated type variables, which are strongly typed variables. Additional examples of using **$cast** are presented in section 4.2 on page 79, on enumerated types.

3.9.3 Synthesis guidelines

TIP

Use the compile-time cast operator for synthesis.

The static, compile-time cast operator is synthesizable. The dynamic **$cast** system function might not be supported by synthesis compilers. At the time this book was written, the IEEE 1364.1 Verilog RTL synthesis standards group had not yet defined the synthesis guidelines for SystemVerilog. As a general rule, however, system tasks and system functions are not considered synthesizable constructs. A safe coding style for synthesis is to use the static cast operator for casting values.

3.10 Constants

Verilog constants

Verilog provides three types of constants: **`parameter`**, **`specparam`** and **`localparam`**. In brief:

- **`parameter`** is a constant for which the value can be redefined during elaboration using **`defparam`** or in-line parameter redefinition.
- **`specparam`** is a constant that can be redefined at elaboration time from SDF files.
- **`localparam`** is an elaboration-time constant that cannot be directly redefined, but for which the value can be based on other constants.

it is illegal to assign constants a hierarchical reference

These Verilog constants all receive their final value at elaboration time. Elaboration is essentially the process of a software tool building the hierarchy of the design represented by module instances. Some software tools have separate compile and elaboration phases. Other tools combine compilation and elaboration into a single process. Because the design hierarchy may not yet be fully resolved during elaboration, it is illegal to assign a **`parameter`**, **`specparam`** or **`localparam`** constant a value that is derived from elsewhere in the design hierarchy.

constants are not allowed in automatic tasks and functions

Verilog also restricts the declaration of the **`parameter`**, **`specparam`** and **`localparam`** constants to modules, static tasks, and static functions. It is illegal to declare one of these constants in an automatic task or function, or in a **`begin`**...**`end`** or **`fork`**...**`join`** block.

the C-like const declaration

SystemVerilog adds the ability to declare any variable as a constant, using the **`const`** keyword. The **`const`** form of a constant is not assigned its value until after elaboration is complete. Because a **`const`** constant receives its value after elaboration, a **`const`** constant can:

- Be declared in dynamic contexts such as automatic tasks and functions.
- Be assigned a value of a net or variable instead of a constant expression.
- Be assigned a value of an object that is defined elsewhere in the design hierarchy.

The declaration of a **const** constant must include a type. Any of the Verilog or SystemVerilog variable types can be specified as a **const** constant, including enumerated types and user-defined types.

```
const logic [23:0] C1 = 7; // 24-bit constant

const int C2 = 15;         // 32-bit constant

const real C3 = 3.14;      // real constant

const C4 = 5;           // ERROR, no type
```

const can be used in automatic tasks and functions

A **const** constant is essentially a variable that can only be initialized. Because the **const** form of a constant receives its value at run-time instead of elaboration, a **const** constant can be declared in an automatic task or function, as well as in modules or static tasks and functions. Variables declared in a **begin**...**end** or **fork**...**join** block can also be declared as a **const** constant.

```
task automatic C;
  const int N = 5;  // N is a constant
  ...
endtask
```

3.11 Summary

This chapter introduced and discussed the powerful compilation-unit declaration scope. The proper use of compilation-unit scope declarations can make it easier to model functionality in a more concise manner. A primary usage of compilation-unit scope declarations is to define new types using **typedef**.

SystemVerilog enhances the ability to specify logic values, making it easier to assign values that easily scale to any vector size. Enhancements to the `` `define `` text substitution provide new capabilities to macros within Verilog models and testbenches.

SystemVerilog also adds a number of new 2-state variables to the Verilog language: **bit**, **byte**, **shortint**, **int**, and **longint**. These variable types enable modeling designs at a higher level of abstraction, using 2-state values. The semantic rules for 2-state values are well defined, so that all software tools will interpret and execute Verilog models using 2-state logic in the same way. A new

shortreal type and a **logic** type are also added. The initialization of variables is enhanced, so as to reduce ambiguities that exist in the Verilog standard. This also helps ensure that all types of software tools will interpret SystemVerilog models in the same way. SystemVerilog also enhances the ability to declare variables that are static or automatic (dynamic) in various levels of design hierarchy. These enhancements include the ability to declare constants in **begin**...**end** blocks and in automatic tasks and functions.

The next chapter continues the topic on SystemVerilog types, covering user-defined types and enumerated types.

Chapter 4

SystemVerilog User-Defined and Enumerated Types

SystemVerilog makes a significant extension to the Verilog language by allowing users to define new net and variable types. User-defined types allow modeling complex designs at a more abstract level that is still accurate and synthesizable. Using SystemVerilog's user-defined types, more design functionality can be modeled in fewer lines of code, with the added advantage of making the code more self-documenting and easier to read.

The enhancements presented in this chapter include:

- Using **`typedef`** to create user-defined types
- Using **`enum`** to create enumerated types
- Working with enumerated values

4.1 User-defined types

The Verilog language does not provide a mechanism for the user to extend the language net and variable types. While the existing Verilog types are useful for RTL and gate-level modeling, they do not provide C-like variable types that could be used at higher levels of abstraction. SystemVerilog adds a number of new types for modeling at the system and architectural level. In addition, SystemVerilog adds the ability for the user to define new net and variable types.

typedef defines a user-defined type

SystemVerilog user-defined types are created using the **typedef** keyword, as in C. User-defined types allow new type definitions to be created from existing types. Once a new type has been defined, variables of the new type can be declared. For example:

```
typedef int unsigned uint;
...
uint a, b;  // two variables of type uint
```

4.1.1 Local typedef definitions

using typedef locally

User-defined types can be defined locally, in a package, or externally, in the compilation-unit scope. When a user-defined type will only be used within a specific part of the design, the **typedef** definition can be made within the module or interface representing that portion of the design. Interfaces are presented in Chapter 10. In the code snippet that follows, a user-defined type called `nibble` is declared, which is used for variable declarations within a module called `alu`. Since the `nibble` type is defined locally, only the `alu` module can see the definition. Other modules or interfaces that make up the overall design are not affected by the local definition, and can use the same `nibble` identifier for other purposes without being affected by the local **typedef** definition in module `alu`.

```
module alu (...);

  typedef logic [3:0] nibble;

  nibble opA, opB;  // variables of the
                    // nibble type

  nibble [7:0] data; // a 32-bit vector made
                     // from 8 nibble types
  ...
endmodule
```

4.1.2 Shared typedef definitions

typedef definitions in packages

When a user-defined type is to be used in many different models, the **typedef** definition can be declared in a package. These definitions can then be referenced directly, or imported into each module, interface or program block that uses the user-defined types. The use of packages is discussed in Chapter 2, section 2.1 on page 8.

typedef definitions in $unit

A **typedef** definition can also be declared externally, in the compilation-unit scope. External declarations are made by placing the **typedef** statement outside of any module, interface or program block, as was discussed in Chapter 2, section 2.2 on page 14.

Example 4-1 illustrates the use of a package **typedef** definition to create a user-defined type called `dtype_t`, that will be used throughout the design. The **typedef** definition is within an **`ifdef** conditional compilation directive, that defines `dtype_t` to be either the 2-state **bit** type or the 4-state **logic** type. Using conditional compilation, all design blocks that use the `dtype_t` user-defined type can be quickly modified to model either 2-state or 4-state logic.

Example 4-1: Directly referencing typedef definitions from a package

```
package chip_types;
  `ifdef TWO_STATE
    typedef bit dtype_t;
  `else
    typedef logic dtype_t;
  `endif
endpackage

module counter
(output chip_types::dtype_t [15:0] count,
 input  chip_types::dtype_t clock, resetN);

  always @(posedge clock, negedge resetN)
    if (!resetN) count <= 0;
    else         count <= count + 1;
endmodule
```

importing package definitions into $unit

It is also possible to import package definitions into the $unit compilation-unit space. This can be useful when many ports of a module are of user-defined types, and it becomes tedious to directly reference the package name for each port declaration. Example 4-2 illustrates importing a package definition into the $unit space, for use as a module port type.

Example 4-2: Importing package typedef definitions into $unit

```
package chip_types;
  `ifdef TWO_STATE
    typedef bit dtype_t;
  `else
    typedef logic dtype_t;
  `endif
endpackage

import chip_types::dtype_t;  // import definition into $unit

module counter
(output dtype_t [15:0] count,
 input  dtype_t clock, resetN);

  always @(posedge clock, negedge resetN)
    if (!resetN) count <= 0;
    else         count <= count + 1;
endmodule
```

If the package contains many typedefs, instead of importing specific package items into the $unit compilation-unit space, the package can be wildcard imported into $unit.

```
import chip_types::*;  // wildcard import
```

4.1.3 Naming convention for user-defined types

A user-defined type can be any legal name in the Verilog language. In large designs, and when using external compilation-unit scope declarations, the source code where a new user-defined type is defined and the source code where a user-defined type is used could be separated by many lines of code, or in separate files. This separation of the **`typedef`** definition and the usage of the new types can make it difficult to read and maintain the code for large designs. When a name is used in the source code, it might not be obvious that the name is actually a user-defined type.

To make source code easier to read and maintain, a common naming convention is to end all user-defined types with the characters "`_t`". This naming convention is used in example 4-1, above, as well as in many subsequent examples in this book.

4.2 Enumerated types

Enumerated types provide a means to declare an abstract variable that can have a specific list of valid values. Each value is identified with a user-defined name, or *label*. In the following example, variable `RGB` can have the values of `red`, `green` and `blue`:

```
enum {red,green,blue} RGB;
```

Verilog style for labeling values

Verilog uses constants in place of enumerated types

The Verilog language does not have enumerated types. To create pseudo labels for data values, it is necessary to define a **parameter** constant to represent each value, and assign a value to that constant. Alternatively, Verilog's **`define** text substitution macro can be used to define a set of macro names with specific values for each name.

The following example shows a simple state machine sequence modeled using Verilog **parameter** constants and **`define** macro names: The parameters are used to define a set of states for the state machine, and the macro names are used to define a set of instruction words that are decoded by the state machine.

Example 4-3: State machine modeled with Verilog 'define and parameter constants

```
`define FETCH 3'h0
`define WRITE 3'h1
`define ADD   3'h2
`define SUB   3'h3
`define MULT  3'h4
`define DIV   3'h5
`define SHIFT 3'h6
`define NOP   3'h7

module controller (output reg         read, write,
                   input  wire [2:0] instruction,
                   input  wire        clock, resetN);

  parameter WAITE = 0,
            LOAD  = 1,
            STORE = 2;

  reg [1:0] State, NextState;
```

```
  always @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= NextState;

  always @(State) begin
    case (State)
      WAITE:  NextState = LOAD;
      LOAD:   NextState = STORE;
      STORE:  NextState = WAITE;
    endcase
  end

  always @(State, instruction) begin
    read = 0; write = 0;
    if (State == LOAD && instruction == `FETCH)
      read = 1;
    else if (State == STORE && instruction == `WRITE)
      write = 1;
  end
endmodule
```

constants do not limit the legal set of values

The variables that use the constant values—`State` and `NextState` in the preceding example—must be declared as standard Verilog variable types. This means a software tool cannot limit the valid values of those signals to just the values of the constants. There is nothing that would limit `State` or `NextState` in the example above from having a value of 3, or a value with one or more bits set to X or Z. Therefore, the model itself must add some limit checking on the values. At a minimum, a synthesis "full case" pragma would be required to specify to synthesis tools that the state variable only uses the values of the constants that are listed in the case items. The use of synthesis pragmas, however, would not affect simulation, which could result in mismatches between simulation behavior and the structural design created by synthesis.

SystemVerilog style for labeling values

SystemVerilog adds enumerated type declarations to the Verilog language, using the **`enum`** keyword, as in C. In its basic form, the declaration of an enumerated type is similar to C.

```
enum {WAITE, LOAD, STORE} State, NextState;
```

enumerated values are identified with labels

Enumerated types can make a model or test program more readable by providing a way to incorporate meaningful labels for the values a variable can have. This can make the code more self-documenting and easier to debug. Enumerated types can be referenced or displayed using the enumerated labels.

Example 4-4 shows the same simple state sequencer as example 4-3, but modified to use SystemVerilog enumerated types.

Example 4-4: State machine modeled with enumerated types

```
package chip_types;
  typedef enum {FETCH, WRITE, ADD, SUB,
                MULT, DIV, SHIFT, NOP  } instr_t;
endpackage

import chip_types::*;  // import package definitions into $unit

module controller (output logic   read, write,
                   input  instr_t instruction,
                   input  wire    clock, resetN);

  enum {WAITE, LOAD, STORE} State, NextState;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= NextState;

  always_comb begin
    case (State)
      WAITE: NextState = LOAD;
      LOAD:  NextState = STORE;
      STORE: NextState = WAITE;
    endcase
  end

  always_comb begin
    read = 0; write = 0;
    if (State == LOAD && instruction == FETCH)
      read = 1;
    else if (State == STORE && instruction == WRITE)
      write = 1;
  end
endmodule
```

enumerated types limit the legal set of values

In this example, the variables `State` and `NextState` can only have the valid values of `WAITE`, `LOAD`, and `STORE`. All software tools will interpret the legal value limits for these enumerated type variables in the same way, including simulation, synthesis and formal verification.

The SystemVerilog specialized **`always_ff`** and **`always_comb`** procedural blocks used in the preceding example are discussed in more detail in Chapter 6.

Importing enumerated types from packages

NOTE Importing an enumerated type definition name does not automatically import the enumerated value labels.

When an enumerated type definition is imported from a package, only the typed name is imported. The value labels in the enumerated list are not imported and made visible in the name space in which the enumerated type name is imported. The following code snippet will not work.

```
package chip_types;
  typedef enum {WAITE, LOAD, READY} states_t;
endpackage

module chip (...);
  import chip_types::states_t; // imports the
                               // typedef name,
                               // only
states_t  state, next_state;
always_ff @(posedge clk, negedge resetN)
  if (!resetN)
    state <= WAITE;   // ERROR: "WAITE" has not
                      // been imported!
  else
    state <= next_state;
  ...
endmodule
```

In order to make the enumerated type labels visible, either each label must be explicitly imported, or the package must be wildcard imported. A wildcard import will make both the enumerated type name and the enumerated value labels visible in the scope of the import statement. For example:

```
import chip_types::*; // wildcard import
```

4.2.1 Enumerated type label sequences

In addition to specifying a set of unique labels, SystemVerilog provides two shorthand notations to specify a range of labels in an enumerated type list.

Table 4-1: Specifying a sequence of enumerated list labels

`state`	creates a single label called **`state`**
`state[N]`	creates a sequence of labels, beginning with **`state0`**, **`state1`**, ... **`stateN-1`**
`state[N:M]`	creates a sequence of labels, beginning with **`stateN`**, and ending with **`stateM`**. If **`N`** is less than **`M`**, the sequence will increment from **`N`** to **`M`**. If **`N`** is greater than **`M`**, the sequence will decrement from **`N`** to **`M`**.

The following example creates an enumerated list with the labels `RESET`, `S0` through `S4`, and `W6` through `W9`:

```
enum {RESET, S[5], W[6:9]} state;
```

4.2.2 Enumerated type label scope

enumerated labels must be unique

The labels within an enumerated type list must be unique within that scope. The scopes that can contain enumerated type declarations are the compilation unit, modules, interfaces, programs, **`begin`**...**`end`** blocks, **`fork`**...**`join`** blocks, tasks and functions.

The following code fragment will result in an error, because the enumerated label `GO` is used twice in the same scope:

```
module FSM (...);
  enum {GO, STOP} fsm1_state;
  ...
  enum {WAITE, GO, DONE} fsm2_state;  // ERROR
  ...
```

This error in the preceding example can be corrected by placing at least one of the enumerated type declarations in a **begin**...**end** block, which has its own naming scope.

```
module FSM (...);
  ...
  always @(posedge clock)
    begin: fsm1
      enum {STOP, GO} fsm1_state;
      ...
    end

  always @(posedge clock)
    begin: fsm2
      enum {WAITE, GO, DONE} fsm2_state;
      ...
    end
  ...
```

4.2.3 Enumerated type values

enumerated type labels have a default value

By default, the actual value represented by the label in an enumerated type list is an integer of the **int** type. The first label in the enumerated list is represented with a value of 0, the second label with a value of 1, the third with a value of 2, and so on.

users can specify the label's value

SystemVerilog allows the value for each label in the enumerated list to be explicitly declared. This allows the abstract enumerated type to be refined, if needed, to represent more detailed hardware characteristics. For example, a state machine sequence can be explicitly modeled to have one-hot values, one-cold values, Johnson-count, Gray-code, or other type of values.

In the following example, the variable `state` can have the values `ONE`, `FIVE` or `TEN`. Each label in the enumerated list is represented as an integer value that corresponds to the label.

```
enum {ONE  = 1,
      FIVE = 5,
      TEN  = 10 } state;
```

It is not necessary to specify the value of each label in the enumerated list. If unspecified, the value representing each label will be incremented by 1 from the previous label. In the next example, the label `A` is explicitly given a value of 1, `B` is automatically given the incremented value of 2 and `C` the incremented value of 3. `X` is explicitly defined to have a value of 24, and `Y` and `Z` are given the incremented values of 25 and 26, respectively.

```
enum {A=1, B, C, X=24, Y, Z} list1;
```

label values must be unique Each label in the enumerated list must have a unique value. An error will result if two labels have the same value. The following example will generate an error, because C and D would have the same value of 3:

```
enum {A=1, B, C, D=3} list2;  // ERROR
```

4.2.4 Base type of enumerated types

the default base type of an enumerated type is int Enumerated types are variables or nets with a set of labeled values. As such, enumerated types have a Verilog or SystemVerilog *base type*. The default base type for enumerated types is **int**, which is a 32-bit 2-state type.

the base type can be explicitly defined In order to represent hardware at a more detailed level, SystemVerilog allows an explicit base type for the enumerated types to be declared. For example:

```
// enumerated type with a 1-bit wide,
// 2-state base type
enum bit {TRUE, FALSE} Boolean;

// enumerated type with a 2-bit wide,
// 4-state base type
enum logic [1:0] {WAITE, LOAD, READY} state;
```

enum value size If an enumerated label of an explicitly-typed enumerated type is assigned a value, the size must match the size of the base type.

```
enum logic [2:0] {WAITE = 3'b001,
                  LOAD  = 3'b010,
                  READY = 3'b100} state;
```

It is an error to assign a label a value that is a different size than the size declared for the base type of the enumerated type. The following example is incorrect. The **enum** variable defaults to an **int** base type. An error will result from assigning a 3-bit value to the labels.

```
enum {WAITE = 3'b001,          // ERROR!
      LOAD  = 3'b010,
      READY = 3'b100} state;
```

It is also an error to have more labels in the enumerated list than the base type size can represent.

```
enum logic {A=1'b0, B, C} list5;
  // ERROR: too many labels for 1-bit size
```

4-state enumerated types

If the base type of the enumerated values is a 4-state type, it is legal to assign values of X or Z to the enumerated labels.

```
enum logic {ON=1'b1, OFF=1'bz} out;
```

If a value of X or Z is assigned to a label in an enumerated list, the next label must also have an explicit value assigned. It is an error to attempt to have an automatically incremented value following a label that is assigned an X or Z value.

```
enum logic [1:0]
  {WAITE, ERR=2'bxx, LOAD, READY} state;
  // ERROR: cannot determine a value for LOAD
```

4.2.5 Typed and anonymous enumerations

typed enumerated types are defined using typedef

Enumerated types can be declared as a user-defined type. This provides a convenient way to declare several variables or nets with the same enumerated value sets. An enumerated type declared using **typedef** is commonly referred to as a *typed enumerated type*. If typedef is not used, the enumerated type is commonly referred to as an *anonymous enumerated type*.

```
typedef enum {WAITE, LOAD, READY} states_t;

states_t  state, next_state;
```

4.2.6 Strong typing on enumerated type operations

most variable types are loosely typed

Most Verilog and SystemVerilog variable types are loosely typed, meaning that any value of any type can be assigned to a variable. The value will be automatically converted to the type of the variable, following conversion rules specified in the Verilog or SystemVerilog standard.

enumerated types are strongly typed

Enumerated types are the exception to this general nature of Verilog. Enumerated types are semi-strongly typed. An enumerated type can only be assigned:

- A label from its enumerated type list
- Another enumerated type of the same type (that is, declared with the same typedef definition)
- A value cast to the typedef type of the enumerated type

operations use the base type of the label

When an operation is performed on an enumerated type value, the enumerated value is automatically converted to the base type and internal value that represents the label in the enumerated type list. If a base type for the enumerated type is not explicitly declared, the base type and labels will default to **int** types.

In the following example:

```
typedef enum {WAITE, LOAD, READY} states_t;
states_t state, next_state;
int foo;
```

`WAITE` will be represented as an **int** with a value of 0, `LOAD` as an **int** with a value of 1, and `READY` as an **int** value of 2.

The following assignment operation on the enumerated type is legal:

```
state = next_state;  // legal operation
```

The `state` and `next_state` are both enumerated type variables of the same type (`states_t`). A value in one enumerated type variable can be assigned to another enumerated type variable of the same type.

The assignment statement below is also legal. The enumerated type of `state` is represented as a base type of **int**, which is added to the literal integer 1. The result of the operation is an **int** value, which is assigned to a variable of type **int**.

```
foo = state + 1;   // legal operation
```

The converse of the preceding example is illegal. An error will result if a value that is not of the same enumerated type is assigned to an enumerated type variable. For example:

```
state = foo + 1;   // ERROR: illegal assignment
```

In this example, the resulting type of `foo + 1` is an **`int`**, which is not the same type as `state`, which is a `states_t` type.

The next examples are also illegal, and will result in errors:

```
state = state + 1;      // illegal operation

state++;                // illegal operation

next_state += state;    // illegal operation
```

The enumerated type of `state` is represented as a base type of **`int`**, which is added to the literal integer 1. The result of the operation is an **`int`** value. It is an error to directly assign this **`int`** result to a variable of the enumerated type `state`, which is a `states_t` type.

4.2.7 Casting expressions to enumerated types

casting values to an enumerated type

The result of an operation can be cast to a typed enumerated type, and then assigned to an enumerated type variable of the same type. Either the SystemVerilog cast operator or the dynamic **`$cast`** system function can be used (see section 3.9 on page 67 in Chapter 3).

```
typedef enum {WAITE, LOAD, READY} states_t;
states_t state, next_state;

next_state = states_t'(state++);     // legal

$cast(next_state, state + 1);        // legal
```

using the cast operator

As discussed earlier in section 3.9 on page 67, there is an important distinction between using the cast operator and the dynamic **`$cast`** system function. The cast operator will always perform the cast operation and assignment. There is no checking that the value to be assigned is in the legal range of the enumerated type set. Using the preceding enumerated type example for `state` and `next_state`, if `state` had a value of `READY`, which is represented as a value of 2, incrementing it by one would result in an integer value of 3. Assigning this value to `next_state` is out of the range of values within the enumerated type list for `next_state`.

This out-of-range value can result in indeterminate behavior. Different software tools may do different things with the out-of-range value. If an out-of-range value is assigned, the actual value that might end up stored in the enumerated type during pre-synthesis simulation of the RTL model might be different than the functionality of the gate-level netlist generated by synthesis.

To avoid ambiguous behavior, it is important that a model be coded so that an out-of-range value is never assigned to an enumerated type variable. The static cast operator cannot always detect when an out-of-range value will be assigned, because the cast operator does not do run-time error checking.

using the $cast system function

The dynamic **`$cast`** system function verifies that the expression result is a legal value before changing the destination variable. In the preceding example, if the result of incrementing `state` is out-of-range for `next_state`, then the call to `$cast(next_state, state+1)` will not change `next_state`, and a run-time error will be reported.

The two ways to perform a cast allow the modeler to make an intelligent trade-off in modeling styles. The dynamic cast is safe because of its run-time error checking. However, this run-time checking adds some amount of processing overhead to the operation, which can affect software tool performance. Also, the **`$cast`** system function may not be synthesizable. The compile-time cast operator does not perform run-time checking, allowing the cast operation to be optimized for better run-time performance.

Users can choose which casting method to use, based on the nature of the model. If it is known that out-of-range values will not occur, the faster compile-time cast operator can be used. If there is the possibility of out-of-range values, then the safer **`$cast`** system function can be used. Note that the SystemVerilog **`assert`** statement can also be used to catch out-of-range values, but an assertion will not prevent the out-of-range assignment from taking place. Assertions are discussed in the companion book, *SystemVerilog for Verification*[1].

4.2.8 Special system tasks and methods for enumerated types

iterating through the enumerated type list

SystemVerilog provides several built-in functions, referred to as *methods*, to iterate through the values in an enumerated type list. These methods automatically handle the semi-strongly typed nature of enumerated types, making it easy to do things such as increment

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

to the next value in the enumerated type list, jump to the beginning of the list, or jump to the end of the list. Using these methods, it is not necessary to know the labels or values within the enumerated list.

enumerated type methods use a C++ syntax

These special methods for working with enumerated lists are called in a manner similar to C++ class methods. That is, the name of the method is appended to the end of the enumerated type name, with a period as a separator.

<enum_variable_name>**.first** — returns the value of the first member in the enumerated list of the specified variable.

<enum_variable_name>**.last** — returns the value of the last member in the enumerated list.

<enum_variable_name>**.next(**<N>**)** — returns the value of the next member in the enumerated list. Optionally, an integer value can be specified as an argument to **next**. In this case, the Nth next value in the enumerated list is returned, starting from the position of the current value of the enumerated type. When the end of the enumerated list is reached, a wrap to the start of the list occurs. If the current value of the enumerated type is not a member of the enumerated list, the value of the first member in the list is returned.

<enum_variable_name>**.prev(**<N>**)** — returns the value of the previous member in the enumerated list. As with the **next** method, an optional integer value can be specified as an argument to **prev**. In this case, the Nth previous value in the enumerated list is returned, starting from the position of the current value of the enumerated type. When the beginning of the enumerated list is reached, a wrap to the end of the list occurs. If the current value of the enumerated type is not a member of the enumerated list, the value of the last member is returned.

<enum_variable_name>**.num** — returns the number of labels in the enumerated list of the given variable.

<enum_variable_name>**.name** — returns the string representation of the label for the value in the given enumerated type. If the value is not a member of the enumeration, the **name** method returns an empty string.

Example 4-5 illustrates a state machine model that sequences through its states, using some of the enumeration methods listed above. The example is a simple 0 to 15 *confidence counter*, where:

- The `in_sync` output is initially 0; it is set when the counter reaches 8; `in_sync` is cleared again if the counter goes to 0.
- If the `compare` and `synced` input flags are both false, the counter stays at its current count.
- If the `compare` flag and the `synced` flag are both true, the counter increments by 1 (but cannot go beyond 15).
- If the `compare` flag is true but the `synced` flag is false, the counter decrements by 2 (but cannot go below 0).

Example 4-5: Using special methods to iterate through enumerated type lists

```
module confidence_counter(input  logic synced, compare,
                                       resetN, clock,
                          output logic in_sync);

  enum {cnt[0:15]} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= cnt0;
    else         State <= Next;

  always_comb begin
    Next = State;  // default NextState value
    case (State)
      cnt0 :   if (compare &&  synced) Next = State.next;
      cnt1 : begin
               if (compare &&  synced) Next = State.next;
               if (compare && !synced) Next = State.first;
             end
      cnt15:   if (compare && !synced) Next = State.prev(2);
      default begin
               if (compare &&  synced) Next = State.next;
               if (compare && !synced) Next = State.prev(2);
              end
    endcase
  end

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) in_sync <= 0;
    else begin
      if (State == cnt8) in_sync <= 1;
      if (State == cnt0) in_sync <= 0;
    end
```

```
endmodule
```

The preceding example uses SystemVerilog's specialized procedural blocks, **always_ff** and **always_comb**. These procedural blocks are discussed in more detail in Chapter 6.

4.2.9 Printing enumerated types

printing enumerated type values and labels

Enumerated type values can be printed as either the internal value of the label, or as the name of the label. Printing the enumerated type directly will print the internal value of the enumerated type. The name of the label representing the current value is accessed using the enumerated type **name** method. This method returns a string containing the name. This string can then be passed to `$display` for printing.

Example 4-6: Printing enumerated types by value and by name

```
module FSM (input  logic       clock, resetN,
            output logic [3:0] control);

  enum logic [2:0] {WAITE=3'b001,
                    LOAD =3'b010,
                    READY=3'b010} State, Next;

  always @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= Next;

  always_comb begin
    $display("\nCurrent state is %s (%b)", State.name, State);
    case (State)
      WAITE: Next = LOAD;
      LOAD:  Next = READY;
      READY: Next = WAITE;
    endcase
    $display("Next state will be %s (%b)", Next.name, Next);
  end

  assign control = State;

endmodule
```

4.3 Summary

The C-like **`typedef`** definition allows users to define new types built up from the predefined types or other user-defined types in Verilog and SystemVerilog. User-defined types can be used as module ports and passed in/out of tasks and functions.

Enumerated types allow the declaration of variables with a limited set of valid values, and the representation of those values with abstract labels instead of hardware-centric logic values. Enumerated types allow modeling a more abstract level than Verilog, making it possible to model larger designs with fewer lines of code. Hardware implementation details can be added to enumerated type declarations, if desired, such as assigning 1-hot encoding values to an enumerated type list that represents state machine states.

SystemVerilog also adds a **`class`** type, enabling an object-oriented style of modeling. Class objects and object-oriented programming are primarily intended for verification, and are not currently synthesizable. Details and examples of SystemVerilog classes can be found in the companion book, *SystemVerilog for Verification*[1].

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

Chapter 5

SystemVerilog Arrays, Structures and Unions

SystemVerilog adds several enhancements to Verilog for representing large amounts of data. The Verilog array construct is extended both in how data can be represented and for operations on arrays. Structure and union types have been added to Verilog as a means to represent collections of variables.

This section presents:

- Structures
- Unions
- Operations on structures and unions
- Unpacked arrays
- Packed arrays
- Operations on arrays
- Array **`foreach`** loop
- Special system functions for working with arrays
- The $bits "sizeof" system function

5.1 Structures

Design data often has logical groups of signals, such as all the control signals for a bus protocol, or all the signals used within a state controller. The Verilog language does not have a convenient mechanism for collecting common signals into a group. Instead, designers must use ad-hoc grouping methods such as naming conventions where each signal in a group starts or ends with a common set of characters.

structures are defined using a C-like syntax

SystemVerilog adds C-like structures to Verilog. A structure is a convenient way of grouping several pieces of related information together. A structure is declared using the **struct** keyword. Structure members can be any variable type, including user-defined types, and any constant type. An example structure declaration is:

```
struct {
  int          a, b;     // 32-bit variables
  opcode_t opcode;       // user-defined type
  logic [23:0] address;  // 24-bit variable
  bit          error;    // 1-bit 2-state var.
} Instruction_Word;
```

the C "tag" is not allowed

The structure declaration syntax in SystemVerilog is very similar to the C language. The one difference is that C allows for an optional "tag" after the **struct** keyword and before the opening brace. SystemVerilog does not allow a tag.

structures are a collection of variables and/or constants

A structure is a collection of variables and/or constants under a single name. The entire collection can be referenced, using the name of the structure. Each member within the structure also has a name, which is used to select it from the structure. A structure member is referenced the same as in C.

```
<structure_name>.<variable_name>
```

For example, to assign a value to the opcode member of the preceding structure, the reference is:

```
Instruction_Word.address = 32'hF000001E;
```

structures are different than arrays

A structure differs from an array, in that an array is a collection of elements that are all the same type and size, whereas a structure is a collection of variables and/or constants that can be different types and sizes. Another difference is that the elements of an array are referenced by an index into the array, whereas the members of a structure are referenced by a member name.

5.1.1 Structure declarations

Variables or nets can be defined as a structure

structures can be variables or nets

A structure is a collection of variables, which can be accessed separately or as a whole. A structure as a whole can be declared as a variable using the **var** keyword. A structure can also be defined as a net, using any of the Verilog net types, such as **wire** or **tri**. When defined as a net type, all members of the structure must be 4-state types.

```
var struct {                  // structure variable
  logic [31:0] a, b;
  logic [ 7:0] opcode;
  logic [23:0] address;
} Instruction_Word_var;

wire struct {                 // structure net
  logic [31:0] a, b;
  logic [ 7:0] opcode;
  logic [23:0] address;
} Instruction_Word_net;
```

Declaring a structure as a **var** or net type is optional. If not specified, then the structure as a whole is considered to be a variable.

```
struct {                      // structure variable
  logic [31:0] a, b;
  logic [ 7:0] opcode;
  logic [23:0] address;
} Instruction_Word_var;
```

Note that, though a structure as a whole can be declared as a net type, net types cannot be used within structures. Nets can be grouped together under a single name using SystemVerilog interfaces, which are discussed in Chapter 10.

Typed and anonymous structures

structures can be user-defined types

User-defined types can be created from structures, using the **typedef** keyword, as discussed in section 4.1 on page 75. Declaring a structure as a user-defined type does not allocate any storage. Before values can be stored in the members of a structure that is defined as a user-defined type, a variable of that user-defined type must be declared.

```
typedef struct {  // structure definition
  logic [31:0] a, b;
  logic [ 7:0] opcode;
  logic [23:0] address;
} instruction_word_t;

instruction_word_t IW;  // structure allocation
```

When a structure is declared without using **typedef**, it is referred to as an *anonymous structure*.

```
struct {
  logic [31:0] a, b;
  logic [ 7:0] opcode;
  logic [23:0] address;
} instruction;
```

Local and shared structure definitions

structure definitions can be shared using packages or $unit

A typed structure can be defined within a module or interface, allowing its use throughout that design block. If a typed structure definition needs to be used in more than one design block, or as a port of a module or interface, then the structure definition should be placed in a package, and imported into design blocks or the $unit compilation-unit space. Typed structures can also be defined directly in the $unit compilation-unit space. Definitions in packages and in $unit are discussed in section 2.1 on page 8 and section 2.2 on page 14 in Chapter 2.

5.1.2 Assigning values to structures

Initializing structures

structures can be initialized using a list of values

The members of a structure can be initialized at the time the structure is instantiated, using a set of values enclosed between the tokens **'{** and **}**. The number of values between the braces must exactly match the number of members in the structure.

```
typedef struct {
```

```
  logic [31:0] a, b;
  logic [ 7:0] opcode;
  logic [23:0] address;
} instruction_word_t;

instruction_word_t IW = '{100, 3, 8'hFF, 0};
```

A similar syntax is used for defining structure constants or structure parameters.

NOTE The SystemVerilog value list syntax is not the same as C.

SystemVerilog uses the tokens **'{ }** to enclose a value list, whereas C uses **{ }**. Early versions of the SystemVerilog draft standard used simple **{ }** braces to delimit value lists, like C. The final version of the IEEE SystemVerilog changed the delimiter to **'{ }** to distinguish the list of values from Verilog's **{ }** concatenation operator.

Assigning to structure members

three ways to assign to structures

A value can be assigned to any member of a structure by referencing the name of the member.

```
typedef struct {
  logic [31:0] a, b;
  logic [ 7:0] opcode;
  logic [23:0] address;
} instr_t;

instr_t IW;

always @(posedge clock, negedge resetN)
  if (!resetN) begin
    IW.a = 100;  // reference structure member
    IW.b = 5;
    IW.opcode = 8'hFF;
    IW.address = 0;
  end
  else begin
    ...
  end
```

Assigning structure expressions to structures

a structure expression is enclosed within '{ ... }

A complete structure can be assigned a structure expression. A structure expression is formed using a comma-separated list of values enclosed between the tokens **'{** and **}**, just as when initializing a structure. The braces must contain a value for each member of the structure.

```
always @(posedge clock, negedge resetN)
  if (!resetN) IW = '{100, 5, 8'hFF, 0};
  else begin
    ...
  end
```

a structure expression can be listed by order or by member name

The values in the structure expression can be listed in the order in which they are defined in the structure. Alternatively, the structure expression can specify the names of the structure members to which values are being assigned, where the member name and the value are separated by a colon. When member names are specified, the expression list can be in any order.

```
IW = '{address:0, opcode:8'hFF, a:100, b:5};
```

It is illegal to mix listing by name and listing by order in the same structure expression.

```
IW = '{address:0, 8'hFF, 100, 5};  // ERROR
```

Default values in structure expressions

some or all members of a structure can be assigned a default value

A structure expression can specify a value for multiple members of a structure by specifying a *default value*. The default value can be specified for all members of a structure, using the **default** keyword.

```
IW = '{default:0}; // set all members of IW to 0
```

The default value can also be specified just for members of a specific type within the structure, using the keyword for the type. The **default** keyword or type keyword is separated from the value by a colon.

```
typedef struct {
  real         r0, r1;
  int          i0, i1;
```

```
  logic [ 7:0] opcode;
  logic [23:0] address;
} instruction_word_t;

instruction_word_t IW;

always @(posedge clock, negedge resetN)
  if (!resetN)
    IW = '{ real:1.0, default:0 };
    // assign all real members a default of 1.0
    // and all other members a default of 0
  else begin
    ...
  end
```

The default value assigned to structure members must be compatible with the type of the member. Compatible values are ones that can be cast to the member's type.

default value precedence

There is a precedence in how structure members are assigned values. The **`default`** keyword has the lowest precedence, and will be overridden by any type-specific defaults. Type-specific default values will be overridden by any explicitly named member values. The following structure expression will assign `r0` a value of 1.0, `r1` a value of 3.1415, and all other members of the structure a value of 0.

```
typedef struct {
  real         r0, r1;
  int          i0, i1;
  logic [ 7:0] opcode;
  logic [23:0] address;
} instruction_word_t;

instruction_word_t IW;

    IW = '{ real:1.0, default:0, r1:3.1415 };
```

5.1.3 Packed and unpacked structures

unpacked structures can have padding

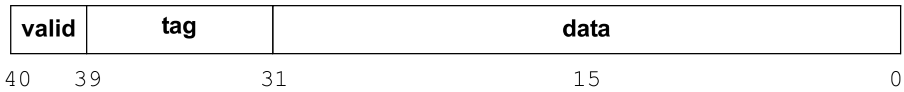
By default, a structure is *unpacked*. This means the members of the structure are treated as independent variables or constants that are grouped together under a common name. SystemVerilog does not specify how software tools should store the members of an unpacked structure. The layout of the storage can vary from one software tool to another.

packed structures are stored without padding

A structure can be explicitly declared as a *packed* structure, using the **packed** keyword. A packed structure stores all members of the structure as contiguous bits, in a specified order. A packed structure is stored as a vector, with the first member of the structure being the left-most field of the vector. The right-most bit of the last member in the structure is the least-significant bit of the vector, and is numbered as bit 0. This is illustrated in Figure 5-1.

```
struct packed {
  logic        valid;
  logic [ 7:0] tag;
  logic [31:0] data;
} data_word;
```

Figure 5-1: Packed structures are stored as a vector



The members of a packed structure can be referenced by either the name of the member or by using a part select of the vector represented by the structure. The following two assignments will both assign to the `tag` member of the `data_word` structure:

```
data_word.tag = 8'hf0;

data_word[39:32] = 8'hf0;  // same bits as tag
```

NOTE Packed structures can only contain integral values.

packed structures must contain packed variables

All members of a packed structure must be integral values. An integral value is a value that can be represented as a vector, such as **byte**, **int** and vectors created using **bit** or **logic** types. A structure cannot be packed if any of the members of the structure cannot be represented as a vector. This means a packed structure cannot contain **real** or **shortreal** variables, unpacked structures, unpacked unions, or unpacked arrays.

Operations on packed structures

packed structures are seen as vectors

Because a packed structure is stored as a vector, operations on the complete structure are treated as vector operations. Therefore, math operations, logical operations, and any other operation that can be performed on vectors can also be performed on packed structures.

```
typedef struct packed {
  logic        valid;
  logic [ 7:0] tag;
  logic [31:0] data;
} data_word_t;

data_word_t packet_in, packet_out;

always @(posedge clock)
  packet_out <= packet_in << 2;
```

Note that when a packed structure is assigned a list of values between the tokens **'{** and **}**, as discussed in section 5.1.2 on page 98, values in the list are assigned to members of the structure. The packed structure is treated the same as an unpacked structure in this circumstance, rather than as a vector. The values within the **'{ }** braces are separate values for each structure member, and not a concatenation of values.

```
packet_in = '{1, '1, 1024};
```

The preceding line assigns 1 to `valid`, FF (hex) to `tag`, and 1024 (decimal) to `data`.

Signed packed structures

a packed structures used as a vector can be signed or unsigned

Packed structures can be declared with the **signed** or **unsigned** keywords. These modifiers affect how the entire structure is perceived when used as a vector in mathematical or relational operations. They do not affect how members of the structure are perceived. Each member of the structure is considered signed or unsigned, based on the type declaration of that member. A part-select of a packed structure is always unsigned, the same as part selects of vectors in Verilog.

```
typedef struct packed signed {
  logic              valid;
  logic       [ 7:0] tag;
```

```
  logic signed [31:0] data;
} data_word_t;

data_word_t  A, B;

always @(posedge clock)
  if ( A < B )          // signed comparison
    ...
```

5.1.4 Passing structures through ports

ports can be declared as a structure type

Structures can be passed through module and interface ports. The structure must first be defined as a user-defined type using **typedef**, which then allows the module or interface port to be declared as the structure type.

```
package definitions;

  typedef enum {ADD, SUB, MULT, DIV} opcode_t;

  typedef struct {
    logic [31:0]  a, b;
    opcode_t      opcode;
    logic [23:0]  address;
    logic         error;
  } instruction_word_t;

endpackage

module alu
 (input definitions::instruction_word_t IW,
  input wire                            clock);
  ...
endmodule
```

An alternative style to explicitly naming the package containing the typedef definition as part of the module port would be to import the package into the $unit compilation-unit declaration space. It is also possible to directly define the user-defined types in the $unit space. Importing packages and using the $unit compilation-unit space are discussed in Chapter 2.

When an unpacked structure is passed through a module port, a structure of the exact same type must be connected on each side of the port. Anonymous structures declared in two different modules, even if they have the exact same name, members and member names, are not the same type of structure. Passing unpacked structures through module ports is discussed in more detail in section 9.6.2 on page 252.

5.1.5 Passing structures as arguments to tasks and functions

structures can be passed to tasks and functions

Structures can be passed as arguments to a task or function. To do so, the structure must be defined as a user-defined type using **typedef**, so that the task or function argument can then be declared as the structure type.

```
module processor (...);
  ...
  typedef enum {ADD, SUB, MULT, DIV} opcode_t;

  typedef struct {  // typedef is local
    logic [31:0]  a, b;
    opcode_t      opcode;
    logic [23:0]  address;
    logic         error;
  } instruction_word_t;

  function alu (input instruction_word_t IW);
    ...
  endfunction
endmodule
```

When a task or function is called that has an unpacked structure as a formal argument, a structure of the exact same type must be passed to the task or function. An anonymous structure, even if it has the exact same members and member names, is not the same type of structure.

5.1.6 Synthesis guidelines

Both unpacked and packed structures are synthesizable. Synthesis supports passing structures through module ports, and in/out of tasks and functions. Assigning values to structures by member name and as a list of values is supported.

5.2 Unions

a union only stores a single value

SystemVerilog adds C-like unions to Verilog. A union is a single storage element that can have multiple representations. Each representation of the storage can be a different type.

The declaration syntax for a union is similar to a structure, and members of a union are referenced in the same way as structures.

```
union {
  int  i;
  int unsigned u;
} data;
...
data.i = -5;
$display("data is %d", data.i);

data.u = -5;
$display("now data is %d", data.u);
```

unions reduce storage and may improve performance

Although the declaration syntax is similar, a union is very different than a structure. A structure can store several values. It is a collection of variables under a single name. A union can only store one value. A typical application of unions is when a value might be represented as several different types, but only as one type at any specific moment in time.

Typed and anonymous unions

A union can be defined as a type using **`typedef`**, in the same way as structures. A union that is defined as a user-defined type is referred to as a *typed union*. If typedef is not used, the union is referred to as an *anonymous union*.

```
typedef union {   // typed union
  int  i;
  int unsigned u;
} data_t;

data_t a, b;  // two variables of type data_t
```

5.2.1 Unpacked unions

An *unpacked* union can contain any variable type, including **`real`** types, unpacked structures and unpacked arrays. Software tools can store values in unpacked unions in an arbitrary manner. There is no requirement that each tool align the storage of the different types used within the union in the same way.

Unpacked unions are not synthesizable. They are an abstract type which are useful for high-level system and transaction level models. As such, it may be useful to store any type in the union including 4-state types, 2-state types, and non-synthesizable types such as real types.

NOTE Reading from an unpacked union member that is different than the last member written may cause indeterminate results.

If a value is stored using one union member, and read back from a different union member, then the value that is read is not defined, and may yield different results in different software tools.

The following example is not synthesizable, but shows how an unpacked union can store very different value types. The example shows a union that can store a value as either an **`int`** type or a **`real`** type. Since these types are stored very differently, it is important that a value always be read back from the union in the same type with which it is written. Therefore, the example contains extra logic to track how values were stored in the union. The union is a member of a structure. A second member of the structure is a flag that can be set to indicate that a real value has been stored in the union. When a value is read from the union, the flag can be checked to determine what type the union is storing.

```
struct {
  bit is_real;
  union {
    int  i;
    real r;
  } value;
} data;
//...
always @(posedge write) begin
  case (operation_type)
    INT_OP: begin
              data.value.i <= 5;
              data.is_real <= 0;
            end
    FP_OP:  begin
              data.value.r <= 3.1415;
              data.is_real <= 1;
            end
  endcase
end
//...
always @(posedge read) begin
```

```
    if (data.is_real)
      real_operand <= data.value.r;
    else
      int_operand <= data.value.i;
  end
```

5.2.2 Tagged unions

A union can be declared as **tagged**.

```
union tagged {
  int  i;
  real r;
} data;
```

tagged unions contain an implicit tag member

A tagged union contains an implicit member that stores a *tag*, which represents the name of the last union member into which a value was stored. When a value is stored in a tagged union using a *tagged expression*, the implicit tag automatically stores information as to which member the value was written.

values are stored in tagged unions using a tagged expression

A value can be written into a tagged union member using a tagged expression. A tagged expression has the keyword tagged followed by the member name, followed by a value to be stored. A tagged expression is assigned to the name of the union. For example:

```
data = tagged i 5;  // store the value 5 in
                    // data.i, and set the
                    // implicit tag
```

Values are read from the tagged union using the union member name.

```
d_out = data.i;  // read value from union
```

tagged unions check that the union is used in a consistent way

Tagged unions require that software tools monitor the usage of the union, and generate an error message if a value is read from a different union member than the member into which a tagged expression value was last written. For example, if the last tagged expression to write to the union specified member `i` (an **`int`** type), then the following line of code will result in a run-time error:

```
d_out = data.r;  // ERROR: member does not match
                 // the union’s implicit tag
```

Once a value has been assigned to a tagged union using a tagged expression, values can be written to the same union member using the member name. If the member name specified does not match the current union tag, a run-time error will result.

```
data.i = 7;  // write to union member; member
             // name must match the current
             // union tag
```

It is still the designer's responsibility to ensure that the design consistently reads values from the union member in which data was last stored. If, however, a design flaw should use the union in an inconsistent way, software tools must inform the designer of the error.

5.2.3 Packed unions

packed union members all have the same size

A union can be declared as **packed** in the same way as a structure. In a packed union, the number of bits of each union member must be the same. This ensures that a packed union will represent its storage with the same number of bits, regardless of member in which a value is stored. Because of this restrictions, packed unions are syntheiszable.

A packed union can only store integral values, which are values made up of 1 or more contiguous bits. If any member of a packed union is a 4-state type, then the union is 4-state. A packed union cannot contain **real** or **shortreal** variables, unpacked structures, unpacked unions, or unpacked arrays.

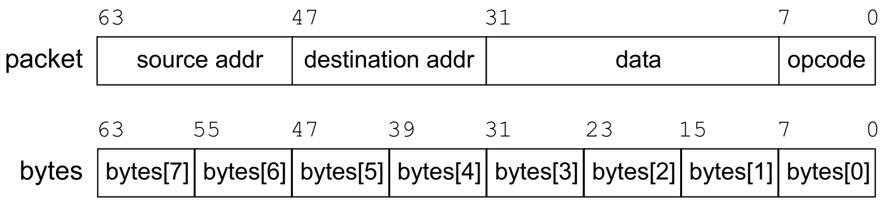
A packed union allows data to be written using one format and read back using a different format. The design model does not need to do any special processing to keep track of how data was stored. This is because the data in a packed union will always be stored using the same number of bits.

The following example defines a packed union in which a value can be represented in two ways: either as a data packet (using a packed structure) or as an array of bytes.

```
typedef struct packed {
  logic [15:0] source_address;
  logic [15:0] destination_address;
  logic [23:0] data;
```

```
    logic [ 7:0] opcode;
  } data_packet_t;

  union packed {
    data_packet_t  packet;    // packed structure
    logic [7:0][7:0] bytes;   // packed array
  } dreg;
```

Figure 5-2: Packed union with two representations of the same storage



Because the union is packed, the information will be stored using the same bit alignment, regardless of which union representation is used. This means a value could be loaded using the array of `bytes` format (perhaps from a serial input stream of bytes), and then the same value can be read using the `data_packet` format.

```
always @(posedge clock, negedge resetN)
  if (!resetN) begin
    dreg.packet <= '0;  // store as packet type
    i <= 0;
  end
  else if (load_data) begin
    dreg.bytes[i] <= byte_in; // store as bytes
    i <= i + 1;
  end

always @(posedge clock)
  if (data_ready)
    case (dreg.packet.opcode) // read as packet
    ...
```

Packed, tagged unions

A union can be declared as both packed and tagged. In this case, the union members can be different bit sizes, but must still be only integral types (1 or more contiguous bits). Packed, tagged unions only permit reading from the same union member that matches the member of the last tagged expression written into the union.

```
union tagged packed {
  logic [15:0] short_word;
  logic [31:0] word;
  logic [63:0] long_word;
} data_word;
```

5.2.4 Synthesis guidelines

NOTE Only packed unions are synthesizable.

packed unions can be synthesized

A union only stores a single value, regardless of how many type representations are in the union. To realize the storage of a union in hardware, all members of the union must be stored as the same vector size using the same bit alignment. Packed unions represent the storage of a union in this way, and are synthesizable. An unpacked union does not guarantee that each type will be stored in the same way, and is therefore not synthesizable.

Packed, tagged unions are intended to be synthesizable, but at the time this book was written, were not widely supported by synthesis compilers.

5.2.5 An example of using structures and unions

Structures provide a mechanism to group related data together under a common name. Each piece of data can be referenced individually by name, or the entire group can be referenced as a whole. Unions allow one piece of storage to be used in multiple ways.

The following example models a simple Arithmetic Logic Unit that can operate on either signed or unsigned values. The ALU opcode, the two operands, and a flag to indicate if the operation data is signed or unsigned, are passed into the ALU as a single instruction word, represented as a structure. The ALU can operate on either signed values or unsigned values, but not both at the same time. Therefore the signed and unsigned values are modeled as a union of two types. This allows one variable to represent both signed and unsigned values.

Chapter 11 presents another example of using structures and unions to represent complex information in a simple and intuitive form.

Example 5-1: Using structures and unions

```
package definitions;

  typedef enum {ADD, SUB, MULT, DIV, SL, SR} opcode_t;

  typedef enum {UNSIGNED, SIGNED} operand_type_t;

  typedef union packed {
    logic        [31:0] u_data;
    logic signed [31:0] s_data;
  } data_t;

  typedef struct packed {
    opcode_t        opc;
    operand_type_t  op_type;
    data_t          op_a;
    data_t          op_b;
  } instr_t;

endpackage

import definitions::*;   // import package into $unit space

module alu
(input  instr_t IW,
 output data_t  alu_out);

  always @(IW) begin
    if (IW.op_type == SIGNED) begin
      case (IW.opc)
        ADD : alu_out.s_data = IW.op_a.s_data + IW.op_b.s_data;
        SUB : alu_out.s_data = IW.op_a.s_data - IW.op_b.s_data;
        MULT: alu_out.s_data = IW.op_a.s_data * IW.op_b.s_data;
        DIV : alu_out.s_data = IW.op_a.s_data / IW.op_b.s_data;
        SL  : alu_out.s_data = IW.op_a.s_data <<< 2;
        SR  : alu_out.s_data = IW.op_a.s_data >>> 2;
      endcase
    end
    else begin
      case (IW.opc)
        ADD : alu_out.u_data = IW.op_a.u_data + IW.op_b.u_data;
        SUB : alu_out.u_data = IW.op_a.u_data - IW.op_b.u_data;
        MULT: alu_out.u_data = IW.op_a.u_data * IW.op_b.u_data;
        DIV : alu_out.u_data = IW.op_a.u_data / IW.op_b.u_data;
        SL  : alu_out.u_data = IW.op_a.u_data << 2;
        SR  : alu_out.u_data = IW.op_a.u_data >> 2;
```

```
      endcase
    end
  end
endmodule
```

5.3 Arrays

5.3.1 Unpacked arrays

Verilog-1995 arrays

The basic syntax of a Verilog array declaration is:

```
<data_type> <vector_size> <array_name> <array_dimensions>
```

For example:

```
reg [15:0] RAM [0:4095];  // memory array
```

Verilog-1995 only permitted one-dimensional arrays. A one-dimensional array is often referred to as a memory, since its primary purpose is to model the storage of hardware memory devices such as RAMs and ROMs. Verilog-1995 also limited array declarations to just the variable types **reg**, **integer** and **time**.

Verilog arrays

Verilog-2001 significantly enhanced Verilog-1995 arrays by allowing any variable or net type except the **event** type to be declared as an array, and by allowing multi-dimensional arrays. Beginning with Verilog-2001, both variable types and net types can be used in arrays.

```
// a 1-dimensional unpacked array of
// 1024 1-bit nets
wire n [0:1023];

// a 1-dimensional unpacked array of
// 256 8-bit variables
reg [7:0] LUT [0:255];

// a 1-dimensional unpacked array of
// 1024 real variables
real r [0:1023];

// a 3-dimensional unpacked array of
```

```
// 32-bit int variables
integer i [7:0][3:0][7:0];
```

Verilog restricts array access to one element at a time

Verilog restricts the access to arrays to just one element of the array at a time, or a bit-select or part-select of a single element. Any reading or writing to multiple elements of an array is an error.

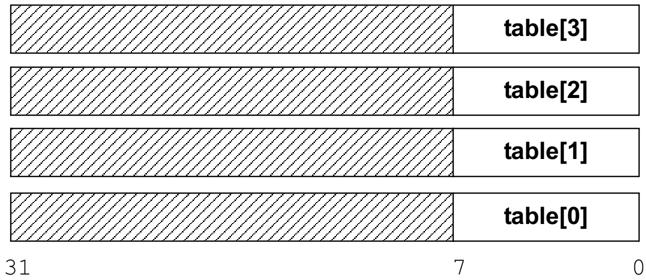
```
integer i [7:0][3:0][7:0];
integer j;

j = i[3][0][1];  // legal: selects 1 element

j = i[3][0];     // illegal: selects 8 elements
```

unpacked arrays store each element independently

SystemVerilog refers to the Verilog style of array declarations as *unpacked arrays*. With unpacked arrays, each element of the array may be stored independently from other elements, but grouped under a common array name. Verilog does not define how software tools should store the elements in the array. For example, given an array of 8-bit wide elements, a simulator or other software tool might store each 8-bit element in 32-bit words. Figure 5-3 illustrates how the following declaration might be stored within memory.

```
wire [7:0] table [3:0];
```

Figure 5-3: Unpacked arrays can store each element independently



SystemVerilog enhancements to unpacked arrays

SystemVerilog allows unpacked arrays of any type

SystemVerilog extends unpacked array dimensions to include the Verilog **`event`** type, and the SystemVerilog types: **`logic`**, **`bit`**, **`byte`**, **`int`**, **`longint`**, **`shortreal`**, and **`real`**. Unpacked arrays of user-defined types defined using **typedef** can also be declared, including types using **struct** and **enum**.

```
bit [63:0] d_array [1:128]; // array of vectors

shortreal cosines [0:89];  // array of floats

typedef enum {Mo, Tu, We, Th, Fr, Sa, Su} Week;
Week Year [1:52];  // array of Week types
```

SystemVerilog can reference all or slices of an array

SystemVerilog also adds to Verilog the ability to reference an entire unpacked array, or a slice of multiple elements within an unpacked array. A slice is one or more contiguously numbered elements within one dimension of an array. These enhancements make it possible to copy the contents of an entire array, or a specific dimension of an array into another array.

NOTE The left-hand and right-hand sides of an unpacked array copy must have identical layouts and types.

copying into multiple elements of an unpacked array

In order to directly copy multiple elements into an unpacked array, the layout and element type of the array or array slice on the left-hand side of the assignment must exactly match the layout and element type of the right-hand side. That is, the element type and size and the number of dimensions copied must be the same.

The following examples are legal. Even though the array dimensions are not numbered the same, the size and layout of each array is the same.

```
int  a1 [7:0][1023:0];  // unpacked array
int  a2 [1:8][1:1024];  // unpacked array

a2 = a1;         // copy an entire array

a2[3] = a1[0];  // copy a slice of an array
```

Array copying is discussed in more detail later in this chapter, in section 5.3.7 on page 124.

Simplified unpacked array declarations

C arrays are specified by size

C language arrays always begin with address 0. Therefore, an array declaration in C only requires that the size of the array be specified. For example:

```
int array [20];  // a C array with addresses
                 // from 0 to 19
```

Verilog arrays are specified by address range

Hardware addressing does not always begin with address 0. Therefore, Verilog requires that array declarations specify a starting address and an ending address of an array dimension.

```
int array [64:83];  // a Verilog array with
                    // addresses from 64 to 83
```

SystemVerilog unpacked arrays can also be specified by size

SystemVerilog adds C-like array declarations to Verilog, allowing *unpacked* arrays to be specified with a dimension size, instead of starting and ending addresses. The array declaration:

```
logic [31:0] data [1024];
```

is equivalent to the declaration:

```
logic [31:0] data [0:1023];
```

As in C, the unpacked array elements are numbered, starting with address 0 and ending with address size-1.

The simplified C-style array declarations cannot be used with vector declarations (packed arrays). The following example is a syntax error.

```
logic [32] d; // illegal vector declaration
```

5.3.2 Packed arrays

The Verilog language allows vectors to be created out of single-bit types, such as **`reg`** and **`wire`**. The vector range comes *before* the signal name, whereas an unpacked array range comes after the signal name.

Verilog vectors are one-dimensional packed arrays

SystemVerilog refers to vector declarations as ***packed arrays***. A Verilog vector is a one-dimensional packed array.

```
wire [3:0] select;  // 4-bit "packed array"

reg [63:0] data;    // 64-bit "packed array"
```

SystemVerilog allows multi-dimensional packed arrays

SystemVerilog adds the ability to declare multiple dimensions in a packed array.

```
logic [3:0][7:0] data;  // 2-D packed array
```

packed arrays have no padding

SystemVerilog defines how the elements of a packed array are stored. The entire array must be stored as contiguous bits, which is the same as a vector. Each dimension of a packed array is a sub field within the vector.

In the packed array declaration above, there is an array of 4 8-bit sub-arrays. Figure 5-4 illustrates how the two-dimensional array above will be stored, regardless of the software compiler, operating system or platform.

Figure 5-4: Packed arrays are stored as contiguous elements

```
logic [3:0][7:0] data;  // 2-D packed array
```

31	23	15	7 0
data[3][7:0]	**data[2][7:0]**	**data[1][7:0]**	**data[0][7:0]**

Packed array types

NOTE Only bit-wise types can be packed.

Packed arrays must be formed using bit-wise types (**`logic`**, **`bit`** or **`reg`**), other packed arrays, packed structures, and packed unions. Packed arrays can also be formed from any of the Verilog net data types (**`wire`**, **`uwire`**, **`wand`**, **`tri`**, **`triand`**, **`trior`**, **`tri0`**, **`tri1`** or **`trireg`**).

```
typedef struct packed {
  logic [ 7:0] crc;
  logic [63:0] data;
} data_word;

data_word [7:0] darray;  // 1-D packed array of
                         // packed structures
```

Referencing packed arrays

A packed array can be referenced as a whole, as bit-selects, or as part-selects. Multidimensional packed arrays can also be referenced in slices. A slice is one or more contiguous dimensions of an array.

```
logic [3:0][7:0] data;  // 2-D packed array

wire [31:0] out = data;          // whole array
wire sign = data[3][7];          // bit-select
wire [3:0] nib = data [0][3:0];  // part-select
byte high_byte;
assign high_byte = data[3];      // 8-bit slice
logic [15:0] word;
assign word = data[1:0];         // 2 slices
```

Operations on packed arrays

any vector operation can be performed on packed arrays

Because packed arrays are stored as vectors, any legal operation that can be performed on a Verilog vector can also be performed on packed arrays. This includes being able to do bit-selects and part-selects from the packed array, concatenation operations, math operations, relational operations, bit-wise operations, and logical operations.

```
logic [3:0][15:0] a, b, result; // packed arrays
...
result = (a << 1) + b;
```

packed arrays use Verilog vector rules

There is no semantic difference between a Verilog vector and a SystemVerilog packed array. Packed arrays use the standard Verilog vector rules for operations and assignment statements. When there is a mismatch in vector sizes, a packed array will be truncated on the left or extended to the left, just as with a Verilog vector.

5.3.3 Using packed and unpacked arrays

The ability to declare multi-dimensional arrays as either packed arrays or unpacked arrays gives a great deal of flexibility on how to represent large amounts of complex data. Some general guidelines on when to use each type of array follow.

use unpacked arrays to model memories, and with abstract types

Use unpacked arrays to model:

- Arrays of **byte**, **int**, **integer**, **real**, unpacked structures, unpacked unions, and other types that are not bit-wise types
- Arrays where typically one element at a time is accessed, such as with RAMs and ROMs

```
module ROM (...);
  byte mem [0:4095]; // unpacked array of bytes
  assign data = select? mem[address]: 'z;
  ...
```

use packed arrays to create vectors with sub-fields

Use packed arrays to model:

- Vectors made up of 1-bit types (the same as in Verilog)
- Vectors where it is useful to access sub-fields of the vector

```
logic [39:0][15:0] packet; // 40 16-bit words

packet = input_stream;  // assign to all words

data = packet[24];      // select 1 16-bit word

tag = packet[3][7:0];   // select part of 1 word
```

5.3.4 Initializing arrays at declaration

Packed array initialization

packed arrays are initialized the same as with vectors

Packed arrays can be initialized at declaration using a simple assignment, like vectors in Verilog. The assignment can be a constant value, a concatenation of constant values or a replication of constant values.

```
logic [3:0][7:0] a = 32'h0;          // vector assignment

logic [3:0][7:0] b = {16'hz,16'h0}; // concatenate operator

logic [3:0][7:0] c = {16{2'b01}};    // replicate operator
```

In the examples above, the **{ }** braces represent the Verilog concatenate operator.

Unpacked array initialization

unpacked arrays are initialized with a list of values

Unpacked arrays can be initialized at declaration, using a list of values enclosed between **'{** and **}** braces for each array dimension. This syntax is similar to assigning a list of values to an array in C, but with the added apostrophe before the opening brace. Using **'{** as the opening delimiter shows that enclosed values are a list of expressions, not the Verilog concatenation of expressions. Note that the C shortcut of omitting the inner braces is not allowed in SystemVerilog. The assignment requires nested sets of braces that exactly match the dimensions of the array.

```
int d [0:1][0:3] = '{ '{7,3,0,5}, '{2,0,1,6} };
// d[0][0] = 7
// d[0][1] = 3
// d[0][2] = 0
// d[0][3] = 5

// d[1][0] = 2
// d[1][1] = 0
// d[1][2] = 1
// d[1][3] = 6
```

SystemVerilog provides a shortcut for declaring a list of values. An inner list for one dimension of an array can be repeated any number of times using a Verilog-like replicate factor. The replicate factor is not followed by an apostrophe.

```
int e [0:1][0:3] = '{ 2{7,3,0,5} };
// e[0][0] = 7
// e[0][1] = 3
// e[0][2] = 0
// e[0][3] = 5

// e[1][0] = 7
// e[1][1] = 3
// e[1][2] = 0
// e[1][3] = 5
```

NOTE The **'{ }** list and **'{n{ }}** replicated list operators are not the same as the Verilog **{ }** concatenate and **{n{ }}** replicate operators.

the {} braces are used two ways

When initializing an *unpacked* array, the **'{ }** braces represent a list of values. This is not the same as a Verilog concatenate operation. As a list of values, each value is assigned to its corresponding element, following the same rules as Verilog assignment statements. This means unsized literal values can be specified in the list, as well as real values.

The Verilog concatenation and replication operators use the **`{ }`** braces, without the leading apostrophe. These operators require that literal values have a size specified, in order to create the resultant single vector. Unsized numbers and real values are not allowed in concatenation and replication operators.

Specifying a default value for unpacked arrays

an array can be initialized to a default value

SystemVerilog provides a mechanism to initialize all the elements of an unpacked array, or a slice of an unpacked array, by specifying a default *value*. The default value is specified within **`'{ }`** braces using the **`default`** keyword, which is separated from the value by a colon. The value assigned to the array must be compatible with the type of the array. A value is compatible if it can be cast to that type.

```
int a1 [0:7][0:1023] = '{default:8'h55};
```

An unpacked array can also be an array of structures or other user-defined types (see section 5.3.11 on page 128). These constructs can contain multiple types. To allow initializing different types within an array to different values, the default value can also be specified using the keyword for the type instead of the **`default`** keyword. A default assignment to the array will automatically descend into structures or unions to find variables of the specified type. Refer to section 5.1.2 on page 98, for an example of specifying default values based on types.

5.3.5 Assigning values to arrays

Assigning values to unpacked arrays

The Verilog language supports two ways to assign values to *unpacked* arrays:

- A single element can be assigned a value.
- A bit-select or part select of a single element can be assigned a value (added as part of the Verilog-2001 standard).

SystemVerilog extends Verilog with two additional ways to assign values to unpacked arrays:

- The entire array can be assigned a list of values.
- A slice of the array can be assigned a list of values.

The list of values is specified between **'{ }** braces, the same as with initializing unpacked arrays, as discussed in section 5.3.4 on page 119.

```
byte a [0:3][0:3];

a[1][0] = 8'h5;  // assign to one element

a = '{'{0,1,2,3},
      '{4,5,6,7},
      '{7,6,5,4},
      '{3,2,1,0}};
// assign a list of values to the full array

a[3] = '{'hF, 'hA, 'hC, 'hE};
// assign list of values to slice of the array
```

The list of assignments to an unpacked array can also specify a default assignment, using the **default** keyword. As procedural assignments, specific portions of an array can be set to different default values.

```
always @(posedge clock, negedge resetN)
  if (!resetN) begin
    a = '{default:0};    // init entire array
    a[0] = '{default:4}; // init slice of array
  end
  else begin
    //...
  end
```

Assigning values to packed arrays

multi-dimensional packed arrays are vectors with sub-fields

Packed arrays are vectors (that might happen to have sub-fields), and can be assigned values, just as with Verilog vectors. A packed array can be assigned a value:

- To one element of the array
- To the entire array (vector)

- To a part select of the array
- To a slice (multiple contiguous sub-fields) of the array

```
logic [1:0][1:0][7:0] a;  // 3-D packed array

a[1][1][0] = 1'b0;   // assign to one bit

a = 32'hF1A3C5E7;    // assign to full array

a[1][0][3:0] = 4'hF; // assign to a part select

a[0] = 16'hFACE;     // assign to a slice

a = {16'bz, 16'b0};  // assign concatenation
```

5.3.6 Copying arrays

This subsection describes the rules for the four possible combinations of assigning arrays to arrays.

Assigning packed arrays to packed arrays

assigning packed array to packed array is allowed

A packed array can be assigned to another packed array. Since packed arrays are treated as vectors, the arrays can be of different sizes and types. Standard Verilog assignment rules for vectors are used to truncate or extend the arrays if there is a mismatch in array sizes.

```
bit   [1:0][15:0] a;  // 32 bit 2-state vector
logic [3:0][ 7:0] b;  // 32 bit 4-state vector
logic [15:0]      c;  // 16 bit 4-state vector
logic [39:0]      d;  // 40 bit 4-state vector

b = a;  // assign 32-bit array to 32-bit array
c = a;  // upper 16 bits will be truncated
d = a;  // upper 8 bits will be zero filled
```

Assigning unpacked arrays to unpacked arrays

assigning unpacked array to unpacked array is allowed

Unpacked arrays can be directly assigned to unpacked arrays only if both arrays have exactly the same number of dimensions and element sizes, and are of the same types. The assignment is done by copying each element of one array to its corresponding element in the destination array. The array elements in the two arrays do not need to be numbered the same. It is the layout of the arrays and the types that must match exactly.

```
logic [31:0] a [2:0][9:0];
logic [0:31] b [1:3][1:10];

a = b;  // assign unpacked array to unpacked
        // array
```

assigning unpacked arrays of different sizes requires casting

If the two unpacked arrays are not identical in layout, the assignment can still be made using a bit-stream cast operation. Bit-stream casting is presented later in this chapter, in section 5.3.7 on page 124.

Assigning unpacked arrays to packed arrays

assigning unpacked arrays to packed arrays requires casting

An unpacked array cannot be directly assigned to a packed array. This is because in the unpacked array, each element is stored independently and therefore cannot be treated as an integral expression (a vector). However unpacked arrays can be assigned to packed arrays using bit-stream casting, as discussed in section 5.3.7 on page 124.

Assigning packed arrays to unpacked arrays

assigning packed arrays to unpacked arrays requires casting

A packed array cannot be directly assigned to an unpacked array. Even if the dimensions of the two arrays are identical, the packed array is treated as a vector, which cannot be directly assigned to an unpacked array, where each array element can be stored independent from other elements. However, the assignment can be made using a bit-stream cast operation.

5.3.7 Copying arrays and structures using bit-stream casting

a bit-stream cast converts arrays to a temporary vector of bits

A bit-stream cast temporarily converts an unpacked array to a stream of bits in vector form. The identity of separate elements within the array is lost—the temporary vector is simply a stream of bits. This temporary vector can then be assigned to another array, which can be either a packed array or an unpacked array. The total number of bits represented by the source and destination arrays must be the same. However, the size of each element in the two arrays can be different.

Bit-stream casting provides a mechanism for:

- assigning an unpacked array to an unpacked array of a different layout
- assigning an unpacked array to a packed array
- assigning a packed array to an unpacked array
- assigning a structure to a packed or unpacked array
- assigning a fixed or dynamically sized array to a dynamically sized array
- assigning a structure to another structure with a different layout.

Bit-stream casting uses the SystemVerilog static cast operator. The casting requires that at least the destination array be represented as a user-defined type, using **typedef**.

```
typedef int data_t [3:0][7:0]; // unpacked type
data_t a;                  // unpacked array
int b [1:0][3:0][3:0];     // unpacked array

a = data_t'(b);    // assign unpacked array to
                   // unpacked array of a
                   // different layout
```

The cast operation is performed by converting the source array (or structure) into a temporary vector representation (a stream of bits) and then assigning groups of bits to each element of the destination array. The assignment is made from left to right, such that the left-most bits of the source bit-stream are assigned to the first element of the destination array, the next left-most bits to the second element, and so forth.

5.3.8 Arrays of arrays

an array can mix packed and unpacked dimensions

It is common to have a combination of unpacked arrays and packed arrays. Indeed, a standard Verilog memory array is actually a mix of array types. The following example declares an unpacked array of 64-bit packed arrays:

```
logic [63:0] mem [0:4095];
```

This next example declares an unpacked array of 32-bit elements, where each element is a packed array, divided into 4 8-bit sub fields:

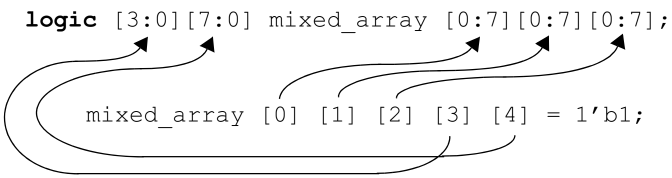
```
wire [3:0][7:0] data [0:1023];
```

Indexing arrays of arrays

unpacked dimensions are indexed before packed dimensions

When indexing arrays of arrays, unpacked dimensions are referenced first, from the left-most dimension to the right-most dimension. Packed dimensions (vector fields) are referenced second, from the left-most dimension to the right-most dimension. Figure 5-5 illustrates the order in which dimensions are selected in a mixed packed and unpacked multi-dimensional array.

Figure 5-5: Selection order for mixed packed/unpacked multi-dimensional array



5.3.9 Using user-defined types with arrays

arrays can contain user-defined types

User-defined types can be used as elements of an array. The following example defines a user type for an unsigned integer, and declares an unpacked array of 128 of the unsigned integers.

```
typedef int unsigned uint;

uint u_array [0:127];  // array of user types
```

User-defined types can also be defined from an array definition. These user types can then be used in other array definitions, creating a compound array.

```
typedef logic [3:0] nibble;  // packed array

nibble [31:0] big_word;      // packed array
```

The preceding example is equivalent to:

```
logic [31:0][3:0] big_word;
```

Another example of a compound array built up from user-defined types is:

```
typedef logic [3:0] nibble;  // packed array

typedef nibble nib_array [0:3];  // unpacked

nib_array compound_array [0:7];  // unpacked
```

This last example is equivalent to:

```
logic [3:0] compound_array [0:7][0:3];
```

5.3.10 Passing arrays through ports and to tasks and functions

In Verilog, a packed array is referred to as a vector, and is limited to a single dimension. Verilog allows packed arrays to be passed through module ports, or to be passed in or out of tasks and functions. Verilog does not allow unpacked arrays to be passed through module ports, tasks or functions.

SystemVerilog allows unpacked arrays as ports and arguments

SystemVerilog extends Verilog by allowing arrays of any type and any number of dimensions to be passed through ports or task/function arguments.

To pass an array through a port, or as an argument to a task or function, the port or task/function formal argument must also be declared as an array. Arrays that are passed through a port follow the same rules and restrictions as arrays that are assigned to other arrays, as discussed in section 5.3.6 on page 123.

```
module CPU (...);
  ...
  logic [7:0] lookup_table [0:255];

  lookup i1 (.LUT(lookup_table));
  ...
endmodule

module lookup (output logic [7:0] LUT [0:255]);
  ...
  initial load(LUT);  //task call

  task load (inout logic [7:0] t [0:255]);
    ...
```

```
  endtask
endmodule
```

5.3.11 Arrays of structures and unions

arrays can contain structures or unions

Packed and unpacked arrays can include structures and unions as elements in the array. In a packed array, the structure or union must also be packed.

```
typedef struct packed {  // packed structure
  logic [31:0] a;
  logic [ 7:0] b;
} packet_t;

packet_t [23:0] packet_array;  // packed array
                              // of 24 structures


typedef struct {  // unpacked structure
  int  a;
  real b;
} data_t;

data_t data_array [23:0];  // unpacked array
                           // of 24 structures
```

5.3.12 Arrays in structures and unions

structures and unions can contain arrays

Structures and unions can include packed or unpacked arrays. A packed structure or union can only include packed arrays.

```
struct packed {          // packed structure
  logic parity;
  logic [3:0][ 7:0] data;  // 2-D packed array
} data_word;


struct {             // unpacked structure
  logic data_ready;
  logic [7:0] data [0:3];   // unpacked array
} packet_t;
```

5.3.13 Synthesis guidelines

Arrays and assignments involving arrays are synthesizable. Specifically:

- Arrays declarations — Both unpacked and packed arrays are synthesizable. The arrays can have any number of dimensions.
- Assigning values to arrays — synthesis supports assigning values to individual elements of an array, bit-selects or part-selects of an array element, array slices, or entire arrays. Assigning lists of literal values to arrays is also synthesizable, including literals using the **`default`** keyword.
- Copying arrays — Synthesis supports packed arrays directly assigned to packed arrays. Synthesis also supports unpacked arrays directly assigned to unpacked arrays of the same layout. Assigning any type of array to any type of array using bit-stream casting is also synthesizable.
- Arrays in structures and unions — The use of arrays within structures and unions is synthesizable. Unions must be packed, which means arrays within the union must be packed).
- Arrays of structures or unions — Arrays of structures and arrays of unions are synthesizable (unions must be packed). A structure or union must be typed (using **`typedef`**) in order to define an array of the structure or union.
- Passing arrays — Arrays passed through module ports, or as arguments to a task or function, is synthesizable.

5.3.14 An example of using arrays

The following example models an instruction register using a packed array of 32 instructions. Each instruction is a compound value, represented as a packed structure. The operands within an instruction can be signed or unsigned, which are represented as a union of two types. The inputs to this instruction register are the separate operands, opcode, and a flag indicating if the operands are signed or unsigned. The model loads these separate pieces of information into the instruction register. The output of the model is the array of 32 instructions.

Example 5-2: Using arrays of structures to model an instruction register

```
package definitions;

  typedef enum {ADD, SUB, MULT, DIV, SL, SR} opcode_t;

  typedef enum {UNSIGNED, SIGNED} operand_type_t;

  typedef union packed {
```

```
    logic [31:0]        u_data;
    logic signed [31:0] s_data;
  } data_t;

  typedef struct packed {
    opcode_t        opc;
    operand_type_t  op_type;
    data_t          op_a;
    data_t          op_b;
  } instr_t;

endpackage

import definitions::*;   // import package into $unit space

module instruction_register (
  output instr_t [0:31] instr_reg, // packed array of structures
  input  data_t         operand_a,
  input  data_t         operand_b,
  input  operand_type_t op_type,
  input  opcode_t       opcode,
  input  logic [4:0]    write_pointer
);

  always @(write_pointer) begin
    instr_reg[write_pointer].op_type = op_type;
    instr_reg[write_pointer].opc     = opcode;

    // use op_type to determine the operand type stored
    // in the input operand union
    if (op_type == SIGNED) begin
      instr_reg[write_pointer].op_a.s_data = operand_a.s_data;
      instr_reg[write_pointer].op_b.s_data = operand_b.s_data;
    end
    else begin
      instr_reg[write_pointer].op_a.u_data = operand_a.u_data;
      instr_reg[write_pointer].op_b.u_data = operand_b.u_data;
    end
  end
endmodule
```

5.4 The foreach array looping construct

SystemVerilog adds a **foreach** loop, which can be used to iterate over the elements of single- and multi-dimensional arrays, without *foreach loops traverse arrays of any number of dimensions*

having to specify the size of each array dimension. The argument to a **`foreach`** loop is the name of an array followed by a comma-separated list of loop variables enclosed in square brackets. Each loop variable corresponds to one of the dimensions of the array.

```
int sum [1:8] [1:3];

foreach ( sum[i,j] )
  sum[i][j] = i + j;   // initialize array
```

The mapping of loop variables to array indexes is determined by the dimension cardinality, as described in section 5.3.8 on page 125. Multiple loop variables create nested loops that iterate over the given indexes. The outer loops correspond to lower cardinality indexes. In the example above, the outermost loop iterates over `i` and the innermost loop iterates over `j`.

foreach loop variables are not declatred

It is not necessary to specify a loop variable for each dimension of an array. A dimension can be skipped by showing a variable position using two commas, without a variable name. Empty loop variables indicate that the loop will not iterate over that dimension of the array. Contiguous empty loop variables at the end of the variable list can be omitted without listing the additional commas.

The following example is a function that generates a check bit for each byte in a 128-bit vector. The vector is represented as a two-dimensional packed array of 16 8-bit elements. A **`foreach`** loop is specified with just one variable, which represents the first dimension (the `[15:0]` dimension) of the array.

```
function [15:0] gen_crc (logic [15:0] [7:0] d);
  foreach (gen_crc[i]) gen_crc[i] = ^d[i];
endfunction
```

Loop variables are automatic, read-only, and local to the loop. The type of each loop variable is implicitly declared to be consistent with the type of array index, which will be **`int`** for the types of arrays that have been presented in this book. (SystemVerilog also has associative arrays, which might use a different type for its indices. Associative arrays are not synthesizable).

5.5 Array querying system functions

special system functions for working with arrays

SystemVerilog adds several special system functions for working with arrays. These system functions allow writing verification routines that work with any size array. They may also be useful in abstract models.

$dimensions(`array_name`**)**

- Returns the number of dimensions in the array (returns 0 if the object is not an array)

$left(`array_name, dimension`**)**

- Returns the most-significant bit (msb) number of the specified dimension. Dimensions begin with the number 1, starting from the left-most unpacked dimension. After the right-most unpacked dimension, the dimension number continues with the left-most packed dimension, and ends with the right-most packed dimension. For the array:

```
logic [1:2][7:0] word [0:3][4:1];
```

`$left(word,1)` will return 0
`$left(word,2)` will return 4
`$left(word,3)` will return 1
`$left(word,4)` will return 7

$right(`array_name, dimension`**)**

- Returns the least-significant bit (lsb) number of the specified dimension. Dimensions are numbered the same as with `$left`.

$low(`array_name, dimension`**)**

- Returns the lowest bit number of the specified dimension, which may be either the msb or the lsb. Dimensions are numbered the same as with `$left`. For the array:

```
logic [7:0] word [1:4];
```

`$low(word,1)` returns 1, and `$low(word,2)` returns 0.

$high(array_name, dimension**)**

- Returns the highest bit number of the specified dimension, which may be either the msb or the lsb. Dimensions are numbered the same as with `$left`.

$size(array_name, dimension**)**

- Returns the total number of elements in the specified dimension (same as `$high - $low + 1`). Dimensions are numbered the same as with `$left`.

$increment(array_name, dimension**)**

- Returns 1 if `$left` is greater than or equal to `$right`, and -1 if `$left` is less than `$right`. Dimensions are numbered the same as with `$left`.

The following code snippet shows how some of these special array system functions can be used to increment through an array, without needing to hard code the size of each array dimension.

```
logic [3:0][7:0] array [0:1023];

int d = $dimensions(array);
if (d > 0) begin   // object is an array
  for (int j = $right(array,1);
           j != ($left(array,1)
                 + $increment(array,1) );
           j += $increment(array,1))
    begin
      ...  // do something
    end
end
```

In this example:
`$right(array,1)` returns `1023`
`$left(array,1)` returns `0`
`$increment(array,1)` returns `-1`

Therefore, the **`for`** loop expands to:

```
for (int j = 1023; j != -1; j += -1)
  begin
    ...
  end
```

The example above could also have been implemented using a **foreach** loop, as follows:

```
foreach ( array[j] )
  begin
    ...
  end
```

The **foreach** loop is discussed earlier in this chapter, in section 5.4 on page 130. When iterating over entire dimensions, and when the total number of loop dimensions is known, the **foreach** loop may be a simpler and more intuitive style than using the array query functions. The advantage of the array query functions is that they provide more information about how an array is declared, including how many dimensions an array contains. This information can be used to iterate of portions of certain dimensions.

Synthesis guidelines

These array query functions are synthesizable, provided that the array has a fixed size, and the dimension number argument is a constant, or is not specified at all. This is an exception to the general rule that synthesis compilers do not support the usage of system tasks or functions. The foreach loop is also synthesizable, provided the array has a fixed size.

5.6 The $bits “sizeof” system function

$bits is similar to C’s sizeof function

SystemVerilog adds a **$bits** system function, which returns how many bits are represented by any expression. The expression can contain any type of value, including packed or unpacked arrays, structures, unions, and literal numbers. The syntax of `$bits` is:

```
$bits(expression)
```

Some examples of using `$bits` are:

```
bit   [63:0] a;
logic [63:0] b;
wire  [3:0][7:0] c [0:15];
struct packed {byte tag; logic [31:0] addr;} d;
```

- `$bits(a)` returns 64

- `$bits(b)` returns 64
- `$bits(c)` returns 512
- `$bits(d)` returns 40
- `$bits(a+b)` returns 128

Synthesis guidelines

The **`$bits`** system function is synthesizable, provided the argument to $bits is not a dynamically sized array. The return value of **`$bits`** can be determined statically at elaboration time, and is therefore treated as a simple literal value for synthesis.

5.7 Dynamic arrays, associative arrays, sparse arrays and strings

SystemVerilog also adds dynamic array types to Verilog:

- Dynamic arrays
- Associative arrays
- Sparse arrays
- Strings (character arrays)

NOTE These special array types are not synthesizable.

Dynamically sized arrays are not synthesizable, and are intended for use in verification routines and for modeling at very high levels of abstraction. The focus of this book is on writing models with SystemVerilog that are synthesizable. Therefore, these array types are not covered in the following subsections. More details on these object-oriented array types can be found in the companion book, *SystemVerilog for Verification*[1].

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

5.8 Summary

SystemVerilog adds the ability to represent complex data sets as single entities. Structures allow variables to be encapsulated into a single object. The structure can be referenced as a whole. Members within the structure can be referenced by name. Structures can be packed, allowing the structure to be manipulated as a single vector. SystemVerilog unions provide a way to model a single piece of storage at an abstract level, where the value stored can be represented as any variable type.

SystemVerilog also extends Verilog arrays in a number of ways. With SystemVerilog, arrays can be assigned values as a whole. All of an array, or slices of one dimension of an array, can be copied to another array. The basic Verilog vector declaration is extended to permit multiple dimensions, in the form of a packed array. A packed array is essentially a vector that can have multiple sub fields. SystemVerilog also provides a number of new array query system functions that are used to determine the characteristics of the array.

Chapter 11 contains a more extensive example of using structures, unions and arrays to represent complex data in a manner that is concise, intuitive and efficient, and yet is fully synthesizable.

Chapter 6

SystemVerilog Procedural Blocks, Tasks and Functions

The Verilog language provides a general purpose procedural block, called **`always`**, that is used to model a variety of hardware types as well as verification routines. Because of the general purpose application of the **`always`** procedural block, the design intent is not readily apparent.

SystemVerilog extends Verilog by adding hardware type-specific procedural blocks that clearly indicate the designer's intent. By reducing the ambiguity of the general purpose **`always`** procedural block, simulation, synthesis, formal checkers, lint checkers, and other EDA software tools can perform their tasks with greater accuracy, and with greater consistency between different tools.

SystemVerilog also provides a number of enhancements to Verilog tasks and functions. Some of these enhancements make the Verilog HDL easier to use, and others substantially increase the power of using tasks and functions for modeling large, complex designs.

The topics covered in this chapter include:

- Combinational logic procedural blocks
- Latched logic procedural blocks
- Sequential logic procedural blocks
- Task and function enhancements

6.1 Verilog general purpose always procedural block

an always procedural block is an infinite loop

The Verilog **always** procedural block is an infinite loop that repeatedly executes the statements within the loop. In order for simulation time to advance, the loop must contain some type of time control or event control. This can be in the form of a fixed delay, represented with the **#** token, a delay until an expression evaluates as true, represented with the **wait** keyword, or a delay until an expression changes value, represented with the @ token. Verilog's general purpose **always** procedural block can contain any number of time controls or event controls, and the controls can be specified anywhere within the procedural block.

The following example illustrates using these time and event controls. The example is syntactically correct, but does not follow proper synthesis modeling guidelines.

infinite loop

```
always
  begin
    wait (resetN == 0) // level-sensitive delay
    @(negedge clock)   // edge-sensitive delay
      #2 t <= d;       // time-based delay
    @(posedge clock)
      #1.5 q <= t;
  end
```

Sensitivity lists

an edge event control can be used as a sensitivity list

An edge sensitive event control at the very beginning of an **always** procedural block is typically referred to as the sensitivity list for that procedural block. Since no statement within the procedural block can execute until the edge-sensitive event control is satisfied, the entire block is sensitive to changes on the signals listed in the event control. In the following example, the execution of statements in the procedural block are sensitive to changes on `a` and `b`.

```
always @(a, b)  // sensitivity list
  begin
    sum  = a + b;
    diff = a - b;
    prod = a * b;
  end
```

General purpose usage of always procedural blocks

always can represent any type of logic

The Verilog **`always`** procedural block is used for general purpose modeling. At the RTL level, the **`always`** procedural block can be used to model *combinatorial logic* (often referred to as *combinational logic*), *latched logic*, and *sequential logic*. At more abstract modeling levels, an **`always`** procedural block can be used to model *algorithmic logic* behavior without clearly representing the implementation details of that behavior, such as an implicit state machine that performs a number of operations on data over multiple clock cycles. The same general purpose **`always`** procedural block is also used in testbenches to model clock oscillators and other verification tasks that need to be repeated throughout the verification process.

Inferring implementation from always procedural blocks

tools must infer design intent from the procedural block's contents

The multi-function role of the general purpose **`always`** procedural block places a substantial burden on software tools such as synthesis compilers and formal verification. It is not enough for these types of tools to execute the statements within the procedural block. Synthesis compilers and formal verification tools must also try to deduce what type of hardware is being represented—combinational, latched or sequential logic. In order to infer the proper type of hardware implementation, synthesis compilers and formal tools must examine the statements and event controls within the procedural block.

The following **`always`** procedural block is syntactically correct, but is not synthesizable. The procedural block will compile and simulate without any compilation or run-time errors, but a synthesis compiler or formal verification tool would probably have errors, because the functionality within does not clearly indicate whether the designer was trying to model combinational, sequential or latched logic.

```
always @(posedge clock) begin
  wait (!resetN)
  if (mode) q1 = a + b;
  else      q1 = a - b;
  q2 <= q1 | (q2 << 2);
  q2++;
end
```

In order to determine how the behavior of this example can be realized in hardware, synthesis compilers and formal tools must examine the behavior of the code logic, and determine exactly when each statement will be executed and when each variable will be updated. A few, but not all, of the factors these tools must consider are:

- What type of hardware can be inferred from the sensitivity list?
- What can be inferred from **`if`**...**`else`** and **`case`** decisions?
- What can be inferred from assignment statements and the operators within those statements?
- Is every variable written to by this procedural block updated in each loop of the **`always`** procedural block? That is, is there any implied storage within the procedural block's functionality that would infer latched behavior?
- Are there assignments in the procedural block that never actually update the variable on the left-side? (In the preceding example the `q2++` statement will never actually increment `q2`, because the line before is a nonblocking assignment that updates its left-hand side, which is `q2`, after the ++ operation).
- Could other procedural blocks elsewhere in the same module affect the variables being written into by this procedural block?

synthesis guidelines for always procedural blocks

In order to reduce the ambiguity of what hardware should be inferred from the general purpose **`always`** procedural block, synthesis compilers place a number of restrictions and guidelines on the usage of **`always`** blocks. The rules for synthesis are covered in the IEEE 1364.1 standard for Verilog Register Transfer Level Synthesis[1]. Some highlights of these restrictions and guidelines are:

combinational logic

To represent combinational logic with a general purpose **`always`** procedural block:

- The **`always`** keyword must be followed by an edge-sensitive event control (the @ token).
- The sensitivity list of the event control cannot contain **`posedge`** or **`negedge`** qualifiers.
- The sensitivity list should include all inputs to the procedural block. Inputs are any signal read by the procedural block, where

1. 1364.1-2002 IEEE Standard for Verilog Register Transfer Level Synthesis. See page xxvii.

that signal receives its value from outside the procedural block.

- The procedural block cannot contain any other event controls.
- All variables written to by the procedural block must be updated for all possible input conditions.
- Any variables written to by the procedural block cannot be written to by any other procedural block.

latched logic

To represent latched logic with a general purpose **`always`** procedural block:

- The **`always`** keyword must be followed by an edge-sensitive event control (the @ token).
- The sensitivity list of the event control cannot contain **`posedge`** or **`negedge`** qualifiers.
- The sensitivity list should include all inputs to the procedural block. Inputs are any signal read by the procedural block, where that signal receives its value from outside the procedural block.
- The procedural block cannot contain any other event controls.
- At least one variable written to by the procedural block must *not* be updated for some input conditions.
- Any variables written to by the procedural block cannot be written to by any other procedural block.

sequential logic

To represent sequential logic with a general purpose **`always`** procedural block:

- The **`always`** keyword must be followed by an edge-sensitive event control (the @ token).
- All signals in the event control sensitivity list must be qualified with **`posedge`** or **`negedge`** qualifiers.
- The procedural block cannot contain any other event controls.
- Any variables written to by the procedural block cannot be written to by any other procedural block.

modeling guidelines cannot be enforced for a general purpose procedural block

Since Verilog **`always`** procedural blocks are general purpose procedural blocks, these synthesis guidelines cannot be enforced other by software tools. Simulation tools, for example, must allow **`always`** procedural blocks to be used in a variety of ways, and not just within the context imposed by synthesis compilers. Because simulation and synthesis are not enforcing the same semantic rules for **always** procedural blocks, mismatches in simulation and synthesis results can occur if the designer does not follow strict, self-imposed modeling guidelines. Formal verification tools may also require that self-imposed modeling guidelines be followed, to prevent mismatches in simulation results and formal verification results.

6.2 SystemVerilog specialized procedural blocks

SystemVerilog adds three specialized procedural blocks to reduce the ambiguity of the Verilog general purpose **always** procedural block when modeling hardware. These are: **always_comb**, **always_latch** and **always_ff**.

specialized procedural blocks are synthesizable

These specialized procedural blocks are infinite loops, the same as an **always** procedural block. However, the procedural blocks add syntactic and semantic rules that enforce a modeling style compatible with the IEEE 1364.1 synthesis standard. These specialized procedural blocks are used to model synthesizable RTL logic.

The specialized **always_comb**, **always_latch** and **always_ff** procedural blocks indicate the design intent. Software tools do not need to infer from context what the designer intended, as must be done with the general purpose **always** procedural block. If the content of a specialized procedural block does not match the rules for that type of logic, software tools can issue warning messages.

specific procedural block types document design intent

By using **always_comb**, **always_latch**, and **always_ff** procedural blocks, the engineer's intent is clearly documented for both software tools and for other engineers who review or maintain the model. Note, however, that SystemVerilog does not require software tools to verify that a procedural block's contents match the type of logic specified with the specific type of always procedural block. Warning messages regarding the procedural block's contents are optional.

6.2.1 Combinational logic procedural blocks

always_comb represents combinational logic

The **always_comb** procedural block is used to indicate the intent to model combinational logic.

```
always_comb
```

```
if (!mode)
  y = a + b;
else
  y = a - b;
```

always_comb infers its sensitivity list

Unlike the general purpose **always** procedural block, it is not necessary to specify a sensitivity list with **always_comb**. A combinational logic sensitivity list can be automatically inferred, because software tools know that the intent is to represent combinational logic. This inferred sensitivity list includes every signal that is read by the procedural block, if the signal receives its value from outside the procedural block. Temporary variables that are only assigned values using blocking assignments, and are only read within the procedural block, are not included in the sensitivity list. SystemVerilog also includes in the sensitivity list any signals read by functions called from the procedural block, except for temporary variables that are only assigned and read within the function. The rules for inferring the sensitivity of bit selects, part selects and array indexing are described in the SystemVerilog LRM.

Because the semantic rules for **always_comb** are standardized, all software tools will infer the same sensitivity list. This eliminates the risk of mismatches that can occur with a general purpose **always** procedural block, should the designer inadvertently specify an incorrect sensitivity list.

shared variables are prohibited

The **always_comb** procedural block also requires that variables on the left-hand side of assignments cannot be written to by any other procedural block. This restriction prevents a form of shared variable usage that does not behave like combinational logic. The restriction matches the guidelines for synthesis, and ensures that all software tools—not just synthesis—are enforcing the same modeling rules.

Non-ambiguous design intent

tools do not need to infer design intent

An important advantage of **always_comb** over the general purpose **always** procedural block is that when **always_comb** is specified, the designer's intent is clearly stated. Software tools no longer need to examine the contents of the procedural block to try to infer what type of logic the engineer intended to model. Instead, with the intent of the procedural block explicitly stated, software tools can examine the contents of the procedural block and issue warning messages if the contents do not represent combinational logic.

In the following example with a general purpose **always** procedural block, a software tool cannot know what type of logic the designer intended to represent, and consequently will infer that latched logic was intended, instead of combinational logic.

```
always @(a, en)
  if (en) y = a;
```

With SystemVerilog, this same example could be written as follows:

```
always_comb
  if (en) y = a;
```

Software tools can then tell from the **always_comb** keyword that the designer's intent was to model combinational logic, and can issue a warning that a latch would be required to realize the procedural block's functionality in hardware.

The correct way to model the example above as combinational logic would be to include an **else** branch so that the output `y` would be updated for all conditions of `en`. If the intent were that `y` did not change when `en` was false, then the correct way to model the logic would be to use an **always_latch** procedural block, as described in section 6.2.2 on page 150 of this chapter.

Checking that the content matches the type of procedural block is optional in the IEEE SystemVerilog standard. Some software tools, such as lint checkers and synthesis compilers will most likely perform these optional checks. Other tools, such as simulators, might not perform these checks.

Automatic evaluation at time zero

always_comb ensures outputs start off consistent with input values

The **always_comb** procedural block also differs from generic **always** procedural blocks in that an **always_comb** procedural block will automatically trigger once at simulation time zero, after all **initial** and **always** procedural blocks have been activated. This automatic evaluation occurs regardless of whether or not there are any changes on the signals in the inferred sensitivity list. This special semantic of **always_comb** ensures that the outputs of the combinational logic are consistent with the values of the inputs to the logic at simulation time zero. This automatic evaluation at time zero can be especially important when modeling with 2-state variables, which, by default, begin simulation with a logic 0. A reset may not cause events on the signals in the combinational logic sensitivity list. If there are no events, a general-purpose **always** procedural block will not trigger and, therefore, the output variables will not be updated.

The following example illustrates this difference between **always_comb** and general-purpose **always** procedural blocks. The model represents a simple Finite State Machine modeled using enumerated types. The three possible states are WAITE, LOAD and STORE. When the state machine is reset, it returns to the WAITE state. The combinational logic of the state machine decodes the current state, and if the current state is WAITE, sets the next state to be LOAD. On each positive edge of clock, the state sequence logic will set the State variable to the value of the NextState variable.

The code listed in example 6-1 models this state machine with Verilog's general purpose **always** procedure.

Example 6-1: A state machine modeled with **always** procedural blocks

```
module controller (output logic   read, write,
                   input  instr_t instruction,
                   input  logic   clock, resetN);

  enum {WAITE, LOAD, STORE} State, NextState;

  always @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= NextState;

  always @(State) begin
    case (State)
      WAITE:  NextState = LOAD;
      LOAD:   NextState = STORE;
      STORE:  NextState = WAITE;
    endcase
  end

  always @(State, instruction) begin
    read = 0; write = 0;
    if (State == LOAD && instruction == FETCH)       read = 1;
    else if (State == STORE && instruction == WRITE) write = 1;
  end
endmodule
```

Only triggers when state changes value

2-state enumerated types can lock up FSM models

There is a simulation subtlety in example 6-1. At simulation time zero, enumerated types default to the default value of the base type of the enumerated type. The base type, unless explicitly declared otherwise, is a 2-state **`int`** type. The initial value when simulation begins for **`int`** is 0, which is also the value of `WAITE` in the enumerated list of values. Therefore, both the `State` variable and the `NextState` variable default to the value of `WAITE`. On a positive edge of `clock`, the state sequence logic will set `State` to `NextState`. Since both variables have the same value, however, `State` does not actually change. Since there is no change on `State`, the **`always`** `@(State)` procedural block does not trigger, and the `NextState` variable does not get updated to a new value. The simulation of this model is locked, because the `State` and the `NextState` variables have the same values. This problem continues to exist even when reset is applied. A reset sets `State` to the value of `WAITE`, which is the same as its current value. Since `State` does not change, the **`always`** `@(State)` procedural block does not trigger, perpetuating the problem that `State` and `NextState` have the same value.

This locked state problem is a simulation anomaly, due to how Verilog sensitivity lists work. The problem would not exist in actual hardware, or even a gate-level model of the hardware. In actual hardware, the outputs of combinational logic will reflect the value of the inputs to that logic. If the inputs to the hardware decoder have the value of `WAITE`, the output, which is `NextState`, will be the value of `LOAD`. In abstract RTL simulation, however, `NextState` does not correctly reflect the inputs of the combinational decoder logic, because at simulation time zero, nothing has triggered the procedural block to cause `NextState` to be updated from its default initial value.

Example 6-2, below, makes one simple change to this example. The **`always`** `@(State)` is replaced with **`always_comb`**. The **`always_comb`** procedural block will infer a sensitivity list for all external variables that are read by the block, which in this example is `State`. Therefore, the **`always_comb`** infers the same sensitivity list as in example 6-1:

Even though the sensitivity lists are the same, there is an important difference between **`always_comb`** and using **`always`** `@(State)`. An **`always_comb`** procedural block automatically executes one time at simulation time zero, after all procedural blocks have been activated. In this example, this means that at simulation time zero, `NextState` will be updated to reflect the value of `State` at time zero. When the first positive edge of `clock` occurs, `State` will transition to the value of `NextState`, which is a different value. This will trigger the **`always_comb`** procedure, which will then update `NextState` to reflect the new value of `State`. Using **`always_comb`**, the simulation lock problem illustrated in example 6-1 will not occur.

Example 6-2: A state machine modeled with **`always_comb`** procedural blocks

```
module controller (output logic   read, write,
                   input  instr_t instruction,
                   input  logic   clock, resetN);

  enum {WAITE, LOAD, STORE} State, NextState;

  always @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= NextState;

  always_comb begin
    case (State)
      WAITE:  NextState = LOAD;
      LOAD:   NextState = STORE;
      STORE:  NextState = WAITE;
    endcase
  end

  always_comb begin
    read = 0; write = 0;
    if (State == LOAD && instruction == FETCH)       read = 1;
    else if (State == STORE && instruction == WRITE) write = 1;
  end
endmodule
```

Infers `@(State)` — the block automatically executes once at time zero, even if not triggered

always_comb versus always @*

The Verilog-2001 standard added the ability to specify a wildcard for the @ event control, using either **`@*`** or **`@(*)`**. The primary intent of the wildcard is to allow modeling combinational logic sensitivity lists without having to specify all the signals within the list.

```
always @*    // combinational logic sensitivity
  if (!mode)
    y = a + b;
  else
    y = a - b;
```

always @ does not have combinational logic semantics*

The inferred sensitivity list of Verilog's `@*` is a convenient shortcut, and can simplify modeling complex procedural blocks with combinational logic. However, the `@*` construct does not require that the contents of the general-purpose **`always`** procedural block adhere to synthesizable combinational logic modeling guidelines.

The specialized **`always_comb`** procedural block not only infers the combinational logic sensitivity list, but also restricts other procedural blocks from writing to the same variables so as to help ensure true combinatorial behavior. In addition, **`always_comb`** executes automatically at time zero, to ensure output values are consistent with input values, whereas the `@*` sensitivity list will only trigger if at least one of the inferred signals in the list changes. This difference was illustrated in examples 6-1 and 6-2, above.

@ can be used incorrectly*

The `@` event control can be used both at the beginning of a procedural block, as a sensitivity list, as well as to delay execution of any statements within a procedural block. Synthesis guidelines do not support combinational event controls within a procedural block. Since `@*` is merely the event control with a wildcard to infer the signals in its event control list, it is syntactically possible to use (or misuse) `@*` within a procedural block, where it cannot be synthesized.

@ sensitivity list may not be complete*

Another important distinction between `@*` and **`always_comb`** is in the sensitivity lists inferred. The Verilog standard defines that `@*` will infer sensitivity to all variables read in the statement or statement group that follows the `@*`. When used at the very beginning of a procedural block, this effectively infers sensitivity to all signals read within that procedural block. If a procedural block calls a function, `@*` will only infer sensitivity to the arguments of the task/function call.

calling functions from combinational logic blocks

A common problem in large designs is that the amount of code in a combinational procedural block can become cumbersome. One solution to prevent the size of a combinational procedural block from getting too large, is to partition the logic into multiple procedural blocks. This partitioning, however, can lead to convoluted spaghetti code, where many signals propagate through several procedural blocks. Another solution is to keep the combinational logic within one procedural block, but break the logic down to smaller sub-blocks using functions. Since functions synthesize to combinational logic, this is an effective method of structuring the code within large combinational procedural blocks.

@ might infer an incomplete sensitivity list*

The Verilog `@*` might not infer a complete sensitivity its when functions are used to structure large blocks of combinational logic. The sensitivity list inferred by always `@*` only looks at the signals read directly by the **`always`** procedural block. It does not infer sensitivity to the signals read from within any functions called by the procedural block. Therefore, each function call must list all signals to be read by each function as inputs to the function, and each function definition must list these signals as formal input arguments. This modeling style restriction is not a synthesis restriction; it is only necessary due to the limitation of `@*`. If, as the design evolves, the signals used by a function should change, then this change must be made in both the function formal argument list and from where the function is called. This additional coding and code management reduces the benefit of using functions to structure large combinational procedural blocks.

always_comb sensitivity list includes signals read by functions

SystemVerilog's **`always_comb`** procedural block eliminates this limitation of `@*`. An **`always_comb`** procedural block is sensitive to both the signals read within the block and the signals read by any function called from the block. This allows a function to be written without formal arguments. If during the design process, the signals that need to be referenced by the function change, no changes need to be made to the function formal argument list or to the code that called the function.

The following example illustrates the difference in sensitivity lists inferred by `@*` and **`always_comb`**. In this example, the procedural block using `@*` will only be sensitive to changes on `data`. The **`always_comb`** procedure will be sensitive to changes on `data`, `sel`, `c`, `d` and `e`.

```
always @* begin        <-- Infers @(data)
  a1 = data << 1;
  b1 = decode();
  ...
end
```

```
always_comb begin        Infers
  a2 = data << 1;        @(data, sel, c, d, e)
  b2 = decode();
  ...
end

function decode;  // function with no inputs
  begin
    case (sel)
      2'b01:   decode = d | e;
      2'b10:   decode = d & e;
      default: decode = c;
    endcase
  end
endfunction
```

6.2.2 Latched logic procedural blocks

always_latch represents latched logic

The **always_latch** procedural block is used to indicate that the intent of the procedural block is to model latched-based logic. **always_latch** infers its sensitivity list, just like **always_comb**.

```
always_latch
  if (enable) q <= d;
```

always_latch has the same semantics as always_comb

An **always_latch** procedural block follows the same semantic rules as with **always_comb**. The rules for what is to be included in the sensitivity list are the same for the two types of procedural blocks. Variables written in an **always_latch** procedural block cannot be written by any other procedural block. The **always_latch** procedural blocks also automatically execute once at time zero, in order to ensure that outputs of the latched logic are consistent with the input values at time zero.

tools can verify always_latch contents represent latched logic

What makes **always_latch** different than **always_comb** is that software tools can determine that the designer's intent is to model latched logic, and perform different checks on the code within the procedural block than the checks that would be performed for combinational logic. For example, with latched logic, the variables representing the outputs of the procedural block do not need to be set for all possible input conditions. In the example above, a software tool could produce an error or warning if **always_comb** had been used, because the **if** statement without a matching **else** branch infers storage that combinational logic does not have. By specifying

always_latch, software tools know that the designer's intent is to have storage in the logic of the design. As with **always_comb**, these additional semantic checks on an **always_latch** procedural block's contents are optional.

An example of using always_latch procedural blocks

The following example illustrates a 5-bit counter that counts from 0 to 31. An input called `ready` controls when the counter starts counting. The `ready` input is only high for a brief time. Therefore, when `ready` goes high, the model latches it as an internal `enable` signal. The latch holds the internal `enable` high until the counter reaches a full count of 31, and then clears the `enable`, preventing the counter from running again until the next time the `ready` input goes high.

Example 6-3: Latched input pulse using an **always_latch** procedural block

```
module register_reader (input clk, ready, resetN,
                        output logic [4:0] read_pointer);

  logic enable;    // internal enable signal for the counter
  logic overflow;  // internal counter overflow flag

  always_latch begin  // latch the ready input
    if (!resetN)
      enable <= 0;
    else if (ready)
      enable <= 1;
    else if (overflow)
      enable <= 0;
  end

  always @(posedge clk, negedge resetN) begin // 5-bit counter
    if (!resetN)
      {overflow,read_pointer} <= 0;
    else if (enable)
      {overflow,read_pointer} <= read_pointer + 1;
  end
endmodule
```

6.2.3 Sequential logic procedural blocks

always_ff represents sequential logic

The **always_ff** specialized procedural block indicates that the designer's intent is to model synthesizable sequential logic behavior.

```
always_ff @(posedge clock, negedge resetN)
  if (!resetN) q <= 0;
  else         q <= d;
```

A sensitivity list must be specified with an **always_ff** procedural block. This allows the engineer to model either synchronous or asynchronous set and/or reset logic, based on the contents of the sensitivity list.

tools can verify that always_ff contents represent sequential logic

By using **always_ff** to model sequential logic, software tools do not need to examine the procedural block's contents to try to infer the type of logic intended. With the intent clearly indicated by the specialized procedural block type, software tools can instead examine the procedural block's contents and warn if the contents cannot be synthesized as sequential logic. As with **always_comb** and **always_latch**, these additional semantic checks on an **always_ff** procedural block's contents are optional.

Sequential logic sensitivity lists

always_ff enforces synthesizable sensitivity lists

The **always_ff** procedural block requires that every signal in the sensitivity list must be qualified with either **posedge** or **negedge**. This is a synthesis requirement for sequential logic sensitivity list. Making this rule a syntactical requirement helps ensure that simulation results will match synthesis results. An **always_ff** procedural block also prohibits using event controls anywhere except at the beginning of the procedural block. Event controls within the procedural block do not represent a sensitivity list for the procedural block, and are not allowed. This is also a synthesis requirement for RTL models of sequential logic.

6.2.4 Synthesis guidelines

The specialized **always_comb**, **always_latch**, and **always_ff** procedural blocks are synthesizable. These specialized procedural blocks are a better modeling choice than Verilog's general purpose **always** procedural block whenever a model is intended to be used with both simulation and synthesis tools. The specialized procedural blocks require simulators and other software tools to check for rules that are required by synthesis compilers. The use of **`always_comb`**, **`always_latch`**, and **`always_ff`** procedural blocks can help eliminate potential modeling errors early in the design process, before models are ready to synthesize.

6.3 Enhancements to tasks and functions

SystemVerilog makes several enhancements to Verilog tasks and functions. These enhancements make it easier to model large designs in an efficient and intuitive manner.

6.3.1 Implicit task and function statement grouping

begin...end groups multiple statements

In Verilog, multiple statements within a task or function must be grouped using **`begin`**...**`end`**. Tasks also allow multiple statements to be grouped using **`fork`**...**`join`**.

SystemVerilog infers begin...end

SystemVerilog simplifies task and function definitions by not requiring the **`begin`**...**`end`** grouping for multiple statements. If the grouping is omitted, multiple statements within a task or function are executed sequentially, as if within a **`begin`**...**`end`** block.

```
function states_t NextState(states_t State);
  NextState = State;   // default next state
  case (State)
    WAITE:  if (start) NextState = LOAD;
    LOAD:   if (done)  NextState = STORE;
    STORE:             NextState = WAITE;
  endcase
endfunction
```

6.3.2 Returning function values

functions create an implied variable of the same name and type

In Verilog, the function name itself is an inferred variable that is the same type as the function. The return value of a function is set by assigning a value to the name of the function. A function exits when the execution flow reaches the end of the function. The last value that was written into the inferred variable of the name of function is the value returned by the function.

```
function [31:0] add_and_inc (input [31:0] a,b);
  begin
    add_and_inc = a + b + 1;
  end
endfunction
```

SystemVerilog adds a **return** statement, which allows functions to return a value using **return**, as in C.

```
function int add_and_inc (input int a, b);
  return a + b + 1;
endfunction
```

return has priority over returning the value in the function name

To maintain backward compatibility with Verilog, the return value of a function can be specified using either the **return** statement or by assigning to the function name. The **return** statement takes precedence. If a **return** statement is executed, that is the value returned. If the end of the function is reached without executing a **return** statement, then the last value assigned to the function name is the return value, as it is in Verilog. Even when using the **return** statement, the name of the function is still an inferred variable, and can be used as temporary storage before executing the **return** statement. For example:

```
function int add_and_inc (input int a, b);
  add_and_inc = a + b;
  return ++add_and_inc;
endfunction
```

6.3.3 Returning before the end of tasks and functions

Verilog must reach the end of a task or function to exit

In Verilog, a task or function exits when the execution flow reaches the end, which is denoted by **endtask** or **endfunction**. In order to exit before the end a task or function is reached using Verilog, conditional statements such as **if**...**else** must be used to force the execution flow to jump to the end of the task or function. A task can also be forced to jump to its end using the **disable** keyword, but this will affect all currently running invocations of a re-entrant task. The following example requires extra coding to prevent executing the function if the input to the function is less than or equal to 1.

```
function automatic int log2 (input int n);
  if (n <=1)
    log2 = 1;
  else begin  // skip this code when n<=1
```

```
    log2 = 0;
    while (n > 1) begin
      n = n/2;
      log2 = log2+1;
    end
  end
endfunction
```

the return statement can be used to exit before the end

The SystemVerilog **return** statement can be used to exit a task or function at any time in the execution flow, without having to reach the end of the task or function. Using **return**, the example above can be simplified as follows:

```
function automatic int log2 (input int n);
  if (n <=1) return 1;  // abort function
  log2 = 0;
  while (n > 1) begin
    n = n/2;
    log2++;
  end
endfunction
```

Using **return** to exit a task or function before the end is reached can simplify the coding within the task or function, and make the execution flow more intuitive and readable.

6.3.4 Void functions

Verilog functions must return a value

In Verilog, functions must have a return value. When the function is called, the calling code must receive the return value.

void functions do not return a value

SystemVerilog adds a **void** type, similar to C. Functions can be explicitly declared as a **void** type, indicating that there is no return value from the function. Void functions are called as statements, like tasks, but have the syntax and semantic restrictions of functions. For example, functions cannot have any type of delay or event control, and cannot use nonblocking assignment statements. Another benefit of void functions is that they overcome the limitation that functions cannot call tasks, making it difficult to add coding structure to a complex function. A function can call other functions, however. Functions can call void functions, and accomplish the same structured coding style of using tasks.

Another SystemVerilog enhancement is that functions can have **output** and **inout** formal arguments. This allows a void function, which has no return value, to still propagate changes to the scope that called the function. Function formal arguments are discussed in more detail later in this chapter, in section 6.3.6 on page 157.

```
typedef struct {
  logic        valid;
  logic [ 7:0] check;
  logic [63:0] data;
} packet_t;

function void fill_packet (
  input  logic [63:0] data_in,
  output packet_t data_out );

  data_out.data = data_in;
  for (int i=0; i<=7; i++)
    data_out.check[i] = ^data_in[(8*i)+:8];
  data_out.valid = 1;
endfunction
```

Synthesis guidelines

TIP

In synthesizable models, use void functions in place of tasks.

An advantage of void functions is that they can be called like a task, but must adhere to the restrictions for function contents. These restrictions, such as the requirement that functions cannot contain any event controls, help ensure proper synthesis results.

6.3.5 Passing task/function arguments by name

Verilog passes argument values by position

When a task or function is called, Verilog only allows values to be passed to the task or function in the same order in which the formal arguments of the task or function are defined. Unintentional coding errors can occur if values are passed to a task or function in the wrong order. In the following example, the order in which the arguments are passed to the divide function is important. In the call to the function, however, it is not apparent whether or not the arguments are in the correct order.

```
always @(posedge clock)
  result <= divide(b, a);

function int divide (input int numerator,
```

```
                                          denominator);
    if (denominator == 0) begin
      $display("Error! divide by zero");
      return 0;
    end
    else
      return numerator / denominator;
  endfunction
```

SystemVerilog can pass argument values by name

SystemVerilog adds the ability to pass argument values to a task or function using the names of formal arguments, rather than the order of the formal arguments. Named argument values can be passed in any order, and will be explicitly passed through the specified formal argument. The syntax for named argument passing is the same as Verilog's syntax for named port connections to a module instance.

With SystemVerilog, the call to the function above can be coded as:

```
    // SystemVerilog style function call
    always @(posedge clock)
      result <= divide(.denominator(b),
                       .numerator(a) );
```

named argument passing can reduce errors

Using named argument passing removes any ambiguity as to which formal argument of each value is to be passed. The code for the task or function call clearly documents the designer's intent, and reduces the risk of inadvertent design errors that could be difficult to detect and debug.

6.3.6 Enhanced function formal arguments

Verilog functions can only have inputs

In Verilog, functions can only have inputs. The only output from a Verilog function is its single return value.

```
    // Verilog style function formal arguments
  function [63:0] add (input [63:0] a, b);
    ...
  endfunction
```

SystemVerilog functions can have inputs and outputs

SystemVerilog allows the formal arguments of functions to be declared as **input**, **output** or **inout**, the same a s with tasks. Allowing the function to have any number of outputs, in addition to the function return value greatly extends what can be modeled using functions.

The following code snippet shows a function that returns the result of an addition operation, plus an output formal argument that indicates if the addition operation resulted in an overflow.

```
  // SystemVerilog style function formal args
function [63:0] add (input  [63:0] a, b,
                     output        overflow);
  {overflow,add} = a + b;
endfunction
```

Restrictions on calling functions with outputs

In order to prevent undesirable—and unsynthesizable—side effects, SystemVerilog restricts from where functions with **`output`** or **`inout`** arguments can be called. A function with output or inout arguments can *not* be called from:

- an event expression.
- an expression within a procedural continuous assignment.
- an expression that is not within a procedural statement.

6.3.7 Functions with no formal arguments

SystemVerilog functions can have no arguments

Verilog allows a task to have any number of formal arguments, including none. However, Verilog requires that functions have at least one input formal argument, even if the function never uses the value of that argument. SystemVerilog allows functions with no formal arguments, the same as with Verilog tasks. An example of using functions without arguments, and the benefits this style can offer, is presented in the latter part of section 6.2.1, under **`always_comb`** versus `@*`, on page 147.

6.3.8 Default formal argument direction and type

In Verilog, the direction of each formal argument to a task or function must be explicitly declared as an **`input`** for functions, or as **`input`**, **`output`**, or **`inout`** for tasks. A comma-separated list of arguments can follow a direction declaration. Each argument in the list will be the last direction declared.

```
function integer compare (input integer a,
                          input integer b);
  ...
```

```
endfunction


task mytask (input a, b, output y1, y2);
...
endtask
```

the default formal argument direction is input

SystemVerilog simplifies the task and function declaration syntax, by making the default direction **input**. Until a formal argument direction is declared, all arguments are assumed to be inputs. Once a direction is declared, subsequent arguments will be that direction, the same as in Verilog.

```
function int compare (int a, b);
  ...
endfunction


// a and b are inputs, y1 and y2 are outputs
task mytask (a, b, output y1, y2);
...
endtask
```

the default formal argument type is logic

In Verilog, each formal argument of a task or function is assumed to be a **reg** type, unless explicitly declared as another variable type. SystemVerilog makes the default type for task or function arguments the **logic** type. Since **logic** is synonymous with **reg**, this is fully compatible with Verilog.

6.3.9 Default formal argument values

each formal argument can have a default value

SystemVerilog allows an optional default value to be defined for each formal argument of a task or function. The default value is specified using a syntax similar to setting the initial value of a variable. In the following example, the formal argument `count` has a default value of 0, and `step` has a default value of 1.

```
function int incrementer(int count=0, step=1);
  incrementer = count + step;
endfunction
```

a call to a task or function can leave some arguments unspecified

When a task or function is called, it is not necessary to pass a value to the arguments that have default argument values. If nothing is passed into the task or function for that argument position, the default value is used for that call of the task or function. In the call to the `incrementer` function below, only one value is passed into the function, which will be passed into the first formal argument of the function. The second formal argument, `step`, will use its default value of 1.

```
always @(posedge clock)
  result = incrementer( data_bus );
```

TIP

Default formal argument values allow task or function calls to only pass values to the arguments unique to that call.

Specifying default argument values allows a task or function to be defined that can be used in multiple ways. In the preceding example, if the function to increment a value is called with just one argument, its default is to increment the value passed in by one. However, the function can also be passed a second value when it is called, where the second value specifies the increment amount.

SystemVerilog also changes the semantics for calling tasks or functions. Verilog requires that a task or function call have the exact same number of argument expressions as the number of task/function formal arguments. SystemVerilog allows the task or function call to have fewer argument expressions than the number of formal arguments, as in the preceding example, so long as the formal arguments that are not passed a value have a default value.

If a task or function call does not pass a value to an argument of the task or function, then the formal definition of the argument must have a default value. An error will result if a formal argument without a default value is not passed in a value.

6.3.10 Arrays, structures and unions as formal arguments

formal arguments can be structures, unions or arrays

SystemVerilog allows unpacked arrays, packed or unpacked structures and packed, unpacked, or tagged unions to be passed in or out of tasks and functions. For structures or unions, the formal argument must be defined as a structure or union type (where **`typedef`** is used to define the type). Packed arrays are treated as a vector when passed to a task or function. If the size of a packed array argument of the call does not match the size of the formal argument, the vector is truncated or expanded, following Verilog vector assignment rules. For unpacked arrays, the task or function call array argument that is passed to the task or function must exactly match the layout and element types of the definition of the array formal argument. To match, the call argument and formal argument must have the same number of array dimensions and dimension sizes, and the same packed size for each element. An example of using an unpacked array formal argument and an unpacked structure formal argument follow:

```
typedef struct {
  logic        valid;
  logic [ 7:0] check;
  logic [63:0] data;
} packet_t;

function void fill_packet (
  input logic [7:0] data_in [0:7], // array arg
  output packet_t data_out );  // structure arg

  for (int i=0; i<=7; i++) begin
    data_out.data[(8*i)+:8] = data_in[i];
    data_out.check[i] = ^data_in[i];
  end
  data_out.valid = 1;
endfunction
```

6.3.11 Passing argument values by reference instead of copy

values are passed to tasks and functions by copy

When a task or function is called, inputs are copied into the task or function. These values then become local values within the task or function. When the task or function returns at the end of its execution, all outputs are copied out to the caller of the task or function.

Verilog has implicit pass by reference task/ function arguments

Verilog can reference signals that were not passed in to the task of function. For functions, this simplifies writing the function when that function is only called from one location. The function does not need to have formal arguments specified, and the call to the function does not need to list the signals to pass to the function. This style is sometimes used to break a complex procedural block into smaller, structured coding blocks. For tasks, external references to signals allows the task to sense when the external signal changes value, and for changes made within the task to immediately be sensed outside of the task, before the task has completed execution.

external name referencing uses hardcoded names

Verilog's ability for a task or function to reference external signals is useful in both test code and RTL models. External references are synthesizable. In RTL code, external signal referencing allows values of signals to be read and/or modified without having to copy values in and out of the task or function. However, external references requires that the external signal name must be hardcoded into the task or function. This limits the ability to code a general purpose task or function that can be called several times in a module, with different signals used for each call. SystemVerilog compounds this limitiation with the addition of the ability to define tasks and function in packages, which can then be imported into any number of design blocks. Hardcoded signal names within the task or function does not work well with this mult-use methodology.

SystemVerilog has explicit pass by reference task/function arguments

SystemVerilog extends automatic tasks and functions by adding the capability to pass values by reference instead of by copy. To pass a value by reference, the formal argument is declared using the keyword **`ref`** instead of the direction keywords **`input`**, **`output`** or **`inout`**. The name of the **`ref`** argument becomes an alias for the hierarchical reference to the actual storage for the value passed to the task or function. Within the task or function, the local argument name is used instead of the external signal name. Pass by reference provides the capabilities of Verilog's external name referencing, without having the limitations of hardcoding the external signal names into the task or function.

a ref formal arguments is an alias to the actual value

Passing by reference allows a variable to be declared in just the calling scope, and not duplicated within a task or function. Instead, the task or function refers to the variable in the scope from which it is called. Referencing a signal that was not passed into a task or function is the same as if a reference to the external signal had been implicitly passed to the task or function.

NOTE Only automatic tasks and functions can have **`ref`** arguments.

In order to have **`ref`** arguments, a task or function must be automatic. The task or function can be explicitly declared as automatic, or it can be inferred as automatic by being declared in a module, interface or program that is defined as automatic.

In the example below, a structure called `data_packet` and an array called `raw_data` are allocated in module `chip`. These objects are then passed as arguments in a call to the `fill_packet` function. Within `fill_packet`, the formal arguments are declared as **`ref`** arguments, instead of inputs and outputs. The formal argument `data_in` becomes an alias within the function for the `raw_data` array in the calling scope, `chip`. The formal argument `data_out` becomes an alias for the `data_packet` structure within `chip`.

```
module chip (...);

  typedef struct {
    logic        valid;
    logic [ 7:0] check;
    logic [63:0] data;
  } packet_t;

  packet_t  data_packet;
  logic [7:0] raw_data [0:7];

  always @(posedge clock)
    if (data_ready)
      fill_packet (.data_in(raw_data),
                   .data_out(data_packet) );

  function automatic void fill_packet (
    ref logic [7:0] data_in [0:7],  // ref arg
    ref packet_t data_out );        // ref arg

    for (int i=0; i<=7; i++) begin
      data_out.data[(8*i)+:8] = data_in[i];
      data_out.check[i] = ^data_in[i];
    end
    data_out.valid = 1;
  endfunction
  ...
endmodule
```

Read-only reference arguments

pass by reference can be read-only

A reference formal argument can be declared to only allow reading of the object that is referenced, by declaring the formal argument as **`const ref`**. This can be used to allow the task or function to reference the information in the calling scope, but prohibit the task or function from modifying the information within the calling scope.

```
function automatic void fill_packet (
  const ref logic [7:0] data_in [0:7],
  ref  packet_t data_out );
  ...
endfunction
```

Task ref arguments are sensitive to changes

pass by reference allows sensitivity to changes

An important characteristic of **ref** arguments is that the logic of a task can be sensitive to when the signal in the calling scope changes value. This sensitivity to changes does not apply to function **ref** arguments. Since functions must execute in zero time, the function cannot contain timing controls that sense changes to arguments. In the following example, the `received` packet and `done` flag are passed by reference. This allows the **wait** statement to observe when the flag becomes true in the module that calls the task. If `done` had been copied in as an input, the **wait** statement would be looking at the local copy of `done`, which would not be updated when the `done` flag changed in the calling module.

```
typedef struct {
  logic        valid;
  logic [ 7:0] check;
  logic [63:0] data;
} packet_t;

packet_t send_packet, receive_packet;

task automatic check_results (
  input packet_t sent,
  ref   packet_t received,
  ref   logic    done );

  static int error_count;

  wait (done)
  if (sent !== received) begin
    error_count++;
    $display("ERROR! received bad packet");
  end
endtask
```

Ref arguments can read current values

In the preceding example, the `sent` packet is an input, which is copied in at the time the task is called. The `received` packet is passed by reference, instead of by copy. When the `done` flag changes, the task will compare the current value of the `received` packet with the copy of the `sent` packet from the time when the task was called. If the `received` packet had been copied in, the comparison would have been made using the value of the `received` packet at the time the task was called, instead of at the time the `done` flag became true.

Ref arguments can propagate changes immediately

When task outputs are passed by copy, the value is not copied back to the calling scope until the task exits. If there are time controls or event controls between when the local copy of the task argument is changed and when the task exits, the calling scope will see the change to the variable when the task exits, and not when the local copy inside the task is assigned.

When a task output is passed by reference, the task is making its assignment directly to the variable in the calling scope. Any event controls in the calling scope that are sensitive to changes on the variable will see the change immediately, instead of waiting until the task completes its execution and output arguments are copied back to the calling scope.

Restrictions on calling functions with ref arguments

A function with **`ref`** formal arguments can modify values outside the scope of the function, and therefore has the same restrictions as functions with output arguments. A function with **`output`**, **`inout`** or **`ref`** arguments can *not* be called from:

- an event expression
- an expression within a continuous assignment
- an expression within a procedural continuous assignment
- an expression that is not within a procedural statement

6.3.12 Named task and function ends

SystemVerilog allows a name to be specified with the **`endtask`** or **`endfunction`** keyword. The syntax is:

```
endtask : <task_name>

endfunction : <function_name>
```

The white space before and after the colon is optional. The name specified must be the same as the name of the corresponding task or function. For example:

```
function int add_and_inc (int a, b);
  return a + b + 1;
endfunction : add_and_inc


task automatic check_results (
  input packet_t sent,
  ref   packet_t received,
  ref   logic    done );
  static int error_count;
  ...
endtask: check_results
```

Specifying a name with the **endtask** or **endfunction** keyword can help make large blocks of code easier to read, thus making the model more maintainable.

6.3.13 Empty tasks and functions

a task or function can be empty

Verilog requires that tasks and functions contain at least one statement (which can be an empty **begin**...**end** statement group). SystemVerilog allows tasks and functions to be completely empty, with no statements or statement groups at all. An empty function will return the current value of the implicit variable that represents the name of the function.

An empty task or function is a place holder for partially completed code. In a top-down design flow, creating an empty task or function can serve as documentation in a model for the place where more detailed functionality will be filled in later in the design flow.

6.4 Summary

This chapter has presented the **always_comb**, **always_latch**, and **always_ff** specialized procedural blocks that SystemVerilog adds to the Verilog standard. These specialized procedural blocks add semantics that increase the accuracy and portability for modeling hardware, particularly at the synthesizable RTL level of modeling. Also important is that these specialized procedural blocks make the designer's intent clear as to what type of logic the procedural block should represent. Software tools can then examine the contents of the procedural block, and issue warnings if the code within the procedural block cannot be properly realized with the intended type of hardware.

SystemVerilog also adds a number of enhancements to Verilog tasks and functions. These enhancements include simplifications of Verilog syntax or semantic rules, as well as new capabilities for how tasks and functions can be used. Both types of changes allow modeling larger and more complex designs more quickly and with less coding.

Chapter 7

SystemVerilog Procedural Statements

SystemVerilog adds several new operators and procedural statements to the Verilog language that allow modeling more concise synthesizable RTL code. Additional enhancements convey the designer's intent, helping to ensure that all software tools interpret the procedural statements in the same way. This chapter covers the operators and procedural statements that are synthesizable, and offers guidelines on how to properly use these new constructs.

This SystemVerilog features presented in this chapter include:

- New operators
- Enhanced **`for`** loop
- New **`do`**...**`while`** bottom testing loop
- New **`foreach`** loop
- New jump statements
- Enhanced block names
- Statement labels
- Unique and priority decisions

7.1 New operators

7.1.1 Increment and decrement operators

++ and -- operators

SystemVerilog adds the **++** increment operator and the **--** decrement operator to the Verilog language. These operators are used in the same way as in C. For example:

```
for (i = 0; i <= 31; i++ ) begin
  ...
end
```

Post-increment and pre-increment

As in C, the increment and decrement operators can be used to either pre-increment/pre-decrement a variable, or to post-increment/post-decrement a variable. Table 7-1 shows the four ways in which the increment and decrement operators can be used.

Table 7-1: Increment and decrement operations

Statement	Operation	Description
`j = i++;`	post-increment	j is assigned the value of i, and then i is incremented by 1
`j = ++i;`	pre-increment	i is incremented by 1, and j is assigned the value of i
`j = i--;`	post-decrement	j is assigned the value of i, and then i is decremented by 1
`j = --i;`	pre-decrement	i is decremented by 1, and j is assigned the value of i

The following code fragments show how pre-increment versus post increment can affect the termination value of a loop.

```
while (i++ < LIMIT) begin: loop1
  ...  // last value of i will be LIMIT
end

while (++j < LIMIT) begin: loop2
  ...  // last value of j will be LIMIT-1
end
```

In `loop1`, the current value of `i` will first be compared to `LIMIT`, and then `i` will be incremented. Therefore, the last value of `i` within the loop will be equal to `LIMIT`.

In `loop2`, the current value of `j` will first be incremented, and then the new value compared to `LIMIT`. Therefore, the last value of `j` within the loop will be one less than `LIMIT`.

Avoiding race conditions

The Verilog language has two assignment operators, *blocking* and *nonblocking*. The blocking assignment is represented with a single equal token (**=**), and the nonblocking assignment is represented with a less-than-equal token (**<=**).

```
out = in;   // blocking assignment
out <= in;  // nonblocking assignment
```

blocking and nonblocking assignments

A full explanation of blocking and nonblocking assignments is beyond the scope of this book. A number of books on the Verilog language discuss the behavior of these constructs. The primary purpose of these two assignment operators is to accurately emulate the behavior of combinational and sequential logic in zero delay models. Proper usage of these two types of assignments is critical, in order to prevent simulation event race conditions. A general guideline is to use blocking assignments to model combinational logic, and nonblocking assignments to model sequential logic.

NOTE The `++` and `--` operators behave as blocking assignments.

++ and -- behave as blocking assignments

The increment and decrement operators behave as blocking assignments. The following two statements are semantically equivalent:

```
i++;        // increment i with blocking assign
i = i + 1;  // increment i with blocking assign
```

++ and -- can have race conditions in sequential logic

Just as it is possible to misuse the Verilog blocking assignment, creating a race condition within simulation, it is also possible to misuse the increment and decrement operators. The following example illustrates how an increment or decrement operator could be used in a manner that would create a simulation race condition. In this example, a simple counter is incremented using the `++` operator.

The counter, which would be implemented as sequential logic using some form of flip-flops, is modeled using a sequential logic **`always_ff`** procedural block. Another sequential logic procedural block reads the current value of the counter, and performs some type of functionality based on the value of the counter.

```
always_ff @(posedge clock)
  if (!resetN) count <= 0;
  else count++;  // same as count = count + 1;

always_ff @(posedge clock)
  case (state)
    HOLD: if (count == MAX)
              ...
```

Will `count` in this example be read by the second procedural block before or after `count` is incremented? This example has two procedural blocks that trigger at the same time, on the positive edge of `clock`. This creates a race condition, between the procedural block that increments `count` and the procedural block that reads the value of `count`. The defined behavior of a blocking assignment is that the software tool can execute the code above in either order. This means a concurrent process can read the value of a variable that is incremented with the `++` operator (or decremented with the `--` operator) before or after the variable has changed.

The pre-increment and pre-decrement operations will not resolve this race condition between two concurrent statements. Pre- and post- increment/decrement operations affect what order a variable is read and changed within the same statement. They do not affect the order of reading and changing between concurrent statements.

A nonblocking assignment is required to resolve the race condition in the preceding example. The behavior of a nonblocking assignment is that all concurrent processes will read the value of a variable before the assignment updates the value of the variable. This properly models the behavior of a transition propagating through sequential logic, such as the counter in this example.

TIP

Avoid using `++` and `--` on variables where nonblocking assignment behavior is required.

guidelines for using ++ and --

To prevent potential race conditions, the increment and decrement operators should only be used to model combinational logic.

Sequential and latched logic procedural blocks should not use the increment and decrement operators to modify any variables that are to be read outside of the procedural block. Temporary variables that are only read within a sequential or latched logic procedural block can use the ++ and -- operators without race conditions. For example, a variable used to control a **for** loop can use the ++ or -- operators even within a sequential procedural block, so long as the variable is not read anywhere outside of the procedural block.

The proper way to model the preceding example is shown below. The ++ operator is not used, because count is representing the output of sequential logic that is to be read by another concurrent procedural block.

```
always_ff @(posedge clock)
  if (!resetN) count <= 0;
  else count <= count + 1; // nonblocking assign

always_ff @(posedge clock)
  case (state)
    HOLD: if (count == MAX)
            ...
```

Synthesis guidelines

Both the pre- and post- forms of the increment and decrement operators are synthesizable. However, some synthesis compilers only support increment and decrement operations when used as a separate statement.

```
i++;         // synthesizable

if (--i)     // not synthesizable

sum = i++;   // not synthesizable
```

7.1.2 Assignment operators

+= and other assignment operators

SystemVerilog adds several additional types of assignment operators to Verilog. These new operators combine some type of operation with the assignment.

All of the new assignment operators have the same general syntax. For example, the **+=** operator is used as:

```
out += in;  // add in to out, and assign result
             // back to out
```

The **+=** operator is a short cut for the statement:

```
out = out + in;  // add and assign result to out
```

Table 7-2 lists the assignment operators which SystemVerilog adds to the Verilog language.

Table 7-2: SystemVerilog assignment operators

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>+=</td><td>add right-hand side to left-hand side and assign</td></tr>
<tr><td>-=</td><td>subtract right-hand side from left-hand side and assign</td></tr>
<tr><td>*=</td><td>multiply left-hand side by right-hand side and assign</td></tr>
<tr><td>/=</td><td>divide left-hand side by right-hand side and assign</td></tr>
<tr><td>%=</td><td>divide left-hand side by right-hand side and assign the remainder</td></tr>
<tr><td>&=</td><td>bitwise AND right-hand side with left-hand side and assign</td></tr>
<tr><td>|=</td><td>bitwise OR right-hand side with left-hand side and assign</td></tr>
<tr><td>^=</td><td>bitwise exclusive OR right-hand side with left-hand side and assign</td></tr>
<tr><td><<=</td><td>bitwise left-shift the left-hand side by the number of times indicated by the right-hand side and assign</td></tr>
<tr><td>>>=</td><td>bitwise right-shift the left-hand side by the number of times indicated by the right-hand side and assign</td></tr>
<tr><td><<<=</td><td>arithmetic left-shift the left-hand side by the number of times indicated by the right-hand side and assign</td></tr>
<tr><td>>>>=</td><td>arithmetic right-shift the left-hand side by the number of times indicated by the right-hand side and assign</td></tr>
</table>

NOTE Assignment operators behave as blocking assignments.

assignment operators are blocking assignments

The assignment operators have a blocking assignment behavior. To avoid simulation race conditions, the same care needs to be taken with these assignment operators as with the `++` and `--` increment and decrement operators, as described in section 7.1.1 on page 170.

Synthesis guidelines

The assignment operators are synthesizable, but synthesis compilers may place restrictions on multiply and divide operations. Some synthesis compilers do not support the use of assignment operators in compound expressions.

```
b += 5;          // synthesizable

b = (a+=5);    // not synthesizable
```

Example 7-1 illustrates using the SystemVerilog assignment operators. The operators are used in a combinational logic procedural block, which is the correct type of procedural block for blocking assignment behavior.

Example 7-1: Using SystemVerilog assignment operators

```
package definitions;
  typedef enum logic [2:0] {ADD,SUB,MULT,DIV,SL,SR} opcode_t;

  typedef enum logic {UNSIGNED, SIGNED} operand_type_t;

  typedef union packed {
    logic [23:0]        u_data;
    logic signed [23:0] s_data;
  } data_t;

  typedef struct packed {
    opcode_t         opc;
    operand_type_t   op_type;
    data_t           op_a;
    data_t           op_b;
  } instruction_t;
endpackage

import definitions::*;  // import package into $unit space

module alu (input instruction_t instr, output data_t alu_out);
  always_comb begin
    if (instr.op_type == SIGNED) begin
      alu_out.s_data = instr.op_a.s_data;
      unique case (instr.opc)
        ADD    : alu_out.s_data +=   instr.op_b.s_data;
        SUB    : alu_out.s_data -=   instr.op_b.s_data;
        MULT   : alu_out.s_data *=   instr.op_b.s_data;
        DIV    : alu_out.s_data /=   instr.op_b.s_data;
        SL     : alu_out.s_data <<<= 2;
        SR     : alu_out.s_data >>>= 2;
      endcase
```

```
    end
    else begin
      alu_out.u_data = instr.op_a.u_data;
       unique case (instr.opc)
        ADD    : alu_out.u_data +=   instr.op_b.u_data;
        SUB    : alu_out.u_data -=   instr.op_b.u_data;
        MULT   : alu_out.u_data *=   instr.op_b.u_data;
        DIV    : alu_out.u_data /=   instr.op_b.u_data;
        SL     : alu_out.u_data <<=  2;
        SR     : alu_out.u_data >>=  2;
      endcase
    end
  end
endmodule
```

7.1.3 Equality operators with don't care wildcards

Verilog has logical equality and case equality operators

The Verilog language has two types of equality operators, the **==** *logical equality operator* and the **===** *case equality operator* (also called the *identity operator*). Both operators compare two expressions, and return true if the expressions are the same, and false if they are different. A true result is represented a a one-bit logic 1 return value (`1'b1`), and a false result as a one-bit logic 0 return value (`1'b0`).

The two operators handle logic X and logic Z values in the operands differently:

- The **==** logical equality operator will consider any comparison where there are bits with X or Z values are in either operand to be unknown, and return a one-bit logic X (`1'bx`).
- The **===** case equality operator will perform a bit-wise comparison of the two operands, and look for an exact match of 0, 1, X and Z values in both operands. If the operands are identical, the operator will return true, otherwise, the operator will return false.

Each of these operators has a not-equal counterpart, **!=** and **!==**. These operators invert the results of the true/false test, returning true if the operands are not equal, and false if they are equal. An unknown result remains unknown.

the SystemVerilog wildcard equality operator allows masking out bits

SystemVerilog adds two new comparison operators, **==?** and **!=?**. These operators allow for don't-care bits to be masked from the comparison. The **==?** operator, referred to as the *wildcard equality operator*, performs a bit-wise comparison of its two operands, similar to the **==** logical equality operator. With the **==?** wildcard equality operator, however, a logic X or a logic Z in a bit position of the right-hand operand is treated as a wildcard that will match any value in the corresponding bit position of the other operand.

Table 7-3 shows the differences in the types of equality operators.

Table 7-3: SystemVerilog equality operators

a	b	a == b	a === b	a ==? b	a != b	a !== b	a !=? b
0000	0000	true	true	true	false	false	false
0000	0101	false	false	false	true	true	true
010Z	0101	unknown	false	unknown	unknown	false	unknown
010Z	010Z	unknown	true	true	unknown	false	false
010X	010Z	unknown	false	true	unknown	true	false
010X	010X	unknown	true	true	unknown	false	false

Observe that in the table above, X or Z bits in `a` are not masked out by `a ==? b` or `a !=? b`. These operators only consider X or Z bits in the right-hand operand as mask bits. X or Z bits in the left-hand operand are considered literal 4-state values. In Verilog, logic X in a number can be represented by the characters `x` or `X`, and logic Z in a number can be represented by the characters z, Z or ?.

```
logic [7:0] opcode;
...
if (opcode ==? 8'b11011???) // mask out low bits
  ...
```

If the operands are not the same size, then the wildcard equality operators will expand the vectors to the same size before performing the comparison. The vector expansion rules are the same as with the logical equality operators.

Synthesis guidelines

To synthesize the wildcard equality operator, the masked bits must be constant expressions. That is, the right-hand operand cannot be a variable where the masked bits could change during simulation.

```
logic [3:0] a, b;
logic       y1, y2;

assign y1 = (a ==? 4'b1??1); //synthesizable

assign y2 = (a ==? b);       //non synthesizable
```

7.1.4 Set membership operator — inside

SystemVerilog adds an operator to test if a value matches anywhere within a set of values. The operator uses the keyword, **inside**.

```
logic [2:0] a;

if ( a inside {3'b001, 3'b010, 3'b100} )
  ...
```

As with the **==?** wildcard equality operator, the **inside** operator can simplify comparing a value to several possibilities. Without the **inside** operator, the preceding **if** decision would likely have been coded as:

```
if ( (a==3'b001) || (a==3'b010) || (a==3'b100} )
  ...
```

With the **inside** operator, the set of values to which the first value is matched can be other signals.

```
if ( data inside {bus1, bus2, bus3, bus4} )
  ...
```

The set of values can also be an array. The next example tests to see if the value of 13 occurs anywhere in an array called `d_array`.

```
int d_array [0:1023];

if ( 13 inside {d_array} )
  ...
```

The **inside** operator uses the value Z or X (Z can also be represented with ?) to represent don't care conditions. The following test will be true if a has a value of `3’b101`, `3’b111`, `3’b1x1`, or `3’b1z1`. As with the **==?** wildcard equality operator, synthesis only permits the masked bits to be specified in constant expressions.

```
logic [2:0] a;

if (a inside {3’b1?1})
  ...
```

The **inside** operator can be used with **case** statements, as well as with **if** statements.

```
always_comb begin
  case (instruction) inside
    4'b0???:           opc = instruction[2:0];
    4'b1000, 4'b1100: opc = 3'b000;
    default:           opc = 3'b111;
  endcase
end
```

The **inside** operator is similar to the **casex** statement, but with two important differences. First, the inside operator can be used with both **if** decisions and **case** statements. Second, the **casex** statement treats Z and X values on both sides of the comparison as don’t care bits. The **inside** operator only treats Z and X values in the set of expressions after the **inside** keyword (the right-hand side of the comparison) as masked, don’t care bits. Bits in the first operand, the one before the **inside** keyword, are not treated as don’t care bits.

Synthesis guidelines

The **inside** operator is synthesizable. When masked expressions are used, synthesis requires that the expressions in the value set (on the right-hand side of the **inside** operator) be constant expressions. At the time this book was written, some synthesis compilers were not yet supporting the **inside** operator.

7.2 Operand enhancements

7.2.1 Operations on 2-state and 4-state types

operations with all 2-state types use Verilog operation rules

Verilog defines the rules for operations on a mix of most operand types. SystemVerilog extends these rules to also cover operations on 2-state types, which Verilog does not have. Operations on the new SystemVerilog types are performed using the same Verilog rules. This means most operations can return a value of 0, 1 or X for each bit of the result. When operations are performed on 2-state types, it is uncommon to see a result of X. Some operations on 2-state types can result in an X, however, such as a divide by 0 error.

7.2.2 Type casting

Verilog does type conversion using assignments

In Verilog, any a value of any type can be assigned to a variable of the same or any other type.Verilog automatically converts values of one type to another type using assignment statements. When a **wire** type is assigned to a **reg** variable, for example, the value on the wire (which has 4-state values, strength levels, and multi-driver resolution) is automatically converted to a **reg** value type (which has 4-state values, but no strength levels or multi-driver resolution). If a **real** type is assigned to a **reg** variable, the floating point value is automatically rounded off to an integer of the size of the **reg** bit-vector format.

The following example uses a temporary variable to convert a floating point result to a 64-bit integer value, which is then added to another integer and assigned to a 64-bit **reg** variable.

```
reg [63:0] a, y, temp;
real r;

temp = r**3; // convert result to 64-bit integer
y = a + temp;
```

SystemVerilog adds a type cast operator

SystemVerilog extends Verilog automatic conversion with a type cast operator. Type casting allows the designer to specify that a conversion should occur at any point during the evaluation of an expression, instead of just as part of an assignment. The syntax for type casting is:

```
type'(expression)
```

This syntax is different than C, which uses the format `(type)expression`. The different syntax is necessary to maintain backward compatibility with how Verilog uses parentheses, and to provide additional casting capabilities not in C (see sections 7.2.3 on page 181 on size casting and 7.2.4 on page 182 on sign casting).

Using SystemVerilog types and type casting, the Verilog example above can be coded without the use of a temporary variable, as follows:

```
longint a, y;
real r;

y = a + longint'(r**3);
```

7.2.3 Size casting

In Verilog, the number of bits of an expression is determined by the operand, the operation, and the context. The IEEE 1364-2005 Verilog standard defines the rules for determining the size of an expression. SystemVerilog follows the same rules as defined in Verilog.

vector widths can be cast to a different size

SystemVerilog extends Verilog by allowing the size of an expression to be cast to a different size. An explicit cast can be used to set the size of an operand, or to set the size of an operation result.

The syntax for the size casting operation is:

```
size’(expression)
```

Some examples of size casting are:

```
logic [15:0] a, b, c, sum;  // 16 bits wide
logic        carry;         // 1 bit wide

sum = a + 16’(5);          // cast operand

{carry,sum} = 17’(a + 3);  // cast result

sum = a + 16’(b - 2) / c;  // cast intermediate
                           // result
```

If an expression is cast to a smaller size than the number of bits in the expression, the left-most bits of the expression are truncated. If the expression is cast to a larger vector size, then the expression is left-extended. An unsigned expression is left-extended with 0. A signed expression is left-extended using sign extension. These are the same rules as when an expression of one size is assigned to a variable or net of a different size.

7.2.4 Sign casting

SystemVerilog follows Verilog rules for determining if an operation result is signed or unsigned. SystemVerilog also allows explicitly casting the signedness of a value. Either the signedness of an operand can be cast, or the signedness of an operation result can be cast.

The syntax for the sign casting operation is:

```
signed'(expression)

unsigned'(expression)
```

Some examples of sign casting are:

```
sum = signed'(a) + signed'(a); // cast operands

if (unsigned'(a-b) <= 5)  // cast intermediate
  ...                     // result
```

The SystemVerilog sign cast operator performs the same conversion as the Verilog **$signed** and **$unsigned** system functions. Sign casting is synthesizable, following the same rules as the **$signed** and **$unsigned** system functions.

7.3 Enhanced for loops

Verilog for loop variables are declared outside the loop

In Verilog, the variable used to control a **for** loop must be declared prior to the loop. When multiple **for** loops might run in parallel (concurrent loops), separate variables must be declared for each loop. In the following example, there are three loops that can be executing at the same time.

```
module chip (...);  // Verilog style loops
  reg [7:0] i;
  integer j, k;

  always @(posedge clock) begin
    for (i = 0; i <= 15; i = i + 1)
      for (j = 511; j >= 0; j = j - 1) begin
```

```
        ...
      end
    end

    always @(posedge clock) begin
      for (k = 1; k <= 1024; k = k + 2) begin
        ...
      end
    end
  endmodule
```

concurrent loops can interfere with each other

Because the variable must be declared outside of the **for** loop, caution must be observed when concurrent procedural blocks within a module have **for** loops. If the same variable is inadvertently used as a loop control in two or more concurrent loops, then each loop will be modifying the control variable used by another loop. Either different variables must be declared at the module level, as in the example above, or local variables must be declared within each concurrent procedural block, as shown in the following example.

```
module chip (...);  // Verilog style loops
  ...
  always @(posedge clock) begin: loop1
    reg [7:0] i;  // local variable
    for (i = 0; i <= 15; i = i + 1) begin
      ...
    end
  end

  always @(posedge clock) begin: loop2
    integer i;   // local variable
    for (i = 1; i <= 1024; i = i + i) begin
      ...
    end
  end
endmodule
```

7.3.1 Local variables within for loop declarations

declaring local loop variables

SystemVerilog simplifies declaring local variables for use in **for** loops. With SystemVerilog, the declaration of the **for** loop variable can be made within the **for** loop itself. This eliminates the need to define several variables at the module level, or to define local variables within named **begin**...**end** blocks.

In the following example, there are two loops that can be executing at the same time. Each loop uses a variable called **i** for the loop control. There is no conflict, however, because the **i** variable is local and unique for each loop.

```
module chip (...); // SystemVerilog style loops
  ...
  always_ff @(posedge clock) begin
    for (bit [4:0] i = 0; i <= 15; i++)
        ...
  end

  always_ff @(posedge clock) begin
    for (int i = 1; i <= 1024; i += 1)
        ...
  end
endmodule
```

local loop variables prevent interference

A variable declared as part of a **for** loop is local to the loop. References to the variable name within the loop will see the local variable, and not any other variable of the same name elsewhere in the containing module, interface, program, task, or function.

NOTE Variables declared as part of a **for** loop are automatic variables.

local loop variables are automatic

When a variable is declared as part of a **for** loop initialization statement, the variable has automatic storage, not static storage. The variable is automatically created and initialized when the **for** loop is invoked, and destroyed when the loop exits. The use of automatic variables has important implications:

- Automatic variables cannot be referenced hierarchically.
- Automatic variables cannot be dumped to VCD files.
- The value of the **for** loop variable cannot be used outside of the **for** loop, because the variable does not exist outside of the loop.

local loop variables do not exist outside of the loop

The following example is illegal. The intent is to use a **for** loop to find the lowest bit that is set within a 64 bit vector. Because the `lo_bit` variable is declared as part of the **for** loop, however, it is only in existence while the loop is running. When the loop terminates, the variable disappears, and cannot be used after the loop.

```
always_comb begin
 for (int lo_bit=0; lo_bit<=63; lo_bit++) begin
    if (data[lo_bit]) break;  // exit loop if
 end                          // bit is set
```

```
 if (lo_bit > 7)  // ERROR: lo_bit is not there
   ...
end
```

When a variable needs to be referenced outside of a loop, the variable must be declared outside of the loop. The following example uses a local variable in an unnamed **begin**...**end** block (another SystemVerilog enhancement, see section 2.3 on page 26 of Chapter 2).

```
always_comb begin
  int lo_bit;  // local variable to the block
  for (lo_bit=0; lo_bit<=63; lo_bit++) begin
    if (data[lo_bit]) break;  // exit loop if
  end                         // bit is set
  if (lo_bit > 7)  // lo_bit has last loop value
    ...
end
```

7.3.2 Multiple for loop assignments

SystemVerilog also enhances Verilog **for** loops by allowing more than one initial assignment statement, and more than one step assignment statement. Multiple initial or step assignments are separated by commas. For example:

```
for (int i=1, j=0; i*j < 128; i++, j+=3)
  ...
```

Each loop variable can be declared as a different type.

```
for (int i=1, byte j=0; i*j < 128; i++, j+=3)
  ...
```

7.3.3 Hierarchically referencing variables declared in for loops

local loop variables do not have a hierarchy path

Local variables declared as part of a **for** loop cannot be referenced hierarchically. A testbench, waveform display, or a VCD file cannot reference the local variable (however, tools may provide proprietary, non-standard ways to access these variables).

```
always_ff @(posedge clock) begin
  for (int i = 0; i <= 15; i++) begin
    ...// i cannot be referenced hierarchically
  end
```

```
end
```

When hierarchical references to a **for** loop control variable are required, the variable should be declared outside of the **for** loop, either at the module level, or in a named **begin**...**end** block.

```
always_ff @(posedge clock) begin : loop
  int i;  // i can be referenced hierarchically
  for (i = 0; i <= 15; i++) begin
    ...
  end
end
```

In this example, the variable `i` can be referenced hierarchically with the last portion of the hierarchy path ending with `.loop.i`.

7.3.4 Synthesis guidelines

SystemVerilog's enhanced **for** loops are synthesizable, following the same synthesis coding guidelines as Verilog **for** loops.

7.4 Bottom testing do...while loop

Verilog has the **while** loop, which executes the loop as long as a loop-control test is true. The control value is tested at the beginning of each pass through the loop.

a while loop might not execute at all

It is possible that a **while** loop might not execute at all. This will occur if the test of the control value is false the very first time the loop is encountered in the execution flow.

This top-testing behavior of the **while** loop can require extra code prior to the loop, in order to ensure that any output variables of the loop are consistent with variables that would have been read by the loop. In the following example, the **while** loop executes as long as an input address is within the range of 128 to 255. If, however, the address is not in this range when the procedural block triggers, the **while** loop will not execute at all. Therefore, the range has to be checked prior to the loop, and the three loop outputs, `done`, `OutOfBound`, and `out` set for out-of-bounds address conditions, based on the value of `addr`.

```
always_comb begin
```

```
    if (addr < 128 || addr > 255) begin
      done = 0;
      OutOfBound = 1;
      out = mem[128];
    end
    else while (addr >= 128 && addr <= 255) begin
      if (addr == 128) begin
        done = 1;
        OutOfBound = 0;
      end
      else begin
        done = 0;
        OutOfBound = 0;
      end
      out = mem[addr];
      addr -= 1;
    end
  end
```

a do...while loop will execute at least once

SystemVerilog adds a **do**...**while** loop, as in C. With the **do**...**while** loop, the control for the loop is tested at the end of each pass of the loop, instead of the beginning. This means that each time the loop is encountered in the execution flow, the loop statements will be executed at least once.

The basic syntax of a **do**...**while** loop is:

```
do <statement or statement block>
while (<condition>);
```

If the **do** portion of the loop contains more than one statement, the statements must be grouped using **begin**...**end** or **fork**...**join**. The **while** statement comes after the block of statements to be executed. Note that there is a semicolon after the **while** statement.

Because the statements within a **do**...**while** loop are guaranteed to execute at least once, all the logic for setting the outputs of the loop can be placed inside the loop. This bottom-testing behavior can simplify the coding of while loops, making the code more concise and more intuitive.

In the next example, the **do**...**while** loop will execute at least once, thereby ensuring that the `done`, `OutOfBound`, and `out` variables are consistent with the input to the loop, which is `addr`. No additional logic is required before the start of the loop.

```
always_comb begin
  do begin
    done = 0;
    OutOfBound = 0;
    out = mem[addr];
    if (addr < 128 || addr > 255) begin
      OutOfBound = 1;
      out = mem[128];
    end
    else if (addr == 128) done = 1;
    addr -= 1;
  end
  while (addr >= 128 && addr <= 255);
end
```

7.4.1 Synthesis guidelines

Verilog **while** loops are synthesizable, with a number of restrictions. These same restrictions apply to SystemVerilog's **do**...**while** loop. The restrictions allow synthesis compilers to statically determine how many times a loop will execute. The example code snippets shown in this section represent behavioral code, and do not meet all of the RTL guidelines for synthesizing **while** and **do**...**while** loops.

7.5 The foreach array looping construct

SystemVerilog adds a **foreach** loop, which can be used to iterate over the elements of single- and multi-dimensional arrays, without having to specify the size of each array dimension. The **foreach** loop is discussed in section 5.4 on page 130 of Chapter 5, on arrays.

7.6 New jump statements — break, continue, return

Verilog uses the **disable** statement as a way to cause the execution flow of a sequence of statements to jump to a different point in the execution flow. Specifically, the **disable** statement causes the execution flow to jump to the end of a named statement group, or to the end of a task.

the disable statement is both a continue and a break

The Verilog **disable** statement can be used a variety of ways. It can be used to jump to the end of a loop, and continue execution with the next pass of the loop. The same **disable** statement can also be used to prematurely break out of all passes of a loop. The multiple usage of the same keyword can make it difficult to read and maintain complex blocks of code. Two ways of using **disable** are illustrated in the next example. The effect of the **disable** statement is determined by the placement of the named blocks being disabled.

```
// find first bit set within a range of bits
always @* begin
  begin: loop
    integer i;
    first_bit = 0;
    for (i=0; i<=63; i=i+1) begin: pass
      if (i < start_range)
        disable pass;  // continue loop
      if (i > end_range)
        disable loop;  // break out of loop
      if ( data[i] ) begin
        first_bit = i;
        disable loop;  // break out of loop
      end
    end // end of one pass of loop
  end // end of the loop
  ... // process data based on first bit set
end
```

the disable statement can be used as a return

The **disable** statement can also be used to return early from a task, before all statements in the task have been executed.

```
task add_up_to_max (input  [ 5:0] max,
                    output [63:0] result);
  integer i;
  begin
    result = 1;
    if (max == 0)
      disable add_up_to_max; // exit task
    for (i=1; i<=63; i=i+1) begin
      result = result + result;
      if (i == max)
        disable add_up_to_max; // exit task
    end
  end
endtask
```

The **disable** statement can also be used to externally disable a concurrent process or task. An external disable is not synthesizable, however.

continue, break and return statements

SystemVerilog adds the C language jump statements: **break**, **continue** and **return**. These jump statements can make code more intuitive and concise. SystemVerilog does *not* include the C **goto** statement.

An important difference between Verilog's **disable** statement and these new jump statements is that the **disable** statement applies to all currently running invocations of a task or block, whereas **break**, **continue** and **return** only apply to the current execution flow.

7.6.1 The continue statement

The C-like **continue** statement jumps to the end of the loop and executes the loop control. Using the **continue** statement, it is not necessary to add named **begin**...**end** blocks to the code, as is required by the **disable** statement.

```
logic [15:0] array [0:255];

always_comb begin
  for (int i = 0; i <= 255; i++) begin : loop
    if (array[i] == 0)
      continue;  // skip empty elements
    transform_function(array[i]);
  end  // end of loop
end
```

7.6.2 The break statement

The C-like **break** statement terminates the execution of a loop immediately. The loop is not executed again unless the execution flow of the procedural block encounters the beginning of the loop again, as a new statement.

```
// find first bit set within a range of bits
always_comb begin
  first_bit = 0;
  for (int i=0; i<=63; i=i+1) begin
    if (i < start_range) continue;
    if (i > end_range)   break;  // exit loop
    if ( data[i] ) begin
```

```
        first_bit = i;
        break;  // exit loop
      end
    end  // end of the loop
    ...  // process data based on first bit set
  end
```

The SystemVerilog **break** statement is used in the same way as a **break** in C to break out of a loop. C also uses the **break** statement to exit from a **switch** statement. SystemVerilog does not use **break** to exit a Verilog **case** statement (analogous to a C **switch** statement). A **case** statement exits automatically after a branch is executed, without needing to execute a **break**.

7.6.3 The return statement

SystemVerilog adds a C-like **return** statement, which is used to return a value from a non-void function, or to exit from a void function or a task. The **return** statement can be executed at any time in the execution flow of the task or function. When the **return** is executed, the task or function exits immediately, without needing to reach the end of the task or function.

```
task add_up_to_max (input  [ 5:0] max,
                    output [63:0] result);
  result = 1;
  if (max == 0) return; // exit task
  for (int i=1; i<=63; i=i+1) begin
    result = result + result;
    if (i == max) return; // exit task
  end
endtask
```

The **return** statement can be used to exit early from either a task or a function. The Verilog **disable** statement can only cause a task to exit early. It cannot be used with functions.

```
function automatic int log2 (input int n);
  if (n <=1) return 1;  // exit function early
  log2 = 0;
  while (n > 1) begin
    n = n/2;
    log2++;
  end
  return log2;
endfunction
```

Note that the **return** keyword must not be followed by an expression in a task or void function, and must be followed by an expression in a non-void function.

7.6.4 Synthesis guidelines

The **break**, **continue**, and **return** jump statements are synthesizable constructs. The synthesis results are the same as if a Verilog **disable** statement had been used to model the same functionality.

7.7 Enhanced block names

Complex code will often have several nested **begin**...**end** statement blocks. In such code, it can be difficult to recognize which **end** is associated with which **begin**.

code can have several nested begin...end blocks

The following example illustrates how a single procedural block might contain several nested **begin**...**end** blocks. Even with proper indenting and keyword bolding as used in this short example, it can be difficult to see which **end** belongs with which **begin**.

Example 7-2: Code snippet with unnamed nested **begin**...**end** blocks

```
always_ff @(posedge clock, posedge reset)
  begin
    logic breakVar;
    if (reset) begin
      ... // reset all outputs
    end
    else begin
      case (SquatState)
        wait_rx_valid:
          begin
            Rxready <= '1;
            breakVar = 1;
            for (int j=0; j<NumRx; j+=1) begin
              for (int i=0; i<NumRx; i+=1) begin
                if (Rxvalid[i] && RoundRobin[i] && breakVar)
                  begin
                    ATMcell <= RxATMcell[i];
                    Rxready[i] <= 0;
```

```
                    SquatState <= wait_rx_not_valid;
                    breakVar = 0;
                  end
              end
            end
          end
        ... // process other SquatState states
      endcase
    end
  end
```

named ends can be paired with named begins

Verilog allows a statement block to have a name, by appending `:<name>` after the **begin** keyword. The block name creates a local hierarchy scope that serves to identify all statements within the block. SystemVerilog allows (but does not require) a matching block name after the **end** keyword. This additional name does not affect the block semantics in any way, but does serve to enhance code readability by documenting which statement group is being completed.

To specify a name to the end of a block, a `:<name>` is appended after the **end** keyword. White space is allowed, but not required, before and after the colon.

```
begin: <block_name>
  ...
end: <block_name>
```

The optional block name that follows an **end** must match exactly the name with the corresponding **begin**. It is an error for the corresponding names to be different.

The following code snippet modifies example 7-2 on the previous page by adding names to the **begin**...**end** statement groups, helping to make the code easier to read.

Example 7-3: Code snippet with named **begin** and named **end** blocks

```
always_ff @(posedge clock, posedge reset)
  begin: FSM_procedure
    logic breakVar;
    if (reset) begin: reset_logic
      ... // reset all outputs
    end: reset_logic
```

```
    else begin: FSM_sequencer
      unique case (SquatState)
        wait_rx_valid:
          begin: rx_valid_state
            Rxready <= '1;
            breakVar = 1;
            for (int j=0; j<NumRx; j+=1) begin: loop1
              for (int i=0; i<NumRx; i+=1) begin: loop2
                if (Rxvalid[i] && RoundRobin[i] && breakVar)
                  begin: match
                    ATMcell <= RxATMcell[i];
                    Rxready[i] <= 0;
                    SquatState <= wait_rx_not_valid;
                    breakVar = 0;
                  end: match
              end: loop2
             end: loop1
          end: rx_valid_state
        ... // process other SquatState states
      endcase
    end: FSM_sequencer
  end: FSM_procedure
```

7.8 Statement labels

a named block identifies a group of statements

In addition to named blocks of statements, SystemVerilog allows a label to be specified before any procedural statement. Statement labels use the same syntax as C:

```
<label> : <statement>
```

a statement label identifies a single statement

A statement label is used to identify a single statement, whereas a named statement block identifies a block of one of more statements.

```
always_comb begin : decode_block
  decoder : case (opcode)
    2'b00:
      outer_loop: for (int i=0; i<=15; i++)
        inner_loop: for (int j=0; j<=15; j++)
           //...
    ... // decode other opcode values
  endcase
end : decode_block
```

a labeled statement can help document code

Statement labels document specific lines of code, which can help make the code more readable, and can make it easier to reference those lines of code in other documentation. Statement labels can also be useful to identify specific lines of code for debug utilities and code coverage analysis tools. Statement labels also allow statements to be referenced by name. A statement that is in the process of execution can be aborted using the **disable** statement, in the same way that a named statement group or task can be disabled.

Labeled statement blocks

a statement block can have a name or a label

A **begin**...**end** block is a statement, and can therefore have either a statement label or a block name.

```
begin: block1   // named block
  ...
end: block1


block2: begin   // labeled block
          ...
        end
```

It is illegal to give a statement block both a label and a block name.

7.9 Enhanced case statements

The Verilog **case**, **casex**, and **casez** statements allow the selection of one branch of logic out of multiple choices. For example:

```
always_comb
  case (opcode)                    case expression
                2'b00:   y = a + b;
case selection  2'b01:   y = a - b;
items           2'b10:   y = a * b;
                2'b11:   y = a / b;
  endcase
```

The expression following the **case**, **casex**, or **casez** keyword is referred to as the *case expression*. The expressions to which the case expression is matched are referred to as the *case selection items*.

simulation and synthesis might interpret case statements differently

The Verilog standard specifically defines that case statements must evaluate the case selection items in the order in which they are listed. This infers that there is a priority to the case items, the same as in a series of **if**...**else**...**if** decisions. Software tools such as synthesis compilers will typically try to optimize out the additional logic required for priority encoding the selection decisions, if the tool can determine that all of the selection items are mutually exclusive.

SystemVerilog provides special **unique** and **priority** modifiers to **case**, **casex**, and **casez** decisions. These modifiers are placed before the **case**, **casex**, or **casez** keywords:

```
unique case (<case_expression>)
  ... // case items
endcase

priority case (<case_expression>)
  ... // case items
endcase
```

7.9.1 Unique case decisions

A **unique case** statement specifies that:

- Only one case select expression matches the case expression when it is evaluated
- One case select expression must match the case expression when it is evaluated

a unique case can be evaluated in parallel

The **unique** modifier allows designers to explicitly specify that the order of the case selection items is not significant, and the selections are permitted to be evaluated in parallel. Software tools can optimize out the inferred priority of the selection order. The **unique** modifier also specifies that the case selection items are complete (or full). Any case expression value that occurs should match one, and only one, case select item. The following example illustrates a case statement where it is obvious that the case selection items are both mutually exclusive and that all possible case select values are specified. The **unique** keyword documents and verifies that these conditions are true.

```
always_comb
  unique case (opcode)
```

```
    2'b00:   y = a + b;
    2'b01:   y = a - b;
    2'b10:   y = a * b;
    2'b11:   y = a / b;
  endcase
```

Checking for unique conditions

a unique case cannot have overlapping conditions

When a **case**, **casex**, or **casez** statement is specified as **unique**, software tools must perform additional semantic checks to verify that each of the case selection items is mutually exclusive. If a case expression value occurs during run time that matches more than one case selection item, the tool must generate a run-time warning message.

In the following code snippet, a **casez** statement is used to allow specific bits of the selection items to be excluded from the comparison with the case expression. When specifying don't care bits, it is easy to inadvertently specify multiple case selection items that could be true at the same time. In the example below, a **casez** statement is used to decode which of three bus request signals is active. The designer's expectation is that the design can only issue one request at a time. The **casez** selection allows comparing to one specific request bit, and masking out the other bits, which could reduce the gate-level logic needed. Since only one request should occur at a time, the order in which the 3 bits are examined should not matter, and there should never be two case items true at the same time.

```
logic [2:0] request;

always_comb
  casez (request)  // design should
                   // only generate one
                   // grant at a time
    3'b1??: slave1_grant = 1;
    3'b?1?: slave2_grant = 1;
    3'b??1: slave3_grant = 1;
  endcase
```

In the preceding example, the **casez** statement will compile for simulation without an error. If a case expression value could match more than one case selection item (two requests occurred at the same time, for example), then only the first matching branch is executed. No run-time warning is generated to alert the designer or verification engineer of a potential design problem. Though the code in the example above is legal, lint check programs and synthesis compilers will generally warn that there is a potential overlap in the case items. However, these tools have no way to determine if the designer intended to have an overlap in the case select expressions.

The **unique** modifier documents that the designer did not intend, or expect, that two case select items could be true at the same time. When the **unique** modifier is added, all software tools, including simulators, will generate a warning any time the case statement is executed and the case expression matches multiple case items.

```
logic [2:0] request;

always_comb
  unique casez (request)  // design should
                          // only generate one
                          // grant at a time
    3'b1??: slave1_grant = 1;
    3'b?1?: slave2_grant = 1;
    3'b??1: slave3_grant = 1;
  endcase
```

Detecting incomplete case selection lists

a unique case must specify all conditions

When a **case**, **casex**, or **casez** statement is specified as **unique**, software tools will issue a run-time warning if the value of the case expression does not match any of the case selection items, and there is no default case.

The following example will result in a run-time warning if, during simulation, `opcode` has a value of 3, 5, 6 or 7:

```
logic [2:0] opcode;  // 3-bit wide vector

always_comb
  unique case (opcode)
    3'b000: y = a + b;
    3'b001: y = a - b;
    3'b010: y = a * b;
    3'b100: y = a / b;
  endcase
```

Though **unique** is primarily a run-time check that one, and only one, case select item is true, software tools may report an overlap warning in unique case expression items at compile time, if the case items are all constant expressions. Tools such as synthesis compilers and lint checkers that do not have a dynamic run time can only perform static checks for select item overlaps.

Using unique case with always_comb

Both **`always_comb`** and **`unique case`** help ensure that the logic of a procedural block can be realized as combinational logic. There are differences in the checks that **`unique case`** performs and the checks that **`always_comb`** performs. The use of both constructs helps ensure that complex procedural blocks will synthesize as the intended logic.

A **`unique case`** statement performs run-time checks to ensure that every case expression value that occurs matches one and only one case selection item, so that a branch of the **`case`** statement is executed for every occurring case expression value. An advantage of run-time checking is that only the actual values that occur during simulation will be checked for errors. A disadvantage of run-time checking is that the quality of the error checking is dependent on the thoroughness of the verification tests.

The **`always_comb`** procedural block has specific semantic rules to ensure combinational logic behavior during simulation (refer to sections 6.2.1 on page 142). Optionally, software tools can perform additional compile-time analysis of the statements within an **`always_comb`** procedural block to check that the statements conform to general guidelines for modeling combinational logic. Having both the static checking of **`always_comb`** and the run-time checking of **`unique case`** helps ensure that the designer's intent has been properly specified.

7.9.2 Priority case statements

A **`priority case`** statement specifies that:

- At least one case select expression must match the case expression when it is evaluated

a priority case might have multiple case item matches

- If more than one case select expression matches the case expression when it is evaluated, the first matching branch must be taken

The **`priority`** modifier indicates that the designer considers it to be OK for two or more case selection expressions to be true at the same time, and that the order of the case selection items is important. In the following example, the designer has specified that there is priority to the order in which interrupt requests are decoded, with `irq0` having the highest priority.

```
always_comb
  priority case (1'b1)
    irq0: irq = 4'b0001;
    irq1: irq = 4'b0010;
    irq2: irq = 4'b0100;
    irq3: irq = 4'b1000;
  endcase
```

Because the model explicitly states that case selection items should be evaluated in order, all software tools must maintain the inferred priority encoding, should it be possible for multiple case selection items to match.

NOTE Synthesis compilers might optimize case selection item evaluation differently than the RTL code, even when priority case is used.

Some synthesis compilers might automatically optimize **`priority case`** statements to parallel evaluation if the compiler sees that the case selection items are mutually exclusive. If it is not possible for multiple case selection items to be true at the same time, the additional priority-encoded logic is not required in the gate-level implementation of the functionality.

Preventing unintentional latched logic

a priority case must specify all conditions

When the **`priority`** modifier is specified with a **`case`**, **`casex`**, or **`casez`** statement, all values of the case expression that occur during run time must have at least one matching case selection item. If there is no matching case selection item, a run-time warning will occur. This ensures that when the case statement is evaluated, a branch will be executed. The logic represented by the case statement can be implemented as combinational logic, without latches.

7.9.3 Unique and priority versus parallel_case and full_case

The IEEE 1364.1 synthesis standard[1] for Verilog specifies special commands, referred to as pragmas, to modify the behavior of synthesis compilers. The 1364.1 pragmas are specified using the Verilog attribute construct. Synthesis compilers also allow pragmas to be hidden within Verilog comments.

synthesis parallel_case pragma

One of the pragmas specified in the Verilog synthesis standard is **parallel_case**. This instructs synthesis compilers to remove priority encoding, and evaluate all case selection items in parallel.

```
always_comb
  (* synthesis, parallel_case *)
  case (opcode)
    2'b00:  y = a + b;
    2'b01:  y = a - b;
    2'b10:  y = a * b;
    2'b11:  y = a / b;
  endcase
```

synthesis full_case pragma

Another pragma is **full_case**. This pragma instructs the synthesis compiler that, for all unspecified case expression values, the outputs assigned within the case statement are unused, and can be optimized out by the synthesis compiler.

```
always_comb
  (* synthesis, full_case *)
  case (State)
    3'b001:  NextState = 3'b010;
    3'b010:  NextState = 3'b100;
    3'b100:  NextState = 3'b001;
  endcase
```

unique and priority do more than synthesis pragmas

For synthesis, a **unique case** is equivalent to enabling both the `full_case` and `parallel_case` pragmas. A **priority case** is equivalent to enabling the `full_case` pragma. However, the SystemVerilog **unique** and **priority** decision modifiers do more than the `parallel_case` and `full_case` pragmas. These modifi-

1. 1364.1-2002 IEEE Standard for Verilog Register Transfer Level Synthesis. See page xxvii of this book for details.

ers reduce the risk of mismatches between software tools, and provide additional semantic checks that can catch potential design problems much earlier in the design cycle.

unique case enforces semantic rules

The **`unique`** case modifier combines the functionality of both the `parallel_case` and `full_case` pragmas, plus added semantic checking. The 1364.1 Verilog synthesis standard states that the `parallel_case` pragma will force a parallel evaluation, even if more than one case selection item will evaluate as true. This could result in more than one branch of a case statement executing at the same time. A **`unique case`** statement will generate run-time warnings, should the designer's assumptions that the case statement is both parallel and complete prove incorrect. The `parallel_case/full_case` pragmas do not impose any checking on the case selection items.

priority case can prevent mismatches

The **`priority`** modifier provides the functionality of the `full_case` synthesis pragma, plus additional semantic checks. When the `full_case` pragma is used, no assignment is made to the outputs of the **`case`** statement for the unspecified values of the case expression. In RTL simulations, these outputs will be unchanged, and reflect the value of previous assignments. In the gate-level design created by synthesis, the outputs will be driven to some optimized value. This driven value can be, and likely will be, different than the value of the outputs in the RTL model. This difference can result in mismatches between pre-synthesis RTL simulations and post-synthesis gate-level simulations, if an unspecified case expression value is encountered. Equivalence checkers will also see a difference in the two models.

Synthesis pragmas modify how synthesis interprets the Verilog case statements, but they do not affect simulation semantics and might not affect the behavior of other software tools. This can lead to mismatches in how different tools interpret the same case statement. The **`unique`** and **`priority`** modifiers are part of the language, instead of being an informational synthesis pragma. As part of the language, simulation, synthesis compilers, formal verification tools, lint checkers and other software tools can apply the same semantic rules, ensuring consistency across various tools.

The run-time semantic checks provided by the **`unique`** and **`priority`** modifiers also help ensure that the logic within a **`case`**, **`casex`**, or **`casez`** statement will behave consistent with the intent specified by the designer. These restrictions can prevent subtle, difficult to detect logic errors within a design.

7.10 Enhanced if...else decisions

The SystemVerilog **unique** and **priority** decision modifiers also work with **if**...**else** decisions. These modifiers can also reduce ambiguities with this type of decision, and can trap potential design errors early in the modeling phase of a design.

The Verilog **if**...**else** statement is often nested to create a series of decisions. For example:

```
logic [2:0] sel;

always_comb begin
  if      (sel == 3'b001) mux_out = a;
  else if (sel == 3'b010) mux_out = b;
  else if (sel == 3'b100) mux_out = c;
end
```

simulation and synthesis might interpret if...else differently

In simulation, a series of **if**...**else**...**if** decisions will be evaluated in the order in which the decisions are listed. To maintain the same ordering in hardware implementation, priority encoded logic would be required. Often, however, the specific order is not essential in the desired logic. The order of the decisions is merely the way the engineer happened to list them in the source code.

7.10.1 Unique if...else decisions

a unique if...else can be evaluated in parallel

The **unique** modifier indicates that the designer's intent is that the order of the decisions is not important. Software tools can optimize out the inferred priority of the decision order. For example:

```
logic [2:0] sel;

always_comb begin
  unique if (sel == 3'b001) mux_out = a;
    else if (sel == 3'b010) mux_out = b;
    else if (sel == 3'b100) mux_out = c;
end
```

Checking for unique conditions

a unique if...else cannot have overlapping conditions

Software tools will perform checking on a **unique if** decision sequence to ensure that all decision conditions in a series of **if**...**else**...**if** decisions are mutually exclusive. This allows the decision series to be executed in parallel, without priority encoding. A software tool will generate a run-time warning if it determines that more than one condition is true. This warning message can occur at either compile time or run-time. This additional checking can help detect modeling errors early in the verification of the model.

In the following example, there is an overlap in the decision conditions. Any or all of the conditions for the first, second and third decisions could be true at the same time. This means that the decisions must be evaluated in the order listed, rather than in parallel. Because the **unique** modifier was specified, software tools can generate a warning that the decision conditions are not mutually exclusive.

```
logic [2:0] sel;

always_comb begin
  unique if (sel[0]) mux_out = a;
    else if (sel[1]) mux_out = b;
    else if (sel[2]) mux_out = c;
end
```

Preventing unintentional latched logic

a unique if...else warns of unspecified conditions

When the **unique** modifier is specified with an **if** decision, software tools are required to generate a run-time warning if the **if** statement is evaluated and no branch is executed. The following example would generate a run-time warning if the **unique if**...**else**...**if** sequence is entered and `sel` has any value other than 1, 2 or 4.

```
always_comb begin
  unique if (sel == 3'b001) mux_out = a;
    else if (sel == 3'b010) mux_out = b;
    else if (sel == 3'b100) mux_out = c;
end
```

This run-time semantic check guarantees that all conditions in the decision sequence that actually occur during run time have been fully specified. When the decision sequence is evaluated, one branch will be executed. This helps ensure that the logic represented by the decisions can be implemented as combinational logic, without the need for latches.

7.10.2 Priority if decisions

a priority if...else must evaluate in order

The **priority** modifier indicates that the designer's intent is that the order of the decisions is important. Software tools should maintain the order of the decision sequence. For example:

```
always_comb begin
  priority if (irq0) irq = 4'b0001;
       else if (irq1) irq = 4'b0010;
       else if (irq2) irq = 4'b0100;
       else if (irq3) irq = 4'b1000;
end
```

Because the model explicitly states that the decision sequence above should be evaluated in order, all software tools should maintain the inferred priority encoding. The **priority** modifier ensures consistent behavior from software tools. Simulators, synthesis compilers, equivalence checkers, and formal verification tools can all interpret the decision sequence in the same way.

Preventing unintentional latched logic

a priority if...else must specify all conditions

As with the **unique** modifier, when the **priority** modifier is specified with an **if** decision, software tools will perform run-time checks that a branch is executed each time an **if**...**else**...**if** sequence is evaluated. A run-time warning will be generated if no branch of a **priority if**...**else**...**if** decision sequence is executed. This helps ensure that all conditions in the decision sequence that actually occur during run time have been fully specified, and that when the decision sequences are evaluated, a branch will be executed. The logic represented by the decision sequence can be implemented as priority-encoded combinational logic, without latches.

Synthesis guidelines

An **if**...**else**...**if** decision sequence that is qualified with **unique** or **priority** is synthesizable.

7.11 Summary

A primary goal of SystemVerilog is to enable modeling large, complex designs more concisely than was possible with Verilog. This chapter presented enhancements to the procedural statements in Verilog that help to achieve that goal. New operators, enhanced **`for`** loops, bottom-testing loops, and **`unique`**/**`priority`** decision modifiers all provide new ways to represent design logic with efficient, intuitive code.

Chapter 8

Modeling Finite State Machines with SystemVerilog

SystemVerilog enables modeling at a higher level of abstraction through the use of 2-state types, enumerated types, and user-defined types. These are complemented by new specialized always procedural blocks, **`always_comb`**, **`always_ff`** and **`always_latch`**. These and other new modeling constructs have been discussed in the previous chapters of this book.

This chapter shows how to use these new levels of model abstractions to effectively model logic such as finite state machines, using a combination of enumerated types and the procedural constructs presented in the previous chapters. Using SystemVerilog, the coding of finite state machines can be simplified and made easier to read and maintain. At the same time, the consistency of how different software tools interpret the Verilog models can be increased.

The SystemVerilog features presented in this chapter include:

- Using enumerated types for modeling Finite State Machines
- Using enumerated types with FSM **`case`** statements
- Using **`always_comb`** with FSM **`case`** statements
- Modeling reset logic with enumerated types and 2-state types

8.1 Modeling state machines with enumerated types

Section 4.2 on page 79 introduced the enumerated type construct that SystemVerilog adds to the Verilog language. This section provides additional guidelines on using enumerated types for modeling hardware logic such as finite state machines.

enumerated types have restricted values

Enumerated types provide a means for defining a variable that has a restricted set of legal values. The values are represented with labels instead of digital logic values.

enumerated types allow abstract FSM models

Enumerated types allow modeling at a higher level of abstraction, and yet still represent accurate, synthesizable, hardware behavior. Example 8-1, which follows, models a simple finite state machine (FSM), using a typical three-procedural block modeling style: one procedural block for incrementing the state machine, one procedural block to determine the next state, and one procedural block to set the state machine output values. The example illustrates a simple traffic light controller. The three possible states are represented as enumerated type variables for the current state and the next state of the state machine.

By using enumerated types, the only possible values of the `State` and `Next` variables are the ones listed in their enumerated type lists. The **`unique`** modifier to the **`case`** statements in the state machine logic helps confirm that the case statements cover all possible values of the `State` and `Next` variables (**`unique case`** statements are discussed in more detail in section 7.9.1 on page 196).

Example 8-1: A finite state machine modeled with enumerated types (poor style)

```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {RED, GREEN, YELLOW} State, Next;  // using enum defaults

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;
```

```
  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (State)
      RED:    if (sensor)              Next = GREEN;
      GREEN:  if (green_downcnt == 0)  Next = YELLOW;
      YELLOW: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state

  always_comb begin: set_outputs
    {green_light, yellow_light, red_light} = 3'b000;
    unique case (State)
      RED:    red_light    = 1'b1;
      GREEN:  green_light  = 1'b1;
      YELLOW: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

Example 8-1, while functionally correct, might not be a good usage of enumerated types for representing hardware. The example uses the default enum base type of **int**, and the default values for each enumerated value label (0, 1 and 2, respectively). These defaults might not accurately reflect hardware behavior in simulation. The **int** type is a 32-bit 2-state type. The actual hardware for the example above, which has only three states, only needs a 2- or 3-bit vector, depending on how the three states are encoded. The gate-level model of the actual hardware implementation will have 4-state semantics.

The default initial value of 2-state types in simulation can hide design problems. This topic is discussed in more detail later in this chapter, in section 8.2 on page 219. The default values of the enumerated labels can also lead to mismatches in the RTL simulation versus the gate-level implementation of the design. Since the values for the enumerated labels were not explicitly specified, synthesis compilers might optimize the gate-level implementation to different values for each state. This makes it more difficult to compare the pre- and post-synthesis model functionality, or to specify assertions that work with both the pre- and post-synthesis models.

8.1.1 Representing state encoding with enumerated types

enumerated types can have an explicit base type

SystemVerilog also allows the base type of an enumerated variable to be defined. This allows a 4-state type, such as **`logic`**, to be used as a base type, which can more accurately represent hardware behavior in RTL simulations.

enumerated type labels can have explicit values

SystemVerilog's enumerated types also allow modeling at a more hardware-like level of abstraction, so that specific state machine architectures can be represented. The logic value of each label in an enumerated type list can be specified. This allows explicitly representing one-hot, one-cold, Gray code, or any other type of state sequence encoding desired.

Example 8-2 modifies the preceding example to explicitly represent one-hot encoding in the state sequencing. The only change between example 8-1 and example 8-2 is the definition of the enumerated type. The rest of the state machine logic remains at an abstract level, using the labels of the enumerated values.

Example 8-2: Specifying one-hot encoding with enumerated types

```
module traffic_light (output logic green_light,
                                  yellow_light,
                                  red_light,
                      input       sensor,
                      input [15:0] green_downcnt,
                                  yellow_downcnt,
                      input       clock, resetN);

  enum logic [2:0] {RED    = 3'b001, // explicit enum definition
                    GREEN  = 3'b010,
                    YELLOW = 3'b100} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (State)
      RED:    if (sensor)              Next = GREEN;
      GREEN:  if (green_downcnt  == 0) Next = YELLOW;
      YELLOW: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state
```

```
  always_comb begin: set_outputs
    {green_light, yellow_light, red_light} = 3'b000;
    unique case (State)
      RED:    red_light    = 1'b1;
      GREEN:  green_light  = 1'b1;
      YELLOW: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

In this example, the enumerated label values that represent the state sequencing are explicitly specified in the RTL model. Synthesis compilers will retain these values in the gate-level implementation. This helps in comparing pre- and post-synthesis model functionality. It also makes in easier to specify verification assertions that work with both the pre- and post-synthesis models. (Synthesis compiler may provide a way to override the explicit enumeration label values, in order to optimize the gate-level implementation; This type of optimization cancels many of the benefits of specifying explicit enumeration values).

Another advantage illustrated in the example above is that the base type of the enumerated `State` and `Next` variables is a 4-state **`logic`** data type. The default initial value of 4-state types is X instead of 0. Should the design not implement reset correctly, it will be obvious in the RTL simulation that there is a design problem. This topic is discussed in more detail later in this chapter, in section 8.2 on page 219.

8.1.2 Reversed case statements with enumerated types

The typical use of a **`case`** statement is to specify a variable as the case expression, and then list explicit values to be matched as the list of case selection items. This is the modeling style shown in the previous two examples.

one-hot state machines can use reversed case statements

Another style for modeling one-hot state machines is the *reversed case statement*. In this style, the case expression and the case selection items are reversed. The case expression is specified as the literal value to be matched, which, for one-hot state machines, is a 1-bit value of 1. The case selection items are each bit of the state variable. In some synthesis compilers, using the reversed case style for one-hot state machines might yield more optimized synthesis results than the standard style of case statements.

Example 8-3 illustrates using a reversed case statement style. In this example, a second enumerated type variable is declared that represents the index number for each bit of the one-hot `State` register. The name `R_BIT`, for example, has a value of 0, which corresponds to bit 0 of the `State` variable (the bit that represents the `RED` state).

Example 8-3: One-hot encoding with reversed case statement style

```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {R_BIT = 0,  // index of RED state in State register
        G_BIT = 1,  // index of GREEN state in State register
        Y_BIT = 2} state_bit;

  // shift a 1 to the bit that represents each state
  enum logic [2:0] {RED    = 3'b001<<R_BIT,
                    GREEN  = 3'b001<<G_BIT,
                    YELLOW = 3'b001<<Y_BIT} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: if (sensor)              Next = GREEN;
      State[G_BIT]: if (green_downcnt  == 0) Next = YELLOW;
      State[Y_BIT]: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state


  always_comb begin: set_outputs
    {red_light, green_light, yellow_light} = 3'b000;
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: red_light    = 1'b1;
```

```
      State[G_BIT]: green_light  = 1'b1;
      State[Y_BIT]: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

a clever coding trick for using enumerated types with 1-hot FSM models

In the example above, the enumerated variable `state_bit` specifies which bit of the state sequencer represents each state (the 1-hot bit). The value for each state label is calculated by shifting a 3-bit value of `001` (binary) to the bit position that is "hot" for that state. A value of `001` shifted 0 times (the value of `R_BIT`) is `001` (binary). A `001` shifted 1 time (the value of `G_BIT`) is `010` (binary), and shifted 2 times (the value of `Y_BIT`) is `100` (binary).

The same enumerated `state_bit` labels, `R_BIT`, `G_BIT` and `Y_BIT`, are used in the functional code to test which bit of `State` is "hot". Thus, the definitions of the enumerated labels for `State` and the bit-selects of the `State` variable are linked together by the definition of `state_bit`. Using this seemingly complex scheme to specify the 1-hot state values serves two important purposes:

- There is no possibility of a coding error that defines different 1-hot bit positions in the two enumerated type definitions.
- Should the design specification change the 1-hot definitions, only the enumerated type specifying the bit positions has to change. The enumerated type defining the state names will automatically reflect the change.

This clever coding trick of using the bit-shift operator to specify the enumerated values of the state variables was shared by Cliff Cummings of Sunburst Design. Additional FSM coding tricks can be found at Cliff's web site, www.sunburst-design.com.

8.1.3 Enumerated types and unique case statements

unique case reduces the ambiguities of case statements

The use of the **`unique`** modifier to the case statement in the preceding example is important. Since a one-hot state machine only has one bit of the state register set at a time, only one of the case selection items will match the literal value of 1 in the case expression. The **`unique`** modifier to the **`case`** statement specifies three things.

First, **`unique case`** specifies that all case selection items can be evaluated in parallel, without priority encoding. Software tools such as synthesis compilers can optimize the decoding logic of the case selection items to create smaller, more efficient implementations. This aspect of **`unique case`** is the same as synthesis `parallel_case` pragma.

Second, **`unique case`** specifies that there should be no overlap in the case selection items. During the run-time execution of tools such as simulation, if the value of the case expression satisfies two or more case selection items, a run-time warning will occur. This semantic check can help trap design errors early in the design process. The synthesis `parallel_case` pragma does not provide this important semantic check.

Third, **`unique case`** specifies that all values of the case expression that occur during simulation must be covered by the case selection items. With **`unique case`**, if a case expression value occurs that does not cause a branch of the case statement to be executed, a run-time warning will occur. This semantic check can also help trap design errors much earlier in the design cycle. This is similar to the `full_case` pragma for synthesis, but the synthesis pragma does not require that other tools perform any checking.

8.1.4 Specifying unused state values

Verilog types can have unused values

As an enumerated type, the `State` variable has a restricted set of values. The `State` variable is a multi-bit vector, which, at the gate-level, can reflect logic values not defined in the enumerated list. A finite state machine with three states requires a 3-bit state register for one-hot encoding. This 3-bit register can contain 8 possible values. The hardware registers represented can hold all possible values, not just the values listed in the enumerated list. The base type of the enumerated type can also represent all 8 of these values.

There are two common modeling styles to indicate that some values of the case expression are not used: specify a default case selection with a logic X assignment, or specify a special synthesis `full_case` pragma. These two styles are discussed in more detail in the following paragraphs.

Using X as a default assignment

a default assignment of X can cover unused conditions

The combination of enumerated types and unique case can eliminate the need for a common Verilog coding style with case statements. This Verilog style is to specify a **`default`** statement to cover all unused values of the case expression. This **`default`** statement assigns a logic X to the variables representing the outputs of the case statement. In the FSM example from above, the case expression is the current state variable, `State`, and the output of the case statement is the next state variable, `Next`.

```
// Verilog style case statement with X default
reg [2:0] State, Next;  // 3-bit variables

case (State)
  3'b001:  Next = 3'b010;
  3'b010:  Next = 3'b100;
  3'b100:  Next = 3'b001;
  default: Next = 3'bXXX;
endcase
```

Synthesis compilers recognize the default assignment of logic X as an indication that any case expression value that falls into the default case is an unused value. This can enable the synthesis compiler to perform additional optimizations and improve the synthesis quality of results.

enumerated types cannot be directly assigned an X value

When enumerated types are used, an assignment of logic X is not a legal assignment. An enumerated type can only be assigned values from its enumerated list. If an X assignment is desired, the base type of the enumerated type must be defined a 4-state type, such as **`logic`**, and an enumerated label must be defined with an explicit value of X. For example:

```
// case statement with enumerated X default
enum logic [2:0] {RED       = 3'b001,
                  GREEN     = 3'b010,
                  YELLOW    = 3'b100,
                  BAD_STATE = 3'bxxx,
                 } State, Next;

case (State)
  RED:     Next = GREEN;
  GREEN:   Next = YELLOW;
  YELLOW:  Next = RED;
  default: Next = BAD_STATE;
endcase
```

enumerated types can eliminate unused conditions

With SystemVerilog, the `BAD_STATE` enumerated value and the **`default`** case item are not needed. The combination of enumerated types and **`unique case`** statements eliminates the need for using a logic X assignment to show that not all case expression values are used. The enumerated type limits the values of its variables to just the values listed in the enumerated value set. These are the only values that need to be listed in the case statement. The defined set of values that an enumerated type can hold, along with the additional **`unique case`** semantic checking (discussed in section 8.1.3 on page 213) help ensure that pre-synthesis RTL model and the post-synthesis gate-level model are the same for both simulation and equivalence checking.

As discussed in the preceding paragraphs, using **`unique case`** combines the functionality of both the synthesis `parallel_case` and `full_case` pragmas. The **`unique case`** also provides semantic checks to ensure that all values of an enumerated type used as a case expression truly meet the requirements to be implemented as parallel, combinational logic. Any unintended or unexpected case expression values will be trapped as run-time warnings by a **`unique case`** statement.

8.1.5 Assigning state values to enumerated type variables

enumerated types can only be assigned values in their type set

Enumerated types are more strongly typed than other Verilog and SystemVerilog variables. Enumerated types can only be assigned a value that is a member of the type list of that enumerated type. An enumerated type can be assigned the value of another enumerated type, but only if both enumerated types are from the same definition. Section 4.2.6 on page 86 of Chapter 4, discusses the assignment rules for enumerated types in more details.

A common Verilog style when using one-hot state sequences is to first clear the next state variable, and then set just the one bit of next state variable that indicates what the next state will be. This style will not work with enumerated types. Consider the following code snippet:

Example 8-4: Code snippet with illegal assignments to enumerated types

```
enum {R_BIT = 0,  // index of RED state in State register
      G_BIT = 1,  // index of GREEN state in State register
      Y_BIT = 2} state_bit;
```

```
// shift a 1 to the bit that represents each state
enum logic [2:0] {RED    = 3'b001<<R_BIT,
                  GREEN  = 3'b001<<G_BIT,
                  YELLOW = 3'b001<<Y_BIT} State, Next;

...

always_comb begin: set_next_state
  Next = 3’b000;  // clear Next - ERROR: ILLEGAL ASSIGNMENT
  unique case (1’b1)   // reversed case statement
  // WARNING: FOLLOWING ASSIGNMENTS ARE POTENTIAL DESIGN ERRORS
    State[R_BIT]: if (sensor == 1)       Next[G_BIT] = 1’b1;
    State[G_BIT]: if (green_downcnt==0)  Next[Y_BIT] = 1’b1;
    State[Y_BIT]: if (yellow_downcnt==0) Next[R_BIT] = 1’b1;
  endcase
end: set_next_state
...
```

There are two problems with the code snippet above. First, a default assignment of all zeros is made to the `Next` variable. This is an illegal assignment. An enumerated type must be assigned labels from its enumerated list, not literal values.

Second, within the **`case`** statements, assignments are made to individual bits of the `Next` variable. Assigning to a discrete bit of an enumerated type may be allowed by compilers, but it is not a good style when using enumerated types. By assigning to a bit of an enumerated type variable, an illegal value could be created that is not in the enumerated type list. This would result in design errors that could be difficult to debug.

TIP

Assign an enumerated type variable a label from its enumerated list, instead of a value.

Assignments to enumerated type variables should be from the list of labels for that type. Assigning to bit-selects or part-selects of an enumerated type should be avoided. When assignments to bits of a variable are required, the variable should be declared as standard type, such as **`bit`** or **`logic`**, instead of an enumerated type. Example 8-3 on page 212 shows the correct way to model a Verilog “reverse case statement” when using enumerated types.

8.1.6 Performing operations on enumerated type variables

Enumerated types differ from most other Verilog types in that they are strongly typed variables. For example, it is illegal to directly assign a literal value to an enumerated type. When an operation is performed on an enumerated type variable, the value of the variable is the type of the base type of the enumerated type. By default, this is an **int** type, but can be explicitly declared as other types.

The following example will result in an error. The operation `State + 1` will result in an **int** value. Directly assigning this **int** value to the `Next` variable, which is an enumerated type variable, is illegal.

```
enum {RED, GREEN, YELLOW} State, Next;

Next = State + 1;  // ILLEGAL ASSIGNMENT
```

A value of a different type can be assigned to an enumerated type using type casting. SystemVerilog provides both a static cast operator and a dynamic cast system function.

```
typedef enum {RED, GREEN, YELLOW} states_t;
states_t State, Next;

Next = states_t'(State + 1); // static cast

$cast(Next, State + 1);      // dynamic cast
```

A static cast operation coerces an expression to a new type without performing any checking on whether the value coerced is a valid value for the new type. If, for example, the current value of `State` were `YELLOW`, then `State + 1` would result in an out-of-bounds value. Using static casting, this out-of-bounds value would not be trapped. The SystemVerilog standard allows software tools to handle out-of-bounds assignments in a nondeterministic manner. This means the new value of the `Next` variable in the preceding static cast assignment could, and likely will, have different values in different software tools.

A dynamic cast performs run-time checking on the value being cast. If the value is out-of-range, then an error message is generated, and the target variable is not changed. By using dynamic casting, inadvertent design errors can be trapped, and the design corrected to prevent the out-of-bounds values.

SystemVerilog also provides a number of special enumerated type methods for performing basic operations on enumerated type variables. These methods allow incrementing or decrementing a value within the list of legal values for the enumerated type.

```
Next = State.next;  // enumerated method
```

Section 4.2.8 on page 89 discusses the various enumerated methods in more detail.

Each of these styles of assigning the result of an operation to an enumerated type has advantages. Using the enumerated type methods ensures the assigned value will always be within the set of values in the enumerated type list. The dynamic cast operator provides run-time errors for out-of-range values. Static casting does not perform any error checking, but might yield better simulation run-time performance compared to using methods or dynamic casting. With static casting, however, the burden is on the designer to ensure that an out-of-bound value will never occur.

8.2 Using 2-state types in FSM models

8.2.1 Resetting FSMs with 2-state and enumerated types

At the beginning of simulation, 4-state types are logic X. Within a model such as a finite state machine, a logic X on 4-state variables can serve as an indication that the model has not been reset, or that the reset logic has not been properly modeled.

2-state types begin simulation with a default value of logic 0 instead of an X. Since the typical action of reset is to set most variables to 0, it can appear that the model has been reset, even if there is faulty reset logic.

Enumerated types begin simulation with a default value of the base type of the enumerated type. If the state variables are defined using the default base type and label values, and if reset also sets enumerated values to the first item in the list, then a similar situation can occur as with 2-state variables. The default base type is **`int`**, which has an un-initialized value of 0 at the beginning of simulation. The default value for the first label in an enumerated list is 0, which is the same as the un-initialized value of the 2-state base type. The design can appear to have been reset, even if reset is never asserted, or if the design reset logic has errors.

The following example will lock-up in the `WAITE` state. This is because both the `State` and `Next` variables begin simulation with a value of 0, which is also the value of the first value in their enumerated lists, `WAITE`. At every positive edge of clock, `State` is assigned the value it already has, and therefore no transition occurs. Since there is no transition, the **`always`** `@(State)` procedural block that decodes `Next` is not triggered, and therefore `Next` is not changed from its initial value of `WAITE`.

```
enum {WAITE, LOAD, STORE} State, Next;

always @(posedge clock, negedge resetN)
  if (!resetN) State <= WAITE;
  else State <= Next;

always @(State)
  case (State)
    WAITE:  Next = LOAD;
    LOAD:   Next = STORE;
    STORE:  Next = WAITE;
  endcase
```

Applying reset does not fix this state lock-up problem. Reset changes the `State` variable to `WAITE`, which is the same value that `State` begins simulation with. Therefore there is no change to the `State` variable and the next state decode logic is not triggered. `Next` continues to keep its initial value, which is also `WAITE`.

This lock-up at the start of simulation can be fixed in two ways. The first way is to explicitly declare the enumerated variable with a 4-state base type, such as **`logic`**. Simulation will then begin with `State` and `Next` having an un-initialized value of X. This is a clear indication that these variables have been reset. It also more accurately reflects the nature of hardware, where flip-flops can power up in an indeterminate state. In RTL simulation, when reset is applied, the `State` variable will transition from X to its reset value of `WAITE`. This transition will trigger the logic that decodes `Next`, setting `Next` to its appropriate value of `LOAD`.

The second fix for the FSM lock up, when using an enumerated type with its default base type and label values, is to replace **`always`** `@(state)` with the SystemVerilog **`always_comb`** procedural block. An **always_comb** procedural block automatically executes its statements once at simulation time zero, even if there were no transitions on its inferred sensitivity list. By executing the decode logic at time zero, the initial value of `State` will be decoded, and the `Next` variable set accordingly. This fixes the start of simulation lock-up problem.

Combining **unique case** along with the use of a 4-state base type for enumerated types also has an advantage. If `State` in the code snippet above had not been reset, it would be a logic X, which will not match any of the case items to which `State` is compared. The **unique case** `(State)` statement will issue a run-time warning whenever no case items match the case expression. (A warning would also be issued if the case expression matches more than one case item.)

Examples 8-2 on page 210 and 8-3 on page 212 illustrate using 4-state enumerated types coupled with **always_comb** and **unique case**. This combination of SystemVerilog constructs not only simplifies writing RTL code, it can trap design problems that in Verilog could have been difficult to detect and debug.

8.3 Summary

This chapter has presented suggestions on modeling techniques when representing hardware behavior at a more abstract level. SystemVerilog provides several enhancements that enable accurately modeling designs that simulate and synthesize correctly. These enhancements help to ensure consistent model behavior across all software tools, including lint checkers, simulators, synthesis compilers, formal verifiers, and equivalence checkers.

Several ideas were presented in this section on how to properly model finite state machines using these new abstract modeling constructs such as: 2-state types, enumerated types, **always_comb** procedural blocks, and **unique** case statements.

Chapter 9

SystemVerilog Design Hierarchy

This chapter presents the many enhancements to Verilog that SystemVerilog adds for representing and working with design hierarchy. The topics that are discussed include:

- Module prototypes
- Nested modules
- Simplified netlists of module instances
- Netlist aliasing
- Passing values through module ports
- Port connections by reference
- Enhanced port declarations
- Parameterized types and polymorphism
- Variable declarations in blocks

9.1 Module prototypes

module instances need more info to be compiled

A module instance in Verilog is a straight-forward and simple method of creating design hierarchy. For tool compilers, however, it is difficult to compile a module instance, because the definition of the module and its ports is in a different place than the module instance. To complete the compilation process of a module instance, the compiler must also at least parse the module definition in order to determine the number of ports, the size and type of the ports, and possibly the order of the ports in the module definition.

extern module declarations

SystemVerilog simplifies the compilation process by allowing users to specify a prototype of the module being instantiated. The prototype is defined using an **extern** keyword, followed by the declaration of a module and its ports. Either the Verilog-1995 or the Verilog-2001 style of module declarations can be used for the prototype. The Verilog-1995 module declaration style is limited to only defining the number of ports and port order of a module. The Verilog-2001 module declaration style defines the number of ports, the port order, the port vector sizes and the port types. Verilog-2001 style module declarations can also include a parameter list, which allows parameterized ports. Examples of Verilog-1995 and Verilog-2001 prototype declarations are:

```
// prototype using Verilog-1995 style
extern module counter (cnt, d, clock, resetN);


// prototype using Verilog-2001 style
extern module counter #(parameter N = 15)
                       (output logic [N:0] cnt,
                        input  wire  [N:0] d,
                        input  wire  clock,
                                     load,
                                     resetN);
```

Prototypes of a module definition also serve to document a design. Large designs can be spread across dozens of source files. When one file contains an instance of another module, some other file needs to be examined to see the definition of the instantiated module. A prototype of the module definition can be listed in the same file in which the module is instantiated.

Extern module declaration visibility

prototypes are local to the containing scope

The **extern module** declaration can be made in any module, at any level of the design hierarchy. The declaration is only visible within the scope in which it is defined. An external module declaration that is made outside of any module or interface boundary will be in the $unit compilation-unit declaration space. Any other module that shares the $unit space, anywhere in the design hierarchy, can instantiate the globally visible module.

In Verilog, modules can be instantiated before they are defined. The prototype for a module is an alternative to the actual definition in a compilation unit, and therefore uses a similar checking system. It is not necessary for the **extern** declaration to be encountered prior to an instance of the module.

9.1.1 Prototype and actual definition

prototype and actual definition must match

SystemVerilog requires that the port list of an **extern module** declaration exactly match the actual module definition, including the order of the ports and the port sizes. It is a fatal error if there is any mismatch in the port lists of the two definitions.

9.1.2 Avoiding port declaration redundancy

module definition can use . shortcut*

SystemVerilog provides a convenient shortcut to reduce source code redundancy. If an **extern module** declaration exists for a module, it is not necessary to repeat the port declarations as part of the module definition. Instead, the actual module definition can simply place the **.*** characters in the port list. Software tools will automatically replace the **.*** with the ports defined in the **extern module** prototype. This saves having to define the same port list twice, once in the external module prototype, and again in the actual module definition. For example:

```
extern module counter #(parameter N = 15)
                       (output logic [N:0] cnt,
                        input  wire  [N:0] d,
                        input  wire  clock,
                                     load,
                                     resetN);

module counter ( .* );
  always @(posedge clock, negedge resetN) begin
```

```
    if (!resetN)   cnt <= 0;
    else if (load) cnt <= d;
    else           cnt <= cnt + 1;
  end
endmodule
```

In this example, using `.*` for the counter module definition infers both the parameter list and the port list from the **`extern`** declaration of the counter.

9.2 Named ending statements

9.2.1 Named module ends

A module is defined between the pair of keywords **`module`** and **`endmodule`**. With the addition of nested modules, a parent module can contain multiple endmodule declarations. This can make it difficult to read a large block of code, and determine visually which endmodule is paired with which module declaration.

SystemVerilog allows a name to be specified with the **`endmodule`** keyword, using the form:

```
endmodule : <module_name>
```

The name specified with **`endmodule`** must be the same as the name of the module with which it is paired.

Specifying a name with **`endmodule`** serves to make SystemVerilog code self-documenting and easier to maintain. Several of the larger SystemVerilog code examples in this book illustrate using named module ends.

9.2.2 Named code block ends

SystemVerilog also allows an ending name to be specified with other named blocks of code. These include the keyword pairs: **`package`**...**`endpackage`**, **`interface`**...**`endinterface`**, **`task`**...**`endtask`**, **`function`**...**`endfunction`**, and **`begin`**...**`end`**, as well as other named coding blocks primarily used in testbench code.

Section 7.7 on page 192 discusses the use of ending names with **`begin`**...**`end`** pairs in more detail.

9.3 Nested (local) module declarations

module names are global

In Verilog, all module names, user-defined primitive (UDP) names, and system task and system function names (declared using the Verilog PLI) are placed in a global name space. The names of these objects can be referenced anywhere in the design hierarchy. This global access to module names provides a simple yet powerful mechanism for defining the design hierarchy. Any module can instantiate any other module, without dependencies on the order in which files are compiled.

access to module names is not restricted

However, Verilog's global access to all elaborated module names makes it impossible to limit access to specific modules. If a complex *Intellectual Property* (IP) model, for example, contains its own hierarchy tree, the module names within the IP model will become globally accessible, allowing any other part of a design to directly instantiate the submodules of the IP model.

global names can cause conflicts

Verilog's global access to all elaborated module names can also result in naming conflicts. For example, if both the user's design and an IP model contained modules named `FSM`, there would be a name collision in the global name scope. If multiple IP models are used in the design, it is possible that a module name conflict will occur between two or more IP models. A name conflict will require that changes be made to either the IP model source code or the design code.

Most software tools provide proprietary solutions for name scope conflict. These solutions, however, usually require some level of user input over the compilation and/or elaboration process. Verilog-2001 added a configuration construct to Verilog, which provide a standard solution for allowing the same module name to be used multiple times, without a conflict in the global module definition name scope. Configurations, however, are verbose, and do not address the problems of limiting where a module can be instantiated.

Nested (local) modules

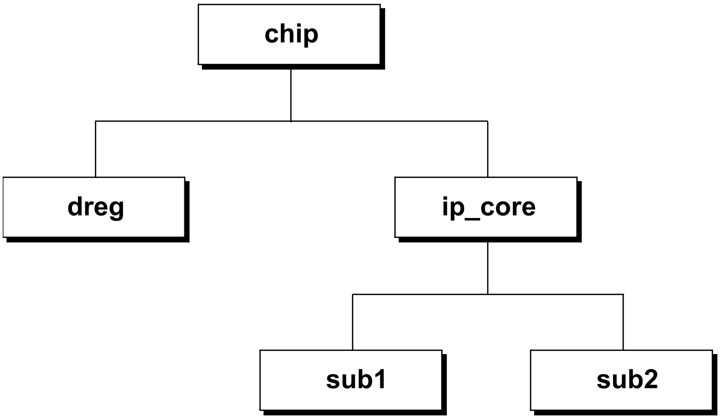
modules declared within modules

SystemVerilog provides a simple and elegant solution for limiting where module names can be instantiated, and avoiding potential conflicts with other modules of the same name. The solution is to allow a module definition to be nested within another module definition. Nested modules are not visible outside of the hierarchy scope in which they are declared. For example:

Example 9-1: Nested module declarations

```
module chip (input wire clock);      // top level of design
  dreg i1 (clock);
  ip_core i2 (clock);
endmodule: chip

module dreg (input wire clock);      // global module definition
  ...
endmodule: register

module ip_core (input wire clock);   // global module definition
  sub1 u1 (...);
  sub2 u2 (...);
  module sub1(...);                  // nested module definition
    ...
  endmodule: sub1

  module sub2(...);                  // nested module definition
    ...
  endmodule: sub2

endmodule: ip_core
```

The instantiated hierarchy tree for example 9-1 is:



Nested module definitions can be in separate files

a common style is to place each module in a separate file

A very common modeling style with Verilog is to place the source code for each module definition in a separate source file. Typically, the file name is the same as the module name. This style, while not a requirement of the Verilog language, is often used, because it helps to develop and maintain the source code of large designs. If several modules are contained in a single file, the source code within that file can become unwieldy and difficult to work with. Keeping each module in a separate file also facilitates the use of revision control software as part of the design process. Revision control tools allow specific users to check out specific files for modification, and can track the revision history of that file. If many modules are contained in the same file, revision control loses some of its effectiveness.

TIP

Use `` `include `` to avoid the convoluted code of multiple modules in the same source code file.

Nesting module definitions can lead to the source code file for the top-level module containing multiple module definitions. In addition, a nested module can become difficult to maintain, or to reuse in other designs, if the source code of the nested module is buried within the top-level module.

Using Verilog's **`include** compiler directive with nested modules can eliminate these potential drawbacks. The definition of each nested module can be placed in a separate file, where it is easy to maintain and to reuse. The top-level module can then include the definitions of the nested module, using `` `include`` directives. This helps make the top-level module more compact and easier to read.

For example:

```
module ip_core (input logic clock);
  ...
  `include sub1.v  // sub1 is a nested module
  `include sub2.v  // sub2 is a nested module
  ...
endmodule

module sub1(...);  // stored in file sub1.v
  ...
endmodule

module sub2(...);  // stored in file sub2.v
  ...
endmodule
```

9.3.1 Nested module name visibility

nested module names are not global

The names of nested modules are not placed in the global module definition name scope with other module names. Nested module names exist in the name scope of the parent module. This means that a nested module can have the same name as a module defined elsewhere in a design, without any conflict in the global module definition name scope.

Because the name of a nested module is only visible locally in the parent module, the nested module can only be instantiated by the parent module, or the hierarchy tree below the nested module. A nested module cannot be instantiated anywhere else in a design hierarchy. In example 9-1, above, the modules `chip`, `dreg`, and `ip_core` are in the global name scope. These modules can be instantiated by any other module, anywhere in the design hierarchy. Modules `sub1` and `sub2` are nested within the definition of the `ip_core` module. These module names are local names within `ip_core`, and can only be instantiated in `ip_core`, or by the modules that are instantiated in `ip_core`.

nested module hierarchy paths

Nested modules have a hierarchical scope name, the same as with any module instance. Variables, nets, and other declarations within a nested module can be referenced hierarchically for verification purposes, just as with declarations in any other module in the design.

Nested modules can instantiate other modules

nested modules can instantiate other modules

A nested module can instantiate other modules. The definitions of these modules can be in three name scopes: the global module definition name scope, the parent of the nested module, or within the nested module (as another nested module definition).

9.3.2 Instantiating nested modules

nested modules are instantiated the same as regular modules

A nested module is instantiated in the same way as a regular module. Nested modules can be explicitly instantiated any number of times within its parent. It can also be instantiated anywhere in the hierarchy tree below the parent module. The only difference between an instance of a nested module and a regular module is that the nested module can only be instantiated in the hierarchy tree at or below its parent module, whereas a regular module can be instantiated anywhere in the design hierarchy.

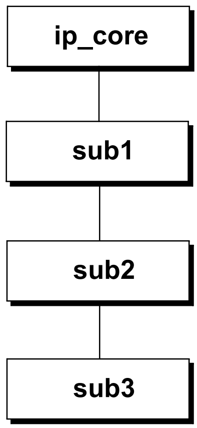
In the following example, module `ip_core` has three nested module definitions: `sub1`, `sub2`, and `sub3`. Even though the nested module definitions are local to `ip_core`, hierarchically, these nested modules are not all direct children of `ip_core`. In this example, `ip_core` instantiates module `sub1`, `sub1` instantiates `sub2`, and `sub2` instantiates `sub3`.

Example 9-2: Hierarchy trees with nested modules

```
module ip_core (input clock);

  sub1 u1 (...);      // instance of nested module sub1

  module sub1 (...);  // nested module definition
    sub2 u2 ();
    ...
  endmodule: sub1
```

```
  module sub2;        // nested module definition
    // sub2 does not have ports, but will look in its source
    // code parent module (ip_core) for identifiers
    sub3 u3 (...);
  endmodule: sub2

  module sub3 (...);  // nested module definition
    ...
  endmodule: sub3

endmodule: ip_core
```

The instantiated hierarchy tree for example 9-2 is:



9.3.3 Nested module name search rules

nested modules have a local scope

A nested module has its own name scope, just as with regular modules. Nested modules can be defined either with ports or without ports. The port names of a nested module become local names within the nested module. Any nets, variables, tasks, functions or other declarations within a nested module are local to that module.

nested modules can reference names in their parent module

Nested modules have a different name search rule than regular modules. Semantically, a nested module is similar to a Verilog task, in that the nested module has visibility to the signals within its parent. As with a task, if a name is referenced that is not in the local scope of the nested module, that name will be searched for in the parent module. If a name is referenced that is not local to the nested module and also does not exist in the parent module, the compilation-unit scope will be searched. This allows a nested module to reference variables, constants, tasks, functions, and user-defined types that are defined externally, in the compilation-unit.

modules that are not nested search upward in the instantiation tree

It is important to note that the upward searching of a nested module is different than the upward searching rules of modules that are not nested. A module that is not nested is in the global module definition scope. There is no source code parent. When a module that is defined in the global module definition scope references an identifier (such as a variable name or function name) that is not declared within the module, the name search path uses the instantiation hierarchy tree, including the $unit compilation-unit scope.

nested modules search upward in the source code

A nested module definition, on the other hand, does have a source-code parent. When an identifier is referenced within a nested module that is not defined within the nested module, the search path is to look in the parent module where the nested module is defined, rather than where the module is instantiated.

9.4 Simplified netlists of module instances

netlists connect module instances together

A netlist is a list of module instances, with nets connecting the ports of the instances together. Netlists are used at many levels of design, from connecting major blocks together at a high-level of abstraction, to connecting together discrete components, such as ASIC cells or gates at a detailed implementation level. Netlists are often generated from software tools, such as synthesis compilers; but netlists are also often defined by hand, such as when connecting design blocks together. Even at the block level, with top-level models, netlists can often be quite large, with a high potential for connection errors that can be difficult to debug.

The Verilog language provides two syntax styles for connecting module instances together: *ordered port connections* and *named port connections*.

using port order to connect module instances

Ordered port connections connect a net or variable to a module instance, using the position of the ports of each module definition. For example, a net called `data_bus` might connect the fifth port of one module instance to the fourteenth port of another module instance. With ordered port connections, the names of each port do not matter. It is the port position that is critical. This requires knowing the exact order of the ports for each module instance being connected.

An example instance of a D-type flip-flop module is shown below.

```
dff d1 (out, /*not used*/, in, clock, reset);
```

The requirement to know the exact position of each port of the module being instantiated is a disadvantage. Unintentional design errors can easily occur when using the port order connection syntax. Modules in complex designs often have dozens of ports. Should a net be connected to the wrong port position, the error will not be obvious from just looking at the netlist. Another disadvantage is that ordered port connections do not clearly document the design intent. It is difficult to look at a module instance that is connected by port order and determine to which port a net is intended to be connected. Because of these disadvantages, many companies discourage the use of ordered port connections in their company style guidelines.

using port names to connect module instances

The second style for connecting modules together in Verilog is to specify the name of each port explicitly, along with the name of the signal that is connected to that port. The basic syntax for each port connection is:

```
.<port_name>(<net_or_variable_name>)
```

An example instance of a D-type flip-flop module using named port connections is shown below. Since the flip-flop ports are explicitly named, it is easy to tell to what port a signal is connected, even without seeing the actual flip-flop module definition.

```
dff d1 (.q(out), .qb(/*not used*/),
        .d(in), .clk(clock), .rst(reset) );
```

Using this named port connection style, it is not necessary to maintain the order of the ports for each module instance. By using named port connections, the potential for inadvertent design errors is reduced, since each port is explicitly connected to a specific net.

Example 9-3 shows a netlist for a small microprocessor, which represents a simplified model of a MicroChip PIC 8-bit processor.

Though it is a small design with just 6 module instances in the netlist, the model illustrates using named port connections. Examples 9-4 and 9-5, which follow, show how SystemVerilog simplifies Verilog netlists.

Example 9-3: Simple netlist using Verilog's named port connections

```
module miniPIC (
    inout  wire [7:0] port_a_pins,
    inout  wire [7:0] port_b_pins,
    inout  wire [7:0] port_c_pins,
    input  wire       clk,
    input  wire       resetN
  );

  wire [11:0] instruct_reg, program_data;
  wire [10:0] program_counter, program_address;
  wire [ 7:0] tmr0_reg, status_reg, fsr_reg, w_reg, option_reg,
              reg_file_out, port_a, port_b, port_c, trisa,
              trisb, trisc, data_bus, alu_a, alu_b;
  wire [ 6:0] reg_file_addr;
  wire [ 3:0] alu_opcode;
  wire [ 1:0] alu_a_sel, alu_b_sel;
  wire        reg_file_sel, special_reg_sel, reg_file_enable,
              w_reg_enable, zero_enable, carry_enable, skip,
              isoption, istris, polarity, carry, zero;

  pc_stack pcs ( // module instance with named port connections
    .program_counter(program_counter),
    .program_address(program_address),
    .clk(clk),
    .resetN(resetN),
    .instruct_reg(instruct_reg),
    .data_bus(data_bus),
    .status_reg(status_reg)
  );

  prom prom (
    .dout(program_data),
    .clk(clk),
    .address(program_address)
  );

  instruction_decode decoder (
    .alu_opcode(alu_opcode),
    .alu_a_sel(alu_a_sel),
    .alu_b_sel(alu_b_sel),
    .w_reg_enable(w_reg_enable),
```

```
    .reg_file_sel(reg_file_sel),
    .zero_enable(zero_enable),
    .carry_enable(carry_enable),
    .polarity(polarity),
    .option(isoption),
    .tris(istris),
    .instruct_reg(instruct_reg)
  );

  register_files regs (
    .dout(reg_file_out),
    .tmr0_reg(tmr0_reg),
    .status_reg(status_reg),
    .fsr_reg(fsr_reg),
    .port_a(port_a),
    .port_b(port_b),
    .port_c(port_c),
    .trisa(trisa),
    .trisb(trisb),
    .trisc(trisc),
    .option_reg(option_reg),
    .w_reg(w_reg),
    .instruct_reg(instruct_reg),
    .program_data(program_data),
    .port_a_pins(port_a_pins),
    .data_bus(data_bus),
    .address(reg_file_addr),
    .clk(clk),
    .resetN(resetN),
    .skip(skip),
    .reg_file_sel(reg_file_sel),
    .zero_enable(zero_enable),
    .carry_enable(carry_enable),
    .w_reg_enable(w_reg_enable),
    .reg_file_enable(reg_file_enable),
    .zero(zero),
    .carry(carry),
    .special_reg_sel(special_reg_sel),
    .isoption(isoption),
    .istris(istris)
  );

  alu alu (
    .y(data_bus),
    .carry_out(carry),
    .zero_out(zero),
    .a(alu_a),
    .b(alu_b),
```

```
    .opcode(alu_opcode),
    .carry_in(status_reg[0])
  );

  glue_logic glue (
    .port_b_pins(port_b_pins),
    .port_c_pins(port_c_pins),
    .alu_a(alu_a),
    .alu_b(alu_b),
    .expan_out(expan_out),
    .expan_addr(expan_addr),
    .reg_file_addr(reg_file_addr),
    .reg_file_enable(reg_file_enable),
    .special_reg_sel(special_reg_sel),
    .expan_read(expan_read),
    .expan_write(expan_write),
    .skip(skip),
    .instruct_reg(instruct_reg),
    .program_counter(program_counter),
    .port_a(port_a),
    .port_b(port_b),
    .port_c(port_c),
    .data_bus(data_bus),
    .expan_in(expan_in),
    .fsr_reg(fsr_reg),
    .tmr0_reg(tmr0_reg),
    .status_reg(status_reg),
    .w_reg(w_reg),
    .reg_file_out(reg_file_out),
    .alu_a_sel(alu_a_sel),
    .alu_b_sel(alu_b_sel),
    .reg_file_sel(reg_file_sel),
    .polarity(polarity),
    .zero(zero)
  );
endmodule
```

Named port connection advantages

named port connections are a preferred style

An advantage of named port connections is that they reduce the risk of an inadvertent design error because a net was connected to the wrong port. In addition, the named port connections better document the intent of the design. In the example above, it is very obvious which signal is intended to be connected to which port of the flip-flop, without having to go look at the source code of each module. Many companies have internal modeling guidelines that require using the named port connection style in netlists, because of these advantages.

Named port connection disadvantages

named port connections are verbose

The disadvantage of the named port connection style is that it is very verbose. Netlists can contain tens or hundreds of module instances, and each instance can have dozens of ports. Both the name of the port and the name of the net connected to the port must be listed for each and every port connection in the netlist. Port and net names can be up to 1024 characters long in Verilog tools. When long, descriptive port names and net names are used, and there are many ports for each module name, the size and verbosity of a netlist using named port connections can become excessively large and difficult to maintain.

9.4.1 Implicit .name port connections

.name is an abbreviation of named port connections

SystemVerilog provides three enhancements that greatly simplify netlists: `.name` (pronounced "*dot-name*") port connections, `.*` (pronounced "*dot-star*") port connections, and *interfaces*. The `.name` and `.*` styles are discussed in the following subsections, and interfaces are presented in Chapter 10.

.name simplifies connections to module instances

The SystemVerilog `.name` port connection syntax combines the advantages of both the conciseness of ordered port connections with self-documenting code and order independence of named-port connections, eliminating the disadvantages of each of the two Verilog styles. In many Verilog netlists, especially top-level netlists that connect major design blocks together, it is common to use the same name for both the port name and the name of the net connected to the port. For example, the module might have a port called `data`, and the interconnected net is also called `data`.

.name infers a connection of a net and port of the same name

Using Verilog's named port connection style, it is necessary to repeat the name twice in order to connect the net to the port, for example: `.data(data)`. SystemVerilog simplifies the named port connection syntax by allowing just the port name to be specified. When only the port name is given, SystemVerilog infers that a net or variable of the same name will automatically be connected to the port. This means the verbose Verilog style of `.data(data)` can be reduced to simply `.data`.

.name can be combined with named port connections

When the name of a net does not match the port to which it is to be connected, the Verilog named port connection is used to explicitly connect the net to the port. As with the Verilog named port connections, an unconnected port can be left either unspecified, or explicitly named with an empty parentheses set to show that there is no connection.

Example 9-4 lists the simple processor model shown previously in example 9-3, but with SystemVerilog's `.name` port connection style for all nets that are the same name as the port. Compare this example to example 9-3, to see how the `.name` syntax reduces the verbosity of named port connections. Using the `.name` connection style, the netlist is easier to read and to maintain.

Example 9-4: Simple netlist using SystemVerilog's .name port connections

```
module miniPIC (
    inout  wire [7:0] port_a_pins,
    inout  wire [7:0] port_b_pins,
    inout  wire [7:0] port_c_pins,
    input  wire       clk,
    input  wire       resetN
  );

  wire [11:0] instruct_reg, program_data;
  wire [10:0] program_counter, program_address;
  wire [ 7:0] tmr0_reg, status_reg, fsr_reg, w_reg, option_reg,
              reg_file_out, port_a, port_b, port_c, trisa,
              trisb, trisc, data_bus, alu_a, alu_b;
  wire [ 6:0] reg_file_addr;
  wire [ 3:0] alu_opcode;
  wire [ 1:0] alu_a_sel, alu_b_sel;
  wire        reg_file_sel, special_reg_sel, reg_file_enable,
              w_reg_enable, zero_enable, carry_enable, skip,
              isoption, istris, polarity, carry, zero;

  pc_stack pcs (  // module instance with .name port connections
    .program_counter,
    .program_address,
    .clk,
    .resetN,
    .instruct_reg,
    .data_bus,
```

```
    .status_reg
  );

  prom prom (
    .dout(program_data),
    .clk,
    .address(program_address)
  );

  instruction_decode decoder (
    .alu_opcode,
    .alu_a_sel,
    .alu_b_sel,
    .w_reg_enable,
    .reg_file_sel,
    .zero_enable,
    .carry_enable,
    .polarity,
    .option(isoption),
    .tris(istris),
    .instruct_reg
  );

  register_files regs (
    .dout(reg_file_out),
    .tmr0_reg,
    .status_reg,
    .fsr_reg,
    .port_a,
    .port_b,
    .port_c,
    .trisa,
    .trisb,
    .trisc,
    .option_reg,
    .w_reg,
    .instruct_reg,
    .program_data,
    .port_a_pins,
    .data_bus,
    .address(reg_file_addr),
    .clk,
    .resetN,
    .skip,
    .reg_file_sel,
    .zero_enable,
    .carry_enable,
    .w_reg_enable,
```

```
    .reg_file_enable,
    .zero,
    .carry,
    .special_reg_sel,
    .isoption,
    .istris
  );

  alu alu (
    .y(data_bus),
    .carry_out(carry),
    .zero_out(zero),
    .a(alu_a),
    .b(alu_b),
    .opcode(alu_opcode),
    .carry_in(status_reg[0])
  );

  glue_logic glue (
    .port_b_pins,
    .port_c_pins,
    .alu_a,
    .alu_b,
    .reg_file_addr,
    .reg_file_enable,
    .special_reg_sel,
    .skip,
    .instruct_reg,
    .program_counter,
    .port_a,
    .port_b,
    .port_c,
    .data_bus,
    .fsr_reg,
    .tmr0_reg,
    .status_reg,
    .w_reg,
    .reg_file_out,
    .alu_a_sel,
    .alu_b_sel,
    .reg_file_sel,
    .polarity,
    .zero
  );
endmodule
```

.name connection inference rules

In order to infer a connection to a named port, the net or variable must match both the port name and the port vector size. In addition, the types on each side of the port must be compatible. Incompatible types are any port connections that would result in a warning or error if a net or variable is explicitly connected to the port. The rules for what connections will result in errors or warnings are defined in the IEEE 1364-2005 Verilog standard, in section 12.3.10[1]. For example, a **tri1** pullup net connected to a **tri0** pull-down net through a module port will result in a warning, per the Verilog standard. Such a connection will not be inferred by the `.name` syntax.

These restrictions reduce the risk of unintentional connections being inferred by the `.name` connection style. Any mismatch in vector sizes and/or types can still be forced, using the full named port connection style, if that is the intent of the designer. Such mismatches must be explicitly specified, however. They will not be inferred from the `.name` syntax.

9.4.2 Implicit .* port connection

. infers connections of all nets and ports of the same name*

SystemVerilog provides an additional short cut to simplify the specification of large netlists. The **.*** syntax indicates that all ports and nets (or variables) of the same name should automatically be connected together for that module instance. As with the `.name` syntax, for a connection to be inferred, the name and vector size must match exactly, and the types connected together must be compatible. Any connections that cannot be inferred by `.*` must be explicitly connected together, using Verilog's named port connection syntax.

Example 9-5 illustrates the use of SystemVerilog's `.*` port connection syntax.

1. IEEE Std 1364-2005, Language Reference Manual (LRM). See page xxvii of this book for details.

Example 9-5: Simple netlist using SystemVerilog's .* port connections

```
module miniPIC (
    inout  wire [7:0] port_a_pins,
    inout  wire [7:0] port_b_pins,
    inout  wire [7:0] port_c_pins,
    input  wire       clk,
    input  wire       resetN
  );

  wire [11:0] instruct_reg, program_data;
  wire [10:0] program_counter, program_address;
  wire [ 7:0] tmr0_reg, status_reg, fsr_reg, w_reg, option_reg,
              reg_file_out, port_a, port_b, port_c, trisa,
              trisb, trisc, data_bus, alu_a, alu_b;
  wire [ 6:0] reg_file_addr;
  wire [ 3:0] alu_opcode;
  wire [ 1:0] alu_a_sel, alu_b_sel;
  wire        reg_file_sel, special_reg_sel, reg_file_enable,
              w_reg_enable, zero_enable, carry_enable, skip,
              isoption, istris, polarity, carry, zero;


  pc_stack pcs (  // module instance with .* port connections
    .*
  );

  prom prom (
    .*,
    .dout(program_data),
    .address(program_address)
  );

  instruction_decode decoder (
    .*,
    .option(isoption),
    .tris(istris)
  );

  register_files regs (
    .*,
    .dout(reg_file_out),
    .address(reg_file_addr)
    );

  alu alu (
    .y(data_bus),
    .carry_out(carry),
    .zero_out(zero),
    .a(alu_a),
```

```
    .b(alu_b),
    .opcode(alu_opcode),
    .carry_in(status_reg[0])
  );

  glue_logic glue (
    .*
  );

endmodule
```

NOTE SystemVerilog adds two new types of hierarchy blocks that can also have ports, interfaces (see Chapter 10), and programs (refer to the companion book, *SystemVerilog for Verification*). Instances of these new blocks can also use the `.name` and `.*` inferred port connections. SystemVerilog also allows calls to functions and tasks to use named connections, including the `.name` and `.*` shortcuts. This is covered in section 6.3.5 on page 156.

9.5 Net aliasing

SystemVerilog adds an **alias** statement that allows two different names to reference the same net. For example:

```
wire clock;
wire clk;

alias clk = clock;
```

The net `clk` is an alias for `clock`, and `clock` is an alias for `clk`. Both names refer to the same logical net.

an alias creates two or more names for the same net

Defining an alias for a net does not copy the value of one net to some other net. In the preceding example, `clk` is not a copy of `clock`. Rather, `clk` is `clock`, just referenced by a different name. Any value changes on `clock` will be seen by `clk`, since they are the same net. Conversely, any value changes on `clk` will be seen by `clock`, since they are the same net.

alias versus assign

an alias is not an assignment

The **alias** statement is not the same as the **assign** continuous assignment. An **assign** statement continuously copies an expression on the right-hand side of the assignment to a net or variable on the left-hand side. This is a one-way copy. The net or variable on the left-hand side reflects any changes to the expression on the right-hand side. But, if the value of the net or variable on the left-hand side is changed, the change is not reflected back to the expression on the right-hand side.

changes on any aliased net affect all aliased nets

An **alias** works both ways, instead of one way. Any value changes to the net name on either side of the alias statement will be reflected on the net name on the other side. This is because an alias is effectively one net with two different names.

Multiple aliases

Several nets can be aliased together. A change on any of the net names will be reflected on all of the nets that are aliased together.

```
wire reset, rst, resetN, rstN;

alias rst = reset;
alias reset = resetN;
alias resetN = rstN;
```

The previous set of aliases can also be abbreviated to a single statement containing a series of aliases, as follows:

```
alias rst = reset = resetN = rstN;
```

aliases are not order dependent

The order in which nets are listed in an alias statement does not matter. An alias is not an assignment of values, it is a list of net names that refer to the same object.

9.5.1 Alias rules

SystemVerilog imposes several restrictions on what signals can be aliased to another name.

only net types can be aliased

- Only the net types can be aliased. Variables cannot be aliased. Verilog's net types are **wire**, **uwire**, **wand**, **wor**, **tri**, **triand**, **trior**, **tri0**, **tri1**, and **trireg**.

only nets of the same type can be aliased

- The aliased net type must be the same net type as the net to which it is aliased. A **wire** type can be aliased to a **wire** type, and a **wand** type can be aliased to a **wand** type. It is an error, however, to alias a **wire** to a **wand** or any other type.

only nets of the same size can be aliased

- The aliased net and the net to which it is aliased must be the same vector size. Note, however, that bit and part selects of nets can be aliased, so long as the vector size of the left-hand side and right-hand side of the alias statement are the same.

The following examples are all legal aliases of one net to another:

```
wire [31:0] n1;
wire [3:0][7:0] n2;

alias n2 = n1;  // both n1 and n2 are 32 bits


wire [39:0] d_in;
wire  [7:0] crc;
wire [31:0] data;

alias data = d_in[31:0];   // 32 bit nets
alias crc  = d_in[39:32];  // 8 bit nets
```

9.5.2 Implicit net declarations

implicit nets can be inferred from an alias

An alias statement can infer net declarations. It is not necessary to first explicitly declare each of the nets in the alias. Implicit nets are inferred, following the same rules as in Verilog for inferring an implicit net when an undeclared identifier is connected to a port of a module or primitive instance. In brief, these rules are:

- An undeclared identifier name on either side of an alias statement will infer a net type.
- The default implicit net type is **wire**. This can be changed with the `` `default_nettype `` compiler directive.
- If the net name is listed as a port of the containing module, the implicit net will be the same vector size as the port.
- If the net name is not listed in the containing module's port list, then a 1-bit net is inferred.

The following example infers single bit nets called `reset` and `rstN`, and 64 bit nets called `q` and `d`:

```
module register (output [63:0] q,
                 input  [63:0] d,
                 input         clock, reset);

  wire [63:0] out, in;
  alias in = d;   // infers d is a 64-bit wire
  alias out = q;  // infers q is a 64-bit wire
  alias rstN = reset;  // infers 1-bit wires
  ...
```

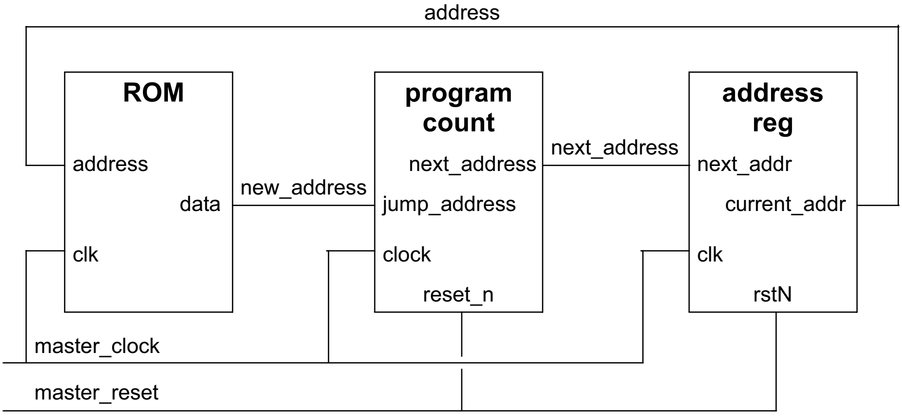
Net aliasing can also be used to define a net that represents part of another net. In the following example, `lo_byte` is an alias for the lower byte of a vector, and `hi_byte` is an alias for the upper byte. Observe that the order of signals in the alias statement does not matter. An alias is not an assignment statement. An alias is just multiple names for the same physical wires.

```
module (...);
  wire [63:0] data;
  wire [7:0] lo_byte, hi_byte;
  alias data[7:0] = lo_byte;
  alias hi_byte = data[63:56];
  ...
endmodule
```

9.5.3 Using aliases with .name and .*

The alias statement enables greater usage of the `.name` and `.*` shortcuts for modeling netlists. These shortcuts are used to connect a module port and net of the same name together, without the verbosity of Verilog's named port connection syntax. In the following example, however, these shortcuts cannot be fully utilized to connect the clock signals together, because the port names are not the same in each of the modules.

Figure 9-1: Diagram of a simple netlist



Example 9-6: Netlist using SystemVerilog's .* port connections without aliases

```
module chip (input wire master_clock,
             input wire master_reset,
             ...);

  wire [31:0] address, new_address, next_address;

  ROM           i1 ( .*,  // infers .address(address)
                     .data(new_address),
                     .clk(master_clock) );

  program_count i2 ( .*,  // infers .next_address(next_address)
                     .jump_address(new_address),
                     .clock(master_clock),
                     .reset_n(master_reset) );

  address_reg   i3 ( .*,  // no connections can be inferred
                     .next_addr(next_address),
                     .current_addr(address),
                     .clk(master_clock),
                     .rstN(master_reset) );

endmodule

module ROM (output wire [31:0] data,
            input  wire [31:0] address,
            input  wire        clk);
  ...
endmodule
```

```
module program_count (output logic [31:0] next_address,
                      input  wire  [31:0] jump_address,
                      input  wire         clock, reset_n);
  ...
endmodule

module address_reg (output wire [31:0] current_addr,
                    input  wire [31:0] next_addr,
                    input  wire        clk, rstN);
  ...
endmodule
```

using aliases can simplify netlists

The `master_clock` in `chip` should be connected to all three modules in the netlist. However, the clock input ports in the modules are not called `master_clock`. In order for the `master_clock` net in the top-level `chip` module to be connected to the clock ports of the other modules, all of the different clock port names must be aliased to `master_clock`. Similar aliases can be used to connect all reset ports to the `master_reset` net, and to connect other ports together that do not have the same name.

Example 9-7 adds these alias statements, which allow the netlist to take full advantage of the `.*` shortcut to connect all modules together. In this example, wires for the vectors are explicitly declared, and wires for the different clock and reset names are implicitly declared from the alias statement.

Example 9-7: Netlist using SystemVerilog's .* connections along with net aliases

```
module chip (input wire master_clock,
             input wire master_reset,
             ...);

  wire [31:0] address, data, new_address, jump_address,
              next_address, next_addr, current_addr;

  alias clk = clock = master_clock;
  alias rstN = reset_n = master_reset;
  alias data = new_address = jump_address;
  alias next_address = next_addr;
  alias current_addr = address;

  ROM           i1 ( .* );
```

```
  program_count i2 ( .* );
  address_reg   i3 ( .* );

endmodule

module ROM (output wire [31:0] data,
            input  wire [31:0] address,
            input  wire        clk);
  ...
endmodule

module program_count (output logic [31:0] next_address,
                      input  wire  [31:0] new_count,
                      input  wire         clock, reset_n);
  ...
endmodule

module address_reg (output wire [31:0] address,
                    input  wire [31:0] next_address,
                    input  wire        clk, rstN);
  ...
endmodule
```

In this example, the `.*` shortcuts infer the following connections to the module ports of the module instances:

```
ROM           i1 (.data(data),
                  .address(address)
                  .clk(clk) );

program_count i2 (.next_address(next_address),
                  .jump_address(jump_address),
                  .clock(clock),
                  .reset_n(reset_n) );

address_reg i3   (.current_addr(current_addr),
                  .next_addr(next_addr),
                  .clk(clk),
                  .rstN(rstN) );
```

Even though different net names are connected to different module instances, such as `clk` to the `ROM` module and `clock` to the `program_count` module, the alias statements make them the same net, and make those nets the same as `master_clock`.

9.6 Passing values through module ports

Verilog restrictions on module ports

The Verilog language places a number of restrictions on what types of values can be passed through the ports of a module. These restrictions affect both the definition of the module and any instances of the module. The following bullets give a brief summary of the Verilog restrictions on module ports:

- Only net types, such as the **`wire`** type, can be used on the receiving side of the port. It is illegal to connect any type of variable, such as **`reg`** or **`integer`** , to the receiving side of a module port.
- Only net, **`reg`**, and **`integer`** types, or a literal integer value can be used on the transmitting side of the port.
- It is illegal to pass the **`real`** type through module ports without first converting it to a vector using the **`$realtobits`** system function, and then converting it back to a real number, after passing through the port, with the **`$bitstoreal`** system function.
- It is illegal to pass unpacked arrays of any number of dimensions through module ports.

9.6.1 All types can be passed through ports

SystemVerilog removes most port restrictions

SystemVerilog removes nearly all restrictions on the types of values that can be passed through module ports. With SystemVerilog:

- Values of any type can be used on both the receiving and transmitting sides of module ports, including real values.
- Packed and unpacked arrays of any number of dimensions can be passed through ports.
- SystemVerilog structures and unions can be passed through module ports.

NOTE SystemVerilog adds two new types of hierarchy blocks that can also have ports, interfaces (see Chapter 10), and programs (refer to the companion book, *SystemVerilog for Verification*). These new blocks have the same port connection rules as modules.

The following example illustrates the flexibility of passing values through module ports in SystemVerilog. In this example, variables are used on both sides of some ports, a structure is passed through a port, and an array, representing a look-up table, is passed through a port.

Example 9-8: Passing structures and arrays through module ports

```
typedef struct packed {
  logic [ 3:0] opcode;
  logic [15:0] operand;
} instruction_t;

module decoder (output logic [23:0]  microcode,
                input  instruction_t instruction,
                input  logic [23:0]  LUT [0:(2**20)-1] );

  ...  // do something with Look-Up-Table and instruction

endmodule

module DSP (input  logic        clock, resetN,
            input  logic [ 3:0] opcode,
            input  logic [15:0] operand,
            output logic [23:0] data );

  logic [23:0] LUT [0:(2**20)-1];  // Look Up Table

  instruction_t instruction;
  logic [23:0]  microcode;

  decoder i1 (microcode, instruction, LUT);

  ...  // do something with microcode output from decoder

endmodule
```

9.6.2 Module port restrictions in SystemVerilog

SystemVerilog does place two restrictions on the values that are passed through module ports. These restrictions are intuitive, and help ensure that the module ports accurately represent the behavior of hardware.

variables can only receive values from a single source

The first restriction is that a variable type can only have a single source that writes a value to the variable at any given moment in time. A source can be:

- a single module output or inout port
- a single primitive output or inout port
- a single continuous assignment
- any number of procedural assignments

multi source logic requires net types

The reason for this single source restriction when writing to variables is that variables simply store the last value written into them. If there were multiple sources, the variable would only reflect the value of the last source to change. Actual hardware behavior for multi-source logic is different. In hardware, multiple sources, or "drivers", are merged together, based on the hardware technology. Some technologies merge values based on the strength of the drivers, some technologies logically-and multiple drivers together, and others logically-or multiple drivers together. This implementation detail of hardware behavior is represented with Verilog net types, such as **`wire`**, **`wand`**, and **`wor`**. Therefore, SystemVerilog requires that a net type be used when a signal has multiple drivers. An error will occur if a variable is connected to two drivers.

Any number of procedural assignments is still considered a single source for writing to the variable. This is because procedural assignments are momentary statements that store a value but do not continuously update that value. For example, in an **`if`**...**`else`** programing statement, either one branch or the other can be used to update the value of the same variable, but both branches do not write to the same variable at the same time. Even multiple procedural assignments to the same variable at the same simulation time behave as temporary writes to the variable, with the last assignment executed representing the value that is actually stored in the variable. A continuous assignment or a connection to an output or inout port, on the other hand, needs to continuously update the variable to reflect the hardware behavior of a continuous electrical source.

unpacked values must have matching layouts

The second restriction SystemVerilog places on values passed through module ports is that unpacked types must be identical in layout on both sides of a module port. SystemVerilog allows structures, unions, and arrays to be specified as either packed or unpacked (see sections 5.1.3 on page 101, 5.2.1 on page 106, and 5.3.1 on page 113, respectively). When arrays, structures or unions are unpacked, the connections must match exactly on each side of the port.

For unpacked arrays, an exact match on each side of the port is when there are the same number of dimensions in the array, each dimension is the same size, and each element of the array is the same size.

For unpacked structures and unions, an exact match on each side of the port means that each side is declared using the same typedef definition. In the following example, the structure connection to the output port of the buffer is illegal. Even though the port and the connection to it are both declared as structures, and the structures have the same declarations within, the two structures are not declared from the same user-defined type, and therefore are not an exact match. The two structures cannot be connected through a module port. In this same example, however, the structure passed through the input port *is* legal. Both the port and the structure connected to it are declared using the same user-defined type definition. These two structures are exactly the same.

```
typedef struct {   // unpacked structure
  logic [23:0] short_word;
  logic [63:0] long_word;
} data_t;

module buffer (input  data_t  in,
               output data_t  out);
  ...
endmodule

module chip (...);

  data_t din;          // unpacked structure

  struct {             // unpacked structure
    logic [23:0] short_word;
    logic [63:0] long_word;
  } dout;

  buffer i1 (.in(din),     // legal connection
             .out(dout)    // illegal connection
            );
  ...
endmodule
```

Packed and unpacked arrays, structures, and unions are discussed in more detail in Chapter 5.

packed values are passed through ports as vectors

The restrictions described above on passing unpacked values through ports do not apply to packed values. Packed values are stored as contiguous bits, are analogous to a vector of bits, and are passed through module ports as vectors. If the array, structure, or union are different sizes on each side of the port, Verilog's standard rules are followed for a mismatch in vector sizes.

9.7 Reference ports

Verilog modules can have **input**, **output** and bidirectional **inout** ports. These port types are used to pass a value of a net or variable from one module instance to another.

a ref port passes a hierarchical reference through a port

SystemVerilog adds a fourth port type, called a **ref** port. A **ref** port passes a hierarchical reference to a variable through a port, instead of passing the value of the variable. The name of the port becomes an alias to hierarchical reference. Any references to that port name directly reference the actual source.

A reference to a variable of any type can be passed through a ref port. This includes all built-in variable types, structures, unions, enumerated types, and other user-defined types. To pass a reference to a variable through a port, the port direction is declared as **ref**, instead of an **input**, **output**, or **inout**. The type of a **ref** port must be the same type as the variable connected to the port.

The following example passes a reference to an array into a module, using a **ref** port.

Example 9-9: Passing a reference to an array through a module **ref** port

```
typedef struct packed {
  logic [ 3:0] opcode;
  logic [15:0] operand;
} instruction_t;

module decoder (output logic [23:0]  microcode,
                input  instruction_t instruction,
                ref    logic [23:0]  LUT [0:(2**20)-1] );

  ...  // do something with Look-Up-Table and instruction

endmodule
```

```
module DSP (input  logic        clock, resetN,
            input  logic [ 3:0] opcode,
            input  logic [15:0] operand,
            output logic [23:0] data );

  logic [23:0] LUT [0:(2**20)-1];  // Look Up Table

  instruction_t instruction;
  logic [23:0]  microcode;

  decoder i1 (microcode, instruction, LUT);

  ...  // do something with microcode output from decoder

endmodule
```

9.7.1 Reference ports as shared variables

NOTE Passing variables through ports by reference creates shared variables, which do not behave like hardware.

Passing a reference to a variable to another module makes it possible for more than one module to write to the same variable. This effectively defines a single variable that can be shared by multiple modules. That is, procedural blocks in more than one module could potentially write values into the same variable.

A variable that is written to by more than one procedural block does not behave the same as a net with multiple sources (drivers). Net types have resolution functionality that continuously merge multiple sources into a single value. A **`wire`** net, for example, resolves multiple drivers, based on strength levels. A **`wand`** net resolves multiple drivers by performing a bit-wise AND operation. Variables do not have multiple driver resolution. Variables simply store the last value deposited. When multiple modules share the same variable through **`ref`** ports, the value of the variable at any given time will be the last value written, which could have come from any of the modules that share the variable.

9.7.2 Synthesis guidelines

NOTE Passing references through ports is not synthesizable.

Passing references to variables through module ports is not synthesizable. It is recommended that the use of **ref** ports should be reserved for abstract modeling levels where synthesis is not a consideration.

9.8 Enhanced port declarations

9.8.1 Verilog-1995 port declarations

Verilog-1995 port declaration style is verbose

Verilog-1995 required a verbose set of declarations to fully declare a module's ports. The module statement contains a port list which defines the names of the ports and the order of the ports. Following the module statement, one or more separate statements are required to declare the direction of the ports. Following the port direction declarations, additional optional statements are required to declare the types of the internal signals represented by the ports. If the types are not specified, the Verilog-1995 syntax infers a net type, which, by default, is the **wire** type. This default type can be changed, using the `` `default_nettype `` compiler directive.

```
module accum (data, result, co, a, b, ci);
  inout   [31:0]  data;
  output  [31:0]  result;
  output          co;
  input   [31:0]  a, b;
  input           ci;

  wire    [31:0]  data;
  reg     [31:0]  result;
  reg             co;
  tri1            ci;
  ...
endmodule
```

9.8.2 Verilog-2001 port declarations

Verilog-2001 port declaration style is more concise

Verilog-2001 introduced ANSI-C style module port declarations, which allow the port names, port size, port direction, and type declarations to be combined in the port list.

```
module accum (inout   wire [31:0]  data,
              output  reg  [31:0]  result,
              output  reg          co,
```

```
                    input         [31:0]  a, b,
                    input   tri1          ci );
   ...
endmodule
```

Verilog-2001 ports have a direction, type and size

With the Verilog-2001 port declaration syntax, the port direction is followed by an optional type declaration, and then an optional vector size declaration. If the optional type is not specified, a default type is inferred, which is the **wire** type, unless changed by the `` `default_nettype `` compiler directive. If the optional vector size is not specified, the port defaults to the default width of the type. Following the optional width declaration is a comma-separated list of one or more port names. Each port in the list will be of the direction, type, and size specified.

in Verilog, all ports must have a direction declared

Verilog-2001's ANSI-C style port declarations greatly simplify the Verilog-1995 syntax for module port declarations. There are, however, three limitations to the Verilog-2001 port declaration syntax:

- All ports must have a direction explicitly declared.
- The type cannot be changed for a subsequent port without re-specifying the port direction.
- The vector size of the port cannot be changed for a subsequent port without re-specifying the port direction and optional type.

In the preceding example, the optional type is specified for all but the `a` and `b` input ports. These two ports will automatically infer the default type. The optional vector size is specified for the `data`, `result`, `a`, and `b` ports; but not for the `co` and `ci` ports. The unsized ports will default to the default size of their respective types, which are both 1 bit wide. The vector sizes for `result` and `co` are different. In order to change the size declaration for `co`, it is necessary to re-specify the port direction and type of `co`. Also, in the preceding example, input ports `a` and `b` do not have a type defined, and therefore default to a **wire** type. In order to change the type for the `ci` input port, the port direction must be re-specified, even though it is the same direction as the preceding ports.

9.8.3 SystemVerilog port declarations

SystemVerilog simplifies the declaration of module ports in several ways.

first port defaults to inout

First, SystemVerilog specifies a default port direction of **inout** (bidirectional). Therefore, it is no longer required to specify a port direction, unless the direction is different than the default.

subsequent ports default to direction of previous port

Secondly, if the next port in the port list has a type defined, but no direction is specified, the direction defaults to the direction of the previous port in the list. This allows the type specification to be changed without re-stating the port direction.

Using SystemVerilog, the Verilog-2001 module declaration for an accumulator shown on the previous page can be simplified to:

```
module accum (wire [31:0]  data,
              output reg [31:0] result, reg co,
              input [31:0] a, b, tri1 ci );
  ...
endmodule
```

The first port in the list, `data`, has a type, but no explicit port direction. Therefore, this port defaults to the direction of **inout**. Port `co` also has a type, but no port direction. This port defaults to the direction of the previous port in the list, which is **output**. Ports `a` and `b` have a port direction declared, but no type. As with Verilog-2001 and Verilog-1995, an implicit net type will be inferred, which by default is the type **wire**. Finally, port `ci` has a type declared, but no port direction. This port will inherit the direction of the previous port in the list, which is **input**.

NOTE SystemVerilog adds two new types of hierarchy blocks that can also have ports, interfaces (see Chapter 10), and programs (refer to the companion book on *SystemVerilog for Verification*). These new blocks have the same port declaration rules as modules.

Backward compatibility

SystemVerilog remains fully backward compatible with Verilog by adding a rule that, if the first port has no direction *and* no type specified, then the Verilog 1995 port list syntax is inferred, and no other port in the list can have a direction or type specified within the port list.

```
module accum (data, result, ...);
  // Verilog-1995 style because first port has
```

```
    // no direction and no type

module accum (data, wire [31:0] result, ...);
    // ERROR: cannot mix Verilog-1995 style with
    // Verilog-2001 or SystemVerilog style
```

9.9 Parameterized types

parameterized modules

Verilog provides the ability to define **parameter** and **localparam** constants, and then use those constants to calculate the vector widths of module ports or other declarations. A parameter is a constant, that can be redefined at elaboration time for each instance of a module. Modules that can be redefined using parameters are often referred to as *parameterized modules*.

polymorphic modules using parameterized types

SystemVerilog adds a significant extension to the concept of redefinable, parameterized modules. With SystemVerilog, the net and variable types of a module can be parameterized. Parameterized types are declared using the **parameter type** pair of keywords. As with other parameters, parameterized types can be redefined for each instance of a module. This capability introduces an additional level of polymorphism to Verilog models. With Verilog, parameter redefinition can be used to change vector sizes and other constant characteristics for each instance of a model. With SystemVerilog, the behavior of a module can be changed based on the net and variable types of a module instance.

Parameterized types are synthesizable, provided the default or redefined types are synthesizable types.

In the following example, the variable type used by an adder is parameterized. By default, the type is **shortint**. Module `big_chip` contains three instances of the adder. Instance `i1` uses the adder's default variable type, making it a 16-bit signed adder. Instance `i2` redefines the variable type to **int**, making this instance a 32-bit signed adder. Instance `i3` redefines the variable type to **int unsigned**, which makes this third instance a 32-bit unsigned adder.

Example 9-10: Polymorphic adder using parameterized variable types

```
module adder #(parameter type ADDERTYPE = shortint)
              (input  ADDERTYPE  a, b,  // redefinable type
               output ADDERTYPE  sum,   // redefinable type
               output logic      carry);

  ADDERTYPE temp;  // local variable with redefinable type

  ... // adder functionality
endmodule

module big_chip( ... );
  shortint     a, b, r1;
  int          c, d, r2;
  int unsigned e, f, r3;
  wire         carry1, carry2, carry3;

  // 16-bit unsigned adder
  adder                  i1 (a, b, r1, carry1);

  // 32-bit signed adder
  adder #(.ADDERTYPE(int)) i2 (c, d, r2, carry2);

  // 32-bit unsigned adder
  adder #(.ADDERTYPE(int unsigned)) i3 (e, f, r3, carry3);
endmodule
```

9.10 Summary

This chapter has presented a number of important extensions to the Verilog language that allow modeling the very large netlists that occur in multi-million gate designs. Constructs such as `.name` and `.*` port connections reduce the verbosity and redundancy in netlists. net aliasing, simplified port declarations, port connections by reference, and relaxed rules on the types of values that can be passed through ports all make representing complex design hierarchy easier to model and maintain.

The next chapter presents SystemVerilog interfaces, which is another powerful construct for simplifying large netlists.

Chapter 10

SystemVerilog Interfaces

SystemVerilog extends the Verilog language with a powerful interface construct. Interfaces offer a new paradigm for modeling abstraction. The use of interfaces can simplify the task of modeling and verifying large, complex designs.

This chapter contains a number of small examples, each one showing specific features of interfaces. These examples have been purposely kept relatively small and simple, in order to focus on specific features of interfaces. Chapter 11 then presents a larger example that uses interfaces in the context of a more complete design.

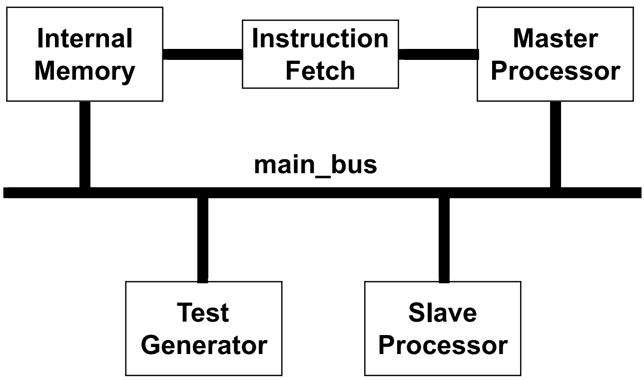
The concepts covered in this chapter are:

- Interface declarations
- Connecting interfaces to module ports
- Differences between interfaces and modules
- Interface ports and directions
- Tasks and functions in interfaces
- Using interface methods
- Procedural blocks in interfaces
- Parameterized interfaces

10.1 Interface concepts

The Verilog language connects modules together through module ports. This is a detailed method of representing the connections between blocks of a design that maps directly to the physical connections that will be in the actual hardware. For large designs, however, using module ports to connect blocks of a design together can become tedious and redundant. Consider the following example that connects five blocks of a design together using a rudimentary bus architecture called `main_bus`, plus some additional connections between some of the design blocks. Figure 10-1 shows the block diagram for this simple design, and example 10-1 lists the Verilog source code for the module declarations involved.

Figure 10-1: Block diagram of a simple design



Example 10-1: Verilog module interconnections for a simple design

```
/********************** Top-level Netlist ********************/
module top (input wire clock, resetN, test_mode);
  wire [15:0] data, address, program_address, jump_address;
  wire [ 7:0] instruction, next_instruction;
  wire [ 3:0] slave_instruction;
  wire        slave_request, slave_ready;
  wire        bus_request, bus_grant;
  wire        mem_read, mem_write;
  wire        data_ready;
```

```
processor proc1 (
  // main_bus ports
  .data(data),
  .address(address),
  .slave_instruction(slave_instruction),
  .slave_request(slave_request),
  .bus_grant(bus_grant),
  .mem_read(mem_read),
  .mem_write(mem_write),
  .bus_request(bus_request),
  .slave_ready(slave_ready),
  // other ports
  .jump_address(jump_address),
  .instruction(instruction),
  .clock(clock),
  .resetN(resetN),
  .test_mode(test_mode)
);
```

signals for main_bus must be individually connected to each module instance

```
slave slave1 (
  // main_bus ports
  .data(data),
  .address(address),
  .bus_request(bus_request),
  .slave_ready(slave_ready),
  .mem_read(mem_read),
  .mem_write(mem_write),
  .slave_instruction(slave_instruction),
  .slave_request(slave_request),
  .bus_grant(bus_grant),
  .data_ready(data_ready),
  // other ports
  .clock(clock),
  .resetN(resetN)
);
```

main_bus connections

```
dual_port_ram ram (
  // main_bus ports
  .data(data),
  .data_ready(data_ready),
  .address(address),
  .mem_read(mem_read),
  .mem_write(mem_write),
  // other ports
  .program_address(program_address),
  .data_b(next_instruction)
);
```

main_bus connections

```
    test_generator test_gen(
    // main_bus ports
    .data(data),
    .address(address),
    .mem_read(mem_read),
    .mem_write(mem_write),
    // other ports
    .clock(clock),
    .resetN(resetN),
    .test_mode(test_mode)
  );

  instruction_reg ir (
    .program_address(program_address),
    .instruction(instruction),
    .jump_address(jump_address),
    .next_instruction(next_instruction),
    .clock(clock),
    .resetN(resetN)
  );
endmodule
```

main_bus connections

```
/******************** Module Definitions *******************/
module processor (
    // main_bus ports
    inout  wire [15:0] data,
    output reg  [15:0] address,
    output reg  [ 3:0] slave_instruction,
    output reg         slave_request,
    output reg         bus_grant,
    output wire        mem_read,
    output wire        mem_write,
    input  wire        bus_request,
    input  wire        slave_ready,
    // other ports
    output reg  [15:0] jump_address,
    input  wire [ 7:0] instruction,
    input  wire        clock,
    input  wire        resetN,
    input  wire        test_mode
  );
  ...  // module functionality code
endmodule
```

ports for main_bus must be individually declared in each module definition

```
module slave (
    // main_bus ports
    inout  wire [15:0] data,
    inout  wire [15:0] address,
    output reg         bus_request,
    output reg         slave_ready,
    output wire        mem_read,
    output wire        mem_write,
    input  wire [ 3:0] slave_instruction,
    input  wire        slave_request,
    input  wire        bus_grant,
    input  wire        data_ready,
    // other ports
    input  wire        clock,
    input  wire        resetN
  );
  ...  // module functionality code
endmodule
```

main_bus port declarations

```
module dual_port_ram (
    // main_bus ports
    inout  wire [15:0] data,
    output wire        data_ready,
    input  wire [15:0] address,
    input  tri0        mem_read,
    input  tri0        mem_write,
    // other ports
    input  wire [15:0] program_address,
    output reg  [ 7:0] data_b
  );
  ...  // module functionality code
endmodule
```

main_bus port declarations

```
module test_generator (
    // main_bus ports
    output wire [15:0] data,
    output reg  [15:0] address,
    output reg         mem_read,
    output reg         mem_write,
    // other ports
    input  wire        clock,
    input  wire        resetN,
    input  wire        test_mode
  );
  ...  // module functionality code
endmodule
```

main_bus port declarations

```
module instruction_reg (
    output reg  [15:0] program_address,
    output reg  [ 7:0] instruction,
    input  wire [15:0] jump_address,
    input  wire [ 7:0] next_instruction,
    input  wire        clock,
    input  wire        resetN
  );
  ...  // module functionality code
endmodule
```

10.1.1 Disadvantages of Verilog's module ports

Verilog's module ports provide a simple and intuitive way of describing the interconnections between the blocks of a design. In large, complex designs, however, Verilog's module ports have several shortcomings. Some of these are:

- Declarations must be duplicated in multiple modules.
- Communication protocols must be duplicated in several modules.
- There is a risk of mismatched declarations in different modules.
- A change in the design specification can require modifications in multiple modules.

connecting modules in a netlist requires redundant port declarations

One disadvantage of using Verilog's module ports to connect major blocks of a design together is readily apparent in the example code above. The signals that make up `main_bus` in the preceding example must be declared in each module that uses the bus, as well as in the top-level netlist that connects the design together. In this simple example, there are only a handful of signals in `main_bus`, so the redundant declarations are mostly just an inconvenience. In a large, complex design, however, this redundancy becomes much more than an inconvenience. A large design could have dozens of modules connected to the same bus, with dozens of duplicated declarations in each module. If the ports of one module should inadvertently be declared differently than the rest of the design, a functional error can occur that may be difficult to find.

The replicated port declarations also mean that, should the specification of the bus change during the design process, or in a next generation of the design, then each and every module that shares the bus must be changed. All netlists used to connect the modules using the bus must also be changed. This wide spread effect of a change is counter to good coding styles. One goal of coding is to structure the code in such a way that a small change in one place should not require changing other areas of the code. A weakness in the Verilog language is that a change to the ports in one module will usually require changes in other modules.

protocols must be duplicated in each module

Another disadvantage of Verilog's module ports is that communication protocols must be duplicated in each module that utilize the interconnecting signals between modules. If, for example, three modules read and write from a shared memory device, then the read and write control logic must be duplicated in each of these modules.

module ports inhibit abstract top-down design

Yet another disadvantage of using module ports to connect the blocks of a design together is that detailed interconnections for the design must be determined very early in the design cycle. This is counter to the top-down design paradigm, where models are first written at an abstract level without extensive design detail. At an abstract level, an interconnecting bus should not require defining each and every signal that makes up the bus. Indeed, very early in the design specification, all that might be known is that the blocks of the design will share certain information. In the block diagram shown in Figure 10-1 on page 264, the `main_bus` is represented as a single connection. Using Verilog's module ports to connect the design blocks together, however, does not allow modeling at that same level of abstraction. Before any block of the design can be modeled, the bus must first be broken down to individual signals.

10.1.2 Advantages of SystemVerilog interfaces

an interface is an abstract port type

SystemVerilog adds a powerful new port type to Verilog, called an **`interface`**. An interface allows a number of signals to be grouped together and represented as a single port. The declarations of the signals that make up the interface are contained in a single location. Each module that uses these signals then has a single port of the interface type, instead of many ports with the discrete signals.

Example 10-2 shows how SystemVerilog's interfaces can reduce the amount of code required to model the simple design shown in Figure 10-1. By encapsulating the signals that make up `main_bus` as an interface, the redundant declarations for these signals within each module are eliminated.

Example 10-2: SystemVerilog module interconnections using interfaces

```
/****************** Interface Definitions ******************/
interface main_bus;
  wire  [15:0] data;
  wire  [15:0] address;
  logic [ 7:0] slave_instruction;
  logic        slave_request;
  logic        bus_grant;
  logic        bus_request;
  logic        slave_ready;
  logic        data_ready;
  logic        mem_read;
  logic        mem_write;
endinterface
```

signals for main_bus are defined in just one place

```
/********************* Top-level Netlist ******************/
module top (input logic clock, resetN, test_mode);
  logic [15:0] program_address, jump_address;
  logic [ 7:0] instruction, next_instruction;

  main_bus bus ( );  // instance of an interface
                     // (instance name is bus)

  processor proc1 (
    // main_bus ports
    .bus(bus),  // interface connection
    // other ports
    .jump_address(jump_address),
    .instruction(instruction),
    .clock(clock),
    .resetN(resetN),
    .test_mode(test_mode)
  );
```

each module instance has a single connection for main_bus

```
  slave slave1 (
    // main_bus ports
    .bus(bus),  // interface connection
    // other ports
    .clock(clock),
    .resetN(resetN)
  );
```

main_bus connections

```
  dual_port_ram ram (
    // main_bus ports
    .bus(bus),  // interface connection
    // other ports
```

main_bus connections

```
    .program_address(program_address),
    .data_b(next_instruction)
  );

    test_generator test_gen(
    // main_bus ports
    .bus(bus),  // interface connection
    // other ports
    .clock(clock),
    .resetN(resetN),
    .test_mode(test_mode)
  );
```

main_bus connections

```
  instruction_reg ir (
    .program_address(program_address),
    .instruction(instruction),
    .jump_address(jump_address),
    .next_instruction(next_instruction),
    .clock(clock),
    .resetN(resetN)
  );
endmodule


/******************** Module Definitions *******************/
module processor (
    // main_bus interface port
    main_bus bus,  // interface port
    // other ports
    output logic [15:0] jump_address,
    input  logic [ 7:0] instruction,
    input  logic        clock,
    input  logic        resetN,
    input  logic        test_mode
  );
  ...  // module functionality code
endmodule
```

each module definition has a single port for main_bus

```
module slave (
    // main_bus interface port
    main_bus bus,  // interface port
    // other ports
    input  logic        clock,
    input  logic        resetN
  );
  ...  // module functionality code
endmodule
```

main_bus port declaration

```
module dual_port_ram (
    // main_bus interface port
    main_bus bus,  // interface port          main_bus port declaration
    // other ports
    input  logic [15:0] program_address,
    output logic [ 7:0] data_b
  );
  ...  // module functionality code
endmodule

module test_generator (
    // main_bus interface port
    main_bus bus,  // interface port          main_bus port declaration
    // other ports
    input  logic        clock,
    input  logic        resetN,
    input  logic        test_mode
  );
  ...  // module functionality code
endmodule

module instruction_reg (
    output logic [15:0] program_address,
    output logic [ 7:0] instruction,
    input  logic [15:0] jump_address,
    input  logic [ 7:0] next_instruction,
    input  logic        clock,
    input  logic        resetN
  );
  ...  // module functionality code
endmodule
```

In example 10-2, above, all the signals that are in common between the major blocks of the design have been encapsulated into a single location—the interface declaration called `main_bus`. The top-level module and all modules that make up these blocks do not repetitively declare these common signals. Instead, these modules simply use the interface as the connection between them.

Encapsulating common signals into a single location eliminates the redundant declarations of Verilog modules. Indeed, in the preceding example, since `clock` and `resetN` are also common to all modules, these signals could have also been brought into the interface.

This further simplification is shown later in this chapter, in example 10-3 on page 274.

10.1.3 SystemVerilog interface contents

interfaces can contain functionality

SystemVerilog interfaces are far more than just a bundle of wires. Interfaces can encapsulate the full details of the communication between the blocks of a design. Using interfaces:

- The discrete signal and ports for communication can be defined in one location, the interface.
- Communication protocols can be defined in the interface.
- Protocol checking and other verification routines can be built directly into the interface.

interfaces eliminate redundant declarations

With Verilog, the communication details must be duplicated in each module that shares a bus or other communication architecture. SystemVerilog allows all the information about a communication architecture and the usage of the architecture to be defined in a single, common location. An interface can contain type declarations, tasks, functions, procedural blocks, program blocks, and assertions. SystemVerilog interfaces also allow multiple views of the interface to be defined. For example, for each module connected to the interface, the `data_bus` signal can be defined to be an input, output or bidirectional port.

All of these capabilities of SystemVerilog interfaces are described in more detail in the following sections of this chapter.

10.1.4 Differences between modules and interfaces

Interfaces are not the same as modules

There are three fundamental differences that make an interface differ from a module. First, an interface cannot contain design hierarchy. Unlike a module, an interface cannot contain instances of modules or primitives that would create a new level of implementation hierarchy. Second, an interface can be used as a module port, which is what allows interfaces to represent communication channels between modules. It is illegal to use a module in a port list. Third, an interface can contain modports, which allow each module connected to the interface to see the interface differently. Modports are described in detail in section 10.6 on page 281.

10.2 Interface declarations

interfaces are defined in a similar way as modules

Syntactically, the definition of an interface is very similar to the definition of a module. An interface can have ports, just as a module does. This allows signals that are external to the interface, such as a clock or reset line, to be brought into the interface and become part of the bundle of signals represented by the interface. Interfaces can also contain declarations of any Verilog or SystemVerilog type, including all variable types, all net types and user-defined types.

Example 10-3 shows a definition for an interface called `main_bus`, with three external signals coming into the interface: `clock`, `resetN` and `test_mode`. These external signals can now be connected to each module through the interface, without having to explicitly connect the signals to each module.

Notice in this example how the instance of interface `main_bus` has the `clock`, `resetN` and `test_mode` signals connected to it, using the same syntax as connecting signals to an instance of a module.

Example 10-3: The interface definition for `main_bus`, with external inputs

```
/******************* Interface Definitions ******************/
interface main_bus (input logic clock, resetN, test_mode);
  wire  [15:0] data;
  wire  [15:0] address;
  logic [ 7:0] slave_instruction;
  logic        slave_request;
  logic        bus_grant;
  logic        bus_request;
  logic        slave_ready;
  logic        data_ready;
  logic        mem_read;
  logic        mem_write;
endinterface

/********************* Top-level Netlist *******************/
module top (input logic clock, resetN, test_mode);
  logic [15:0] program_address, jump_address;
  logic [ 7:0] instruction, next_instruction;

  main_bus bus (    // instance of an interface
    .clock(clock),
    .resetN(resetN),
    .test_mode(test_mode)
  );
```

discrete signals are inputs to the interface

discrete signals are connected to the interface instance

```
  processor proc1 (
    // main_bus ports
    .bus(bus),  // interface connection
    // other ports
    .jump_address(jump_address),
    .instruction(instruction)
  );

  ...

/*** remainder of netlist and module definitions are not  ***/
/*** listed — they are similar to example 10-2, but       ***/
/*** clock and resetN do not need to be passed to each    ***/
/*** module instance as discrete ports.                   ***/
```

discrete signals do not need to be connected to each design block instance

interface instances can use .name and . connections*

The SystemVerilog simplified port connection styles of `.name` and `.*` can also be used with interface port connections. These constructs are covered in section 9.4 on page 233. The previous examples can be made even more concise by combining the use of interfaces with the use of `.*` port connections. This is illustrated in example 10-4, which follows.

Example 10-4: Using interfaces with `.*` connections to simplify complex netlists

```
/****************** Interface Definitions *****************/
interface main_bus (input logic clock, resetN, test_mode);
  wire  [15:0] data;
  wire  [15:0] address;
  logic [ 7:0] slave_instruction;
  logic        slave_request;
  logic        bus_grant;
  logic        bus_request;
  logic        slave_ready;
  logic        data_ready;
  logic        mem_read;
  logic        mem_write;
endinterface

/******************** Top-level Netlist ******************/
module top (input logic clock, resetN, test_mode);
  logic [15:0] program_address, jump_address;
  logic [ 7:0] instruction, next_instruction, data_b;
```

```
  main_bus         bus        ( .* );

  processor        proc1      ( .* );
  slave            slave1     ( .* );
  instruction_reg ir          ( .* );
  test_generator  test_gen    ( .* );
  dual_port_ram    ram        ( .*, .data_b(next_instruction) );

endmodule

/*** remainder of netlist and module definitions are not   ***/
/*** listed — they are similar to example 10-2, but        ***/
/*** clock and resetN do not need to be passed to each     ***/
/*** module instance as discrete ports.                    ***/
```

.* port connections can significantly reduce a netlist (compare to netlist in example 10-2 on page 270).

SystemVerilog greatly simplifies netlists

In the Verilog version of this simple example, which was listed in example 10-1 on page 264, the top-level netlist, module `top`, required 65 lines of code, excluding blank lines and comments. Using SystemVerilog interfaces along with `.*`, example 10-4, above, requires just 10 lines of code, excluding blank lines and comments, to model the same connectivity.

10.2.1 Source code declaration order

an interface name can be used before its definition

The name of an interface can be referenced in two contexts: in a port of a module, and in an instance of the interface. Interfaces can be used as module ports without concern for file order dependencies. Just as with modules, the name of an interface can be referenced before the source code containing the interface definition has been read in by the software tool. This means any module can use an interface as a module port, without concern for the order in which the source code is compiled.

10.2.2 Global and local interface definitions

interfaces can be global declarations

An interface can be defined separately from module definitions, using the keywords **`interface`** and **`endinterface`**. The name of the interface will be in the global module definition name scope, just as with module names. This allows an interface definition to be used as a port by any module, anywhere in the design hierarchy.

interfaces can be limited to specific hierarchy scopes

An interface definition can be nested within a module, making the name of the interface local to that module. Only the containing module can instantiate a locally declared interface. This allows the use of an interface to be limited to just one portion of the design hierarchy, such as to just within an IP model.

10.3 Using interfaces as module ports

With SystemVerilog, a port of a module can be declared as an interface type, instead of the Verilog **input**, **output** or **inout** port directions.

10.3.1 Explicitly named interface ports

a module port can be the name of an interface

A module port can be explicitly declared as a specific type of interface. This is done by using the name of an interface as the port type. The syntax is:

```
module <module_name> (<interface_name> <port_name>);
```

For example:

```
interface chip_bus;
  ...
endinterface


module CACHE (chip_bus pins,  // interface port
              input    clock);
  ...
endmodule
```

An explicitly named interface port can only be connected to an interface of the same name. An error will occur if any other interface definition is connected to the port. Explicitly named interface ports ensure that a wrong interface can never be inadvertently connected to the port. Explicitly naming the interface type that can be connected to the port also serves to document directly within the port declaration exactly how the port is intended to be used.

10.3.2 Generic interface ports

a port can be declared using the interface keyword

A generic interface port defines the port type using the keyword **interface**, instead of a using the name of a specific interface type. The syntax is:

```
module <module_name> (interface <port_name>);
```

When the module is instantiated, any interface can be connected to the generic interface port. This provides flexibility in that the same module can be used in multiple ways, with different interfaces connected to the module. In the following example, module RAM is defined with a generic interface port:

```
module RAM (interface pins,
            input     clock);
  ...
endmodule
```

10.3.3 Synthesis guidelines

Both styles of connecting an interface to a module are synthesizable.

10.4 Instantiating and connecting interfaces

interfaces are instantiated the same way as modules

An instance of an interface is connected to a port of a module instance using a port connection, just as a discrete net would be connected to a port of a module instance. This requires that both the interface and the modules to which it is connected be instantiated.

The syntax for an interface instance is the same as for a module instance. If the definition of the interface has ports, then signals can be connected to the interface instance, using either the port order connection style or the named port connection style, just as with a module instance.

Interface connection rules

NOTE It is illegal to leave an interface port unconnected.

interface ports must be connected

A module **input**, **output** or **inout** port can be left unconnected on a module instance. This is not the case for an interface port. A port that is declared as an interface, whether generic or explicit, must be connected to an interface instance or another interface port. An error will occur if an interface port is left unconnected.

On a module instance, a port that has been declared as an interface type must be connected to an interface instance, or another interface port that is higher up in the hierarchy. If a port declaration has an explicitly named interface type, then it *must* be connected to an interface instance of the identical type. If a port declaration has a generic interface type, then it can be connected to an interface instance of any type.

The SystemVerilog `.name` and `.*` port connection styles can also be used with interface instances, as is illustrated in example 10-4 on page 275. These port connection styles are discussed in section 9.4 on page 233.

Interfaces connected to interface instances

the port of an interface can connect to another interface

A port of an interface can also be defined as an interface. This capability allows one interface to be connected to another interface. The main bus of a design, for example might have one or more sub-busses. Both the main bus and its sub-busses can be modeled as interfaces. The sub-bus interfaces can be represented as ports of the main interface.

10.5 Referencing signals within an interface

signals in an interface are referenced using the port name

Within a module that has an interface port, the signals inside the interface must be accessed using the port name, using the following syntax:

```
<port_name>.<internal_interface_signal_name>
```

In example 10-3 on page 274, the interface definition for `main_bus` contains declarations for `clock` and `resetN`. Module `slave` has an interface port, with the port name of `bus`. The `slave` model can access the `clock` variable within the interface by referencing it as `bus.clock`. For example:

```
always @(posedge bus.clock, negedge bus.resetN)
    ...
```

Example 10-5 lists partial code for module `slave`. The model contains several references to signals within the `main_bus` interface.

Example 10-5: Referencing signals within an interface

```
module slave (
    // main_bus interface port
    main_bus bus
    // other ports
  );
  // internal signals
  logic [15:0] slave_data, slave_address;
  logic [15:0] operand_A, operand_B;
  logic        mem_select, read, write;

  assign bus.address = mem_select? slave_address: 'z;
  assign bus.data = bus.slave_ready? slave_data: 'z;

  enum logic [4:0] {RESET    = 5'b00001,
                    START    = 5'b00010,
                    REQ_DATA = 5'b00100,
                    EXECUTE  = 5'b01000,
                    DONE     = 5'b10000} State, NextState;

  always_ff @(posedge bus.clock, negedge bus.resetN) begin: FSM
    if (!bus.resetN) State <= START;
    else             State <= NextState;
  end

  always_comb begin : FSM_decode
    unique case (State)
      START:    if (!bus.slave_request) begin
                  bus.bus_request = 0;
                  NextState = State;
                end
                else begin
                  operand_A       = bus.data;
                  slave_address   = bus.address;
                  bus.bus_request = 1;
                  NextState       = REQ_DATA;
                end
      ...  // decode other states
    endcase
  end: FSM_decode
endmodule
```

TIP

Use short names for the names of interface ports.

Since signals within an interface are accessed by prepending the interface port name to the signal name, it is convenient to use short names for interface port names. This keeps the reference to the interface signal name short and easy to read. The names within the interface can be descriptive and meaningful, as within any Verilog module.

10.6 Interface modports

Interfaces provide a practical and straightforward way to simplify connections between modules. However, each module connected to an interface may need to see a slightly different view of the connections within the interface. For example, to a slave on a bus, an `interrupt_request` signal might be an output from the slave, whereas to a processor on the same bus, `interrupt_request` would be an input.

modports define interface connections from the perspective of the module

SystemVerilog interfaces provide a means to define different views of the interface signals that each module sees on its interface port. The definition is made within the interface, using the **`modport`** keyword. ***Modport*** is an abbreviation for ***module port***. A modport definition describes the module ports that are represented by the interface. An interface can have any number of modport definitions, each describing how one or more other modules view the signals within the interface.

A modport defines the port direction that the module sees for the signals in the interface. Examples of two **`modport`** declarations are:

```
interface chip_bus (input logic clock, resetN);
  logic interrupt_request, grant, ready;
  logic [31:0] address;
  wire  [63:0] data;

  modport master (input  interrupt_request,
                  input  address,
                  output grant, ready,
                  inout  data,
                  input  clock, resetN);
```

```
    modport slave  (output interrupt_request,
                    output address,
                    input  grant, ready,
                    inout  data,
                    input  clock, resetN);
endinterface
```

The modport definitions do not contain vector sizes or types. This information is defined as part of the signal type declarations in the interface. The modport declaration only defines whether the connecting module sees a signal as an **`input`**, **`output`**, bidirectional **`inout`**, or **`ref`** port.

10.6.1 Specifying which modport view to use

SystemVerilog provides two methods for specifying which modport view a module interface port should use:

- As part of the interface connection to a module instance
- As part of the module port declaration in the module definition

Both of these specification styles are synthesizable.

Selecting the modport in the module instance

the modport can be selected in the module instance

When a module is instantiated and an instance of an interface is connected to a module instance port, the specific modport of the interface can be specified. The connection to the modport is specified as:

```
<interface_instance_name>.<modport_name>
```

For example:

```
chip_bus  bus;  // instance of an interface

primary i1 (bus.master);  // use master modport
```

The following code snippet illustrates connecting two modules together with an interface called `chip_bus`. The module called `primary` is connected to the `master` view of the interface, and the module called `secondary` is connected to the `slave` view of the same interface:

Example 10-6: Selecting which modport to use at the module instance

```
interface chip_bus (input logic clock, resetN);
  modport master (...);
  modport slave  (...);
endinterface

module primary   (interface  pins);  // generic interface port
  ...
endmodule

module secondary (chip_bus pins);    // specific interface port
  ...
endmodule

module chip (input logic clock, resetN);

  chip_bus  bus (clock, resetN);  // instance of an interface

  primary   i1 (bus.master);  // use the master modport view

  secondary i2 (bus.slave);   // use the slave modport view

endmodule
```

When the modport to be used is specified in the module instance, the module definition can use either a generic interface port type or an explicitly named interface port type, as discussed in sections 10.3.2 on page 278, and 10.3.1 on page 277. The preceding example shows a generic interface port definition for `primary` module, and an explicitly named port type for `secondary` module.

Selecting the modport in the module port declaration

the modport can be selected in the module definition

The specific modport of an interface to be used can be specified directly as part of the module port declaration. The modport to be connected to the interface is specified as:

```
<interface_name>.<modport_name>
```

For example:

```
module secondary (chip_bus.slave pins);
  ...
endmodule
```

The explicit interface name must be specified in the port type when the modport to be used is specified as part of the module definition. The instance of the module simply connects an instance of the interface to the module port, without specifying the name of a modport.

The following code snippet shows a more complete context of specifying which modport is to be connected to a module, as part of the definition of the module.

Example 10-7: Selecting which modport to use at the module definition

```
interface chip_bus (input logic clock, resetN);
  modport master (...);
  modport slave  (...);
endinterface

module primary   (chip_bus.master pins);  // use master modport
  ...
endmodule

module secondary (chip_bus.slave pins);  // use slave modport
  ...
endmodule

module chip (input logic clock, resetN);

  chip_bus  bus (clock, resetN);  // instance of an interface

  primary   i1 (bus);  // will use the master modport view

  secondary i2 (bus);  // will use the slave modport view

endmodule
```

NOTE A modport can be selected in either the module instance or the module definition, but not both.

The modport view that a module is to use can only be specified in one place, either on the module instance or as part of the module definition. It is an error to select which modport is to be used in both places.

Connecting to interfaces without specifying a modport

when no modport is used, nets are bidirectional, and variables are references

Even when an interface is defined with modports, modules can still be connected to the complete interface, without specifying a specific modport. However, the port directions of signals within an interface are only defined as part of a modport view. When no modport is specified as part of the connection to the interface, all nets in the interface are assumed to have a bidirectional **inout** direction, and all variables in the interface are assumed to be of type **ref**. A **ref** port passes values by reference, rather than by copy. This allows the module to access the variable in the interface, rather than a copy of the variable. Module reference ports are covered in section 9.7 on page 255.

Synthesis considerations

Synthesis supports both styles of specifying which modport is to be used with a module. Most synthesis compilers will expand the interface port of a module into the individual ports represented in the modport definition. The following code snippets show the pre- and post-synthesis module definitions of a module using an interface with modports.

Pre-synthesis model, with an interface port:

```
module primary (chip_bus.master pins);
  ...
endmodule

interface chip_bus (input wire clock, resetN);
  logic interrupt_request, grant, ready;
  logic [31:0] address;
  wire  [63:0] data;

  modport master (input  interrupt_request,
                  input  address,
                  output grant, ready,
                  inout  data,
                  input  clock, resetN);
endinterface
```

Post-synthesis model:

```
module primary (interrupt_request, address,
```

```
                   grant, ready, data,
                   clock, resetN);
    input  interrupt_request,
    input  [31:0] address,
    output grant, ready,
    inout  [63:0] data,
    input  clock, resetN);
    ... // synthesized model functionality
  endmodule
```

Synthesis compilers might create different names for the separate ports than those shown in the example above.

If no modport is specified when the model is synthesized, then all signals within the interface become bidirectional **`inout`** ports on the synthesized module.

10.6.2 Using modports to define different sets of connections

In a more complex interface between several different modules, it may be that not every module needs to see the same set of signals within the interface. Modports make it possible to create a customized view of the interface for each module connected.

Restricting module access to interface signals

modports limit access to the contents of an interface

A module can only directly access the signals listed in its **`modport`** definition. This makes it possible to have some signals within the interface completely hidden from view to certain modules. For example, the interface might contain a net called `test_clock` that is only used by modules connected to the interface through the `master` modport, and not by modules connected through the `slave` modport.

A **`modport`** does not prohibit the use of a full hierarchy path to access any object in an interface. However, full hierarchy paths are not synthesizable, and are primarily used for verification.

It is also possible to have internal signals within an interface that are not visible through any of the modport views. These internal signals might be used by protocol checkers or other functionality contained within the interface, as discussed later in this chapter. If a module is connected to the interface without specifying a modport, the module will have access to all signals defined in the interface.

Example 10-8 adds modports to the main_bus interface example. The processor module, the slave module and the RAM module all use different modports within the main_bus interface, and the signals within the interface that can be accessed by each of these modules are different. The test block is connected to the main_bus without specifying a modport, giving the test block complete, unrestricted access to all signals within the interface.

Example 10-8: A simple design using an interface with modports

```
/****************** Interface Definitions ******************/
interface main_bus (input logic clock, resetN, test_mode);
  wire  [15:0] data;
  wire  [15:0] address;
  logic [ 7:0] slave_instruction;
  logic        slave_request;
  logic        bus_grant;
  logic        bus_request;
  logic        slave_ready;
  logic        data_ready;
  logic        mem_read;
  logic        mem_write;

  modport master (inout  data,
                  output address,
                  output slave_instruction,
                  output slave_request,
                  output bus_grant,
                  output mem_read,
                  output mem_write,
                  input  bus_request,
                  input  slave_ready,
                  input  data_ready,
                  input  clock,
                  input  resetN,
                  input  test_mode
                 );

  modport slave  (inout  data,
                  inout  address,
                  output mem_read,
                  output mem_write,
                  output bus_request,
                  output slave_ready,
                  input  slave_instruction,
                  input  slave_request,
                  input  bus_grant,
                  input  data_ready,
```

```
                 input  clock,
                 input  resetN
                );

  modport mem    (inout  data,
                  output data_ready,
                  input  address,
                  input  mem_read,
                  input  mem_write
                 );
endinterface

/********************** Top-level Netlist ********************/
module top (input logic clock, resetN, test_mode);
  logic [15:0] program_address, jump_address;
  logic [ 7:0] instruction, next_instruction, data_b;

  main_bus        bus      ( .* );  // instance of an interface

  processor       proc1    (.bus(bus.master), .* );
  slave           slave1   (.bus(bus.slave),  .* );
  instruction_reg ir       ( .* );
  test_generator  test_gen (.bus(bus),        .* );
  dual_port_ram   ram      (.bus(bus.mem),    .* ,
                            .data_b(next_instruction) );
endmodule

/*** remainder of netlist and module definitions are not  ***/
/*** listed — they are similar to example 10-2, but       ***/
/*** clock and resetN do not need to be passed to each    ***/
/*** module instance as discrete ports.                   ***/
```

10.7 Using tasks and functions in interfaces

interfaces can contain functionality

Interfaces can encapsulate the full details of the communication protocol between modules. For instance, the `main_bus` protocol in the previous example includes handshaking signals between the master processor and the slave processor. In regular Verilog, the master processor module would need to contain the procedural code to assert and de-assert its handshake signals at the appropriate time, and to monitor the slave handshake inputs. Conversely, the slave processor would need to contain the procedural code to assert and de-assert its handshake signals, and to monitor the handshake inputs coming from the master processor or the RAM.

Describing the bus protocol within each module that uses a bus leads to duplicated code. If any change needs to be made to the bus protocol, the code for the protocol must be changed in each and every module that shares the bus.

10.7.1 Interface methods

an interface method is a task or function

SystemVerilog allows tasks and functions to be declared within an interface. These tasks and functions are referred to as ***interface methods***. A task or function that is defined within an interface is written using the same syntax as if it had been within a module, and can contain the same types of statements as within a module. These interface methods can operate on any signals within the interface. Values can be passed in to interface methods from outside the interface as input arguments. Values can be written back from interface methods as output arguments or function returns.

methods encapsulate functionality in one place

Interface methods offer several advantages for modeling large designs. Using interface methods, the details of communication from one module to another can be moved to the interface. The code for communicating between modules does not need to be replicated in each module. Instead, the code is only written once, as interface methods, and shared by each module connected using the interface. Within each module, the interface methods are called, instead of implementing the communication protocol functionality within the module. Thus, an interface can be used not only to encapsulate the data connecting modules, but also the communication protocols between the modules.

10.7.2 Importing interface methods

modules can import interface methods

If the interface is connected via a modport, the method must be specified using the **`import`** keyword. The import definition is specified within the interface, as part of a modport definition. Modports specify interface information from the perspective of the module. Hence, an import declaration within a modport indicates that the module is importing the task or function.

The import declaration can be used in two ways:

- Import using just the task or function name
- Import using a full prototype of the task or function

Import using a task or function name

a method can be imported using just its name

The simplest form of importing a task or function is to simply specify the name of the task or function. The basic syntax is:

```
modport ( import <task_function_name> );
```

An example of using this style is:

```
modport in  (import Read,
             import parity_gen,
             input  clock, resetN );
```

Import using a task or function prototype

a method can be imported using a full prototype

The second style of an **import** declaration is to specify a full prototype of the task or function arguments. This style requires that the keyword **task** or **function** follow the **import** keyword. It also requires that the task or function name be followed by a set of parentheses, which contain the formal arguments of the task or funciton. The basic syntax of this style of import declarations is:

```
modport (import task <task_name>(<task_formal_arguments) );

modport (import function <function_name> (<formal_args>) );
```

For example:

```
modport in (import task Read
                      (input  [63:0] data,
                       output [31:0] address),
            import function parity_gen
                      (input [63:0] data),
            input  clock, resetN);
```

A full prototype can serve to document the arguments of the task or function directly as part of the modport declaration. This additional code documentation can be convenient if the actual task or function is defined in a package, and therefore the definition is not in the package source code for easy visual reference.

The full prototype is required when the task or function has been exported from another module (explained in section 10.7.4 on page 293), or when a function has been externally defined using SystemVerilog's Direct Programming Interface (not covered in this book).

Calling imported interface methods

imported methods are accessed using the port name

Importing a task or function through a modport gives the module using the modport access to the interface method. The task or function is called by prepending the interface port name to the task or function name, the same as when a signal within an interface is accessed.

```
<interface_port_name>.<method_name>
```

Alternate methods within interfaces

interfaces can contain alternate methods

Modports provide a way to use different methods and protocols within the same interface. The interface can contain a variety of different methods, each using different protocols or types.

The following code snippet example illustrates an interface called `math_bus`. Within the interface, different read methods are defined, which retrieve either integer data or floating point data through an interface. Two modules are defined, called `integer_math_unit` and `floating_point_unit`, both of which use the same `math_bus` interface. Each module will access different types of information, based on the modport used in the instantiation of the module.

Example 10-9: Using modports to select alternate methods within an interface

```
interface math_bus (input logic clock, resetN);
  int   a_int, b_int, result_int;
  real  a_real, b_real, result_real;
  ...
  task IntegerRead (output int   a_int, b_int);
    ... // do handshaking to fetch a and b values
  endtask

  task FloatingPointRead (output real  a_real, b_real);
    ... // do handshaking to fetch a and b values
  endtask

  modport int_io (import IntegerRead,
                  input  clock, resetN,
                  output result_int);

  modport fp_io  (import FloatingPointRead,
                  input  clock, resetN,
                  output result_real);
endinterface
```

```
/********************* Top-level Netlist ******************/
module dual_mu (input logic clock, resetN);
  math_bus bus_a (clock, resetN); // 1st instance of interface
  math_bus bus_b (clock, resetN); // 2nd instance of interface

  integer_math_unit   i1 (bus_a.int_io);
    // connect to interface using integer types

  floating_point_unit i2 (bus_b.fp_io);
    // connect to interface using real types
endmodule

/******************** Module Definitions ******************/
module integer_math_unit (interface  io);

  int a_reg, b_reg;

  always @(posedge io.clock)
    begin
      io.IntegerRead(a_reg, b_reg);  // call method in
                                     // interface
      ... // process math operation
    end
endmodule

module floating_point_unit (interface  io);

  real a_reg, b_reg;

  always @(posedge io.clock)
    begin
      io.FloatingPointRead(a_reg, b_reg);  // call method in
                                           // interface
      ... // process math operation
    end
endmodule
```

10.7.3 Synthesis guidelines for interface methods

Modules that use tasks or functions imported from interfaces are synthesizable. Synthesis will infer a local copy of the imported task or function within the module. The post-synthesis version of the module will contain the logic of the task or functions; it will no longer look to the interface for that functionality.

NOTE Imported tasks or functions must be declared as automatic and not contain static declarations in order to be synthesized.

An automatic task or function allocates new storage each time it is called. When a module calls an imported task or function, a new copy is allocated. This allows synthesis to treat the task or function as if were a local copy within the module.

10.7.4 Exporting tasks and functions

modules can export methods into an interface

SystemVerilog interfaces and modports provide a mechanism to define a task or function in one module, and then export the task or function through an interface to other modules.

NOTE Exporting tasks and functions is not synthesizable.

exported methods are not synthesizable

Exporting tasks or functions into an interface is not synthesizable. This modeling style should be reserved for abstract models that are not intended to be synthesized.

An export declaration in an interface modport does not require a full prototype of the task or function arguments. Only the task or function name needs to be listed in the modport declaration.

If an exported task or function has default values for any of its formal arguments, then each import declaration of the task or function must have a complete prototype of the task/function arguments. A full prototype for the import declaration is also required if the task or function call uses named argument passing instead of passing by position.

The code fragments in example 10-10 show a function called `check` that is declared in module `CPU`. The function is exported from the CPU through the `master` modport of the `chip_bus` interface. The same function is imported into any modules that use the `slave` modport of the interface. To any module connected to the `slave` modport, the `check` function appears to be part of the interface, just like any other function imported from an interface. Modules using the `slave` modport do not need to know the actual location of the `check` function definition.

Example 10-10: Exporting a function from a module through an interface modport

```
interface chip_bus (input logic clock, resetN);
  logic        request, grant, ready;
  logic [63:0] address, data;

  modport master (output request, ...
                  export check );

  modport slave  (input  request, ...
                  import check );
endinterface

module CPU (chip_bus.master io);

  function check (input parity, input [63:0] data);
    ...
  endfunction
   ...
endmodule
```

Exporting a task or function to the entire interface

The **export** declaration allows a module to export a task or function to an interface through a specific modport of the interface. A task or function can also be exported to an interface without using a modport. This is done by declaring an **extern** prototype of the task or function within the interface. For example:

Example 10-11: Exporting a function from a module into an interface

```
interface chip_bus (input logic clock, resetN);
  logic        request, grant, ready;
  logic [63:0] address, data;

  extern function check(input logic        parity,
                        input logic [63:0] data);

  modport master (output request, ...);

  modport slave  (input  request, ...
                  import function check
                                (input logic        parity,
```

```
                                        input logic [63:0] data) );
endinterface

module CPU (chip_bus.master io);

  function check (input logic parity, input logic [63:0] data);
    ...
  endfunction
   ...
endmodule
```

Restrictions on exporting tasks and functions

NOTE It is illegal to export the same function name from multiple instances of a module. It is legal, however, to export a task name from multiple instances, using an **`extern forkjoin`** declaration.

restrictions on exporting functions

SystemVerilog places a restriction on exporting functions through interfaces. It is illegal to export the same function name from two different modules, or two instances of the same module, into the same interface. For example, module `A` and module `B` cannot both export a function called `check` into the same interface.

restrictions on exporting tasks

SystemVerilog places a restriction on exporting tasks through interfaces. It is illegal to export the same task name from two different modules, or two instances of the same module, into the same interface, unless an **`extern forkjoin`** declaration is used. The multiple export of a task corresponds to a multiple response to a broadcast. Tasks can execute concurrently, each taking a different amount of time to execute statements, and each call returning different values through its outputs. The concurrent response of modules `A` and `B` containing a call to a task called `task1` is conceptually modeled by:

```
fork
  <hierarchical_name_of_module_A>.task1(q, r);
  <hierarchical_name_of_module_B>.task1(q, r);
join
```

extern forkjoin allows multiple instances of exported tasks

Because an interface should not contain the hierarchical names of the modules to which it is connected, the task is declared as

extern forkjoin, which infers the behavior of the **fork**...**join** block above. If the task contains outputs, it is the last instance of the task to finish that determines the final output value.

This construct can be useful for abstract, non-synthesizable transaction level models of busses that have slaves, where each slave determines its own response to broadcast signals (see example 12-2 on page 335 for an example). The **extern forkjoin** can also be used for configuration purposes, such as counting the number of modules connected to an interface. Each module would export the same task, name which increments a counter in the interface.

10.8 Using procedural blocks in interfaces

interfaces can contain protocol checkers and other functionality

In addition to methods (tasks and functions), interfaces can contain Verilog procedural blocks and continuous assignments. This allows an interface to contain functionality that can be described using **always**, **always_comb**, **always_ff**, **always_latch**, **initial** or **final** procedural blocks, and **assign** statements. An interface can also contain verification **program** blocks.

One usage of procedural blocks within interfaces is to facilitate verification of a design. One application of using procedural statements within an interface is to build protocol checkers into the interface. Each time modules pass values through the interface, the built-in protocol checkers can verify that the design protocols are being met. Examples of using procedural code within interfaces are presented in the companion book, *SystemVerilog for Verification*[1].

10.9 Reconfigurable interfaces

Interfaces can use parameter redefinition and generate statements, in the same way as modules. This allows interface models to be defined that can be reconfigured each time an interface is instantiated.

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

Parameterized interfaces

interfaces can use parameters, the same as modules

Parameters can be used in interfaces to make vector sizes and other declarations within the interface reconfigurable using Verilog's parameter redefinition constructs. SystemVerilog also adds the ability to parameterize types, which is covered in section 9.9 on page 260.

Example 10-12, below, adds parameters to example 10-9 on page 291 shown earlier, which uses different modports to pass either integer data or real data through the same interface. In this example, the variable types of the interface are parameterized, so that each instance of the interface can be configured to use integer or real types.

Example 10-12: Using parameters in an interface

```
interface math_bus #(parameter type DTYPE = int)
                    (input logic clock);

  DTYPE a, b, result; // parameterized types
  ...
  task Read (output DTYPE   a, b);
    ... // do handshaking to fetch a and b values
  endtask

  modport int_io (import Read,
                  input  clock,
                  output result);

  modport fp_io  (import Read,
                  input  clock,
                  output result);
endinterface

module top (input logic clock, resetN);
  math_bus                 bus_a(clock);  // use int data
  math_bus (#.DTYPE(real)) bus_b(clock);  // use real data

  integer_math_unit   i1 (bus_a.int_io);
    // connect to interface using integer types

  floating_point_unit i2 (bus_b.fp_io);
    // connect to interface using real types

endmodule // end of module top
```

The preceding example uses the Verilog-2001 style for declaring parameters within a module and for parameter redefinition. The older Verilog-1995 style of declaring parameters and doing parameter redefinition can also be used with interfaces.

Using generate blocks

interfaces can use generate blocks

The Verilog-2001 generate statement can also be used to create reconfigurable interfaces. Generate blocks can be used to replicate continuous assignment statements or procedural blocks within an interface any number of times.

10.10 Verification with interfaces

Using only Verilog-style module ports, without interfaces, a typical design and verification paradigm is to develop and test each module of a design, independent of other modules in the design. After each module is independently verified, the modules are connected together to test the communication between modules. If there is a problem with the communication protocols, it may be necessary to make design changes to multiple modules.

communication protocols can be verified before a design is modeled

Interfaces enable a different paradigm for verification. With interfaces, the communication channels can be developed as interfaces independently from other modules. Since an interface can contain methods for the communication protocols, the interface can be tested and verified independent of the rest of the design. Modules that use the interface can be written knowing that the communication between modules has already been verified.

Verification of designs that use interfaces is covered in much greater detail in the companion book, *SystemVerilog for Verification*[1].

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

10.11 Summary

This chapter has presented one of more powerful additions to the Verilog language for modeling very large designs: interfaces. An interface encapsulates the communication between major blocks of a design. Using interfaces, the detailed and redundant module port and netlist declarations are greatly simplified. The details are moved to one modeling block, where they are defined once, instead of in many different modules. An interface can be defined globally, so it can be used by any module anywhere in the design hierarchy. An interface can also be defined to be local to one hierarchy scope, so that only that scope can use the interface.

Interfaces do more than provide a way to bundle signals together. The interface modport definition provides a simple yet powerful way to customize the interface for each module that it is connected to. The ability to incorporate methods (tasks and functions) and procedural code within an interface make it possible instrument and drive the simulation model in one convenient location.

Chapter 11
A Complete Design Modeled with SystemVerilog

This chapter brings together the many concepts presented in previous chapters of this book, and shows how the SystemVerilog enhancements to Verilog can be used to model large designs much more efficiently than with the standard Verilog HDL. The example presented in this chapter shows how SystemVerilog can be used to model at a much higher level of data abstraction than Verilog, and yet be fully synthesizable.

11.1 SystemVerilog ATM example

The design used as an example for this chapter is based upon an example from Janick Bergeron's Verification Guild[1]. The original example is a non-synthesizable behavioral model written in Verilog (using the Verilog-1995 standard). The example is a description of a quad Asynchronous Transfer Mode (ATM) user-to-network interface and forwarding node. For this book, this example has been modified in three significant ways. First, the code has been re-written in order to use many SystemVerilog constructs. Second, the non-synthesizable behavioral models have been rewritten using the SystemVerilog synthesizable subset. Third, the model has been

1. The Verification Guild is an independent e-mail newsletter and moderated discussion forum on hardware verification. Information on the Verification Guild example used as a basis for the example in this chapter can be found at http://verificationguild.com/dload/vg_project/spec.pdf.

made configurable, so that it can be easily scaled from a 4x4 quad switch to a 16x16 switch, or any other desired configuration.

The example in this chapter illustrates how the use of SystemVerilog structures, unions, and arrays significantly simplifies the representation of complex design data. The use of interfaces and interface methods further simplifies the communication of complex data between the blocks of a design.

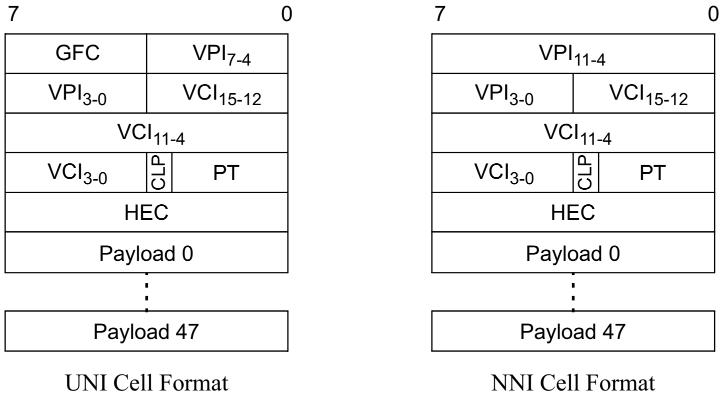
The SystemVerilog coding style used in this example also shows how the design can be automatically sized and configured from a single source. Using `+define` invocation options, the architecture of the design can be configured as an NxP port forwarding node, where N and P can be any positive value. Rather than producing a fixed 4x4 design, as was the case in the original Verilog-1995 example, this SystemVerilog version can produce a 128x128, 16x128, 128x16, or any other configuration imaginable. The sizing and instantiation of the module and data declarations is handled implicitly (including the relatively simple testbench used with this example).

11.2 Data abstraction

SystemVerilog allows the designer to raise the level of abstraction for the data representation. In Verilog, the type set is rather limited in comparison to SystemVerilog. What is needed is a set of types that reflects the nature of the design.

The two ATM formats used in this ATM design are the **`UNI`** format and the **`NNI`** format.

Figure 11-1: UNI and NNI cell formats



An ATM cell simply consists of 53 bytes of data. This can be modeled as an array of bytes in Verilog, but the meaning of those bytes within the cell is lost when modeled in this manner. Using packed structure definitions for the two different formats is easy in SystemVerilog, and makes each cell member easily identifiable:

UNI Cell Structure

```
typedef struct packed {
  logic         [ 3:0] GFC;
  logic         [ 7:0] VPI;
  logic         [15:0] VCI;
  logic                CLP;
  logic         [ 2:0] PT;
  logic         [ 7:0] HEC;
  logic [0:47]  [ 7:0] Payload;
} uniType;
```

NNI Cell Structure

```
typedef struct packed {
  logic         [11:0] VPI;
  logic         [15:0] VCI;
  logic                CLP;
  logic         [ 2:0] PT;
  logic         [ 7:0] HEC;
  logic [0:47]  [ 7:0] Payload;
} nniType;
```

An important advantage of this level of data abstraction is that the 53 byte array of data can now be easily treated as though it were either of these formats, or as a simple array of bytes. This can be done by using a packed union of the two data packet formats:

Union of UNI / NNI / byte stream

```
typedef union packed {
  uniType uni;
  nniType nni;
  logic [0:52] [7:0] Mem;
} ATMCellType;
```

When an object is declared of type `ATMCellType`, its members can be accessed as though it were either a `uniType` cell, or an `nniType` cell, depending upon which fields need to be accessed.

A useful extension to this abstract data representation is to use data tagging as part of the testbench. For either type of cell (UNI or NNI), the last 48 bytes of data are the payload, which is user defined. These fields can be used as part of the test procedures, in order to carry part of the test data through the switch. In this particular example, the payload can be used to record at which input port the data arrived, and what was its sequence in all packets arriving at that port. This is easily done by defining another structure, that is only used by the testbench:

Test view cell format (payload section)

```
typedef struct packed {
  logic [0:4 ] [7:0] Header;
  logic [0:3 ] [7:0] PortID;
  logic [0:3 ] [7:0] PacketID;
  logic [0:39] [7:0] Padding;
} tstType;
```

All 5 bytes of the UNI/NNI header are encapsulated in a single field called `Header`. The fields that are used for the data tagging are the `PortID` and `PacketID` fields, which form part of the payload for the UNI/NNI ATM cells. This third abstract representation of the 53 bytes of data can be added to the packed union.

Union of UNI / NNI / test view / byte stream

```
typedef union packed {
  uniType uni;
  nniType nni;
  tstType tst;
  logic [0:52] [7:0] Mem;
} ATMCellType;
```

The 53 bytes of data can now be easily configured in four different ways:

- as a UNI cell
- as an NNI cell
- as a testbench tagged packet
- as an array of 53 bytes of data

Because the array, union, and structures are packed, the mapping of the corresponding bits are guaranteed when data is written using one format, and read in another format.

11.3 Interface encapsulation

The example in this chapter is based on the UTOPIA interface specifications from the ATM Forum Technical Committee[1]. This interface has been encapsulated in a SystemVerilog interface definition called `Utopia`. This definition contains the signals of the interface, an instance of an `ATMCellType` (described above), a set of modports (indicating dataflow direction), and a nested interface called `Method`, which is an instance of `UtopiaMethod`.

The nested `UtopiaMethod` interface contains the testbench transaction level interface routines, and is not synthesizable. By separating it from the rest of the interface, it does not clutter the design. The instance of this testbench interface can easily be excluded from synthesis using synthesis off/on pragmas.

1. ATM Forum Technical Committee, UTOPIA Specification Level 1, Version 2.01, Document af-phy-0017.000, March 21, 1994 (available at http://www.mfaforum.org/ftp/pub/approved-specs/af-phy-0017.000.pdf) and ATM Forum Technical Committee, UTOPIA Level 2, Version 1.0, Document af-phy-0039.000, June 1995 (available at http://www.mfaforum.org/ftp/pub/approved-specs/af-phy-0039.000.pdf).

Example 11-1: Utopia ATM interface, modeled as a SystemVerilog interface

```
interface Utopia;
  parameter int IfWidth = 8;

  logic clk_in;
  logic clk_out;
  logic [IfWidth-1:0] data;
  logic soc;
  logic en;
  logic clav;
  logic valid;
  logic ready;
  logic reset;
  logic selected;

  ATMCellType ATMcell;  // union of structures for ATM cells

  modport TopReceive (
    input  clk_in, data, soc, clav, ready, reset,
    output clk_out, en, ATMcell, valid );

  modport TopTransmit (
    input  clk_in, clav, ATMcell, valid, reset,
    output clk_out, data, soc, en, ready );

  modport CoreReceive (
    input  clk_in, data, soc, clav, ready, reset,
    output clk_out, en, ATMcell, valid );

  modport CoreTransmit (
    input  clk_in, clav, ATMcell, valid, reset,
    output clk_out, data, soc, en, ready );

  `ifndef SYNTHESIS // synthesis ignores this code
    UtopiaMethod Method ();  // interface with testing methods
  `endif
endinterface
```

In addition to the `Utopia` interface, there is a management interface, called `CPU`, and a look-up table interface, called `LookupTable`. The `LookupTable` interface is used in the core of the device called `squat`, in order to provide a latch-based read/write look-up table. The storage variable type of this look-up table is defined through a type parameter called `dType`, which means it can be instantiated to store any built-in or user-defined type (as will be shown later).

Example 11-2: Cell rewriting and forwarding configuration

```
typedef struct packed {
  logic [`TxPorts-1:0] FWD;
  logic [11:0] VPI;
} CellCfgType;

interface CPU;
  logic         BusMode;
  logic [11:0]  Addr;
  logic         Sel;
  CellCfgType   DataIn;
  CellCfgType   DataOut;
  logic         Rd_DS;
  logic         Wr_RW;
  logic         Rdy_Dtack;

  modport Peripheral (
    input  BusMode, Addr, Sel, DataIn, Rd_DS, Wr_RW,
    output DataOut, Rdy_Dtack
  );

  `ifndef SYNTHESIS // synthesis ignores this code
    CPUMethod Method ();  // interface with testing methods
  `endif
endinterface

interface LookupTable;
  parameter int  Asize  = 8;
  parameter int  Arange = 1<<Asize;
  parameter type dType  = logic;

  dType Mem [0:Arange-1];

  // Function to perform write
  function void write (input [Asize-1:0] addr,
                       input dType data );
    Mem[addr] = data;
  endfunction

  // Function to perform read
  function dType read (input logic [Asize-1:0] addr);
    return (Mem[addr]);
  endfunction
endinterface
```

All the above definitions are contained in a file called `definitions.sv`, which is guarded as follows:

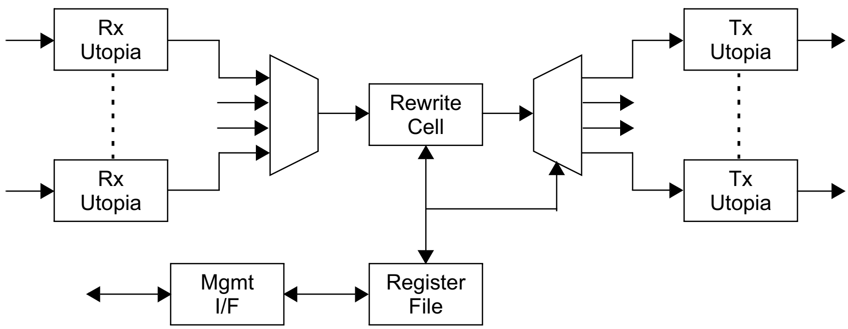
```
`ifndef _INCL_DEFINITIONS
`define _INCL_DEFINITIONS
  ...
`endif  // _INCL_DEFINITIONS
```

The conditional compilation guard allows the `definitions.sv` file to be included in multiple files without producing an error when multiple files are compiled at the same time.

11.4 Design top level: squat

The top level of the design is called `squat`. This module can process an array of receiver and transmitter `Utopia` interfaces, and provide a programmable `CPU` interface.

Figure 11-2: Design top-level structural diagram



The number of Utopia Receive interfaces is defined by a module parameter called `NumRx`, and the number of Utopia Transmit interfaces is defined by a module parameter called `NumTx`.

An instance of the interface `LookupTable` uses the user-defined type `CellCfgType` as the storage type `dType`. The `LookupTable` interface is written to by an **`always_latch`** block which, given a write condition, calls the method `lut.write`, which is the `write` method in the interface `LookupTable`.

The same interface is read from an **`always_comb`** block that, given a read condition, calls the method `lut.read`, which is the `read` method in the interface `LookupTable`.

Generate blocks are used to iterate across the number of Utopia Receive and Transmit interfaces, connecting the interfaces to generated instances of utopia receive and transmit modules respectively.

The `rst` reset input is synchronized to the clock, in order to remove possible design race conditions.

A state variable `SquatState` in the `squat` module is defined using an enumerated type, followed by a variable of that type. The width of the variable is constrained by a range which is used during synthesis for register sizing.

```
typedef enum logic [0:1] {
  wait_rx_valid,
  wait_rx_not_valid,
  wait_tx_ready,
  wait_tx_not_ready } StateType;
StateType SquatState;
```

This variable is used to store the state of the machine when processing incoming port packets (processed by utopia receive modules), prior to transmit (via utopia transmit modules). The state machine uses a round robin indicator to balance the precedence of incoming packets, which ensures each input port has equal priority for being serviced by the forwarding routine.

Example 11-3: ATM squat top-level module

```
`include "definitions.sv"

module squat
  #(parameter int NumRx = 4, parameter int NumTx = 4)
   (// NumRx x Level 1 Utopia ATM layer Rx Interfaces
    Utopia /* .TopReceive */ Rx[0:NumRx-1],

    // NumTx x Level 1 Utopia ATM layer Tx Interfaces
    Utopia /* .TopTransmit */ Tx[0:NumTx-1],

    // Utopia Level 2 parallel management interface
    // Intel-style Utopia parallel management interface
    CPU.Peripheral mif,
```

```
    // Miscellaneous control interfaces
    input logic rst, clk
   );

  // Register file
  LookupTable #(.Asize(8), .dType(CellCfgType)) lut();

  //
  // Hardware reset
  //
  logic reset;
  always_ff @(posedge clk) begin
    reset <= rst;
  end

  const logic [2:0] WriteCycle = 3'b010;
  const logic [2:0] ReadCycle = 3'b001;
  always_latch begin  // configure look-up table
    if (mif.BusMode == 1'b1) begin
      unique case ({mif.Sel, mif.Rd_DS, mif.Wr_RW})
        WriteCycle: lut.write(mif.Addr, mif.DataIn);
      endcase
    end
  end

  always_comb begin
    mif.Rdy_Dtack <= 1'bz;
    mif.DataOut <= 8'hzz;
    if (mif.BusMode == 1'b1) begin
      unique case ({mif.Sel, mif.Rd_DS, mif.Wr_RW})
        WriteCycle: mif.Rdy_Dtack <= 1'b0;
        ReadCycle: begin
                     mif.Rdy_Dtack <= 1'b0;
                     mif.DataOut <= lut.read(mif.Addr);
                   end
      endcase
    end
  end

  //
  // ATM-layer Utopia interface receivers
  //
  genvar RxIter;
  generate
    for (RxIter=0; RxIter<NumRx; RxIter+=1) begin: RxGen
      assign Rx[RxIter].clk_in = clk;
      assign Rx[RxIter].reset  = reset;
      utopia1_atm_rx atm_rx(Rx[RxIter].CoreReceive);
```

```
    end
  endgenerate

  //
  // ATM-layer Utopia interface transmitters
  //
  genvar TxIter;
  generate
    for (TxIter=0; TxIter<NumTx; TxIter+=1) begin: TxGen
      assign Tx[TxIter].clk_in = clk;
      assign Tx[TxIter].reset  = reset;
      utopia1_atm_tx atm_tx(Tx[TxIter].CoreTransmit);
    end
  endgenerate

  //
  // Function to compute the HEC value
  //
  function logic [7:0] hec (input logic [31:0] hdr);
    logic [7:0] syndrom[0:255];
    logic [7:0] RtnCode;
    logic [7:0] sndrm;

    // Generate the CRC-8 syndrom table
    for (int unsigned i=0; i<256; i+=1) begin
      sndrm = i;
      repeat (8) begin
      if (sndrm[7] == 1'b1)
        sndrm = (sndrm << 1) ^ 8'h07;
      else
        sndrm = sndrm << 1;
      end
      syndrom[i] = sndrm;
    end

    RtnCode = 8'h00;
    repeat (4) begin
      RtnCode = syndrom[RtnCode ^ hdr[31:24]];
      hdr = hdr << 8;
    end
    RtnCode = RtnCode ^ 8'h55;
    return RtnCode;
  endfunction

  //
  // Rewriting and forwarding process
  //
```

```
logic [0:NumTx-1] forward;

typedef enum logic [0:1] {wait_rx_valid,
                          wait_rx_not_valid,
                          wait_tx_ready,
                          wait_tx_not_ready } StateType;
StateType SquatState;

logic [0:NumTx-1] Txvalid;
logic [0:NumTx-1] Txready;
logic [0:NumTx-1] Txsel_in;
logic [0:NumTx-1] Txsel_out;
logic [0:NumRx-1] Rxvalid;
logic [0:NumRx-1] Rxready;
logic [0:NumRx-1] RoundRobin;
ATMCellType [0:NumRx-1] RxATMcell;
ATMCellType [0:NumTx-1] TxATMcell;

generate
  for (TxIter=0; TxIter<NumTx; TxIter+=1) begin: GenTx
    assign Tx[TxIter].valid    = Txvalid[TxIter];
    assign Txready[TxIter]     = Tx[TxIter].ready;
    assign Txsel_in[TxIter]    = Tx[TxIter].selected;
    assign Tx[TxIter].selected = Txsel_out[TxIter];
    assign Tx[TxIter].ATMcell  = TxATMcell[TxIter];
  end
endgenerate
generate
  for (RxIter=0; RxIter<NumRx; RxIter+=1) begin: GenRx
    assign Rxvalid[RxIter]   = Rx[RxIter].valid;
    assign Rx[RxIter].ready  = Rxready[RxIter];
    assign RxATMcell[RxIter] = Rx[RxIter].ATMcell;
  end
endgenerate

ATMCellType ATMcell;
always_ff @(posedge clk, posedge reset) begin: FSM
  logic breakVar;
  if (reset) begin: reset_logic
    Rxready <= '1;
    Txvalid <= '0;
    Txsel_out <= '0;
    SquatState <= wait_rx_valid;
    forward <= 0;
    RoundRobin = 1;
  end: reset_logic
  else begin: FSM_sequencer
    unique case (SquatState)
```

```
wait_rx_valid: begin: rx_valid_state
  Rxready <= '1;
  breakVar = 1;
  for (int j=0; j<NumRx; j+=1) begin: loop1
    for (int i=0; i<NumRx; i+=1) begin: loop2
      if (Rxvalid[i] && RoundRobin[i] && breakVar)
        begin: match
          ATMcell <= RxATMcell[i];
          Rxready[i] <= 0;
          SquatState <= wait_rx_not_valid;
          breakVar = 0;
        end: match
    end: loop2
    if (breakVar)
      RoundRobin={RoundRobin[1:$bits(RoundRobin)-1],
                  RoundRobin[0]};
  end: loop1
end: rx_valid_state

wait_rx_not_valid: begin: rx_not_valid_state
  if (ATMcell.uni.HEC != hec(ATMcell.Mem[0:3])) begin
    SquatState <= wait_rx_valid;
    `ifndef SYNTHESIS // synthesis ignores this code
      $write("Bad HEC: ATMcell.uni.HEC(0x%h) != ",
        ATMcell.uni.HEC);
      $display("ATMcell.Mem[0:3](0x%h)",
        hec(ATMcell.Mem[0:3]));
    `endif
  end
  else begin
    // Get the forward ports & new VPI
    {forward, ATMcell.nni.VPI} <=
      lut.read(ATMcell.uni.VPI);
    // Recompute the HEC
    ATMcell.nni.HEC <= hec(ATMcell.Mem[0:3]);
    SquatState <= wait_tx_ready;
  end
end: rx_not_valid_state

wait_tx_ready: begin: tx_valid_state
  if (forward) begin
    for (int i=0; i<NumTx; i+=1) begin
      if (forward[i] && Txready[i]) begin
        TxATMcell[i] <= ATMcell;
        Txvalid[i] <= 1;
        Txsel_out[i] <= 1;
      end
    end
```

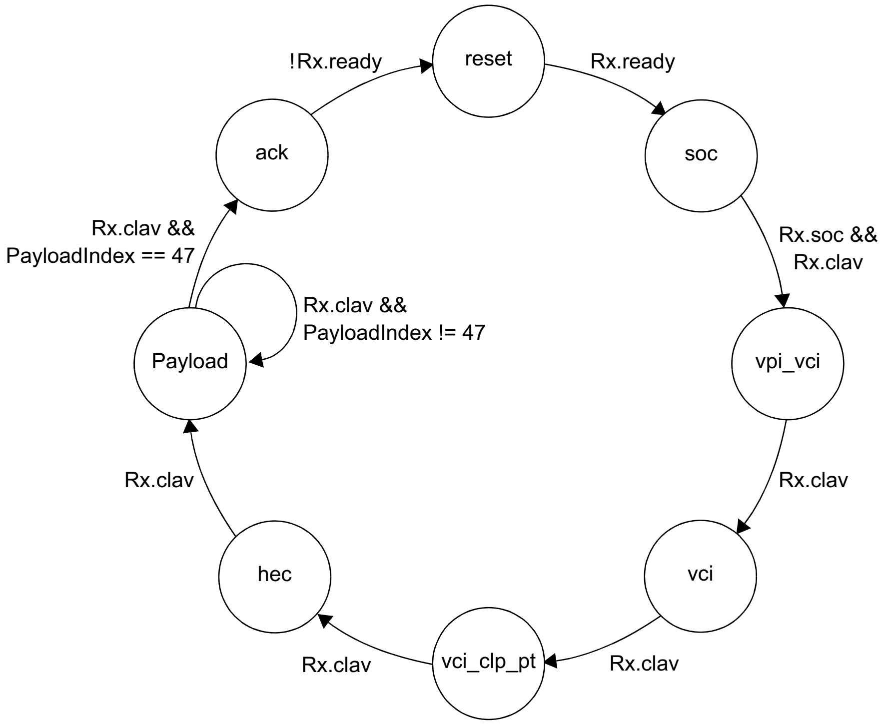
```
            SquatState <= wait_tx_not_ready;
          end
          else begin
            SquatState <= wait_rx_valid;
          end
        end: tx_valid_state

        wait_tx_not_ready: begin: tx_not_valid_state
          for (int i=0; i<NumTx; i+=1) begin
            if (forward[i] && !Txready[i] && Txsel_in[i]) begin
              Txvalid[i] <= 0;
              Txsel_out[i] <= 0;
              forward[i] <= 0;
            end
          end
          if (forward)
            SquatState <= wait_tx_ready;
          else
            SquatState <= wait_rx_valid;
        end: tx_not_valid_state

        default: begin: unknown_state
          SquatState <= wait_rx_valid;
          `ifndef SYNTHESIS // synthesis ignores this code
            $display("Unknown condition"); $finish();
          `endif
        end: unknown_state
      endcase
    end: FSM_sequencer
  end: FSM
endmodule
```

11.5 Receivers and transmitters

11.5.1 Receiver state machine

The receiver in the generate loop has a state machine with 8 states.

Figure 11-3: Receiver state flow diagram



Example 11-4: Utopia ATM receiver

```
module utopia1_atm_rx ( Utopia.CoreReceive Rx );

  // 25MHz Rx clk out
  assign Rx.clk_out = Rx.clk_in;

  // Listen to the interface, collecting byte.
  // A complete cell is then copied to the cell buffer
  logic [0:5] PayloadIndex;
  enum logic [0:2] { reset, soc, vpi_vci, vci, vci_clp_pt, hec,
                     payload, ack } UtopiaStatus;
```

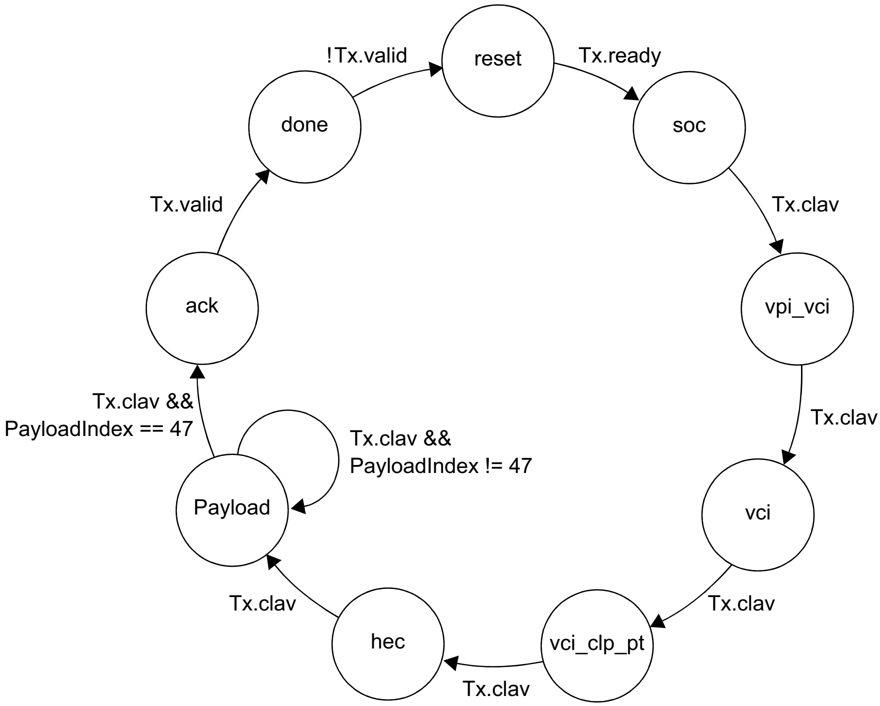
```
always_ff @(posedge Rx.clk_in, posedge Rx.reset) begin: FSM
  if (Rx.reset) begin
    Rx.valid <= 0;
    Rx.en <= 1;
    UtopiaStatus <= reset;
  end
  else begin: FSM_sequencer
    unique case (UtopiaStatus)
      reset: begin: reset_state
        if (Rx.ready) begin
          UtopiaStatus <= soc;
          Rx.en <= 0;
        end
      end: reset_state

      soc: begin: soc_state
        if (Rx.soc && Rx.clav) begin
          {Rx.ATMcell.uni.GFC,
           Rx.ATMcell.uni.VPI[7:4]} <= Rx.data;
          UtopiaStatus <= vpi_vci;
        end
      end: soc_state

      vpi_vci: begin: vpi_vci_state
        if (Rx.clav) begin
          {Rx.ATMcell.uni.VPI[3:0],
           Rx.ATMcell.uni.VCI[15:12]} <= Rx.data;
          UtopiaStatus <= vci;
        end
      end: vpi_vci_state

      vci: begin: vci_state
        if (Rx.clav) begin
          Rx.ATMcell.uni.VCI[11:4] <= Rx.data;
          UtopiaStatus <= vci_clp_pt;
        end
      end: vci_state

      vci_clp_pt: begin: vci_clp_pt_state
        if (Rx.clav) begin
          {Rx.ATMcell.uni.VCI[3:0], Rx.ATMcell.uni.CLP,
           Rx.ATMcell.uni.PT} <= Rx.data;
          UtopiaStatus <= hec;
        end
      end: vci_clp_pt_state

      hec: begin: hec_state
        if (Rx.clav) begin
          Rx.ATMcell.uni.HEC <= Rx.data;
          UtopiaStatus <= payload;
```

```
          PayloadIndex = 0;  /* Blocking Assignment, due to
                                blocking increment in
                                payload state */
        end
      end: hec_state

      payload: begin: payload_state
        if (Rx.clav) begin
          Rx.ATMcell.uni.Payload[PayloadIndex] <= Rx.data;
          if (PayloadIndex==47) begin
            UtopiaStatus <= ack;
            Rx.valid <= 1;
            Rx.en <= 1;
          end
          PayloadIndex++;
        end
      end: payload_state

      ack: begin: ack_state
        if (!Rx.ready) begin
          UtopiaStatus <= reset;
          Rx.valid <= 0;
        end
      end: ack_state

      default: UtopiaStatus <= reset;
    endcase
  end: FSM_sequencer
 end: FSM
endmodule
```

11.5.2 Transmitter state machine

The transmitter in the generate loop has a state machine with 9 states.

Figure 11-4: Transmitter state flow diagram



Example 11-5: Utopia ATM transmitter

```
module utopia1_atm_tx ( Utopia.CoreTransmit Tx );

  assign Tx.clk_out = Tx.clk_in;

  logic [0:5] PayloadIndex;  // 0 to 47
  enum logic [0:3] { reset, soc, vpi_vci, vci, vci_clp_pt, hec,
                     payload, ack, done } UtopiaStatus;
```

```
always_ff @(posedge Tx.clk_in, posedge Tx.reset) begin: FSM
  if (Tx.reset) begin
    Tx.soc <= 0;
    Tx.en <= 1;
    Tx.ready <= 1;
    UtopiaStatus <= reset;
  end
  else begin: FSM_sequencer
    unique case (UtopiaStatus)
      reset: begin: reset_state
        Tx.en <= 1;
        Tx.ready <= 1;
        if (Tx.valid) begin
          Tx.ready <= 0;
          UtopiaStatus <= soc;
        end
      end: reset_state

      soc: begin: soc_state
        if (Tx.clav) begin
          Tx.soc <= 1;
          Tx.data <= Tx.ATMcell.nni.VPI[11:4];
          UtopiaStatus <= vpi_vci;
        end
        Tx.en <= !Tx.clav;
      end: soc_state

      vpi_vci: begin: vpi_vci_state
        Tx.soc <= 0;
        if (Tx.clav) begin
          Tx.data <= {Tx.ATMcell.nni.VPI[3:0],
                      Tx.ATMcell.nni.VCI[15:12]};
          UtopiaStatus <= vci;
        end
        Tx.en <= !Tx.clav;
      end: vpi_vci_state

      vci: begin: vci_state
        if (Tx.clav) begin
          Tx.data <= Tx.ATMcell.nni.VCI[11:4];
          UtopiaStatus <= vci_clp_pt;
        end
        Tx.en <= !Tx.clav;
      end: vci_state

      vci_clp_pt: begin: vci_clp_pt_state
        if (Tx.clav) begin
          Tx.data <= {Tx.ATMcell.nni.VCI[3:0],
                      Tx.ATMcell.nni.CLP, Tx.ATMcell.nni.PT};
          UtopiaStatus <= hec;
```

```
        end
        Tx.en <= !Tx.clav;
      end: vci_clp_pt_state

      hec: begin: hec_state
        if (Tx.clav) begin
          Tx.data <= Tx.ATMcell.nni.HEC;
          UtopiaStatus <= payload;
          PayloadIndex = 0;
        end
        Tx.en <= !Tx.clav;
      end: hec_state

      payload: begin: payload_state
        if (Tx.clav) begin
          Tx.data <= Tx.ATMcell.nni.Payload[PayloadIndex];
          if (PayloadIndex==47) UtopiaStatus <= ack;
          PayloadIndex++;
        end
        Tx.en <= !Tx.clav;
      end: payload_state

      ack: begin: ack_state
        Tx.en <= 1;
        if (!Tx.valid) begin
          Tx.ready <= 1;
          UtopiaStatus <= done;
        end
      end: ack_state

      done: begin: done_state
        if (!Tx.valid) begin
          Tx.ready <= 0;
          UtopiaStatus <= reset;
        end
      end: done_state
    endcase
  end: FSM_sequencer
 end: FSM
endmodule
```

11.6 Testbench

The testbench send and receive methods for the Utopia interface are encapsulated in the `UtopiaMethod` interface.

Example 11-6: UtopiaMethod interface for encapsulating test methods

```
interface UtopiaMethod;
  task automatic Initialise ();
  endtask

  task automatic Send (input ATMCellType Pkt, input int PortID);
    static int PacketID;
    PacketID++;

    Pkt.tst.PortID = PortID;
    Pkt.tst.PacketID = PacketID;

    // iterate through bytes of packet, deasserting
    // Start Of Cell indicater
    @(negedge Utopia.clk_out);
    Utopia.clav <= 1;
    for (int i=0; i<=52; i++) begin
      // If not enabled, loop
      while (Utopia.en === 1'b1) @(negedge Utopia.clk_out);

      // Assert Start Of Cell indicater, assert enable,
      // send byte 0 (i==0)
      Utopia.soc <= (i==0) ? 1'b1 : 1'b0;
      Utopia.data <= Pkt.Mem[i];
      @(negedge Utopia.clk_out);
    end
    Utopia.data <= 8'bx;
    Utopia.clav <= 0;
  endtask

  task automatic Receive (input int PortID);
    ATMCellType Pkt;

    Utopia.clav = 1;
    while (Utopia.soc!==1'b1 && Utopia.en!==1'b0)
      @(negedge Utopia.clk_out);
    for (int i=0; i<=52; i++) begin
      // If not enabled, loop
      while (Utopia.en!==1'b0) @(negedge Utopia.clk_out);
      Pkt.Mem[i] = Utopia.data;
      @(negedge Utopia.clk_out);
    end
```

```
    Utopia.clav = 0;
    // Write Rxed data to logfile
    `ifdef verbose
  $write("Received packet at port %0d from port %0d PKT(%0d)\n",
      PortID, Pkt.tst.PortID, Pkt.tst.PacketID);
      //PortID, Pkt.nni.Payload[0], Pkt.nni.Payload[1:4]);
    `endif
  endtask
endinterface
```

The testbench `HostWrite` and `HostRead` methods for the `CPU` interface are encapsulated in the `CPUMethod` interface.

Example 11-7: CPUMethod interface for encapsulating test methods

```
interface CPUMethod;
  task automatic Initialise_Host ();
    CPU.BusMode <= 1;
    CPU.Addr <= 0;
    CPU.DataIn <= 0;
    CPU.Sel <= 1;
    CPU.Rd_DS <= 1;
    CPU.Wr_RW <= 1;
  endtask

  task automatic HostWrite (int a, CellCfgType d); // configure
    #10 CPU.Addr <= a; CPU.DataIn <= d; CPU.Sel <= 0;
    #10 CPU.Wr_RW <= 0;
    while (CPU.Rdy_Dtack!==0) #10;
    #10 CPU.Wr_RW <= 1; CPU.Sel <= 1;
    while (CPU.Rdy_Dtack==0) #10;
  endtask

  task automatic HostRead (int a, output CellCfgType d);
    #10 CPU.Addr <= a; CPU.Sel <= 0;
    #10 CPU.Rd_DS <= 0;
    while (CPU.Rdy_Dtack!==0) #10;
    #10 d = CPU.DataOut; CPU.Rd_DS <= 1; CPU.Sel <= 1;
    while (CPU.Rdy_Dtack==0) #10;
  endtask
endinterface
```

The main testbench module uses the encapsulated methods listed above.

Example 11-8: Utopia ATM testbench

```
`include "definitions.sv"

module test;
  parameter int NumRx = `RxPorts;
  parameter int NumTx = `TxPorts;

  // NumRx x Level 1 Utopia Rx Interfaces
  Utopia Rx[0:NumRx-1] ();

  // NumTx x Level 1 Utopia Tx Interfaces
  Utopia Tx[0:NumTx-1] ();

  // Intel-style Utopia parallel management interface
  CPU mif ();

  // Miscellaneous control interfaces
  logic rst;
  logic clk;
  logic Initialised;

  `include "./testbench_instance.sv"

  task automatic RandomPkt (inout ATMCellType Pkt, inout seed);
    Pkt.uni.GFC = $random(seed);
    Pkt.uni.VPI = $random(seed) & 8'hff;
    Pkt.uni.VCI = $random(seed);
    Pkt.uni.CLP = $random(seed);
    Pkt.uni.PT  = $random(seed);
    Pkt.uni.HEC = hec(Pkt.Mem[0:3]);
    for (int i=0; i<=47; i++) begin
      Pkt.uni.Payload[i] = 47-i; //$random(seed);
    end
  endtask

  logic [7:0] syndrom[0:255];
  initial begin: gen_syndrom
    int i;
    logic [7:0] sndrm;
    for (i = 0; i < 256; i = i + 1 ) begin
      sndrm = i;
      repeat (8) begin
        if (sndrm[7] === 1'b1)
          sndrm = (sndrm << 1) ^ 8'h07;
```

```
        else
          sndrm = sndrm << 1;
      end
      syndrom[i] = sndrm;
    end
  end

  // Function to compute the HEC value
  function automatic logic [7:0] hec (logic [31:0] hdr);
    logic [7:0] rtn;
    rtn = 8'h00;
    repeat (4) begin
      rtn = syndrom[rtn ^ hdr[31:24]];
      hdr = hdr << 8;
    end
    rtn = rtn ^ 8'h55;
    return rtn;
  endfunction

  // System Clock and Reset
  initial begin
    #0 rst = 0; clk = 0;
    #5 rst = 1;
    #5 clk = 1;
    #5 rst = 0; clk = 0;
    forever begin
      #5 clk = 1;
      #5 clk = 0;
    end
  end

  CellCfgType lookup [255:0]; // copy of look-up table

  function logic [0:NumTx-1] find (logic [11:0] VPI);
    for (int i=0; i<=255; i++) begin
      if (lookup[i].VPI == VPI) begin
        return lookup[i].FWD;
      end
    end
    return 0;
  endfunction

  // Stimulus
  initial begin
    automatic int seed=1;
    CellCfgType CellFwd;

    $display("Configuration RxPorts=%0d TxPorts=%0d",
```

```
              `RxPorts, `TxPorts);
    mif.Method.Initialise_Host();

    // Configure through Host interface
    repeat (10) @(negedge clk);
    $display("Loading Memory");
    for (int i=0; i<=255; i++) begin
      CellFwd.FWD = i;
      `ifdef FWDALL
        CellFwd.FWD = '1;
      `endif
      CellFwd.VPI = i;
      mif.Method.HostWrite(i, CellFwd);
      lookup[i] = CellFwd;
    end

    // Verify memory
    $display("Verifying Memory");
    for (int i=0; i<=255; i++) begin
      mif.Method.HostRead(i, CellFwd);
      if (lookup[i] != CellFwd) begin
        $display("Error, Mem Location 0x%h contains 0x%h,
expected 0x%h",
                 i, lookup[i], CellFwd);
        $stop;
      end
    end
    $display("Memory Verified");

    Initialised=1;
    repeat (5000000) @(negedge clk);
    $display("Error Timeout");
    $finish;
  end

  int TxPktCtr [0:NumTx-1];
  logic [0:NumRx-1] RxGenInProgress;
  genvar RxIter;
  genvar TxIter;
  generate // replicate access to ports
    for (RxIter=0; RxIter<NumRx; RxIter++) begin: RxGen
      initial begin: Sender
        int seed;
        logic [0:NumTx-1] TxPortTarget;
        ATMCellType Pkt;

        Rx[RxIter].data=0;
        Rx[RxIter].soc=0;
```

```
      Rx[RxIter].en=1;
      Rx[RxIter].clav=0;
      Rx[RxIter].ready=0;

      RxGenInProgress[RxIter] = 1;
      wait (Initialised === 1'b1);
      seed=RxIter+1;
      Rx[RxIter].Method.Initialise();
      repeat (200) begin
        RandomPkt(Pkt, seed);
        TxPortTarget = find(Pkt.uni.VPI);

        // Increment counter if output packet expected
        for (int i=0; i<NumTx; i++) begin
          if (TxPortTarget[i]) begin
            TxPktCtr[i]++;
            //$display("port %0d ->> %0d", RxIter, i);
          end
        end

        Rx[RxIter].Method.Send(Pkt, RxIter);
        //$display("Port %d sent packet", RxIter);
        repeat ($random(seed)%200) @(negedge clk);
      end
      RxGenInProgress[RxIter] = 0;
    end
  end
endgenerate

// Response - open files for response
generate
  for (TxIter=0; TxIter<NumTx; TxIter++) begin: TxGen
    initial begin: Receiver
      wait (Tx[TxIter].reset===1);
      wait (Tx[TxIter].reset===0);
      forever begin
        Tx[TxIter].Method.Receive(TxIter);
        TxPktCtr[TxIter]--;
      end
    end
  end
endgenerate

// Check for all detected packets
logic [0:NumTx-1] TxDetectEnd;
generate
  for (TxIter=0; TxIter<NumTx; TxIter++) begin: TxDetect
    initial begin
```

```
        TxDetectEnd[TxIter] = 1'b1;
        wait (Initialised === 1'b1);
        wait (RxGenInProgress === 0);
        wait (TxPktCtr[TxIter] == 0)
        TxDetectEnd[TxIter] = 1'b0;
        $display("TxPktCtr[%0d] == %d",
                 TxIter, TxPktCtr[TxIter]);
      end
    end
  endgenerate

  initial begin
    wait (Initialised === 1'b1);
    wait (RxGenInProgress === 0);
    wait (TxDetectEnd === 0);
    $finish;
  end

endmodule
```

The testbench instance of the design is contained in a separate file, so that pre-and post-synthesis versions can be used.

```
squat #(NumRx, NumTx) squat(Rx, Tx, mif, rst, clk);
```

11.7 Summary

This chapter has presented a larger example, modeled using the SystemVerilog extensions to the Verilog HDL. Structures are used to encapsulate all the variables related to `NNI` and `UNI` packets. This allows these many individual signals to be referenced using the structure names, instead of having to reference each signal individually. This encapsulation simplifies the amount of code required to represent complex sets of information. The concise code is easier to read, to test, and to maintain.

These `NNI` and `UNI` structures are grouped together as a union, which allows a single piece of storage to represent either type of packet. Because the union is packed, a value can be stored as one packet type, and retrieved as the other packet type. This further simplifies the code required to transfer a packet from one format to another.

The communication between the major blocks of the design is encapsulated into interfaces. This moves the declarations of the several ports of each module in the design to a central location. The port declarations within each module are minimized to a single interface port. The redundancy of declaring the same ports in several modules is eliminated.

SystemVerilog constructs are also used to simplify the code required to verify the design. The same union used to store the `NNI` and `UNI` packets is used to store test values as an array of bytes. The testbench can load the union variable using bytes, and the value can be read by the design as an `NNI` or `UNI` packet. It is not necessary to copy test values into each variable that makes up a packet.

SystemVerilog includes a large number of additional enhancements for verification that are not illustrated in this example. These enhancements are covered in the companion book, *SystemVerilog for Verification*[1].

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

Chapter 12

Behavioral and Transaction Level Modeling

This chapter defines Transaction Level Modeling (TLM) as an adjunct to behavioral modeling. The chapter explains how TLM can be used, and shows how SystemVerilog is suited to TLM.

Behavioral modeling can be used to provide a high level executable specification for development of both RTL code and the testbench. Transaction level modeling allows the system executable specification to be partitioned into executable specifications of the subsystems.

The executable specifications shown in this chapter are generally not considered synthesizable. However, there are some tools called “high level” or “behavioral” synthesis tools which are able to handle particular categories of behavioral or transaction level modeling.

The topics covered in this chapter include:

- Definition of a transaction
- Transaction level model of a bus
- Multiple slaves
- Arbitration between multiple masters
- Semaphores
- Interfacing transaction level with register transfer level models

12.1 Behavioral modeling

Behavioral modeling (or behavior level modeling) is a style where the state machines of the control logic are not explicitly coded.

An *implicit state machine* is an **always** block which has more than one event control in it. For instance, the following code generates a 1 pulse after the reset falls:

```
always begin
  do @(posedge clock) while (reset);
  @(posedge clock) a = 1;
  @(posedge clock) a = 0;
end
```

An RTL description would have an explicit state register, as follows:

```
logic [1:0] state;

always_ff @(posedge clock)
  if (reset) state = 0;
  else if (state == 0)
    begin state = 1; a = 1; end
  else if (state == 1)
    begin state = 2; a = 0; end
  else state = 0;
```

Note that there is an even more abstract style of behavioral modeling that is not cycle-accurate, and therefore can be used before the detailed scheduling of the design as an executable specification. An example is an image processing algorithm that is to be implemented in hardware.

12.2 What is a transaction?

In everyday life, a transaction is an interaction between two people or organizations to transfer information, money, etc. In a digital system, a transaction is a transfer of data and control between two subsystems. This normally means a request and a response. A transaction has attributes such as type, data, start time, duration, and status. It may also contain sub-transactions.

A key concept of TLMs is the suppressing of uninteresting parts of the communication. For example, if a customer has to pay $20 for a book in a shop, he can perform the transaction at many levels.

Lowest level—20 transactions of $1 each

"$20 please"

"Here is $1", hands over the $1 bill

"Thanks"

"Here is $1", hands over the $1 bill

"Thanks"

"Here is $1", hands over the $1 bill

"Thanks"

... (17 more $1 transactions)

"OK that's $20, here is the book"

"Thanks"

Slightly higher level—4 transactions of $5 each

"$20 please"

"Here is $5", hands over the $5 bill

"Thanks"

"Here is $5", hands over the $5 bill

"Thanks"

"Here is $5", hands over the $5 bill

"Thanks"

"Here is $5", hands over the $5 bill

"OK that's $20, here is the book"

"Thanks"

Higher level—1 transaction of $20

"$20 please"

"Here is $20", hands over the $20 bill

"OK that's $20, here is the book"

"Thanks"

This illustrates a key benefit of TLMs, that of efficiency. Engineers only need to model the level that they are interested in. One of the key motivators in the use of TLMs is the hiding of the detail such that the caller does not know the details of the transactions. This provides a much higher level representation of the interface between blocks.

Note that it is not just the abstracting of the data (e.g. using the $20 total), but also the removal of the control (less low level communication), that increases the TLM abstraction and potential simulation performance. At the highest level, the book buyer is only interested in paying the $20, and does not really care whether it is in $1s or $5s or a $20. Hiding detail allows different implementations of a protocol to exist without the caller knowing, or needing to know, which level is being used, and then being able to switch in and out different TLMs as needed. Switching in and out different TLMs may be done for efficiency reasons, to use a less detailed more efficient TLM, or maybe during the life of a project, where in the beginning only high level details are defined, and then. more details are added over the life of the project.

12.3 Transaction level modeling in SystemVerilog

Whereas behavior level modeling raises the abstraction of the block functionality, transaction level modeling raises the abstraction level of communication between blocks and subsystems, by hiding the details of both control and data flow across interfaces.

In SystemVerilog, a key use of the **`interface`** construct is to be able to separate the descriptions of the functionality of modules and the communication between them.

Transaction level modeling is a concept, and not a feature of a specific language, though there are certain language constructs that are useful for writing transaction level models (TLMs). These include:

- Structural hierarchy
- Function and task calls across hierarchy boundaries

- Records or structures
- The ability to package data with function/task calls
- The ability to parallelize and serialize data
- Semaphores to control shared resources

A fundamental capability that is needed for TLMs is to be able to encapsulate the lower level details of information exchange into function and task calls across an interface. The caller only needs to know what data is sent and returned, with the details of the transmission being hidden in the function/task call.

The transaction request is made by calling the task or function across the interface/module boundary. Using SystemVerilog's **`interface`** and function/task calling mechanisms makes creating TLMs in SystemVerilog extremely simple. The term *method* is used to describe such function/task calls, since they are similar to methods in object-oriented languages.

12.3.1 Memory subsystem example

Example 12-1 illustrates a simple memory subsystem. Initially this is coded as read and write tasks called by a single testbench. The testbench tries a range of addresses, and tests the error flag.

Example 12-1: Simple memory subsystem with read and write tasks

```
module TopTasks;

  logic [20:0] A;
  logic [15:0] D;
  logic        E;
  parameter    LOWER = 20'h00000;
  parameter    UPPER = 20'h7ffff;
  logic [15:0] Mem[LOWER:UPPER];

  task ReadMem(input  logic [19:0] Address,
               output logic [15:0] Data,
               output logic        Error);
    if (Address >= LOWER && Address <= UPPER) begin
      Data = Mem[Address];
      Error = 0;
    end
    else Error = 1;
  endtask
```

```
  task WriteMem(input  logic [19:0] Address,
                input  logic [15:0] Data,
                output logic        Error);
    if (Address >= LOWER && Address <= UPPER) begin
      Mem[Address] = Data;
      Error = 0;
    end
    else Error = 1;
  endtask

  initial begin
    for (A = 0; A < 21'h100000; A = A + 21'h40000) begin
      fork
        #1000;
        WriteMem(A[19:0], 0, E);
      join
      if (E) $display ("%t bus error on write %h", $time, A);
        else $display ("%t write OK %h", $time, A);

      fork
        #1000;
        ReadMem(A[19:0], D, E);
      join
      if (E) $display ("%t bus error on read %h", $time, A);
        else $display ("%t read OK %h", $time, A);
    end
  end

endmodule : TopTasks
```

This example gives the following display output:

```
1000 write OK 000000
2000 read OK 000000
3000 write OK 040000
4000 read OK 040000
5000 bus error on write 080000
6000 bus error on read 080000
7000 bus error on write 0c0000
8000 bus error on read 0c0000
```

12.4 Transaction level models via interfaces

The next example partitions the memory subsystem into three modules, two memory units and a testbench. The modules are connected by an interface. In this design, the address regions are wired into the memory units. One, and only one, memory should respond to each read or write. If no unit responds, there is a bus error.

This broadcast request with single response can be conveniently modeled with the **`extern forkjoin`** task construct in SystemVerilog interfaces. This behaves like a **`fork`**...**`join`** containing multiple task calls. The difference is that the number of calls is not defined, which allows the same interface code to be used for any number of memory units. The output values are written to the actual arguments for each task call, and the valid task call delays its response so that it overwrites the invalid ones.

Example 12-2: Two memory subsystems connected by an interface

```
module TopTLM;

  Membus Mbus();
  Tester T(Mbus);
  Memory #(.Lo(20'h00000), .Hi(20'h3ffff))
           M1(Mbus);  // lower addrs
  Memory #(.Lo(20'h40000), .Hi(20'h7ffff))
           M2(Mbus);  // higher addrs

endmodule : TopTLM


// Interface header
interface Membus;

  extern forkjoin task ReadMem  (input  logic [19:0] Address,
                                 output logic [15:0] Data,
                                        bit          Error);

  extern forkjoin task WriteMem (input  logic [19:0] Address,
                                 input  logic [15:0] Data,
                                 output bit          Error);

  extern task Request();
  extern task Relinquish();

endinterface
```

```
module Tester (interface Bus);

  logic [15:0] D;
  logic E;

  int A;

  initial begin
    for (A = 0; A < 21'h100000; A = A + 21'h40000) begin
      fork
        #1000;
        Bus.WriteMem(A[19:0], 0, E);
      join
      if (E) $display ("%t bus error on write %h", $time, A);
        else $display ("%t write OK %h", $time, A);

      fork
        #1000;
        Bus.ReadMem(A[19:0], D, E);
      join
      if (E) $display ("%t bus error on read %h", $time, A);
        else $display ("%t read OK %h", $time, A);
    end
  end

endmodule


// Memory Modules
// forkjoin task model delays if OK (last wins)
module Memory(interface Bus);

  parameter Lo = 20'h00000;
  parameter Hi = 20'h3ffff;
  logic [15:0] Mem[Lo:Hi];

  task Bus.ReadMem(input  logic [19:0] Address,
                   output logic [15:0] Data,
                   output logic        Error);

    if (Address >= Lo && Address <= Hi) begin
      #100 Data = Mem[Address];
      Error = 0;
    end
    else Error = 1;
  endtask
```

```
  task Bus.WriteMem(input  logic [19:0] Address,
                    input  logic [15:0] Data,
                    output logic        Error);

    if (Address >= Lo && Address <= Hi) begin
      #100 Mem[Address] = Data;
      Error = 0;
    end
    else Error = 1;
  endtask

endmodule
```

This example gives the following display output:

```
1000 write OK 000000
2000 read OK 000000
3000 write OK 040000
4000 read OK 040000
5000 bus error on write 080000
6000 bus error on read 080000
7000 bus error on write 0c0000
8000 bus error on read 0c0000
```

12.5 Bus arbitration

If there are two bus masters, it is necessary to prevent both masters from accessing the bus at the same time. The abstract mechanism for modeling such a resource sharing is the *semaphore*. SystemVerilog includes a built-in semaphore class object. In this chapter, however, an interface model is used. This illustrates how the class behavior can be described, using interfaces and interface methods.

The `Semaphore` interface in the following example has a number of keys, corresponding to resources. The default is one. The `get` task waits for the key(s) to be available, and then removes them. The `put` task replaces the key(s).

The model below has an `arbiter` module containing the semaphore. An alternative would be to put the semaphore in the interface, but this would differ from the RTL implementation hierarchy.

Example 12-3: TLM model with bus arbitration using semaphores

```
module TopArbTLM;

  Membus Mbus();
  Tester T1(Mbus);
  Tester T2(Mbus);
  Arbiter A(Mbus);
  Memory #(.Lo(20'h00000), .Hi(20'h3ffff)) M1(Mbus);
  Memory #(.Lo(20'h40000), .Hi(20'h7ffff)) M2(Mbus);

endmodule : TopArbTLM


interface Membus; // repeated from previous example

  extern forkjoin task ReadMem  (input  logic [19:0] Address,
                                 output logic [15:0] Data,
                                        bit          Error);

  extern forkjoin task WriteMem (input  logic [19:0] Address,
                                 input  logic [15:0] Data,
                                 output bit          Error);

  extern task Request();
  extern task Relinquish();

endinterface


interface Semaphore #(parameter int unsigned initial_keys = 1);
  int unsigned keys = initial_keys;

  task get(int unsigned n = 1);
    wait (n <= keys);
    keys -= n;
  endtask

  task put (int unsigned n = 1);
    keys += n;
  endtask

endinterface


module Arbiter (interface Bus);
  Semaphore s ();  // built-in type would use semaphore s = new;
```

```
  task Bus.Request();
    s.get();
  endtask

  task Bus.Relinquish();
    s.put();
  endtask

endmodule


module Tester (interface Bus);
  logic [15:0] D;
  logic        E;
  int          A;

  initial begin : test_block
    for (A = 0; A < 21'h100000; A = A + 21'h40000)
    begin : loop
      fork
        #1000;
        begin
          Bus.Request;
          Bus.WriteMem(A[19:0], 0, E);
          if (E) $display("%t bus error on write %h", $time, A);
            else $display ("%t write OK %h", $time, A);
          Bus.Relinquish;
        end
      join
      fork
        #1000;
        begin
          Bus.Request;
          Bus.ReadMem(A[19:0], D, E);
          if (E) $display("%t bus error on read %h", $time, A);
            else $display ("%t read OK %h", $time, A);
          Bus.Relinquish;
        end
      join
    end : loop
  end : test_block

endmodule
```

```
// Memory Modules
// forkjoin task model delays if OK (last wins)
module Memory (interface Bus); // repeated from previous example

  parameter Lo = 20'h00000;
  parameter Hi = 20'h3ffff;
  logic [15:0] Mem[Lo:Hi];

  task Bus.ReadMem(input  logic [19:0] Address,
                   output logic [15:0] Data,
                   output logic        Error);

    if (Address >= Lo && Address <= Hi) begin
      #100 Data = Mem[Address];
      Error = 0;
    end
    else Error = 1;
  endtask

  task Bus.WriteMem(input  logic [19:0] Address,
                    input  logic [15:0] Data,
                    output logic        Error);

    if (Address >= Lo && Address <= Hi) begin
      #100 Mem[Address] = Data;
      Error = 0;
    end
    else Error = 1;
  endtask

endmodule
```

This example gives the following output:

```
100 write OK 00000000
200 write OK 00000000
1100 read OK 00000000
1200 read OK 00000000
2100 write OK 00040000
2200 write OK 00040000
3100 read OK 00040000
3200 read OK 00040000
4000 bus error on write 00080000
4000 bus error on write 00080000
5000 bus error on read 00080000
```

```
5000 bus error on read 00080000
6000 bus error on write 000c0000
6000 bus error on write 000c0000
7000 bus error on read 000c0000
7000 bus error on read 000c0000
```

12.6 Transactors, adapters, and bus functional models

For TLMs to be useful for hardware design, it is necessary to connect them to the RTL models via code which is variously called *transactors*, *adapters*, and *bus functional models* (*BFMs*). These can be either master or slave adapters, depending on the direction of control.

The master adapter contains tasks, called by the master subsystem TLM, which encapsulate the protocol and manipulate the signals to communicate with an RTL model of the slave subsystem.

The slave adapter contains processes, which monitor signals from an RTL model of the master subsystem and call the tasks or functions in the TLM of the slave subsystem.

12.6.1 Master adapter as module

One way to code adapters is to make them modules which translate a transaction level interface to a pin level interface, or vice-versa. The adapter has two interface ports, the transaction level and the pin level.

Example 12-4: Adapter modeled as a module

```
module TopTLMPLM;

  Multibus TLMbus();
  Multibus PLMbus();

  Tester T(TLMbus);
  MultibusMaster MM (TLMbus, PLMbus);
  MultibusArbiter MA (PLMbus);
  Clock Clk(PLMbus);
  MultibusMonitor MO(PLMbus);
```

```
  MemoryPIN  #(.Lo(20'h00000), .Hi(20'h3ffff))
    M1 (PLMbus.ADR, PLMbus.DAT, PLMbus.MRDC, PLMbus.MWTC,
        PLMbus.XACK, PLMbus.BCLK);
  MemoryPIN  #(.Lo(20'h40000), .Hi(20'h7ffff))
    M2 (PLMbus.ADR, PLMbus.DAT, PLMbus.MRDC, PLMbus.MWTC,
        PLMbus.XACK, PLMbus.BCLK);

endmodule : TopTLMPLM
```

The example below is a simplified version of the Intel Multibus (now IEEE 796). This allows multiple masters and multiple slaves. Each master has a request wire BREQ to the arbiter module and a priority input wire BPRN from the arbiter, i.e. the parallel priority technique specified in the standard.

Example 12-5: Simplified Intel Multibus with multiple masters and slaves

```
// Interface header
interface Multibus;
  parameter int MASTERS = 1; // number of bus masters

  // structural communication
  tri [19:0]                       ADR;  // address bus (inverted)
  tri [15:0]                       DAT;  // data bus (inverted)
  wand /*active0*/           MRDC, MWTC; // mem read/write commands
  wand /*active0*/                 XACK; // transfer acknowledge
  wand /*active0*/ [1:MASTERS]     BREQ; // bus request
  wand /*active0*/                 CBRQ; // common bus request
  wire /*active0*/                 BUSY; // bus busy
  wire /*active0*/ [1:MASTERS]     BPRN; // bus priority to master
  logic                            BCLK; // bus clock; driven
                                         // by only one master
  logic                            CCLK; // constant clock
  wand                             INIT; // initialize


  // Tasks - Behavioral communication

  extern task Request  (input int n);
  extern task Relinquish (input int n);
  extern forkjoin task ReadMem  (input  logic [19:0] Address,
                                  output logic [15:0] Data,
                                         bit          Error);
```

```
  extern forkjoin task WriteMem (input  logic [19:0] Address,
                                 input  logic [15:0] Data,
                                 output bit          Error);

endinterface


module Clock (Multibus Bus);

  always begin  // clock
    #50 Bus.CCLK = 0;
    #50 Bus.CCLK = 1;
  end

endmodule : Clock
```

The master adapter is coded with tasks which drive wires and have the same prototype as the transaction level slave. If only a single driver is allowed for a wire, a logic variable can be used directly. If multiple drivers are allowed, the adapter needs a continuous assignment to model the buffering to the wire.

If the master does not already have control of the bus, the master has to request it from the arbiter, wait for the priority to be granted, and then wait for the previous master to relinquish the bus. These actions are encapsulated in the task `GetBus`.

If no slave responds to the address, then a time out occurs and the read or write task returns with the error flag set.

Example 12-6: Simple Multibus TLM example with master adapter as a module

```
module MultibusMaster (interface Tasks, interface Wires);
    parameter int Number = 1;   // number of master for
                                // request/grant

  enum {IDLE, READY, READ, WRITE} Master_State;

  logic [19:0] adr  = 'z;  assign Wires.ADR = adr;
  logic [15:0] dat  = 'z;  assign Wires.DAT = dat;
  logic        mrdc = 1;   assign Wires.MRDC = mrdc;
  logic        mwtc = 1;   assign Wires.MWTC = mwtc;
  logic        breq = 1;   assign Wires.BREQ[Number] = breq;
```

```
logic        cbrq = 1;   assign Wires.CBRQ = cbrq;
logic        busy = 1;   assign Wires.BUSY = busy;

assign Wires.BCLK = Wires.CCLK;

task Tasks.ReadMem (input  logic [19:0] Address,
                    output logic [15:0] Data,
                    output logic        Error);

  if (Master_State == IDLE) GetBus();
    else assert (Master_State == READY);
  Master_State = READ;
  Data = 'x; Error = 1;  // default if no slave responds
  adr = ~Address;
  #50 mrdc = 0;  //min delay
  fork
    begin: ok
      @(negedge Wires.XACK) Data = ~ Wires.DAT;
      EndRead();
      @(posedge Wires.XACK) Error = 0;
      disable timeout;
    end
    begin: timeout  // Timeout if no acknowledgement
      #900 Error = 1;
      EndRead();
      disable ok;
    end
  join
  FreeBus();
endtask

task Tasks.WriteMem (input  logic [19:0] Address,
                     input  logic [15:0] Data,
                     output logic        Error);
  if (Master_State == IDLE) GetBus();
    else assert (Master_State == READY);
  Master_State = WRITE;
  Error = 1;  // default if no slave responds
  GetBus();
  adr = ~Address;
  dat = ~Data;
  #50 mwtc = 0;
  fork
    begin: ok
      @(negedge Wires.XACK) EndWrite();
      @(posedge Wires.XACK) Error = 0;
      disable timeout;
    end
```

```
      begin: timeout  // Timeout if no acknowledgement
        #900 Error = 1;
        EndWrite();
        disable ok;
      end
    join
    FreeBus();
  endtask

  task EndRead();
    mrdc = 1;
    #50 adr = 'z;
  endtask

  task EndWrite();
    mwtc = 1;
    #60 adr = 'z;
    dat = 'z;
  endtask

  task GetBus();
    @(negedge Wires.BCLK) breq = 0;
    cbrq = 0;
    @(negedge Wires.BPRN[Number]);
    @(negedge Wires.BCLK iff !Wires.BPRN[Number]);
    #50 busy = 0;
    cbrq = 1;
  endtask

  task FreeBus();
    breq = 1;
    if (Wires.CBRQ) Master_State = READY;
    else begin
      Master_State = IDLE;
      busy = 1;  // relinquish the bus if CBRQ asserted
    end
  endtask

endmodule: MultibusMaster


module Tester (interface Bus); // repeated from previous example
  logic [15:0] D;
  logic        E;
  int          A;
```

```
  initial begin
    for (A = 0; A < 21'h100000; A = A + 21'h40000)
    begin
      fork #1000; Bus.WriteMem(A[19:0], 0, E); join
      if (E) $display ("%t bus error on write %h", $time, A);
        else $display ("%t write OK %h", $time, A);
      fork #1000; Bus.ReadMem(A[19:0], D, E); join
      if (E) $display ("%t bus error on read %h", $time, A);
        else $display ("%t read OK %h", $time, A);
    end
  end

  initial # 10000 $finish;

endmodule


module MultibusArbiter #(parameter MASTERS = 1)(interface Bus);
  logic [1:MASTERS] bprn = '1; assign Bus.BPRN = bprn;
  int last = 0;
  int i;

  always @(negedge Bus.BCLK)
    if (Bus.CBRQ == 0) begin // request
      i = last+1;
      forever begin
        if (i > MASTERS) i = 1;
        if (Bus.BREQ[i] == 0) break;
        assert (i != last); else $fatal(0, "no bus master");
        i++;
        if (i > MASTERS) i = 1;
      end
      last = i;
      #50 bprn [i] = 0; //$display("bprn[%b] = %b", i, bprn);
    end
    else if (Bus.BUSY == 0) begin // relinquish
      #50 bprn [last] = 1;
    end
endmodule : MultibusArbiter


module MultibusMonitor (interface Bus);

  initial $monitor(
       "ADR=%h DAT=%h MRDC=%b MWTC=%b XACK=%b BREQ=%b CBRQ=%b
BUSY=%b BPRN=%b",
        Bus.ADR, Bus.DAT, Bus.MRDC, Bus.MWTC, Bus.XACK,
        Bus.BREQ, Bus.CBRQ, Bus.BUSY, Bus.BPRN);
endmodule
```

```
// Memory Module with pin level interface
module MemoryPIN (
    input  [19:0]            ADR,     // address bus
    inout  [15:0]            DAT,     // data bus
    input  /*active0*/       MRDC,    // memory read
    input  /*active0*/       MWTC,    // memory write
    output logic /*active0*/ XACK,    // acknowledge
    input                    CCLK
  );

  parameter Lo = 20'h00000;
  parameter Hi = 20'h3ffff;
  logic [15:0] Mem[Lo:Hi];
  logic [15:0] Bufdat;
  logic        Bufena = 0;  //default disables buffers

  initial XACK = 1; // default disables

  assign DAT = Bufena ? Bufdat : 'z;

  always @(posedge CCLK)
  begin
    automatic logic [19:0] Address = ~ADR;
    if (MRDC == 0 && Address >= Lo && Address <= Hi)   // read
    begin
      Bufdat <= ~Mem[Address];
      Bufena <= 1;
      XACK <= 0;
    end
    else if (MWTC == 0 && Address >= Lo && Address <= Hi)
    begin                                               // write
      Mem[Address] = ~DAT;
      XACK <= 0;
    end
    else begin
      XACK <= 1;
      Bufena <= 0;
    end
  end

endmodule: MemoryPIN
```

12.6.2 Adapter in an interface

Another way to code adapters is to put them in the interface. This is straightforward for a single adapter, but not for multiple ones, because of name collisions.

These require modified versions of the interface, which is quite easy for master adapters, since unused tasks should not interfere with the model. Slave adapters, on the other hand, call tasks or functions in the slave TLM, and there will be an elaboration error if the slave TLM is missing. So a different version of the interface is needed. The example below shows a master adapter.

Example 12-7: Simple Multibus TLM example with master adapter as an interface

```
module TopInterfaceAdapter;

  Multibus Mbus();

  Tester T(Mbus);
  MultibusArbiter MA(Mbus);
  Clock Clk(Mbus);
  MultibusMonitor MO(Mbus);

  /* MemoryPIN  #(.Lo(20'h00000), .Hi(20'h3ffff)) M1 (Mbus);
     MemoryPIN  #(.Lo(20'h40000), .Hi(20'h7ffff)) M2 (Mbus); */

  MemoryPIN  #(.Lo(20'h00000), .Hi(20'h3ffff)) M1 (Mbus.ADR,
      Mbus.DAT, Mbus.MRDC, Mbus.MWTC, Mbus.XACK, Mbus.BCLK);
  MemoryPIN  #(.Lo(20'h40000), .Hi(20'h7ffff)) M2 (Mbus.ADR,
      Mbus.DAT, Mbus.MRDC, Mbus.MWTC, Mbus.XACK, Mbus.BCLK);

endmodule : TopInterfaceAdapter


// Interface header
interface Multibus;
  parameter int MASTERS = 1;  // number of bus masters
  parameter int Number = 1;

  // structural communication
  tri  [19:0]                  ADR;  // address bus
  tri  [15:0]                  DAT;  // data bus
  wand /*active0*/       MRDC, MWTC; // mem read/write commands
  wand /*active0*/             XACK; // acknowledge
  wand /*active0*/ [1:MASTERS] BREQ;
  wand /*active0*/             CBRQ;
```

```
  wire /*active0*/              BUSY;
  wire /*active0*/ [1:MASTERS] BPRN;
  logic                         BCLK;
  logic                         CCLK;

  wand                          INIT;

// Master Adapter converts ReadMem/WriteMem calls into waveforms
  enum {IDLE, READ, WRITE} Master_State;

  logic [19:0] adr  = 'z;  assign ADR = adr;
  logic [15:0] dat  = 'z;  assign DAT = dat;
  logic        mrdc = 1;   assign MRDC = mrdc;
  logic        mwtc = 1;   assign MWTC = mwtc;
  logic        breq = 1;   assign BREQ[Number] = breq;
  logic        cbrq = 1;   assign CBRQ = cbrq;
  logic        busy = 1;   assign BUSY = busy;

  task ReadMem (input  logic [19:0] Address,
                output logic [15:0] Data,
                output logic        Error);
    assert (Master_State == IDLE);
    Master_State = READ;
    Data = 'x;
    Error = 1;  // default if no slave responds
    GetBus();
    adr = ~Address;
    #50 mrdc = 0;  //min delay
    fork
      begin: ok
        @(negedge XACK) Data = ~ DAT;
        EndRead();
        @(posedge XACK) Error = 0;
        disable timeout;
      end
      begin: timeout  // Timeout if no acknowledgement
        #900 Error = 1;
        EndRead();
        disable ok;
      end
    join
    FreeBus();
    Master_State = IDLE;
  endtask

  task WriteMem (input  logic [19:0] Address,
                 input  logic [15:0] Data,
                 output logic        Error);
```

```
    assert (Master_State == IDLE);
    Master_State = WRITE;
    Error = 1;  // default if no slave responds
    GetBus();
    adr = ~Address;
    dat = ~Data;
    #50 mwtc = 0;
    fork
      begin: ok
        @(negedge XACK) EndWrite();
        @(posedge XACK) Error = 0;
        disable timeout;
      end
      begin: timeout  // Timeout if no acknowledgement
        #900 Error = 1;
        EndWrite();
        disable ok;
      end
    join
    FreeBus();
    Master_State = IDLE;
  endtask

  task EndRead();
    mrdc = 1;
    #50 adr = 'z;
  endtask

  task EndWrite();
    mwtc = 1;
    #60 adr = 'z;
    dat = 'z;
  endtask

  task GetBus();
    breq = 0;
    cbrq = 0;
    @(negedge BCLK iff !BPRN[Number]);
    #50 busy = 0;
    cbrq = 1;
  endtask

  task FreeBus();
    breq = 1;
    busy = 1;
  endtask

endinterface
```

```
module Clock (Multibus Bus);

  always begin  // clock
    #50 Bus.BCLK = 0;
    #50 Bus.BCLK = 1;
  end

  initial # 10000 $finish;

endmodule : Clock


module Tester (interface Bus);
  logic [15:0] D;
  logic E;
  int A;
  initial begin
    for (A = 0; A < 21'h100000; A = A + 21'h40000)
    begin
      fork
        #1000;
        Bus.WriteMem(A[19:0], 0, E);
      join
      if (E) $display ("%t bus error on write %h", $time, A);
        else $display ("%t write OK %h", $time, A);
      fork
        #1000;
        Bus.ReadMem(A[19:0], D, E);
      join
      if (E) $display ("%t bus error on read %h", $time, A);
        else $display ("%t read OK %h", $time, A);
    end
  end

endmodule


module MultibusArbiter #(parameter MASTERS = 1)(interface Bus);
  logic [1:MASTERS] bprn = '1; assign Bus.BPRN = bprn;
  int last = 0;
  int i;

  always @(negedge Bus.BCLK)
    if (Bus.CBRQ == 0) begin // request
      i = last+1;
      forever begin
        if (i > MASTERS) i = 1;
```

```
        if (Bus.BREQ[i] == 0) break;
        assert (i != last); else $fatal(0, "no bus master");
        i++;
        if (i > MASTERS) i = 1;
      end
      last = i;
      #50 bprn [i] = 0; //$display("bprn[%b] = %b", i, bprn);
    end
    else if (Bus.BUSY == 0) begin // relinquish
      #50 bprn [last] = 1;
    end
endmodule : MultibusArbiter


module MultibusMonitor (interface Bus);

  initial $monitor(
      "ADR=%h DAT=%h MRDC=%b MWTC=%b XACK=%b BREQ=%b CBRQ=%b
BUSY=%b BPRN=%b",
      Bus.ADR, Bus.DAT, Bus.MRDC, Bus.MWTC, Bus.XACK, Bus.BREQ,
      Bus.CBRQ, Bus.BUSY, Bus.BPRN);

endmodule


// Memory Module with pin level interface
module MemoryPIN (
    input  [19:0]            ADR,     // address bus
    inout  [15:0]            DAT,     // data bus
    input  /*active0*/       MRDC,    // memory read
    input  /*active0*/       MWTC,    // memory write
    output logic /*active0*/ XACK,    // acknowledge
    input                    CCLK
  );

  parameter Lo = 20'h00000;
  parameter Hi = 20'h3ffff;
  logic [15:0] Mem[Lo:Hi];
  logic [15:0] Bufdat;
  logic        Bufena = 0;  //default disables buffers

  initial XACK = 1;  // default disables

  assign DAT = Bufena ? Bufdat : 'z;

  always @(posedge CCLK) begin
    automatic logic [19:0] Address = ~ADR;
    if ( MRDC == 0 && Address >= Lo && Address <= Hi)   // read
```

```
    begin
      Bufdat <= ~Mem[Address];
      Bufena <= 1;
      XACK <= 0;
    end
    else if (MWTC == 0 && Address >= Lo && Address <= Hi)
    begin                                                       // write
      Mem[Address] = ~DAT;
      XACK <= 0;
    end
    else begin
      XACK <= 1;
      Bufena <= 0;
    end
  end

endmodule: MemoryPIN
```

12.7 More complex transactions

The transactions modeled above are simple, in the sense that there is only one at a time. This allows the lifetime of the transaction to correspond to the lifetime of the task call initiating it. The task can contain the data relevant to the transaction, such as start time.

Other systems may allow one transaction to start before the previous one has finished (overlapping or pipelining). They may even allow out-of-order completion (split transactions). In these cases, the data about the transaction cannot be contained in a single task. Either a new process (thread) must be spawned to control or monitor the transaction and to hold relevant data, or a dynamic data object must be created to store the information about the transaction.

These more elaborate transaction level models and their language constructs are typically used in verification, and are therefore described in the companion book, *SystemVerilog for Verification*[1].

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

12.8 Summary

Transactions have traditionally been used in system modeling and in hardware verification. TLM has not been used much by hardware designers. One of the reasons is that Verilog-2005 and VHDL-2000 do not have the ability to define an interface with methods, whereas some programming and verification languages have classes, which can be used in a similar way.

SystemVerilog brings the interface and method constructs into HDL, allowing the hardware designer to take advantage of the TLM technique, and to represent the rest of the system at a more abstract level, with the benefits of simplicity and simulation performance.

Over time, new tools are likely to be developed for verification (and maybe for synthesis) of the transaction level modeling style presented in this chapter.

Appendix A
The SystemVerilog Formal Definition (BNF)

This appendix contains the formal definition of the SystemVerilog standard. The definition is taken directly from Annex A of the IEEE 1800-2005 *SystemVerilog Language Reference Manual* (SystemVerilog LRM)[1].

The formal definition of SystemVerilog is described in Backus-Naur Form (BNF). The variant of BNF used in this appendix is as follows:

- Bold text represents literal words themselves (these are called terminals). For example: **module**.
- Non-bold text (possibly with underscores) represents syntactic categories (i.e. non terminals). For example: port_identifier.
- Syntactic categories are defined using the form:
  syntactic_category ::= definition
- A vertical bar (|) separates alternatives.
- Square brackets ([ ]) enclose optional items.
- Braces ({ }) enclose items which can be repeated zero or more times.

1. Appendix A reprinted with permission from Annex A of the IEEE Std. 1800-2005 SystemVerilog: Unified Hardware Design, Specification and Verification Language by IEEE. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner.

A.1 Source text

A.1.1 Library source text

library_text ::= { library_descriptions }

library_descriptions ::=
  library_declaration
  | include_statement
  | config_declaration
  | **;**

library_declaration ::=
  library library_identifier file_path_spec { **,** file_path_spec }
    [**-incdir** file_path_spec { **,** file_path_spec }] **;**

include_statement ::= **include** file_path_spec **;**

A.1.2 Configuration source text

config_declaration ::=
  config config_identifier **;**
    design_statement
    { config_rule_statement }
  endconfig [**:** config_identifier]

design_statement ::= **design** { [library_identifier **.**] cell_identifier } **;**

config_rule_statement ::=
  default_clause liblist_clause
  | inst_clause liblist_clause
  | inst_clause use_clause
  | cell_clause liblist_clause
  | cell_clause use_clause
  | **;**

default_clause ::= **default**

inst_clause ::= **instance** inst_name

inst_name ::= topmodule_identifier { **.** instance_identifier }

cell_clause ::= **cell** [library_identifier **.**] cell_identifier

liblist_clause ::= **liblist** {library_identifier}

use_clause ::= **use** [library_identifier **.**] cell_identifier [**: config**]

A.1.3 Module and primitive source text

source_text ::= [timeunits_declaration] { description }

description ::=
  module_declaration
  | udp_declaration
  | interface_declaration
  | program_declaration
  | package_declaration
  | { attribute_instance } package_item
  | { attribute_instance } bind_directive

module_nonansi_header ::=
  { attribute_instance } module_keyword [lifetime] module_identifier [parameter_port_list]
    list_of_ports **;**

module_ansi_header ::=
{ attribute_instance } module_keyword [lifetime] module_identifier [parameter_port_list]
[list_of_port_declarations] **;**

module_declaration ::=
module_nonansi_header [timeunits_declaration] { module_item }
endmodule [**:** module_identifier]
| module_ansi_header [timeunits_declaration] { non_port_module_item }
endmodule [**:** module_identifier]
| { attribute_instance } module_keyword [lifetime] module_identifier **(.*) ;**
[timeunits_declaration] { module_item } **endmodule** [**:** module_identifier]
| **extern** module_nonansi_header
| **extern** module_ansi_header

module_keyword ::= **module** | **macromodule**

interface_nonansi_header ::=
{ attribute_instance } **interface** [lifetime] interface_identifier
[parameter_port_list] list_of_ports **;**

interface_ansi_header ::=
{attribute_instance } **interface** [lifetime] interface_identifier
[parameter_port_list] [list_of_port_declarations] **;**

interface_declaration ::=
interface_nonansi_header [timeunits_declaration] { interface_item }
endinterface [**:** interface_identifier]
| interface_ansi_header [timeunits_declaration] { non_port_interface_item }
endinterface [**:** interface_identifier]
| { attribute_instance } **interface** interface_identifier **(.*) ;**
[timeunits_declaration] { interface_item }
endinterface [**:** interface_identifier]
| **extern** interface_nonansi_header
| **extern** interface_ansi_header

program_nonansi_header ::=
{ attribute_instance } **program** [lifetime] program_identifier
[parameter_port_list] list_of_ports **;**

program_ansi_header ::=
{attribute_instance } **program** [lifetime] program_identifier
[parameter_port_list] [list_of_port_declarations] **;**

program_declaration ::=
program_nonansi_header [timeunits_declaration] { program_item }
endprogram [**:** program_identifier]
| program_ansi_header [timeunits_declaration] { non_port_program_item }
endprogram [**:** program_identifier]
| { attribute_instance } **program** program_identifier **(.*) ;**
[timeunits_declaration] { program_item }
endprogram [**:** program_identifier]
| **extern** program_nonansi_header
| **extern** program_ansi_header

class_declaration ::=
[**virtual**] **class** [lifetime] class_identifier [parameter_port_list]
[**extends** class_type [**(** list_of_arguments **)**]]**;**
{ class_item }

endclass [**:** class_identifier]

package_declaration ::=
{ attribute_instance } **package** package_identifier **;**
[timeunits_declaration] { { attribute_instance } package_item }
endpackage [**:** package_identifier]

timeunits_declaration ::=
timeunit time_literal **;**
| **timeprecision** time_literal **;**
| **timeunit** time_literal **;**
timeprecision time_literal **;**
| **timeprecision** time_literal **;**
timeunit time_literal **;**

A.1.4 Module parameters and ports

parameter_port_list ::=
 # (list_of_param_assignments { **,** parameter_port_declaration } **)**
| **# (** parameter_port_declaration { **,** parameter_port_declaration } **)**
| **#( )**

parameter_port_declaration ::=
parameter_declaration
| data_type list_of_param_assignments
| **type** list_of_type_assignments

list_of_ports ::= **(** port { **,** port } **)**

list_of_port_declarations[26] ::=
([{ attribute_instance} ansi_port_declaration { **,** { attribute_instance} ansi_port_declaration }] **)**

port_declaration ::=
{ attribute_instance } inout_declaration
| { attribute_instance } input_declaration
| { attribute_instance } output_declaration
| { attribute_instance } ref_declaration
| { attribute_instance } interface_port_declaration

port ::=
[port_expression]
| **.** port_identifier **(** [port_expression] **)**

port_expression ::=
port_reference
| **{** port_reference { **,** port_reference } **}**

port_reference ::=
port_identifier constant_select

port_direction ::= **input** | **output** | **inout** | **ref**

net_port_header ::= [port_direction] net_port_type

variable_port_header ::= [port_direction] variable_port_type

interface_port_header ::=
interface_identifier [**.** modport_identifier]
| **interface** [**.** modport_identifier]

ansi_port_declaration ::=
[net_port_header | interface_port_header] port_identifier { unpacked_dimension }
| [variable_port_header] port_identifier variable_dimension [**=** constant_expression]

| [net_port_header | variable_port_header] **.** port_identifier **(** [expression] **)**

A.1.5 Module items

module_common_item ::=
module_or_generate_item_declaration
| interface_instantiation
| program_instantiation
| concurrent_assertion_item
| bind_directive
| continuous_assign
| net_alias
| initial_construct
| final_construct
| always_construct
| loop_generate_construct
| conditional_generate_construct

module_item ::=
port_declaration **;**
| non_port_module_item

module_or_generate_item ::=
{ attribute_instance } parameter_override
| { attribute_instance } gate_instantiation
| { attribute_instance } udp_instantiation
| { attribute_instance } module_instantiation
| { attribute_instance } module_common_item

module_or_generate_item_declaration ::=
package_or_generate_item_declaration
| genvar_declaration
| clocking_declaration
| **default clocking** clocking_identifier **;**

non_port_module_item ::=
generate_region
| module_or_generate_item
| specify_block
| { attribute_instance } specparam_declaration
| program_declaration
| module_declaration
| interface_declaration
| timeunits_declaration[18]

parameter_override ::= **defparam** list_of_defparam_assignments **;**

bind_directive ::=
bind bind_target_scope [**:** bind_target_instance_list] bind_instantiation **;**
| **bind** bind_target_instance bind_instantiation **;**

bind_target_scope ::=
module_identifier
| interface_identifier

bind_target_instance ::=
hierarchical_identifier constant_bit_select

bind_target_instance_list ::=

bind_target_instance {**,** bind_target_instance }

bind_instantiation ::=
program_instantiation
| module_instantiation
| interface_instantiation

A.1.6 Interface items

interface_or_generate_item ::=
{ attribute_instance } module_common_item
| { attribute_instance } modport_declaration
| { attribute_instance } extern_tf_declaration

extern_tf_declaration ::=
extern method_prototype **;**
| **extern forkjoin** task_prototype **;**

interface_item ::=
port_declaration **;**
| non_port_interface_item

non_port_interface_item ::=
generate_region
| { attribute_instance } specparam_declaration
| interface_or_generate_item
| program_declaration
| interface_declaration
| timeunits_declaration[18]

A.1.7 Program items

program_item ::=
port_declaration **;**
| non_port_program_item

non_port_program_item ::=
{ attribute_instance } continuous_assign
| { attribute_instance } module_or_generate_item_declaration
| { attribute_instance } specparam_declaration
| { attribute_instance } initial_construct
| { attribute_instance } concurrent_assertion_item
| { attribute_instance } timeunits_declaration[18]

A.1.8 Class items

class_item ::=
{ attribute_instance } class_property
| { attribute_instance } class_method
| { attribute_instance } class_constraint
| { attribute_instance } type_declaration
| { attribute_instance } class_declaration
| { attribute_instance } timeunits_declaration[18]
| { attribute_instance } covergroup_declaration
| **;**

class_property ::=
{ property_qualifier } data_declaration

| **const** { class_item_qualifier } data_type const_identifier [**=** constant_expression] **;**

class_method ::=
{ method_qualifier } task_declaration
| { method_qualifier } function_declaration
| **extern** { method_qualifier } method_prototype **;**
| { method_qualifier } class_constructor_declaration
| **extern** { method_qualifier } class_constructor_prototype

class_constructor_prototype ::=
function new ([tf_port_list] **) ;**

class_constraint ::=
constraint_prototype
| constraint_declaration

class_item_qualifier[7] ::=
static
| **protected**
| **local**

property_qualifier[7] ::=
rand
| **randc**
| class_item_qualifier

method_qualifier[7] ::=
virtual
| class_item_qualifier

method_prototype ::=
task_prototype
| function_prototype

class_constructor_declaration ::=
function [class_scope] **new** [**(** [tf_port_list] **)**] **;**
{ block_item_declaration }
[**super . new** [**(** list_of_arguments **)**] **;**]
{ function_statement_or_null }
endfunction [**: new**]

A.1.9 Constraints

constraint_declaration ::= [**static**] **constraint** constraint_identifier constraint_block

constraint_block ::= **{** { constraint_block_item } **}**

constraint_block_item ::=
solve identifier_list **before** identifier_list **;**
| constraint_expression

constraint_expression ::=
expression_or_dist **;**
| expression –> constraint_set
| **if (** expression **)** constraint_set [**else** constraint_set]
| **foreach (** array_identifier **[** loop_variables **])** constraint_set

constraint_set ::=
constraint_expression
| **{** { constraint_expression } **}**

dist_list ::= dist_item { **,** dist_item } dist_item ::= value_range [dist_weight]

dist_weight ::=
    := expression
  | **:/** expression

constraint_prototype ::= [**static**] **constraint** constraint_identifier **;**

extern_constraint_declaration ::=
    [**static**] **constraint** class_scope constraint_identifier constraint_block

identifier_list ::= identifier { **,** identifier }

A.1.10 Package items

package_item ::=
    package_or_generate_item_declaration
  | specparam_declaration
  | anonymous_program
  | timeunits_declaration[18]

package_or_generate_item_declaration ::=
    net_declaration
  | data_declaration
  | task_declaration
  | function_declaration
  | dpi_import_export
  | extern_constraint_declaration
  | class_declaration
  | class_constructor_declaration
  | parameter_declaration **;**
  | local_parameter_declaration
  | covergroup_declaration
  | overload_declaration
  | concurrent_assertion_item_declaration
  | **;**

anonymous_program ::= **program ;** { anonymous_program_item } **endprogram**

anonymous_program_item ::=
    task_declaration
  | function_declaration
  | class_declaration
  | covergroup_declaration
  | class_constructor_declaration
  | **;**

A.2 Declarations

A.2.1 Declaration types

A.2.1.1 Module parameter declarations

local_parameter_declaration ::=
    localparam data_type_or_implicit list_of_param_assignments **;**

parameter_declaration ::=
    parameter data_type_or_implicit list_of_param_assignments
  | **parameter type** list_of_type_assignments

specparam_declaration ::=
  specparam [packed_dimension] list_of_specparam_assignments **;**

A.2.1.2 Port declarations

inout_declaration ::=
  inout net_port_type list_of_port_identifiers

input_declaration ::=
  input net_port_type list_of_port_identifiers
  | **input** variable_port_type list_of_variable_identifiers

output_declaration ::=
  output net_port_type list_of_port_identifiers
  | **output** variable_port_type list_of_variable_port_identifiers

interface_port_declaration ::=
  interface_identifier list_of_interface_identifiers
  | interface_identifier **.** modport_identifier list_of_interface_identifiers

ref_declaration ::= **ref** variable_port_type list_of_port_identifiers

A.2.1.3 Type declarations

data_declaration[15] ::=
  [**const**] [**var**] [lifetime] data_type_or_implicit list_of_variable_decl_assignments **;**
  | type_declaration
  | package_import_declaration
  | virtual_interface_declaration

package_import_declaration ::=
  import package_import_item { **,** package_import_item } **;**

package_import_item ::=
  package_identifier **::** identifier
  | package_identifier **::** *****

genvar_declaration ::= **genvar** list_of_genvar_identifiers **;**

net_declaration[14] ::=
  net_type [drive_strength | charge_strength] [**vectored** | **scalared**]
    data_type_or_implicit [delay3] list_of_net_decl_assignments **;**

type_declaration ::=
  typedef data_type type_identifier variable_dimension **;**
  | **typedef** interface_instance_identifier **.** type_identifier type_identifier **;**
  | **typedef** [**enum** | **struct** | **union** | **class**] type_identifier **;**

lifetime ::= **static** | **automatic**

A.2.2 Declaration data types

A.2.2.1 Net and variable types

casting_type ::= simple_type | constant_primary | signing

data_type ::=
  integer_vector_type [signing] { packed_dimension }
  | integer_atom_type [signing]
  | non_integer_type
  | struct_union [**packed** [signing]] **{** struct_union_member { struct_union_member } **}**
    { packed_dimension }[13]

| **enum** [enum_base_type] **{** enum_name_declaration { **,** enum_name_declaration } **}**
| **string**
| **chandle**
| **virtual** [**interface**] interface_identifier
| [class_scope | package_scope] type_identifier { packed_dimension }
| class_type
| **event**
| ps_covergroup_identifier

data_type_or_implicit ::=
data_type
| [signing] { packed_dimension }

enum_base_type ::=
integer_atom_type [signing]
| integer_vector_type [signing] [packed_dimension]
| type_identifier [packed_dimension][24]

enum_name_declaration ::=
enum_identifier [**[** integral_number [**:** integral_number] **]**] [**=** constant_expression]

class_scope ::= class_type **::**

class_type ::=
ps_class_identifier [parameter_value_assignment]
{ **::** class_identifier [parameter_value_assignment] }

integer_type ::= integer_vector_type | integer_atom_type

integer_atom_type ::= **byte** | **shortint** | **int** | **longint** | **integer** | **time**

integer_vector_type ::= **bit** | **logic** | **reg**

non_integer_type ::= **shortreal** | **real** | **realtime**

net_type ::= **supply0** | **supply1** | **tri** | **triand** | **trior** | **trireg** | **tri0** | **tri1** | **uwire** | **wire** | **wand** | **wor**

net_port_type[32] ::=
[net_type] data_type_or_implicit

variable_port_type ::= var_data_type

var_data_type ::= data_type | **var** data_type_or_implicit

signing ::= **signed** | **unsigned**

simple_type ::= integer_type | non_integer_type | ps_type_identifier | ps_parameter_identifier

struct_union_member[27] ::=
{ attribute_instance } data_type_or_void list_of_member_identifiers **;**

data_type_or_void ::= data_type | **void**

struct_union ::= **struct** | **union** [**tagged**]

A.2.2.2 Strengths

drive_strength ::=
(strength0 , strength1 **)**
| **(** strength1 , strength0 **)**
| **(** strength0 , **highz1)**
| **(** strength1 , **highz0)**
| **(highz0** , strength1 **)**
| **(highz1** , strength0 **)**

strength0 ::= **supply0** | **strong0** | **pull0** | **weak0** strength1 ::= **supply1** | **strong1** | **pull1** | **weak1**

charge_strength ::= **(small)** | **(medium)** | **(large)**

A.2.2.3 Delays

delay3 ::= **#** delay_value | **# (** mintypmax_expression [**,** mintypmax_expression [**,** mintypmax_expression]] **)**

delay2 ::= **#** delay_value | **# (** mintypmax_expression [**,** mintypmax_expression] **)**

delay_value ::=
  unsigned_number
  | real_number
  | ps_identifier
  | time_literal

A.2.3 Declaration lists

list_of_defparam_assignments ::= defparam_assignment { **,** defparam_assignment }

list_of_genvar_identifiers ::= genvar_identifier { **,** genvar_identifier }

list_of_interface_identifiers ::= interface_identifier { unpacked_dimension }
  { **,** interface_identifier { unpacked_dimension } }

list_of_member_identifiers ::=
  member_identifier variable_dimension { **,** member_identifier variable_dimension }

list_of_net_decl_assignments ::= net_decl_assignment { **,** net_decl_assignment }

list_of_param_assignments ::= param_assignment { **,** param_assignment }

list_of_port_identifiers ::= port_identifier { unpacked_dimension }
  { **,** port_identifier { unpacked_dimension } }

list_of_udp_port_identifiers ::= port_identifier { **,** port_identifier }

list_of_specparam_assignments ::= specparam_assignment { **,** specparam_assignment }

list_of_tf_variable_identifiers ::= port_identifier variable_dimension [**=** expression]
  { **,** port_identifier variable_dimension [**=** expression] }

list_of_type_assignments ::= type_assignment { **,** type_assignment }

list_of_variable_decl_assignments ::= variable_decl_assignment { **,** variable_decl_assignment }

list_of_variable_identifiers ::= variable_identifier variable_dimension
  { **,** variable_identifier variable_dimension }

list_of_variable_port_identifiers ::= port_identifier variable_dimension [**=** constant_expression]
  { **,** port_identifier variable_dimension [**=** constant_expression] }

list_of_virtual_interface_decl ::=
  variable_identifier [**=** interface_instance_identifier]
  { **,** variable_identifier [**=** interface_instance_identifier] }

A.2.4 Declaration assignments

defparam_assignment ::= hierarchical_parameter_identifier **=** constant_mintypmax_expression

net_decl_assignment ::= net_identifier { unpacked_dimension } [**=** expression]

param_assignment ::= parameter_identifier { unpacked_dimension } **=** constant_param_expression

specparam_assignment ::=
  specparam_identifier **=** constant_mintypmax_expression
  | pulse_control_specparam

type_assignment ::=
  type_identifier **=** data_type

| type_identifier = **$typeof (** expression[28] **)**
| type_identifier = **$typeof (** data_type **)**

pulse_control_specparam ::=
PATHPULSE$ = (reject_limit_value [**,** error_limit_value] **)**
| **PATHPULSE$**specify_input_terminal_descriptor**$**specify_output_terminal_descriptor
= (reject_limit_value [**,** error_limit_value] **)**

error_limit_value ::= limit_value

reject_limit_value ::= limit_value

limit_value ::= constant_mintypmax_expression

variable_decl_assignment ::=
variable_identifier variable_dimension [**=** expression]
| dynamic_array_variable_identifier **[ ]** [**=** dynamic_array_new]
| class_variable_identifier [**=** class_new]
| [covergroup_variable_identifier] **= new** [**(** list_of_arguments **)**][16]

class_new[20] ::= **new** [**(** list_of_arguments **)** | expression]

dynamic_array_new ::= **new [** expression **]** [**(** expression **)**]

A.2.5 Declaration ranges

unpacked_dimension ::= **[** constant_range **]**
| **[** constant_expression **]**

packed_dimension[11] ::=
[constant_range **]**
| unsized_dimension

associative_dimension ::=
[data_type **]**
| **[*]**

variable_dimension[12] ::=
{ sized_or_unsized_dimension }
| associative_dimension
| queue_dimension

queue_dimension ::= **[$** [**:** constant_expression] **]**

unsized_dimension[11] ::= **[ ]**

sized_or_unsized_dimension ::= unpacked_dimension | unsized_dimension

A.2.6 Function declarations

function_data_type ::= data_type | **void**

function_data_type_or_implicit ::=
function_data_type
| [signing] { packed_dimension }

function_declaration ::= **function** [lifetime] function_body_declaration

function_body_declaration ::=
function_data_type_or_implicit
[interface_identifier **.** | class_scope] function_identifier **;**
{ tf_item_declaration }
{ function_statement_or_null }
endfunction [**:** function_identifier]
| function_data_type_or_implicit

[interface_identifier **.** | class_scope] function_identifier **(** [tf_port_list] **) ;**
{ block_item_declaration }
{ function_statement_or_null }
endfunction [**:** function_identifier]

function_prototype ::= **function** function_data_type function_identifier **(** [tf_port_list] **)**

dpi_import_export ::=
import dpi_spec_string [dpi_function_import_property] [c_identifier **=**] dpi_function_proto **;**
| **import** dpi_spec_string [dpi_task_import_property] [c_identifier **=**] dpi_task_proto **;**
| **export** dpi_spec_string [c_identifier **=**] **function** function_identifier **;**
| **export** dpi_spec_string [c_identifier =] **task** task_identifier **;**

dpi_spec_string ::= **"DPI-C"** | **"DPI"**

dpi_function_import_property ::= **context** | **pure**

dpi_task_import_property ::= **context**

dpi_function_proto[8,9] ::= function_prototype

dpi_task_proto[9] ::= task_prototype

A.2.7 Task declarations

task_declaration ::= **task** [lifetime] task_body_declaration

task_body_declaration ::=
[interface_identifier **.** | class_scope] task_identifier **;**
{ tf_item_declaration }
{ statement_or_null }
endtask [**:** task_identifier]
| [interface_identifier **.** | class_scope] task_identifier **(** [tf_port_list] **) ;**
{ block_item_declaration }
{ statement_or_null }
endtask [**:** task_identifier]

tf_item_declaration ::=
block_item_declaration
| tf_port_declaration

tf_port_list ::=
tf_port_item { **,** tf_port_item }

tf_port_item[33] ::=
{ attribute_instance }
[tf_port_direction] [**var**] data_type_or_implicit
[port_identifier variable_dimension [= expression]]

tf_port_direction ::= port_direction | **const ref**

tf_port_declaration ::=
{ attribute_instance } tf_port_direction [**var**] data_type_or_implicit list_of_tf_variable_identifiers
;

task_prototype ::= **task** task_identifier **(** [tf_port_list] **)**

A.2.8 Block item declarations

block_item_declaration ::=
{ attribute_instance } data_declaration
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**
| { attribute_instance } overload_declaration

overload_declaration ::=
bind overload_operator **function** data_type function_identifier **(** overload_proto_formals **) ;**

overload_operator ::= **+** | **++** | **–** | **– –** | ***** | ****** | **/** | **%** | **==** | **!=** | **<** | **<=** | **>** | **>=** | **=**

overload_proto_formals ::= data_type {**,** data_type}

A.2.9 Interface declarations

virtual_interface_declaration ::=
virtual [**interface**] interface_identifier list_of_virtual_interface_decl **;**

modport_declaration ::= **modport** modport_item { **,** modport_item } **;**

modport_item ::= modport_identifier **(** modport_ports_declaration { **,** modport_ports_declaration } **)**

modport_ports_declaration ::=
{ attribute_instance } modport_simple_ports_declaration
| { attribute_instance } modport_hierarchical_ports_declaration
| { attribute_instance } modport_tf_ports_declaration
| { attribute_instance } modport_clocking_declaration

modport_clocking_declaration ::= **clocking** clocking_identifier

modport_simple_ports_declaration ::=
port_direction modport_simple_port { **,** modport_simple_port }

modport_simple_port ::=
port_identifier
| **.** port_identifier **(** [expression] **)**

modport_hierarchical_ports_declaration ::=
interface_instance_identifier [**[** constant_expression **]**] **.** modport_identifier

modport_tf_ports_declaration ::=
import_export modport_tf_port { **,** modport_tf_port }

modport_tf_port ::=
method_prototype
| tf_identifier

import_export ::= **import** | **export**

A.2.10 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 [**(** [list_of_arguments] **)**]

concurrent_assertion_item_declaration ::=

property_declaration
| sequence_declaration

property_declaration ::=
property property_identifier [**(** [tf_port_list] **)**] **;**
{ 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 [**(** [tf_port_list] **)**] **;**
{ 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 [**(** [list_of_arguments] **)**]

formal_list_item ::=
formal_identifier [**=** actual_arg_expr]

list_of_formals ::= formal_list_item { **,** formal_list_item }

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 ::=
var_data_type list_of_variable_identifiers **;**

A.2.11 Covergroup declarations

covergroup_declaration ::=
covergroup covergroup_identifier [**(** [tf_port_list] **)**] [coverage_event] **;**
{ coverage_spec_or_option }
endgroup [**:** covergroup_identifier]

coverage_spec_or_option ::=
{attribute_instance} coverage_spec
| {attribute_instance} coverage_option **;**

coverage_option ::=
option.member_identifier = expression
| **type_option.**member_identifier = expression

coverage_spec ::=
cover_point
| cover_cross

coverage_event ::=
clocking_event
| **@@(** block_event_expression **)**

block_event_expression ::=
block_event_expression **or** block_event_expression
| **begin** hierarchical_btf_identifier

| **end** hierarchical_btf_identifier

hierarchical_btf_identifier ::=
hierarchical_tf_identifier
| hierarchical_block_identifier
| hierarchical_identifier [class_scope] method_identifier

cover_point ::= [cover_point_identifier **:**] **coverpoint** expression [**iff (** expression **)**] bins_or_empty

bins_or_empty ::=
{ {attribute_instance} { bins_or_options **;** } **}**
| **;**

bins_or_options ::=
coverage_option
| [**wildcard**] bins_keyword bin_identifier [**[** [expression] **]**] **= {** range_list **}** [**iff (** expression **)**]
| [**wildcard**] bins_keyword bin_identifier [**[ ]**] **=** trans_list [**iff (** expression **)**]
| bins_keyword bin_identifier [**[** [expression] **]**] **= default** [**iff (** expression **)**]
| bins_keyword bin_identifier **= default sequence** [**iff (** expression **)**]

bins_keyword::= **bins** | **illegal_bins** | **ignore_bins**

range_list ::= value_range { **,** value_range }

trans_list ::= **(** trans_set **)** { **, (** trans_set **)** }

trans_set ::= trans_range_list **=>** trans_range_list { **=>** trans_range_list }

trans_range_list ::=
trans_item
| trans_item [**[*** repeat_range **]**]
| trans_item [**[–>** repeat_range **]**]
| trans_item [**[=** repeat_range **]**]

trans_item ::= range_list

repeat_range ::=
expression
| expression **:** expression

cover_cross ::= [cover_point_identifier **:**] **cross** list_of_coverpoints [**iff (** expression **)**]
select_bins_or_empty

list_of_coverpoints ::= cross_item **,** cross_item { **,** cross_item }

cross_item ::=
cover_point_identifier
| variable_identifier

select_bins_or_empty ::=
{ { bins_selection_or_option **;** } **}**
| **;**

bins_selection_or_option ::=
{ attribute_instance } coverage_option
| { attribute_instance } bins_selection

bins_selection ::= bins_keyword bin_identifier **=** select_expression [**iff (** expression **)**]

select_expression ::=
select_condition
| **!** select_condition
| select_expression **&&** select_expression
| select_expression || select_expression
| **(** select_expression **)**

select_condition ::= **binsof (** bins_expression **)** [**intersect {** open_range_list **}**]

bins_expression ::=
    variable_identifier
  | cover_point_identifier [**.** bins_identifier]

open_range_list ::= open_value_range { **,** open_value_range }

open_value_range ::= value_range[21]

A.3 Primitive instances

A.3.1 Primitive instantiation and instances

gate_instantiation ::=
    cmos_switchtype [delay3] cmos_switch_instance { **,** cmos_switch_instance } **;**
  | enable_gatetype [drive_strength] [delay3] enable_gate_instance { **,** enable_gate_instance } **;**
  | mos_switchtype [delay3] mos_switch_instance { **,** mos_switch_instance } **;**
  | n_input_gatetype [drive_strength] [delay2] n_input_gate_instance { **,** n_input_gate_instance } **;**
  | n_output_gatetype [drive_strength] [delay2] n_output_gate_instance
      { **,** n_output_gate_instance } **;**
  | pass_en_switchtype [delay2] pass_enable_switch_instance { , pass_enable_switch_instance } **;**
  | pass_switchtype pass_switch_instance { **,** pass_switch_instance } **;**
  | **pulldown** [pulldown_strength] pull_gate_instance { **,** pull_gate_instance } **;**
  | **pullup** [pullup_strength] pull_gate_instance { **,** pull_gate_instance } **;**

cmos_switch_instance ::= [name_of_instance] **(** output_terminal **,** input_terminal **,**
      ncontrol_terminal **,** pcontrol_terminal **)**

enable_gate_instance ::= [name_of_instance] **(** output_terminal **,** input_terminal **,** enable_terminal **)**

mos_switch_instance ::= [name_of_instance] **(** output_terminal **,** input_terminal **,** enable_terminal **)**

n_input_gate_instance ::= [name_of_instance] **(** output_terminal **,** input_terminal { **,** input_terminal } **)**

n_output_gate_instance ::= [name_of_instance] **(** output_terminal { **,** output_terminal } **,**
      input_terminal **)**

pass_switch_instance ::= [name_of_instance] **(** inout_terminal **,** inout_terminal **)**

pass_enable_switch_instance ::= [name_of_instance] **(** inout_terminal , inout_terminal **,**
      enable_terminal **)**

pull_gate_instance ::= [name_of_instance] **(** output_terminal **)**

A.3.2 Primitive strengths

pulldown_strength ::=
    (strength0 **,** strength1 **)**
  | **(** strength1 **,** strength0 **)**
  | **(** strength0 **)**

pullup_strength ::=
    (strength0 **,** strength1 **)**
  | **(** strength1 **,** strength0 **)**
  | **(** strength1 **)**

A.3.3 Primitive terminals

enable_terminal ::= expression

inout_terminal ::= net_lvalue

input_terminal ::= expression

ncontrol_terminal ::= expression

output_terminal ::= net_lvalue

pcontrol_terminal ::= expression

A.3.4 Primitive gate and switch types

cmos_switchtype ::= **cmos** | **rcmos**

enable_gatetype ::= **bufif0** | **bufif1** | **notif0** | **notif1**

mos_switchtype ::= **nmos** | **pmos** | **rnmos** | **rpmos**

n_input_gatetype ::= **and** | **nand** | **or** | **nor** | **xor** | **xnor**

n_output_gatetype ::= **buf** | **not**

pass_en_switchtype ::= **tranif0** | **tranif1** | **rtranif1** | **rtranif0**

pass_switchtype ::= **tran** | **rtran**

A.4 Module, interface and generated instantiation

A.4.1 Instantiation

A.4.1.1 Module instantiation

module_instantiation ::=
    module_identifier [parameter_value_assignment] hierarchical_instance { **,** hierarchical_instance }
  ;

parameter_value_assignment ::= **# (** list_of_parameter_assignments **)**

list_of_parameter_assignments ::=
    ordered_parameter_assignment { **,** ordered_parameter_assignment }
  | named_parameter_assignment { **,** named_parameter_assignment }

ordered_parameter_assignment ::= param_expression

named_parameter_assignment ::= **.** parameter_identifier **(** [param_expression] **)**

hierarchical_instance ::= name_of_instance **(** [list_of_port_connections] **)**

name_of_instance ::= instance_identifier { unpacked_dimension }

list_of_port_connections[17] ::=
    ordered_port_connection { **,** ordered_port_connection }
  | named_port_connection { **,** named_port_connection }

ordered_port_connection ::= { attribute_instance } [expression]

named_port_connection ::=
    { attribute_instance } **.** port_identifier [**(** [expression] **)**]
  | { attribute_instance } **.***

A.4.1.2 Interface instantiation

interface_instantiation ::=
    interface_identifier [parameter_value_assignment] hierarchical_instance { **,** hierarchical_instance
  } **;**

A.4.1.3 Program instantiation

program_instantiation ::=
    program_identifier [parameter_value_assignment] hierarchical_instance { **,** hierarchical_instance }
  ;

A.4.2 Generated instantiation

module_or_interface_or_generate_item[30] ::=
    module_or_generate_item
  | interface_or_generate_item

generate_region ::=
    generate { module_or_interface_or_generate_item } **endgenerate**

loop_generate_construct ::=
    for (genvar_initialization **;** genvar_expression **;** genvar_iteration **)**
        generate_block

genvar_initialization ::=
    [**genvar**] genvar_identifier **=** constant_expression

genvar_iteration ::=
    genvar_identifier assignment_operator genvar_expression
  | inc_or_dec_operator genvar_identifier
  | genvar_identifier inc_or_dec_operator

conditional_generate_construct ::=
    if_generate_construct
  | case_generate_construct

if_generate_construct ::=
    if (constant_expression **)** generate_block_or_null [**else** generate_block_or_null]

case_generate_construct ::=
    case (constant_expression **)** case_generate_item { case_generate_item } **endcase**

case_generate_item ::=
    constant_expression { **,** constant_expression } **:** generate_block_or_null
  | **default** [**:**] generate_block_or_null

generate_block ::=
    module_or_interface_or_generate_item
  | [generate_block_identifier **:**] **begin** [**:** generate_block_identifier]
        { module_or_interface_or_generate_item }
    end [**:** generate_block_identifier]

generate_block_or_null ::= generate_block | **;**

A.5 UDP declaration and instantiation

A.5.1 UDP declaration

udp_nonansi_declaration ::=
    { attribute_instance } **primitive** udp_identifier **(** udp_port_list **) ;**

udp_ansi_declaration ::=
    { attribute_instance } **primitive** udp_identifier **(** udp_declaration_port_list **) ;**

udp_declaration ::=
    udp_nonansi_declaration udp_port_declaration { udp_port_declaration }
        udp_body
    endprimitive [**:** udp_identifier]
  | udp_ansi_declaration
        udp_body
    endprimitive [**:** udp_identifier]
  | **extern** udp_nonansi_declaration
  | **extern** udp_ansi_declaration
  | { attribute_instance } **primitive** udp_identifier **(.*) ;**

{ udp_port_declaration }
udp_body
endprimitive [**:** udp_identifier]

A.5.2 UDP ports

udp_port_list ::= output_port_identifier **,** input_port_identifier { **,** input_port_identifier }

udp_declaration_port_list ::= udp_output_declaration **,** udp_input_declaration { **,** udp_input_declaration }

udp_port_declaration ::=
udp_output_declaration **;**
| udp_input_declaration **;**
| udp_reg_declaration **;**

udp_output_declaration ::=
{ attribute_instance } **output** port_identifier
| { attribute_instance } **output reg** port_identifier [**=** constant_expression]

udp_input_declaration ::= { attribute_instance } **input** list_of_udp_port_identifiers

udp_reg_declaration ::= { attribute_instance } **reg** variable_identifier

A.5.3 UDP body

udp_body ::= combinational_body | sequential_body

combinational_body ::= **table** combinational_entry { combinational_entry } **endtable**

combinational_entry ::= level_input_list **:** output_symbol **;**

sequential_body ::= [udp_initial_statement] **table** sequential_entry { sequential_entry } **endtable**

udp_initial_statement ::= **initial** output_port_identifier **=** init_val **;**

init_val ::= **1'b0** | **1'b1** | **1'bx** | **1'bX** | **1'B0** | **1'B1** | **1'Bx** | **1'BX** | **1** | **0**

sequential_entry ::= seq_input_list **:** current_state **:** next_state **;**

seq_input_list ::= level_input_list | edge_input_list

level_input_list ::= level_symbol { level_symbol }

edge_input_list ::= { level_symbol } edge_indicator { level_symbol }

edge_indicator ::= **(** level_symbol level_symbol **)** | edge_symbol

current_state ::= level_symbol

next_state ::= output_symbol | **-**

output_symbol ::= **0** | **1** | **x** | **X**

level_symbol ::= **0** | **1** | **x** | **X** | **?** | **b** | **B**

edge_symbol ::= **r** | **R** | **f** | **F** | **p** | **P** | **n** | **N** | *****

A.5.4 UDP instantiation

udp_instantiation ::= udp_identifier [drive_strength] [delay2] udp_instance { **,** udp_instance } **;**

udp_instance ::= [name_of_instance] **(** output_terminal **,** input_terminal { **,** input_terminal } **)**

A.6 Behavioral statements

A.6.1 Continuous assignment and net alias statements

continuous_assign ::=
assign [drive_strength] [delay3] list_of_net_assignments **;**
| **assign** [delay_control] list_of_variable_assignments **;**

list_of_net_assignments ::= net_assignment { **,** net_assignment }

list_of_variable_assignments ::= variable_assignment { **,** variable_assignment } net_alias ::= **alias** net_lvalue = net_lvalue { = net_lvalue } **;**

net_assignment ::= net_lvalue = expression

A.6.2 Procedural blocks and assignments

initial_construct ::= **initial** statement_or_null

always_construct ::= always_keyword statement

always_keyword ::= **always** | **always_comb** | **always_latch** | **always_ff**

final_construct ::= **final** function_statement

blocking_assignment ::=
    variable_lvalue = delay_or_event_control expression
  | hierarchical_dynamic_array_variable_identifier = dynamic_array_new
  | [implicit_class_handle **.** | class_scope | package_scope] hierarchical_variable_identifier
      select = class_new
  | operator_assignment

operator_assignment ::= variable_lvalue assignment_operator expression

assignment_operator ::=
    = | **+=** | **-=** | ***=** | **/=** | **%=** | **&=** | **|=** | **^=** | **<<=** | **>>=** | **<<<=** | **>>>=**

nonblocking_assignment ::= variable_lvalue <= [delay_or_event_control] expression

procedural_continuous_assignment ::=
    assign variable_assignment
  | **deassign** variable_lvalue
  | **force** variable_assignment
  | **force** net_assignment
  | **release** variable_lvalue
  | **release** net_lvalue

variable_assignment ::= variable_lvalue = expression

A.6.3 Parallel and sequential blocks

action_block ::=
    statement_or_null
  | [statement] **else** statement_or_null

seq_block ::=
    begin [**:** block_identifier] { block_item_declaration } { statement_or_null }
    end [**:** block_identifier]

par_block ::=
    fork [**:** block_identifier] { block_item_declaration } { statement_or_null }
    join_keyword [**:** block_identifier]

join_keyword ::= **join** | **join_any** | **join_none**

A.6.4 Statements

statement_or_null ::=
    statement
  | { attribute_instance } **;**

statement ::= [block_identifier **:**] { attribute_instance } statement_item

statement_item ::=
    blocking_assignment **;**
  | nonblocking_assignment **;**
  | procedural_continuous_assignment **;**

| case_statement
| conditional_statement
| inc_or_dec_expression **;**
| subroutine_call_statement
| disable_statement
| event_trigger
| loop_statement
| jump_statement
| par_block
| procedural_timing_control_statement
| seq_block
| wait_statement
| procedural_assertion_statement
| clocking_drive **;**
| randsequence_statement
| randcase_statement
| expect_property_statement

function_statement ::= statement

function_statement_or_null ::=
function_statement
| { attribute_instance } **;**

variable_identifier_list ::= variable_identifier { **,** variable_identifier }

A.6.5 Timing control statements

procedural_timing_control_statement ::=
procedural_timing_control statement_or_null

delay_or_event_control ::=
delay_control
| event_control
| **repeat (** expression **)** event_control

delay_control ::=
delay_value
| **# (** mintypmax_expression **)**

event_control ::=
@ hierarchical_event_identifier
| **@ (** event_expression **)**
| **@***
| **@ (*)**
| **@** sequence_instance

event_expression ::=
[edge_identifier] expression [**iff** expression]
| sequence_instance [**iff** expression]
| event_expression **or** event_expression
| event_expression **,** event_expression

procedural_timing_control ::=
delay_control
| event_control
| cycle_delay

jump_statement ::=

return [expression] **;**
| **break ;**
| **continue ;**

wait_statement ::=
wait (expression **)** statement_or_null
| **wait fork ;**
| **wait_order (** hierarchical_identifier { **,** hierarchical_identifier } **)** action_block

event_trigger ::=
-> hierarchical_event_identifier **;**
|**->>** [delay_or_event_control] hierarchical_event_identifier **;**

disable_statement ::=
disable hierarchical_task_identifier **;**
| **disable** hierarchical_block_identifier **;**
| **disable fork ;**

A.6.6 Conditional statements

conditional_statement ::=
if (cond_predicate **)** statement_or_null [**else** statement_or_null]
| unique_priority_if_statement

unique_priority_if_statement ::=
[unique_priority] **if (** cond_predicate **)** statement_or_null
{ **else if (** cond_predicate **)** statement_or_null }
[**else** statement_or_null]

unique_priority ::= **unique** | **priority**

cond_predicate ::=
expression_or_cond_pattern { **&&&** expression_or_cond_pattern }

expression_or_cond_pattern ::=
expression | cond_pattern

cond_pattern ::= expression **matches** pattern

A.6.7 Case statements

case_statement ::=
[unique_priority] case_keyword **(** expression **)** case_item { case_item } **endcase**
| [unique_priority] case_keyword **(** expression **)** **matches** case_pattern_item { case_pattern_item }
endcase

case_keyword ::= **case** | **casez** | **casex**

case_item ::=
expression { **,** expression } **:** statement_or_null
| **default** [**:**] statement_or_null

case_pattern_item ::=
pattern [**&&&** expression] **:** statement_or_null
| **default** [**:**] statement_or_null

randcase_statement ::=
randcase randcase_item { randcase_item } **endcase**

randcase_item ::= expression **:** statement_or_null

A.6.7.1 Patterns

pattern ::=

. variable_identifier
| **.***
| constant_expression
| **tagged** member_identifier [pattern]
| **'{** pattern { **,** pattern } **}**
| **'{** member_identifier **:** pattern { **,** member_identifier **:** pattern } **}**

assignment_pattern ::=
'{ expression { **,** expression } **}**
| **'{** structure_pattern_key **:** expression { **,** structure_pattern_key **:** expression } **}**
| **'{** array_pattern_key **:** expression { **,** array_pattern_key **:** expression } **}**
| **'{** constant_expression **{** expression { **,** expression } **} }**

structure_pattern_key ::= member_identifier | assignment_pattern_key

array_pattern_key ::= constant_expression | assignment_pattern_key

assignment_pattern_key ::= simple_type | **default**

assignment_pattern_expression ::=
[assignment_pattern_expression_type] assignment_pattern

assignment_pattern_expression_type ::= ps_type_identifier | ps_parameter_identifier | integer_atom_type

constant_assignment_pattern_expression[34] ::= assignment_pattern_expression

A.6.8 Looping statements

loop_statement ::=
forever statement_or_null
| **repeat (** expression **)** statement_or_null
| **while (** expression **)** statement_or_null
| **for (** for_initialization **;** expression **;** for_step **)**
statement_or_null
| **do** statement_or_null **while (** expression **) ;**
| **foreach (** array_identifier **[** loop_variables **])** statement

for_initialization ::=
list_of_variable_assignments
| for_variable_declaration { **,** for_variable_declaration }

for_variable_declaration ::=
data_type variable_identifier **=** expression { **,** variable_identifier **=** expression }

for_step ::= for_step_assignment { **,** for_step_assignment }

for_step_assignment ::=
operator_assignment
| inc_or_dec_expression
| function_subroutine_call

loop_variables ::= [index_variable_identifier] { **,** [index_variable_identifier] }

A.6.9 Subroutine call statements

subroutine_call_statement ::=
subroutine_call **;**
| **void ' (** function_subroutine_call **) ;**

A.6.10 Assertion statements

procedural_assertion_statement ::=
concurrent_assertion_statement
| immediate_assert_statement

immediate_assert_statement ::=
  assert (expression **)** action_block

A.6.11 Clocking block

clocking_declaration ::= [**default**] **clocking** [clocking_identifier] clocking_event **;**
  { clocking_item }
  endclocking [**:** clocking_identifier]

clocking_event ::=
  @ identifier
  | **@ (** event_expression **)**

clocking_item ::=
  default default_skew **;**
  | clocking_direction list_of_clocking_decl_assign **;**
  | { attribute_instance } concurrent_assertion_item_declaration

default_skew ::=
  input clocking_skew
  | **output** clocking_skew
  | **input** clocking_skew **output** clocking_skew

clocking_direction ::=
  input [clocking_skew]
  | **output** [clocking_skew]
  | **input** [clocking_skew] **output** [clocking_skew]
  | **inout**

list_of_clocking_decl_assign ::= clocking_decl_assign { , clocking_decl_assign }

clocking_decl_assign ::= signal_identifier [**=** hierarchical_identifier]

clocking_skew ::=
  edge_identifier [delay_control]
  | delay_control

clocking_drive ::=
  clockvar_expression **<=** [cycle_delay] expression
  | cycle_delay clockvar_expression **<=** expression

cycle_delay ::=
  ## integral_number
  | **##** identifier
  | **## (** expression **)**

clockvar ::= hierarchical_identifier

clockvar_expression ::= clockvar select

A.6.12 Randsequence

randsequence_statement ::= **randsequence (** [production_identifier] **)**
  production { production }
  endsequence

production ::= [function_data_type] production_identifier [**(** tf_port_list **)**] **:** rs_rule { | rs_rule } **;**

rs_rule ::= rs_production_list [**:=** weight_specification [rs_code_block]]

rs_production_list ::=
  rs_prod { rs_prod }
  | **rand join** [**(** expression **)**] production_item production_item { production_item }

weight_specification ::=

integral_number
| ps_identifier
| **(** expression **)**

rs_code_block ::= **{** { data_declaration } { statement_or_null } **}**

rs_prod ::=
production_item
| rs_code_block
| rs_if_else
| rs_repeat
| rs_case

production_item ::= production_identifier [**(** list_of_arguments **)**]

rs_if_else ::= **if (** expression **)** production_item [**else** production_item]

rs_repeat ::= **repeat (** expression **)** production_item

rs_case ::= **case (** expression **)** rs_case_item { rs_case_item } **endcase**

rs_case_item ::=
expression { **,** expression } **:** production_item **;**
| **default** [**:**] production_item **;**

A.7 Specify section

A.7.1 Specify block declaration

specify_block ::= **specify** { specify_item } **endspecify**

specify_item ::=
specparam_declaration
| pulsestyle_declaration
| showcancelled_declaration
| path_declaration
| system_timing_check

pulsestyle_declaration ::=
pulsestyle_onevent list_of_path_outputs **;**
| **pulsestyle_ondetect** list_of_path_outputs **;**

showcancelled_declaration ::=
showcancelled list_of_path_outputs **;**
| **noshowcancelled** list_of_path_outputs **;**

A.7.2 Specify path declarations

path_declaration ::=
simple_path_declaration **;**
| edge_sensitive_path_declaration **;**
| state_dependent_path_declaration **;**

simple_path_declaration ::=
parallel_path_description **=** path_delay_value
| full_path_description **=** path_delay_value

parallel_path_description ::=
(specify_input_terminal_descriptor [polarity_operator] **=>** specify_output_terminal_descriptor **)**

full_path_description ::=
(list_of_path_inputs [polarity_operator] ***>** list_of_path_outputs **)**

list_of_path_inputs ::=

specify_input_terminal_descriptor { **,** specify_input_terminal_descriptor }

list_of_path_outputs ::=
specify_output_terminal_descriptor { **,** specify_output_terminal_descriptor }

A.7.3 Specify block terminals

specify_input_terminal_descriptor ::=
input_identifier [**[** constant_range_expression **]**]

specify_output_terminal_descriptor ::=
output_identifier [**[** constant_range_expression **]**]

input_identifier ::= input_port_identifier | inout_port_identifier | interface_identifier**.**port_identifier

output_identifier ::= output_port_identifier | inout_port_identifier | interface_identifier**.**port_identifier

A.7.4 Specify path delays

path_delay_value ::=
list_of_path_delay_expressions
| **(** list_of_path_delay_expressions **)**

list_of_path_delay_expressions ::=
t_path_delay_expression
| trise_path_delay_expression **,** tfall_path_delay_expression
| trise_path_delay_expression **,** tfall_path_delay_expression **,** tz_path_delay_expression
| t01_path_delay_expression **,** t10_path_delay_expression **,** t0z_path_delay_expression **,**
tz1_path_delay_expression **,** t1z_path_delay_expression **,** tz0_path_delay_expression
| t01_path_delay_expression **,** t10_path_delay_expression **,** t0z_path_delay_expression **,**
tz1_path_delay_expression **,** t1z_path_delay_expression **,** tz0_path_delay_expression **,**
t0x_path_delay_expression **,** tx1_path_delay_expression **,** t1x_path_delay_expression **,**
tx0_path_delay_expression **,** txz_path_delay_expression **,** tzx_path_delay_expression

t_path_delay_expression ::= path_delay_expression

trise_path_delay_expression ::= path_delay_expression

tfall_path_delay_expression ::= path_delay_expression

tz_path_delay_expression ::= path_delay_expression

t01_path_delay_expression ::= path_delay_expression

t10_path_delay_expression ::= path_delay_expression

t0z_path_delay_expression ::= path_delay_expression

tz1_path_delay_expression ::= path_delay_expression

t1z_path_delay_expression ::= path_delay_expression

tz0_path_delay_expression ::= path_delay_expression

t0x_path_delay_expression ::= path_delay_expression

tx1_path_delay_expression ::= path_delay_expression

t1x_path_delay_expression ::= path_delay_expression

tx0_path_delay_expression ::= path_delay_expression

txz_path_delay_expression ::= path_delay_expression

tzx_path_delay_expression ::= path_delay_expression

path_delay_expression ::= constant_mintypmax_expression

edge_sensitive_path_declaration ::=
parallel_edge_sensitive_path_description **=** path_delay_value
| full_edge_sensitive_path_description **=** path_delay_value

parallel_edge_sensitive_path_description ::=
    ([edge_identifier] specify_input_terminal_descriptor **=>**
        (specify_output_terminal_descriptor [polarity_operator] **:** data_source_expression **))**

full_edge_sensitive_path_description ::=
    ([edge_identifier] list_of_path_inputs ***>**
        (list_of_path_outputs [polarity_operator] **:** data_source_expression **))**

data_source_expression ::= expression

edge_identifier ::= **posedge** | **negedge**

state_dependent_path_declaration ::=
    if (module_path_expression **)** simple_path_declaration
  | **if (** module_path_expression **)** edge_sensitive_path_declaration
  | **ifnone** simple_path_declaration

polarity_operator ::= **+** | **-**

A.7.5 System timing checks

A.7.5.1 System timing check commands

system_timing_check ::=
    $setup_timing_check
  | $hold_timing_check
  | $setuphold_timing_check
  | $recovery_timing_check
  | $removal_timing_check
  | $recrem_timing_check
  | $skew_timing_check
  | $timeskew_timing_check
  | $fullskew_timing_check
  | $period_timing_check
  | $width_timing_check
  | $nochange_timing_check

$setup_timing_check ::=
    $setup (data_event **,** reference_event **,** timing_check_limit [**,** [notifier]] **) ;**

$hold_timing_check ::=
    $hold (reference_event **,** data_event **,** timing_check_limit [**,** [notifier]] **) ;**

$setuphold_timing_check ::=
    $setuphold (reference_event **,** data_event **,** timing_check_limit **,** timing_check_limit
        [**,** [notifier] [**,** [stamptime_condition] [**,** [checktime_condition]
        [**,** [delayed_reference] [**,** [delayed_data]]]]]] **) ;**

$recovery_timing_check ::=
    $recovery (reference_event **,** data_event **,** timing_check_limit [**,** [notifier]] **) ;**

$removal_timing_check ::=
    $removal (reference_event **,** data_event **,** timing_check_limit [**,** [notifier]] **) ;**

$recrem_timing_check ::=
    $recrem (reference_event **,** data_event **,** timing_check_limit **,** timing_check_limit
        [**,** [notifier] [**,** [stamptime_condition] [**,** [checktime_condition]
        [**,** [delayed_reference] [**,** [delayed_data]]]]]] **) ;**

$skew_timing_check ::=
    $skew (reference_event **,** data_event **,** timing_check_limit [**,** [notifier]] **) ;**

$timeskew_timing_check ::=

$timeskew (reference_event **,** data_event **,** timing_check_limit
[**,** [notifier] [**,** [event_based_flag] [**,** [remain_active_flag]]]] **) ;**

$fullskew_timing_check ::=
$fullskew (reference_event **,** data_event **,** timing_check_limit **,** timing_check_limit
[**,** [notifier] [**,** [event_based_flag] [**,** [remain_active_flag]]]] **) ;**

$period_timing_check ::=
$period (controlled_reference_event **,** timing_check_limit [**,** [notifier]] **) ;**

$width_timing_check ::=
$width (controlled_reference_event **,** timing_check_limit **,** threshold [**,** [notifier]] **) ;**

$nochange_timing_check ::=
$nochange (reference_event **,** data_event **,** start_edge_offset **,**
end_edge_offset [**,** [notifier]] **) ;**

A.7.5.2 System timing check command arguments

checktime_condition ::= mintypmax_expression

controlled_reference_event ::= controlled_timing_check_event

data_event ::= timing_check_event

delayed_data ::=
terminal_identifier
| terminal_identifier **[** constant_mintypmax_expression **]**

delayed_reference ::=
terminal_identifier
| terminal_identifier **[** constant_mintypmax_expression **]**

end_edge_offset ::= mintypmax_expression

event_based_flag ::= constant_expression

notifier ::= variable_identifier

reference_event ::= timing_check_event

remain_active_flag ::= constant_mintypmax_expression

stamptime_condition ::= mintypmax_expression

start_edge_offset ::= mintypmax_expression

threshold ::=constant_expression

timing_check_limit ::= expression

A.7.5.3 System timing check event definitions

timing_check_event ::=
[timing_check_event_control] specify_terminal_descriptor [**&&&** timing_check_condition]

controlled_timing_check_event ::=
timing_check_event_control specify_terminal_descriptor [**&&&** timing_check_condition]

timing_check_event_control ::=
posedge
| **negedge**
| edge_control_specifier

specify_terminal_descriptor ::=
specify_input_terminal_descriptor
| specify_output_terminal_descriptor

edge_control_specifier ::= **edge [** edge_descriptor { **,** edge_descriptor } **]** edge_descriptor[1] ::= **01** | **10** | z_or_x zero_or_one | zero_or_one z_or_x

zero_or_one ::= **0** | **1**

z_or_x ::= **x** | **X** | **z** | **Z**

timing_check_condition ::=
scalar_timing_check_condition
| **(** scalar_timing_check_condition **)**

scalar_timing_check_condition ::=
expression
| ~ expression
| expression **==** scalar_constant
| expression **===** scalar_constant
| expression **!=** scalar_constant
| expression **!==** scalar_constant

scalar_constant ::= **1'b0** | **1'b1** | **1'B0** | **1'B1** | **'b0** | **'b1** | **'B0** | **'B1** | **1** | **0**

A.8 Expressions

A.8.1 Concatenations

concatenation ::=
{ expression { **,** expression } **}**

constant_concatenation ::=
{ constant_expression { **,** constant_expression } **}**

constant_multiple_concatenation ::= **{** constant_expression constant_concatenation **}**

module_path_concatenation ::= **{** module_path_expression { **,** module_path_expression } **}**

module_path_multiple_concatenation ::= **{** constant_expression module_path_concatenation **}**

multiple_concatenation ::= **{** expression concatenation **}**[19]

streaming_concatenation ::= **{** stream_operator [slice_size] stream_concatenation **}**

stream_operator ::= **>>** | **<<**

slice_size ::= simple_type | constant_expression

stream_concatenation ::= **{** stream_expression { **,** stream_expression } **}**

stream_expression ::= expression [**with [** array_range_expression **]**]

array_range_expression ::=
expression
| expression **:** expression
| expression **+:** expression
| expression **-:** expression

empty_queue[22] ::= **{ }**

A.8.2 Subroutine calls

constant_function_call ::= function_subroutine_call[25]

tf_call ::= ps_or_hierarchical_tf_identifier { attribute_instance } [**(** list_of_arguments **)**]

system_tf_call ::=
system_tf_identifier [**(** list_of_arguments **)**]
| system_tf_identifier **(** data_type [**,** expression] **)**

subroutine_call ::=
tf_call

| system_tf_call
| method_call
| randomize_call

function_subroutine_call ::= subroutine_call

list_of_arguments ::=
[expression] { **,** [expression] } { **, .** identifier **(** [expression] **)** }
| **.** identifier **(** [expression] **)** { **, .** identifier **(** [expression] **)** }

method_call ::= method_call_root **.** method_call_body

method_call_body ::=
method_identifier { attribute_instance } [**(** list_of_arguments **)**]
| built_in_method_call

built_in_method_call ::=
array_manipulation_call
| randomize_call

array_manipulation_call ::=
array_method_name { attribute_instance }
[**(** list_of_arguments **)**]
[**with (** expression **)**]

randomize_call ::=
randomize { attribute_instance }
[**(** [variable_identifier_list | **null**] **)**]
[**with** constraint_block]

method_call_root ::= expression | implicit_class_handle

array_method_name ::=
method_identifier | **unique** | **and** | **or** | **xor**

A.8.3 Expressions

inc_or_dec_expression ::=
inc_or_dec_operator { attribute_instance } variable_lvalue
| variable_lvalue { attribute_instance } inc_or_dec_operator

conditional_expression ::= cond_predicate **?** { attribute_instance } expression **:** expression

constant_expression ::=
constant_primary
| unary_operator { attribute_instance } constant_primary
| constant_expression binary_operator { attribute_instance } constant_expression
| constant_expression **?** { attribute_instance } constant_expression **:** constant_expression

constant_mintypmax_expression ::=
constant_expression
| constant_expression **:** constant_expression **:** constant_expression

constant_param_expression ::=
constant_mintypmax_expression | data_type | **$**

param_expression ::= mintypmax_expression | data_type

constant_range_expression ::=
constant_expression
| constant_part_select_range

constant_part_select_range ::=
constant_range
| constant_indexed_range

constant_range ::= constant_expression **:** constant_expression

constant_indexed_range ::=
  constant_expression **+:** constant_expression
  | constant_expression **-:** constant_expression

expression ::=
  primary
  | unary_operator { attribute_instance } primary
  | inc_or_dec_expression
  | **(** operator_assignment **)**
  | expression binary_operator { attribute_instance } expression
  | conditional_expression
  | inside_expression
  | tagged_union_expression

tagged_union_expression ::=
  tagged member_identifier [expression]

inside_expression ::= expression **inside {** open_range_list **}**

value_range ::=
  expression
  | **[** expression **:** expression **]**

mintypmax_expression ::=
  expression
  | expression **:** expression **:** expression

module_path_conditional_expression ::= module_path_expression **?** { attribute_instance }
  module_path_expression **:** module_path_expression

module_path_expression ::=
  module_path_primary
  | unary_module_path_operator { attribute_instance } module_path_primary
  | module_path_expression binary_module_path_operator { attribute_instance }
    module_path_expression
  | module_path_conditional_expression

module_path_mintypmax_expression ::=
  module_path_expression
  | module_path_expression **:** module_path_expression **:** module_path_expression

part_select_range ::= constant_range | indexed_range

indexed_range ::=
  expression **+:** constant_expression
  | expression **-:** constant_expression

genvar_expression ::= constant_expression

A.8.4 Primaries

constant_primary ::=
  primary_literal
  | ps_parameter_identifier constant_select

  | ps_specparam_identifier [constant_range_expression]

| genvar_identifier[31]
| [package_scope | class_scope] enum_identifier
| constant_concatenation
| constant_multiple_concatenation
| constant_function_call
| (constant_mintypmax_expression)
| constant_cast
| constant_assignment_pattern_expression

module_path_primary ::=
number
| identifier
| module_path_concatenation
| module_path_multiple_concatenation
| function_subroutine_call
| **(** module_path_mintypmax_expression **)**

primary ::=
primary_literal
| [implicit_class_handle **.** | class_scope | package_scope] hierarchical_identifier select
| empty_queue
| concatenation
| multiple_concatenation
| function_subroutine_call
| **(** mintypmax_expression **)**
| cast
| assignment_pattern_expression
| streaming_concatenation
| sequence_method_call
| **$**[23]
| **null**

time_literal[5] ::=
unsigned_number time_unit
| fixed_point_number time_unit

time_unit ::= **s** | **ms** | **us** | **ns** | **ps** | **fs** | **step**

implicit_class_handle[6] ::= **this** | **super** | **this . super**

bit_select ::= { **[** expression **]** }

select ::=
[{ **.** member_identifier bit_select } **.** member_identifier] bit_select [**[** part_select_range **]**]

constant_bit_select ::= { **[** constant_expression **]** }

constant_select ::=
[{ **.** member_identifier constant_bit_select } **.** member_identifier] constant_bit_select
[**[** constant_part_select_range **]**]

primary_literal ::= number | time_literal | unbased_unsized_literal | string_literal

constant_cast ::=
casting_type **' (** constant_expression **)**

cast ::=
casting_type **' (** expression **)**

A.8.5 Expression left-side values

net_lvalue ::=
    ps_or_hierarchical_net_identifier constant_select
  | { net_lvalue { , net_lvalue } }

variable_lvalue ::=
    [implicit_class_handle . | package_scope] hierarchical_variable_identifier select
  | { variable_lvalue { , variable_lvalue } }
  | streaming_concatenation[29]

A.8.6 Operators

unary_operator ::=
    + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~

binary_operator ::=
    + | - | * | / | % | == | != | === | !== | ==? | !=? | && | || | **
  | < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | << | >>> | <<<

inc_or_dec_operator ::= ++ | --

unary_module_path_operator ::=
    ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~

binary_module_path_operator ::=
    == | != | && | || | & | | | ^ | ^~ | ~^

A.8.7 Numbers

number ::=
    integral_number
  | real_number

integral_number ::=
    decimal_number
  | octal_number
  | binary_number
  | hex_number

decimal_number ::=
    unsigned_number
  | [size] decimal_base unsigned_number
  | [size] decimal_base x_digit { _ }
  | [size] decimal_base z_digit { _ }

binary_number ::= [size] binary_base binary_value

octal_number ::= [size] octal_base octal_value

hex_number ::= [size] hex_base hex_value

sign ::= + | -

size ::= non_zero_unsigned_number

non_zero_unsigned_number[1] ::= non_zero_decimal_digit { _ | decimal_digit}

real_number[1] ::=
    fixed_point_number
  | unsigned_number [. unsigned_number] exp [sign] unsigned_number

fixed_point_number[1] ::= unsigned_number . unsigned_number

exp ::= e | E

unsigned_number[1] ::= decimal_digit { _ | decimal_digit } binary_value[1] ::= binary_digit { _ | binary_digit }

octal_value[1] ::= octal_digit { _ | octal_digit }

hex_value[1] ::= hex_digit { _ | hex_digit }

decimal_base[1] ::= **'[s|S]d** | **'[s|S]D**

binary_base[1] ::= **'[s|S]b** | **'[s|S]B**

octal_base[1] ::= **'[s|S]o** | **'[s|S]O**

hex_base[1] ::= **'[s|S]h** | **'[s|S]H**

non_zero_decimal_digit ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

decimal_digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

binary_digit ::= x_digit | z_digit | **0** | **1**

octal_digit ::= x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

hex_digit ::= x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **a** | **b** | **c** | **d** | **e** | **f** | **A** | **B** | **C** | **D** | **E** | **F**

x_digit ::= **x** | **X**

z_digit ::= **z** | **Z** | **?**

unbased_unsized_literal ::= **'0** | **'1** | **'**z_or_x [10]

A.8.8 Strings

string_literal ::= **"** { Any_ASCII_Characters } **"**

A.9 General

A.9.1 Attributes

attribute_instance ::= **(*** attr_spec { **,** attr_spec } ***)**

attr_spec ::= attr_name [**=** constant_expression]

attr_name ::= identifier

A.9.2 Comments

comment ::=
    one_line_comment
  | block_comment

one_line_comment ::= **//** comment_text **\n**

block_comment ::= **/*** comment_text ***/**

comment_text ::= { Any_ASCII_character }

A.9.3 Identifiers

array_identifier ::= identifier

block_identifier ::= identifier

bin_identifier ::= identifier

c_identifier[2] ::= [**a-zA-Z_**] { [**a-zA-Z0-9_**] }

cell_identifier ::= identifier

class_identifier ::= identifier

class_variable_identifier ::= variable_identifier

clocking_identifier ::= identifier

config_identifier ::= identifier

const_identifier ::= identifier

constraint_identifier ::= identifier

covergroup_identifier ::= identifier

covergroup_variable_identifier ::= variable_identifier

cover_point_identifier ::= identifier

dynamic_array_variable_identifier ::= variable_identifier

enum_identifier ::= identifier

escaped_identifier ::= \ {any_ASCII_character_except_white_space} white_space

formal_identifier ::= identifier

function_identifier ::= identifier

generate_block_identifier ::= identifier

genvar_identifier ::= identifier

hierarchical_block_identifier ::= hierarchical_identifier

hierarchical_dynamic_array_variable_identifier ::= hierarchical_variable_identifier

hierarchical_event_identifier ::= hierarchical_identifier

hierarchical_identifier ::= [**$root .**] { identifier constant_bit_select **.** } identifier

hierarchical_net_identifier ::= hierarchical_identifier

hierarchical_parameter_identifier ::= hierarchical_identifier

hierarchical_task_identifier ::= hierarchical_identifier

hierarchical_tf_identifier ::= hierarchical_identifier

hierarchical_variable_identifier ::= hierarchical_identifier

identifier ::=
  simple_identifier
  | escaped_identifier

index_variable_identifier ::= identifier

interface_identifier ::= identifier

interface_instance_identifier ::= identifier

inout_port_identifier ::= identifier

input_port_identifier ::= identifier

instance_identifier ::= identifier

library_identifier ::= identifier

member_identifier ::= identifier

method_identifier ::= identifier

modport_identifier ::= identifier

module_identifier ::= identifier

net_identifier ::= identifier

output_port_identifier ::= identifier

package_identifier ::= identifier

package_scope ::=
  package_identifier **::**
  | **$unit ::**

parameter_identifier ::= identifier

port_identifier ::= identifier

production_identifier ::= identifier

program_identifier ::= identifier

property_identifier ::= identifier

ps_class_identifier ::= [package_scope] class_identifier

ps_covergroup_identifier ::= [package_scope] covergroup_identifier

ps_identifier ::= [package_scope] identifier

ps_or_hierarchical_net_identifier ::= [package_scope] net_identifier | hierarchical_net_identifier

ps_or_hierarchical_tf_identifier ::= [package_scope] tf_identifier | hierarchical_tf_identifier

ps_parameter_identifier ::=
  [package_scope] parameter_identifier
  | { generate_block_identifier [**[** constant_expression **]**] **.** } parameter_identifier

ps_property_identifier ::= [package_scope] property_identifier

ps_sequence_identifier ::= [package_scope] sequence_identifier

ps_specparam_identifier ::= [package_scope] specparam_identifier

ps_type_identifier ::= [package_scope] type_identifier

sequence_identifier ::= identifier

signal_identifier ::= identifier

simple_identifier[2] ::= [**a-zA-Z_**] { [**a-zA-Z0-9_$**] }

specparam_identifier ::= identifier

system_tf_identifier[3] ::= **$**[**a-zA-Z0-9_$**]{ [**a-zA-Z0-9_$**] }

task_identifier ::= identifier

tf_identifier ::= identifier

terminal_identifier ::= identifier

topmodule_identifier ::= identifier

type_identifier ::= identifier

udp_identifier ::= identifier

variable_identifier ::= identifier

A.9.4 White space

white_space ::= space | tab | newline | eof[4]

A.10 Footnotes (normative)

1) Embedded spaces are illegal.
2) A simple_identifier, c_identifier, and arrayed_reference shall start with an alpha or underscore (_) character, shall have at least one character, and shall not have any spaces.
3) The $ character in a system_tf_identifier shall not be followed by white_space. A system_tf_identifier shall not be escaped.
4) End of file.
5) The unsigned number or fixed point number in time_literal shall not be followed by a white_space.
6) implicit_class_handle shall only appear within the scope of a class_declaration or out-of-block method declaration.
7) In any one declaration, only one of **protected** or **local** is allowed, only one of **rand** or **randc** is allowed, and **static** and/or **virtual** can appear only once.

8) dpi_function_proto return types are restricted to small values, as per 28.4.5.
9) Formals of dpi_function_proto and dpi_task_proto cannot use pass by reference mode and class types cannot be passed at all; for the complete set of restrictions see 28.4.6.
10) The apostrophe (') in unbased_unsized_literal shall not be followed by white_space.
11) unsized_dimension is permitted only in declarations of import DPI functions, see dpi_function_proto.
12) More than one unsized dimension is permitted only in declarations of import DPI functions, see dpi_function_proto.
13) When a packed dimension is used with the **struct** or **union** keyword, the **packed** keyword shall also be used.
14) A charge strength shall only be used with the **trireg** keyword. When the **vectored** or **scalared** keyword is used, there shall be at least one packed dimension.
15) In a data_declaration that is not within the procedural context, it shall be illegal to use the **automatic** keyword. In a data_declaration, it shall be illegal to omit the explicit data_type before a list_of_variable_decl_assignments unless the **var** keyword is used.
16) It shall be legal to omit the covergroup_variable_identifier from a covergroup instantiation only if this implicit instantiation is within a class that has no other instantiation of the covergroup.
17) The **.*** token shall appear at most once in a list of port connections.
18) A timeunits_declaration shall be legal as a non_port_module_item, non_port_interface_item, non_port_program_item, package_item or class_item only if it repeats and matches a previous timeunits_declaration within the same time scope.
19) In a multiple_concatenation, it shall be illegal for the multiplier not to be a constant_expression unless the type of the concatenation is string.
20) In a shallow copy the expression must evaluate to an object handle.
21) It shall be legal to use the **$** primary in an open_value_range of the form **[** expression **: $]** or **[$:** expression **]**.
22) **{ }** shall only be legal in the context of a queue.
23) The **$** primary shall be legal only in a select for a queue variable or in an open_value_range.
24) A type_identifier shall be legal as an enum_base_type if it denotes an integer_atom_type, with which an additional packed dimension is not permitted, or an integer_vector_type.
25) In a constant_function_call, all arguments shall be constant_expressions.
26) The list_of_port_declarations syntax is explained in 19.8, which also imposes various semantic restrictions, e.g., a **ref** port must be of a variable type and an **inout** port must not be. It shall be illegal to initialize a port that is not a variable **output** port.
27) It shall be legal to declare a **void** struct_union_member only within tagged unions.
28) The expression that is used as the argument to the $typeof system function shall contain no hierarchical references.

29) A streaming_concatenation expression shall not be nested within another variable_lvalue. A streaming_concatenation shall not be the target of the increment or decrement operator nor the target of any assignment operator except the simple (=) or nonblocking assignment (<=) operator.

30) Within an interface_declaration, it shall only be legal for a module_or_interface_or_generate_item to be an interface_or_generate_item. Within a module_declaration, except when also within an interface_declaration, it shall only be legal for a module_or_interface_or_generate_item to be a module_or_generate_item.

31) A genvar_identifier shall be legal in a constant_primary only within a genvar_expression.

32) When a net_port_type contains a data_type, it shall only be legal to omit the explicit net_type when declaring an **inout** port.

33) In a tf_port_item, it shall be illegal to omit the explicit port_identifier except within a function_prototype or task_prototype.

34) In a constant_assignment_pattern_expression, all member expressions shall be constant expressions.

Appendix B
Verilog and SystemVerilog Reserved Keywords

The SystemVerilog-2005 standard is an extension to the Verilog-2005 standard. As part of this extension, SystemVerilog adds several new keywords to Verilog. This appendix lists:

- The original Verilog-1995 reserved keyword list
- Additional reserved keywords in the Verilog-2001 standard
- Additional reserved keywords in the Verilog-2005 standard
- Additional reserved keywords in the SystemVerilog-2005 standard

The appendix also covers compiler directives in the Verilog-2005 standard that allow mixing models together that were written based on the reserved keywords from different generations of the Verilog and SystemVerilog standards.

B.1 Verilog-1995 reserved keywords

Table B-1 lists the reserved keywords used in the Verilog language, as it was standardized by the IEEE in 1995.

always	ifnone	rpmos
and	initial	rtran
assign	inout	rtranif0
begin	input	rtranif1
buf	integer	scalared
bufif0	join	small
bufif1	large	specify
case	macromodule	specparam
casex	medium	strong0
casez	module	strong1
cmos	nand	supply0
deassign	negedge	supply1
default	nmos	table
defparam	nor	task
disable	not	time
edge	notif0	tran
else	notif1	tranif0
end	or	tranif1
endcase	output	tri
endmodule	parameter	tri0
endfunction	pmos	tri1
endprimitive	posedge	triand
endspecify	primitive	trior
endtable	pull0	trireg
endtask	pull1	vectored
event	pullup	wait
for	pulldown	wand
force	rcmos	weak0
forever	real	weak1
fork	realtime	while
function	reg	wire
highz0	release	wor
highz1	repeat	xnor
if	rnmos	xor

Table B-1: Verilog-1995 reserved keywords

B.2 Verilog-2001 reserved keywords

The IEEE 1364-2001 Verilog standard added several new keywords to the Verilog reserved keyword list. The additional keywords are listed in Table B-2.

automatic	**genvar**	**noshowcancelled**
cell	**incdir**	**pulsestyle_onevent**
config	**include**	**pulsestyle_ondetect**
design	**instance**	**showcancelled**
endconfig	**liblist**	**signed**
endgenerate	**library**	**unsigned**
generate	**localparam**	**use**

Table B-2: Verilog-2001 additional reserved keywords beyond Verilog-1995

B.3 Verilog-2005 reserved keywords

The IEEE 1364-2005 Verilog standard adds just one new keyword to the Verilog reserved keywords, which is listed in Table B-3, below.

uwire

Table B-3: Verilog-2005 additional reserved keywords beyond Verilog-2001

B.4 SystemVerilog-2005 reserved keywords

The IEEE 1800-2005 SystemVerilog standard adds a significant number of new keywords to the Verilog-2005 standard. Table B-3, lists the additional SystemVerilog keywords.

alias	endproperty	protected
always_comb	endsequence	pure
always_ff	enum	rand
always_latch	expect	randc
assert	export	randcase
assume	extends	randsequence
before	extern	ref
bind	final	return
bins	first_match	sequence
binsof	foreach	shortint
bit	forkjoin	shortreal
break	iff	solve
byte	ignore_bins	static
chandle	illegal_bins	string
class	import	struct
clocking	inside	super
const	int	tagged
constraint	interface	this
context	intersect	throughout
continue	join_any	timeprecision
cover	join_none	timeunit
covergroup	local	type
coverpoint	logic	typedef
cross	longint	union
dist	matches	unique
do	modport	var
endclass	new	virtual
endclocking	null	void
endgroup	package	wait_order
endinterface	packed	wildcard
endpackage	priority	with
endprimitive	program	within
endprogram	property	

Table B-4: SystemVerilog-2005 additional reserved keywords beyond Verilog-2005

B.5 Version compatibility

In general, each version of the Verilog standard is backward compatible with previous versions, and the SystemVerilog standard is backward compatible with Verilog. This allows models written in different versions of the standards to be mixed together in simulation, synthesis, or with other software tools.

The reserved keyword lists in later versions of the Verilog and SystemVerilog standards are not backward compatible, however. For example, if a Verilog model had been written based on the Verilog-2005 keyword list, "`priority`" is not a reserved word, and can be used as an identifier name in the source code. If, however, that source code is read in by a software tool that is using the SystemVerilog keyword list, the "`priority`" is a reserved word, and a syntax error will occur when the Verilog model is parsed.

To allow mixing models written based on different versions of reserved keywords, the Verilog-2005 standard provides a pair of compiler, **`` `begin_keywords ``** and **`` `end_keywords ``**, . These directives specify what identifiers are reserved as keywords within a block of source code, based on a specific version of the IEEE Verilog or SystemVerilog standard.

The **`` `begin_keywords ``** directive is followed by one of the following version specifiers:

```
"1364-1995"
"1364-2001"
"1364-2001-noconfig"
"1364-2005"
"1800-2005"
```

The "1364-2001-noconfig" version specifier is similar to the "1364-2001" specifier, except that the Verilog-2001 keywords used to define configurations are excluded from the reserved keyword list. The configuration keywords are: **`cell`**, **`config`**, **`design`**, **`endconfig`**, **`incdir`**, **`include`**, **`instance`**, **`liblist`**, **`library`** and **`use`**.

An example usage of the reserved keyword compatibility directives is:

```
`begin_keywords "1364-2005"   // use Verilog-2005 keywords
 module m2 (...);
   wire priority;   // OK: "priority" is not a Verilog keyword
   ...
 endmodule
`end_keywords
```

From the point where the **`begin_keywords** directive is encountered until the **`end_keywords** directive is encountered, the reserved keyword list of the specified version will be used. These directives must be specified outside of any design blocks, including modules, interfaces, programs and packages. It is illegal to attempt to change the keyword list inside a design block.

The **`begin_keywords** directive remains in effect until its corresponding **`end_keywords** directive is encountered. The directive pair can span multiple design blocks, and multiple files, when these blocks or files are read in by a single invocation of the compiler.

The **`begin_keywords** directives can be nested in the source code compilation stream. If a **`begin_keywords** directive is in effect, and a new **`begin_keywords** directive is encountered before an **`end_keywords** directive, the outer directive will be stacked, and the most recent directive will be in effect until its corresponding **`end_keywords** directive is encountered. The outer **`begin_keywords** directive will then be popped off the stack, and become in effect again.

If no **`begin_keywords** is in effect, a default keyword list for the software tool will be used. Different tools can, and often will, use different default keyword lists. Software tools typically provide one or more ways to specify what default keyword list should be used. Two common methods for specifying the default reserved keyword list are invocation options and source file extension names. A de facto standard that applies to many, but not all, software tools is that files ending with `.v` are assumed to use the reserved keyword list from Verilog-2001 or 2005, and files ending with `.sv` are assumed to use the SystemVerilog-2005 reserved keyword list.

NOTE The **`begin_keywords** directive only specifies the set of identifiers that are reserved as keywords.

The **`begin_keywords** directive does not affect the semantics, tokens and other aspects of the Verilog language. Some versions of the Verilog standard have made changes to the language semantics, and/or have added new operators to the language. How software tools handle these types of differences in versions of the standard is left up to the software tools.

Appendix C
A History of SUPERLOG, the Beginning of SystemVerilog

Simon Davidmann, one of the co-authors of this book, has been involved with the development of Hardware Description Languages since 1978. He has provided this brief history of the primary developments that have led from rudimentary gate-level modeling in the 1970s to the advanced SystemVerilog Hardware Design and Verification Language of 2005. His perspective of the development process of HDLs and the industry leaders that have brought about this evolution makes an interesting appendix to this book on using SystemVerilog for design.

C.1 Early days

The current Hardware Description Languages (HDLs) as we know them have roots in the latter part of the 20th century. The first HDL that included both register transfer and timing constructs was the HILO [1] language, developed in the late 1970s in the UK by a team at Brunel University led by Peter Flake, which included Phil Moorby and Simon Davidmann (see Photo 1, below). The language, associated simulators, and test generator were funded in part by the UK's Ministry of Defence and were targeted to produce and validate tests for PCBs and ICs. The development team at Brunel was spun out in 1983 into the UK's Cirrus Computers Ltd. and thence in 1984 into GenRad, Inc. in the USA for commercialization.

Photo 1: HILO-2 team circa 1981. (left to right) Simon Davidmann, Peter Flake, Phil Moorby, Gerry Musgrave, Bob Harris, Richard Wilson

In the early 1980s, the gate array based ASIC market started its growth to prominence. Though it had some success there, GenRad did not focus HILO development on ASIC design. Gateway Design Automation was founded in Massachusetts by Prabhu Goel specifically to build ASIC verification tools. Prabhu Goel was the first user of HILO in the U.S. Phil Moorby joined Gateway, moved to the U.S., and conceived the Verilog HDL and Verilog-XL simulator. He based this initial version of Verilog (Verilog-86) on the HILO-2 gate level language and mechanisms, improving the bidirectional capabilities, and dramatically changed the higher level constructs (borrowing from C, Pascal and Occam) while improving the timing capabilities, and making them a fundamental part of the behavioral language. Verilog-XL was a significant commercial success, partly due to the inclusion of gate level, structural, and behavioral constructs all in one language.

During the late 1980s, designers were predominantly using schematic capture packages to edit their structural designs, and gate level libraries supplied by ASIC vendors for their implementations. These vendors were very concerned about timing accuracy for design 'sign off', and so Gateway added the 'specify block' and PLI delay calculators. The certification of Verilog-XL by all the ASIC vendors, driven by Martin Harding's ASIC Business Group within Gateway, was one of the key reasons why Verilog was so successful.

In the mid 1980s, Synopsys started to work with Verilog and ASIC vendors to produce its logic optimization and re-targeting tools. The piece that was missing was the Verilog Register Transfer Level (RTL) synthesis technology, which Synopsys released in 1988/89.

By the early part of the 1990s, the design flow had changed from the 1980s methodology of schematics to Verilog RTL design and verification, Verilog RTL synthesis and functional simulation, and Verilog gate level timing simulation 'sign off'. As this move to an RTL methodology based on Verilog was taking place, Cadence Design Systems acquired Gateway, and thus took control of the (then) proprietary Verilog language. Most of the other EDA vendors did not have access to Verilog tools or a Verilog language license from Cadence, and a large number started to back the VHDL [2] language as a public standard. VHDL was developed in the early 1980s for the US Department of Defense to provide a consistent way to document chip designs, and it was first approved as an IEEE standard in 1987.

C.2 Opening up Verilog: towards an IEEE standard

HDL users in Europe and Japan are particularly keen on adopting standards, and not proprietary solutions. They started to adopt VHDL, as it was already public and an IEEE standard. Even though VHDL was originally developed as a language for documenting design, EDA vendors developed tools around it, and their customers starting using it for RTL design and verification.

In 1989, under the guidance of its Director of Strategic Marketing, Venk Shukla, Cadence responded to this swing away from Verilog by forming Open Verilog International (OVI), as a non-profit industry standards organization, donating Verilog to it, and thus placing the Verilog language and PLI into the public domain. This version became know as OVI Verilog 1.0.

OVI promoted and marketed Verilog and, by working with the IEEE, turned the Verilog HDL into the IEEE 1364 Verilog HDL (Verilog-95). There was a false start to this within OVI, as many people wanted to extend Verilog, and thus OVI quickly made many changes to the Verilog language, as donated by Cadence. This Verilog 2.0 from OVI was rejected by the IEEE committee, who selected the proven and widely used OVI Verilog 1.0 as the basis for IEEE 1364.

This OVI promotion and marketing, and IEEE standardization, stemmed the move away from Verilog. Competitive simulators such as VCS and NC-Verilog appeared and, by 2000, Verilog returned to being the dominant HDL.

C.3 Co-Design Automation

As Verilog was becoming standardized in the mid 1990s, discussion started regarding on what languages and/or language features were needed at higher levels of abstraction. Verilog was behind VHDL in this respect.

During 1995, Peter Flake and Simon Davidmann started collaborating again to develop their ideas on next generation simulators and languages for design and verification. In September 1997, they founded Co-Design Automation, Inc., which was incorporated in California with the specific business plan of developing a new simulator and a new language—ultimately called SUPERLOG, being a superset of Verilog—to augment the then current HDLs.

Many people have asked why the company that developed SUPERLOG (SystemVerilog) was called Co-Design, when the outcome of their endeavors was to evolve Verilog from being an HDL to being an integrated Hardware Design and Verification Language (HDVL). The answer is simple… the original business plan was to evolve Verilog to be of use for hardware design, software design, and verification—i.e. to be useful for codesign as well as verification—which was a significant challenge. The company succeeded in evolving Verilog to unify the design and verification tasks.

Co-Design obtained its first seed round of funding in June 1998. One of the seed investors was Andy Bechtolsheim, a co-founder of Sun Microsystems and later an engineering VP at Cisco. He was very interested to see a new HDL developed to make digital designers more productive. Another key investor in the Co-Design seed round was Rich Davenport, CEO of Simulation Technologies (developer of the VirSim simulation debugger), who shared the founders' vision and who became a Co-Design board member from inception through to final successful acquisition. Other early investors were John Sanguinetti, the developer of VCS, who went on to found C2/Cynapps/Forte and develop C/C++ synthesis, and Rajeev Madhavan who was CEO and a founder of Ambit, and then of Magma. Many of the key technology visionaries in EDA were backing the Co-Design vision of extending Verilog and creating a super Verilog.

C.4 Moving to C++ class libraries or Java: the land of the free?

April 15, 1998 was a milestone, as it saw the formation and first meeting of the OVI Architectural Language Committee (ALC). This included personnel from Cisco, Sun, National, Motorola, Cadence (owners of NC-Verilog), Viewlogic (owners of VCS) and Co-Design. It was convened to discuss 'developing an architectural/algorithmic language with verification and analysis orientation with a processor modeling extension that is targeted for advanced processor architecture development. This OVI committee work started with all good intentions, but by January 1999 had become de focussed by many people steering the committee down the route of adopting existing software languages or class libraries—the two main camps being based around C++ class libraries (two proposals) and Java based methodologies (also two proposals).

Even though there were a few believers that a better Verilog was needed, most people in the EDA industry were getting excited about C++ class libraries or Java based approaches to hardware design. This was the middle of the late 1990s internet dot com 'free' bubble, and so many people thought that it would be a good idea to find a way to use C++ or Java as a digital design language, and get all the EDA tools they would need for free ☺.

C.5 Marketing SUPERLOG

The Co-Design team saw the situation in a different light. In May 1999, Dave Kelf joined Co-Design as VP marketing and started to develop plans for informing the world about the company's direction for a unified HDL/HVL. Co-Design attended the June 1999 DAC conference and exhibition with a tiny 10ft by 10ft booth. An informative article by Peter Clarke in the US EE Times [EE1] the week before the conference caused a very busy time for Co-Design staff at the show. All employees (except Peter Heller, the CFO) attended (see Photo 2) and, being a small company, the software development engineers had to be pressed into giving demos at the exhibition booth.

At DAC 1999, the hot topic was definitely new design and verification languages. SUPERLOG/Co-Design was listed as one of the 10 'must see' items of DAC by Gary Smith of Dataquest [DQ1]. To quote from Gary in the EE Times article: "The Verilog guys are saying they have run out of steam. The VHDL guys are pretty much saying VHDL is dead. C++ is not going to work at all, and the C guys can't come up with a solution unless they really restrict the problem. Co-Design has a fair chance of establishing its language."

Photo 2: The whole of Co-Design attends DAC 1999 to launch the SUPERLOG debate—(left to right) Dave Kelf, Christian Burisch, Lee Moore, James Kenney, Simon Davidmann, Peter Flake, Matthew Hall.

In January 2000, Peter Flake made the first public technical presentation of SUPERLOG at Asia Pacific DAC (ASP-DAC) in Japan [3]. This was followed by another presentation at the HDL Conference (HDLCon) in February [4]. Later that year, in September, Simon Davidmann made a keynote presentation at the Forum on Design Languages conference (FDL) that explained the process of developing languages [5].

The idea was to add the capabilities of software programming languages and high level verification languages, all within the one familiar design language. The SUPERLOG language was continually being polished from inception through 2001, and was proven in Co-Design's simulator (SYSTEMSIM) and in its translator to Verilog (SYSTEMEX).

During 2000, as the Co-Design products were gaining acceptance with early adopters, it became obvious to many sophisticated EDA watchers and users that evolving the known and well liked Verilog HDL into a super HDL was a better approach than replacing it with a software language. This is exactly what Co-Design had pioneered with its unified HDL/HVL: SUPERLOG. Co-Design was placed under pressure by some of its partners and customers to accelerate the process of getting SUPERLOG standardized as the next generation of Verilog. Many of the engineers participating in developing the IEEE 1364 Verilog-2001 specification got very excited about SUPERLOG, and were also keen to see it become folded into the next IEEE Verilog. The press picked up on these undercurrents, and in August 2000 Richard Goering of EE Times stated “Wouldn’t it be funny if the EDA vendors pushing C/C++ for hardware design were wrong, and Co-Design’s SUPERLOG language wound up as the real next generation HDL?” [EE2]. Also, John Cooley started to have many users and supporters writing into ESNUG about their like of SUPERLOG and its direction, prompting an article in November 2000 on “the SUPERLOG evolution” [EE3].

Several EDA companies became supportive of the SUPERLOG vision, and wanted to get more involved. Dave Kelf responded to this, and created the S2K (SUPERLOG 2000) partners program, where members could get early access to SUPERLOG language technology, and help SUPERLOG on its path to industry adoption and standardization. By early 2001, the EDA world of languages started to settle into two camps: the ‘evolve Verilog camp’ centered around SUPERLOG for next generation RTL methodologies, and the C++ class library approach centered around the open source SystemC [6] class library put in the public domain by Synopsys, for high level systems modeling. While there was all this discussion regarding EDA languages, there was little, if any, discussion about evolving VHDL.

A tutorial [7] at the HDL Conference (HDLCon) in February 2001 was the first detailed disclosure of the SUPERLOG syntax. A year later at HDLCon in March 2002, Co-Design presented two tutorials: one on verification using SUPERLOG’s verification features [8] and the other on SystemVerilog (SUPERLOG) interfaces [9] and communication based design.

When building SUPERLOG, the hard challenge for the Co-Design language development team was the balance of controlling the language to make it easy, quick, and efficient to modify and improve as needed, while having a path to openness and standardization. The solution to this dilemma came with the donation of the design subset of SUPERLOG to Accellera[1] and the creation of what was initially called the Accellera Verilog++ committee. The design part of SUPERLOG was termed the Extended Synthesizable Subset (ESS) and this SUPERLOG ESS was officially donated to Accellera in May 2001.

C.6 SystemVerilog

From May 2001 through May 2002, a small group of dedicated HDL enthusiasts, EDA developers, IEEE 1364 committee members, and users worked hard in the Accellera committee, focused on turning the Co-Design donation of the SUPERLOG ESS into a public standard. Accellera was very keen on working on the SUPERLOG donation, and the Accellera Board Chairman, Dennis Brophy, and Technical Committee Chairman, Vassilios Gerousis, were very supportive. Co-Design had up to 25% of its employees attending regular Accellera committee meetings.

In May 2002, this new language extension to the Verilog HDL was approved by the Accellera board of directors, and became known as SystemVerilog 3.0 [10]. Copies of the Accellera standard were distributed at the June DAC 2002.

Meanwhile, it was announced that Intel had made a strategic investment in Co-Design. Intel has a policy of not endorsing suppliers' products, but it is interesting to note that, a year later, at DAC 2002, Intel was one of the public supporters of the SystemVerilog 3.0 standard. There they said that they had been using it for a while,

1. OVI's focus was Verilog only and, for almost 10 years, promoted Verilog with the annual International Verilog Conference in Santa Clara. With the demise of support and development for the VHDL language, OVI merged with VHDL International to form Accellera, and IVC became the HDL Conference (HDLCon), now recently renamed Design and Verification Conference (DVCon) (www.dvcon.org). Accellera is now a language neutral non-profit organization that promotes EDA language standards (www.accellera.org).

and saw it as fundamental technology for future advanced processor design.

Almost all of SystemVerilog 3.0 is SUPERLOG, but not vice-versa. Much of SUPERLOG was not donated to Accellera for SystemVerilog 3.0. A couple of features were added by the committee: data types for enumerations and implicit port connections. The SUPERLOG Design Assertion Subset was developed concurrently with the committee.

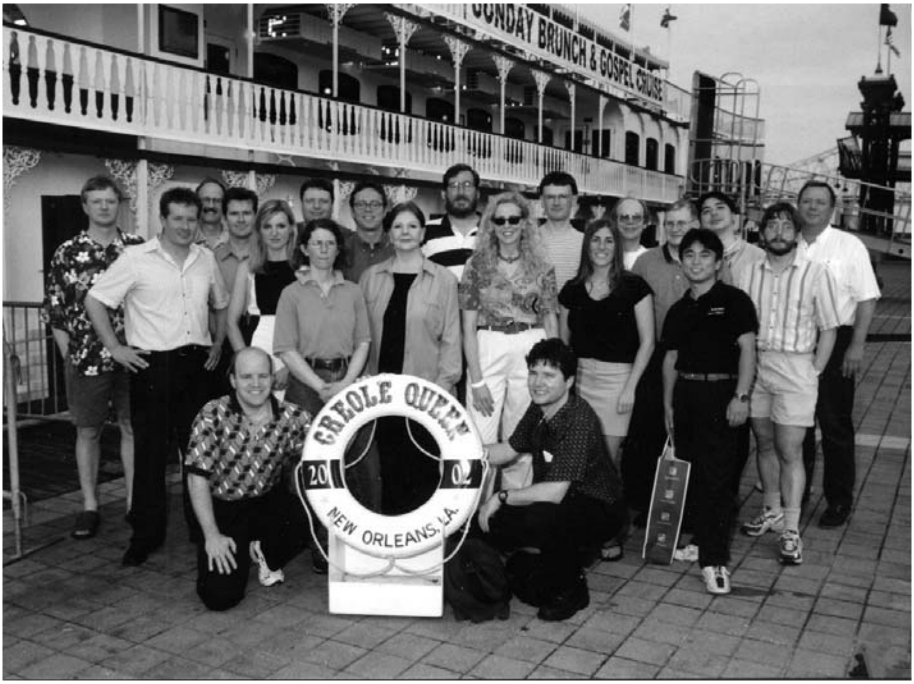


Photo 3: DAC 2002 was attended by most of the Co-Design staff.

C.7 SystemVerilog 3.1 and beyond

After the June DAC 2002, work started in Accellera on extending SystemVerilog into the testbench area, and to improve the assertions into a full temporal logic. Donations were made by other companies, with the majority coming from Synopsys. This evolution of SystemVerilog, currently at revision 3.1, was released at DAC 2003.

Co-Design was acquired by Synopsys in September 2002, and several Co-Design staff stayed involved with the Accellera SystemVerilog work.

At the Design and Verification Conference (DVCon) held in San Jose in February 2003, Aart de Geus, co-founder, Chairman, and CEO of Synopsys, delivered the keynote speech, and explained how SystemVerilog was a key component of his company's language strategy moving forward.

The benefit to users is, of course, that they will be able to design and verify in much more efficient ways than was previously possible with the older, lower level HDL capabilities.

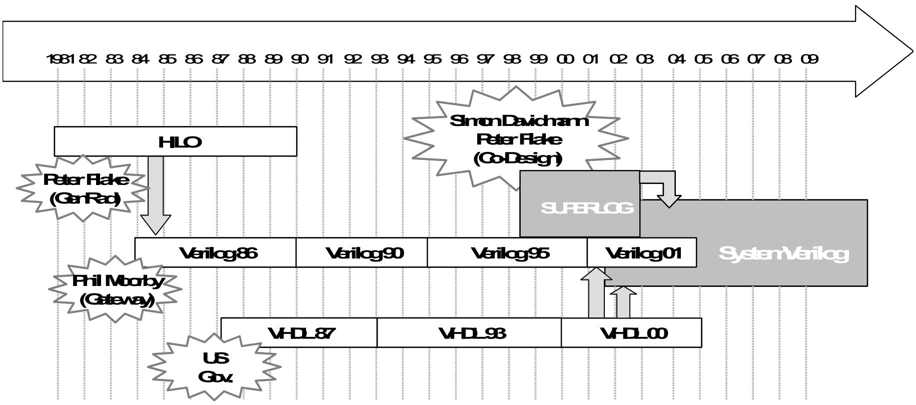


Figure 12-1: History: Evolution from HILO, Verilog, SUPERLOG to SystemVerilog

C.8 References:

[1] "The HILO Simulation Language", P.L. Flake et al, Proc. International Symposium on Computer Hardware Description Languages and their Applications, 1975.

[2] The IEEE 1076 VHDL-1987 language – developed to document US DoD designs.

[3] “SUPERLOG – a Unified Design Language for System-on-Chip.” P. Flake, S. Davidmann, ASP-DAC, Yokahama, Japan, 2000.

[4] February 2000, International HDL Conference, Santa Clara. Paper: “SUPERLOG – Evolving Verilog and C for System-on-Chip Design.” P. Flake, S. Davidmann, D. Kelf.

[5] September 2000, Forum on Design Languages, Tübingen, Germany. Keynote paper: “Evolving the Next Design Language”, Simon Davidmann, Peter Flake

[6] SystemC. (www.systemc.org) now maintained by the Open SystemC Initiative (OSCI).

[7] February 2001, International HDL Conference, San Jose. “A practical approach to System verification and hardware design”. Tutorial presented by Peter Flake and Dave Rich, which showed and explained SUPERLOG constructs: local modules, $root, explicit time, C type system, structs, typedefs, unions, 2 state variables, logic, packed/unpacked types, strings, enums, safe pointers, dynamic memory, queues, lists, bump operators, extended loops, enhanced always blocks, recursive functions, dynamic processes, interfaces and modports, explicit FSMs.

[8] March 2002 International HDL Conference, San Jose, “Advanced Verification with SUPERLOG”. Tutorial presented by Dave Rich and Tom Fitzpatrick. Like 2001 HDLcon tutorial, but also included examples of SUPERLOG associated arrays, constrained random, weighted case, semaphores, classes, polymorphism, functional coverage, assertions, CBlend (direct C interface), HW/SW platform simulation with embedded ARM core.

[9] March 2002 International HDL Conference, San Jose, “A communication based design platform: The power of SystemVerilog (SUPERLOG) interfaces” – Tutorial presented by Tom Fitzpatrick, Co-Design Automation, which focused on use model of interfaces and illustrates the use of CBlend for embedded processor simulation environment.

[10] SystemVerilog 3.0_LRM.pdf from Accellera.

[EE1] EE Times US, May 31, 1999. www.edtn.com/story/tech/OEG19990531S0003 “Startup spins next generation system design language” – Peter Clarke. An informative article that provides a very good summary of Co-Design and its SUPERLOG vision.

[EE2] EE Times, US, August 24, 2000. www.eetimes.com/story/OEG20000824S0031 “Is SUPERLOG another HDL?” – Richard Goering.

[EE3] EE Times, US, November 6, 2000. www.eetimes.com/story/OEG20001106S0024 “The SUPERLOG evolution” – John Cooley

[DQ1] Gartner Dataquest has a team focusing on analyzing the EDA market segment – Gary Smith is the leader of this group, who predicts trends. Each year at DAC, Dataquest has a pre-DAC briefing where Gary produces his ‘must see, hot technologies/companies’ list.

C.9 Who's Who in the evolution of SUPERLOG and SystemVerilog 3.0

Peter Flake – inventor of HILO language, the first HDL with timing, and developer of test generators for HILO-1and HILO-2. Co-founder of Co-Design and developer of SUPERLOG/SystemVerilog.

Phil Moorby – developed fault free and fault simulator for HILO-2. Invented the Verilog language and the Verilog-XL simulator. Became Chief Scientist at Co-Design.

Simon Davidmann – developer in the HILO team and first European employee of Gateway who developed Verilog. Joined Chronologic Simulation as one of first employees to market and sell VCS simulator in Europe. Co-Founder and CEO of Co-Design and co-developer of SUPERLOG/SystemVerilog.

Martin Harding – started and managed ASIC Business Group within Gateway making Verilog a *de facto* standard with ASIC vendors. Seed round investor in Co-Design.

Venk Shukla – Strategic marketing director within Cadence who initiated the formation of OVI to open up the Verilog language and put it on its path to IEEE standardization. Became a board member of Co-Design.

Andy Bechtolsheim – a co-founder of Sun Microsystems, developer of the Sun workstations, currently engineering VP at Cisco, and latterly a Silicon Valley angel investor. Liked vision of new HDL and became seed round investor in Co-Design.

Rich Davenport – Sales director at Gateway, founder of Simulation Technologies, and President/COO of Summit Design. Became lead investor in Co-Design seed round in 1998, shared the vision of unified design/verification language and tool. Became Co-Design board member from inception through to successful acquisition.

John Sanguinetti – founder and CEO of Chronologic Simulation, developer of VCS, the first compiled Verilog simulator. Shared the Co-Design vision of a unified HDL and became a seed round investor. Later John focused on C++ based synthesis technologies within Forte Design.

Rajeev Madhavan – founder of LogicVision, Ambit Design Systems and Magma Design Automation. Saw significant benefits in unifying the different HDL and HVL requirements and became seed round investor in Co-Design.

Dave Kelf – an early user of Verilog. Moved into marketing and was responsible for the product marketing of Cadence's NC-Verilog simulator. VP Marketing at Co-Design.

Stuart Sutherland, Cliff Cummings, Stefen Boyd, Mike McNamara, Anders Norstom, Bob Beckwith, Tom Fitzpatrick, and Kurt Baty – IEEE Verilog developers and early supporters of SUPERLOG.

Richard Goering and Peter Clarke – editors with EE Times in the US, kept a watchful eye on the 'new' language debate as it evolved, and played a key role in the industry by assessing the players and their messages, and ensuring that the lively discussions were made public and brought to their readers' attention. Over a period of 2 years, there were many front cover articles in EE Times that covered the language debate with 5 of them featuring Co-Design.

Gary Smith – Chief EDA Analyst at Gartner Dataquest. Closely watches evolving technologies and identifies trends. In 1999 identified Co-Design and SUPERLOG as a potential winner.

Raj Singh and Raj Parekh – partners at Redwood Ventures, both with significant histories in design and EDA. Started a venture capital business to invest in new technologies, became intrigued with Co-Design opportunity, and invested in first venture round. Held board seat from investment through acquisition.

Peter Heller – co-founder and CFO of Co-Design – involved with the creation of the European offices of many successful EDA startups including Verilog developers Gateway Design Automation and VCS developers Chronologic Simulation – structured Co-Design with US and UK legal entities and managed all legal and financial issues from startup through financing to ultimate acquisition by Synopsys.

Don Thomas – Professor at CMU, early pioneer in HDL methodologies, wrote the first book on Verilog with Phil Moorby. Don was a member of Co-Design's Technical Advisory Board from the beginning.

Index

Symbols

!=? operator 177
$bits system function 134
$cast dynamic cast function 69, 88
$dimensions system function 132
$high system function 133
$increment system function 133
$left system function 132
$low system function 132
$right system function 132
$size system function 133
-- operator 170–173
%= operator 174
&= operator 174
*= operator 174
++ operator 170–173
+= operator 174
.* port connections 242–244, 249
.*. port connections 275
.name port connections 238–242, 275
/= operator 174
:: scope resoluiton opeartor 10
<<<= operator 174
<<= operator 174
-= operator 174
==? operator 177
>>= operator 174
>>>= operator 174
@* 147
\" 40
^= operator 174
|= operator 174
‘" 40
‘\‘" 40
“ 41
‘define 39, 41
‘include, with packages 22
‘timescale 28
’{ ... } list of values 100, 120

Numerics

2-state data types 44, 48, 219–221
2-state operations 180
4-state data types 43

A

Accellera standards organization 2
acknowledgements xxx
alias statement 244–250
always @* 147
always_comb 52, 142–150, 199
always_ff 52, 152
always_latch 52, 150–151
anonymous enumerated type 86
anonymous structure 98
anonymous union 106
arrays
  ’{ ... } list of values 120
  copying 123
  declaration and usage 113–134
  indexing 126
  initializing 119
  packed 116–118
  passing to tasks and functions 160
  query functions 132
  unpacked 113
assignment operators 173–176
automatic 14
automatic variables 57–61

B

Backus-Naur Form 355
begin...end
  block names 192
  optional in tasks and functions 153
  variable declarations 26
behavioral modeling 330
bit data type 44
bit-stream casting 124

BNF 355
break statement 190
byte data type 44

C

case expression 195
case selection item 195
case statements
  priority 199–202
  unique........ 196–202, 208, 213
casting
  $cast dynamic casting 69, 88
  bit-stream 124
  cast operator 67, 88, 181, 182
companion book on verification xxvii
compilation unit
  description and usage 14–25
  extern module declarations 225
  timeunit declarations 32
  typedef declarations 77
const 9, 71
constants 71
continue statement 190

D

data types
  2-state 44
  4-state 43
  relaxed rules 52
decrement operator 170–173
default structure values 100
disable statement 154, 188
do...while loop 186–188
dynamic casting 69

E

enumerated types
  anonymous 86
  avoiding FSM lockup 146, 219
  base type 85
  declaration and usage 79–92
  importing from packages 82
  modeling FSMs 208–219
examples
  about xxv
  obtaining copies of xxvi
export declaration 294
extern declarations 224–226, 294
extern forkjoin 295
external (global) declarations 15

F

first method 90
for loop enhancements 182–186
foreach loop 130, 134
full_case 201
functions
  begin...end 153
  default argument type and direction 158
  default argument value 159
  empty 166
  formal arguments 157
  in packages 9
  named endfunction 165
  passing arguments by name 156
  reference arguments 161–165
  return statement 154
  void 155

G

global declarations 16

H

HDVL, definition of 2

I

IEEE 1364 Verilog standard xxvii
IEEE 1364.1 Verilog synthesis standard xxvii
IEEE 1800 SystemVerilog standard xxvii
if statements
  priority 205
  unique 203–205
import keyword 11, 289
importing package definitions 77
importing packages 11–14
increment operator 170–173
inside operator 178
int data type 44
interfaces
  concepts 264–273
  contents 273
  declarations 274
  exporting methods 293
  extern forkjoin 295
  importing methods 289
  methods 289–296
  modports 281–288
  module port declarations, explicit 277
  module port declarations, generic 278
  parameterized 297
  procedural blocks 296
  referencing signal within 279

L

labels 194
Language Reference Manual
  SystemVerilog xxvii
  Verilog xxvii
last method 90
literal values 38
localparam 9
longint data type 44
LRM
  SystemVerilog xxvii
  Verilog xxvii

M

methods ....
  first 90
  last 90
  name 90
  next 90
  num 90
  prev 90
module prototypes 224–226

N

name method 90
named end of blocks 226
named endmodule 226
named port connections 233–238
nested interface declarations 277
nested modules 227–233
net aliasing 244–250
next method 90
num method 90

O

operators
  -- 170–173
  !=? 177
  %= 174
  &= 174
  *= 174
  ++ 170–173
  += 174
  /= 174
  :: 10
  <<<= 174
  <<= 174
  -= 174
  ==? 177
  >>= 174
  >>>= 174
  ^= 174
  |= 174
  inside 178
ordered port connections 233

P

packages 8–14
  typedef declarations 76
packed arrays 116–118
packed structures 101
packed unions 109
parallel_case 201
parameter 9, 71
parameterized data types 260
port connections
  .* 242–244, 249, 275
  .name 238–242, 275
  data type rules 251
  named 233–238
  ordered 233
  ref ports 255
port declarations, simplified 258
prev method 90
priority case 199–202
priority if 205
prototypes
  interface task/function 290
  modules 224–226

R

ref module ports 255
ref task/function arguments 162–165
return statement 154, 191

S

scope resolution operator 10
shortint data type 44
shortreal data type 46
signed modifier 55
sizeof, see $bits
statement labels 194
static variables 56–61
structures
  '{ ... } list of values 100
  anonymous 98
  declaration and usage 96–105
  default values 100
  initializing 98
  net type 97
  packed 101
  passing to tasks and functions 160
  unpacked 101
  variable type 97

synthesis guidelines
  $bits 135
  ++ and -- operators 173
  2-state and 4-state data types 48
  always_comb 152
  always_comb, always_ff 152
  always_ff 152
  always_latch 152
  array query system functions 134
  assignment operators 175
  automatic variables 60
  break and continue 192
  casting 70
  compilation-unit declarations 25
  do...while loops 188
  for loops 186
  functions in packages 14
  inside operator 179
  interfaces 278, 285, 292
  priority case 201
  priority if 205
  ref ports 256
  return 192
  structures 105
  tasks in packages 14
  unions 111
  unique case 201, 214, 216
  unique if 205
  void functions 156
  wildcard equality operator 178
system functions
  $bits 134
  $dimensions 132
  $high 133
  $increment 133
  $left 132
  $low 132
  $right 132
  $size 133
SystemVerilog 3.0 2
SystemVerilog 3.1 3
SystemVerilog 3.1a 3

T

tagged unions 108
tasks
  begin...end 153
  default argument type and direction 158
  default argument value 159
  empty 166
  in packages 9
  named endtask 165
  passing arguments by name 156
  reference arguments 161–165
  return statement 154
time units and precision 28–34
timeprecision keyword 32
timeunit keyword 32
Transaction Level Modeling 329–354
type casting 67–70
type keyword 260
typedef 9
typedef, declaration and usage 75–78
types versus data types 42

U

unions
  anonymous 106
  declaration and usage 105–113
  packed 109
  tagged 108
  unpacked 106
unique case 196–202, 208, 213
unique if 203–205
unnamed blocks 27
unpacked arrays 113
unpacked structures 101
unpacked unions 106
unsigned modifier 55
user-defined types 75–78
uwire 54

V

var keyword 47, 97
variable initialization 59–67
vectors, filling 38
void data type 46
void functions 155

W

wildcard equality operator 176, 177
wildcard import 12