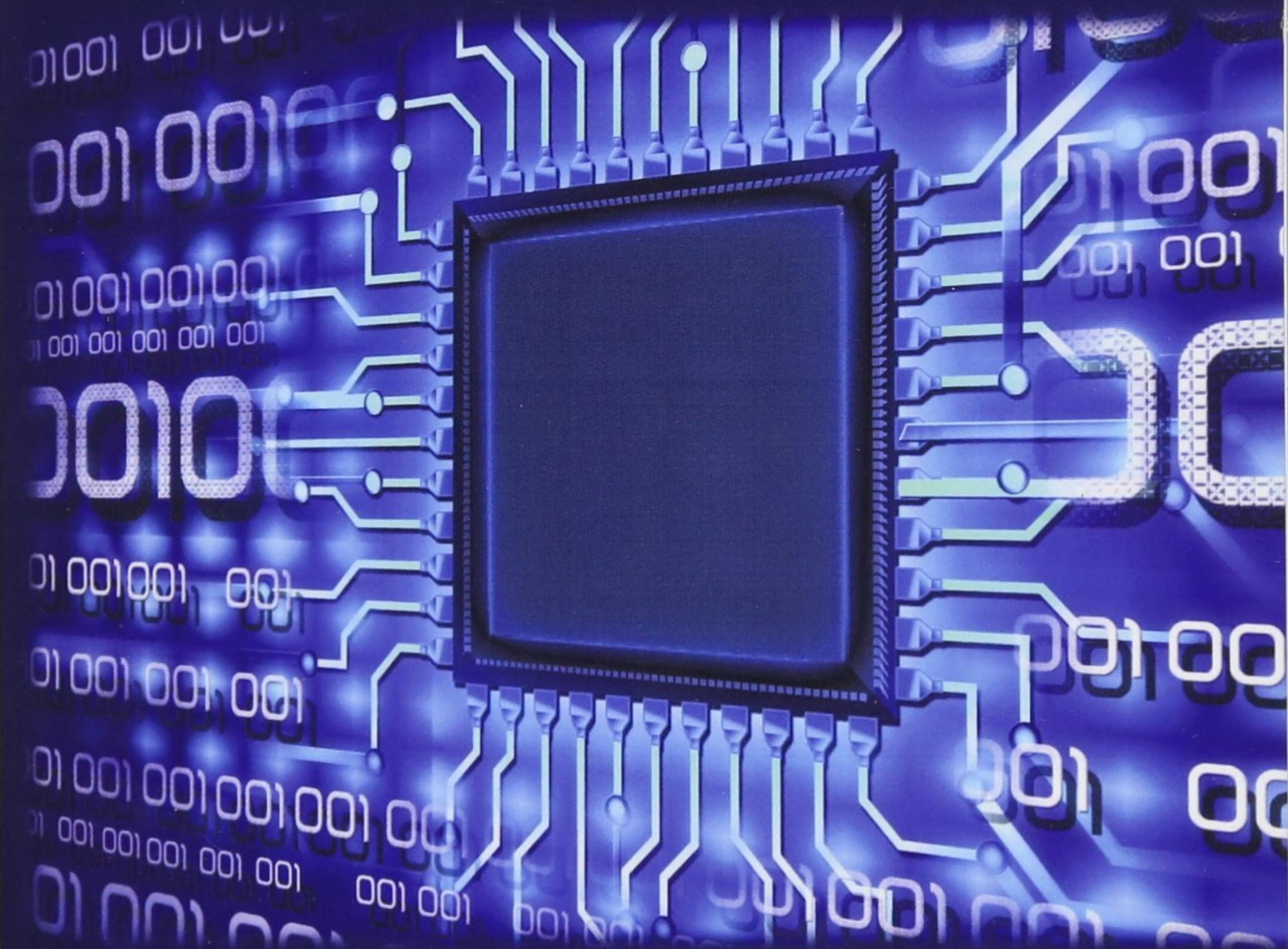


RTL Modeling with SystemVerilog for Simulation and Synthesis

using SystemVerilog for ASIC and FPGA design



Stuart Sutherland

RTL Modeling with SystemVerilog

for Simulation and Synthesis

using SystemVerilog for ASIC and FPGA design

Other books authored or co-authored by Stuart Sutherland:

Verilog and SystemVerilog Gotchas: 101 Common Coding Error and How to Avoid Them

Common coding mistakes and guidelines on how to write correct code. Co-authored with Don Mills.

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

Describes what SystemVerilog-2005 added to the Verilog-2001 language for RTL modeling. Assumes the reader is familiar with Verilog-2001. Written by Stuart Sutherland, with advice and contributions from Simon Davidmann and Peter Flake. Includes an appendix with a detailed history of Hardware Description Languages by Peter Flake.

Verilog-2001: A Guide to the New Features in the Verilog Hardware Description Language

Describes what Verilog-2001 added to the original Verilog-1995 language. Assumes the reader is familiar with Verilog-1995.

The Verilog PLI Handbook: A Tutorial and Reference Manual on the Verilog Programming Language Interface, Second Edition

A comprehensive reference and tutorial on Verilog-2001 PLI and VPI programming interfaces into Verilog simulation.

Verilog HDL Quick Reference Guide, based on the Verilog-2001 Standard

A concise reference on the syntax of the complete Verilog-2001 language.

Verilog PLI Quick Reference Guide, based on the Verilog-2001 Standard

A concise reference on the Verilog-2001 Programming Language Interface, with complete object relationship diagrams.



RTL Modeling with SystemVerilog for Simulation and Synthesis

using SystemVerilog for ASIC and FPGA design

Stuart Sutherland

SUTHERLAND
training engineers to be **HDL**
SystemVerilog and UVM Wizards
sutherland-hdl.com

published by:
Sutherland HDL, Inc.
Tualatin, Oregon, USA
sutherland-hdl.com

printed by:
CreateSpace, An Amazon.com Company
eStore: www.CreateSpace.com/7164313

ISBN-13: 978-1-5467-7634-5

ISBN-10: 1-5467-7634-6

Copyright © 2017, Sutherland HDL, Inc.

All rights reserved. This work may not be translated, copied, or reproduced in whole or in part without the express written permission of the copyright owner, except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.

The use in this work of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Sutherland HDL, Inc.
22805 SW 92nd Place
Tualatin, OR 97062-7225

E-mail: info@sutherland-hdl.com
Phone: +1-503-692-0898

URL: sutherland-hdl.com

Dedication

To my wonderful wife, LeeAnn, and my children, Ammon, Tamara, Hannah, Seth and Samuel, and each of their families — Families are forever!

*Stuart Sutherland
Portland, Oregon, USA*

About the Author

Stuart Sutherland provides expert instruction on using SystemVerilog and Verilog. He has been involved in defining the Verilog and SystemVerilog languages since the beginning of IEEE standardization work in 1993, and is a member of the IEEE SystemVerilog standards committee, where he has served as one of the technical editors for every version of the IEEE Verilog and SystemVerilog Language Reference Manuals (LRMs). Stuart has more than 25 years of experience with Verilog and SystemVerilog, and has authored and co-authored numerous papers on these languages (available at www.sutherland-hdl.com). He has also authored "*The Verilog PLI Handbook*", "*Verilog-2001: A Guide to the New Features of the Verilog HDL*", and "*SystemVerilog for Design: A Guide to Using the SystemVerilog Enhancements to Verilog for Hardware Design*" (co-authored with Simon Davidmann and Peter Flake), and "*Verilog and SystemVerilog Gotchas: 101 Common Coding Error and How to Avoid Them*" (co-authored with Don Mills)".

Stuart is the founder of *Sutherland HDL, Inc.*, which specializes in providing expert SystemVerilog training and consulting services. He holds a Bachelor's Degree in Computer Science with an emphasis in Electronic Engineering Technology (Weber State University (Ogden, Utah) and Franklin Pierce College, Nashua, New Hampshire) and a Master's Degree in Education with an emphasis on eLearning course development (Northcentral University, Prescott, Arizona).

Table of Contents

Table of Contents	ix
List of Examples	xvii
List of Figures	xxi
Foreword	xxv
Preface	xxvii
Why this book	xxvii
Intended audience for this book	xxviii
Topics covered in this book	xxviii
Book examples	xxix
Obtaining copies of the examples	xxx
Simulators and synthesis compilers used in this book	xxx
Other sources of information	xxxi
Acknowledgements	xxxi
Chapter 1: SystemVerilog Simulation and Synthesis	1
1.1 Verilog and SystemVerilog — a brief history	1
1.1.1 The Original Verilog	2
1.1.2 Open Verilog and VHDL	3
1.1.3 IEEE Verilog-95 and Verilog-2001	3
1.1.4 SystemVerilog extensions to Verilog — a separate standard	4
1.1.5 SystemVerilog replaces Verilog	5
1.2 RTL and gate-level modeling	6
1.2.1 Abstraction	6
1.2.2 Gate-level models	7
1.2.3 RTL models	10
1.2.4 Behavioral and transaction-level models	11
1.3 Defining an RTL synthesis subset of SystemVerilog	12
1.4 Modeling for ASICs and FPGAs	12
1.4.1 Standard cell ASICs	12
1.4.2 FPGAs	15
1.4.3 RTL coding styles for ASICs and FPGAs	16
1.5 SystemVerilog simulation	17
1.5.1 SystemVerilog simulators	21
1.5.2 Compilation and elaboration	21
1.5.3 Simulation time and event scheduling	23

1.6	Digital synthesis	31
1.6.1	SystemVerilog synthesis compilers	32
1.6.2	Synthesis Compilation	33
1.6.3	Constraints	34
1.7	SystemVerilog lint checkers	35
1.8	Logic Equivalence Checkers	36
1.9	Summary	37

Chapter 2: RTL Modeling Fundamentals..... **39**

2.1	Modules and procedural blocks	39
2.2	SystemVerilog language rules	40
2.2.1	Comments	40
2.2.2	White space.....	43
2.2.3	Reserved keywords	44
2.2.4	Keyword backward compatibility — ‘begin_keywords.....	46
2.2.5	Identifiers (user-defined names)	49
2.2.6	Naming conventions and guidelines	50
2.2.7	System tasks and functions	51
2.2.8	Compiler directives.....	52
2.3	Modules	52
2.4	Modules instances and hierarchy	54
2.4.1	Port order connections	55
2.4.2	Named port connections	56
2.4.3	The dot-name inferred named port connection shortcut.....	57
2.4.4	The dot-star inferred named port connection shortcut.....	58
2.5	Summary	59

Chapter 3: Net and Variable types **61**

3.1	Four-state data values	61
3.2	Literal values (numbers)	62
3.2.1	Literal integer values	62
3.2.2	Vector fill literal values	65
3.2.3	Floating-point literal values (real numbers).....	66
3.3	Types and data types	66
3.3.1	Net types and variable types	66
3.3.2	Two-state and four-state data types (bit and logic).....	67
3.4	Variable types	67
3.4.1	Synthesizable variable data types	67
3.4.2	Variable declaration rules	70
3.4.3	Variable assignment rules.....	74
3.4.4	Uninitialized variables	74
3.4.5	In-line variable initialization.....	75
3.5	Net types	76
3.5.1	Synthesizable net types.....	77
3.5.2	Net declaration rules	79

3.5.3	Implicit net declarations.....	80
3.5.4	Net assignment and connection rules.....	83
3.6	Port declarations	84
3.6.1	Synthesizable port declarations	84
3.6.2	Non synthesizable port declarations	87
3.6.3	Module port declaration recommendations	88
3.7	Unpacked arrays of nets and variables	89
3.7.1	Accessing array elements.....	90
3.7.2	Copying arrays.....	91
3.7.3	Array list assignments.....	91
3.7.4	Bit-select and part-select of array elements.....	92
3.8	Parameter constants	93
3.8.1	Parameter declarations	94
3.8.2	Parameter overrides (parameter redefinition)	97
3.9	Const variables	99
3.10	Summary	99

Chapter 4: User-defined Types and Packages 101

4.1	User-defined types	101
4.1.1	Naming conventions for user-defined types	102
4.1.2	Local typedef definitions	102
4.1.3	Shared typedef definitions	102
4.2	SystemVerilog packages	102
4.2.1	Package declarations.....	103
4.2.2	Using package items	104
4.2.3	Importing from multiple packages.....	108
4.2.4	Package chaining	109
4.2.5	Package compilation order.....	110
4.2.6	Synthesis considerations	111
4.3	The \$unit declaration space	112
4.4	Enumerated types	114
4.4.1	Enumerated type declaration syntax	114
4.4.2	Importing enumerated types from packages	117
4.4.3	Enumerated type assignment rules	118
4.4.4	Enumerated type methods.....	121
4.4.5	Traditional Verilog coding style without enumerated types.....	124
4.5	Structures	124
4.5.1	Structure declarations	124
4.5.2	Assigning to structure members	125
4.5.3	Assigning to entire structures	125
4.5.4	Typed and anonymous structures	126
4.5.5	Copying structures	127
4.5.6	Packed and unpacked structures	127
4.5.7	Passing structures through ports and to tasks and functions.....	129
4.5.8	Traditional Verilog versus structures.....	130

4.5.9	Synthesis considerations	130
4.6	Unions	131
4.6.1	Typed and anonymous unions	131
4.6.2	Assigning to, and reading from, union variables	132
4.6.3	Unpacked, packed and tagged unions.....	132
4.6.4	Passing unions through ports and to tasks and functions.....	134
4.7	Using arrays with structures and unions	136
4.8	Summary	139

Chapter 5: RTL Expression Operators.....141

5.1	Operator expression rules	141
5.1.1	4-state and 2-state operations.....	142
5.1.2	X-optimism and X-pessimism	142
5.1.3	Expression vector sizes and automatic vector extension.....	144
5.1.4	Signed and unsigned expressions	145
5.1.5	Integer (vector) and real (floating-point) expressions	145
5.2	Concatenate and replicate operators	146
5.3	Conditional (ternary) operator	150
5.4	Bitwise operators	153
5.5	Reduction operators	158
5.6	Logical operators	160
5.6.1	Difference between negate and invert operations.....	161
5.6.2	Short circuiting logical operations.....	163
5.6.3	Non-synthesizable logical operators	164
5.7	Comparison operators (equality and relational)	164
5.8	Case equality (identity) operators	168
5.9	Set membership (inside) operator	171
5.10	Shift operators	173
5.10.1	Synthesizing shift operations.....	174
5.10.2	Synthesizing rotate operations	177
5.11	Streaming operators (pack and unpack)	181
5.12	Arithmetic operators	184
5.12.1	Integer and floating-point arithmetic	186
5.12.2	Unsigned and signed arithmetic might synthesize to the same gates	188
5.13	Increment and decrement operators	189
5.13.1	Proper usage of increment and decrement operators	190
5.13.2	An example of correct usage of increment and decrement operators	192
5.13.3	Compound operations with increment and decrement operators.....	194
5.13.4	An anecdotal story on the increment and decrement operators	195
5.14	Assignment operators	196
5.15	Cast operators and type conversions	198
5.15.1	Type casting.....	200
5.15.2	Size casting	202
5.15.3	Signedness casting	206

5.16 Operator precedence	209
5.17 Summary	210

Chapter 6: RTL Programming Statements 211

6.1 SystemVerilog procedural blocks	211
6.1.1 Sensitivity lists.....	212
6.1.2 Begin-end statement groups.....	214
6.1.3 Using variables and nets in procedural blocks.....	216
6.2 Decision statements	216
6.2.1 if-else statements.....	216
6.2.2 Case statements.....	222
6.2.3 Unique and priority decision modifiers	227
6.3 Looping statements	228
6.3.1 For loops	228
6.3.2 Repeat loops.....	233
6.3.3 While and do-while loops	235
6.3.4 Foreach loops and looping through arrays.....	236
6.4 Jump statements	238
6.4.1 The continue and break jump statements.....	239
6.4.2 The disable jump statement	240
6.5 No-op statement	241
6.6 Functions and tasks in RTL modeling	243
6.6.1 Functions.....	243
6.6.2 Tasks	248
6.7 Summary	249

Chapter 7: Modeling Combinational Logic 251

7.1 Continuous assignments (Boolean expressions)	252
7.1.1 Explicit and inferred continuous assignments	254
7.1.2 Multiple continuous assignments.....	254
7.1.3 Using both continuous assignments and always procedures	255
7.2 The always and always_comb procedures	256
7.2.1 Synthesizing combinational logic always procedures	257
7.2.2 Modeling with the general purpose always procedure	257
7.2.3 Modeling with the RTL-specific always_comb procedure.....	260
7.2.4 Using blocking (combinational logic) assignments.....	261
7.2.5 Avoiding unintentional latches in combinational logic procedures.....	262
7.3 Using functions to represent combinational logic	263
7.4 Combinational logic decision priority	265
7.4.1 Removing unnecessary priority encoding from case decisions	266
7.4.2 The unique and unique0 decision modifiers	266
7.4.3 The obsolete parallel_case synthesis pragma	270
7.5 Summary	271

Chapter 8: Modeling Sequential Logic.....273

8.1	RTL models of flip-flops and registers	274
8.1.1	Synthesis requirements for RTL sequential logic.....	274
8.1.2	Always procedures and always_ff procedures.....	275
8.1.3	Sequential logic clock-to-Q propagation and setup/hold times	276
8.1.4	Using nonblocking (sequential logic) assignments	278
8.1.5	Synchronous and asynchronous resets.....	286
8.1.6	Multiple clocks and clock domain crossing (CDC).....	295
8.1.7	Additional RTL sequential logic modeling considerations	297
8.2	Modeling Finite State Machines (FSMs)	299
8.2.1	Mealy and Moore FSM architectures	301
8.2.2	State encoding.....	302
8.2.3	One, two and three-procedure FSM coding styles.....	305
8.2.4	A complete FSM example	309
8.2.5	Reverse case statement one-hot decoder	313
8.2.6	Avoiding latches in state machine decoders	317
8.3	Modeling memory devices such as RAMs	317
8.3.1	Modeling asynchronous and synchronous memory devices.....	319
8.3.2	Loading memory models using \$readmemb and \$readmemh	320
8.4	Summary	322

Chapter 9: Modeling Latches and Avoiding Unintentional Latches323

9.1	Modeling Latches	323
9.1.1	Modeling latches with the general purpose always procedure	324
9.1.2	Modeling latches with the always_latch procedure	325
9.2	Unintentional latch inference	327
9.3	Avoiding latches in intentionally incomplete decisions	329
9.3.1	Latch avoidance coding style trade-offs	330
9.3.2	A small example to illustrate avoiding unintentional latches	332
9.3.3	Latch avoidance style 1 — Default case item with known values	335
9.3.4	Latch avoidance style 2—Pre-case assignment, known values	338
9.3.5	Latch avoidance style 3 — unique and priority decision modifiers	340
9.3.6	Latch avoidance style 4 — X assignments for unused decision values.....	345
9.3.7	Latch avoidance style 5 — the full_case synthesis pragma.....	350
9.3.8	Additional notes about synthesis pragmas.....	351
9.4	Summary	353

Chapter 10: Modeling Communication Buses — Interface Ports355

10.1	Interface port concepts	356
10.1.1	Traditional Verilog bus connections.....	357
10.1.2	SystemVerilog interface definitions	361
10.1.3	Referencing signals within an interface.....	365
10.1.4	Differences between modules and interfaces.....	365
10.1.5	Source code declaration order.....	366
10.2	Using interfaces as module ports	366

10.2.1	Generic interface ports	366
10.2.2	Type-specific interface ports	367
10.3	Interface modports	367
10.3.1	Specifying which modport view to use.....	368
10.3.2	Using modports to define different sets of connections	371
10.4	Interface methods (tasks and functions)	372
10.4.1	Calling methods defined in an interface	374
10.4.2	Synthesizing interface methods	375
10.4.3	Abstract, non-synthesizable interface methods	375
10.5	Interface procedural code	376
10.6	Parameterized interfaces	378
10.7	Synthesizing interfaces	379
10.8	Summary	382

List of Appendices	383
---------------------------------	------------

Appendix A: Best Practice Coding Guidelines	385
--	------------

Appendix B: SystemVerilog Reserved Keywords	391
--	------------

B.1	All SystemVerilog-2012 reserved keywords	391
B.2	Verilog-1995 reserved keywords	393
B.3	Verilog-2001 reserved keywords	394
B.4	Verilog-2005 reserved keywords	394
B.5	SystemVerilog-2005 reserved keywords	395
B.6	SystemVerilog-2009 reserved keywords	396
B.7	SystemVerilog-2012 reserved keywords	396
B.8	SystemVerilog-2017 reserved keywords	396

Appendix C: X Optimism and X Pessimism in RTL Models	397
---	------------

C.1	Introducing My X	398
C.2	How did my one (or zero) become my X?	399
C.2.1	Uninitialized 4-state variables	399
C.2.2	Uninitialized registers and latches	400
C.2.3	Low power logic shutdown or power-up	401
C.2.4	Unconnected module input ports	401
C.2.5	Multi-driver Conflicts (Bus Contention)	401
C.2.6	Operations with an unknown result	401
C.2.7	Out-of-range bit-selects and array indices	401
C.2.8	Logic gates with unknown output values	402
C.2.9	Setup or hold timing violations	402
C.2.10	User-assigned X values in hardware models	402
C.2.11	Testbench X injection	403

C.3	An optimistic X — is that good or bad?	403
C.3.1	If...else statements.....	404
C.3.2	Case statements without a default-X assignment	407
C.3.3	Casex, casez and case...inside statements.....	408
C.3.4	Bitwise, unary reduction, and logical operators	411
C.3.5	And, nand, or, nor, logic primitives.....	412
C.3.6	User-defined primitives	412
C.3.7	Array index with X or Z bits for write operations	412
C.3.8	Net data types.....	413
C.3.9	Posedge and negedge edge sensitivity	414
C.4	A pessimistic X — is that any better?	415
C.4.1	If...else statements with X assignments	416
C.4.2	Conditional operator	417
C.4.3	Case statements with X assignments	419
C.4.4	Edge-sensitive X pessimism	420
C.4.5	Bitwise, unary reduction, and logical operators	420
C.4.6	Equality, relational, and arithmetic operators.....	421
C.4.7	User-defined primitives	422
C.4.8	Bit-select, part-select, array index on right-hand side of assignments	423
C.4.9	Shift operations	423
C.4.10	X-pessimism summary	424
C.5	Eliminating my X by using 2-state simulation	424
C.6	Eliminating some of my X with 2-state data types	426
C.7	Breaking the rules—simulator-specific X-propagation options	428
C.8	Changing the rules — A SystemVerilog enhancement wish list	429
C.9	Detecting and stopping my X at the door	430
C.10	Minimizing problems with my X	432
C.10.1	2-state versus 4-state guidelines	432
C.10.2	Register initialization guidelines	433
C.10.3	X-assignment guidelines.....	433
C.10.4	Trapping X guidelines	433
C.11	Conclusions	434
C.11.1	About the author	435
C.12	Acknowledgments	435
C.13	References	436
	Appendix D: Additional Resources.....	437
	Index	441

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, referred to as *snippets*, that illustrate specific features of SystemVerilog. The source code for the full examples can be downloaded from <http://www.sutherland-hdl.com>. Navigate the menus to “***SystemVerilog Book Examples***”.

The Preface provides more details regarding the code examples in this book.

Chapter 1: SystemVerilog Simulation and Synthesis

Example 1-1:	SystemVerilog gate-level model of 1-bit adder with carry	8
Example 1-2:	SystemVerilog RTL model of 1-bit adder with carry	10
Example 1-3:	SystemVerilog RTL model of 32-bit adder/subtractor	11
Example 1-4:	Design model with input and output ports (a 32-bit adder/subtractor)	18
Example 1-5:	Testbench for the 32-bit adder/subtractor model	18
Example 1-6:	Top-level module connecting the testbench to the design	20
Example 1-7:	A clock oscillator, stimulus and flip flop to illustrate event scheduling	29

Chapter 2: RTL Modeling Fundamentals

Example 2-1:	RTL model showing two styles of comments	41
Example 2-2:	SystemVerilog RTL model with minimum white space	44
Example 2-3:	SystemVerilog RTL model with good use of white space	44
Example 2-4:	Using ‘ begin_keywords ’ with a legacy Verilog-2001 model	47
Example 2-5:	Using ‘ begin_keywords ’ with a SystemVerilog-2012 model	48

Chapter 3: Net and Variable types

Example 3-1:	Example of undeclared identifiers creating implicit nets	80
Example 3-2:	Changing the net type for implicit nets	81
Example 3-3:	Module port declaration using recommended coding guidelines	89
Example 3-4:	Add module with parameterized port widths	94
Example 3-5:	Model of a configurable RAM using a module parameter list	96
Example 3-6:	Adder with configurable data types	97

Chapter 4: User-defined Types and Packages

Example 4-1:	A package definition with several package items	103
Example 4-2:	Using a package wildcard import	105
Example 4-3:	Importing specific package items into a module	106
Example 4-4:	Explicit package references using the :: scope resolution operator	107
Example 4-5:	Using enumerated type methods for a state machine sequencer	123
Example 4-6:	Package containing structure and union definitions	134

Example 4-7: Arithmetic Logical Unit (ALU) with structure and union ports	135
Example 4-8: Using arrays of structures to model an instruction register.....	137

Chapter 5: RTL Expression Operators

Example 5-1: Using concatenate operators: multiple input status register.....	147
Example 5-2: Using concatenate operators: adder with a carry bit	149
Example 5-3: Using the conditional operator: multiplexed 4-bit register D input	151
Example 5-4: Using the conditional operator: 4-bit adder with tri-state outputs	152
Example 5-5: Using bitwise operators: multiplexed N-bit wide AND/XOR operation	156
Example 5-6: Using reduction operators: parity checker using XOR	159
Example 5-7: Using logical operators: set flag when values are within a range	163
Example 5-8: Using comparison operators: a relationship comparator.....	167
Example 5-9: Using case equality operators: a comparator for high address range.....	170
Example 5-10: Using the set membership operator: a decoder for specific addresses.....	172
Example 5-11: Using the shift operator: divide-by-two by shifting right one bit	175
Example 5-12: Using the shift operator: multiply by a power of two by shifting left.....	176
Example 5-13: Performing a rotate operation using concatenate and shift operators	179
Example 5-14: Using the streaming operator: reverse bits of a parameterized vector	183
Example 5-15: Using arithmetic operators with unsigned data types	186
Example 5-16: Using arithmetic operators with signed data types	187
Example 5-17: Using arithmetic operators with real data types	187
Example 5-18: Using increment and decrement operators	192
Example 5-19: Using assignment operators	197
Example 5-20: Using size casting.....	205
Example 5-21: Using sign casting for a mixed signed and unsigned comparator	207

Chapter 6: RTL Programming Statements

Example 6-1: Using if-else to model multiplexor functionality	219
Example 6-2: Using if without else to model latch functionality	220
Example 6-3: Using an if-else-if series to model a priority encoder	220
Example 6-4: Using if-else-if series to model a flip-flop with reset and chip-enable	221
Example 6-5: Using a case statement to model a 4-to-1 MUX.....	225
Example 6-6: Using an case-inside to model a priority encoder	226
Example 6-7: Using a for loop to operate on bits of vectors.....	229
Example 6-8: Using a for loop to find the lowest bit that is set in a vector	231
Example 6-9: Using a repeat loop to raise a value to the power of an exponent.....	234
Example 6-10: Controlling for loop execution using continue and break	239

Chapter 7: Modeling Combinational Logic

Example 7-1: Add, multiply, subtract dataflow processing with registered output	255
Example 7-2: Function that defines an algorithmic multiply operation	264
Example 7-3: State decoder with inferred priority encoded logic (partial code).....	267
Example 7-4: State decoder with unique parallel encoded logic (partial code)	268

Chapter 8: Modeling Sequential Logic

Example 8-1: RTL model of a 4-bit Johnson counter.....	279
Example 8-2: 4-bit Johnson counter incorrectly modeled with blocking assignments	282
Example 8-3: RTL model of an 8-bit serial-to-parallel finite state machine.....	310

Chapter 9: Modeling Latches and Avoiding Unintentional Latches

Example 9-1: Using intentional latches for a cycle-stealing pipeline.....	326
Example 9-2: Simple round-robin state machine that will infer latches.....	334

Chapter 10: Modeling Communication Buses — Interface Ports

Example 10-1: Master and slave module connections using separate ports.....	358
Example 10-2: An interface definition for the 8-signal simple AMBA AHB bus	362
Example 10-3: Master and slave modules with interface ports	363
Example 10-4: Netlist connecting the master and slave interface ports	364
Example 10-5: Interface with modports for custom views of interface signals	371
Example 10-6: Interface with internal methods (functions) for parity logic	373
Example 10-7: Interface with internal procedural code to generate bus functionality	376
Example 10-8: Parameterized interface with configurable bus data word size	378

List of Figures

Chapter 1: SystemVerilog Simulation and Synthesis

Figure 1-1:	Verilog-95 and Verilog-2001 language features	4
Figure 1-2:	Verilog-2005 with SystemVerilog language extensions	5
Figure 1-3:	SystemVerilog modeling abstraction levels	7
Figure 1-4:	1-bit adder with carry, represented with logic gates	8
Figure 1-5:	Typical RTL-based ASIC design flow	13
Figure 1-6:	Typical RTL-based FPGA design flow	16
Figure 1-7:	Simulation time line and time slots	26
Figure 1-8:	Simplified SystemVerilog event scheduling flow	28
Figure 1-9:	Simulation time line and time slots with some events scheduled	30
Figure 1-10:	SystemVerilog synthesis tool flow	31
Figure 1-11:	Diagram of a simple circuit requiring synthesis constraints	34

Chapter 2: RTL Modeling Fundamentals

Figure 2-1:	SystemVerilog module contents	53
Figure 2-2:	Design partitioning using sub blocks	54

Chapter 3: Net and Variable types

Figure 3-1:	Vectors with subfields	73
-------------	------------------------------	----

Chapter 4: User-defined Types and Packages

Figure 4-1:	State diagram for a confidence counter state machine	122
Figure 4-2:	Packed structures are stored as a vector	128
Figure 4-3:	Packed union with two representations of the same storage	133
Figure 4-4:	Synthesis result for Example 4-7: ALU with structure and union ports	136
Figure 4-5:	Synthesis result for Example 4-8: instruction register with structures	138

Chapter 5: RTL Expression Operators

Figure 5-1:	Synthesis result for Example 5-1: Concatenate operator (status register)	148
Figure 5-2:	Synthesis result for Example 5-2: Add operator (adder with carry in/out)	149
Figure 5-3:	Synthesis result for Example 5-3: Conditional operator (mux'ed register)	151
Figure 5-4:	Synthesis result for Example 5-4: Conditional operator (tri-state output)	152
Figure 5-5:	Synthesis result for Example 5-5: Bitwise AND and OR operations	157
Figure 5-6:	Synthesis result for Example 5-6: Reduction XOR (parity checker)	159
Figure 5-7:	Synthesis result for Example 5-7: Logical operators (in-range compare)	163
Figure 5-8:	Synthesis result for Example 5-8: Relational operators (comparator)	168
Figure 5-9:	Synthesis result for Example 5-9: Case equality, ==? (comparator)	170
Figure 5-10:	Synthesis result for Example 5-10: Inside operator (boundary detector)	172
Figure 5-11:	Bitwise and arithmetic shift operations	174
Figure 5-12:	Synthesis result for Example 5-11: Shift operator, right-shift by 1 bit	175
Figure 5-13:	Synthesis result for Example 5-12: Shift operator, variable left shifts	176
Figure 5-14:	Rotate a variable number of times using concatenate and shift operators	178
Figure 5-15:	Synthesis result for Example 5-13: Concatenate and shift (rotate)	179

Figure 5-16: Synthesis result for Example 5-14: Streaming operator (bit reversal)	183
Figure 5-17: Synthesis result for Example 5-15: Arithmetic operation, unsigned	187
Figure 5-18: Synthesis result for Example 5-16: Arithmetic operation, signed	187
Figure 5-19: Synthesis result for Example 5-18: Increment and decrement operators	193
Figure 5-20: Synthesis result after mapping to a Xilinx Virtex®-7 FPGA	193
Figure 5-21: Synthesis result after mapping to a Xilinx CoolRunner™-II CPLD	194
Figure 5-22: Synthesis result for Example 5-19: Assignment operators	197
Figure 5-23: Synthesis result for Example 5-20: Size casting	205
Figure 5-24: Synthesis result for Example 5-21: Sign casting	208

Chapter 6: RTL Programming Statements

Figure 6-1: Synthesis result for Example 6-1: if-else as a MUX	219
Figure 6-2: Synthesis result for Example 6-2: if-else as a latch	220
Figure 6-3: Synthesis result for Example 6-3: if-else as a priority encoder	221
Figure 6-4: Synthesis result for Example 6-4: if-else as a chip-enable flip-flop	222
Figure 6-5: Synthesis result for Example 6-5: case statement as a 4-to-1 MUX	226
Figure 6-6: Synthesis result for Example 6-6: case...inside as a priority encoder	227
Figure 6-7: Synthesis result for Example 6-7: for-loop to operate on vector bits	230
Figure 6-8: Synthesis result for Example 6-8: for-loop to find lowest bit set	232
Figure 6-9: Synthesis result for Example 6-9: repeat loop to raise to an exponent	234
Figure 6-10: Synthesis result for Example 6-10	240

Chapter 7: Modeling Combinational Logic

Figure 7-1: Synthesis result for Example 7-1: Continuous assignment as comb. logic	255
Figure 7-2: Synthesis result for Example 7-2: Function as combinational logic	264
Figure 7-3: Synthesis result for Example 7-3: Reverse case statement with priority	267
Figure 7-4: Synthesis result for Example 7-4: Reverse case statement, using unique	268

Chapter 8: Modeling Sequential Logic

Figure 8-1: 4-bit Johnson counter diagram	277
Figure 8-2: Simplified SystemVerilog event scheduling flow	279
Figure 8-3: Synthesis result for Example 8-1: Nonblocking assignments, J-Counter	280
Figure 8-4: Synthesis result for Example 8-2: Blocking assignments, bad J-Counter	283
Figure 8-5: Blocking assignment to intermediate temporary variable	284
Figure 8-6: Nonblocking assignment to intermediate temporary variable	284
Figure 8-7: Synthesis result: Async reset DFF mapped to Xilinx Virtex®-6 FPGA	288
Figure 8-8: Synthesis result: Async reset mapped to Xilinx CoolRunner™-II CPLD	288
Figure 8-9: Waveform showing result of incorrectly modeled asynchronous reset	289
Figure 8-10: Synthesis result for a chip-enable flip-flop	290
Figure 8-11: External logic to create the functionality of a chip-enable flip-flop	290
Figure 8-12: Synthesis result for an asynchronous set-reset flip-flop	293
Figure 8-13: Two flip-flop clock synchronizer for 1-bit control signals	296
Figure 8-14: An 8-bit serial value of hex CA, plus a start bit	299
Figure 8-15: State flow for an 8-bit serial-to-parallel Finite State Machine	300
Figure 8-16: Primary functional blocks in a Finite State Machine	305
Figure 8-17: Functional block diagram for a serial-to-parallel finite state machine	310
Figure 8-18: Synthesis result for Example 8-3: Simple-SPI using a state machine	312

Chapter 9: Modeling Latches and Avoiding Unintentional Latches

Figure 9-1:	Synthesis result for Example 9-1: Pipeline with intentional latches	327
Figure 9-2:	Round-robin Finite State Machine state flow	332
Figure 9-3:	Synthesis result for Example 9-2: FSM with unintended latches	335
Figure 9-4:	Synthesis result when using a default case item to prevent latches	336
Figure 9-5:	Synthesis result using a pre-case assignment to prevent latches	339
Figure 9-6:	Synthesis result when using a unique case statement to prevent latches	343
Figure 9-7:	Synthesis result using a default case X assignment to prevent latches	347
Figure 9-8:	Synthesis results when using a pre-case X assignment	348

Chapter 10: Modeling Communication Buses — Interface Ports

Figure 10-1:	Block diagram connecting a Master and Slave using separate ports	357
Figure 10-2:	Block diagram connecting a Master and Slave using interface ports	361

Appendix A: Best Practice Coding Guidelines**Appendix B: SystemVerilog Reserved Keywords****Appendix C: X Optimism and X Pessimism in RTL Models**

Figure C-1:	Flip-flop with synchronous reset	404
Figure C-2:	2-to-1 selection — MUX gate implementation	405
Figure C-3:	2-to-1 selection — NAND gate implementation	405
Figure C-4:	Clock divider with pessimistic X lock-up	416

Appendix D: Additional Resources

Foreword

*by Phil Moorby
The creator of the Verilog language*

Verilog is now over 30 years old, and has spanned the years of designing with graphical schematic entry tools of a few thousand gates, to modern RTL design using tools supporting millions, if not billions, of gates, all following the enduring prediction of Moore's law. Verilog addressed the simulation and verification 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.

Behind the scenes, there has been a steady process of inventing and learning what was needed and what worked (and what did not work!) to improve the language to keep up with the inevitable growth demands. From the public's point of view, there were the stepping-stones from one published standard to the next: the first published standard in 1995, the eagerly awaited update of Verilog in 2001, the final of the older Verilog standard in 2005, and the matured SystemVerilog standard in 2012, just to name some of the main stones.

I have always held the belief that for hardware designers to achieve their best in inventing new ideas they must think (if not dream) in a self contained, consistent and concise language. It is often said when learning a new natural language that your brain doesn't get it until you realize that you are speaking it in your dreams.

Over the last 15 years, Verilog has been extended and matured into the SystemVerilog language of today, and includes major new abstract constructs, test-bench verification, formal analysis, and C-based API's. 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. Many published textbooks on the design side of the new SystemVerilog assumed that the reader was familiar with Verilog, and simply explained the new extensions. It is time to leave behind the stepping-stones and to teach a single consistent and concise language in a single book, and maybe not even refer to the old ways at all!

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 learn the design aspects of SystemVerilog and the future of hardware design.

Happy inventing...

*Phil Moorby,
Montana Systems, Inc.
Massachusetts, 2016*

Preface

SystemVerilog, officially the **IEEE Std 1800™** standard, is a “*Hardware Design and Verification Language*”. The language serves a dual purpose: to model digital design behavior, and to program verification testbenches to stimulate and verify the design models.

This book is based on the **IEEE Std 1800-2012** and proposed **IEEE Std 1800-2017** SystemVerilog standards. The 1800-2012 SystemVerilog standard was the version currently in use at the time this book was written. The 1800-2017 standard was in the process of being finalized.

SystemVerilog is the latest generation of what was originally called **Verilog**. SystemVerilog adds powerful language constructs for modeling and verifying the behavior of designs that are ever increasing in size and complexity. These extensions to Verilog fall into two major groups: design modeling enhancements, and verification enhancements.

This book, ***RTL Modeling with SystemVerilog for Simulation and Synthesis***, focuses on using SystemVerilog for modeling digital ASIC and FPGA designs at the RTL level of abstraction. A companion book, ***SystemVerilog for Verification***¹, covers verifying correct functionality of large, complex designs.

Why this book

I (Stuart Sutherland) teach corporate-level SystemVerilog training workshops for companies throughout the world, and provide SystemVerilog consulting services. As a course developer and trainer, I have been disappointed with the offering of SystemVerilog books for design and synthesis. There are a few books that offer a primer-like overview of SystemVerilog, many books that focus on the verification aspects of SystemVerilog, and several books that cover the long-obsolete Verilog-2001 language for hardware design. A few of these older Verilog based books have been updated to show some SystemVerilog features, but the traditional Verilog roots are still evident in the coding styles and examples of those books.

This book addresses these shortcomings. The book was written with SystemVerilog as its starting point, rather than starting with traditional Verilog and adding SystemVerilog features. The focus is writing RTL models of digital designs, using SystemVerilog constructs that are synthesizable for both ASIC or FPGA devices. Proper coding styles for simulation and synthesis are emphasized throughout the book.

1. Chris Spear and Greg Tumbush, “*SystemVerilog for Verification, Third Edition*”, New York, NY: Springer 2012, 978-1-4614-0715-7.

Intended audience for this book

This book is for all engineers who are involved with digital IC design. The book is intended to serve as both a learning guide and a reference manual on the RTL synthesis subset of the SystemVerilog language. The book presents SystemVerilog in the context of examples, with an emphasis on correct, best-practice coding styles.

NOTE

This book assumes the reader is already familiar with digital logic design.

The text and examples in this book assume and require an understanding of digital logic. Concepts such as AND, OR and Exclusive-OR gates, multiplexors, flip-flops, and state machines are not defined in this book. This book can be a useful resource in conjunction with learning and applying digital design engineering skills.

Topics covered in this book

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

Chapter 1 presents a brief overview of simulating and synthesizing the SystemVerilog language. The major differences between SystemVerilog and traditional Verilog are also presented.

Chapter 2 provides an overview of RTL modeling in SystemVerilog. Topics include SystemVerilog language rules, design partitioning, and netlists.

Chapter 3 goes into detail on the many data types in SystemVerilog, and which data types are useful in RTL modeling. The appropriate use of 2-state and 4-state types is discussed. The chapter also presents using data arrays as synthesizable, RTL modeling constructs.

Chapter 4 presents user-defined types, including enumerated types, structures, and unions. The use of packages as a place to declare user-defined types is also covered.

Chapter 5 explains the many programming operators in SystemVerilog, and shows how to use these operators to code accurate and deterministic RTL models.

Chapter 6 covers the programming statements in SystemVerilog, with an emphasis on proper RTL coding guidelines in order to ensure the code will synthesize to the gate-level implementation intended. Several programming statements that SystemVerilog adds to the original Verilog language make it possible to model using fewer lines of code compared to standard Verilog.

Chapter 7 gives an in-depth look at writing RTL models of combinational logic. Best-practice coding recommendations are given for writing models that will simulate and synthesize correctly.

Chapter 8 examines the correct way to model RTL sequential logic behavior. Topics include synchronous and asynchronous resets, set/reset flip-flops, chip-enable flip-flops, and memory devices, such as RAMs.

Chapter 9 presents the proper way to model latches in RTL models, and how to avoid unintentional latches.

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

Appendix A summarizes the best-practice coding guidelines and recommendations that are made in each chapter of the book.

Appendix B lists the set of reserved keywords for each generation of the Verilog and SystemVerilog standards.

Appendix C is a reprint of a paper entitled *I'm Still In Love With My X*, regarding how X values propagate in RTL models. The paper recommends ways to minimize or catch potential problems with X-optimism and X-pessimism in RTL models.

Appendix D lists some additional resources that are closely related to the topics discussed in this book.

Book examples

The examples in this book illustrate specific SystemVerilog constructs in a realistic, though small, context. Complete code examples list the code between two horizontal lines, as shown below. This book use a convention of showing all SystemVerilog keywords in bold.

SystemVerilog RTL model of 32-bit adder/subtractor (same as Example 1-3, page 11)

```
module rtl_adder_subtractor
  (input logic          clk,    // 1-bit scalar input
   input logic          mode,   // 1-bit scalar input
   input logic [31:0] a, b,   // 32-bit vector inputs
   output logic [31:0] sum); // 32-bit vector output
);

  always_ff @(posedge clk) begin
    if (mode == 0) sum <= a + b;
    else           sum <= a - b;
  end
endmodule: rtl_adder_subtractor
```

Each chapter also contains many shorter examples, referred to a *code snippets*. These snippets are not complete models, and are not encapsulated between horizontal lines. The full source code, such as variable declarations, is not included in these code

snippets. This was done in order to focus on specific aspects of SystemVerilog constructs without clutter from surrounding code.

Obtaining copies of the examples

The complete code for all the examples listed in this book is available for personal, non-commercial use. They can be downloaded from the **Sutherland HDL** website, at sutherland-hdl.com/books/sv_rtl_synthesis/sv_rtl_synthesis_book_examples.zip.

Simulators and synthesis compilers used in this book

NOTE

This book strives to be vendor and software tool neutral. While specific products were used to test the examples in this book, all examples should run with any simulator or synthesis compiler that adheres to the IEEE 1800-2012 SystemVerilog standard.

The examples in this book have been tested with multiple simulation and synthesis tools, including (listed alphabetically by company name):

- The **Cadence Genus RTL Compiler[®]** synthesis compiler.
- The *Intel* (formerly *Altera*) **Quartus[®] Prime** synthesis compiler.
- The *Mentor Graphics* **QuestaTM** simulator and **Precision RTL SynthesisTM** compiler.
- The *Synopsys* **VCS[®]** simulator, **DC-Ultra[®]** synthesis compiler, and **Synplify-Pro[®]** synthesis compiler. The **SpyGlass[®]** Lint RTL rule checker was also used with certain examples.
- The *Xilinx* **Vivado[®]** synthesis compiler.

The software versions used for testing the book examples were the latest versions available to the author in Q1-2017. (A few tools did not support a SystemVerilog language feature used in one or two examples, but will probably support those language features in future versions of the tool.)

The *Mentor Graphics Precision RTL SynthesisTM* compiler was used to generate the synthesis schematic output shown with many of the examples. This compiler was selected because the schematics created by this tool were easy to capture in black-and-white, and to adapt to the page size of the book.

Other sources of information

Some other resources which can serve as companions to this book include:

IEEE Std 1800-2012, SystemVerilog Language Reference Manual LRM)—IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language.

Copyright 2013, IEEE, Inc., New York, NY. ISBN 978-0-7381-8110-3. 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. It can be downloaded for free from <https://standards.ieee.org/getieee/1800/download/1800-2012.pdf>.

SystemVerilog for Verification—A Guide to Learning the Testbench Language Features, third edition by Chris Spear and Greg Tumbush.

Copyright 2012, Springer, New York, New York. ISBN 978-1-4614-0715-7.

The Spear and Tumbush book is a companion to this book, with a focus on the verification side of SystemVerilog. For more information, refer to the publisher's web site: <http://www.springer.com/engineering/circuits+%26+systems/book/978-1-4614-0714-0>.

Additional resources related to the topics in this book are listed in Appendix D.

Acknowledgements

I am grateful to all those who have helped with this book. I would like to specifically thank those that provided invaluable feedback by reviewing specific chapters the book for technical content and accuracy. These reviewers include: **Leah Clark, Clifford Cummings, Steve Golson, Kelly Larson, Don Mills and Chris Spear**. I am also grateful to **Shalom Bresticker**, who answered many technical questions over the period of time that I wrote this book.

Special recognition is extended to **Don Mills**, who provided valuable feedback and assistance throughout the writing process. Don recommended ideas for many of the book examples, and helped with testing the code examples on multiple simulators and synthesis compilers.

I am especially appreciative of **Phil Moorby**, the creator of the original Verilog language and simulator, for writing the foreword for this book and for creating a long-lasting design and verification language for the digital design industry.

I would also like to recognize and thank my wonderful wife, **LeeAnn Sutherland**, for her painstaking reviews of this book for grammar, punctuation and readability.

Chapter 1

SystemVerilog Simulation and Synthesis

Abstract — This chapter explores the general concepts of modeling hardware using SystemVerilog, and the roles of simulation and synthesis in the hardware design flow. Some of the major topics presented in this section are:

- The difference between Verilog and SystemVerilog
- RTL and gate-level modeling
- Defining an RTL synthesis subset of SystemVerilog
- Modeling ASICs and FPGAs
- Model verification testbenches
- The role and usage of digital simulation with SystemVerilog
- The role and usage of digital synthesis with SystemVerilog
- The role and usage of SystemVerilog lint checkers

1.1 Verilog and SystemVerilog — a brief history

Verilog and *SystemVerilog* are synonymous names for the same Hardware Description Language (HDL). *SystemVerilog* is the newer name for the official IEEE language standard, and replaces the original *Verilog* name.

Verilog began as a proprietary design language in the early 1980s, for use with a digital simulator sold by Gateway Design Automation. The proprietary Verilog HDL was opened to the public domain in 1989, and standardized by the IEEE as an international standard in 1995 as **IEEE Std 1364-1995™** (commonly referred to as “**Verilog-95**”). The IEEE updated the Verilog standard in 2001 as the **1364-2001™** standard, referred to as “**Verilog-2001**”. The last official version under the Verilog name was **IEEE Std 1364-2005™**. In that same year, the IEEE released an extensive set of enhancements to the Verilog HDL. These enhancements were initially documented under a different standards number and name, the **IEEE Std 1800-2005™ SystemVerilog** standard. In 2009, the IEEE terminated the IEEE-1364 standard, and merged Verilog-2005 into the SystemVerilog standard, with the standards number **IEEE Std 1800-2009™** standard. Additional design and verification enhancements were added in 2012, as the **IEEE Std 1800-2012™** standard, referred to as **System-**

Verilog-2012. At the time this book was writing, the IEEE was nearing completion of a proposed **IEEE Std 1800-2017™** standard, or **SystemVerilog-2017**. This version only corrects errata in the 2012 version of the standard, and adds clarifications on the language syntax and semantic rules.

1.1.1 The Original Verilog

Verilog began in the early 1980s as a proprietary *Hardware Description Language* (HDL) from a company called Gateway Design Automation. The primary author of the original Verilog HDL is Phil Moorby. In the early 1980s, digital simulation was becoming popular. Several *Electronic Design Automation* (EDA) companies provided digital simulators, but there were no standard Hardware Description Languages to use with these simulators. Instead, each simulator company provided a proprietary modeling language specific to that simulator. Gateway Design Automation was no different. The simulator product was named “Verilog-XL” (short for “Verification Logic, Accelerated”), and its accompanying modeling language was called “Verilog”.

The Verilog-XL simulator and the Verilog HDL became the dominant simulator and language for digital design in the latter half of the 1980s. Some factors that attributed to this popularity included: 1) speed and capacity, 2) ASIC timing accuracy, 3) an integrated design and verification language, and 4) digital synthesis.

1. The Verilog-XL simulator was faster and had a larger design size capacity than most, if not all, of its contemporary competing simulators, allowing companies to more efficiently design larger, more complex digital integrated circuits (ICs).
2. In the latter half of the 1980s, many electronic design companies were switching from custom ICs to Application Specific ICs (ASICs). Gateway Design Automation worked closely with major ASIC suppliers, and Verilog-XL became the golden reference simulator for ensuring timing accurate ASIC simulations. This preference by ASIC suppliers helped make Verilog a preferred language for companies involved in designing ASICs.
3. The major digital simulators in the 1970s and early 1980s typically involved working with two proprietary languages: a gate-level modeling language to model the digital logic, and a separate proprietary language to model stimulus and response checking for simulation. Gateway Design Automation departed from this tradition, and integrated gate-level modeling, abstract functional modeling, stimulus and response checking all into a single language, called Verilog.
4. The fourth reason many companies adopted the Verilog language for the design of ASICs was the ability to synthesize abstract Verilog models into gate-level models. In the latter half of the 1980s, Synopsys, Inc. struck an agreement with Gateway Design Automation to use the proprietary Verilog language with the Synopsys Design Compiler (DC) digital synthesis tool. The ability to both simulate and synthesize the Verilog language was a tremendous advantage over all other proprietary digital modeling languages at that time.

1.1.2 Open Verilog and VHDL

The rapid growth and popularity of the Verilog language hit a sudden slow down at the beginning of the 1990s. The Institute of Electrical and Electronics Engineers (IEEE, often pronounced as “I-triple-E”) released the VHDL language as the first industry standard, non-proprietary Hardware Description Language. Similar to Verilog, VHDL also provided an integrated digital modeling and verification language, with support from ASIC vendors (at first by using certified Verilog ASIC libraries in a VHDL design flow). As VHDL simulators and synthesis compilers became available, many design companies began to shy away from using proprietary languages, including Verilog. Other factors, such as a mandate from the United States Department of Defense (DoD) to use VHDL as a documentation language for designs being created for the DoD, also led to a movement away from Verilog and towards VHDL. (The DoD did not require that design work be done in VHDL, only that the final documentation be in VHDL.)

Gateway Design Automation countered this move away from proprietary HDLs by preparing to release Verilog to the public domain. This preparation required separating the Verilog language documentation from the Verilog-XL simulator product documentation. While this work was in progress, Gateway Design Automation was acquired by Cadence Design Systems. Cadence completed the effort, and Verilog was officially made a public domain language in 1991. A not-for-profit organization called Open Verilog International (OVI) was formed to control the public Verilog language and to promote its usage.

The release of Verilog to the public domain effectively stemmed the flow away from Verilog towards VHDL. For the next two decades, the two HDLs co-existed, and, arguably, maintained a somewhat even overall usage in the world-wide electronic design industry. The advent of SystemVerilog in 2005, however, tipped the balance away from VHDL, and Verilog, under its new name of SystemVerilog, has again become the more dominant HDL used in digital design and verification.

1.1.3 IEEE Verilog-95 and Verilog-2001

The IEEE took over the public domain Verilog language in 1993, and released an official IEEE Verilog HDL standard two years later, as IEEE 1364-1995, nicknamed “*Verilog-95*”. Five years later, the IEEE released 1364-2001, nicknamed “*Verilog-2001*”, with a number of enhancements for modeling and verifying digital designs.

Figure 1-1 shows the major language features in Verilog-95, and the major new features that were added in Verilog-2001. Note that this figure is not intended to be an all-comprehensive list of language features. The intent is to show the major new features Verilog-2001 added to the original Verilog language.

Figure 1-1: Verilog-95 and Verilog-2001 language features

Verilog-2001			
ANSI C style ports	standard file I/O	(* attributes *)	multi dimensional arrays
generate	\$value\$plusargs	configurations	signed types
localparam	'ifndef `elsif `line	memory part selects	automatic
constant functions	@*	variable part select	** (power operator)
Verilog-1995 (created in 1984)			
modules	\$finish \$fopen \$fclose	initial	begin-end + = * /
parameters	\$display \$write	disable	while %
function/tasks	\$monitor	events	for forever >> <<
always @	'define `ifdef `else	wait # @	if-else
assign	'include `timescale	fork-join	repeat
		2D memory	

1.1.4 SystemVerilog extensions to Verilog — a separate standard

By 2001, the size and complexity of the typical digital IC had evolved tremendously from the 1980s, when both the Verilog and VHDL languages were first created. Even with the new features added in Verilog-2001, it was becoming more and more difficult to efficiently model these large designs and the massive amounts of stimulus and response checking code needed to verify these complex designs.

To address the language limitations of Verilog-2001, work was begun on defining a substantial set of new features for the Verilog language. These extensions were generalized into two primary categories:

- Enhancements primarily addressing the needs of modeling digital logic functionality more efficiently and more accurately.
- Enhancements for writing efficient, race-free verification code for very large, complex designs.

The initial work for defining this next generation of Verilog was done outside of the IEEE, by an independent not-for-profit organization called Accellera (now Accellera Systems Initiative). Accellera is a think-tank organization comprised of representatives from companies that develop Electronic Design Automation (EDA) software tools and companies that use those software tools. Accellera was formed in the mid 1990s by the merger of the Verilog and VHDL user groups. Later, other EDA groups also merged into Accellera, such as the SystemC Initiative. Accellera is responsible for the initial development efforts of many of the EDA engineering standards in use today. Many of these Accellera standards eventually became IEEE standards.

In late 2002, Accellera released the first version of these major extensions to be added to the IEEE Verilog-2001 language. During the development of these extensions to Verilog-2001, these new language features were referred to as “Verilog++”, but a last minute decision was made to release the extensions as “SystemVerilog 3.0”. The use of 3.0 was chosen to show that, when the extensions were combined with Verilog, it would be the third generation of the Verilog language (with Verilog-95 being the first generation, and Verilog-2001 the second generation). Accellera continued to define more extensions to Verilog, and a year later, in 2003, released a SystemVerilog 3.1 standard.

It is important to note that the Accellera SystemVerilog 3.1 document was not a complete, stand-alone language. It was a set of extensions to the IEEE 1364-2001 Verilog language. Accellera's initial intent was that the IEEE would then add these extensions to the next version of the IEEE 1364 Verilog standard, targeted to be 1364-2005, nicknamed Verilog-2005. For multiple reasons, however, the IEEE Verilog standards committee decided not to immediately merge these extensions into the actual Verilog 1364 standard. Instead, the IEEE assigned a new standards number to these extensions. In 2005, the IEEE released the 1364-2005 Verilog standard and, at the same time, the 1800-2005 SystemVerilog extensions to Verilog standard.

Figure 1-2 shows the major features that SystemVerilog added to Verilog-2001. The figure also shows that 4 features were incorporated into the Verilog 1364-2005 document, instead of the SystemVerilog 1800-2005 standard. Figure 1-2 does not delineate between the 2005, 2009, 2012 and 2017 versions of SystemVerilog. Most of the new capabilities that SystemVerilog added to traditional Verilog were made in the SystemVerilog-2005 version. Only a small number of additional features were added in the 2009 and 2012 versions, and nothing new as added in the 2017 version.

Figure 1-2: Verilog-2005 with SystemVerilog language extensions

SystemVerilog-2005/2009/2012/2017				
verification	assertions test program blocks clocking domains process control	mailboxes semaphores constrained random values direct C function calls	classes inheritance strings references	dynamic arrays associative arrays queues checkers
design	interfaces nested hierarchy unrestricted ports automatic port connect enhanced literals time values and units specialized procedures	packed arrays array assignments unique/priority case/if void functions function input defaults function array args parameterized types	break continue return do-while function inside aliasing const	2-state types shortreal type globals let macros enum typedef structures unions 2-state types packages \$unit ++ -- += -= *= /= >>= <<= >>>= <<<= &= = ^= %= ==? !=? inside streaming casting
Verilog-2005				
uwire	'begin_keywords	'pragma	\$clog2	
Verilog-2001				
ANSI C style ports generate localparam constant functions	standard file I/O \$value\$plusargs 'ifndef 'elsif 'line @*	(* attributes *) configurations memory part selects variable part select	multi dimensional arrays signed types automatic ** (power operator)	
Verilog-1995 (created in 1984)				
modules parameters function/tasks always @ assign	\$finish \$fopen \$fclose \$display \$write \$monitor 'define 'ifdef 'else 'include 'timescale	initial disable events wait # @ fork-join	wire reg integer real time packed arrays 2D memory	begin-end while for forever if-else repeat + = * / % >> << repeat

1.1.5 SystemVerilog replaces Verilog

Immediately after releasing these two separate standards, the IEEE began work on actually merging the two standards together, Stuart Sutherland (the author of this book) did the editing work to merge these two large documents. In addition to merging the two standards, the IEEE defined a number of additional SystemVerilog features. The merged Verilog and SystemVerilog standards was released as the IEEE

1800-2009 SystemVerilog standard. At that time, the IEEE terminated the old Verilog-1364 standard. The name “Verilog” officially became “SystemVerilog”.

The complexity of hardware designs and verifying those designs continues to evolve, and the IEEE continues to evolve the SystemVerilog standard to keep pace. In 2012, the IEEE released an 1800-2012 SystemVerilog standard. At the time this book was written, the IEEE was working on an 1800-2017 version of SystemVerilog. The SystemVerilog-2017 version primarily makes clarifications to the SystemVerilog standard, and does not add any new language features to the 2012 standard.

This book is based on the 2012/2017 versions of SystemVerilog.

The IEEE’s decision in 2005 to release two separate standards — one containing the traditional Verilog language (1364-2005), and one containing only extensions to Verilog and called SystemVerilog (1800-2005) — has been a source of confusion among engineers. One common misconception that seems to persist is that Verilog is a hardware modeling language and SystemVerilog is a verification language. *This misconception is not true!* The original Verilog language was always an integrated modeling and verification language. SystemVerilog extended, in substantial ways, both the modeling aspects and the verification aspects of the original Verilog HDL. SystemVerilog is both a digital modeling language and a digital verification language.

Simon Davidmann, one of the early pioneers in digital simulation, has written a more detailed history on the origins of Verilog and SystemVerilog, which can be found in the appendix of the book “*SystemVerilog for Design, Second Edition*”¹.

This books focus. The focus of this book is on the design aspects of SystemVerilog. The author recommends “*SystemVerilog for Verification, Third Edition*”² as a companion to this book to learn about the verification aspects of the language.

1.2 RTL and gate-level modeling

This section defines the terminology commonly used to describe the levels of detail in which hardware functionality can be modeled using SystemVerilog.

1.2.1 Abstraction

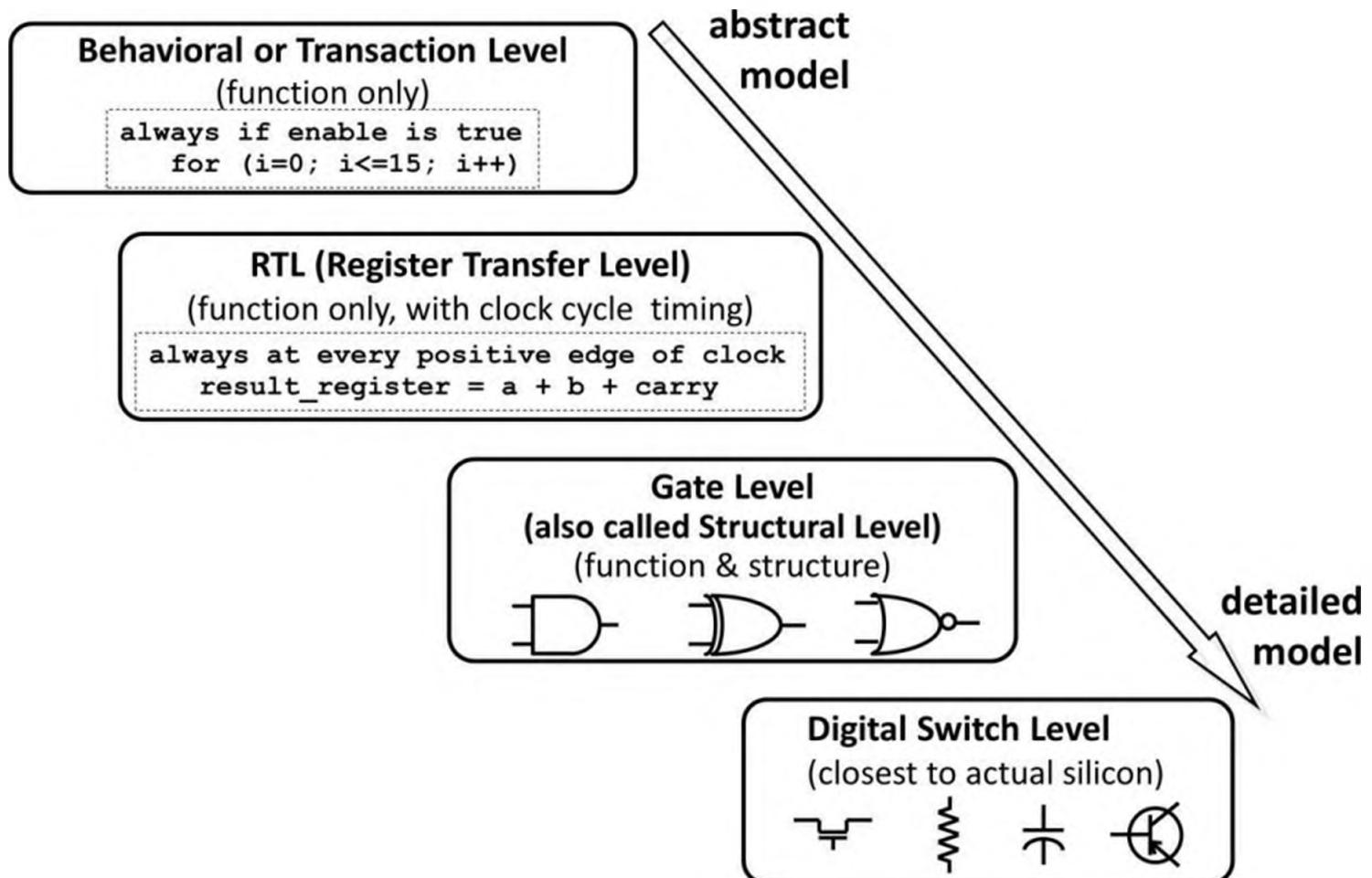
SystemVerilog is capable of modeling digital logic at many different levels of detail, referred to as “*abstraction levels*”. Abstract means lack of detail. The more abstract a digital model is, the less detail that model contains about the hardware that it represents.

1. Stuart Sutherland, Simon Davidmann and Peter Flake, “*SystemVerilog for Design, Second Edition*”, New York, NY: Springer 2016, 978-0-3873-3399-1.

2. Chris Spear and Greg Tumbush, “*SystemVerilog for Verification, Third Edition*”, New York, NY: Springer 2012, 978-1-4614-0715-7.

Figure 1-3 shows the main levels of modeling abstraction available in SystemVerilog.

Figure 1-3: SystemVerilog modeling abstraction levels



1.2.2 Gate-level models

SystemVerilog supports modeling digital logic using gate-level primitives. Digital logic gates are a detailed model that closely approximates silicon implementation.

SystemVerilog provides several built-in gate-level primitives, and allows engineers to define additional primitives, which are referred to User Defined Primitives (UDPs). The built-in primitives in SystemVerilog are listed in Table 1-1:

Table 1-1: SystemVerilog gate-level primitives

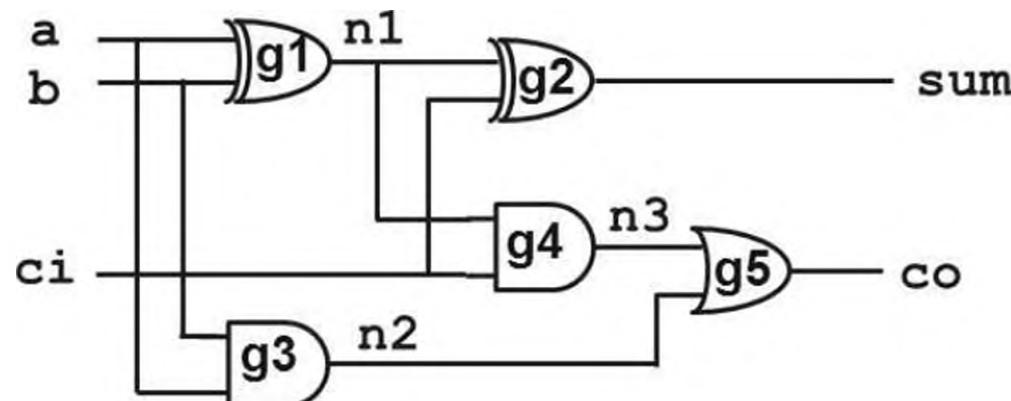
Primitive	Description
and	AND gate with 2 or more inputs and 1 output
nand	NAND gate with 2 or more inputs and 1 output
or	OR gate with 2 or more inputs and 1 output
nor	NOR gate with 2 or more inputs and 1 output
xor	Exclusive-OR gate with 2 or more inputs and 1 output
xnor	Exclusive-NOR gate with 2 or more inputs and 1 output
buf	Buffer gate with 1 input and 1 or more outputs
not	Inverter gate with 1 input and 1 or more outputs
bufif0	Tri-state buffer gate with 1 input, 1 output, and 1 active-low enable

Table 1-1: SystemVerilog gate-level primitives (continued)

Primitive	Description
bufif1	Tri-state buffer gate with 1 input, 1 output, and 1 active-high enable
notif0	Tri-state inverter gate with 1 input, 1 output, and 1 active-low enable
notif1	Tri-state inverter gate with 1 input, 1 output, and 1 active-high enable

SystemVerilog also provides a means for ASIC and FPGA library developers to add to the built-in set of primitives by defining User-Defined Primitives (UDPs). UDPs are defined in a table format, where each line in the table lists a set of input values and the resulting output values. Both combinational logic and sequential logic (such as flip-flops) primitives can be defined.

Figure 1-4 shows a gate-level circuit of a 1-bit adder with carry. Example 1-1 shows the SystemVerilog code that models this circuit using primitives.

Figure 1-4: 1-bit adder with carry, represented with logic gates**Example 1-1:** SystemVerilog gate-level model of 1-bit adder with carry

```

module gate_adder
(input wire a, b, ci,
 output wire sum, co
);
    timeunit 1ns; timeprecision 1ns;

    wire n1, n2, n3;

    xor           g1 (n1, a, b);
    xor #1.3     g2 (sum, n1, ci);
    and           g3 (n2, a, b);
    and           g4 (n3, n1, ci);
    or  #(1.5,1.8) g5 (co, n2, n3);

endmodule: gate_adder

```

The syntax of gate-level primitives is straightforward:

<gate_type> <delay> <instance_name> (<outputs>, <inputs>);

Many of the gate-level primitives can have a variable number of inputs. For example, an **and** primitive can represent a 2-input, 3-input, or 4-input AND gate, as follows:

```
and i1 (o1, a, b);      // 2-input AND gate
and i2 (o2, a, b, c);    // 3-input AND gate
and i3 (o3, a, b, c, d); // 4-input AND gate
```

The instance name for primitives is optional, but is good code documentation. It makes it easier to maintain code and to relate SystemVerilog source code to schematics or other representations of the design. The instance name is user-defined, and can be any legal SystemVerilog name.

Gate level primitives can be modeled with propagation delays. If no delay is specified, then a change on an input to the gate will be immediately reflected on the output of the gate. The delay is an expression, that can be a simple value, as in instance g2 in Example 1-1, or a more complex expression, as in instance g5. Gate g2 in the example above has a propagation delay of 1.3 nanoseconds, meaning that when there is a transition on one of the gate inputs, it will be 1.3 nanoseconds before the output of the gate, sum, will change. Gate g5 breaks the propagation delay into different delays for rising and falling transitions on the output. If the value of co is transitioning from a 0 to 1, the change will be delayed by 1.5 nanoseconds. If co is transitioning from a 1 to 0, the change will be delayed by 1.8 nanoseconds.

Gate-level modeling can represent the propagation delays of actual silicon with a high degree of accuracy. The functionality of the logic gates reflects the functionality of the transistor combinations that would be used in silicon, and the gate delays can reflect the propagation delays through those transistors. This accuracy is used by ASIC and FPGA suppliers to model the detailed behavior of specific devices. Section 1.6 (page 31) in this chapter explores this usage of gate-level models in ASIC and FPGA design flows.

Gate-level models are usually generated by software tools or engineers specializing in library development. Design engineers designing at the RTL level seldom, if ever, model with gate-level primitives. Rather, RTL designers use netlists of gate-level models, where the netlists were generated by synthesizing the RTL models. The gate-level models are provided by the vendor of the target ASIC or FPGA device. There is much more to gate-level modeling than has been shown in this section, but this book does not go into any further detail on this topic.

Switch-level modeling. SystemVerilog can also model digital circuits at the transistor level, using switch primitives (such as **pmos**, **nmos** and **cmos**), resistive switch primitives (such as **rpmos**, **rnmos** and **rcmos**), and capacitive nets. This level of modeling can closely represent actual silicon implementation. However, since these constructs only model digital behavior, they are seldom used. Transistors, resistors and capacitors are analog devices. Digital simulation does not accurately reflect transistor behavior. Switch-level modeling is typically not used in ASIC and FPGA design flows with SystemVerilog, and is not discussed in detail in this book.

1.2.3 RTL models

A more abstract level of modeling — and the one that is the focus of this book — is *Register Transfer Level*, or *RTL*. This level of modeling represents digital functionality using programming statements and operators. RTL models are functional models that do not contain the details on how that functionality will be implemented in silicon. Because of this abstraction, complex digital functionality can be modeled more quickly and concisely than at the detailed gate-level. RTL models also simulate substantially faster than gate-level and switch-level models, making it possible to verify much larger and more complex designs.

SystemVerilog provides two primary constructs for RTL modeling: *continuous assignments* and *always procedural blocks*.

Continuous assignments begin with an **assign** keyword, and can represent simple combinational logic. The previous Example 1-1 illustrated a gate-level model of a 1-bit adder. Example 1-2 shows how this same 1-bit adder functionality can be modeled at a more abstract level by using a continuous assignment:

Example 1-2: SystemVerilog RTL model of 1-bit adder with carry

```
module rtl_adder
  input  logic a, b, ci,
  output logic sum, co
);

assign {co,sum} = a + b + ci;

endmodule: rtl_adder
```

One advantage of RTL modeling is that the code is more self-documenting. It can be difficult to look at the gate-level model in Example 1-1 and recognize what the model represents, especially if there were no comments and meaningful names. On the other hand, it is much easier to look at the code in the RTL model in Example 1-2 and recognize that the functionality is an adder.

Another powerful advantage of RTL modeling is the ability to work with vectors and bundles of data. A *vector* is a signal that is more than one bit wide. The detailed switch-level and gate-level of modeling operate on one 1-bit wide signals, which are referred to as *scalar* signals in SystemVerilog. To model a 32-bit adder would require modeling switches or gates that operated on each individual bit, the same as in actual silicon. The continuous assignment statement in Example 1-2 above can model an adder of any size, simply by changing the declarations of the signals.

More complex functionality can be modeled using *procedural blocks*. A procedural block encapsulates one or more lines of programming statements, along with information about when the statements should be executed. There are four types of *always* procedures that are used at the RTL level: **always**, **always_comb**, **always_ff** and

always_latch. Chapter 6, section 6.1 (page 211) explores the use of *always* procedural blocks in greater detail.

The following example concisely represents a 32-bit adder/subtractor with registered outputs:

Example 1-3: SystemVerilog RTL model of 32-bit adder/subtractor

```
module rtl_adder_subtractor
  (input logic      clk,    // 1-bit scalar input
   input logic      mode,   // 1-bit scalar input
   input logic [31:0] a, b, // 32-bit vector inputs
   output logic [31:0] sum); // 32-bit vector output
);

  always_ff @(posedge clk) begin
    if (mode == 0) sum <= a + b;
    else           sum <= a - b;
  end

endmodule: rtl_adder_subtractor
```

In a typical simulation and synthesis design flow, engineers will spend most of their time modeling at the RTL level and verifying RTL functionality. The focus of this book is on writing RTL models that will simulate and synthesize optimally.

1.2.4 Behavioral and transaction-level models

SystemVerilog procedural blocks can be used to model at a higher level of abstraction than RTL. Models at this abstraction are often referred to as *behavioral models* (also called *bus-functional* or *algorithmic* models). Behavioral models can appear very similar to an RTL model because both RTL and behavioral models use *always* procedural blocks. Behavioral models differ from RTL in either, or both, of two ways.

- An RTL procedural block executes its programming statements in a single clock cycle, or in zero cycles if combinational logic. A behavioral procedural block can take an arbitrary number of clock cycles to execute its statements.
- An RTL model must adhere to strict language restrictions in order to be synthesizable by RTL synthesis compilers. A behavioral model can utilize the full SystemVerilog language.

A high level of abstraction is *transaction level* modeling. Transaction models are commonly used in verification code, and are typically modeled using SystemVerilog's object-oriented programming constructs.

The behavioral and transaction levels of abstraction are not synthesizable by RTL synthesis compilers, and are not discussed in this book.

1.3 Defining an RTL synthesis subset of SystemVerilog

SystemVerilog is both a hardware design language and a hardware verification language. The official IEEE SystemVerilog standard does not distinguish between these two objectives, and does not specify a synthesis subset of the full SystemVerilog language. Instead, the IEEE has left it up to companies who provide RTL synthesis compilers to define which SystemVerilog language constructs are supported by specific products.

The lack of a SystemVerilog synthesis standard has resulted in each synthesis compiler supporting a different subset of the SystemVerilog standard. This means design engineers need to be careful when writing models intended for synthesis. It is essential to refer to the documentation for the synthesis compiler to be used, and adhere to the language subset of that compiler. Models written for one synthesis compiler might require modification to work with a different synthesis compiler.

This book defines a subset of SystemVerilog that will work with most SystemVerilog RTL synthesis compilers commercially available at the time this book was written. Design engineers can be confident that models will synthesize on any major SystemVerilog synthesis compiler by adhering to the language restrictions and coding guidelines defined in this book.

1.4 Modeling for ASICs and FPGAs

A full definition of ASIC and FPGA technologies is beyond the scope of this book, which is about proper digital logic modeling styles with the SystemVerilog language.

The purpose of this section is to look at how SystemVerilog modeling styles can be affected by ASIC and FPGA technology. Details on ASIC and FPGA implementation and the appropriate applications for these technologies are left to other engineering books and discussions. In order to meet this objective on RTL modeling best practices, however, it is important to understand the basic concepts of ASICs and FPGAs.

1.4.1 Standard cell ASICs

ASIC is an acronym for *Application Specific Integrated Circuit*. Unlike general purpose ICs that can perform many types of functions, such as microprocessors, ASICs are designed to perform a specific task (hence the name, “application specific”). Controllers, audio format conversion, and video processing are examples of application specific tasks suitable for ASICs. An ASIC can also include one or more embedded processors in order to perform general operations, as well as its specific tasks. An ASIC with embedded processors is often referred to as a *System on Chip* (SoC).

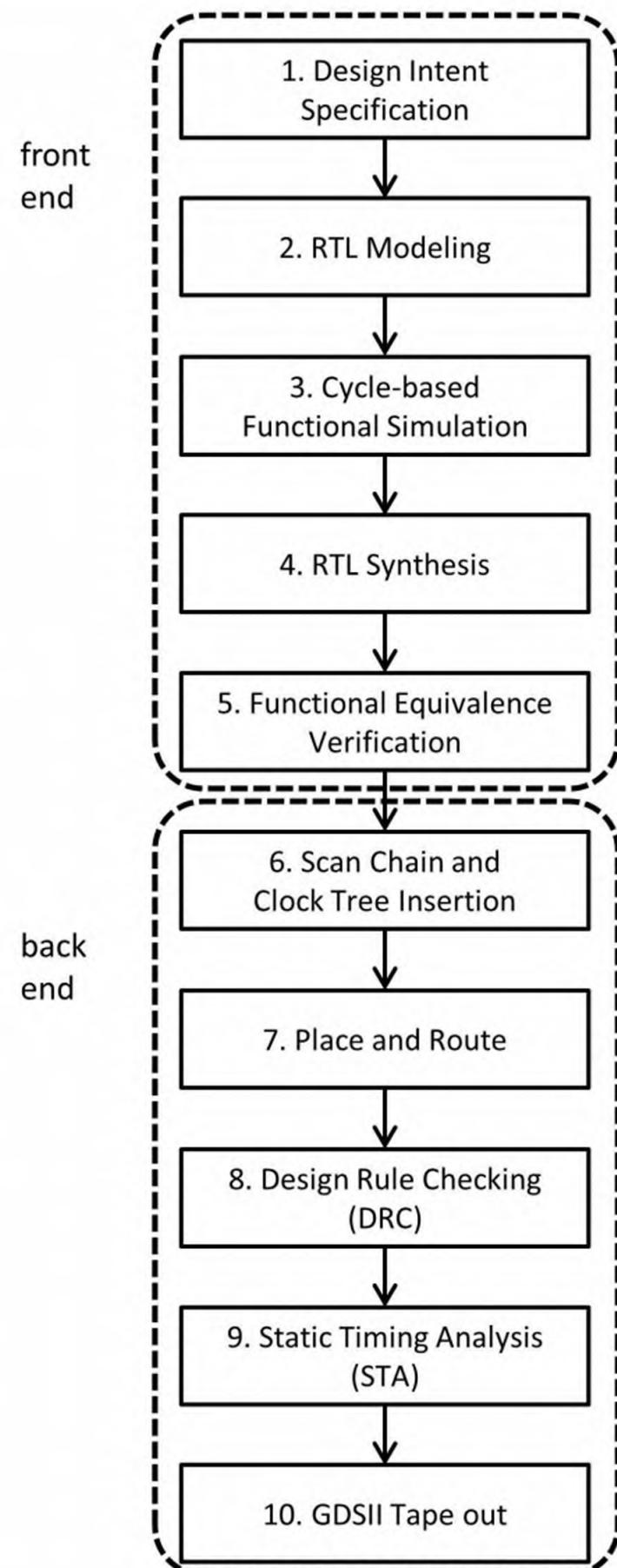
Companies that provide ASICs are referred to as *ASIC vendors*. Some of these vendors provide both the ASIC technology and do the actual fabrication and production

of the ICs. Other ASIC vendors provide the technology for the ASIC, but leave the fabrication and production to some other source.

Most ASIC technologies use *standard cells*, which are pre-designed blocks of logic consisting of one to several logic gates. An ASIC cell library might have a few hundred standard cells, such AND, NAND, OR, NOR, Exclusive-OR, Exclusive-NOR, 2-to-1 MUX, D-type flip flop, latch, etc. Each cell will have well defined electrical characteristics, such as propagation delays, setup and hold times, and capacitance.

Designing an ASIC involves selecting appropriate cells from the library and connecting them together to perform the desired functionality. Software tools are used throughout this process. The typical flow for ASIC design is shown in Figure 1-5.

Figure 1-5: Typical RTL-based ASIC design flow



The steps illustrated in Figure 1-5 are:

1. The first step is, of course, the specification of what the design is intended to do.
2. The desired functionality is modeled at the abstract RTL level of modeling. At this stage the focus is on functionality, not on physical implementation.
3. Simulation is used to verify the functionality. Section 1.5 (page 17) discusses simulation in more detail.
4. Synthesis is used to map the RTL functionality to the appropriate standard cells of the target ASIC type. The output from synthesis is referred to as a *gate-level netlist*. This synthesis process is described in more detail in section 1.6 (page 31).
5. Simulation and/or Logic Equivalence Checkers (a form of formal verification) are used to verify that the gate-level implementation is functionally equivalent to the RTL functionality.
6. Clock tree synthesis is used to distribute clock drivers evenly across the design. Often, a scan chain insertion tool is used to add testability to the design.
7. Place and route software calculates how to layout the cells in the actual silicon and route the interconnecting traces. The output of the place and route software is a *Graphic Data System* file (GDSII, pronounced gee-dee-ess-two) file. GDSII is a binary format that contains information about the geometric shapes (polygons) and other data needed to actually construct the IC in silicon.
8. Design Rule Checking (DRC) is performed to ensure that all rules defined by the foundry where the ASIC will be fabricated have been adhered to, such as gate fan out loading.
9. Static timing analysis (STA) is performed to ensure that setup/hold times are being met, after allowing for the delay effects of the interconnecting nets and clock tree skews.
10. The final step is to ship the GDSII file and other data to the foundry that will be used to manufacture the ASIC. Passing these files on to the foundry is referred to as “*taping out*” the ASIC, because, in the early days of ASIC design, magnetic tapes were used to send these files to the foundry.

These steps in the ASIC design flow have been generalized for the purposes of this book. There are many details that have been left out, and not all companies follow this exact flow. Sometimes step 9, static timing analysis, is performed earlier in the design flow, and might be performed multiple times in the flow.

The focus of this book is on RTL modeling for simulation and synthesis, steps 2 and 3 in Figure 1-5. This level of modeling is at the front end of the design process. Design details such as clock trees, scan chains, and timing analysis come later in the design flow, and are outside the scope of this book. At the RTL level, design engineers focus on implementing the desired functionality, and not on implementation details. It is still important, however, to understand what comes after the front end

steps of modeling, simulation, and synthesis. RTL coding styles can impact the effectiveness of the tools used later in the design flow.

There are other types of ASIC technologies that do not use standard cells, such as full-custom, gate-array, and structured ASICs. SystemVerilog can be used in a similar way to design these other types of ASICs, though the software tools involved might be different. The synthesis compilers used — and the SystemVerilog language constructs supported by those compilers — can be very different with these other technologies. This book focuses on modeling with SystemVerilog for the more general standard cell ASIC technology.

1.4.2 FPGAs

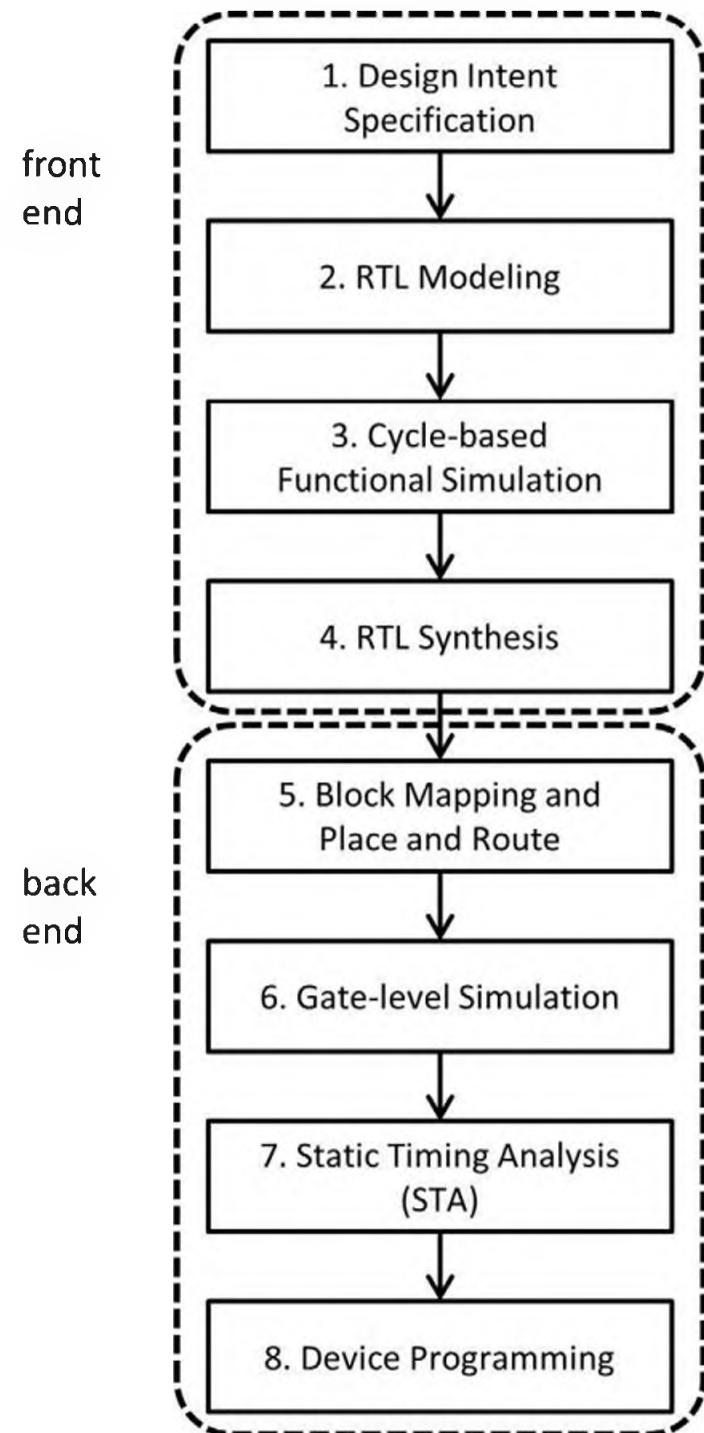
FPGA is an acronym for *Field Programmable Gate Array*. An FPGA is an integrated circuit containing a fixed number of logic blocks that can be configured after the IC is manufactured (whereas the contents and layout of an ASIC must be determined prior to manufacturing). Historically, FPGAs could not contain as much functionality as an ASIC and ran at slower clock speeds, which were important considerations when designing at the RTL level. Recent advancements in FPGA technology have significantly narrowed the difference between FPGAs and ASICs. In general, an FPGA can be used to implement the same functionality as an ASIC.

FPGAs contain an array of many small logic components referred to as Configurable Logic Blocks (CLBs). Some FPGA vendors refer to these blocks as Logic Array Blocks (LABs). A typical CLB might contain one or more Look-up Tables (LUTs), some multiplexers (MUXes), and a storage element such as a D-type flip flop. The look-up tables in most FPGAs are small RAMs that are programmed with logic operations such as AND, OR and XOR. Selecting a desired operation from the LUT allows a CLB to be used in a variety of ways, from a simple AND or XOR gate to much more complex combinational functionality. The CLBs in some FPGAs might also have other functionality, such as an adder. A MUX allows the combinational result to be directly output from the CLB (asynchronous output), or to be registered in the storage element (synchronous output).

An FPGA will have been manufactured with an array containing many hundreds or thousands of CLBs, along with configurable interconnections that can be “programmed” to a desired configuration of CLBs. An FPGA also contains I/O pads, which can be configured to connect to either one column or one row of the CLB array.

A typical design flow for a complex FPGA is shown in Figure 1-6.

Figure 1-6: Typical RTL-based FPGA design flow



The front end of the design flow for an FPGA is similar to that of an ASIC, but the back end is different. The primary difference in the back-end portion of this FPGA flow from the ASIC flow is the placement and routing of FPGAs. With ASICs, the place and route software determines how the IC will be manufactured. With FPGAs, the synthesis and place and route software determine how the pre-manufactured IC will be programmed. This book is focused on the front-end steps 2 and 3, RTL modeling and simulation, where there is very little difference between ASIC and FPGA design.

1.4.3 *RTL coding styles for ASICs and FPGAs*

Ideally, the same RTL code can be used to target either an ASIC or an FPGA. The engineering focus when working at the RTL level is on designing and verifying functionality, without having to be concerned about implementation details. It is the role of the synthesis compiler to map the RTL functionality to a specific ASIC or FPGA technology.

The truth comes close to this ideal. Most, but not all, RTL code will synthesize equally well for both ASICs and FPGAs. There are exceptions to this generality, however. Some aspects of an RTL model need to take into consideration whether the design will be implemented in an ASIC or an FPGA. These aspects include:

Resets. Most ASIC cell libraries include both synchronous and asynchronous reset flip-flops. Design engineers can write RTL models using the reset type deemed best for the design. Some FPGAs are not as flexible, and only have flip-flops with one type of reset (typically synchronous). While a synthesis compiler can map RTL models with asynchronous resets to gate-level synchronous resets, or vice versa, extra logic gates will be required. Many FPGAs also support global reset functionality and power-up flip-flop states, that ASICs do not have. Chapter 8, section 8.1.5 (page 286) discusses modeling resets in more detail.

Vector sizes. ASICs are fairly unconstrained for maximum vector widths and vector operations. Complex operations on large vectors will require a lot of logic gates, but the standard cell architecture used in most ASICs can accommodate these operations. FPGAs are more rigid in this regard. The predefined number of CLBs and their placement within an FPGA can limit the ability to implement complex operations on very large vectors, either due to the number of CLBs available, or the complexity of routing the interconnections between the CLBs. This difference between ASICs and FPGAs does mean that, even at the RTL level of abstraction, the design engineer must keep in mind the limits of the device to which the functionality will be targeted.

Most of the examples, coding styles, and guidelines presented in this book apply equally to both ASIC and FPGA design. Specific mention is made for the rare exceptions when targeting an ASIC or FPGA impacts RTL coding style.

1.5 SystemVerilog simulation

Digital simulation is a software program that applies logic value changes, called *stimulus*, to the inputs of a *model* of a digital circuit, propagates that stimulus through the model in the same way in which actual silicon would propagate those logic value changes, and provides mechanisms for observing and *verifying* the results of that stimulus.

SystemVerilog is a digital simulation language that works with zeros and ones. The language does not represent analog voltage, capacitance, and resistance. SystemVerilog provides programming constructs to model digital circuits, to model stimulus generators, and to model verification checkers.

This book focuses strictly on the first of these aspects, modeling digital circuits, and this topic will be discussed and illustrated in detail in subsequent chapters. Example 1.4 illustrates a simple digital circuit model that can be simulated. This is the same circuit shown earlier in this chapter as Example 1.3 (page 12).

Example 1-4: Design model with input and output ports (a 32-bit adder/subtractor)

```
module rtl_adder_subtractor
  (input logic clk, // 1-bit scalar input
   input logic mode, // 1-bit scalar input
   input logic [31:0] a, b, // 32-bit vector inputs
   output logic [31:0] sum // 32-bit vector output
);

always_ff @(posedge clk) begin
  if (mode == 0) sum <= a + b;
  else sum <= a - b;
end

endmodule: rtl_adder_subtractor
```

Observe in this example that the model has *input ports* and *output ports*. In order to simulate this model, stimulus must be provided that applies logic values to the input ports, and a response checker must be provided to observe the output ports. Although not the focus of this book, a brief overview of stimulus and response checking is provided here to show what is involved to simulate a SystemVerilog model.

A *testbench* is used to encapsulate the stimulus generation and response verification. There are many ways a testbench can be modeled in SystemVerilog, and the code within the testbench can range from simple programming statements to elaborate object-oriented, transaction-level programming. Example 1-5 illustrates a simple testbench for the 32-bit adder/subtractor design.

Example 1-5: Testbench for the 32-bit adder/subtractor model

```
module test
  (output logic [31:0] a, b,
   output logic mode,
   input logic [31:0] sum,
   input logic clk
);
  timeunit 1ns; timeprecision 1ns;

  // generate stimulus
  initial begin
    repeat (10) begin
      @(negedge clk) ;
      void' (std::randomize(a) with {a >= 10; a <= 20;});
      void' (std::randomize(b) with {b <= 10;});
      void' (std::randomize(mode));
      @(negedge clk) check_results;
    end
    @(negedge clk) $finish;
  end
```

```

// verify results
task check_results;
    $display("At %0d: \t a=%0d b=%0d mode=%b sum=%0d",
             $time, a, b, mode, sum);
    case (mode)
        1'b0: if (sum !== a + b)
            $error("expected sum = %0d", a + b);
        1'b1: if (sum !== a - b)
            $error("expected sum = %0d", a - b);
    endcase
endtask

endmodule: test

```

Initial and always procedures. The main block of code in Example 1-5 is an *initial procedure*, which is a type of *procedural blocks*. Procedural blocks contain programming statements and timing information to instruct simulators what to do, and when to do it. SystemVerilog has two primary types of procedural blocks, *initial procedures* and *always procedures*.

Initial procedures are defined with the keyword **initial**. An initial procedure, despite its name, is not used for initializing designs. Rather, an initial procedure executes its programming statements one time. When the last statement is reached, the initial procedure is not executed again for a given run of a simulation. Initial procedures are not synthesizable, and are not used for RTL modeling. This book focuses on writing RTL models for simulation and synthesis, and therefore does not discuss initial procedures in any more depth.

Always procedures are defined with the keywords, **always**, **always_comb**, **always_ff** and **always_latch**. An always procedure is an infinite loop. When the procedure has completed execution of the last statement in the procedure, the procedure automatically returns to the beginning, and starts the procedure again. For RTL modeling, an always procedure must begin with a sensitivity list, such as the **@(posedge clk)** definition shown in Example 1-4 (page 18). The various forms of always procedures are discussed in more detail in Chapters 6 through 9.

A procedural block can contain a single statement, or a group of statements. Multiple statements in a procedural block are grouped together between the keywords **begin** and **end** (verification code can also group statements between the keywords **fork** and **join**, **join_any** or **join_none**). Statements between **begin** and **end** are executed in the order in which they are listed, i.e.: beginning with the first statement and ending with the last statement.

The initial procedure in Example 1-5 contains a **repeat** loop. This loop is defined to execute 10 times. Each pass of the loop:

1. Delays to the negative edge of the **clk** signal.
2. Generates random values for the **a**, **b** and **mode** inputs of the design.

3. Delays until the next negative edge of `clk`, and then calls a `check_results` task (a subroutine) to verify that the output of the design matches a calculated expected result.

The design operates on the positive edge of its clock input. The testbench uses the opposite edge of this same clock, in order to avoid driving the inputs and reading the outputs of the design on the clock edge that is used by the design. If the testbench drove values on the positive edge of the clock, there would be zero setup time for those inputs to become stable before the design used the inputs. Similarly, if the testbench verified the design results on the positive edge of clock, there would be zero time for those design outputs to become stable.

Modifying and reading a value at the same instant of time is referred to as a *simulation race condition*. Using the opposite edge of the design clock to drive stimulus is a simple way for a testbench to avoid simulation race conditions with the design, such as meeting design setup and hold time requirements. SystemVerilog provides much more effective ways for a testbench to avoid race conditions with the design being tested, which are beyond the scope of this book. The author recommends the book ***SystemVerilog for Verification*** by Chris Spear and Greg Tumbush³ for more details on SystemVerilog's versatile and powerful verification constructs, and proper verification coding styles.

The testbench is modeled as a module with input and output ports, similar to the design being verified. The last step is to connect the testbench ports to the design ports, and generate the clock. This is done in a top-level module. Example 1-6 shows the code for this.

Example 1-6: Top-level module connecting the testbench to the design

```
module top;
  timeunit 1ns; timeprecision 1ns;

  logic [31:0] a, b;
  logic        mode;
  logic [31:0] sum;
  logic        clk;

  test                  test (.*);
  rtl_adder_subtractor dut  (.*);

  initial begin
    clk <= 0;
    forever #5 clk = ~clk;
  end
endmodule: top
```

3. “*SystemVerilog for Verification, Third Edition*”, Chris Spear and Greg Tumbush. Copyright 2012, Springer, New York, NY. ISBN: 978-1-4614-0715-7.

1.5.1 SystemVerilog simulators

There are several commercial simulators available that provide excellent support for the SystemVerilog language. This book is simulator neutral, and does not provide details on any specific simulator product. Nonetheless, readers are encouraged to write and simulate many of the examples in this book, as well as examples conceived on their own. Refer to the documentation provided with a specific simulator product for information on invoking and running that simulator.

All SystemVerilog simulators have a number of things in common, and which are vital for understanding how to write SystemVerilog RTL models that will simulate correctly. These features include: compilation, elaboration, simulation time, and simulation event scheduling. The following pages discuss these aspects of simulation.

1.5.2 Compilation and elaboration

SystemVerilog source code needs to be compiled and elaborated in order to be simulated. *Compilation* involves checking SystemVerilog source code to ensure it is syntactically and semantically correct, according to the rules defined in the IEEE SystemVerilog standard. *Elaboration* binds together the modules and components that make up the design and testbench. Elaboration also resolves configurable code, such as the final values of constants, vector sizes, and simulation time scaling.

The IEEE SystemVerilog standard does not define the exact compilation and elaboration process. The standard allows each simulator vendor to define this process and the division between compilation and elaboration in the manner the vendor deems to be best for that product. Some simulators combine the compilation and elaboration process as a single step, while other simulators divide these processes into separate steps. Some simulators might catch certain types of errors in the source code during the compilation phase, while other simulators catch those errors during the elaboration phase. These differences do not affect the RTL coding styles and guidelines discussed in this book, but it is helpful to be aware of how the simulator being used handles the compilation and elaboration of the RTL source code. Refer to the documentation for a specific simulator on how that product handles compilation and elaboration.

1.5.2.1 Source code order

The SystemVerilog language, like most, if not all, programming languages, does have certain dependencies in source code order. In particular, user-defined type declarations and declaration packages must be compiled before those definitions are referenced. It is common for user-defined type declarations and packages to be in separate files from the RTL code that uses the declarations. This means designers must be careful that these files are compiled in the correct order, so that declarations are compiled before being referenced.

Not all declarations are order dependent. For example, SystemVerilog allows module names to be referenced before the module is compiled. Within a module, tasks and functions can be called before being defined, as long as the definition is within the module.

1.5.2.2 Global declarations and the \$unit declaration space

SystemVerilog allows some types of definitions to be made in a global declaration space called **\$unit**. Declarations in **\$unit** can be shared by multiple files. Global declarations are compilation order dependent, and must be compiled before being referenced. The global **\$unit** is not a self-contained modeling space — any file can add definitions to **\$unit**. This can lead to haphazard global definitions that make it difficult to ensure that definitions are compiled before being referenced.

SystemVerilog compiler directives, such as '**define**' text macros and '**timescale**' time scaling, also fall in the **\$unit** space, global, and must be compiled before the code affected by the directive.

Best Practice Guideline 1-1

Use packages for shared declarations instead of the **\$unit** declaration space.

Packages are discussed in Chapter 4, section 4.2 (page 102).

1.5.2.3 Single-file and multi-file compilation

The IEEE SystemVerilog standard defines the rules for two compilation/elaboration paradigms when multiple files are involved: *multi-file compilation* and *single-file compilation*.

The multi-file compilation paradigm allows multiple source code files to be compiled together. Global declarations and compiler directives in one file will be visible to the source code in other files that are compiled after the declarations and directives.

The single-file compilation paradigm allows each file to be compiled independently. Any global declarations or compiler directives in one file are only visible within that file. Other files will not see those declarations or directives, regardless of the order in which the files are compiled.

All simulators and synthesis compilers support the multi-file paradigm, but not all tools support single-file compilation. However, tools that support both paradigms do not necessarily use the same paradigm by default. Some tools use single-file compilation by default, and require a tool-specific invocation option for multi-file compilations. Other tools use multi-file compilation by default, and require an invocation option for single-file compilation or incremental re-compilation.

1.5.2.4 Simulator product limitations

In a perfect world, all SystemVerilog simulators would support the entire SystemVerilog language, and in exactly the same way. No simulator is perfect, however (despite what the EDA vendor's salesman might claim). While all major commercial SystemVerilog simulators support most of the SystemVerilog language, few, if any, SystemVerilog simulators have implemented 100% of the latest SystemVerilog standard. It seems inevitable that there will always be a SystemVerilog feature that works in some simulators but not in another simulator, or a language feature that gets different simulation results in different simulators.

One reason a simulator might not fully support the entire SystemVerilog language is that SystemVerilog is a large and complex language standard. It is tedious, time consuming, and an intense engineering effort to implement every feature in the SystemVerilog standard. A simulator vendor might intend to implement all of the SystemVerilog standard, but it can require many man months to do so.

A second reason for variances in simulator support for SystemVerilog is ambiguities in how the SystemVerilog standard should be interpreted. Some of these ambiguities are due to the complexity of the language. Despite the best efforts of the IEEE 1800 standards committee, sometimes the wording used in the standard can be interpreted in more than one way. Perhaps surprising to some, is that the IEEE SystemVerilog also has intentional ambiguities. There are times when the IEEE standards committee purposely allows simulators to differ, in order to allow each simulator vendor to optimize simulation performance and implementation in the manner deemed best by that vendor.

A third, and unfortunate, reason not all simulators support the full SystemVerilog language is that some simulator vendors see no reason to do so. This might be simply that the simulator vendor, like all engineering companies, has limited engineering resources and many engineering tasks to complete. A compromise is made on where the engineering efforts are focused, and support for certain SystemVerilog language features ends up a low priority. Another reason a simulator vendor might chose not to support the full SystemVerilog language is that the vendor has targeted its product to a market where, at least according to that simulator vendor, only a subset of the SystemVerilog language is needed.

1.5.3 Simulation time and event scheduling

The SystemVerilog standard defines rules for how simulated time is represented within simulation. These rules are discussed in section 1.5.3.1, below.

The standard also defines rules for the order in which programming statements are evaluated and logic value changes are propagated during simulation. The evaluation of a programming statement and the change of logic value are referred to as simulation events, and are discussed beginning in section 1.5.3.3 (page 26).

1.5.3.1 Time units and time precision

Simulation requires representing the passing of time. The SystemVerilog standard allows time to be specified in units ranging anywhere from 1 femtosecond to 100 seconds.

SystemVerilog also allows the time precision to be specified. Time precision controls how many decimal places of accuracy simulation should use. Any delays in a module with a greater number of decimal places will be rounded off to the precision.

Precision is specified relative to the time units. If the time unit is 1 nanosecond, for example, a precision of 1 picosecond would allow for 3 decimal places of accuracy (a nanosecond is 10^{-9} and a picosecond is 10^{-12} , yielding a difference of 10^{-3} , or 3 decimal places).

The units of time and time precision used in a SystemVerilog module can be specified at the module level, using **timeunit** and **timeprecision** statements. For example:

```
module adder
  (input logic a, b, ci,
   output logic sum, co
  );
  timeunit 1ns;           // delays are in nanoseconds
  timeprecision 1ps;      // 2 decimal places of accuracy
  ...
endmodule: adder
```

The time precision can also be specified in conjunction with the time unit, using a slash (/) to separate the units and precision.

```
timeunit 1ns/1ps;
```

Traditional Verilog's 'timescale compiler directive. It is also possible to specify time units and time precision on a semi-global basis, using a `**timescale** compiler directive. This directive must be specified outside of the module boundary, in the \$unit declaration space (see section 1.5.2, page 21).

```
`timescale 1ns/1ps
module adder
  (input logic a, b, ci,
   output logic sum, co
  );
  ...
endmodule: adder
```

Compiler directives, such as '**timescale**', are not fully global. They only affect source code that is compiled after the directive is read in, and in the same invocation of the compiler. Any source code that was compiled before the directive was encountered is not affected by the directive. Likewise, any code that is compiled as a separate single-file compilation is not affected by the directive. The `'**timescale**' directive can affect multiple files when more than one file is compiled at the same time. The

order in which multiple files are compiled becomes critical when only some of the files have a `timescale directive. Compiling the files in a different order can lead to different simulation results because of the semi-global behavior of `timescale.

Best Practice Guideline 1-2

Use the SystemVerilog timeunit keyword to specify simulation time units and precision, instead of the old `timescale compiler directive.

The timeunit and timeprecision keywords are local to the module in which they are used, and do not have the compilation order side effects of compiler directives.

Simulation-level time units. The IEEE SystemVerilog standard does not specify a default time unit or precision that simulators should use if no timeunit and timeprecision statement has been specified, and no `timescale compiler directive is in effect. Different simulators have different behaviors in this circumstance, ranging from making it a compilation error, to having an implied default unit and precision.

All simulators provide an invocation option to set the default time unit and time precision to be used in any module that does not have this information declared within the module or by a `timescale compiler directive. An example invocation option that works with most simulators is:

```
-timescale=1ns/1nps
```

Refer to the documentation of the specific simulator for the command-line invocation option that sets the simulation time units and precision.

NOTE

Most code examples in this book do not include timeunit and timeprecision statements. These statements were omitted in order to focus on the RTL code in each example.

The example files for this book that can be downloaded (see page xxx in the Preface) have a timeunit and timeprecision statement in every module, interface or package, even when there are no delays within the model. This ensures that the models will compile with the same time units on all simulators.

1.5.3.2 Propagation delays

Propagation delays can be represented at both the detailed gate-level of modeling and at the more abstract RTL level of modeling. Most RTL models are written with zero propagation delays. Timing at the RTL level is typically on clock cycle boundar-

ies, with no propagation delay within a clock cycle. Propagation delays are frequently used in verification testbenches.

The `#` token is used to represent a delay. The following code snippet shows the use of a delay to model a clock oscillator in a testbench.

```
always #5 clk = ~clk;
```

A specific time unit can be specified with a delay. This is particularly useful when a delay requires a different time unit than the units of the module.

```
always #5ns clk = ~clk;
```

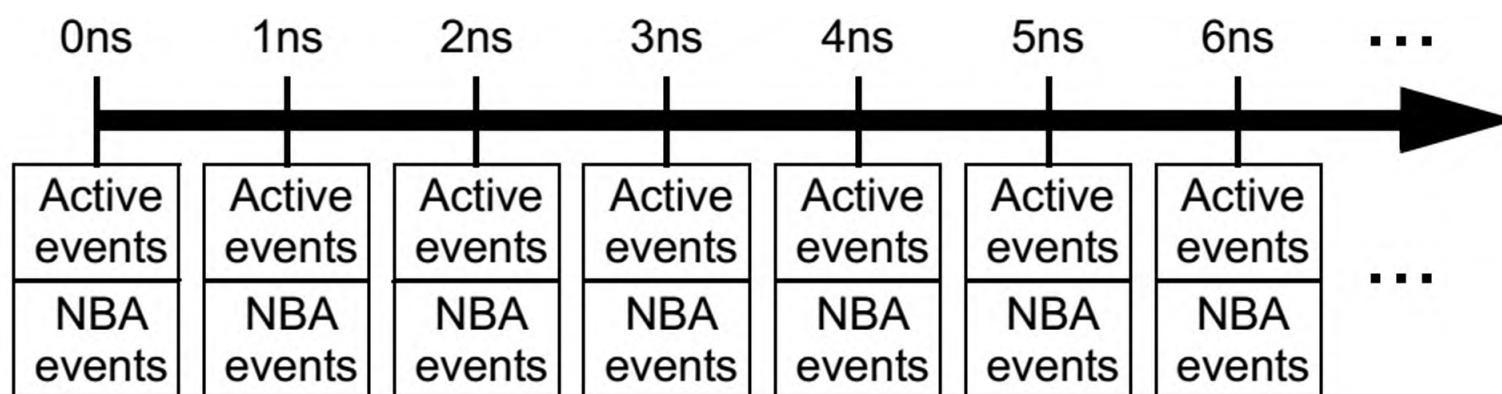
1.5.3.3 Simulating time and simulation event scheduling

A logic value change is referred to as a simulation event. The SystemVerilog standard defines the general rules regarding how all SystemVerilog simulators should propagate logic value changes within simulation. The algorithm is complex, and is described in detail in the IEEE 1800 SystemVerilog standard. A brief overview is provided in this section.

Simulators maintain a *simulation time line* that represents moments in time in which logic values are scheduled to occur. Each moment in time is referred to as a *simulation time slot*. Conceptually, there is a time slot in this time line for every increment of the finest precision of time represented in the SystemVerilog source code that has been compiled and elaborated. Thus, if the smallest precision used anywhere in the source code is 1 picosecond, the time line would have time slots at 0ps, 1ps, 2ps, 3ps, and so on, until the end of the simulation.

A common practice is for RTL models to use a time unit of 1 nanoseconds and a precision of 1 nanosecond (no decimal places of precision). Figure 1-7 illustrates what the time line might look like for a typical RTL model, assuming the smallest time precision of all source code elaborated into the simulation is 1 nanosecond.

Figure 1-7: Simulation time line and time slots



In practice, SystemVerilog simulators will optimize the simulation time line, and only create time slots for the times in which actual simulation activity will occur. This optimization means that the time precision used in the source code has little or no real impact on simulation run-time performance.

1.5.3.4 Event regions and event scheduling

Each simulation time slot is divided into several *event regions*. An event region is used to schedule an activity that must be processed by the simulator. For example, if a clock oscillator changes values every 5 nanoseconds, beginning at time 0, then the simulator would schedule a simulation event for the clock signal at 0ns, 5ns, 10ns, and so forth in the simulation time line. Likewise, if a programming statement will be executed at simulation time 7 nanoseconds, the simulator will schedule an event to evaluate that statement at the 7ns time slot in the simulation time line.

Simulators can schedule events to be processed in the current time slot, or in any future time slot. Simulators cannot schedule events in past time slots. SystemVerilog simulators only move forward in time, and never backwards, although many simulators do allow resetting simulation back to time 0 and starting simulation over again.

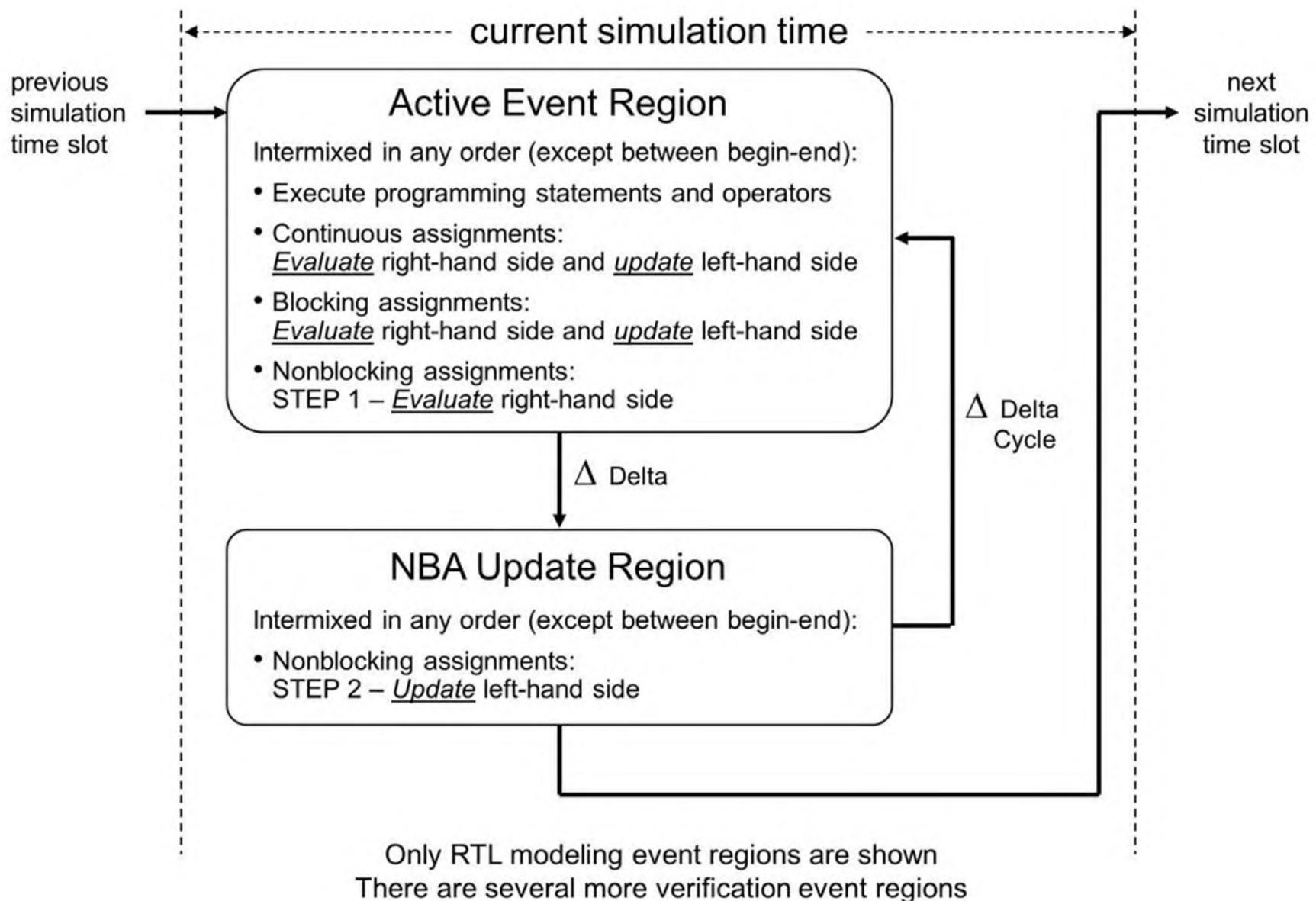
1.5.3.5 Active and NBA Update events, blocking and nonblocking assignments

SystemVerilog divides a simulation time slot into several *event regions*, which are processed in a controlled order. This provides design and verification engineers some control over the order in which events are processed. Most of the event regions are for verification purposes, and are not discussed in this book. RTL and gate-level models primarily use two of these event regions: the *Active event region* and the *NBA Update event region*. The relationship of these two regions is shown in Figure 1-8.

There are several important details to understand regarding how event regions are processed during simulation:

- Events in a begin-end statement group are scheduled into an event region in the order in which the statements are listed, and are executed in that order when the event region is processed.
- Events from concurrent processes, such as multiple always procedures, are scheduled into the event region in an arbitrary order chosen by the simulator. The RTL designer has no control over the order in which concurrent events are scheduled, but the designer does have control regarding the region in which events are scheduled.
- Simulators will execute all scheduled events in a region before transitioning to the next region. As events are processed, they are permanently removed from the event list. Each region will be empty before simulation proceeds to the next region.
- As events are processed in a later region, they can possibly schedule new events in a previous region. After the later region has been processed, simulation will cycle back through the event regions to process any newly scheduled events. The iterations through all event regions will continue until all regions are empty (i.e.: no new events for that moment of simulation time are being scheduled).
- The transition from one event region to the next is referred to as a *delta*. Each iteration through all event regions is referred to as a *delta cycle*.

Figure 1-8: Simplified SystemVerilog event scheduling flow



NOTE

Proper usage of these two event regions is critical in order to obtain correct RTL simulation results.

SystemVerilog has two types of assignment operators, *blocking assignments* and *nonblocking assignments*. The blocking assignment is represented with an equal sign ($=$), and is used to model combinational logic, such as Boolean operations and multiplexors. The nonblocking assignment is represented with a less-than-equal sign ($<=$), and is used to model sequential logic, such as latches and flip flops. Blocking assignments are scheduled in the Active event region. Nonblocking assignments are scheduled for the right-hand side to be evaluated in the Active event region, and the left-hand side to be updated in the NBA Update region. (NBA stands for nonblocking assignment).

Chapters 7 and 8 discuss modeling RTL of combinational logic and sequential logic in more detail. These chapters show — and emphasize — the proper usage of blocking and nonblocking assignments in zero-delay RTL models.

1.5.3.6 Event scheduling example

Example 1-7, shows an RTL model of an 8-bit D-type register that loads its `d` input on every positive edge of `clk`, a top-level module that contains a clock oscillator that has a 10 nanosecond period and begins at simulation time zero with a logic value of 0, and a test module that generates stimulus on the `d` input.

Example 1-7: A clock oscillator, stimulus and flip flop to illustrate event scheduling

```
//////////  
// Design module with RTL model of a D-type register  
//////////  
module d_reg (input logic clk,  
               input logic [7:0] d,  
               output logic [7:0] q  
             );  
timeunit 1ns; timeprecision 1ns;  
  
always @(posedge clk)  
  q <= d;  
  
endmodule: d_reg  
  
//////////  
// Top-level verification module with clock oscillator  
//////////  
module top;  
  timeunit 1ns; timeprecision 1ns;  
  logic clk;  
  logic [7:0] d;  
  logic [7:0] q;  
  
  test i1 (.*) ; // connect top module to test module  
  d_reg i2 (.*) ; // connect top module to d_reg module  
  
  initial begin // clock oscillator  
    clk <= 0; // initialize clock at time 0  
    forever #5 clk = ~clk; // toggle clock every 5ns  
  end  
endmodule: top  
  
//////////  
// Test module with stimulus generator  
//////////  
module test (input logic clock,  
              output logic [7:0] d,  
              input logic [7:0] q  
            );  
timeunit 1ns; timeprecision 1ns;
```

```

initial begin
    d = 1;
    #7 d = 2;
    #10 d = 3;
    #10 $finish;
end
endmodule: test

```

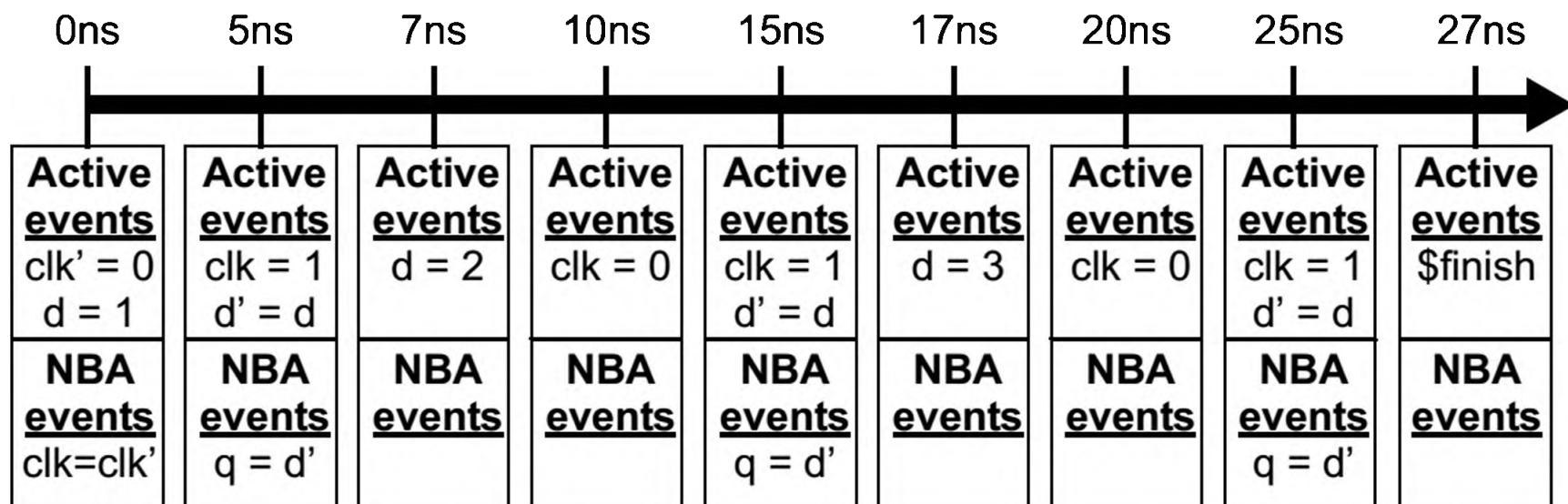
For the clock initialization in the module top, the simulator will schedule an Active event to evaluate the right-hand side of the assignment to `clock` at 0 nanoseconds, and a Nonblocking update assignment to change the value of `clk` at 0 nanoseconds. Active events are also scheduled to change `clock` at 5 nanoseconds, 10 nanoseconds, and so forth.

For the test stimulus, the simulator will schedule Active events on `d` at times 0 nanoseconds, 7 nanoseconds, and 17 nanoseconds ($7 + 10$), and schedule a `$finish` command at time 27 nanoseconds ($7 + 10 + 10$).

For the RTL flip flop, the simulator will schedule Active events to evaluate `d` at times 5 nanoseconds, 15 nanoseconds and 25 nanoseconds (the positive edges of `clk`), and NBA Update events to update `q` at times 5 nanoseconds, 15 nanoseconds and 25 nanoseconds.

Figure 1-9 illustrates what the simulation time line might look like when Example 1-7 is simulated. This diagram shows the two-step process for nonblocking assignments as an Active event that assigns to a internal temporary variable that is the prime of the signal name, and an NBA update event that assigns the internal temporary variable to the actual signal.

Figure 1-9: Simulation time line and time slots with some events scheduled



Note that Figure 1-9 is a conceptual illustration of simulation behavior. It does not reflect how simulators implement this behavior in the simulator's internal algorithms. The full event scheduling algorithm is much more complex than shown in this diagram, and is described in full in the IEEE 1800 SystemVerilog standard.

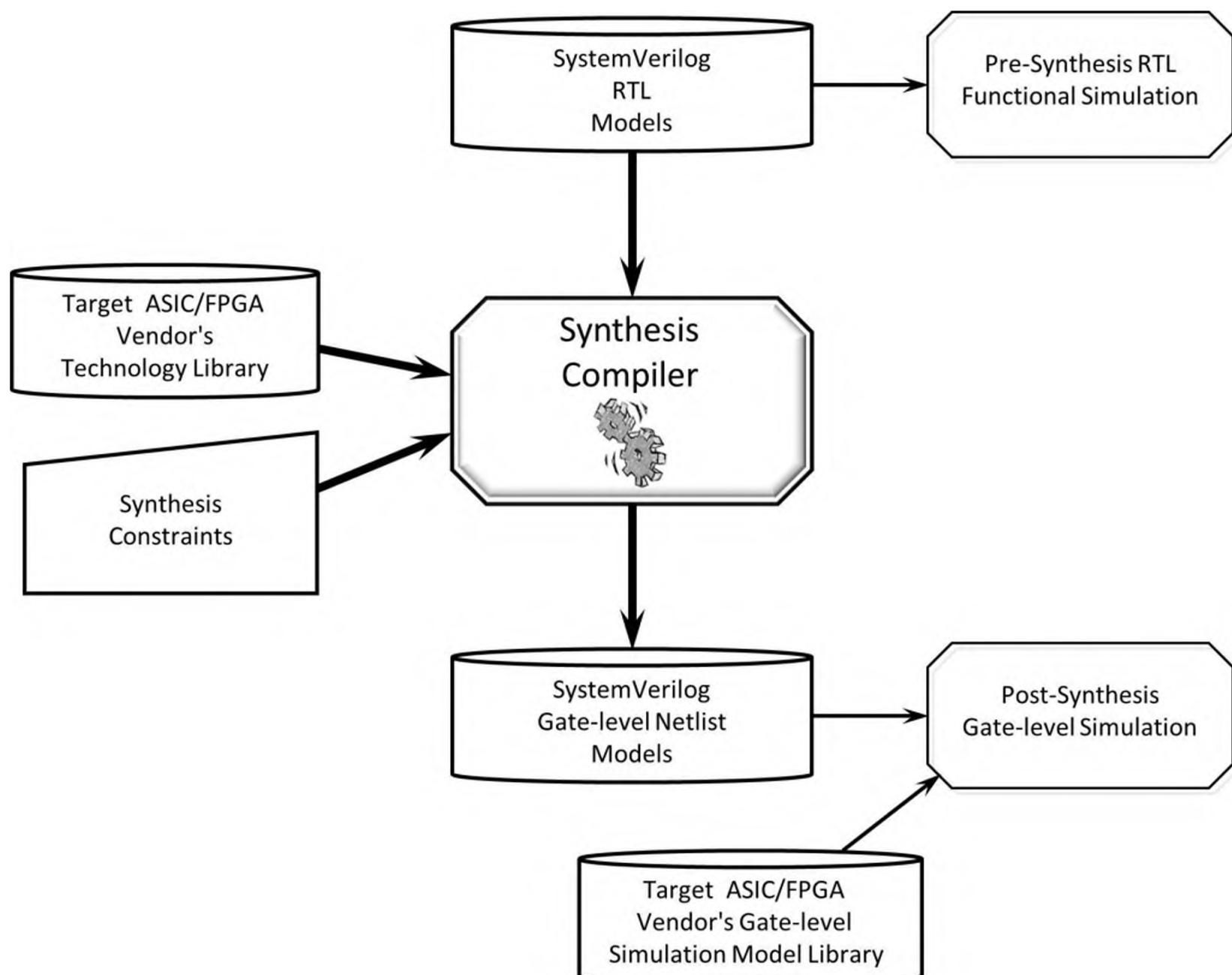
1.6 Digital synthesis

A synthesis compiler adds the implementation details to an abstract RTL model. A synthesis compiler:

1. Translates RTL functionality into equivalent functionality, represented with generic logic gates.
2. Maps the generic gates to a specific ASIC or FPGA target implementation.
3. Performs logic optimizations to meet clock speed requirements.
4. Performs logic optimizations to meet area and power requirements.
5. Performs logic optimizations to meet setup and hold times.

Figure 1-10 illustrates the general flow for digital synthesis with SystemVerilog.

Figure 1-10: SystemVerilog synthesis tool flow



A synthesis compiler requires three primary types of input information:

1. ***The SystemVerilog RTL models*** — these models are written by design engineers, and represent the functional behavior that needs to be realized in the ASIC or FPGA.

2. ***The technology library for the target ASIC or FPGA*** — this library is provided by the ASIC or FPGA vendor, and contains the definitions of the standard cells (for ASIC) or gate-array blocks (for FPGA) that are available to implement the desired functionality.
3. ***Synthesis constraint definitions*** — these constraints are defined by the design engineer, and provide the synthesis compiler information that is not available in the RTL code, such as the desired clock speed, area and power goals that need to be realized in the ASIC or FPGA.

For front-end design and verification purposes, the primary output of synthesis is a *gate-level netlist*. A netlist is a list of components and the wires (called nets) that connect these components together. The components referenced in the netlist will be ASIC standard cells or FPGA blocks that were used to implement the desired functionality. This netlist can be in a number of formats, including EDIF, VHDL, Verilog-2001 or SystemVerilog. Only the SystemVerilog output will be used in this book.

Simulation models of each component are required in order to simulate a SystemVerilog netlist. A simulation library written in SystemVerilog will be provided by the target ASIC or FPGA vendor. Often, these libraries only use the Verilog-2001 subset of SystemVerilog. These components are modeled at the gate-level with detailed propagation delays. The models are very different than the abstract RTL models written by design engineers. This book does not examine this low-level of modeling.

1.6.1 SystemVerilog synthesis compilers

There are several SystemVerilog synthesis compilers available that support the SystemVerilog language. Electronic Design Automation (EDA) companies, such as Cadence, Mentor Graphics, and Synopsys, sell commercial synthesis compilers. Some FPGA vendors, such as Xilinx and Intel (formerly Altera) provide proprietary synthesis compilers that are specific to that vendor's technology. The examples in this book were tested with several synthesis compilers (see page xxx in the Preface).

Synthesis compiler limitations. The synthesizable RTL models shown in this book reflect the subset of the SystemVerilog language that is supported by most major synthesis compilers.

SystemVerilog is a dual purpose language. One purpose is to model the behavior of digital hardware. A second purpose is to code verification programs to test the hardware models. These two purposes have very different language requirements. Many general purpose programming constructs are useful for both purposes, such as and if-else decision or a for-loop. Other language features are intended strictly for verification, such as constrained random test generation. These verification constructs do not represent hardware functionality, and are not intended to be supported by synthesis compilers.

The IEEE has not identified an official synthesizable subset for SystemVerilog. This shortcoming of the standard has led to important deviations in what each synthe-

sis compiler supports for a synthesizable SystemVerilog language subset. Furthermore, the subset identified by a specific synthesis compiler can, and almost certainly will, change from one version of a synthesis product to the next version of that product.

This book is synthesis compiler neutral. The examples in this book will work with nearly all major synthesis compilers. There, however, a few examples that, at the time this book was written, would compile with most major synthesis tools, but did not compile all of the synthesis compilers.

The *Mentor Graphics Precision RTL Synthesis™* compiler was used to generate the synthesis schematic output shown with many of the examples. This compiler was used because the schematics created by this tool were easy to capture in black-and-white, and to adapt to the page size of the book.

The book does not provide details on using any specific synthesis product. Readers are encouraged to synthesize the examples in this book, as well as examples conceived on their own. Refer to the documentation provided with a specific synthesis compiler product for information on invoking and running that compiler.

1.6.2 Synthesis Compilation

Synthesis compilers have a different goal than simulation compilers. Both types of compilers need to check the SystemVerilog RTL source code for syntactic correctness, but that is where the similarity ends. Simulation is a dynamic process that involves simulated time, event scheduling, applying stimulus and verifying outputs. Synthesis is a static translation and optimization process that does not involve any of these simulation goals. Synthesis compilers need to ensure that the code meets the language restrictions necessary in order to translate RTL functionality into the types of logic gates supported in ASIC and FPGA implementations. These restrictions include checking that the RTL code has clearly defined clock cycle activity, single driver logic, etc. Synthesis compilers only need to compile the RTL models. Synthesis does not need to compile the testbench code with its stimulus generation and output verification.

1.6.2.1 Single-file and multi-file compilation

Large designs are partitioned into many sub blocks. Typically, each sub block will be stored in a separate file. To simulate a partitioned design, simulation requires that all of these sub blocks be compiled and connected together. Synthesis, on the other hand, can often compile and treat each sub block separately. Indeed, it is often necessary to do this. Synthesis optimizations and technology mapping are compute intensive processes. Synthesizing too many sub blocks together can lead to suboptimal *Quality of Results* (QoR).

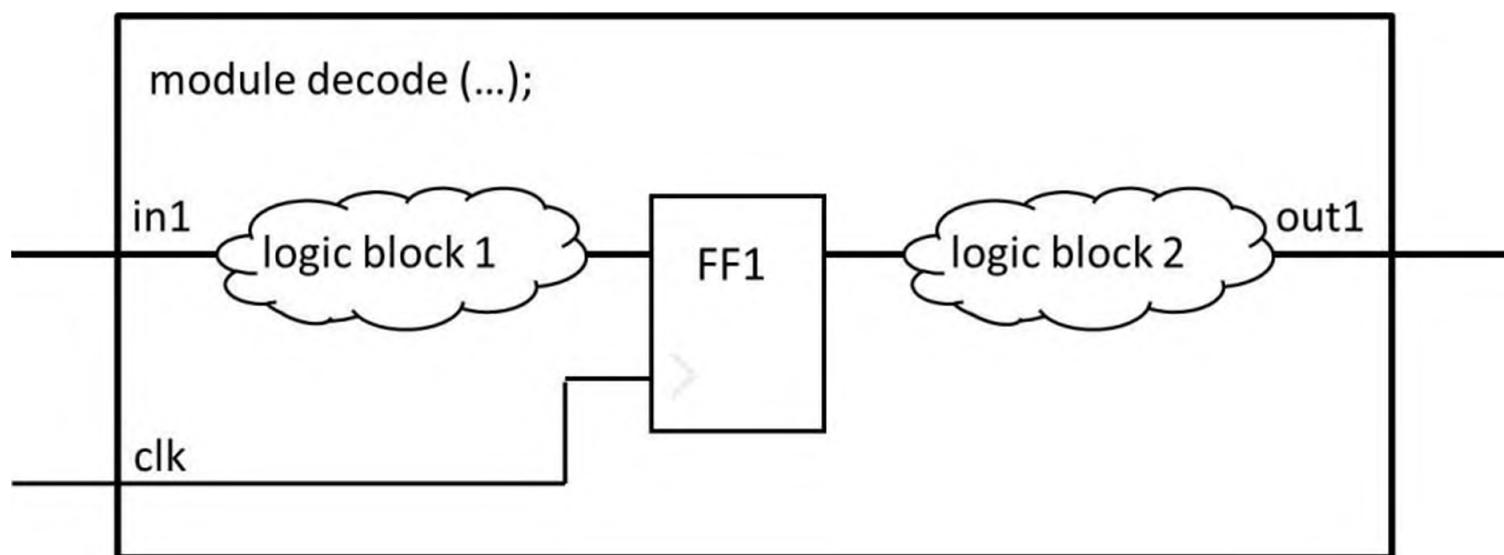
There are two important considerations when synthesizing sub blocks of a design. First, any definitions used in the sub block that come from definition packages will require that the package be compiled along with the sub block, and in the proper

order. If the same package is used by several sub blocks, the package will need to be recompiled with each sub block that is compiled separate from other sub blocks. The second consideration is that any global declarations, including `'define` compiler directives, will not be seen in each separate compilation. This same problem exists with simulators that support single-file compilation, and the same guideline discussed in section 1.5.2 (page 21) applies — avoid the use of global declarations and definitions. Single-file and multi-file compilation do not see the same global space.

1.6.3 Constraints

Figure 1-10 on page 31 showed that one of the three primary inputs into synthesis is constraint definitions. Constraints are used to define information that synthesis needs, but which is neither in the RTL models nor the ASIC/FPGA vendor's technology library. Figure 1-11 illustrates a simple circuit where some of the information that synthesis requires must be specified by the design engineer, using synthesis constraints.

Figure 1-11: Diagram of a simple circuit requiring synthesis constraints



The process of synthesizing this functional flow of data into logic gates involves:

- Mapping the inferred flip-flop FF1 to an appropriate flip-flop in the target ASIC or FPGA.
- Mapping the functionality described in `logic_block_1` to the standard cells or logic blocks of the target ASIC or FPGA.
- Optimizing the implementation of `logic_block_1` to meet the setup and hold requirements of FF1.
- Mapping the functionality described in `logic_block_2` to the standard cells or logic blocks of the target ASIC or FPGA.
- Optimizing the implementation of `logic_block_2` to meet the output arrival requirements of the design specification.

In order to realize the simple circuit shown in Figure 1-11 in a target ASIC or FPGA, synthesis compilers must know:

1. The propagation delays, area and power requirements of the standard cells or logic blocks used to implement `logic_block_1` and `logic_block_2`.
2. The setup and hold times of `FF1`.
3. The period or frequency of `clk`, such as 100Mhz.
4. The arrival time of `in1` relative to the active edge of `clk`.
5. The drive capability of the external source for `in1`.
6. The arrival time of `out1` relative to the active edge of `clk`.
7. The output drive requirement for `out1`.

This information will not be in the RTL model. The specifications for the first two items in this list, propagation delays and setup/hold times, will come from the technology library provided by the ASIC or FPGA vendor. The remaining details must be specified by the design engineer who is synthesizing the design. These specifications are referred to as *synthesis constraints*. A larger, more complex design will require many more synthesis constraints. The RTL coding examples in this book will discuss applicable synthesis constraints, where appropriate. Guidelines are also provided for simplifying the constraints that must be specified.

The way in which synthesis constraints are specified varies with different synthesis compilers. This book does not discuss how constraints are specified for any specific synthesis compiler. Readers are encouraged to refer to the product documentation for this information.

The book “*Constraining Designs for Synthesis and Timing Analysis*”⁴ is a good source for more information on specifying timing constraints for synthesis, as well as for static timing analysis.

1.7 SystemVerilog lint checkers

Synthesizable RTL coding rules impose a number of restrictions on how the SystemVerilog language can be used. It can be frustrating to invest many hours of time writing a SystemVerilog model that appears to follow synthesis coding rules and simulates correctly, only to find out it will not synthesize on a specific synthesis compiler. It is possible to periodically run the RTL code through a synthesis compiler as the RTL functionality is being developed, in order to ensure that the final RTL code will be synthesizable. Synthesis compilers are expensive software tools, however, and many companies have only a limited number of synthesis licenses. It is not practical

4. “*Constraining Designs for Synthesis and Timing Analysis*”, Sridhar Gangadharan and Sanjay Churiwala. Copyright 2013, Springer, New York, NY. ISBN 978-1-4614-3268-5.

to tie up a synthesis license in order to periodically check that RTL code adheres to synthesis coding rules while that code is being developed.

Lint checkers can be used in place of a synthesis compiler to check that RTL code meets synthesizable RTL coding rules. A SystemVerilog *lint checker* is a software tool that parses SystemVerilog source code and checks it against specific coding rules. Lint checkers are configurable. Engineers can enable or disable specific checks, and often add checks for guidelines or requirements at a specific company or on a specific project.

There are a number of commercial lint checkers for SystemVerilog. This book does not go into details on running any specific lint checker. Readers are encouraged to use a lint checker of their choice when trying out the examples in this book or other SystemVerilog RTL models.

1.8 Logic Equivalence Checkers

A Logic Equivalence Checker (LEC) is an engineering tool that analyzes the functionality of two models to determine if the models are logically the same. LEC tools can compare two versions of an RTL model, two versions of a gate-level model. The most common application of logic equivalence checking, however, is to compare the functionality of an RTL model with the post-synthesis gate-level gate-level model. This type of checking is often done after Engineering Change Orders (ECOs) or other types of changes have been made to the gate-level design that was created during the synthesis process.

Logic equivalence checking is one type of formal verification, and is sometimes referred to as formal verification. The term *formal verification*, however, is a more general term that can also encompass other types of engineering tools. Another type of formal verification is the use of SystemVerilog Assertions (SVA) to describe design behavior that should, or should not, occur.

The use of assertions in design and verification is not discussed in this book. The author of this book is a proponent on the use of SystemVerilog in design code, and having design engineers add assertions into RTL models as part of writing the RTL code. More information on this topic can be found in the white paper, “*Who Put Assertions In My RTL Code? And Why? How RTL Design Engineers Can Benefit from the Use of SystemVerilog Assertions*”⁵.

5. “*Who Put Assertions In My RTL Code? And Why? How RTL Design Engineers Can Benefit from the Use of SystemVerilog Assertions*”, Stuart Sutherland. Presented at the 2013 Silicon Valley Synopsys Users Group Conference (SNUG). Available for download at sutherland-hdl.com.

1.9 Summary

SystemVerilog is an IEEE industry standard for a hardware design and verification language. The standards number is IEEE 1800. The name SystemVerilog replaces the legacy Verilog name. The SystemVerilog language is a full super set of the original Verilog Hardware description language. SystemVerilog is a dual-purpose language, and is used to model digital hardware functionality, as well as verification test-benches.

Hardware behavior can be represented at several abstraction levels, ranging from very detailed gate-level models to very abstract transaction-level models. The focus of this book is on writing SystemVerilog models at the Register Transfer Level (RTL). RTL models represent cycle-based behavior, with little or no reference to how the functionality will be implemented in silicon.

SystemVerilog RTL models, when coded correctly, can be simulated and synthesized. Simulation uses a user-defined testbench to apply stimulus to the inputs of the design, and verify the design functions as intended. Simulation does not impose any restrictions on how the SystemVerilog language is used. Synthesis translates abstract RTL functionality into a detailed gate-level implementation, and targets this low-level implementation to a specific ASIC or FPGA technology. Synthesis compilers limit what SystemVerilog language constructs can be used, and how those constructs can be used. Lint checker tools can be used to check that an RTL model adheres to synthesis compiler restrictions.

* * *

Chapter 2

RTL Modeling Fundamentals

Abstract — This chapter covers the general modeling constructs used for writing Register Transfer Level (RTL) models in SystemVerilog. The topics presented in this section include:

- Modules and procedural blocks
- SystemVerilog language rules (comments, reserved keywords, user-defined names)
- Best practices for naming conventions
- System tasks and functions
- Compiler directives
- Module declarations
- Module port declarations
- Hierarchical models
- Module instances
- Port connections
- Netlists

2.1 Modules and procedural blocks

Modules are the primary modeling block in SystemVerilog. Modules are declared with the keyword `module`, and are completed with the keyword `endmodule`. A module is a container that holds information about a model, including input and outputs to the module, data type declarations, and executable code. Section 2.3 (page 52) discusses module declarations in more detail.

Executable code within a module is contained in *procedural blocks*. SystemVerilog has two primary types of procedural blocks, *initial procedures*, defined with the keyword `initial`, and *always procedures*, defined with the keywords, `always`, `always_comb`, `always_ff` and `always_latch`. Initial procedures are not synthesizable, and are not used for RTL modeling. Always procedures are infinite loops. When the procedure has completed execution of the last statement in the procedure, the procedure automatically returns to the beginning, and starts the procedure again.

For RTL modeling, an always procedure begins with a sensitivity list, such as `@(posedge clock)`. The various forms of always procedures, and how they are used for RTL modeling, are discussed in more detail in Chapters 6 through 9.

A procedural block can contain a single statement, or a group of statements. Multiple statements in a procedural block are grouped together between the keywords **begin** and **end**. Statements between **begin** and **end** are executed in the order in which they are listed.

2.2 SystemVerilog language rules

SystemVerilog is a specialized programming language. Like any programming language, SystemVerilog has a number of syntax and semantic rules that must be followed. *Syntax* refers to the legal combinations of symbols in a programming language. Syntax is the reserved words and tokens of the language, and the order and context in which these words and tokens can be used. The formal syntax in SystemVerilog is defined using a common programming convention called *Backus–Naur Form* (BNF). BNF can be difficult to read, but accurately describes the SystemVerilog language syntax. It is an essential part of the official IEEE 1800 SystemVerilog standard for companies that implement SystemVerilog software tools, such as simulators and synthesis compilers. Engineers using SystemVerilog often refer to “the BNF” when discussing SystemVerilog syntax.

Semantics refers to the meaning that should be inferred from syntactically legal code. In the code `sum = a + b;`, the semantics includes what type of addition should be inferred, such as: whether the addition should be performed using integer arithmetic or floating-point arithmetic, signed or unsigned arithmetic, and with or without an overflow. The division between syntax and semantics is not always obvious. It is common for texts on SystemVerilog to use the term syntax to mean both syntax and semantics. This book follows that loose convention.

Engineers interested in the exact syntax and semantics of the full SystemVerilog language can refer to the IEEE 1800 SystemVerilog standard (see Appendix D for how to obtain a copy of the standard).

2.2.1 Comments

SystemVerilog has two types of general comments: *one-line* and *block*. There are also two special types of comments: *pragmas* and *attributes*.

One-line comments begin with a `//` token, and are terminated with a new line. The comment can begin anywhere on a line, and comments out the rest of that line. According to the SystemVerilog standard, the `//` token, and all text following it up to the new line, are ignored. Synthesis compilers, however, do not fully adhere to this rule. Synthesis compilers hide synthesis-specific commands within a form of comment called a pragma, which is discussed in on the following page.

Block comments begin with a `/*` token, and are terminated with a `*/` token. New lines between the token are ignored, allowing the comment to span any number of lines. Block comments cannot be nested. Once a `/*` starting token is encountered, a parser will ignore all text, including another `/*`, until an ending `*/` token is encountered. A nested `/*` is not seen as the start of a nested comment.

Best Practice Guideline 2-1

The code portions of a model should only contain one-line comments that begin with `//`. Do not use block comments in the code body that encapsulate the comment between `/*` and `*/`.

Block comments are useful for temporarily commenting out a section of code when debugging a model. Since block comments cannot be nested, it is difficult to temporarily comment out code sections if block comments have been used within that section. Block comments are okay for model headers.

Example 2-1 shows the use of both one-line and block comments.

Example 2-1: RTL model showing two styles of comments

```
/*
 * RTL model of a 32-bit adder/subtractor
 *
 * Developed for Project X.
 *
 * Creator: Stuart Sutherland.
 *
 * Specification:
 * Performs unsigned 32-bit arithmetic, with no overflow or
 * underflow. A mode control selects whether the operation is
 * an add or a subtract.
 * - Add when mode is low
 * - Subtract when mode is high
 * The output is registered.
 * The register has an active low, asynchronous reset.
 *
 * NOTE: This model is intended to be synthesized in conjunction
 * with blocks that provide registered values for the a, b, and
 * mode inputs. These blocks must use the same clock, so that
 * no clock synchronizers are needed within this model.
 *
 * Revision History:
 * 1.0: 25 Jun 2016: Initial development
 * 1.1: 7 Jul 2016: Changed mode to match revised design spec.
 *
 */
```

```

module rtl_adder_subtractor
(input logic clk, // clock input
 input logic rstN, // active low reset input
 input logic mode, // add/subtract control input
 input logic [31:0] a, b, // 32-bit inputs
 output logic [31:0] sum // 32-bit output
);

// registered adder/subtractor with async reset
always_ff @(posedge clk or negedge rstN) // async reset
  if (!rstN) sum <=0; // active low reset
  else case (mode)
    1'b0: sum <= a + b; // unsigned integer add, no overflow
    1'b1: sum <= a - b; // unsigned integer subtract, no
  endcase // underflow
endmodule: rtl_adder_subtractor

```

Attributes begin with a `(*` token, and are terminated with a `*)` token. Attributes must be on a single line, and cannot contain a new line. An attribute is a special type of comment that contains information for specific software tools. A synthesis attribute, for example, will be ignored by simulators, but will be read by synthesis compilers. An attribute comment is associated with a specific language construct. For example, a module can have an attribute associated with it, and a programming statement can have an attribute associated with it. Attributes are used extensively by mixed analog/digital simulators. An old, obsolete Verilog synthesis standard, IEEE1364.1, defined several synthesis attributes. These attributes are seldom used, and are not discussed in this book.

Pragmas are a special form of a one-line or block comment. A pragma comment begins with a word that identifies the rest of the comment as containing information for specific software tools. Synthesis compilers make extensive use of pragmas to provide information that aids the synthesis process, but which is irrelevant to simulation. Comments that begin with the word **synthesis** or **pragma** are recognized as a pragma by nearly all commercial synthesis compilers.

An example of using synthesis pragmas is:

```

always_comb begin
  if (select == 0) y = a;
  // synthesis translate_off
  else if ($isunknown(select))
    $warning("select has incorrect value at %t", $realtime);
  // synthesis translate_on
  else y = b;

```

Another example synthesis pragma is:

```

case (mode) // synthesis full_case

```

The `// synthesis translate_off` pragma informs synthesis compilers to ignore any source code that follows, until a `// synthesis translate_on` pragma is encountered. These pragmas are useful for hiding from synthesis any debugging or error handling code intended only for simulation. The `// synthesis full_case` pragma directs synthesis to perform specific logic optimizations on case decision statements. This directive is discussed in more detail in section 9.3.6 (page 345) of Chapter 9.

NOTE

At the time this book was written, one commercial synthesis compiler did not recognize `// synthesis` as a synthesis pragma. That compiler required that pragmas start with `// pragma` or `// synopsys`.

Some synthesis compilers might also recognize other words within a comment as defining a synthesis pragma. These other words are tool-specific, and the comment might not be treated as a pragma by every synthesis compiler.

Synthesis pragmas are important constructs for writing synthesizable RTL models. Other synthesis pragmas will be discussed throughout this book.

NOTE

Care must be taken when using synthesis pragmas. Simulators ignore these comments, and not simulate the effects of the pragma. This means that the code that is simulated and verified for correct functionality might not be the same behavior implemented by a synthesis compiler.

Coding guidelines on the proper usage of specific pragmas are presented as those pragmas are discussed in this book, .

2.2.2 *White space*

White space is used for two purposes: to separate words, and to make source code easier to read. The white space characters in SystemVerilog are spaces, tabs, new lines, and, in some contexts, end-of-file.

In the following code snippet, white space is required to separate the words `module` and `top`, and to separate `logic` and `a`. White space is optional between the names `a`, `b` and `c`, because some other token, in this case a comma, separates these names.

```
module top;
  logic a,b,c;
```

Although white space is optional whenever there is some other language token to separate words, good use of white space is important for making code more readable

and maintainable. Example 2-2 illustrates a simple RTL 32-bit adder with minimal white space. Example 2-3 shows the same RTL adder, but with added white space. Although both examples are syntactically correct and functionally identical, it should be obvious that the second example is a more readable coding style.

Example 2-2: SystemVerilog RTL model with minimum white space

```
module rtl_adder_bad_style(input logic[31:0]a,b,output logic
[31:0]sum,output logic co);always_comb begin{co,sum}=a+b;end
endmodule:rtl_adder_bad_style
```

Example 2-3: SystemVerilog RTL model with good use of white space

```
module rtl_adder_good_style
( input logic [31:0] a, b,
  output logic [31:0] sum,
  output logic          co
);
  always_comb begin
    {co, sum} = a + b;
  end
endmodule: rtl_adder_good_style
```

Spaces versus tabs. Example 2-3 uses white space to indent code. One level of indentation is used between the `module...endmodule` block. A second level of indentation is used for the code between the `begin...end` block. Indentation helps make it easier to see what code is within each block. Either the space character or a tab can be used to indent code. Programmers are often passionate about whether spaces or tabs should be used for indentation. In practice, either style works fine. It is helpful, however, if all members of a design team use one style or the other. Consistent use of either spaces or tabs within a project makes it easier for engineers to read and maintain code developed by other engineers on the project.

The author of this book prefers using spaces for indentation, and specifically two spaces for each level of indentation. By using spaces, code will indent the same amount on all editors, regardless of what the tab settings might be. Two spaces is enough to see each level of indentation, and yet does not indent so far over that code does not fit well on each line.

2.2.3 Reserved keywords

The SystemVerilog standard reserves a number of words, known as *keywords*, for use by the SystemVerilog language. It is illegal to use these keywords for user-defined names. All reserved keywords are in lower-case. In this book, SystemVerilog keywords appearing in code are in bold, and keywords referred to in the text are in Courier-bold.

Table 2-1 lists the reserved keywords in the 1800-2012 SystemVerilog standard.

Table 2-1: SystemVerilog-2012 reserved keywords

accept_on	endchecker	inside	pullup	sync_accept-
alias	endclass	instance	pulsestyle-_on-detect	_on
always	endclocking	int	pulsestyle-_on-event	sync_reject-
always_comb	endconfig	integer	pure	_on
always_ff	endfunction	interconnect	rand	table
always_latch	endgenerate	interface	randc	tagged
and	endgroup	intersect	randcase	task
assert	endinterface	join	randsequence	this
assign	endmodule	join_any	rcmos	throughout
assume	endpackage	join_none	real	time
automatic	endprimitive	large	realtime	timeprecision
before	endprogram	let	ref	timeunit
begin	endproperty	liblist	reg	tran
bind	endspecify	library	reject_on	tranif0
bins	endsequence	local	release	tranif1
binsof	endtable	localparam	repeat	tri
bit	endtask	logic	restrict	tri0
break	enum	longint	return	tril
buf	event	macromodule	rnmos	triand
bufif0	eventually	matches	rpmos	trior
bufif1	expect	medium	rtran	trireg
byte	export	modport	rtranif0	type
case	extends	module	rtranif1	typedef
casex	extern	nand	s_always	union
casez	final	negedge	s_eventually	unique
cell	first_match	nettype	s_nexttime	unique0
chandle	for	new	s_until	until
checker	force	nexttime	s_until_with	until_with
class	foreach	nmos	scalared	untyped
clocking	forever	nor	sequence	use
cmos	fork	noshowcan-	shortint	uwire
config	forkjoin	celled	shortreal	var
const	function	not	showcancelled	vectored
constraint	generate	notif0	signed	virtual
context	genvar	notif1	small	void
continue	global	null	soft	wait
cover	highz0	or	solve	wait_order
covergroup	highz1	output	specify	wand
coverpoint	if	package	specparam	weak
cross	iff	packed	static	weak0
deassign	ifnone	parameter	string	weak1
default	ignore_bins	pmos	strong	while
defparam	illegal_bins	posedge	strong0	wildcard
design	implements	primitive	strong1	wire
disable	implies	priority	struct	with
dist	import	program	super	within
do	incdir	property	supply0	wor
edge	include	protected	supply1	xnor
else	initial	pullo		xor
end	inout	pull1		
endcase	input	pulldown		

(Some keywords in this table have been hyphenated in order to fit the format of this book. The actual keywords do not contain hyphens.)

2.2.4 *Keyword backward compatibility — ‘begin_keywords’*

The SystemVerilog language has evolved over time, and the IEEE has released several successive versions of the Verilog and SystemVerilog standards. See Chapter 1 section 1.1 (page 1) for a brief history of this evolution. Each version of the standard has reserved additional keywords. Future versions of the SystemVerilog standard will likely reserve more keywords.

This evolution of reserved keywords can mean that code written for one version of the Verilog or SystemVerilog standard might have user-defined names that became reserved keywords in a later version of the standard. SystemVerilog provides a compiler directive pair to handle keyword backward compatibility.

The ‘**begin_keywords**’ directive is followed by a specific version of an IEEE Verilog or SystemVerilog standard, which is specified between quotation marks. Software tools reading the SystemVerilog source code will then use the reserved keyword list from that version of the standard until either a ‘**end_keywords**’ directive or another ‘**begin_keyword**’ directive is encountered. Multiple ‘**begin_keyword**’ directives are stacked. A ‘**end_keywords**’ directive will return to the previous ‘**begin_keywords**’ directive in the stack.

The IEEE versions that can be specified with ‘**begin_keyword**’ are:

- “1364-1995” — uses the keyword list from Verilog-95
- “1364-2001” — uses the keyword list from Verilog-2001
- “1364-2005” — uses the keyword list from Verilog-2005
- “1800-2005” — uses the keyword list from SystemVerilog-2005
- “1800-2009” — uses the keyword list from SystemVerilog-2009
- “1800-2012” — uses the keyword list from SystemVerilog-2012
- “1800-2017” — uses the keyword list from SystemVerilog-2017 (the same as SystemVerilog 1800-2012)

Appendix B lists the set of reserved keywords for each of these standards.

Best Practice Guideline 2-2

Specify a ‘**begin_keywords**’ directive before every module, interface and package. Specify a matching ‘**end_keywords**’ directive at the end of every module, interface and package.

Using these directives documents which version of SystemVerilog was in use when the code was developed, and helps ensure that the code will be compatible with current and future versions of SystemVerilog.

NOTE

SystemVerilog compiler directives are not bound by files. A '**begin_keyword**' directive in one file will affect all subsequent files that are read in by the compiler for that invocation of the compiler. This can result in file order dependencies and different behavior when files are compiled separately instead of together.

To avoid these side effects, every '**begin_keyword**' directive should be paired with a '**end_keywords**' directive in the same file.

Example 2-4 and Example 2-5 illustrate using the '**begin_keyword**' and '**end_keywords**' directives in order to mix a legacy Verilog model with a newer SystemVerilog model. Example 2-4 is written using the older Verilog-2001 standard. The code uses **priority** as a user-defined name, which was legal in Verilog. Example 2-5 is written using SystemVerilog, where **priority** is a reserved keyword. By using the '**begin_keyword**' directives, a SystemVerilog compliant simulator or synthesis compiler can read in both models, either together or separately, even though Example 2-4 is not compatible with SystemVerilog's reserved keyword set.

Note — Examples 2-4 and 2-5 are functionally equivalent, but use different programming constructs. Chapter 6 discusses the programming constructs used in these examples, and Chapter 7 discusses how these two modeling styles can affect synthesis results.

Example 2-4: Using '**begin_keywords**' with a legacy Verilog-2001 model

```
'begin_keywords "1364-2001" // use Verilog-2001 keywords
`timescale 1ns/1ns
module priority_decoder_1
  (input wire [3:0] select,
   output reg [2:0] priority // "priority" is not a keyword
 );
  // return bit number of highest bit set, or 7 if none set
  always @(select) begin
    casez (select) // synthesis full_case
      4'b1???: priority = 4'h3;
      4'b01??: priority = 4'h2;
      4'b001?: priority = 4'h1;
      4'b0001: priority = 4'h0;
      4'b0000: priority = 4'h7;
    endcase
  end
endmodule // priority_decoder_1
`end_keywords
```

Example 2-5: Using 'begin_keywords** with a SystemVerilog-2012 model**

```

`begin_keywords "1800-2012" // use SystemVerilog-2012 keywords
module priority_decoder_2
(input logic [3:0] select,
output logic [2:0] high_bit
);
// return bit number of highest bit set, or 7 if none set
always_comb begin
priority case (1'b1)           // "priority" is a keyword
    select[3]: high_bit = 4'h3;
    select[2]: high_bit = 4'h2;
    select[1]: high_bit = 4'h1;
    select[0]: high_bit = 4'h0;
    default   : high_bit = 4'h7;
endcase
end
endmodule: priority_decoder_2
`end_keywords

```

NOTE

The code examples shown in this book do not include the **'begin_keywords** and **'end_keywords** directives, but the downloadable examples files do have these directives. They are omitted from the book to save space and to focus on the concepts being illustrated in each example.

Using invocation options to distinguish Verilog and SystemVerilog code. Simulators and synthesis compilers also provide invocation options to specify a version of the Verilog or SystemVerilog standard to be used during compilation. These invocation options are not part of the IEEE SystemVerilog standard, and are different for each tool. Using **'begin_keywords** and **'end_keywords** works with all SystemVerilog compliant software tools, and documents the language version used in the model.

Using .v and .sv file names to distinguish Verilog and SystemVerilog code. The IEEE SystemVerilog standard does not require files be named with any specific file extensions. However, simulators and synthesis compilers use the file name extension to determine which reserved keyword list should be used by a compiler. Files ending with **.v** are assumed to be written using an older Verilog reserved keyword list. Files ending with **.sv** are assumed to be written using a newer SystemVerilog reserved keyword list.

A problem with file extensions is that they do not indicate which version of Verilog or SystemVerilog to use. Simulators and synthesis compilers that use file extensions can end up assuming different versions. This means a file ending with **.v** or **.sv** might work correctly on one software tool, and have a keyword conflict with another

software tool. Using the '**begin_keywords**' and '**end_keywords**' directives works with all SystemVerilog compliant software tools, and is preferred over relying on file extensions.

2.2.5 Identifiers (*user-defined names*)

Engineers writing SystemVerilog code need to create names for many types of objects. User-defined names are called *identifiers*.

Legal identifiers. The syntax rules for identifiers are:

- Must begin with the characters: a through z, A through Z, or an underscore (_).
- May contain the characters: a through z, A through Z, 0 through 9, an underscore (_), or a dollar sign (\$).
- May be from 1 to 1024 characters in length.
- May not be a reserved keyword.

Some examples of legal user-defined names are:

add32 master_clock resetN enable_

Case sensitivity. SystemVerilog is *case sensitive*, meaning lower-case and upper-case characters are treated as being different. The identifiers **Input** and **INPUT** are different names, and are not the same as the reserved keyword **input** (all reserved keywords in SystemVerilog are in lower-case).

Escaped names . Characters that are normally illegal can be used in an identifier by escaping the identifier. An escaped name can use any printable ASCII character, can start with a number, and can be a reserved keyword. Escaped identifiers begin with a backslash (\) and are terminated by a white space. All characters following the backslash and up to a white space are escaped, including characters that would normally separate names, such as commas, parentheses, and semicolons. Some examples of escaped names are:

\741s74 \reset- \~enable \module

The backslash is part of the name. Every place that an escaped identifier is referenced must include the backslash at the start of the name, and terminate with a white space after the name.

Name spaces. Identifiers are local to the name space in which they are declared. The constructs in SystemVerilog that introduce a local name space are:

- *Definitions name space* — a global space that can contain declarations of **module**, **primitive**, **program**, and **interface** identifiers.
- *Package name space* — a global space that can contain declarations of **package** identifiers.

- *Component name space* — a local name space introduced by the keyword **module**, **interface**, **package**, **program**, **checker**, and **primitive**. The component name space can contain declarations of tasks, functions, checkers, instance names, named begin-end and fork-join blocks, parameter constants, named events, nets, variables, and user-defined types. A module can also contain nested declarations of modules and programs. The identifier of a nested declaration is local to the module, and is not in the definitions name space.
- *\$unit compilation unit name space* — a pseudo-global space that can contain declarations of tasks, functions, checkers, parameter constants, named events, nets, variables, and user-defined types. Declarations made outside of the component name space are in the **\$unit** space. Multiple **\$unit** names spaces can exist at the same time. Declarations in one **\$unit** space are not shared with other **\$unit** spaces.
- *Block name space* — a local name space that is introduced by tasks, functions, and named or unnamed begin-end and fork-join blocks. A block name space can contain declarations of named blocks, named events, variables, and user-defined types.
- *Class name space* — a local name space introduced by the keyword **class**. The class name space can contain declarations of variables, tasks, functions, and nested class declarations.

The same identifier cannot be declared twice in the same name space. However, it is legal for the same name space to declare an identifier and to reference an identifier of the same name that was declared in a different space. For example, when instantiating a module, the same name can be used for the module name and its instance name.

2.2.6 Naming conventions and guidelines

A full-size ASIC or FPGA model will contain hundreds of user-defined names. Identifiers need to be declared for module names, ports, constants, variables, nets, user-defined types, tasks, functions, and other SystemVerilog constructs. It is important that good naming conventions and guidelines be followed, in order for all of the SystemVerilog code in the project to be maintainable and understandable. Good naming conventions can also prevent design errors that can be tedious to detect and debug. For example, a poorly named active-low signal could easily be mistakenly used as active-high, resulting in a functional bug. The faulty code will compile and simulate or synthesize, and require time and effort to detect during verification. A well-named active-low signal is less likely to be incorrectly used as an active-high signal, thereby saving verification time and effort.

The most important guideline for good naming convention is consistency. Every engineer — and that means *every* engineer — on a design project should use the same convention for naming critical signals, such as clocks, resets, and active-low signals. User-defined types and constants should also have a consistent naming convention. The naming of other identifiers is less critical, but consistency is still helpful.

There are many possible naming conventions. For example, clock signals might all be declared with a prefix of `clock_`, `clk_` or `ck_`, or they might all be declared with

a suffix of `_clock`, `_clk` or `_ck`. All active-low signals might be declared with a prefix of `n_`, or a suffix of `_n`.

Consistency within a project for other aspects of modeling is also beneficial. This includes what goes into the comment section at the beginning of each model, code indentation, and the order of input and output ports.

The author of this book does not promote or encourage using one convention over another. What the author does promote is consistency. The consistent naming convention used in this book is:

- Clocks are named `clock` or `clk`, or have `_clk` appended to the name.
- Active-high resets are named `reset` or `rst`, Active-low resets are named `resetN` or `rstN`. Active-high sets are named `set` or `preset`. Active-low sets are named `setN` or `presetN`.
- Other active-low signals have a capital `N` appended to the name.
- Other user-defined types have `_t` appended to the name.
- Constants are in all capital letters.
- Package names have `_pkg` appended to the name.

2.2.7 System tasks and functions

SystemVerilog provides a number of special programming constructs called *system tasks* and *system functions*. A system task performs an operation, and does not have a return value. A system function calculates a value and returns that value. Some system functions perform an operation like a task, and return a pass or fail status value. For convenience, this book lumps both system tasks and system functions under the name *system tasks*.

All system task names begin with a dollar sign (`$`). Most system tasks are for simulation only, and do not represent actual logic gate-level behavior. The `$display` system task, for example, prints a user-defined message during simulation. Some other commonly used system tasks are `$info`, `$warning`, `$error` and `$fatal`. These system tasks print a message with an associated severity level. The message can be either simulator generated or user defined. Simulation-specific system tasks are ignored by synthesis compilers. Although a few of these simulation-specific system tasks will be used in the examples in this book, the details on these system tasks are not discussed in this book, since they do not represent hardware and are ignored by synthesis.

There are a few system tasks that can represent hardware behavior, and are synthesizable. Some synthesizable system tasks are used in examples in later chapters of this book, and will be discussed in conjunction with the example.

2.2.8 Compiler directives

SystemVerilog has special constructs to give commands to compilers that read in SystemVerilog source code. These constructs are referred to as *compiler directives*, and begin with a grave accent (`), sometimes referred to as a backtick, backquote, or reverse apostrophe. Compiler directives are partially global within the **\$unit** compilation name space (see Chapter 1, section 1.5.2.2). When a compiler encounters a directive, it affects all code read in by the compiler from that point on for that invocation of the compiler. The directive has no effect on code that has already been read by the compiler, and has no effect on other compilation units.

Some of the most commonly used compiler directives are:

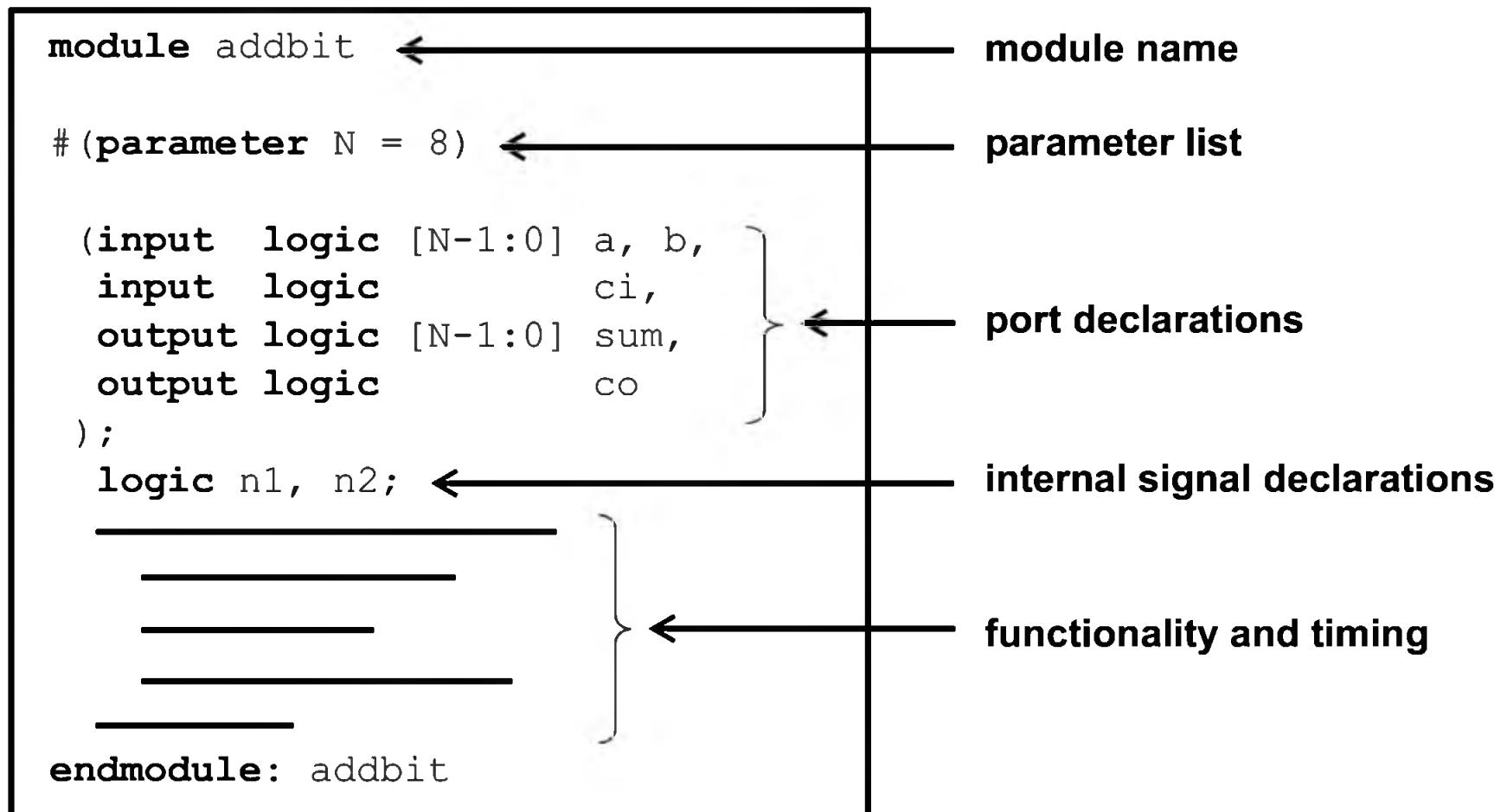
- **'include** — inserts the contents of another file at the point of the **'include** directive. The included file contents must be syntactically legal at the place in which it is inserted.
- **'define** — text substitution macro. Provides similar capability as the C #define pre-processor command.
- **'ifdef**, **'ifndef**, **'else**, **'elsif**, **'endif** — conditional compilation. Allows SystemVerilog source code to be optionally compiled, based on whether a macro name has been defined using a **'define** compiler directive or a **+define+** invocation option.
- **'begin_keywords**, **'end_keywords** — direct compilers to use the reserved keyword list for a specific version of Verilog or SystemVerilog. These directives are discussed in section 2.2.4 (page 46).
- **'timescale** — a legacy Verilog directive to specify time units and precision. This directive was made obsolete by the SystemVerilog **timeunit** and **timeprecision** keywords, as discussed in section 1.5.3 (page 23) in Chapter 1.

SystemVerilog compliant simulation compilers implement all standard compiler directives. Some compiler directives, however, have no meaning for synthesis, such as **'timescale**. These directives are either ignored or not permitted.

2.3 Modules

The primary modeling block in SystemVerilog is the **module**. A module is a container that holds information about a model.

Figure 2-1 shows the basic contents of a module.

Figure 2-1: SystemVerilog module contents

Module name. A module is enclosed between the keywords `module` and `endmodule`. Each module has a name, which is a user-defined identifier that must adhere to the naming rules defined in section 2.2.5 (page 49). Optionally, the same name can be specified after the `endmodule` keyword, separated by a colon. The ending name must be identical to the module name. Specifying an ending name can help with code documentation and maintenance in large, complex models, where there might be many lines of code between the `module` and `endmodule` keywords.

Parameter list. Following the name of the module is an optional *parameter list*, enclosed between the tokens `#(` and `)`. Parameters are used to make modules configurable. The declaration and use of parameters is discussed in Chapter 3, section 3.8 (page 93).

Port declarations. Modules can have any number of *ports*, including none. A port is used to pass data into or out of a module. Ports have a direction, type, data type, size, and name. The direction is declared with the keywords `input`, `output`, or `inout` (bidirectional). SystemVerilog provides an extensive set of built-in types and data types, as well as user-defined types, which can be used for a port type and data type. The various types and data types that are used in synthesizable RTL modeling are discussed in detail in Chapter 3. Syntactically, the size of a port can range from 1-bit wide to 2^{16} (65536) bits wide. In practice, engineers must consider the limitations of the ASIC or FPGA technology that will be used to implement the design. For example, some devices might be able to handle a 64-bit wide data bus, while other devices might only support a maximum of a 32-bit wide data bus.

Internal declarations. Modules might require additional internal data, in addition to the data that comes through its ports. These internal signals also have a type, size and name. The declaration of these signals is also discussed in Chapter 3.

Functionality. At the heart of each module is the functional description of the silicon the module represents. This functionality can be described at a very detailed level, or a very abstract level. The concept of model abstraction was introduced in section 1.2 (page 6) of Chapter 1. This book is focused on representing module functionality at the synthesizable RTL abstraction, and is the topic of subsequent chapters.

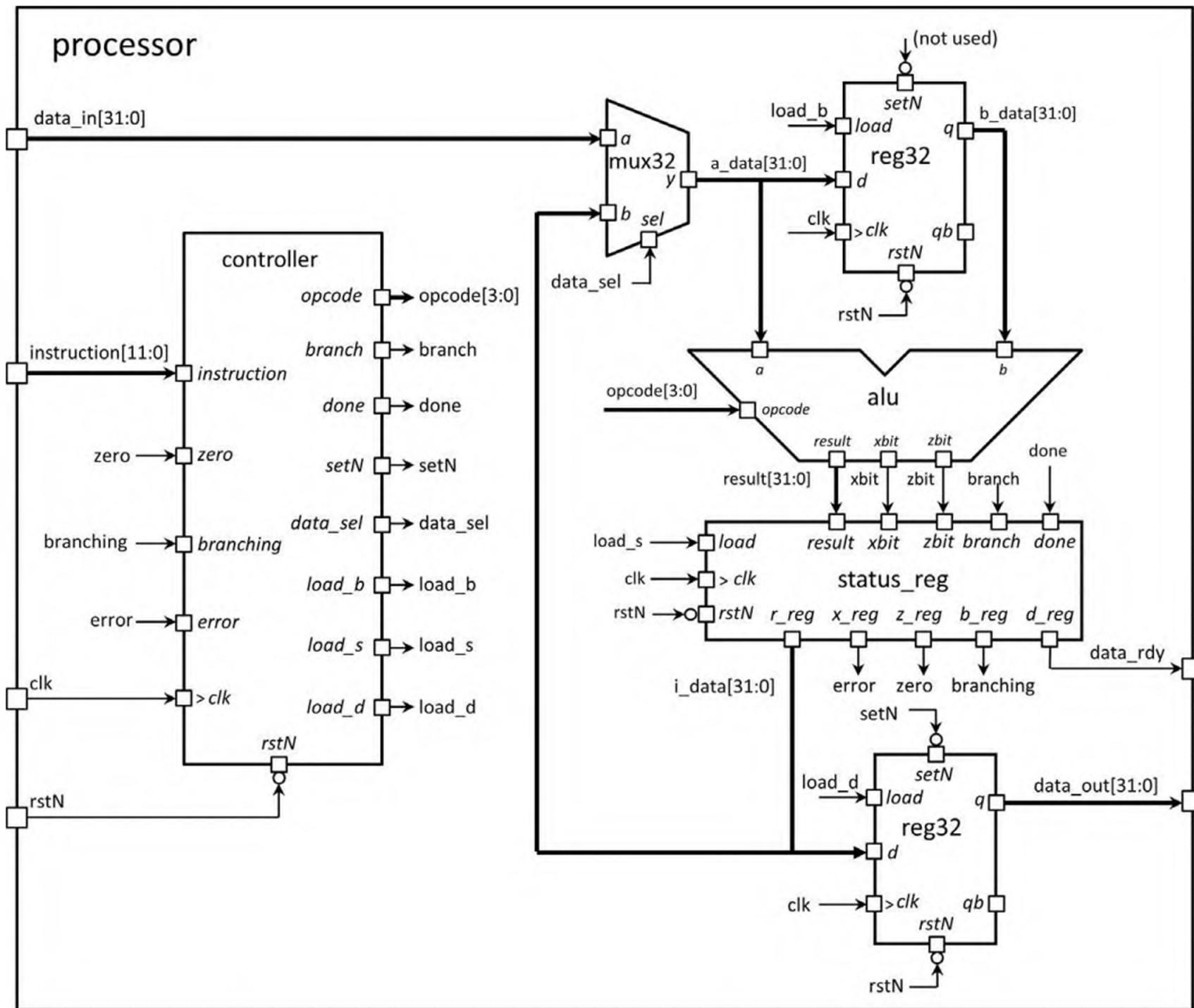
Timing. RTL modeling abstracts away the timing details of silicon. Synthesizable RTL dictates that timing is on clock cycle boundaries. RTL models are often referred to as zero-delay models, because there is no timing detail within a clock cycle.

2.4 Modules instances and hierarchy

Complex designs are partitioned into smaller blocks that are connected together. Each sub block is represented as a module.

Figure 2-2 illustrates a simple design where the processor module has been partitioned into the sub blocks of a controller module, a 32-bit wide mux32 module, an alu module, a status_reg module, and two 32-bit wide reg32 modules.

Figure 2-2: Design partitioning using sub blocks



Netlists and module instances. The SystemVerilog code for the processor module illustrated in Figure 2-2 is a netlist that contains instances of the controller, mux32, alu, status_reg and two reg32 modules. A *netlist* is a list of one or more module instances, and the nets (wires) that connect the instances together. A *module instance* is a reference to the name of a module. The partial code for this netlist is:

```
module processor ( /* port declarations */ );

    ... // internal net declarations

    controller cntlr ( /* port connections */ );
    mux32      mux   ( /* port connections */ );
    alu        alu   ( /* port connections */ );
    status_reg s_reg ( /* port connections */ );
    reg32      b_reg ( /* port connections */ );
    reg32      d_reg ( /* port connections */ );
endmodule: processor
```

The syntax of a module instance is:

```
module_name #(parameter_values) instance_name (connections_to_ports);
```

It is optional to define parameter values. Using parameters to make modules configurable is discussed in more detail in Chapter 3.

The module instance name is required, and must be a unique identifier within the context of the module containing the netlist. The instance name allows the same module name to be instantiated multiple times, and makes each instance unique.

SystemVerilog provides two ways to define the connections to the module instance ports: by *port order*, or by *port name*.

2.4.1 Port order connections

Port order connections connect local net names to the ports of the module instance using the order in which the ports in the module are defined. For example, if the port definition of the reg32 module in Figure 2-2 is:

```
module reg32
  (input logic          load,
   input logic [31:0] d,
   input logic          clk,
   input logic          setN,
   input logic          rstN,
   output logic [31:0] q,
   output logic [31:0] qb
);
```

Then the instances of the two reg32 models in module processor would be:

```
reg32 b_reg (load_b, a_data, clk, , rstN, b_data, );
reg32 d_reg (load_d, i_data, clk, setN, rstN, data_out);
```

NOTE

Port order connections are error prone. A simple coding mistake of listing a connection in the wrong order can result in design bugs that are difficult to debug, resulting in lost engineering time. Port order connections also make it difficult to see which signals are connected to which ports.

Port order connections are not used in this book, and are only introduced here to contrast port order connections to named port connections.

2.4.2 Named port connections

Best Practice Guideline 2-3

Use named port connections for all module instances. Do not use port order connections.

Using named port connections can help prevent accidental connection errors, as well as making code more self-documenting. Named port connections associate a port name with a local signal name connected to that port. There are three forms of named port connections:

- *Explicit named connection* — shown in this section of the book.
- *Dot-name connection shortcut* — shown in section 2.4.3 (page 57).
- *Dot-star connection shortcut* — shown in section 2.4.4 (page 58).

In an explicit named connection, the name of the port is preceded by a period, and the local signal name is enclosed in parentheses. For example:

```
reg32 b_reg (.load(load_b), .d(a_data), .clk(clk),
              .setN(), .rstN(rstN), .q(b_data), .qb() );
reg32 d_reg (.clk(clk), .load(load_d), .setN(setN),
              .rstN(rstN), .d(i_data), .q(data_out) );
```

Some important considerations to note with named port order connections are:

- Port connections can be listed in any order. Instance `d_reg`, above, does not list the port connections in the same order as the port definitions within module `reg32`. (shown previously, in section 2.4.1, page 55).

- Unused ports can be explicitly listed, but with no local signal name in the parentheses, as shown for the connection to the `qb` port in instance `b_reg`, or unused ports can be left out of the connection list, as shown in instance `d_reg`.
- The code is self-documenting. It is visually apparent which nets are connected to which ports of the `reg32` module, without having the file containing the source code for `reg32` open or available.

2.4.3 The dot-name inferred named port connection shortcut

Named port connections have many advantages over port order connections. There is, however, one disadvantage. Named port connections are verbose, and can require considerable replication of names. Observe in the following module instance, that every name is duplicated. The first occurrence of the name, preceded by a period, is the name of the port within the `controller` module. The second occurrence, within the parentheses, is the name of the net to be connected to that port.

```
controller cntlr (.instruction(instruction),
                  .zero(zero),
                  .branching(branching),
                  .error(error),
                  .clk(clk),
                  .rstN(rstN),
                  .opcode(opcode),
                  .branch(branch),
                  .done(done),
                  .setN(setN),
                  .data_sel(data_sel),
                  .load_b(load_b),
                  .load_s(load_s),
                  .load_d(load_d)
);
```

SystemVerilog adds a *dot-name* shortcut to the original Verilog language. This shortcut has all the advantages of named port connections, but eliminates the need to type the same name twice to connect a net to a port. Only the port name needs to be specified. SystemVerilog infers that a net or variable of the same name is connected to the port. This means the verbose Verilog style of a connection such as `.clk(clk)` named port connections can be reduced to simply `.clk`. For example:

```
controller cntlr (.instruction, .zero, .branching, .error,
                  .clk, .rstN, .opcode, .branch, .done,
                  .setN, .data_sel,
                  .load_b, .load_s, .load_d );
```

The full explicit connection syntax and the dot-name shortcut can be mixed. If the name of a net does not match the port to which it is to be connected, the explicit named port connection is used to explicitly connect a net with a different name to the port. Using the dot-name connection style has all the advantages of named port con-

nections over port order connections, and adds the advantage of a netlist that is more concise, easier to read, and easier to maintain.

Dot-name shortcut rules. The dot-name named connection shortcut requires the following condition be met in order to infer a connection between a named port and a net or variable:

- A net or variable with a name that exactly matches the port name must be declared prior to the module instance.
- The net or variable vector size must exactly match the port vector size.
- The data types on each side of the port must be compatible. Incompatible types are defined in the IEEE SystemVerilog standard. For example, a `tri1` pull-up net connected to a `tri0` pull-down net through a module port is not compatible. Such a connection will not be inferred by the dot-name shortcut.

These restrictions reduce the risk of unintentional connections being inferred by the dot-name shortcut.

The dot-name inferred connection shortcut also resolves a hazard that exists with the port order and named port connections — the shortcut will not infer an implicit net.

When port order or named port connections are used, SystemVerilog will infer a net declaration for any undeclared net names used in a netlist. Using inferred nets can be convenient, in that it can save having to explicitly declare internal interconnecting nets. However, inferred nets can also result in design bugs. An incorrectly typed name in a netlist will not be an error. Instead, a net will be inferred for the mistyped name, which will not be connected to other module instances in the netlist. The netlist will compile and simulate or synthesize, but not function correctly. (Inferred nets are discussed in Chapter 3, section 3.5.3, page 80.)

The dot-name shortcut will not infer an implicit net, and therefore helps to avoid the hazards associated with implicit nets.

2.4.4 *The dot-star inferred named port connection shortcut*

SystemVerilog has a second convenient shortcut for coding netlists with named port connections, referred to as the *dot-star* shortcut.

The dot-star inferred named port connection syntax is represented with a special token, `.*`. Dot-star is a wildcard that indicates that all ports and signals of the same name should automatically be connected together for that module instance. All nets connected to the `controller` module from Figure 2-2 (page 54) have the same name as the port. The dot-star connection shortcut can infer all connections to the controller instance as follows:

```
controller cntlr (.*) ; // infer connections to all ports
```

As with the dot-name shortcut, for a connection to be inferred, all nets must be explicitly declared, the name and vector size must match exactly, and the types connected together must be compatible. Any connections that cannot be inferred by dot-star must be explicitly connected together, using the full named port connection syntax, as shown in the following code snippet.

```
alu alu (.a(a_data),  
         .b(b_data),  
         .*           // infer all other connections  
);
```

With dot-star, the only connections that need to be listed are the ones where the port name and the connecting net name are not the same. This can be an advantage versus the dot-name shortcut because it makes these differences more obvious. A disadvantage, however, is that it is not easy to see what connections have been inferred by dot-star. This can make code maintenance and debugging more difficult.

Another advantage of dot-star over dot-name is that the dot-star will not allow a port to be inadvertently left unconnected. Unlike explicit named port connections and the dot-name shortcut, the dot-star shortcut will not infer an unconnected port. Ports must be explicitly shown as not having a connection using an empty parentheses, such as `.qb()`.

2.5 Summary

As with all programming languages, SystemVerilog has specific syntax and semantic rules which must be followed when writing SystemVerilog models. This chapter has examined the essential rules for writing RTL models. Important considerations include the proper use of white space and comments, and proper naming conventions.

The SystemVerilog language has been around since 1984, and was originally called Verilog. The language has evolved over this 30-year period. There have been several versions of the Verilog and SystemVerilog standard. Each version of the standard has reserved additional keywords that were not reserved in prior standards. SystemVerilog provides a pair of compiler directives, to help ensure that legacy models will compile correctly with compilers that are based on a later version of the standard. These directives are '**begin_keywords**' and '**end_keywords**'.

Large designs are partitioned into sub blocks, with each block represented as a module. A higher level module is used to instantiate and connect these sub modules. SystemVerilog provides two ways to connect modules: port order connections (which have several hazards) and named port connections. Named port connections are more verbose than port order connections, but can help prevent subtle, difficult to find, connection errors. The dot-name and dot-star named port connection shortcuts help simplify large netlists, while retaining the advantages of named port connections.

Chapter 3

Net and Variable types

Abstract — SystemVerilog has two major groups of data types, *nets* and *variables*. There are a number of predefined types within these two groups, which are used to model both designs and verification testbenches. The major concepts discussed in this chapter are:

- Two-state and four-state values
- Literal values
- Variable types
- Net types
- Arrays of nets and variables
- Assignment rules for nets and variables
- Port types
- Parameter constants

3.1 Four-state data values

For RTL modeling, SystemVerilog uses a four-value representation of the values that can occur in silicon.

- **0** represents an abstract digital low, with no voltage or current associated with it.
- **1** represents an abstract digital high, with no voltage or current associated with it.
- **Z** represents an abstract digital high-impedance. In a multi-driver circuit, a value of 0 or 1 will override a Z. Some programming operators and programming statements treat Z values as don't care values (see Chapter 5, section 5.9, page 171, and Chapter 6 section 6.2.2, page 223).
- **X** represents either an uninitialized value, an uncertain value, or a conflict of values in a multi-driver circuit. In certain RTL model contexts, synthesis compilers treat an X value as a don't-care value (see Chapter 9, section 9.3.6, page 345).

The values of 0, 1 and Z are an abstraction of values that can exist in actual silicon. The value of X is not an actual silicon value. Simulators use X to indicate a degree of

uncertainty in how physical silicon would behave under specific circumstances, such as when simulation cannot predict whether an actual silicon value would be a 0 or 1 (or Z for a tri-state device). For synthesis, an X value also provides design engineers a way to specify “don’t-care” conditions, where the engineer is not concerned about whether actual silicon will have a 0 or a 1 value for a specific condition.

3.2 Literal values (numbers)

A *literal value* is an integer or real (floating-point) number. SystemVerilog provides several ways to specify literal values. There are also several semantic rules for literal values that are important to understand when writing RTL models.

3.2.1 *Literal integer values*

A literal integer value is a whole number, with no fractional decimal places. (The IEEE 1800 SystemVerilog standard uses the term “integer literal” instead of “literal integer”.) Literal integers can be specified in a variety of ways:

- Simple decimal integer values
- Binary, octal, decimal, or hexadecimal integer values
- Sized literal integer values
- Signed or unsigned literal integer values

Both simulation and synthesis tools need to know or assume specific characteristics about literal integer values. These characteristics are:

- The bit width (vector size) of the value
- The signedness of the value (signed or unsigned)
- The base of the value (also known as the radix)
- 2-state or 4-state value

These characteristics affect both operations and assignments of the value.

Simple decimal literal integers. A literal integer value can be specified as a simple number, such as the number 9, as shown in the following code snippet:

```
result = d + 9;
```

A simple literal number is treated by simulation and synthesis as:

- A 32-bit wide value
- A signed value
- A decimal value
- A 2-state value (no bits can be Z or X)

These characteristics, along with the characteristics of `d`, will affect how the addition is performed, and how the assignment to `result` is performed. These effects are discussed in Chapter 5 on SystemVerilog operators and operations.

Binary, octal, decimal, and hexadecimal literal integers. A specific base of binary, octal, decimal or hexadecimal can be specified for literal integer values. The base is specified using an apostrophe ('') (sometimes referred to as a “tick”) followed by one of the letters: `b` or `B` for binary, `o` or `O` for octal, `d` or `D` for decimal, `h` or `H` for hexadecimal. Some examples are:

```
result = 'd9 + 'h2F + 'b1010;
```

An explicit base literal number with no size specified is treated by simulation and synthesis as:

- A 32-bit wide value
- An unsigned value (note the difference from a simple literal integer, which is signed)
- A value in the base specified
- A 4-state value (any or all bits can be X or Z)

Each bit of a binary value can be 0, 1, X or Z. Each 3-bit group of an octal value can be 0 through 7, X or Z. Each digit of a decimal value can be 0 through 9, X or Z. Each 4-bit group of a hexadecimal value can be 0 through 9, A through F, X or Z.

Signed literal integers. By default, a literal value with a base specified is treated as an unsigned value in operations and assignments. This default can be overridden by adding the letter `s` or `S` after the apostrophe and before the base specifier.

```
result = 'sd9 + 'sh2F + 'sb1010;
```

Signed values are treated differently than unsigned values in certain operations and in assignment statements. The effects of signed and unsigned values are discussed in Chapter 5 on operators and operations.

Sized literal integers . By default, a simple literal number and a literal number with a base specified are treated as 32-bit values in operations, programming statements, and assignment statements. This default does not accurately represent hardware models that use other vector sizes.

A value with a specific base can also have a specific bit-width specified. The number of bits used to represent the value is specified before the apostrophe, signedness and base specification.

```
result = 16'd9 + 8'h2F + 4'b1010;
```

NOTE

Synthesis compilers and lint checkers may generate warning messages when the size of a literal value is not the same as the variable on the left-hand side of an assignment statement. These size mismatch warning messages can hide other messages that require attention. Using explicitly sized literal values will prevent unintentional size mismatch warnings.

Best Practice Guideline 3-1

Only use binary and hexadecimal literal integers in RTL models. These number bases have an intuitive meaning in digital logic.

The use of octal values has been obsolete for decades. Literal decimal values can be easily confused with other numbers. The old engineering joke applies here... “*There are 10 types of people in the world, those that understand binary, and those that don’t.*”

3.2.1.1 Mismatched size and value rules

It is legal to specify a literal integer a bit-width that is different than the number of bits required to represent the value. For example:

```
4'hFACE // 4-bit width, 16-bit unsigned value  
16'sh8 // 16-bit size, 4 bit signed value, MSB of value set  
32'bZ // 32-bit width, 1-bit unsigned value
```

SystemVerilog always adjusts the value to match the specified size. The rules are:

- When the size is fewer bits than the value, the left-most bits of value are truncated.
- When the size is more bits than the value, the value is left-extended. The additional bits are filled using the following rules:
 - If the left-most bit of the value is 0 or 1, the additional upper bits are filled with 0.
 - If the left-most bit of the value is Z, the additional upper bits are filled with Z.
 - If the left-most bit of the value is X, the additional upper bits are filled with X.

Note that the value is not sign extended, even if the literal integer is specified as a signed integer. Sign extension occurs when a signed literal value is used in operations and assignment statements, which are discussed in Chapter 5.

The value adjustments for the preceding code snippet are:

```
4'hFACE // truncates to 4'hE  
16'sh8 // extends to 16'sh0008  
32'bZ // extends to 32'bZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
```

Best Practice Guideline 3-2

Use a lint check program (also known as a modeling rule checker) in conjunction with simulation, and before synthesizing the RTL model.

Simulators might report a non-fatal warning message when a truncation occurs, but are not required to report a warning. Simulators silently will extend a literal value to match a size, without generating any warnings. There is a risk of verifying design functionality in simulation without realizing there was a size/value mismatch. Using a lint program will reveal any mismatches in literal values.

3.2.1.2 Additional literal value rules

The question mark (?) can be used in place of Z to represent hi-impedance. In most contexts, the letter Z is a more intuitive way to represent hi-impedance. However, there are some operators and programming statements that use hi-impedance values to indicate a don't-care condition. Using a question mark for hi-impedance can be more intuitive for these don't-care conditions.

The underscore character (`_`) can be used anywhere in a literal value. The underscore is ignored by simulation, synthesis compilers, and other tools that can parse SystemVerilog code. Adding an underscore to a number can help make long numbers more readable, especially binary values. Underscores can also be used to show sub-fields within a value.

```
16'b0000_0110_1100_0001 // show 4-bit nibbles for readability  
20'h2 FACE // 20-bit value with 4-bit opcode and 16-bit data
```

3.2.2 Vector fill literal values

SystemVerilog provides a special form of unsized literal integers that sets all bits of a vector of any size to 0, 1, X or Z. The vector size of the literal value is automatically determined, based on its context.

- '`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

An example of using the vector fill literal integers is:

```
always_ff @(posedge clk)
  if (!setN) // active low set
    q <= '1; // set all bits of q to 1, regardless of size
  else
    q <= d;
```

Vector fill literal integers are an important construct for modeling scalable designs that can have different vector widths for different design configurations. Section 3.8 (page 93) in this chapter discusses modeling configurable vector sizes.

These vector fill literal integers were not part of traditional Verilog. They were added as part of the SystemVerilog extensions to the original Verilog language.

3.2.3 Floating-point literal values (*real numbers*)

SystemVerilog refers to floating-point values as *real numbers*. Real numbers are represented using 64-bit double-precision floating point. Literal floating-point values are specified by using a decimal point in a literal number. A value must be specified on both sides of the decimal point.

3.1567
5.0
0.5

NOTE

RTL synthesis compilers typically do not support real (floating-point) expressions. High-level Synthesis (HLS) tools can be used for complex arithmetic design. Floating point and fixed point design is outside the scope of this book on RTL modeling.

3.3 Types and data types

SystemVerilog provides two general groups of data types, *nets* and *variables*. Nets and variables have both a *type* and a *data type*. *Type* indicates that the signal is a net or variable. *Data type* indicates the value system of the net or variable, which is either 2-state or 4-state. For simplicity, this book uses the term *data type* to mean both the *type* and *data type* of a signal.

Data types are used by software tools, such as simulators and synthesis compilers, to determine how to store data and process changes on that data. Data types affect operations, and are used in RTL modeling to indicate the silicon behavior desired. For example, data types are used to determine if an adder should be integer based or floating-point based, and whether signed or unsigned arithmetic should be performed.

3.3.1 Net types and variable types

Variables are used as temporary storage for programming. This temporary storage is for simulation. Actual silicon often does not need the same temporary storage, depending on the programming context in which the variable is used. SystemVerilog has several variable types, which are discussed in section 3.4.

Nets are used to connect design blocks together. A net transfers data values from a source, referred to as a driver, to a destination or receiver. SystemVerilog provides several net types, which are discussed in more detail in section 3.5 (page 76).

3.3.2 Two-state and four-state data types (bit and logic)

SystemVerilog variables can be either 2-state data types or 4-state data types. With 2-state, each bit of a variable can have the value of 0 or 1. With 4-state, each bit of a variable can have the value of 0, 1, Z or X. SystemVerilog nets can only be 4-state data types. The keyword **bit** defines that a variable is a 2-state data type. The keyword **logic** defines that variable or net is a 4-state data type.

3.4 Variable types

Variables are required on the left-hand side of procedural block assignments. The signals `sum` and `out` in the following code examples must be variables.

```
always_comb begin           // combinational logic
    sum = a + b;
end

always_ff @(posedge clk)    // sequential logic
    if (!rstN) out <= '0;   // active-low reset
    else         out <= sum;
```

Variables provide temporary storage for simulation. The **always_comb** procedure in the preceding code snippet will execute the assignment statement `sum = a + b;` every time `a` or `b` changes value. The value of `sum` must be stored by simulation until the next time either `a` or `b` changes. Similarly, the **always_ff** procedure will execute the if-else decision statement on every positive edge of `clock`. The value of `out` must be stored by simulation between the clock cycles.

The temporary storage required by simulation does not necessarily mean that actual silicon will require storage. The **always_comb** procedure in the preceding code snippet will be implemented as combinational logic in silicon. As such, the value of `sum` will continually reflect the output of an adder, and not require any type of storage in hardware. On the other hand, the **always_ff** procedure will be implemented in silicon as a flip-flop, which is a hardware storage device.

3.4.1 Synthesizable variable data types

Variables are declared by specifying both a *type* and a *data type*. The *type* is the keyword **var**, which can be specified explicitly or implicitly inferred.

The **var** keyword is seldom used in actual SystemVerilog code. Instead, the **var** type is inferred from other keywords and context.

SystemVerilog has keywords for several built-in variable *data types*. These keywords infer either **var logic** (4-state) or **var bit** (2-state) variable type. Several variable data types represent silicon behavior, and are synthesizable. These synthesizable data types are listed in Table 3-1.

Table 3-1: Synthesizable variable data types

Type	Representation
reg	An obsolete general purpose 4-state variable of a user-defined vector size; equivalent to var logic
logic	Usually infers a general purpose var logic 4-state variable of a user-defined vector size, except on module input/inout ports, where wire logic is inferred
integer	A 32-bit 4-state variable; equivalent to var logic [31:0]
bit	A general purpose 2-state var variable with a user-defined vector size; defaults to a 1-bit size if no size is specified
int	A 32-bit 2-state variable; equivalent to var bit [31:0]; synthesis compilers treat int as the 4-state integer type
byte	An 8-bit 2-state variable; equivalent to var bit [7:0]
shortint	A 16-bit 2-state variable; equivalent to var bit [15:0]
longint	A 64-bit 2-state variable; equivalent to var bit [63:0]

Best Practice Guideline 3-3

Use the 4-state **logic** data type to infer variables in RTL models. Do not use 2-state types in RTL models. An exception to this guideline is to use the **int** type to declare for-loop iterator variables.

The use of 4-state variables allows simulators to use an X value when there is an ambiguity as to what a value would be in actual hardware.

The context dependent logic data type. In almost all contexts, the **logic** data type infers a 4-state variable the same as **reg**. The keyword **logic** is not actually a variable type, it is a data type that indicates a net or variable can have 4-state values. However, a variable is inferred when the **logic** keyword is used by itself, or in conjunction with the declaration of a module **output** port. There is an exception where

`logic` does not infer a variable. A net type will be inferred, when `logic` is used in conjunction with the declaration of a module `input` or `inout` port.

The obsolete reg data type . The `reg` data type is an obsolete data type left over from the original Verilog language. The `logic` type should be used instead of `reg`. The original Verilog language used the `reg` data type as a general purpose variable. Unfortunately, the use of keyword `reg` is a misnomer that might seem to be short for “*register*”, a hardware device built with flip-flops. In actuality, there is no correlation between using a `reg` variable and the hardware that is inferred. It is the context in which a variable is used that determines if the hardware represented is combinational logic or sequential flip-flop logic. Using `logic` instead of `reg` can help prevent this misconception that a hardware register will be inferred.

An X value can indicate a design problem. When an X value occurs during simulation, it is often an indication that there is a design problem. Some types of design bugs that will result in an X value include:

- Registers that were not reset or otherwise initialized.
- Circuitry that did not correctly retain state during low power mode.
- Unconnected module input ports (unconnected input ports float at high-impedance, which often results in an X value as the high-impedance value propagates to other logic).
- Multi-driver conflicts (bus contention).
- Operations with an unknown result.
- Out-of-range bit-selects and array indices.
- Setup or hold timing violations.

Avoid 2-state data types in RTL models. The `bit`, `byte`, `shortint`, `int` and `longint` data types only store 2-state values. These types cannot represent a high-impedance (Z value), and cannot use an X value to represent uninitialized or unknown simulation conditions. An X value that indicates potential design bugs, such as those in the list above, will not occur when 2-state data types are used. Since 2-state data types can only have a 0 or 1 value, a design with errors can appear to be functioning correctly during simulation. This is not good! An appropriate place to use 2-state variables is for randomized stimulus in verification testbenches.

Non synthesizable variable types. SystemVerilog has several variable types that are intended primarily for verification, and are not generally supported by RTL synthesis compilers. Table 3-2 lists these additional variable types. These data types are not used in any examples in this book that are intended to be synthesized

Table 3-2: Non synthesizable variable data types

Type	Representation
real	A double precision floating-point variable
shortreal	A single precision floating-point variable
time	A 64-bit unsigned 4-state variable with timeunit and timeprecision attributes
realtime	A double precision floating-point variable, identical to real
string	A dynamically sized array of byte types that can store a string of 8-bit ASCII characters
event	A pointer variable that stores a handle to a simulation synchronization object
class handle	A pointer variable that stores a handle to a class object (the declaration type is the name of a class, not the keyword class)
chandle	A pointer variable that stores pointers passed into simulation from the SystemVerilog Direct Programming Interface
virtual interface	A pointer variable that stores a handle to an interface port (the interface keyword is optional)

3.4.2 Variable declaration rules

Variables are declared by specifying both a *type* and a *data type*. The *type* is the keyword **var**, which can be explicitly specified or implicitly inferred.

NOTE

The **var** keyword is seldom used in actual SystemVerilog code. Instead, the **var** type is inferred from other keywords and context.

Some example variable declarations are:

```
logic v1; // infers var logic (a 1-bit 4-state variable)
bit v2; // infers var bit (1-bit 2-state variable)
integer v3; // infers var integer (32-bit 4-state variable)
int v4; // infers var int (a 32-bit 2-state variable)
```

The only place where the **var** keyword is required is when declaring an **input** or **inout** port as a 4-state variable. These port directions will default to a net type if not explicitly declared as a variable. This is an appropriate default. It is very seldom that an input port needs to be a variable. Port declarations are discussed in more detail in section 3.6.1 (page 84).

Scalar variables. A *scalar* variable is a 1-bit variable. The **reg**, **logic** and **bit** data types default to 1-bit scalars.

```
logic v5;           // a 1-bit scalar variable
logic v6, v7, v8;  // a list of three scalar variables
```

Vector variables (packed arrays). A *vector* is an array of consecutive bits. The IEEE SystemVerilog standard refers to vectors as *packed arrays*. The **reg**, **logic** and **bit** data types can represent a vector of any size: The size of the vectors is declared by specifying a range of bits in square brackets ([]), followed by the vector name. The range is declared as [*most-significant_bit_number* : *least-significant_bit_number*]. The most-significant bit (*MSB*) and least-significant bit (*LSB*) can be any number, and the LSB can be smaller or larger than the MSB. A vector range where the LSB is the smaller number is referred to as *little endian*. A vector range where the LSB is the larger number is referred to as *big endian*.

```
logic [31:0] v9;    // 32-bit vector, little endian
logic [1:32] v10;   // 32-bit vector, big endian
```

The most common convention in RTL modeling is little endian, and with 0 as the LSB of the vector range. Variable v13 above illustrates this convention. All examples in this book use a little endian convention.

The **byte**, **shortint**, **int**, **longint** and **integer** data types have a predefined vector size, as described in Table 3-1 (page 68). The predefined ranges are little endian, with the LSB numbered as bit 0.

Signed and unsigned variables. The value stored in a vector variable can be treated as either *signed* or *unsigned* in operations. An unsigned variable only stores positive values. A signed variable can store positive and negative values. SystemVerilog uses two's-complement to represent negative values. The most significant bit of a signed variable is the sign bit. When the sign bit is set, the remaining bits of the vector represent a negative value in two's-complement form.

By default, the **reg**, **logic**, **bit** and **time** data types are unsigned variables, and the **byte**, **shortint**, **int**, **integer**, and **longint** data types are signed variables. This default can be changed by explicitly declaring a variable as **signed** or **unsigned**.

```
logic signed [23:0] v11; // 24-bit signed 4-state variable
int unsigned v12;        // 32-bit unsigned 2-state variable
```

Constant bit selects and part selects. A vector can be referenced in its entirety, or in part. A *bit select* references a single bit of a vector. A bit select is performed using the name of the vector, followed by a bit number in square brackets ([]). A *part select* references multiple contiguous bits of a vector. A part select is performed using the name of the vector, followed by a range of bit numbers in square brackets ([]).

```

logic [31:0] data;           // 32-bit vector variable
logic [31:0] dbus;          // 32-bit vector variable
logic [ 7:0] byte2;         // 8-bit vector variable
logic      msb;            // 1-bit scalar variable

assign dbus = data;        // reference entire data variable
assign msb = data[31];      // bit select of data, bit 31
assign byte2 = data[23:16]; // part select of data bits 23
                           // down to 16

```

Part selects must meet two rules: the range of bits must be contiguous, and the endian of the part select must be the same endian as the vector declaration. The result of a bit select or part select is always unsigned, even if the full variable is signed.

Variable bit and part selects. The bit select for `msb` in the preceding code snippet used a hard-coded bit number. This is referred to as a *fixed bit select*. The index number of a bit select can also be a variable. For example:

```

logic [31:0] data;           // 32-bit vector variable
logic      bit_out;         // 1-bit scalar variable

always @(posedge shift_clk)
  if (shift_enable) begin
    for (int i=0; i<=31; i++) begin
      @(posedge shift_clk) bit_out <= data[i];
    end
  end

```

The starting point of a part select can also be variable. The part select can either increment or decrement from the variable starting point. The total number of bits selected is a fixed range. The form of a variable part select is:

[*starting_point_variable* +: *part_select_size*]
[*starting_point_variable* -: *part_select_size*]

The `+`: token indicates to increment from the starting point bit number. The `-:` token indicates to decrement from the starting point bit number.

The following example uses a variable part select to iterate through the bytes of a 32-bit vector.

```

logic [31:0] data;           // 32-bit vector variable
logic [ 7:0] byte_out;       // 8-bit vector variable

always @(posedge shift_clk)
  if (shift_enable) begin
    for (int i=0; i<=31; i=i+8) begin
      @(posedge shift_clk) byte_out <= data[i+:8];
    end
  end

```

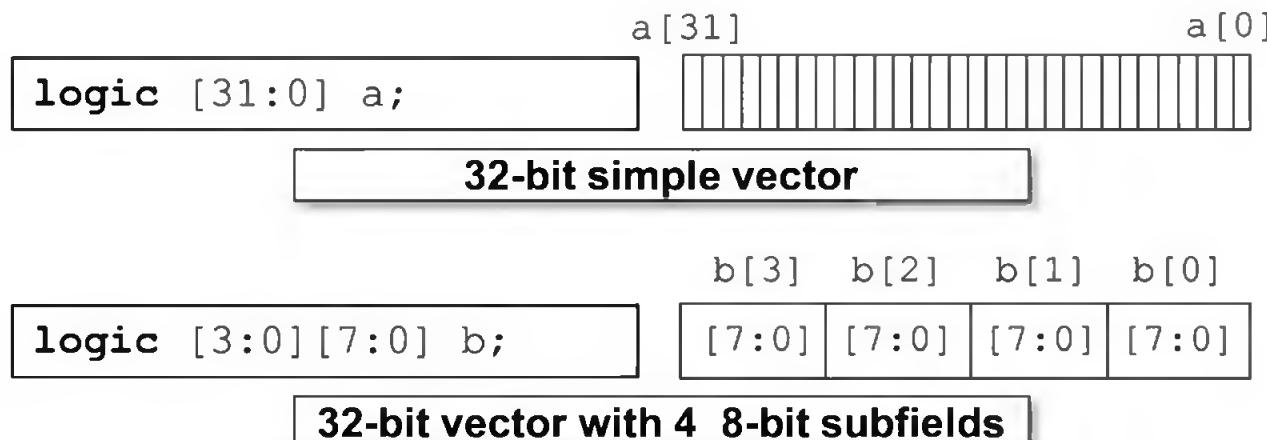
Variable bit and part selects are synthesizable. However, the preceding code snippets illustrating variable bit and part selects do not meet other RTL coding restrictions required by some synthesis compilers. Chapter 8 discusses synthesis requirements for sequential logic in more detail.

Vectors with subfields. Vectors can be declared with subfields by using two or more sets of square brackets to define the vector range. The following code snippet shows the difference between a simple 32-bit vector and a 32-bit vector with subfields:

```
logic [31:0] a;           // 32-bit simple vector
logic [3:0][7:0] b;       // 32-bit vector, subdivided into
                         // 4 8-bit subfields
```

Figure 3-1 illustrates the difference in these two declarations.

Figure 3-1: Vectors with subfields



For the declaration:

```
logic [3:0][7:0] b;       // 32-bit vector, subdivided into
                         // 4 8-bit subfields
```

The first range, [3:0], defines how many subfields there are in the vector. In this example, there are four subfields, indexed as b[0], b[1], b[2], and b[3]. The second range, [7:0], defines the size of each subfield, which is 8 bits in this example. Figure 3-1 illustrates the layout of a simple 32 bit vector and a 32-bit vector subdivided into 4 bytes.

Subfields of a subdivided vector can be referenced using a single index, instead of a part select. The following code snippet illustrates looping through the bytes of vector b, and is more straightforward, because each byte is a subfield of the vector.

```
always @(posedge shift_clk)
  if (shift_enable) begin
    for (int i=0; i<=3; i++) begin
      @(posedge shift_clk) byte_out <= b[i];
    end
  end
```

A bit select of a subdivided vector requires multiple indices. Selecting bit 7 of the third byte of vector b is coded as: b[3][7].

Best Practice Guideline 3-4

Use a simple vector declaration when a design mostly selects either the entire vector or individual bits of the vector. Use a vector with subfields when a design frequently selects parts of a vector, and those parts fall on known boundaries, such as byte or word boundaries.

Selecting a subfield of a vector instead of using fixed or variable part selects of a simple vector makes code easier to write and maintain.

3.4.3 Variable assignment rules

Variables can be assigned a value in several ways:

- As the left-hand side of a procedural assignment statement (within **always**, **always_comb**, **always_latch**, **always_ff** or **initial** procedural blocks, or in a task or function).
- As the left-hand side of a continuous assignment statement (using the **assign** statement).
- As the result of an assignment operator, such as the **++** increment operator.
- As an input to a module, task or function.
- As a connection to an output port of a module instance, task instance, function instance, or primitive instance.

Variables can only be assigned by a single source. For example, if a variable is assigned a value from an **assign** continuous assignment statement, then it is illegal to also assign the variable a value in a procedural block or from a module input port. However, any number of procedural assignments to the same variable is considered a single source. This rule is important in order for code such as the following to work:

```
logic [15:0] q; // 16-bit 4-state unsigned variable
always_ff @(posedge clk)
  if (!rstN) q <= '0; // a procedural assignment to q
  else       q <= d; // another procedural assignment to q
```

The semantic restriction of a single source for variable assignments is important in RTL modeling. The restriction helps ensure that abstract RTL simulation behavior and post-synthesis implementation behavior are the same.

The **always_ff**, **always_comb** and **always_latch** procedural blocks further restrict procedural assignments to a variable to only be within one procedure, which enforces a requirement by synthesis compilers. Multiple assignments a variable in the same procedure are treated as a single driver.

3.4.4 Uninitialized variables

A variable is *uninitialized* until a value has been assigned to the variable. The uninitialized value of 4-state variables is 'X (all bits set to X). The uninitialized value of 2-state variables is '0 (all bits set to 0).

In the following example, variable q is uninitialized until the first positive edge of clk occurs. As a 4-state logic type, q will have an X value until that first clock, at which point q will either be assigned the value of 0 or the value of d. This X value could indicate a design problem if a positive edge of clk did not occur, perhaps due to clock gating or some other circumstance.

```
logic [15:0] q; // uninitialized 4-state variable  
always_ff @(posedge clk)  
  if (!rstN) q <= '0; // active-low synchronous reset  
  else      q <= d; // clocked assignment to q
```

NOTE

Uninitialized 2-state variables can hide design problems. An uninitialized 2-state variable has the value of 0, which can appear to be a legitimate reset value. This can potentially hide problems with reset logic in a design.

3.4.5 In-line variable initialization

SystemVerilog allows variables to be initialized as part of declaring the variable, referred to as *in-line initialization*. For example:

```
int i = 5;
```

In-line initialization of a variable is only executed one time, at the beginning of simulation.

Some FPGA devices can be programmed so that registers power-up in a known state, without having to be reset. In-line variable initialization can be used to model power-up states of these sequential devices such as a flip-flop.

NOTE

In-line variable initialization is not supported in ASIC technologies, and might be supported by some FPGA technologies.

When targeting a device that does not support programmable power-up states, synthesis compilers will either: (a) not allow in-line initialization, or (b) ignore it. A mismatch in RTL simulation behavior and the synthesized gate-level implementation can occur when in-line initialization is ignored.

Best Practice Guideline 3-5

Only use variable initialization in RTL models that will be implemented as an FPGA, and only to model power-up values of flip-flops.

For ASIC design, reset functionality should be used to initialize variables. Do not use in-line initialization. For FPGA design, only use in-line initialization if it is certain that the RTL model will always be targeted to a device that supports power-up register states. The use of in-line initialization in RTL models effectively locks the model to only be used that type of FPGA device.

Best Practice Guideline 3-6

Only use in-line variable initialization in RTL models. Do not use initial procedures to initialize variables.

Synthesis compilers and target FPGA devices that support in-line variable initialization also allow using **initial** procedures to model the power-up value of flip-flops.

Sequential logic reset and the appropriate use of variable initialization in RTL models is discussed in Chapter 8, section 8.1.5 (page 286).

3.5 Net types

Nets are used to connect design elements together, such as connecting the output port of one module to the input port of another module. Nets differ from variables in three significant ways:

- Nets do not have temporary storage like variables. Instead, nets reflect the current value of the driver(s) of the net. (A capacitive **trireg** net appears to store a value, but is actually representing the behavior of a capacitor driving the net).
- Nets can resolve the resultant value of multiple drivers, where variables can only have a single source (if multiple procedural assignments are made to a variable, the last assignment is the resultant value, rather than resolving the result of all assignments).
- Nets reflect both a driver value (0, 1, Z or X) and a driver strength.

The strength level of a driver is represented in steps from 0 to 7. Each level is represented by a keyword. The default strength level for most modeling constructs is strong, which is a level 6. Strength levels are important for transistor-level modeling, but are not used in RTL modeling. The representation and usage of strengths is outside the scope of this book on RTL modeling.

3.5.1 Synthesizable net types

Nets are declared by specifying both a *type* and a *data type*. The *type* can be any of the keywords listed in Tables 3-3 and 3-4. The *data type* must be the keyword **logic**, which can be specified explicitly or implicitly inferred.

Each SystemVerilog net type has specific semantic rules that affect how multiple drivers are resolved. While all net types represent silicon behavior, not all net types can be represented in standard ASIC and FPGA technologies. Table 3-3 lists the net types supported by ASIC and FPGA synthesis compilers.

Table 3-3: Synthesizable net types

Type	Representation
wire	An interconnecting net that resolves multiple drivers using CMOS behavior
tri	A synonym for wire , and identical in all ways. Can be used to emphasize nets that are expected to have tri-state values
supply0	An interconnecting net that has a constant logic 0, at the supply strength level. Can be used to represent a ground rail (GND, VSS)
supply1	An interconnecting net that has a constant logic 1, at the supply strength level. Can be used to represent a supply rail (VCC, VDD)

Non synthesizable net types. SystemVerilog has several net types that are not universally supported by synthesis compilers, which are listed in Table 3-4 (page 77).

Table 3-4: Generally non-synthesizable net types

Type	Representation
uwire	An interconnecting net that does not permit or resolve multiple drivers
pull0	An interconnecting net that has the behavior of having a pull-down resistor tied to the net
pull1	An interconnecting net that has the behavior of having a pull-up resistor tied to the net
wand	An interconnecting net that resolves multiple drivers by ANDing the driven values
triand	A synonym for wand , and identical in all ways; can be used to emphasize nets that are expected to have tri-state values
wor	An interconnecting net that resolves multiple drivers by ORing the driven values
trior	A synonym for wor , and identical in all ways; can be used to emphasize nets that are expected to have tri-state values
trireg	An interconnecting net with capacitance; if all drivers are at high-impedance, the capacitance reflects the last resolved driven value

NOTE

Some RTL synthesis compilers might support one or more of these net types. A best-practice coding style is to not use these types in order to ensure the RTL model is compatible with any synthesis compiler. If one of these types is used, design engineers should check that all tools used in the project support that type.

Modeling CMOS technology. Most ASIC and FPGA devices are implemented using CMOS technology. The behavior of CMOS interconnections is represented using the `wire` and `tri` net types. The `wire` type is the most commonly used net type, and is the default net type when nets are implicitly inferred (see section 3.5.3, page 80).

Single-driver and multi-driver logic. Most interconnecting nets in ASIC and FPGA designs connect a single driver to one or more receivers. The rare exception is a shared bus, where multiple drivers are connected to one or more receivers. An example is RAM devices that have a bidirectional data bus used to both write values into the RAM and read values from the RAM. ASIC and FPGA devices often have a certain number of bidirectional I/O pads for reading and driving values.

Best Practice Guideline 3-7

Use a `logic` data type to connect design components together whenever the design intent is to have single driver functionality. Use `wire` or `tri` net types only when the design intent is to permit multiple drivers.

Declaring interconnections as `logic` will infer a variable instead of a net type. Variables only allow a single source (driver). If the same signal is inadvertently connected to more than one module output.

Although most interconnecting nets are intended to only have a single driver, the synthesizable net types, such as `wire`, permit more than one driver. Engineers need to be careful to avoid coding errors when using net types. A simple typographical error in a netlist can result in the same net being unintentionally connected to more than one driver. This type of error will not be caught during compilation and elaboration. The error results in a functional bug that must be detected during simulation.

Using variables instead of nets to connect design blocks. SystemVerilog also allows a variable to be used to connect design elements together. Variables do not permit multiple drivers (see section 3.4.3, page 74). An elaboration error will occur if the same variable is accidentally connected to more than one driver.

Declaring input ports as variable type instead of net types. By default, input and inout ports infer to a net type, specifically the **wire** type, unless the **'default_netttype** specifies a different net type (see section 3.5.3, page 80). This inference of a net type can result in hard-to-detect modeling errors where multiple drivers are connected to the same input port (or a value is back driven onto the input port from within a module). These modeling errors are legal in SystemVerilog because net types permit multiple drivers.

Unintentional multiple drivers of an input port can be prevented by explicitly declaring the input port as a **var logic** type. Variables do not permit multiple drivers. Inadvertent multiple drivers will be reported as a coding error when the design modules are compiled and elaborated.

Using uwire to prevent multiple drivers. The **uwire** net type can also be used to prevent inadvertent multiple drivers of an input port. The **uwire** type was added to SystemVerilog as part of the 1364-2005 Verilog standard, specifically to make unintentional multiple drivers be a compile/elaboration error. An input port can be explicitly declared **uwire** type, or the default net type can be changed to **uwire**, as discussed in section 3.5.3 (page 80). The **uwire** type does not permit multiple drivers. Inadvertent multiple drivers will be reported as a coding error when the design modules are compiled and elaborated.

NOTE

At the time this book was written, most synthesis compilers, and some simulators, had not yet added support for the **uwire** type, even though it has been part of the Verilog/SystemVerilog standard since 2005. The examples in this book use the **wire** or **tri** type when a multi-driver net is required.

3.5.2 Net declaration rules

Nets are declared by specifying a net *type* and an optional *data type*. The data type must be the 4-state **logic** data type, or a user-defined type derived from 4-state **logic** data types. If the data type is not explicitly specified, then a **logic** data type is implicitly inferred.

The default size of all net types is scalar (1-bit). Nets can be explicitly declared as vectors of any size using the same syntax as with variables (see section 3.4.2, page 70). Only variable vector declarations can be divided into subfields, however. Net vectors cannot be divided into subfields.

Some example synthesizable net declarations are:

```
wire logic n1;      // 1-bit 4-state net with CMOS resolution
supply1      vcc;    // 1-bit 4-state net with a constant 1
tri [31:0] data;   // 32-bit vector net with CMOS resolution
```

By default, all net types are unsigned. Nets can be explicitly declared as signed or unsigned in the same way as variables. See section 3.4.2 (page 70).

Net bit and part selects. Any specific bit or group of bits can be selected from a net vector using the same syntax as with variable vectors, as described in section 3.4.2 (page 70). Both constant and variable bit and part selects can be performed on nets.

3.5.3 Implicit net declarations

An undeclared signal will infer a net type in several contexts:

- A module **input**, **inout** or **output** port with no type or data type explicitly declared, or inherited from a previous port declaration
- A module **input** or **inout** port with a **logic** or **reg** data type explicitly declared, or inherited from a previous port declaration
- A connection to a port of a module instance or interface instance, or a terminal of a primitive instance
- The left-hand side of a continuous assignment statement

By default, the implicit net type that is inferred is the **wire** type. The vector size of the implicit net is based on local context. If the net is inferred from a module port declaration, then the vector size of the implicit net will be the size of the port. If the net is inferred from a connection to a module, interface or primitive instance, then a scalar net is inferred. A scalar net is also inferred if the net is inferred from the left-hand side of a continuous assignment. Example 3-1 illustrates several implicit net declarations.

Example 3-1: Example of undeclared identifiers creating implicit nets

```
module mixed_rtl_and_gate_adder
  (input      a,    //implicit wire net, 4-state logic data type
   input logic b,    //implicit wire net, 4-state logic data type
   input reg   ci,   //implicit wire net, 4-state logic data type
   output     sum,  //implicit wire net, 4-state logic data type
   output logic co); //implicit variable, 4-state logic data type

  xor g1 (n1, a, b); // undeclared n1 is implicit wire net
  xor g2 (sum, n1, ci);
  and g3 (n2, a, b); // undeclared n2 is implicit wire net

  assign n3 = n1 & ci; // undeclared n3 is implicit wire net

  always_comb begin
    co = n2 | n3;           // OK because n2 and n3 were
    end                     // previously inferred as net types
  endmodule: mixed_rtl_and_gate_adder
```

The dot-name and dot-star inferred port connections (see sections 2.4.3, page 57, and 2.4.4, page 58, in Chapter 2) do not infer implicit internal nets. Implicit nets that are inferred from port declarations can be used with the dot-name and dot-star inferred port connections, but all internal nets must be explicitly declared in order to use these port connection shortcuts.

Changing the default implicit net type. The implicit net type can be changed using the compiler directive ``default_nettype`. The directive is followed by any SystemVerilog net type. All SystemVerilog code that is compiled after the directive will use the specified net type whenever an implicit net is inferred. The ``default_nettype` must be specified outside of a module or interface boundary.

Example 3-2 defines the implicit net type to be the `uwire` (single driver) type,

Example 3-2: Changing the net type for implicit nets

```

`default_nettype uwire      // change default for implicit nets
module mixed_rtl_and_gate_adder
  input      a,    // implicit uwire net, logic data type
  input logic b,    // implicit uwire net, logic data type
  input reg   ci,   // implicit uwire net, logic data type
  output     sum,  // implicit uwire net, logic data type
  output logic co; // implicit variable, logic data type
);

  xor  g1 (n1, a, b);    // undeclared n1 is implicit uwire net
  xor  g2 (sum, n1, ci);
  and g3 (n2, a, b);    // undeclared n2 is implicit uwire net

  assign n3 = n1 & ci;   // undeclared n3 is implicit uwire net

  always_comb begin
    co = n2 | n3;        // OK because n2 and n3 were
    end                  // previously inferred as net types
endmodule: mixed_rtl_and_gate_adder
`default_nettype wire      // reset default for implicit nets

```

Turning off implicit net declarations. There are advantages and disadvantages of implicit nets. Large, complex netlists will likely require several dozen 1-bit nets in order to connect the design blocks. Having to explicitly declare these many nets is tedious and time consuming. Explicitly declaring large numbers of interconnecting nets can also require a lot of typing, with the inherit risk of typographical errors that require debugging. Implicit nets can reduce the time required to write netlist models and reduce typographical errors.

A disadvantage of implicit nets, however, is that an incorrectly spelled name in a connection to a module, interface or primitive instance will not be detected as a connection error. The incorrect name will infer an implicit net. The result is a functional bug that must be detected, debugged, and corrected. Another disadvantage is that a net inferred from a connection to an instance will be a 1-bit net, regardless of the size of the port to which the net is connected. A connection size mismatch will result in a warning message, but simulation or synthesis will still proceed. The size mismatch can also result in a functional bug that must be detected and corrected.

The pros and cons of implicit nets versus explicitly declaring nets is a topic that is often adamantly debated between Verilog and SystemVerilog engineers. It is really a matter of user preference. Both coding styles work well, and both styles have advantages and disadvantages.

For engineers, or companies, who prefer explicitly declaring all nets, SystemVerilog provides a way to disable implicit nets. This makes it mandatory to explicitly declare all nets. Disabling implicit nets is done by setting the compiler directive:

```
`default_nettype none // turn off implicit nets
```

This directive must be set outside of a module, and will remain in effect for all subsequent modules that are compiled into the same compilation unit, or until another ``default_nettype` directive is encountered.

Using implicit nets, or disabling implicit nets, is often a personal preference, and sometimes a coding guideline within a company. The examples in this book assume that implicit nets are enabled, and that the default implicit net type is **wire**.

NOTE

The ``default_nettype` directive can affect multiple files. Compiler directives are quasi-global in a compilation unit. When multiple files are compiled in the same compilation unit, a compiler directive has no effect on any files compiled before the directive is encountered, but does affect all files compiled after the directive is encountered.

Best Practice Guideline 3-8

If the default net type is changed, always use ``default_nettype` as a pair of directives, with the first directive setting the default to the desired net type, and the second directive setting the default back to **wire**.

Setting the default net type back to **wire** after any module that changes the default will prevent unintentional side effects of affecting other files that expect a default of **wire**.

3.5.4 Net assignment and connection rules

Assigning values to nets. Nets can receive a value from two types of sources: as a connection to an output or inout port, and as the left-hand side of a continuous assignment (an **assign** statement). Nets cannot be used on the left-hand side of procedural assignments.

Continuous assignments are evaluated throughout simulation. Any changes on the right-hand side of the assignment cause the right-hand side expression to be re-evaluated and the left-hand side updated. The left-hand side can be a variable or a net. Continuous assignments to a net can be explicit or implicit. An explicit continuous assignment begins with the keyword **assign**.

```
wire [15:0] sum;
assign sum = a + b;      // explicit continuous assignment
```

An implicit continuous assignment combines the declaration of a net and the assignment to that net. The **assign** keyword is not used in the combination.

```
wire [15:0] sum = a + b; // net with implicit continuous
                          // assignment
```

Be careful not to confuse in-line variable initialization and implicit continuous assignments.

```
logic [15:0] v1 = a + b; // in-line variable initialization

wire [15:0] n1 = a + b; // net with implicit continuous
                        // assignment
```

The syntax for these two constructs might appear to be similar, but the behavior is quite different. The in-line variable initialization is a one-time evaluation and assignment. In the preceding example, the variable **v1** is not updated if the values of **a** or **b** change later in simulation. The implicit continuous assignment is, as its name implies, an expression that is continuously being evaluated throughout simulation. In the preceding example, the net **n1** is updated every time the values of **a** or **b** change during simulation.

Connection size mismatches. A net is used to connect design blocks together, such as connecting an output port of one module to an input port of one or more other modules. Typically, the vector widths of the ports and the interconnecting net are the same, but SystemVerilog allows the vector sizes to be different. For example, a 16-bit scalar net can connect a 32-bit wide output port to an 8-bit wide input port. This mismatch in sizes is probably a design error, but, in SystemVerilog, only a warning is generated.

The SystemVerilog language has rules for resolving port/connection mismatches:

- *A port has fewer bits than the net or variable connected to it* — the left-most bits of the value are truncated, resulting in the most-significant bits of the value being lost.

- *A port has more bits than the net or variable connected to it* — the value of the net or variable is left-extended. If either the port, net/variable is unsigned, the value is zero-extended. If both the port and the net/variable are signed, the value is sign-extended.

Simulators and synthesis compilers will generate warning messages for connection size mismatches. These warning should not be ignored! Connection mismatches are usually a design error that needs to be corrected.

The dot-name and dot-star inferred port connections (see sections 2.4.3, page 57, and 2.4.4, page 58, in Chapter 2) do not allow connection size mismatches.

3.6 Port declarations

Module definitions include a port list, which is enclosed in parentheses. Ports are used to pass data into or out of a module. Modules can have four types of ports: **input**, **output**, bidirectional **inout**, and interface. Input, output and inout ports are discrete ports, where each port communicates a single value or user-defined type. Interface ports are compound ports, that can communicate a collection of several values. This section describes the syntax and usage guidelines of discrete ports. Interface ports are described in Chapter 10.

3.6.1 Synthesizable port declarations

A port declaration defines a port's *direction*, *type*, *data type*, *sign*, *size* and *name*.

- The *port direction* is declared with the keywords **input**, **output**, or **inout**.
- The *port type* and *data type* can be a variable or any of the net types and data types described in sections 3.4 (page 67) and 3.5 (page 76).
- The *port sign* can be signed or unsigned.
- The *port size* can range from 1-bit wide to 2^{16} (65,536) bits wide. In practice, engineers must consider the size limitations of the ASIC or FPGA technology that will be used to implement the design.

Ports are declared in a *module port list*, which is enclosed in simple parentheses. Ports can be listed in any order. Some engineers prefer listing inputs first, followed by outputs. Other engineers prefer listing outputs first, followed by inputs. Some companies have strict coding style rules regarding the order of ports, and other companies leave the order up to the engineer(s) writing the module definition. Engineers also differ widely on coding style regarding the use of indentation, and whether to list multiple ports on the same line or separate lines.

SystemVerilog provides three coding styles for declaring port lists and port declarations: *combined-style*, *legacy-style* and *legacy-style with combined type and size*.

Combined-style port lists. The *combined-style* port list puts the full declaration of each port within the port list parentheses. This style is preferred by most engineers.

```
module alu
  (input  wire logic signed [31:0] a,           // 32-bit input
   input  wire logic signed [31:0] b,           // 32-bit input
   input  wire logic      [ 3:0] opcode,        // 4-bit input
   output var logic signed [31:0] result,        // 32-bit output
   output var logic          overflow,         // 1-bit output
   output var logic          error);           // 1-bit output
```

Observe that each port declaration is separated by a comma, and that the last port in the list does not have a comma before the closing parenthesis.

Multiple ports of the same direction, type, data type and size can be declared using a comma-separated list of port names. By combining the declarations of similar ports, the preceding port list can be simplified as follows:

```
module alu
  (input  wire logic signed [31:0] a, b,       // 32-bit inputs
   input  wire logic      [ 3:0] opcode,        // 4-bit input
   output var logic signed [31:0] result,        // 32-bit output
   output var logic          overflow,         // 1-bit
   output var logic          error);           // 1-bit
```

The IEEE SystemVerilog standard refers to the combined-style of port declarations as an *ANSI-style port list*, because the style is similar to the ANSI C style for function declarations. This style of port declaration was added to Verilog as part of the Verilog-2001 standard.

Legacy-style port lists. The original Verilog-1995 standard separated the port list and the declarations of the type, data type, sign and size of each port. The SystemVerilog standard refers to this separated style as *non-ANSI style port lists*. This style is similar to the original, pre-ANSI C style for function declarations. The following example uses Verilog-2001 data types. The SystemVerilog **logic** type can also be used with the legacy Verilog-style port list.

```
module alu (a, b, opcode, result, overflow, error);
  input  [31:0] a, b;                      // 32-bit inputs
  input  [ 3:0] opcode;                    // 4-bit input
  output [31:0] result;                   // 32-bit output
  output      overflow, error;           // 1-bit outputs

  wire signed [31:0] a, b;                // 32-bit nets
  wire      [ 3:0] opcode;              // 4-bit net
  reg  signed [31:0] result;            // 32-bit variable
  reg      overflow, error;           // 1-bit variables
```

Observe that each port declaration is terminated by a semicolon, but a comma-separated list of port names can be used for ports that have the same direction and size, or

the same type, data type, and size (such as port `a` and `b`, or `overflow` and `error` in the preceding port declarations).

If the port direction, type, data type, sign and size are all omitted on the first port in the port list, then a legacy non-ANSI style port list is assumed for the entire port list. All ports in a port list must be either the combined ANSI style or the legacy non-ANSI style. It is illegal to mix the two styles in the same port list.

Legacy-style port lists with combined direction and size . The Verilog-2001 standard allows the legacy-style port list to combine the direction declaration and the type/data type declaration into a single statement.

```
module alu_4 (a, b, opcode, result, overflow, error);
    input wire signed [31:0] a, b;           // 32-bit inputs
    input wire      [ 3:0] opcode;          // 4-bit input
    output reg signed [31:0] result;         // 32-bit output
    output reg                  overflow, error; // 1-bit output
```

Module port defaults. There are implicit defaults for each port's direction, type, data type, signedness, and size. The port *type* can be a net, such as `wire`, or a variable, such as `var`. The port *data type* can be `logic` (4-state) or `bit` (2-state). The default rules for port direction, type, data type, signedness and size are:

- *No direction specified* — The default direction for module ports is `inout`, but only until a direction has been defined. Once a direction has been specified, that direction applies to all subsequent ports until a new direction is specified.
- *No type specified* — The default type ports is `wire` when no data type, such as `logic`, is specified. When a data type is specified, the default type is `wire` for input and inout ports and `var` for output ports. and `wire` can be changed using the ``default_nettype` compiler directive, as described in section 3.5.3 (page 80).
- *No data type specified* — The default data type for all ports is `logic` (4-state).
- *No signedness specified* — The default signedness is the default signedness of the port's data type. The `reg`, `logic`, `bit` and `time` data types default to unsigned. The `byte`, `shortint`, `int`, `integer`, and `longint` data types default to signed.
- *No size specified* — The default size is the default size of the port's data type. The `reg`, `logic` and `bit` data types default to 1-bit wide. The default size for other data types is discussed in section 3.4.2 (page 70).

The following code snippet is not a realistic RTL coding style, but serves to illustrate the implicit defaults of module port declarations.

```
module alu                                // IMPLICIT DEFAULTS:
  (wire logic signed [31:0] a, b,          // inout
   wire logic [3:0]             opcode,     // inout, unsigned
   output signed [31:0]          result,     // wire, logic
   output var                  overflow,   // logic, unsigned, 1-bit
   output bit                  error       // var, unsigned, 1-bit
);
```

Although the port declarations in the preceding code snippet are synthesizable, it is not a recommended coding style for synthesizable RTL models. Section 3.6.3 (page 88) provides some coding guidelines for port declarations.

Inherited port declarations. An explicit declaration of a port’s direction, type, data type, signedness, or size can be inherited by subsequent ports in a port list. Inherited port characteristics are “sticky”, because a characteristic sticks (remains in effect) until it is changed.

The port declaration inheritance rules are:

- *Inherited port direction* — An explicit port direction declaration remains in effect until a new direction is specified, even when the port type changes.
- *Inherited port type* — An explicit port type declaration remains in effect until a new direction or type is specified.
- *Inherited port data type and data type* — An explicit port data type declaration remains in effect until a new direction or type or data type is specified.
- *Inherited port signedness* — An explicit port signedness declaration remains in effect until a new direction or type or data type or size is specified.
- *Inherited port size* — An explicit port size declaration remains in effect until a new direction or type or data type or size is specified.

The next code snippet is not a recommended RTL coding style, but illustrates how subsequent ports will inherit characteristics from prior port declarations in a module port list.

```
module alu                                // INHERITED
  (input wire logic signed [31:0] a,      // CHARACTERISTICS:
   b,                                     // all
   tri1 logic [3:0] opcode,    // only direction
   output logic [31:0] result,   // none
   var logic [1:0] overflow,  // only direction,
   error logic [1:0] // only direction,
   // type, and
   // data type
  );

```

3.6.2 Non synthesizable port declarations

SystemVerilog has several additional port types and declaration capabilities that are not universally supported by major synthesis compilers, including:

- Module **ref** reference ports
- Module interconnect ports
- Input port default values (such as **input logic [7:0] a=0**)
- Output port initialization (such as **output logic [7:0] y=1**)

- Port expressions (such as `.b({c,d})`)
- Extern modules and nested modules with implicit ports

Some synthesis compilers might support some of these constructs, but they are not discussed in this book, because they are not supported by all major synthesis compilers at the time this book was written. These constructs can be useful for verification, and are outside the scope of this book on RTL modeling.

3.6.3 Module port declaration recommendations

SystemVerilog provides considerable capability and flexibility for declaring module ports, as has been shown in this section. Engineers should adopt a consistent coding style for port declarations in order to ensure the models are self-documenting, easier to maintain, and easier to re-use in future projects.

Best Practice Guideline 3-9

Use the ANSI-C style declarations for module port lists. Declare both input ports and output ports as a `logic` type.

Some best-practice coding recommendations for declaring module ports include:

- Use combined ANSI-C style port lists, so that all port information is contained within the port list.
- Declare the direction of each port, rather than relying on default port directions and inherited (sticky) port directions.
- Declare all ports data types as the `logic` data type. Avoid 2-state data types in RTL models — they can hide design bugs.
- Do not declare the port type. Allow the language to infer a `wire` or `var` type. The implicit default type for input and output ports works well for synthesizable RTL level models. *Exception:* Tri-state ports can optionally be declared as a `tri` type. The `tri` type is identical to `wire`, but the explicit declaration helps document that the port is expected to be tri-stated.
- Declare each port on a separate line. This allows adding a comment to describe the usage or assumptions regarding each port. *Exception:* It can be acceptable to have a comma-separated list of port names that all have the same direction, data type, size, and similar usage.

Example 3-3 illustrates a module port list using these coding guidelines.

Example 3-3: Module port declaration using recommended coding guidelines

```
module alu
  input logic signed [31:0] a, b,      // ALU operand inputs
  input logic      [ 3:0] opcode,     // ALU operation code
  output logic signed [31:0] result,    // Operation result
  output logic          overflow,   // Set if result overflow
  output logic          error,      // Set if operation error
);
  /////////////////
  //           model functionality not shown      //
  /////////////////
endmodule: alu
```

Traditional Verilog considerations. Traditional Verilog, prior to SystemVerilog, did not have the **logic** data type, and had different rules for implicit default port types. Traditional Verilog would assume a port type of **wire** for all ports, unless the port was explicitly declared as **reg**, which would infer a variable. Engineers had to be careful to use explicit with port declarations in order to ensure each port would have the correct type and data type for the functionality within the module. Getting all the declarations correct often required compiling code, checking for compilation errors — or worse, warnings that were easy to overlook, fixing the errors or warning, and compiling again. A change to the way functionality was modeled, could often result in new compilation errors because a change to the port data types was also required.

SystemVerilog makes port declarations much easier. Simply declare all ports as a **logic** data type, and let the language correctly infer the proper net or variable type. SystemVerilog will correctly infer a net or variable for almost all circumstances.

3.7 Unpacked arrays of nets and variables

SystemVerilog has two types of arrays: packed arrays and unpacked arrays. *Packed arrays* are a collection of bits that are stored contiguously, and are commonly referred to as vectors. Packed arrays are discussed in section 3.4.2 (page 70). *Unpacked arrays* are a collection of nets or variables.

Each net or variable in the collection is referred to as an *array element*. Each element of an unpacked array is exactly the same type, data type and vector size. Each unpacked array element can be stored independently from other elements; the elements do not need to be stored contiguously. Software tools, such as simulators and synthesis compilers, can organize the storage of unpacked arrays in whatever form the tool deems optimal.

The basic declaration syntax of an unpacked array is:

type_or_data_type *vector_size* *array_name* *array_dimensions*

The *array_dimensions* defines the total number of elements the array can store. Unpacked arrays can be declared with any number of dimensions, with each dimension storing a specified number of elements. There are two coding styles for declaring the array dimensions: *explicit addresses*, and *array size*.

The *explicit addresses* style specifies the starting address and ending address of the array dimension between square brackets, in the form:

`[start_address : end_address]`

The *start_address* and *end_address* can be any integer value. The array could start with address 0, address 512, or whatever address is required for the hardware being modeled. The range between the start and ending address represents the size (number of elements) for the array dimension.

The *array_size* style defines the number of elements to be stored in square brackets (similar to the C language array declaration style).

`[size]`

With the *array_size* style, the starting address is always 0, and the ending address is always *size - 1*.

Some example unpacked array declarations are:

```
// a 1-dimensional unpacked array of 1024 1-bit nets
wire n [0:1023];

// a 1-dimensional unpacked array of 4096 16-bit variables
logic [15:0] mem [0:4095];

// a 2-dimensional unpacked array of 32 real variables
real look_up_table [0:15][0:15];

// a 3-dimensional unpacked array of 32-bit int variables
int data_array [255][4][4];
```

The preceding declaration for *mem* is a one-dimensional array of 16-bit logic variables. A one-dimensional array is sometimes referred to as a *memory array*, since it is often used to model the storage of hardware memory devices such as RAMs and ROMs.

3.7.1 Accessing array elements

Each element of an unpacked array can be referenced using an array index. The index follows the array name, and is in square brackets. A multidimensional array requires multiple sets of square brackets to select a single element from the array.

```
logic [15:0] mem [0:4095];
data007 = mem[7]; // read one element of mem array

real look_up_table [0:15][0:15];
look_up_table[0][15] = 2.15; // write to one element of array
```

An array index can also be the value of a net or a variable, as in the next example.

```
always_ff @(posedge clk)
    data <= mem[address]; // value of address is array index
```

Chapter 8, section 8.3 (page 317) discusses using arrays to model RAM functionality, and shows examples of complete synchronous and asynchronous RAM models.

3.7.2 Copying arrays

An unpacked array can be copied to another unpacked array by using an assignment statement, provided the two arrays have an identical layout. That is, the two arrays must store the same data types of the same vector size, must have the same number of dimensions, and have the same size for each dimension.

An array copy results in each element of a source array (the right-hand side of the assignment) being copied to its corresponding element in the destination array (the left-hand side of the assignment). The index numbering of the two arrays do not need to be 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];

always_ff @(posedge clk)
    if (data_copy)
        a <= b; // assign unpacked array to an unpacked array
```

In a similar manner to an array copy, a portion of an array, referred to as an *array slice*, can be copied to a slice of another array, provided the layout of the two slices are identical. A slice is one or more contiguously numbered elements within one dimension of an array.

The original Verilog language, before it became SystemVerilog, restricted the access to arrays to just one element of the array at a time. Array copies and reading/writing to multiple elements of an array was not allowed.

3.7.3 Array list assignments

An unpacked array, or a slice of an array, can be assigned a list of values enclosed between '{' and '}' braces for each array dimension.

```
logic [7:0] lut [0:3]; // array with 4 elements
logic [7:0] a, b, c, d; // 4 8-bit variables

always_ff @(posedge clk)
    if (init)
        lut <= '{8'h12, 8'h34, 8'hAB, 8'hCD}; // initialize array
    else if (load)
        lut <= '{a, b, c, d}; // load array
```

The list 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 SystemVerilog concatenation operator, as discussed in Chapter 5, section 5.11 (page 181).

A multidimensional array can also be assigned a list of values by using nested lists. The nested sets of lists must exactly match the dimensions of the array.

```
logic [7:0] data [0:1][0:3]; // 2-by-4 array layout

data = '{ '{0,1,2,3}, '{4,5,6,7} }; // 2-by-4 nested lists
```

This array assignment is equivalent to the separate assignments of:

```
data[0][0] = 0;
data[0][1] = 1;
data[0][2] = 2;
data[0][3] = 3;
data[1][0] = 4;
data[1][1] = 5;
data[1][2] = 6;
data[1][3] = 7;
```

NOTE

The '`{...}`' list is not the same as the '`{...}`' concatenate operator, which is described in Chapter 5, section 5.11 (page 181). The list operator treats each value in the list as a separate value that corresponds to a separate element in an array. The concatenate operator packs the values in the list into a vector.

All elements of an unpacked array can be assigned the same value by specifying a default value. The default value is specified using '`'{default:<value>}`', as shown in the following code snippet:

```
logic [7:0] lut [0:3]; // array with 4 elements
logic [7:0] a, b, c, d; // 4 8-bit variables

always_ff @(posedge clk or negedge rstN) // async reset
  if (!rstN) // active-low reset
    lut = '{default: '0}; // reset entire array
  else
    lut = '{a, b, c, d}; // load array
```

3.7.4 Bit-select and part-select of array elements

A bit or group of bits can be selected from an array element vector. A single element of the array must first be selected, followed by the bit select or part select.

```

logic [15:0] mem [0:4095];
logic [15:0] data;
logic [ 3:0] nibble;
logic lsb;

data = mem[5];           // access entire array element
lsb = mem[5][0];         // bit-select of array element
nibble = mem[5][11:8];   // part-select of array element

```

Passing arrays through ports and to tasks and functions. Unpacked arrays of any type and any number of dimensions can be passed through module ports, or to task and function arguments. The port or task/function formal argument must also be declared as an array. The port or argument array must have an identical layout as the array to be passed (the same rules as an arrays copy).

```

module cpu (...);

...
logic [7:0] data_table [0:255]; // 1-dimensional array

gpu i1 (.lut(data_table)); // connect array to gpu module
...
endmodule: cpu

module gpu
(input logic [7:0] lut [0:255] // array port
 ...
); // other ports
...
endmodule: gpu

```

The original Verilog language only allowed simple vectors to be passed through a module port, or to a task or function argument. To pass the values of the table array in the example above would have required 256 ports, one for each element of the array.

3.8 Parameter constants

Modules can be modeled to be configurable by using a **parameter** construct. Modules containing parameter constants are referred to as *parameterized modules*. Interfaces (described in Chapter 10) can also be parameterized.

Example 3-4 shows a simple parameterized module, where the vector widths of the **a** and **b** input ports, and the **sum** output port, are based on the value of a parameter named **N**. The default value of **N** is 8, making the model an 8-bit wide adder.

Example 3-4: Add module with parameterized port widths

```
module add_n_bits
#(parameter N = 8)
  (input logic [N-1:0] a, b,
   output logic [N-1:0] sum
  );
  assign sum = a + b;           // N-bit wide addition, no carry
endmodule: add_n_bits
```

Parameters are run-time constants, meaning the value of the parameter can be configured during compilation/elaboration time, and becomes fixed once simulation starts running, or when synthesis begins the process of translating RTL functionality into an ASIC or FPGA implementation. Another module can instantiate the `add_n_bits` module and reconfigure parameter `N` for that instance, as shown in the following code snippet.

```
module alu /* ports not shown */;
  logic [31:0] a, b, sum;
  add_n_bits #(N(32)) add32 (./*); // configure as 32 bits
endmodule: alu
```

Sections 3.8.1 and 3.8.2 (page 97) provide details on declaring and overriding module parameters.

3.8.1 Parameter declarations

There are two types of parameter constants supported by synthesis compilers:

- **parameter** — a run-time constant that can be externally modified.
- **localparam** — a run-time constant that can only be set internally.

Parameters can be declared in two places within a module: using a *#(...)* *parameter list* before the module port list (as shown previously in example 3-4, page 94), or as *local declarations* after the module port list.

Parameters declared within a module use a syntax similar to declaring a local variable or net. The general syntax is:

```
parameter data_type signedness size name = value_expression ;
localparam data_type signedness size name = value_expression ;
```

data_type (optional) in synthesizable RTL models can be **logic**, **reg**, **bit** or a user-defined data type. If the data type is not specified, the parameter takes on the data type of the final value assigned to the parameter.

signedness (optional) is declared with the keyword **signed** or **unsigned**. If not specified, the default signedness of the data type is used. If neither the data type nor

the signedness are specified, the parameter takes on the signedness of the final value assigned to the parameter.

size (optional) is specified in the same way as for variable and net vectors. If not specified, the default size of the data type is used. If neither the data type nor the size are specified, the parameter takes on the size of the final value assigned to it.

name can be any legal or escaped identifier name. A common convention is to use all capital letters for constants, though there is no syntactic requirement to do this.

value_expression can be any expression that resolves to a valid value for the parameter data type. The value expression must be a *constant expression*, which means it must be able to be evaluated by a compiler, without running simulation. A constant expression can use literal values, other constants, and calls to *constant functions* (functions that do not have output or inout arguments, or external references).

Multiple parameters of the same explicit or implicit data type, signedness and size can be declared as a comma separated list of names.

Some example local parameter declarations are:

```
module parameter_examples;
    parameter SIZE = 256; // defaults to a logic signed [31:0]
    parameter PI = 3.14; // defaults to a real data type
    parameter string REV = "version 1.1a"; // explicit type
    localparam bit [15:0] N = $clog2(SIZE); // explicit type
    localparam [2:0] READY = 3'b001, // 3 constants, logic type
                    LOAD = 3'b010,
                    STORE = 3'b100;
    ...
endmodule: parameter_examples
```

Parameters SIZE and PI above will take on the data type of the final value assigned to them. The data type could change during compilation and elaboration if the parameter values are overridden by an external assignment.

Parameters REV and N have an explicit data type. The value assigned to the parameter must be compatible with the parameter's data type, and the value will be converted to that data type. This restriction also applies to any external assignments to the parameter that override the declared value.

Parameters READY, LOAD and STORE have an explicit size, and an implicit data type of **logic**. The value assigned to the parameter must be compatible with the parameter's data type, and the value will be converted to that data type.

Parameters N, READY, LOAD and STORE are **localparam** parameters. As such, the value of the parameter cannot be overridden by an external redefinition assignment. However, the value of a localparam can be an expression that is calculated from other parameters that can be overridden, as shown with parameter N.

The value for parameter `N` is the value returned from a call to the `$clog2` system function. This is a constant function that returns the ceiling of the log base 2 of its argument (the log rounded up to an integer value).

Module parameter lists. A *parameter list* is specified before the module port list, and allows parameters to also be used to make the module ports configurable.

A parameter list is enclosed between the tokens `#(` and `)`. The list can contain declarations for any number of parameters. The syntax is the same as for local parameter declarations, with the exception that each declaration is separated by a comma instead of a semicolon. Example 3-5 illustrates the use of a parameter list to model a configurable bus-functional RAM.

Example 3-5: Model of a configurable RAM using a module parameter list

```
module ram
#(parameter  SIZE = 1024,           // address size of the RAM
          D_WIDTH = 8,            // data bus width
          A_WIDTH = $clog2(SIZE) // address bus width
)
  (input logic [A_WIDTH-1:0] addr,
   input logic             rdN, wrN,
   inout tri    [D_WIDTH-1:0] data
);

logic [D_WIDTH-1:0] mem [0:SIZE-1];

assign data = (!rdN) ? mem[addr] : 'z;

always @(wrN, addr, data)
  if (!wrN) mem[addr] <= data;
endmodule: ram
```

Observe in this code that the parameters in the parameter list are separated by commas, and that there is no comma or semicolon after the closing parenthesis of the parameter list.

The `parameter` keyword is optional in a module port list. The `#(` token indicates that a parameter list is beginning, so software tools do not need the `parameter` keyword to know that parameters are being defined.

It is also optional to assign a value to parameters in a parameter list. If a parameter does not have a value in its declaration, it is mandatory that a value be assigned externally from a parameter override, as discussed in section 3.8.2 (page 97).

3.8.1.1 Type parameters

The data types used with a module or interface can also be made configurable by using *type parameters*, which are declared using the keyword pair **parameter type**. The **parameter** keyword is optional when type parameters are declared in a parameter list.

Type parameters are assigned a built-in or user-defined data type, instead of a logic value. Example 3-6 illustrates an adder that can be configured to work with any data type.

Example 3-6: Adder with configurable data types

```
module add_type
  #(parameter type DTTYPE = logic [0:0]) // default is 1-bit
  (input DTTYPE a, b,
   output DTTYPE sum
  );
  assign sum = a + b;
endmodule: add_type
```

3.8.2 Parameter overrides (parameter redefinition)

Parameterized modules can be configured with unique values for each instance of the module. Defining a new value or type for a parameter is referred to as a *parameter override* or *parameter redefinition*.

There are three syntax styles for parameter redefinition: *in-line named redefinition*, *in-line order redefinition*, and *hierarchical defparam redefinition*.

In-line named redefinition redefines parameters by name, in-line with the instantiation of the module. The new parameter values are specified between a # (token and a closing) token, which comes after the module name and before the instance name.

Example 3-5 (page 96), shown earlier in this section, is a model of RAM that contains three parameters, SIZE, D_WIDTH, and A_WIDTH. Two examples of in-line named redefinition for the SIZE and D_WIDTH parameters are:

```
// redefinable RAM size and word width
ram #(SIZE(65536), .D_WIDTH(16)) i1 (.*);
// override both
ram #(.D_WIDTH(24)) i2 (.*);
// override just one parameter
```

The following code snippet illustrates three ways of overriding the DTTYPE type parameter shown in Example 3-6 (page 97):

```
typedef logic signed [23:0] bus24_t; // user-defined type
int ai, bi, si; // 32-bit signed variables
real ar, br, sr; // floating-point variables
bus24_t a24, b24, s24; // 24-bit signed variables
```

```
// instantiate the adder and configure as 32-bit signed type
add_type #(.DTYPE(int) ) i3 (.a(ai), .b(bi), .sum(si) );

// instantiate the adder and configure as floating-point type
add_type #(.DTYPE(real) ) i4 (.a(ar), .b(br), .sum(sr) );

// instantiate the adder and configure as 24-bit signed type
add_type #(.DTYPE(bus24_t)) i5 (.a(a24), .b(b24), .sum(s24));
```

With in-line named redefinition, the order in which parameters are defined within the module does not matter, since each parameter is explicitly named in the redefinition. A specific parameter can be redefined without redefining other parameters.

In-line order redefinition overrides parameters in the order in which they are defined within a module, instead of using the name of the parameter.

```
// redefine the RAM size and word width
ram #(65536, 16) i3 (*.*) // override parameters by order
```

A disadvantage of in-line order redefinition is that the code is not self-documenting. It is difficult to tell which parameter is being overridden to which value. Specifying the override values in the wrong order can result in functional bugs that can be difficult to detect and debug. Another disadvantage is that parameters cannot be skipped over in order to redefine a parameter later in the module declaration order. In the `ram` example, in-line order redefinition could be used to redefine just the `SIZE` parameter, since it is first inside the `ram` module, but in-line order redefinition could not be used to only redefine the `D_WIDTH` parameter.

Hierarchical defparam redefinition is specified with a `defparam` keyword. This type of parameter redefinition is not associated with the instantiation of a parameterized module. Instead, parameters are redefined using a hierarchical path to the parameter. The redefinition can be done from any file and from any place in the design or verification hierarchy. An example hierarchical `defparam` redefinition is:

```
defparam top.dut.main_processor.reg_block1.ram.SIZE = 65536;
defparam top.dut.main_processor.reg_block1.ram.D_WIDTH = 16;
```

The hierarchical `defparam` redefinition is difficult for software tools to compile and elaborate. It is also difficult for engineers to maintain and reuse SystemVerilog design code. Changes to the design code can result in changes to the hierarchy path, which can make hierarchical `defparam` statements invalid. This book does not use hierarchical `defparam` parameter redefinition. Hierarchical `defparam` redefinition is expected to be deprecated (removed) from a future version of the official SystemVerilog language. It is mentioned in this book because the construct is supported by major simulators and synthesis compilers, even though using `defparam` should be considered a *worst-practice*, not a best-practice coding style.

Best Practice Guideline 3-10

Use in-line named parameter redefinition for all parameter overrides. Do not use in-line parameter-order redefinition or defparam statements.

The in-line named parameter redefinition style documents which parameters are being overridden, and prevents inadvertent out-of-order errors.

3.9 Const variables

Any variable can be declared as a constant by specifying a **const** keyword before the variable type. A constant variable can be assigned a value one time. Any subsequent assignments to the variable are illegal. For synthesizable RTL models, the one assignment must be done in-line with the variable declaration. Constant variables cannot be overridden, but can be assigned a parameter value that can be overridden. Constant variables can be used in functions, where parameters cannot be declared.

```
function automatic logic [15:0] do_magic (logic [15:0] a, b);
    const int magic_number = 86;
    ...
endfunction: do_magic
```

3.10 Summary

This chapter has examined the built-in types and data types that are predefined in the SystemVerilog language. The focus has been on the types and data types that are useful for writing RTL models that will simulate and synthesize optimally.

SystemVerilog has both 2-state and 4-state data types. The four-value system of 4-state data types allows accurately modeling hardware behavior. Values of 0, 1 and Z represent physical hardware. The value X is used to model don't-care conditions, where a design engineer does not care if the physical hardware will have a 0 or 1 value. Simulators also use the value of X to indicate potential problems, where simulation cannot determine if actual logic gates would have a 0, 1 or Z. SystemVerilog's two-state types should not be used to model hardware behavior, because they do not have the X value for representing potential design bugs during simulation.

SystemVerilog ***net types***, such as the **wire** type, are used to connect design blocks together. Nets always use 4-state data types, and can resolve a final value when there are multiple sources driving the same net. SystemVerilog ***variable types*** are used to receive values on the left-hand side of assignment statements, and will store the assigned value until another assignment is made to the variable. SystemVerilog has

several net types and variable data types. The syntax for declaring nets and variables has been shown, and important semantic rules discussed. The proper usage of these various nets and variables in RTL models has also been discussed.

SystemVerilog allows models to be written to be configurable by using **parameter** and **localparam** constants. A unique value for a constant can be specified for each instance of a module using parameter overrides, also called parameter redefinition.

* * *

Chapter 4

User-defined Types and Packages

Abstract — Engineers can extend the built-in SystemVerilog types with additional user-defined types. User-defined types are a powerful modeling construct that allow writing RTL models concisely and accurately. Models written with user-defined types can be more easily reused in other projects. This chapter presents the syntax for declaring user-defined types, along with many examples of using user-defined types in RTL models. The major concepts discussed in this chapter are:

- User-defined type declarations
- Declaration packages
- Enumerated types
- Structures
- Unions

4.1 User-defined types

SystemVerilog provides a mechanism for engineers to define new data types, in addition to the built-in data types discussed Chapter 3. User-defined data types allow new type definitions to be created from existing data types.

User-defined types are created using the **typedef** keyword. For example:

```
typedef int unsigned uint_t;

typedef logic [3:0] nibble_t;
```

Once a new data type has been defined, variables and nets of the new data type can be declared.

```
uint_t a, b;           // two variables of type uint_t

nibble_t opA, opB;    // two variables of the nibble_t type

wire nibble_t [7:0] data; // 32-bit net comprising 8 nibble_t
```

4.1.1 *Naming conventions for user-defined types*

A user-defined type name can be any legal identifier. In large designs, the source code where a user-defined type is defined and the source code where it is used could be separated by many lines of code, and could be in different files. If the name of the user-defined type is similar to the names used for modules, nets, or variables, then this separation of the `typedef` definition and the usage of the type can make it difficult to read and maintain the code.

To make source code easier to read and maintain, `typedef` names should use a naming convention that makes it obvious that the name represents a user-defined type. Two common naming conventions are to add an “`_t`” suffix or a “`t_`” prefix to user-defined type names. This book convention uses the “`_t`” suffix convention.

4.1.2 *Local typedef definitions*

User-defined types can be defined locally in a module or interface (interfaces are presented in Chapter 10). A local user-defined type can only be used within the module or interface in which it is defined. Other modules or interfaces that make up the overall design cannot reference that user-defined type.

4.1.3 *Shared typedef definitions*

When a user-defined type is to be used in many different models, the `typedef` definition can be declared in a package.

4.2 SystemVerilog packages

The original Verilog language did not have a place for definitions that would be used in multiple modules. Each module had to have a redundant copy of tasks, functions, constants, and other shared definitions. A legacy Verilog coding style was to place shared definitions into a separate file, which could then be included in other files using the `'include` compiler directive. This directive instructs the compiler to copy the contents of the included file and literally paste those contents into the location of the `'include` directive. Though using file inclusion helps to reduce code redundancy, it is an awkward coding style for code re-use and maintenance.

SystemVerilog added packages to the original Verilog HDL. A *package* is a declaration space that can hold shared definitions. Multiple modules and interfaces can reference these shared definitions either directly, or by importing specific package items, or by importing the entire package. Packages solve the problem of having to duplicate definitions in multiple modules and the awkwardness of using `xxx` to copy definitions into multiple modules.

4.2.1 Package declarations

SystemVerilog packages are defined between the keywords **package** and **end-package**. A package is a separate declaration space. It is not embedded within a Verilog module. The definitions and declarations in a package are referred to as *package items*. The synthesizable definitions that a package can contain are:

- **parameter**, **localparam** and **const** constant declarations
- **typedef** user-defined types
- Automatic **task** and **function** definitions
- **import** statements from other packages
- **export** statements for package chaining
- Time unit definitions

Packages can also contain non-synthesizable verification definitions, such as: classes. Verification constructs are not covered in this book on RTL modeling.

Example 4-1: A package definition with several package items

```
package definitions_pkg;
    localparam VERSION = "1.1";

    `ifdef _64bit
        typedef logic [63:0] word_t;
    `elsif _32bit
        typedef logic [31:0] word_t;
    `else // default is 16 bit
        typedef logic [15:0] word_t;
    `endif

    typedef enum logic [1:0] {ADD, SUB, MULT, DIV2} opcodes_t;

    typedef struct {
        word_t      a, b;
        opcodes_t   opcode_e;
    } instruction_t;

    function automatic word_t multiplier (input word_t a, b);
        // code for a custom n-bit multiplier
    endfunction
endpackage: definitions_pkg
```

The **enum** and **struct** constructs shown in Example 4-1 are discussed later in this chapter. The **word_t** user-defined type definition in the example is within an **'ifdef** conditional compilation directive that defines **word_t** to be either a 16-bit vector, a 32-bit vector, or a 64-bit vector. The **'ifdef** construct allows engineers to choose what code will be compiled at the time the compiler is invoked. All design blocks that

use the `word_t` user-defined type will use the word size that was selected when the package was compiled.

Best Practice Guideline 4-1

Only use `localparam` or `const` definitions for package constants. Do not use `parameter` definitions in packages.

A `parameter` defined in a package is not the same as a `parameter` defined in a module. A module `parameter` constant can be redefined for each instance of the module. A package `parameter` cannot be redefined, since it is not part of a module instance. In a package, a `parameter` is identical to a `localparam`. Parameters and localparams are discussed in Chapter 3, section 3.8 (page 93).

4.2.2 Using package items

The definitions and declarations in a package are referred to as *package items*. Modules and interfaces can reference package items in four ways:

- Wildcard import all package items
- Explicit import of specific package items
- Explicit reference to a specific package and package item
- Import package items into the `$unit` declaration space

The first three methods for referencing package items are discussed in this section. Importing into `$unit` is discussed later in this chapter, in section 4.3 (page 112).

4.2.2.1 Wildcard import of package items

The simplest and most common way for a module to reference package items is to import all items from the package using a *wildcard import* statement. For example:

```
module alu
  import definitions_pkg::*;

  /* module port list */;
```

The `::` double-colons after the package name is a *scope resolution operator*. It instructs a compiler to look in another location (a scope) for information — the `definitions_pkg` package, in this example.

The asterisk (`*`) is the wildcard token. A wildcard import effectively adds the imported package to the search path that SystemVerilog uses.

When a SystemVerilog compiler encounters an identifier (a name), it will first search in the local scope first for the definition of that identifier. A local scope can be a task, function, begin-end block, module, interface or package. If the identifier definition is not found in the local scope, the compiler will then search the next scope

level up until a module or interface or package boundary is reached. If the identifier definition has not been found, then any wildcard imported packages are searched. Finally, tools will search in the `$unit` declaration space (see section 4.3, page 112). The full semantic rules for wildcard imports search rules are more complex than this simple description, and are defined in the IEEE 1800 SystemVerilog standard.

Example 4-2 illustrates using a wildcard import statement.

Example 4-2: Using a package wildcard import

```
module alu
  import definitions_pkg::*;

  (input instruction_t iw,
   input logic clk,
   output word_t result
  );

  always_comb begin
    case (iw.opcode_e)
      ADD : result = iw.a + iw.b;
      SUB : result = iw.a - iw.b;
      MULT: result = multiplier(iw.a, iw.b);
      DIV2: result = iw.a >> 1;
    endcase
  end
endmodule: alu
```

In this example, the wildcard import has the effect of adding the `definitions_pkg` package to the module's identifier search path. The port list can reference the `instruction_t` user-defined type, and the compiler will find that definition in the package. Likewise, the `case` statement can reference the enumerated type labels used by `opcode`, and the definitions for those labels will be found in the package.

4.2.2.2 Explicit importing specific package items

SystemVerilog also allows specific package items to be imported into a module, without adding the entire package to the identifier search path for that module.

The general syntax of explicitly importing a specific package item is:

```
import package_name::item_name;
```

Example 4-3, uses explicit imports to bring specific package items into a module. The explicit imports are more verbose than using a wildcard import, but also make the module more self-documenting. It is readily apparent exactly what package items are being used from the package.

Example 4-3: Importing specific package items into a module

```
module alu
  import definitions_pkg::instruction_t,
           definitions_pkg::word_t;
  (input instruction_t iw,
   input logic clk,
   output word_t result
  );
  import definitions_pkg::ADD;
  import definitions_pkg::SUB;
  import definitions_pkg::MULT;
  import definitions_pkg::DIV2;
  import definitions_pkg::multiplier;

  always_comb begin
    case (iw.opcode_e)
      ADD : result = iw.a + iw.b;
      SUB : result = iw.a - iw.b;
      MULT: result = multiplier(iw.a, iw.b);
      DIV2: result = iw.a >> 1;
    endcase
  end
endmodule: alu
```

NOTE

An explicit import of an enumerated type definition does not import the labels used within that definition. The labels must also be explicitly imported. Enumerated types are discussed in more detail in section 4.4 (page 114), and importing the complete enumerated type definition with its labels is discussed in section 4.4.2 (page 117).

4.2.2.3 Placement of package import statements

A package import statement, whether a wildcard import or a specific item import, can be located:

- Before the module port list — items in the package to be used in the port definitions and within the module.
- After the module port list — the package items can be used within the module, but not within the port definitions.
- Outside of the module definition — the package items are imported into a pseudo-global \$unit declaration space. \$unit has problems and should not be used. The \$unit name space and its hazards are discussed in section 4.3 (page 112).

NOTE

Placing a package import statement before the module port list was added in the SystemVerilog-2009 standard. In SystemVerilog-2005, the import statement could only appear after the port list, or in the **\$unit** space.

Legacy code written prior to SystemVerilog-2009 would sometimes have the import statement outside of the module definition, causing the definitions in a package to be imported into the dangerous **\$unit** name space.

4.2.2.4 Direct package references using the scope resolution operator

The *:: scope resolution operator* can be used to directly reference a package item by specifying the package name and a specific item within the package.

Example 4-4 uses the scope resolution operator to reference several of the package items that are defined in the `definitions_pkg` package shown previously, in Example 4-1 on page 103.

Example 4-4: Explicit package references using the :: scope resolution operator

```
module alu
  (input  definitions_pkg::instruction_t  iw,
   input  logic                           clk,
   output definitions_pkg::word_t        result
  );
  always_ff @(posedge clk) begin
    case (iw.opcode_e)
      definitions_pkg::ADD : result = iw.a + iw.b;
      definitions_pkg::SUB : result = iw.a - iw.b;
      definitions_pkg::MULT: result = definitions_pkg::
                                multiplier(iw.a, iw.b);
      definitions_pkg::DIV2: result = iw.a >> 1;
    endcase
  end
endmodule: alu
```

Explicitly referencing package items can help to document the design source code. In Example 4-4, the use of the package name makes it obvious where the definitions for `instruction_t`, `word_t`, `ADD`, `SUB`, `MULT` and `multiplier` can be found. However, explicitly referencing the package name for every usage of a package item is verbose. A more common way of using definitions in a package is to import the entire package, as discussed previously, in section 4.2.2.1 (page 104). The explicit package reference shown in this section is only needed when there a definition with the same name in multiple packages, and it is necessary to indicate from which package the item is to be imported.

4.2.3 Importing from multiple packages

Larger design projects will often utilize several packages. A module or interface can import from as many packages as needed.

Care must be taken to avoid name conflicts when multiple packages are imported. This is especially true when wildcard imports are used. In the following code snippet, package `cpu_pkg` and `gpu_pkg` both have an identifier called `instruction_t`.

```

package cpu_pkg_pkg;
  typedef logic [31:0] word32_t;
  typedef enum logic [1:0] {ADD, SUB, MULT, DIV2} opcodes_t;
  typedef struct {
    word32_t a, b;
    opcodes_t opcode;
  } instruction_t;
endpackage: cpu_pkg_pkg

package gpu_pkg_pkg;
  typedef enum logic [1:0] {MUL,DIV,SHIFTL,SHIFTR} op_t;
  typedef enum logic {FIXED,FLOAT} operand_type_t;
  typedef struct {
    op_t          opcode;
    operand_type_t op_type;
    logic [63:0]  op_a;
    logic [63:0]  op_b;
  } instruction_t;
endpackage: gpu_pkg_pkg

module processor (...);
  ...
  import cpu_pkg::*;
  import gpu_pkg::*;

  instruction_t instruction; // ERROR: multiple definitions
                            // of instruction_t
endmodule: processor

```

In this code snippet, an `instruction_t` identifier is defined in both packages, and both packages are wildcard imported. The variable `instruction` is declared as an `instruction_t` type. When a simulator, synthesis compiler, or other software tool searches for the definition of `instruction_t`, it will find a definition in both wildcard-imported packages, and will not know which definition to use. The multiple definitions will result in a compilation or elaboration error.

When multiple definitions exist, the source code must either explicitly reference, or explicitly import, the definition to be used in that model. For example:

```

module processor (...);

...
import cpu_pkg::*;
import gpu_pkg::*;
import cpu_pkg::instruction_t;

instruction_t instruction;
...

endmodule: processor

```

An explicit package item reference takes precedence over local definitions or an explicit import of a package item, which takes priority over wildcard imports. The explicit import in the preceding code snippet resolves the ambiguity for which definition of `instruction_t` should be used in the `processor` module.

4.2.4 Package chaining

A package can either explicitly import definitions from another package, or wildcard import another package. The imported items, however, are not automatically visible outside of that package. Consider the following example:

```

package base_types_pkg;
  typedef logic [31:0] word32_t;
  typedef logic [63:0] word64_t;
endpackage: base_types_pkg

package alu_types_pkg;
  import base_types_pkg::*;

  typedef enum logic [1:0] {ADD, SUB, MULT, DIV2} opcodes_t;

  typedef struct {
    word64_t a, b;
    opcodes_t opcode;
  } instr_t;
endpackage: alu_types_pkg

module alu
  import alu_types_pkg::*;
  (input instr_t instruction, // OK: instr_t was imported
   output word64_t result // ERROR: word64_t not found
  );
  ...
endmodule: alu

```

In order for module `alu` to use definitions from both packages, both packages need to be imported into `alu`. SystemVerilog has the ability to *chain* packages, so that a module only has to import the last package in a chain, which would be `alu_types_pkg` in the preceding code snippet. Package chaining is done by using a combination of package **import** and **export** statements.

```

package alu_types_pkg;
  import base_types_pkg::*; // import another package
  export base_types_pkg::*; // export (chain) imported items

  typedef enum logic [1:0] {ADD, SUB, MULT, DIV} opcodes_t;

  typedef struct {
    word64_t a, b;
    opcodes_t opcode;
  } instr_t;
endpackage: alu_types_pkg

```

An export statement can explicitly export a specific item, or use a wildcard to export all items imported from another package. Note that, when using wildcard exports, only the definitions actually used within the package will be exported. In the preceding snippet, the definition for `word32_t` in `base_types_pkg` is not used within `alu_types_pkg`, and therefore is not chained, and is not available in the `alu` module.

The following explicit export could be added to the `alu_types_pkg` example above to chain `word32_t`, so that it would be available in the `alu` module.

```
export base_types_pkg::word32_t; // chain word32 type
```

NOTE

At the time this book was written, some simulators and synthesis compilers were not yet supporting package chaining. The export statement for package chaining was added as part of the SystemVerilog-2009 standard. The SystemVerilog-2005 standard did not define a way to do package chaining.

4.2.5 Package compilation order

SystemVerilog requires that package definitions be compiled before they are referenced. This means that there is file order dependency when compiling packages and modules — the package must be compiled first. It also means that a module that references package items cannot be compiled independently. The package must be compiled along with the module, or have been pre-compiled, if the tool supports separate file compilation.

One way to ensure that packages are compiled before any files that reference the packages or package items is to control the order in which files are listed in the compile command. File compilation order is often controlled using Linux “make” files. Verilog command files (which are read using a `-f` invocation option) and script or batch files can also be used.

Another way to ensure that packages are compiled before any files that reference the packages or package items is to use the ``include` compiler directive to instruct a

compiler to read in the file containing the package. The `include directive is placed at the beginning of each design or testbench file that contains references to the package items.

```
// read in the package used by this model
`include "definitions_pkg.sv" // compile the package

module alu
  import definitions_pkg::*;

  input instruction_t iw,
  input logic clk,
  output word_t result
);

...
endmodule: alu
```

Care must be taken when using the `include directive avoid including the same package multiple times in the same compilation, which SystemVerilog does not allow. This can be done by placing `ifdef (“if defined”) or `ifndef (“if not defined”) conditional compilation directives around the package definition, so that the compiler skips over the entire package if it has already been compiled. Conditional compilation allows SystemVerilog source code to be optionally compiled, based on whether a macro name has been defined using a `define compiler directive.

The following example surrounds the package with an `ifndef conditional compilation directive. The first time the file containing the package is read in by a compiler, the “not defined” test will be true, and the package will be compiled. The lines of code that are compiled contain a `define directive that sets the macro name used by the `ifndef. If this file is read in a second time during the same invocation of the compiler, the “not defined” test will be false, and code between the `ifndef and `endif will be skipped.

```
// Only compile this package if its internal conditional
// compilation flag has not been set. This file sets its
// internal flag the first time it is compiled.
//
`ifndef DEFINITIONS_PKG_ALREADY_COMPILED //if flag is not set
`define DEFINITIONS_PKG_ALREADY_COMPILED // set the flag
package definitions_pkg; // and compile pkg
  ...
  ... // package item definitions
endpackage: definitions_pkg
`endif // end of conditionally compiling this package
```

4.2.6 Synthesis considerations

To be synthesizable, tasks and functions defined in a package must be declared as **automatic**, and cannot contain static variables. The reason for this rule is that synthesis will make a copy of the task or function in each module or interface that references the package task or function. If the task or function were to have static storage

in simulation, that storage would be shared by all references to the task or function. This would be a different behavior than having duplicate copies. By declaring a task or function as `automatic`, new storage is allocated each time it is called, making the behavior the same as a unique copy of the task or function. 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.

For similar reasons, synthesis does not support variable declarations in packages. In simulation, a package variable is 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.

4.3 The \$unit declaration space

NOTE

The `$unit` is a dangerous shared name space that is fraught with hazards. Its use can lead to designs that are difficult to compile and to maintain.

Prior to packages being added to the SystemVerilog standard, a different mechanism was provided to create definitions that could be shared by multiple modules. This mechanism is a pseudo-global name space called `$unit`. Any declaration outside of a named declaration space is defined in the `$unit` name space. In the following example, the definition for `bool_t` is outside of the two modules, and therefore is in the `$unit` declaration space.

```
typedef enum logic {FALSE, TRUE} bool_t;

module alu (...);
    bool_t success_flag;
    ...
endmodule: alu

module decoder (...);
    bool_t a_ok;
    ...
endmodule: decoder
```

`$unit` can contain the same kinds of user definitions as a package, and has the same synthesis restrictions. Unlike a package, however, the `$unit` space can lead to design code that is difficult to maintain, and difficult for software tools to compile. Some of the hazards of using `$unit` are:

- Definitions in `$unit` can be scattered across many files, making code maintenance and code reuse a nightmare.

When a user-defined type, task, function, or other identifier name from a package is referenced, it is relatively easy to locate and maintain the definition of the identifier. There is always an explicit package reference or a package import statement to show where the definition can be found. When a user-defined type, task, function, or other identifier is defined in the **\$unit** space, the definition could be in any file, in any directory, on any server, that makes up the source code of the design and verification testbench. Locating, maintaining, and reusing the definition is difficult, at best.

- When definitions in the **\$unit** space are in multiple files, the files must be compiled in a very specific order.

SystemVerilog requires that definitions be compiled before they are referenced. When **\$unit** declarations are scattered across many files, it can be difficult, and even impossible, to compile all files in the proper order.

- A change to a **\$unit** definition requires recompiling all source code files.

Any change to a definition in **\$unit** will necessitate recompiling all source code that makes up the design and the verification testbench, since any file, anywhere, could use the definition without importing it. Many software tools will not mandate that all files be recompiled, but, if not recompiled, design blocks could end up with obsolete definitions.

- The scope of **\$unit** can be, and often is, different for simulation and synthesis.

Each invocation of a compiler starts a new **\$unit** space that does not share declarations that are in other **\$unit** spaces. Many SystemVerilog simulators compile multiple files together. These tools will see a single **\$unit** space. A **\$unit** definition in one file will be visible to any subsequent file in the single compilation. Most SystemVerilog synthesis compilers, and some simulators, support separate file compilation, where each file can be compiled independently. These tools will see several disconnected **\$unit** spaces. A **\$unit** definition in one file will not be visible to any other file.

- Duplicate identifier names with different definitions can easily occur.

It is illegal in SystemVerilog to define the same name multiple times in the same name space. If one file defines a **bool_t** user-defined type in the **\$unit** space, and another file also defines a **bool_t** user-defined type in **\$unit**, the two files can never be compiled together, since the two definitions would end up in the same **\$unit** space. To avoid this conflict, engineers must add conditional compilation directives using **`define** and **`ifdef**, so that only the first definition encountered by the compiler is actually compiled.

Packages can be imported into **\$unit**, but have all the same hazards as definitions made directly in **\$unit**. Furthermore, care must be taken to not import the same package into the same name space more than once, which is illegal.

Best Practice Guideline 4-2

Avoid using `$unit` like the Bubonic plague! Instead, use packages for shared definitions.

Packages avoid all of the hazards of `$unit`. Using packages provides a controlled name space that is easier to maintain and reuse.

4.4 Enumerated types

Enumerated types provide a means to declare a variable that can have a specific list of valid values. Each value is associated with a label. An enumerated variable is declared with the `enum` keyword, followed by a comma-separated list of labels enclosed in curly braces ({ }).

In the following example, variable `rgb` can have the values of RED, GREEN and BLUE:

```
enum {RED, GREEN, BLUE} rgb_e;
```

The labels in an enumerated list are constants, similar to `localparam` constants. The labels can be any name. This book uses the convention of using all capital letters for constants.

4.4.1 *Enumerated type declaration syntax*

Enumerated types have an underlying data type, referred to as a *base type*, which can be any SystemVerilog built-in data type or a user-defined type. Each label in the enumerated list has a logic value associated with the label.

SystemVerilog provides two styles for declaring enumerated types: *implicit-style* and *explicit-style*.

4.4.1.1 **Implicit-style enumerated declarations**

An *implicit-style enumerated declaration* uses defaults for the base type and the label values. The default base type is `int`. The default for labels is that the first label in the list has a value of 0, and the value for each subsequent label is incremented by one.

In the following implicit-style enumerated declaration:

```
enum {WAITE, LOAD, READY} states_e;
```

- `states_e` is a variable of the `int` data type, which is a 32-bit signed data type. This means the enumerate list could have up to 2,147,483,648 (2^{32-1}) labels.

- WAITE, the first label in the list, has a value of 0, LOAD a value of 1, and READY a value of 2. (The label WAITE is purposely spelled with an “E” at the end to avoid any confusion or conflict with the reserved keyword `wait` in SystemVerilog.)

These defaults are seldom ideal for modeling hardware. The `int` base type is a 2-state type, which means any design problems that result in an X during simulation cannot be reflected in the enumerated variable. The `int` base type is 32-bits wide, which is usually a much larger vector size than the hardware being represented requires. Label values such as 0, 1 and 2 do not represent the encoding used in many types of hardware designs, such as one-hot values, Gray codes, or Johnson counts.

4.4.1.2 Explicit-style enumerated declarations

An *explicit-style enumerated declaration* specifies the base type and the label values. The following declaration represents a 3-bit wide state variable using one-hot encoding:

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

An explicit enumerated declaration imposes several syntax rules that can help prevent coding mistakes:

- The vector width of the base type and the explicit width of the label values must be the same. Unsized literal values are allowed (e.g. WAITE = 1).
- The value for each label must be unique; two labels cannot have the same value.
- There cannot be more labels than the vector width of the base type can represent.

It is not necessary to specify the value of each label in the enumerated list. If a value is not specified, the value 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. D is explicitly defined to have a value of 13, and E and F are given the incremented values of 14 and 15, respectively.

```
enum logic [3:0] {A=1, B, C, D=13, E, F} list1_e;
```

An error will result if two labels end up with the same value. The following example will generate an error, because C and D would have the same value of 3:

```
enum logic [3:0] {A=1, B, C, D=3} list2_e; // ERROR
```

Best Practice Guideline 4-3

Use the explicit-style enumerated type declarations in RTL models, where the base type and label values are specified, rather than inferred.

Specifying the base type and label values has several advantages: It documents the design engineer’s intent, it can more accurately model the gate-level behavior, and it allows more accurate RTL to gate-level logic equivalence checking (see Chapter 1, section 1.8, page 36).

4.4.1.3 Typed and anonymous enumerated types

Enumerated types can be declared as a user-defined type using **typedef**. This provides a convenient way to declare several variables or nets with the same enumerated value sets.

```
typedef enum logic [2:0] {WAITE = 3'b001,
                           LOAD  = 3'b010,
                           READY = 3'b100} states_t;
states_t state_e, next_state_e; // 2 enumerated variables
```

An enumerated type declared using **typedef** is referred to as a *typed enumerated type*. If **typedef** is not used, the enumerated type is referred to as an *anonymous enumerated type*.

4.4.1.4 Enumerated type label sequences

There are two shorthand notations to specify multiple labels with similar names in an enumerated type list.

```
enum logic [1:0] {COUNT_[4]} counts1_e;
```

The COUNT_[4] shorthand notation will generate four labels, with the names COUNT_0, COUNT_1, COUNT_2 and COUNT_3. The value associated with COUNT_0 will default to 0, and the value for each subsequent label will be incremented by one.

The second shortcut allows specifying a range of labels.

```
enum logic [1:0] {COUNT_[8:11]=8} counts2_e;
```

The COUNT_[8:11] shorthand notation will generate four labels, with the names COUNT_8, COUNT_9, COUNT_10 and COUNT_11. The value associated with COUNT_8 is explicitly defined as 8, and the value for subsequent labels will increment by one.

If the first value in the range is less than the second value, as in COUNT_[8:11], the sequence will increment from the first number up to the last. If the first value in the range is greater than the second value, as in COUNT_[11:8], the sequence will decrement from the first number down to the last.

4.4.1.5 Enumerated type label scope

The labels within an enumerated type list must be unique within the scope in which the labels are declared and used. The RTL modeling scopes that can contain enumerated type declarations are the modules, interfaces, packages, begin-end blocks, tasks, functions, and the \$unit compilation unit.

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

```
module controller (...);
  enum logic {GO=1'b1, STOP=1'b0} fsm1_states_e;
  ...
  enum logic [2:0] {READY=3'b001, SET=3'b010, GO=3'b100}
    fsm2_states_e; // ERROR: GO has already been defined
  ...

```

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 name scope.

```
module controller (...);
  ...
  always_ff @(posedge clk) begin: fsm1
    enum logic {GO=1'b1, STOP=1'b0} fsm1_states_e;
    ...
  end: fsm1

  always_ff @(posedge clk) begin: fsm2
    enum logic [2:0] {READY=3'b001, SET=3'b010, GO=3'b100}
      fsm2_states_e;
    ...
  end: fsm2
  ...

```

Giving names to the begin-end blocks as shown above is not required, but helps to document the code for readability and maintenance.

4.4.2 Importing enumerated types from packages

Typed enumerated types can be defined in a package, which allows multiple design blocks and verification code to use the same definition.

NOTE

An explicit import of an enumerated type definition does not import the labels used within that definition.

Using a wildcard import of a package is the easiest solution to this limitation. The wildcard import makes everything in the package available (see section 4.2.2.1, page 104).

When a typed enumerated type definition is imported from a package, only the typed name is imported. The value labels in the enumerated list are not automatically 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_pkg;

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

endpackage: chip_types_pkg

module chip (...);

    import chip_types_pkg::states_t; // only imports the
                                    // states_t name
    states_t state_e, next_state_e;

    always_ff @(posedge clk)           // async reset
        if (!rstN) state_e <= WAITE;   // ERROR: "WAITE" has not
                                         // been imported
        else state_e <= next_state_e;
    ...
endmodule: chip

```

In order to also import the enumerated type labels, 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. The following partial example shows the use of a wildcard import.

```

module chip (...);

    import chip_types_pkg::*; // wildcard import entire package
    states_t state, next_state;
    ...

```

Care must be taken when doing wildcard imports from multiple packages. A compilation or elaboration error will occur if an identifier (a name) is defined in more than one package, and both packages are wildcard imported. For this situation, the identifier to be used must be either explicitly imported or directly referenced. Working with multiple packages is discussed in section 4.2.3 (page 108).

4.4.3 Enumerated type assignment rules

Most SystemVerilog variable types are *loosely typed*, meaning that values of any data type can be assigned to a variable. The value will be converted to the type of the variable by using conversion rules specified in the SystemVerilog standard.

Enumerated types are the exception to SystemVerilog's loosely typed behavior. Enumerated type variables are *semi-strongly typed*, meaning only specific data types can be assigned to the variable.

An enumerated type variable can only be assigned:

- A label from its enumerated type list.
- Another enumerated type variable of the same type. That is, both variables are declared using the same typed or anonymous enumerated type definition.
- A value cast to the type of a typed enumerated type.

These rules are illustrated by example using the following definitions and enumerated variables:

```
typedef enum logic [2:0] {WAITE = 3'b001,  
                           LOAD  = 3'b010,  
                           DONE   = 3'b100} states_t;  
  
typedef enum logic [2:0] {READY = 3'b001,  
                           SET   = 3'b010,  
                           GO    = 3'b100} flags_t;  
  
states_t state, next_state;  
flags_t run_control;
```

The following assignments to the `state` and `next_state` enumerated variables are either legal or illegal, as noted:

```
next_state = LOAD; // legal: LOAD is in enumerated list  
  
state = next_state; // legal: state, next_state are same type  
  
state = 0; // illegal: must use labels, not literal values  
  
state = 3'b100; // illegal: must use labels, even though  
                // it is same value as the DONE label  
  
state = READY; // illegal: READY is not in state's  
                // enumerated list  
  
state = run_control; // illegal: state and run_control are  
                     // from different definitions
```

NOTE

The strongly-typed rules for enumerated types only apply to assignments made to an enumerated variable. The value stored in an enumerated variable is just that — a value, and can be used without restriction in expressions such as comparisons and math operations.

Operations on enumerated type values. When an operation is performed on an enumerated type variable, the value of the enumerated variable is converted to the base type of the enumerated type definition. The result of the operation is no longer an enumerated type. The result can be assigned to a regular, loosely typed variable, but cannot be assigned back to the enumerated variable.

```
logic [2:0] temp;           // non-enumerated variable

temp = next_state + 1;    // legal: temp is loosely typed

state = next_state + 1;   // illegal: next_state + 1 is not
                         // an enumerated expression

state++;                  // illegal: result of ++ is not
                         // an enumerated expression

state += next_state;     // illegal: result of += is not
                         // an enumerated expression
```

Casting expressions to enumerated types. Any value can be cast to a typed enumerated type, and then assigned to a variable of that enumerated type, even if the value does not match one of the labels for the enumerated definition.

```
state = states_t'(temp);  // legal, even if value of temp
                         // does not match any of the label
                         // values

state = states_t('X);    // legal: forces state to an
                         // uninitialized or don't-care
                         // value
```

The cast operator is discussed in Chapter 5, section 5.15 (page 198).

There are occasions in RTL modeling where casting a non-enumerated expression to an enumerated type is necessary. Care must be exercised when using the cast operator, however. Forcing a value into an enumerated variable that is not in its enumerated list can result in undesirable behavior, both in simulation and in synthesis. Using casting puts a burden on the design engineer to ensure that only valid values are forced into the enumerated variable. This is no different than with loosely typed regular variables, where the design engineer needs to ensure that assigned values are valid.

SystemVerilog also has a **\$cast** system function that automatically validates the result of a cast operation. Unfortunately for RTL designers, **\$cast** is not supported by some major synthesis compilers. **\$cast** is useful in verification testbenches, but is not considered a synthesizable construct.

4.4.4 Enumerated type methods

Enumerated types have 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 to the next value in the enumerated type list, and jump to the beginning or end of the list. Using these methods, it is not necessary to know the label names.

NOTE

At the time this book was written, enumerated type methods were supported by some synthesis compilers, but were not universally supported by all synthesis compilers.

The enumerated type methods have limited usefulness for modeling hardware behavior. They are merely shortcuts for what can be done with assignment statements. Due to the synthesis limitations on enumerated type methods, this book only briefly describes these methods and shows a simple example.

Enumerated methods are called by appending the method name to the end of the enumerated type variable name, with a period as a separator. The methods are:

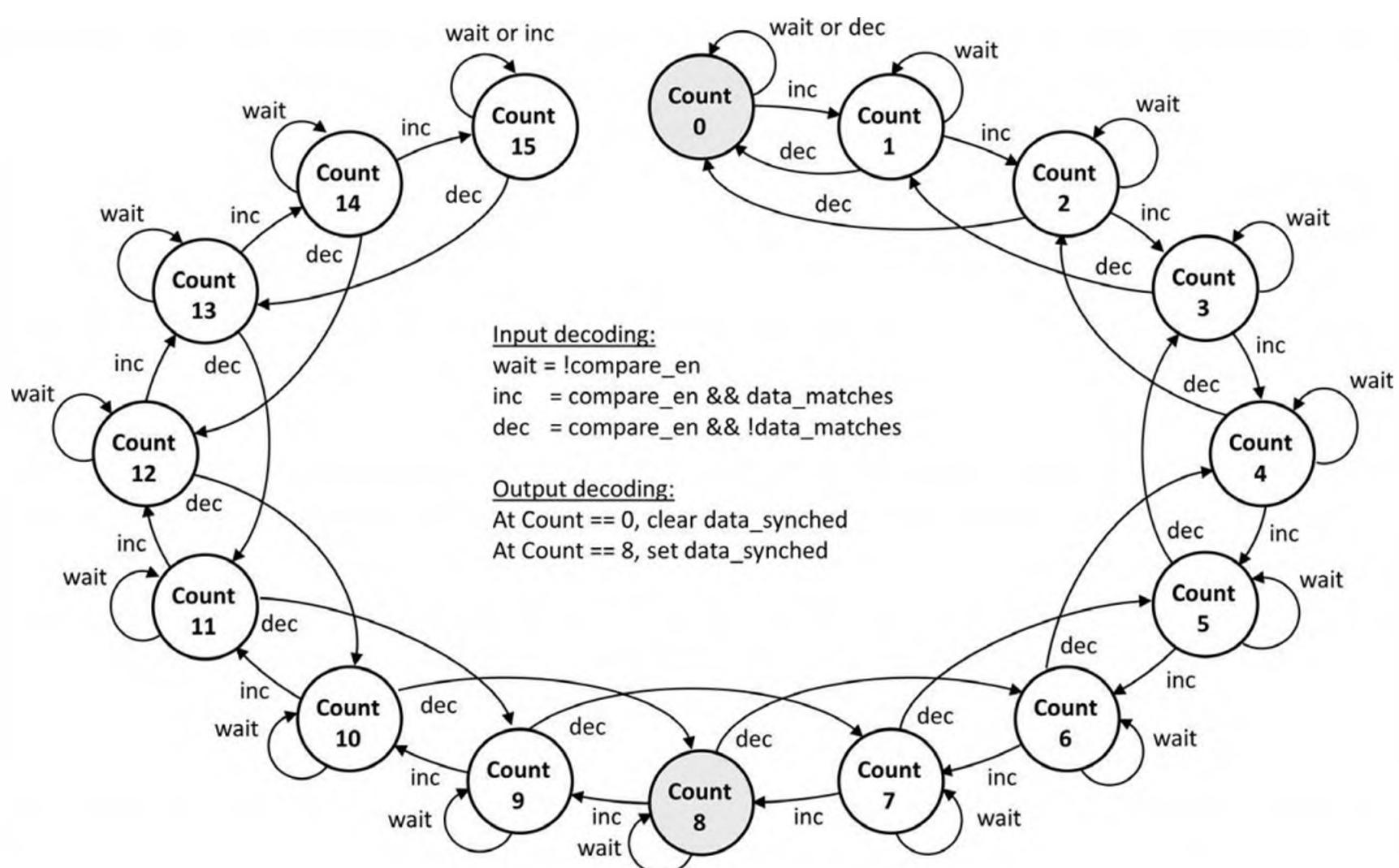
- *enum_variable_name.first* — returns the value of the first member in the enumerated list of the specified variable.
- *enum_variable_name.last* — returns the value of the last member in the enumerated list.
- *enum_variable_name.next(N)* — returns the value of the next member in the enumerated list, based on the current value of the enumerated type variable. 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. When the end of the enumerated list is reached, the method wraps back to the start of the list. If the current value of the enumerated type variable does not match any 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, based on the current value of the enumerated type variable. This method works the same as the **next** method, except that the **prev** method iterates backwards through the list of labels instead of forward.
- *enum_variable_name.num* — returns the number of labels in the enumerated list of the variable.
- *enum_variable_name.name* — returns the string representation of the label for the current value in the enumerated type variable. If the value is not a member of the enumeration, the name method returns an empty string.

Printing enumerated types. Enumerated type values can be printed as either the actual value of the label, or as the name of the label. Printing the enumerated type variable directly will print the current actual logic value of the enumerated type variable. Using the name method allows printing the label representing the current value instead of the actual value.

An example of using enumerated methods. Example 4.4 illustrates using some of these enumerated type methods to model a state machine sequencer. This model is a state machine that either sets or clears a `data_synched` flag. If the `data_matches` input is true for at least 8 consecutive clock cycles, the `data_synched` flag is set. If the `data_matches` input is false for multiple consecutive clock cycles, the `data_synched` flag is cleared. The number of consecutive false `data_matches` required to clear the `data_synched` flag depends on how many consecutive cycles `data_matches` has been true.

Figure 4-1 shows the state flow for this state machine. The state machine represents a counter that can be either incremented or decremented. The counter counts how many consecutive `data_matches` have occurred, up to a maximum of 16. Observe that, for most states, the counter is either incremented by 1, or decremented by 2. The `next` and `prev` enumerated type methods can model this increment or decrement behavior very concisely, but might not be supported by some synthesis compilers.

Figure 4-1: State diagram for a confidence counter state machine



Example 4-5: Using enumerated type methods for a state machine sequencer

```
module confidence_counter
  (input logic data_matches, compare_en, rstN, clk,
   output logic data_synced
  );

  typedef enum logic [3:0] {COUNT[0:15]} states_enum_t;

  states_enum_t CurState, NextState;

  // sequential state sequencer
  always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN) CurState <= COUNT0;           // active low reset
    else       CurState <= NextState;

  // next state combinational logic decoding
  always_comb begin
    if (!compare_en)
      NextState = CurState; // not comparing (no state change)
    else if (data_matches) // compare_en && data_matches
      case (CurState)
        COUNT15 : ; // can't increment past 15
        default: NextState = CurState.next; // increment by 1
      endcase
    else
      case (CurState)
        COUNT0 : ; // can't decrement below 0
        COUNT1 : NextState = CurState.prev(1); // decrement by 1
        default: NextState = CurState.prev(2); // decrement by 2
      endcase
  end

  // data_synced register output
  always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN)
      data_synced <= 0;                   // active low reset
    else
      begin
        if (CurState == COUNT8)
          data_synced <= 1;
        else if (CurState == COUNT0)
          data_synced <= 0;
      end
  endmodule: confidence_counter
```

4.4.5 Traditional Verilog coding style without enumerated types

The Verilog language, before it became SystemVerilog, did not have enumerated types. To create labels for data values, it was necessary to define a **parameter** or **localparam** constant to represent each value, and assign a value to that constant. Alternatively, the `define text substitution macro could be used to define a set of macro names with specific values for each name.

Some examples of using parameters to create labels are:

```
parameter [2:0] WAITE = 3'b001,
               LOAD  = 3'b010,
               DONE   = 3'b100;

reg [2:0] state, next_state;
```

Observe that, when using parameters, the `state` and `next_state` variables are general purpose variables of the **reg** type, instead of enumerated variables. These general variables are loosely typed, meaning any value can be assigned to the variable. Loosely typed assignment rules are discussed in more detail in Chapter 5, section 5.14 (page 196). Using loosely typed assignment rules, the following assignment statements are legal assignments, but are functional bugs:

```
always @(posedge clk or negedge rstN) // async reset
  if (!rstN) state <= 0; // BUG: 0 is not a valid state
  else           state <= next_state;
```

This coding mistake would have been a syntax error with an enumerated type variable. Using the traditional Verilog style of parameters and general purpose variable types does not prevent inadvertent coding mistakes such as this.

4.5 Structures

A structure is used to group together multiple variables under a common name. Designs often have logical groups of signals, such as the control signals for a bus protocol, or the signals used within a state controller. Structures provide an efficient way to bundle together these related variables. All the variables in the structure bundle can be assigned values in a single assignment, or each variable can be assigned a value separately. The structure bundle can be copied to another structure bundle with the same definition, and passed through module ports or in and out of a task or function.

4.5.1 Structure declarations

A structure is declared using the **struct** keyword, similar to the C language. The **struct** keyword is followed by an opening curly brace ({), a list of variable declarations, a closing curly brace (}), and then a name for the structure.

```

typedef enum logic [2:0] {NOP, ADD, SUB, MULT, DIV} opcode_t;

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

```

A structure can bundle together any number of variable data types, including user-defined types. Parameter and localparam constants can also be included in a structure. A parameter in a structure cannot be redefined like parameters in modules. Parameters in structures are treated as localparams.

4.5.2 Assigning to structure members

The variables within a structure are referred to as *structure members*. Each member has a name, which can be used to select that member from the structure. A structure member is referenced using the name of the structure, followed by a period (.) and then the name of the member. This is the same syntax as in C. For example, to assign a value to the address member of the preceding structure, the reference is:

```

always_ff @(posedge clk)
    if (init) instruction_word.address = 32'hF000001E;
    else ...

```

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 constants that can be different types and sizes. Another difference is that the elements of an array are referenced by using an index into the array, whereas the members of a structure are referenced by using the member name.

4.5.3 Assigning to entire structures

An entire structure can be assigned a *structure expression*. A structure expression is formed by using a comma-separated list of values enclosed between the tokens '{ and } , which is the same way for assigning a set of values to an array, as discussed in section 3.7.3 (page 91) in Chapter 3. The braces must contain a value for each member of the structure. For example:

```

struct {
    logic [31:0] a, b;
    opcode_t opcode;
    logic [23:0] address;
    bit error;
} instruction_word;

always_ff @(posedge clk)
    if (init) instruction_word <= '{3, 5, ADD, 24'hC4, '0};
    else ...

```

The values in the structure expression must be listed in the order in which they are defined in the structure, as shown in the preceding example. 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. The member names within the structure expression are referred to as *tags*. When member names are specified, the expression list can be in any order.

```
instruction_word <= '{address:0, opcode:SUB,
a:100, b:7, error:'1};
```

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

Default values in structure expressions. A structure expression can specify a value for multiple members of a structure by specifying a *default value*. The default value is specified using the **default** keyword.

```
instruction_word <= '{default:0}; // set all members to 0
```

A structure expression can also contain a mixture of assignments to specific structure members with a default value for all other members.

```
instruction_word <= '{error:'1, default:0};
```

Enumerated types in structures. The previous two examples with default values have a semantic error. The default value assigned to structure members must be compatible with the data type of the member. Since most SystemVerilog variables are loosely typed, almost any default value will be compatible. Enumerated type variables, however, are more strongly typed. Assignments to an enumerated type variable must be either a label from its enumerated list, or another enumerated variable of the same enumerated type definition (enumerated type assignment rules are discussed in section 4.4.3, page 118).

The two assignment statements to `instruction_word` above attempt to assign `opcode` a default value of 0. This is an illegal value for `opcode`, which is an `opcode_t` enumerated type variable (the `typedef` definition for `opcode_t` is shown in section 4.5.1, page 124). When a member of a structure is an enumerated type variable, the structure expression must specify a legal explicit value for that member. A default value can be specified for all other members. For example:

```
always_ff @(posedge clk)
  if (!rstN) instruction_word <= '{opcode:NOP, default:0};
  else    ...
```

4.5.4 Typed and anonymous structures

User-defined types can be created from structures, using the `typedef` keyword, as discussed in section 4.1 (page 101). Declaring a structure as a user-defined type does not allocate any storage. A net or variable of that user-defined type must be declared before the structure can be used.

```

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

instruction_word_t iw_var; // variable of the structure type
wire instruction_word_t iw_net; // net of the structure type

```

A structure that is declared without using **typedef** is referred to as an *anonymous structure*. A structure that is declared using **typedef** is referred to as a *typed structure*. Both anonymous and typed structures can be defined within a module, but these local definitions can only be used within that module. A typed structure can also be defined in a package, and imported into the design blocks that require the structure definition. A typed structure defined in a package can be used in multiple modules and the verification testbench.

4.5.5 Copying structures

A typed structure can be copied to another typed structure, provided the two structures are declared from the same typed structure definition. The following example uses the structure definition and declarations shown above, in section 4.5.4.

```

always_ff @(posedge clk)
    if (!rstN) iw_var <= '{opcode:NOP, default:0};
    else      iw_var <= iw_net; // copy iw_net structure

```

An anonymous structure cannot be copied as a whole, but can be copied one member at a time.

4.5.6 Packed and unpacked structures

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.

A structure can be explicitly declared as a *packed* structure by using the keyword pair **struct packed**.

```

struct packed {
    logic      valid;
    logic [ 7:0] tag;
    logic [31:0] data;
} data_word;

```

A packed structure stores all members of the structure as contiguous bits, in the same form as a vector. The first member of the structure is 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 4-2.

Figure 4-2: Packed structures are stored as a vector

valid	tag	data
40	39	31

15

0

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.

Referencing packed structures and structure members. Packed structures can be copied, or assigned a structure expression value list, in the same way as unpacked structures. The members of a packed structure can be referenced by the member name, in the same way as an unpacked structure.

Packed structures can also be treated as a vector. Therefore, in addition to structure assignments, vector values can be assigned to packed structures.

```
data_word = 40'h100DEADBEEF;
```

The vector assignment is legal because the members of the structure on the left-hand side of the assignment have been packed together to form a contiguous set of bits, in the same way as a vector.

Because a packed structure is stored as a contiguous set of bits, it is also legal to do vector operations on packed structures, including bit selects and part selects. 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
```

Math operations, logical operations, and any other operation that can be performed on vectors can also be performed on packed structures.

Signed packed structures. 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 with part selects of vectors.

```

typedef struct packed signed {
    logic valid;
    logic [ 7:0] tag;
    logic signed [31:0] data;
} data_word_t;

data_word_t d1, d2;

always_comb begin
    lt = 0; gt = 0;
    if (d1 < d2) lt = '1; // signed comparison
    else if (d1 > d2) gt = '1;
end

```

4.5.7 Passing structures through ports and to tasks and functions

Typed structures can be passed through module and interface ports. The `typedef` definition should be in a package, so that the definition is available for use as a module port type.

```

package definitions_pkg;
    typedef enum logic [2:0] {ADD, SUB, MULT, DIV} opcode_t;

    typedef struct {
        logic [31:0] a, b;
        opcode_t opcode;
        logic [23:0] address;
    } instruction_word_t;
endpackage: definitions_pkg

module alu
    import definitions_pkg::*;

    (input instruction_word_t iw, // user-defined port type
     input wire clk
    );
    ...
endmodule

```

An unpacked structure must be a typed structure in order to pass the structure through ports. The connections to the port must be a structure of the exact same type as the port. That is, both the port and the connections on both sides of the port must all be declared from the same `typedef` definition. This restriction only applies to unpacked structures. A packed structure passed through a module port is treated like a vector. The external connection to the port can be a packed structure of the same type, or any type of vector.

Typed structures can also be passed as arguments to a task or function by declaring the task or function argument as the structure type.

```

module processor (...);

...
typedef enum logic [2:0] {ADD, SUB, MULT, DIV} opcode_t;

typedef struct { // typedef is local to this module
    logic [31:0] a, b;
    opcode_t      opcode;
    logic [23:0] address;
} instruction_word_t;

function calculate_result (input instruction_word_t iw);
    ...
endfunction: calculate_result
endmodule: processor

```

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. A packed structure formal argument is treated as a vector, and can be passed to any type of vector.

4.5.8 Traditional Verilog versus structures

The original Verilog language did not have a convenient mechanism for collecting common signals into a group. In legacy Verilog style models, engineers had to use ad-hoc grouping methods such as naming conventions, where each signal in a group starts or ends with a common set of characters. The original Verilog language also did not have a way to pass a collection of signals through module ports or to tasks and functions. Each signal had to be passed through a separate port or argument.

The addition of structures to the original Verilog language is a powerful and versatile RTL modeling construct. It provides a way to model complex model functionality more concisely and in a more intuitive and more re-usable form. A typed structure defined in a package can be re-used in several modules, as well as in the verification testbenches used to verify the RTL models.

4.5.9 Synthesis considerations

Both anonymous and typed structures, and both unpacked and packed structures are synthesizable. Synthesis supports passing structures through module ports, and in to, or out of, tasks and functions. Assigning values to structures by member name and as a list of values is supported.

Synthesis compilers might be able to optimize unpacked structures better than packed structures. Unpacked structures permit software tools to determine the best way to store or implement each structure member, whereas packed structures dictate how each member is to be organized.

4.6 Unions

A *union* is a single storage element that can have multiple data type representations. The declaration of a union is similar to a structure, but the inferred hardware is very different. A structure is a collection of several variables. A union is a single variable, that can a data type at different times. The variable types a union can store are listed between curly braces ({ }), with a name for each variable type.

```
union {
    int s;
    int unsigned u;
} data;
```

The variable is `data`, in this example. The `data` variable has two possible data types: a signed integer type named `s`, or an unsigned integer value named `u`.

A typical application of unions in RTL modeling is when a value might be represented as several different types, but only as one type at any specific clock cycle. For example, a data bus might sometimes carry a packet of data using the User Network Interface (UNI) telecommunications protocol. At other times, the same data bus might carry a packet of data using the Network to Network Interface (NNI) telecommunications protocol. A SystemVerilog union can represent this dual usage of the same bus. Another usage of unions is to represent a shared hardware resource, such as a hardware register that can store different types of data at different times.

4.6.1 Typed and anonymous unions

A union can be defined as a type using **typedef**, in the same way as a structure. 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*.

```
union {                                     // anonymous (untyped) union
    int s;
    int unsigned u;
} data; // a variable called data

typedef union {                         // typed union
    int s;
    int unsigned u;
} data_t; // a user-defined type

data_t data_in, data_out; // two variables of type data_t
```

Both anonymous and typed unions are synthesizable, but typed unions have advantages for RTL modeling. Typed unions can be:

- Used to declare multiple variables, such as `data_in` and `data_out` in the preceding example.
- Used as a module port type.
- Defined in a package and then used in multiple modules.

4.6.2 Assigning to, and reading from, union variables

The data type of a union is referenced using the name of the union followed by the name that represents the data type, separated by a period (.).

```
union {
    int s;
    int unsigned u;
} data;

data.s = -5;
$display("data.s is %d", data.s);

data.u = -5;
$display("data.u is %d", data.u);
```

In this example, the variable `data` has two possible data types, and a value of negative 5 is stored in each representation. The `data.s` data type will print as -5, a signed integer value. The `data.u` data type will print the same value as 4294967291, an unsigned integer value.

4.6.3 Unpacked, packed and tagged unions

Best Practice Guideline 4-4

Only use packed unions in RTL models.

SystemVerilog has three types of unions: *unpacked unions*, *packed unions* and *tagged unions*. Most synthesis compilers only support packed unions.

Unpacked and tagged unions are not supported by most synthesis compilers. These union types can represent storage for any data type, including data types that are not synthesizable. Unpacked unions and tagged unions can be useful for modeling test-benches and high-level abstract models, but should not be used for RTL modeling.

Packed unions are declared by adding the keyword **packed** immediately after the **union** keyword.

```
typedef union packed {           // packed union type
    int s;
    int unsigned u;
} data_t;
```

Packed unions are synthesizable. A packed union places a number of restrictions on the data types that a union can represent. These restrictions align closely with hardware behavior. In a packed union can only represent vector types, and the vector width must be the same for each data type the union can store. This ensures that a packed union will represent its storage with the same number of bits, regardless of the data type in which a value is stored.

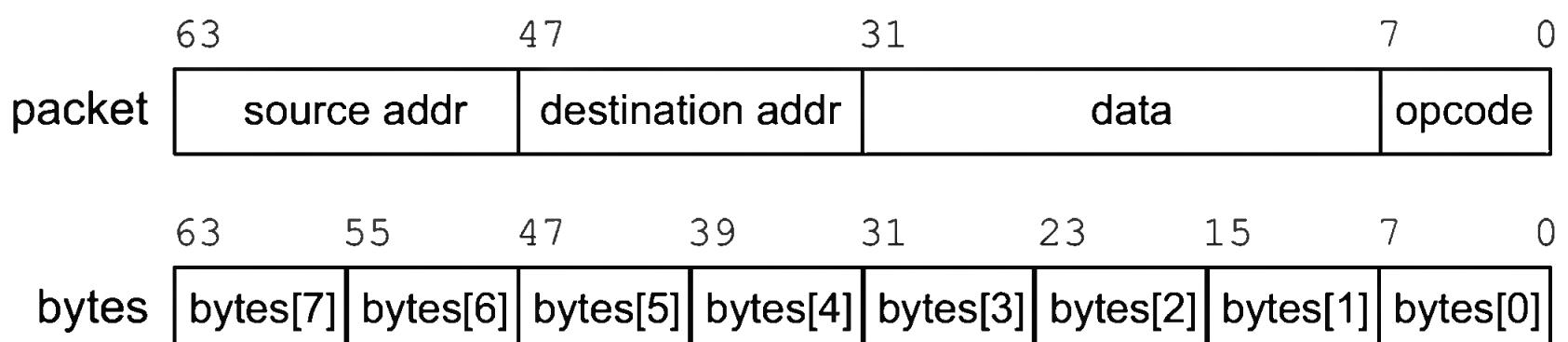
A packed union allows data to be written using one format, and read back using a different format. The design model does not need to do any special processing to keep track of how data was stored. This is because the data in a packed union will always be stored using the same number of bits. The following 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 contiguous 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 4-3 illustrates how the two data types of `dreg` are represented.

Figure 4-3: 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 `bytes` format (perhaps from a serial input stream of bytes), and then the same value can be read using the `data_packet` format.

```
always_ff @ (posedge clk, negedge rstN) // async reset
    if (!rstN) begin // active-low reset
        dreg.packet <= 0; // reset using packet type
        i <= 0;
    end
    else if (load_data) begin
        dreg.bytes[i] <= data_in; // store using bytes type
        i++;
    end
    else if (data_ready) begin
        case (dreg.packet.opcode) // read as packet type
            //...
        endcase
    end
```

4.6.4 Passing unions through ports and to tasks and functions

Typed unions (unions defined using `typedef`) can be used as the data type for module ports and task/function arguments. An unpacked union requires that the same union type be used for the external connection to a port, or the external signal passed to a task or function argument. Packed unions can only represent packed data types, which allows any vector type to be used for the external connection or external value.

Examples 4-6 shows a package that contains definitions for a structure and union. Example 4-7 uses this package in a model of a simple Arithmetic Logic Unit (ALU) that can operate on either signed or unsigned values, but not both at the same time. A flag is used to indicate if the operation data is signed or unsigned. The ALU opcode, the two operands, and a signedness flag are passed into the ALU as a single instruction word, represented as a structure. The ALU output is a single value that can represent either a signed or an unsigned value, modeled as a union of these two types. This allows the same output port to be used for different data types.

Example 4-6: Package containing structure and union definitions

```

`define _4bit           // use 4-bit data for testing synthesis
//`define _32bit         // use 32-bit data word size
//`define _64bit         // use 64-bit data word size
package definitions_pkg;
  `ifdef _4bit
    typedef logic      [ 3:0] uword_t;
    typedef logic signed [ 3:0] sword_t;
  `elsif _64bit
    typedef logic      [63:0] uword_t;
    typedef logic signed [63:0] sword_t;
  `else // default is 32-bit vectors
    typedef logic      [31:0] uword_t;
    typedef logic signed [31:0] sword_t;
  `endif

  typedef enum logic [2:0] {ADD, SUB, MULT, DIV} op_t;
  typedef enum logic {UNSIGNED, SIGNED} operand_type_t;

  // Packed union represents a variable that can store
  // different types
  typedef union packed {
    uword_t u_data;
    sword_t s_data;
  } data_t;

  // Packed structure represents a collection of variables
  typedef struct packed {
    op_t          opcode;
    operand_type_t op_type;
    data_t        op_a;
    data_t        op_b;
  } instr_t;

```

```
endpackage: definitions_pkg
```

Example 4-7: Arithmetic Logical Unit (ALU) with structure and union ports

```
module alu
  import definitions_pkg::*;

  input logic clk, rstN,
  input instr_t iw,           // input is a structure
  output data_t alu_out    // output is a union
);

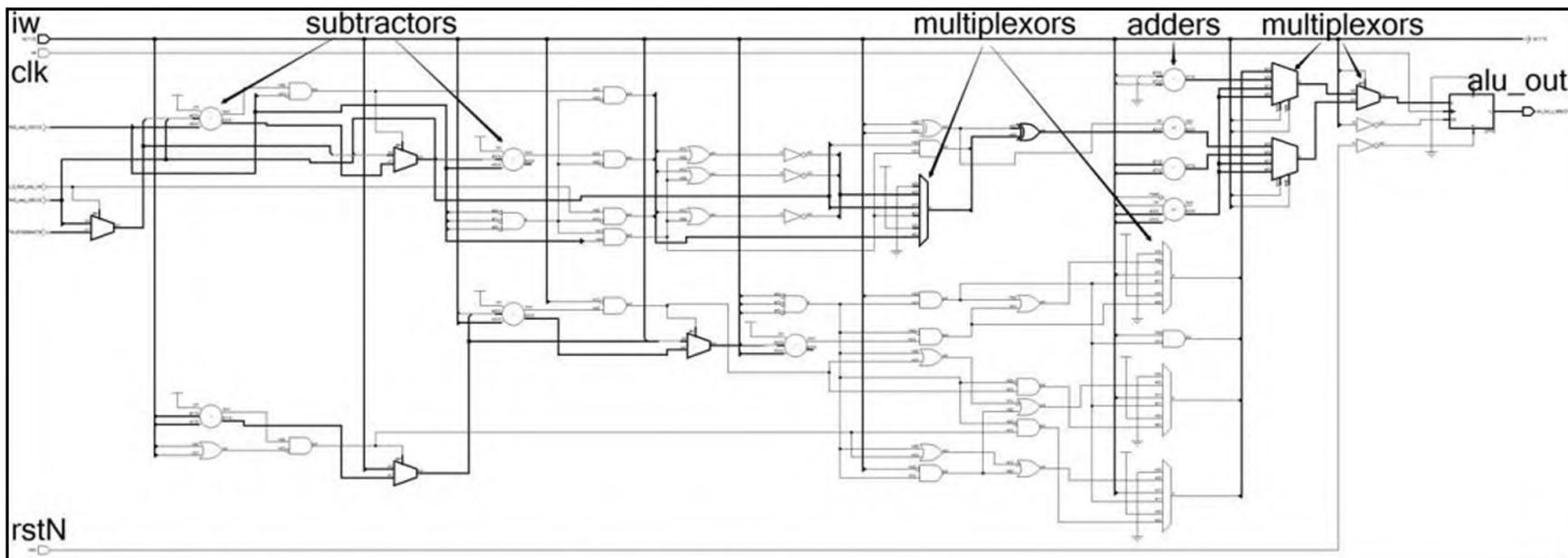
  always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN)                                // active-low
      alu_out <= '0;
    else begin: alu_operations
      if (iw.op_type == SIGNED) begin: signed_ops
        case (iw.opcode)
          ADD : alu_out.s_data <= iw.op_a.s_data
                  + 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;
        endcase
      end: signed_ops
      else begin: unsigned_ops
        case (iw.opcode)
          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;
        endcase
      end: unsigned_ops
    end: alu_operations
endmodule: alu
```

Figure 4-4 shows the result of synthesizing this example. The schematic image is too small to be meaningful because the page size of this book, but illustrates two important characteristics of using structures and unions in RTL models:

- Structures and unions can concisely model a significant amount of functionality. The ability to model more functionality with fewer lines of code is one of the reasons features such as structures and unions were added to the original Verilog.

- Unions, when used with the RTL coding guidelines described in this section, can represent multiplexed functionality, allowing multiple resources (signed and unsigned adders, subtracters, multipliers and dividers in this example) to share the same hardware registers. The circles in Figure 4-4 represent generic arithmetic operations, The trapezoidal symbols represent multiplexors.

Figure 4-4: Synthesis result for Example 4-7: ALU with structure and union ports



Technology independent schematic (no target ASIC or FPGA selected)

4.7 Using arrays with structures and unions

Structures and unions can include packed or unpacked arrays. A packed structure or union can only include packed arrays.

```
typedef struct {           // unpacked structure
    logic data_ready;
    logic [7:0] data [0:3];      // unpacked array
} packet_t;

typedef struct packed {    // packed structure
    logic parity;
    logic [3:0][ 7:0] data;    // 2-D packed array
} data_t;
```

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.

```
packet_t p_array [23:0]; // unpacked array of 24 structures
data_t [23:0] d_array;   // packed array of 24 structures
```

Arrays can contain typed structures and typed unions. Synthesis supports both packed or unpacked structures in arrays.

Example 4-8 illustrates using an array of structures. The example is a model of an instruction register that contains an unpacked 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 include separate operands, an opcode, and a flag indicating if the operands are signed or unsigned. The model loads these separate input values into an instruction register array. A write pointer input controls where the data is loaded. The output of the model is a single instruction structure, selected from the instruction register using a read pointer input.

This example uses the same package items shown previously in Example 4-6 (page 134).

Example 4-8: Using arrays of structures to model an instruction register

```

module instruction_register
  import definitions_pkg::*;
  (input logic clk, rstN, load_en,
   input data_t op_a,
   input data_t op_b,
   input operand_type_t op_type,
   input op_t opcode,
   input logic [4:0] write_pointer,
   input logic [4:0] read_pointer,
   output instruction_t iw
  );
  instruction_t iw_reg [0:31]; // array of structures

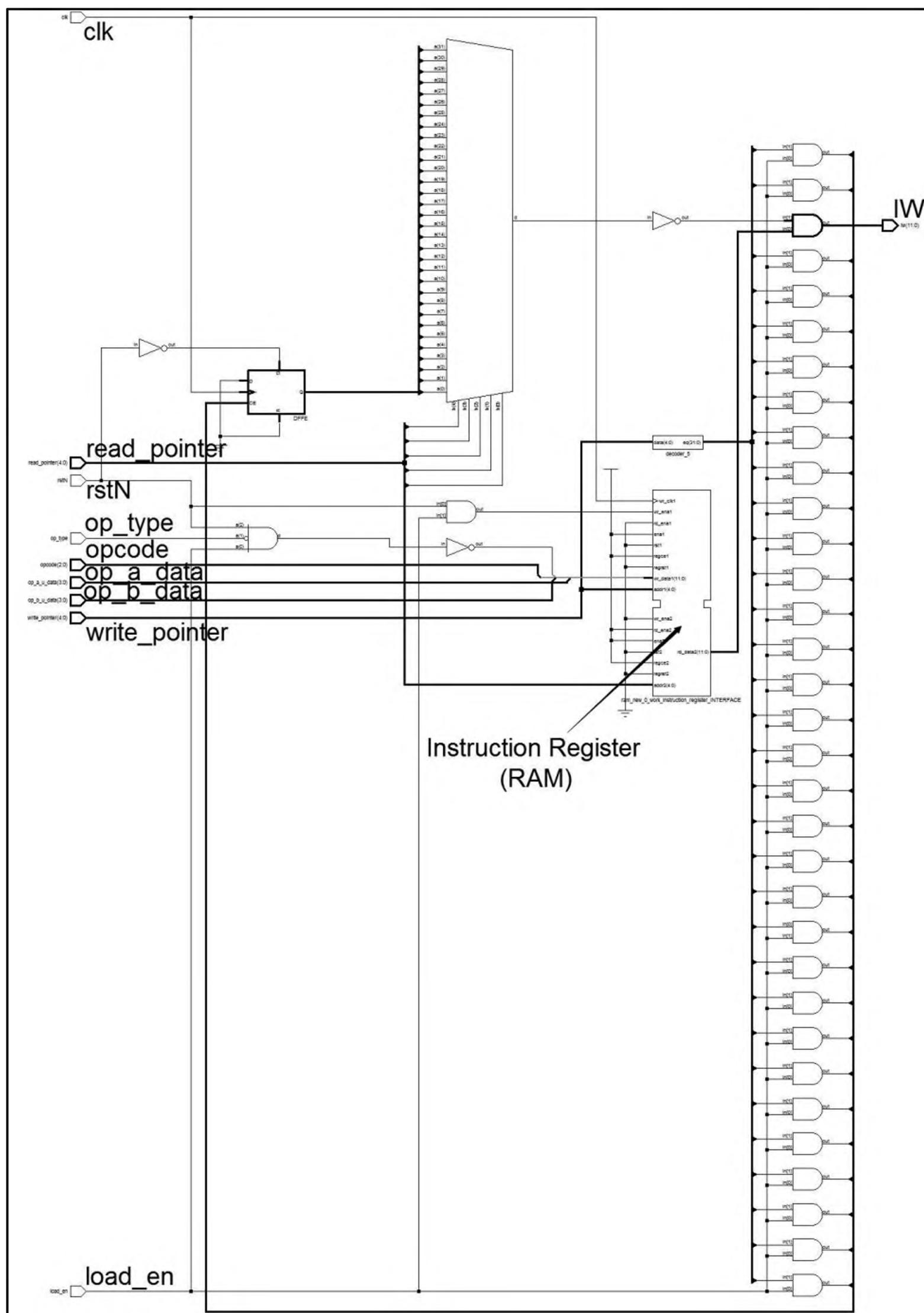
  // write to the register array
  always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN) begin // active-low reset
      foreach (iw_reg[i])
        iw_reg[i] <= '{opcode:ADD,default:0}; // reset values
    end
    else if (load_en) begin
      case (op_type)
        SIGNED: iw_reg[write_pointer] <=
          '{opcode,op_type,op_a.s_data,op_b.s_data};
        UNSIGNED: iw_reg[write_pointer] <=
          '{opcode,op_type,op_a.u_data,op_b.u_data};
      endcase
    end
    // read from the register array
    assign iw = iw_reg[read_pointer];
endmodule: instruction_register

```

Figure 4-5 shows the result of synthesizing this example. The schematic image is too small to be readable in the page size of this book, but illustrates how structures and unions, arrays can be used to model a significant amount of design functionality with very few lines of code. The rectangular symbol towards the upper-right of the schematic is an instance of a generic RAM that the synthesis compiler chose to repre-

sent the storage of the array in the RTL model. The synthesis compiler will implement this generic RAM as one or more synchronous storage devices in the final step of synthesis, where the generic gate-level functionality is mapped to a specific ASIC or FPGA device.

Figure 4-5: Synthesis result for Example 4-8: instruction register with structures



Technology independent schematic (no target ASIC or FPGA selected)

4.8 Summary

The topics presented in this chapter provide powerful ways to manage complex design data in a concise, maintainable and reusable form.

User-defined types, which are declared with a C-like `typedef` definition, allow users to define new types built up from the predefined types or other user-defined types. User-defined types can be used as module ports and passed in or out of tasks and functions.

Declaration packages provide a place to define user-defined types, tasks and functions that can be shared throughout the design and verification code. Shared definitions reduce redundant code in large projects and help ensure consistency throughout a project. Using packages makes code easier to maintain and easier to reuse.

Enumerated types allow the declaration of variables with a defined set of valid values. The valid values are represented with abstract labels instead of hardware-centric logic values. Enumerated types allow modeling at 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 one-hot encoding values to the enumerated labels.

Structures make it possible to bundle several variables together, and work with the complete bundle, while still being able to work with the individual variables. The structure bundles can be copied, assigned a list of values, passed through module ports, and passed into or out of tasks and functions.

Unions give a high-level coding style for modeling shared resources in a design, such as a data bus that can transport different data protocols at different times, or a register that can store different types of data at different times.

All of the topics discussed in this chapter were added to the original Verilog language as part of the newer SystemVerilog generation of the language.

* * *

Chapter 5

RTL Expression Operators

Abstract — Chapter 5 explores the programming operators that are used for RTL simulation and synthesis. Operators evaluate one or more expressions and determine a result. For example, the arithmetic + operator adds two expressions together and returns the sum. The operation could be performed as unsigned integer, signed integer, or floating-point, with or without a carry, and with a 2-state or 4-state result. Understanding the rules of SystemVerilog operators is essential for writing RTL models that simulate and synthesize correctly. The topics covered in this chapter include:

- 2-state and 4-state operations
- X-optimism and X-pessimism
- Expression vector sizes
- Concatenate and replicate operators
- Conditional (ternary) operator
- Bitwise operators
- Unary reduction operators
- Logical operators
- Comparison operators (equality and relational)
- Case equality (identity) operators
- Set membership (inside) operator
- Shift operators
- Streaming operators (pack and unpack)
- Arithmetic operators
- Increment and decrement operators
- Assignment operators
- Cast operators

5.1 Operator expression rules

Operators perform operations on *operands*. Most operators have two operands. For example, in the operation `a + b`, the operands of the `+` (add) operation are `a` and `b`. Each operand is referred to as an *expression*. An expression can be a literal value, a variable, a net, the return of a function call, or the result of another operation. Expressions have a number of characteristics which affect how an operation is performed. These characteristics are discussed in sections 5.1.1 through 5.1.5.

5.1.1 4-state and 2-state operations

Expressions can be either 2-state or 4-state. A *2-state expression* can only have values of 0 or 1 in each bit. Two state expressions cannot have hi-impedance values (represented by the letter Z), unknown values (represented by the letter X) or don't-care values (also represented by X). A *4-state expression* is able to contain the values of 0, 1, Z or X in any bit of the expression.

The rule for operations on 2-state and 4-state expressions is simple — when any operand is a 4-state expression, the result of the operation will be a 4-state expression. All operands must be 2-state expressions in order to have a 2-state result. This rule can affect other operations if the result is then used as an operand of another operation or is evaluated in a programming statement, such as an if-else decision.

Chapter 3 discusses SystemVerilog's 2-state and 4-state data types in more detail. The coding guideline recommended in that chapter is to only use 4-state types for RTL modeling. A primary reason for this is that an X in any bit in the result of an operation can be a good indication that there is a problem with one of the operands. When 2-state types are used, design problems can be hidden because there is no X values in the operation results to indicate a potential bug.

5.1.2 X-optimism and X-pessimism

Most SystemVerilog operators are *X-optimistic*, meaning the operation might produce a known result even if there are X or Z values in the operands. A few SystemVerilog operators, such as arithmetic and relational operators, are *X-pessimistic*, meaning that all bits of the result will automatically be an X, if any operand has any bit with an X or Z value.

X-optimistic operations. An X-optimistic operations can produce a valid result even when one or both operands have bits with X or Z values. Consider the following example and the logic values shown:

```
logic [3:0] a, b, result;
assign a = 4'b01zx;           // some bits are X or Z
assign b = 4'b0000;           // all bits of are zero

assign result = a & b;        // bitwise AND a with b
```

The result of the operation is the value 4'b0000. This is because the & operator models a digital AND logic gate for each bit of its operands. In digital logic, a 0 ANDed with any value will result in a 0. The high-impedance bit (represented by a Z) and the unknown bit (represented by an X) in operand a become zeros in the result because these bits are ANDed with their corresponding bits in b, which have a value of 0. This behavior is referred to as X-optimism. Simulation will have a known result, even though an operand has bits with X or Z values.

X-optimism only applies to values where simulation can accurately predict how actual logic gates behave. In the following example, the b operand is all ones instead of all zeros.

```
assign a = 4'b01zx;           // some bits are X or Z
assign b = 4'b1111;           // all bits of are one

assign result = a & b;         // bitwise AND a with b
```

The result of the operation is the value 4'b01xx. With these operand values, X-optimism does not apply for the two right-most bits. In actual logic gates, a 1 ANDed with high-impedance could result in either a 0 or 1. Which value depends on a number of conditions, such as the type of transistors used to build the AND gate, the impedance and capacitance of the transistor circuit, operating voltage, and even ambient temperature. The abstract RTL AND operator does not have this detailed information. Without those details, digital simulation cannot predict whether a 0 or 1 will result from a 1 ANDed with Z. Similarly, an X represents an unknown value, meaning the actual logic gates value could be 0, 1 or Z. With this ambiguity, digital simulation cannot predict whether a 0 or 1 will result from a 1 ANDed with X.

X-pessimistic operations. A small number of SystemVerilog operators are more pessimistic. If any operand has any bit with an X or Z value, the operation automatically returns a value with all bits set to X. The pessimistic operators arithmetic operators, such as the add operator, and relational operators, such as the less-than operator.

```
logic [3:0] a, b, result;

assign a = 4'b000x;           // some bits are X or Z
assign b = 4'b0101;           // a value of 5 in decimal

assign result = a + b;         // add the values of a and b
```

The result of the operation is 4'bxxxx. This X-pessimism occurs because the arithmetic add operator performs numeric-based addition, rather than a bit-by-bit addition. The value of operand a is 4'b000x, which is not a number. Therefore the result of the operation is an unknown value.

Sections 5.4 through 5.15 of this chapter examine the SystemVerilog RTL operators in greater detail, and whether each operator is X-optimistic or X-pessimistic. These effects are critical to understand in order to write RTL models that correctly represent the desired hardware behavior.

5.1.3 Expression vector sizes and automatic vector extension

Each operand of an operator can be any vector size, including scalar (1-bit). The vector size of the operands can affect how the operation is performed. An important consideration is when the operands of an operator are different vector sizes.

Self-determined operands. Some operators treat each operand independently. It does not matter if the operands are different vector sizes. The operands of these operators are referred to as *self-determined*. In the following example, the `&&` operator performs a logical AND operation, which tests to see if both operands are true. If they are, the operation returns a result of true. Otherwise, the operation returns a false.

```
logic [15:0] a;
logic [31:0] b;
logic       result;

assign result = a && b; // test a and b for true or false
```

Operands `a` and `b` are self-determined. Each operand can be evaluated to be true or false, independent of the vector size of the other operand.

Context-determined operands. Many operators need to first expand the operands to be the same vector size before the operation can be performed. The operands of these operators are referred to as *context-determined*. The operation will left-extend the shortest operand to be the same vector size as the largest operand. In the following example, the `&` operator performs a bitwise AND operation, which ANDs each bit of each operand together, and returns a Boolean result.

```
logic [ 7:0] a;           // 8-bit variable
logic [15:0] b;           // 16-bit variable
logic [15:0] result;      // 16-bit variable

assign result = a & b;    // 16-bit operation
```

In order to AND each bit of `a` with each bit of `b`, the operation will adjust the two operands to the same vector width. The operation will examine the context of the operation to determine the largest operand, and then left-extend the shorter operand to match the size of the largest operand. The extension rules are:

- If the left-most bit is 0 or 1, and the operand is an unsigned type, then the operand is zero-extended (each additional bit is given the value of 0).
- If the left-most bit is 0 or 1, and the operand is a signed type, then the operand is sign-extended (each additional bit is given the value of the left-most bit or the original value. That left-most bit is referred to as the *sign bit*).
- If the left-most bit is Z, then the operand is Z-extended (each additional bit is given the value of Z).
- If the left-most bit is X, then the operand is X-extended (each additional bit is given the value of X).

The size context for arithmetic operations is more complex than that of other operators. The size context takes into account not only the operands of the operator, but also the vector size of all expressions on both the right-hand side and left-hand side of an assignment statement, as shown in the following code:

```
logic [ 7:0] a;           // 8-bit variable
logic [15:0] b;          // 16-bit variable
logic [23:0] result;     // 24-bit variable

assign result = a + b;   // 24-bit operation
```

5.1.4 Signed and unsigned expressions

The arithmetic, comparison and shift operators can perform either signed or unsigned operations. The rule is simple — if all operands on which the operation is performed are signed, then a signed operation is performed. If any of the effected operands are unsigned, then an unsigned operation is performed. The following code snippets illustrate these rules. The type of operation is noted in the comments.

```
logic      [15:0] a, b, u1, u2; // unsigned types
logic signed [15:0] c, d, s1, s2; // signed types

assign u1 = a + b;           // unsigned operation
assign s1 = a + c;           // unsigned operation
assign u2 = c + d;           // signed operation
assign s2 = c + d + a;        // unsigned operation
```

The operation signedness is determined solely by the operands of the operator. It is not affected by the signedness of the left-hand side of the assignment statement.

5.1.5 Integer (vector) and real (floating-point) expressions

All SystemVerilog operators can perform operations on integer values. The IEEE SystemVerilog standard refers to integer values as *integral expressions* — a value that is comprised of one or more contiguous bits. Engineers often refer to these integer or integral values as *vectors*.

SystemVerilog refers to fixed-point and floating-point expressions as *real expressions*. Most types of operations can be performed on real expressions, including: assignment operations, arithmetic operations, logical (true/false) operations, comparison operations and increment/decrement operations. There are a few operations that cannot be performed on real expressions. These are operations that work with bits of a vector, such as bit and part select operations, bitwise operations, shift operations, concatenate operations and streaming operations.

Operations can be performed on a mix of integer and real expressions. The rule is for mixed type operations is that if any operand is a real expression, then the other operand is converted to a real expression, and a floating-point operation is performed.

NOTE

RTL synthesis compilers typically do not support real (floating-point) expressions. High-level Synthesis (HLS) tools can be used for complex arithmetic design. Floating point and fixed point design is outside the scope of this book on RTL modeling.

5.2 Concatenate and replicate operators

The concatenate and replicate operators join multiple expressions together to form a vector expression. The total number of bits in the resultant vector is the sum of all the bits in each sub expression. There are two forms of concatenations, simple and replicated. A simple concatenate joins any number of expressions together. A replicated concatenation joins expressions together and then replicates that result a specified number of times. Table 5-3 shows the general syntax and usage of the concatenate and replicate operators.

Table 5-1: Concatenate and replicate operators for RTL modeling

Operator	Example Usage	Description
{ }	{m, n}	Join m and n together as a vector
{r{ }}	{r{m, n}}	Join m and n together, and replicate r times; r must be a literal integer value. it cannot be a parameter.

The following variables and values are used to show the results of these operators.

```
logic [3:0] a = 4'b1011;
logic [7:0] b = 8'b00010001;
```

Given these values:

- {a,b} results in 101100010001 (binary), a 12-bit value.
- {4'hF,a} results in 11111011 (binary), an 8-bit value.
- {8{2'b10}} results in 10101010101010 (binary), a 16-bit value, with the 2-bit pattern 01 repeated 8 times.
- { {4{a[3]}}, a } results in 11111011 (binary), an 8-bit value with the most-significant bit of a repeated 4 times, and then concatenated to a.

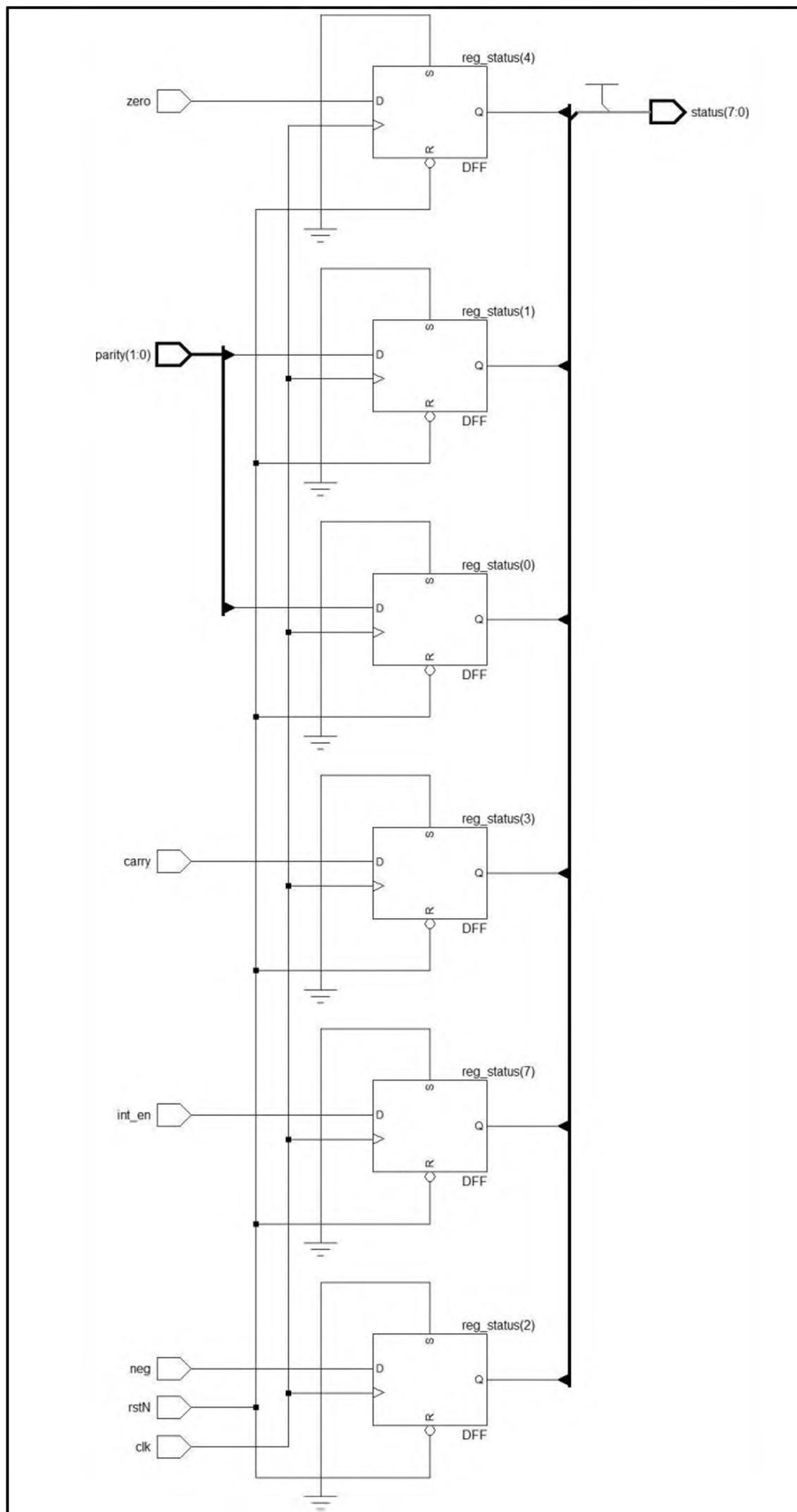
The concatenate and replicate operators are synthesizable. The operators do not directly represent any logic functionality in hardware. They simply represent using multiple signals together, appending literal values to a signal, or a literal value.

Examples 5-1 and 5-2 illustrate two common applications of the concatenate operator in RTL modeling: joining multiple signals together on the right-hand or left-hand side of an assignment statement. Following each example, Figures 5-1 and 5-2 show how the concatenate operators disappear in the gate-level functionality generated by synthesis. Nevertheless, the concatenate operators are a useful construct for representing hardware functionality in a concise way in RTL models.

Example 5-1: Using concatenate operators: multiple input status register

```
//  
// 8-bit status register that stores multiple input values  
  
// -----  
// | int_en | unused | unused | zero | carry | neg | parity |  
// -----  
// NOTE: not-used bits are set to a constant 1  
  
module status_reg  
(input logic clk, // register clock  
 input logic rstN, // active-low reset  
 input logic int_en, // 1-bit interrupt enable  
 input logic zero, // 1-bit result = 0 flag  
 input logic carry, // 1-bit result overflow flag  
 input logic neg, // 1-bit negative result flag  
 input logic [1:0] parity, // 2-bit parity bits  
 output logic [7:0] status // 8-bit status register output  
);  
  
always_ff @(posedge clk or negedge rstN) // async reset  
 if (!rstN) // active-low reset  
   status <= {1'b0,2'b11,5'b0}; // reset  
 else  
   status <= {int_en,2'b11,zero,carry,neg,parity}; // load  
  
endmodule: status_reg
```

Figure 5-1: Synthesis result for Example 5-1: Concatenate operator (status register)



Technology independent schematic (no target ASIC or FPGA selected)

NOTE

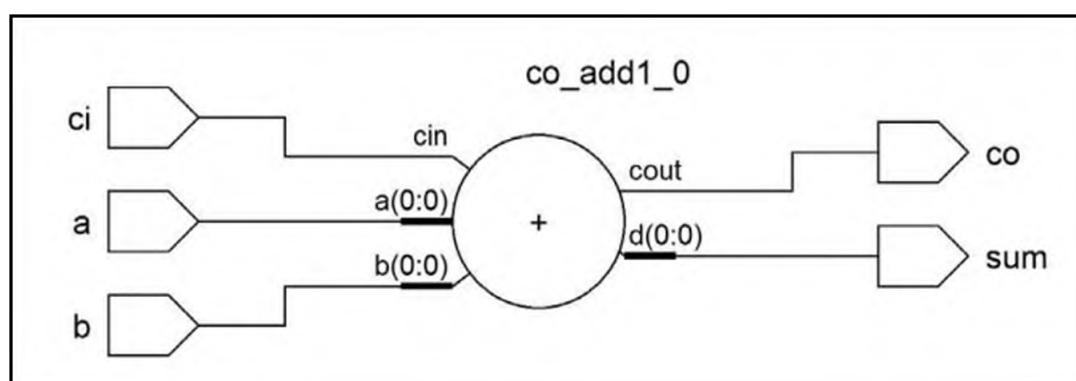
How synthesis compilers implement an operator can be influenced by a number of factors, including: the target device, other operators or programming statements used in conjunction with the operator, the synthesis compiler utilized, and the synthesis options and constraints that were specified.

The status register in Example 5-1 has two unused bits, which have a constant value of 1. The synthesis compiler used to generate the implementation of the status register shown in Figure 5-1 mapped these two unused bits to a simple pull up value on the 8-bit output. Other synthesis compilers, or specifying different synthesis constraints, might map this same RTL functionality differently, such as by using flip-flops that are preset to a value of 1.

Example 5-2: Using concatenate operators: adder with a carry bit

```
module rtl_adder
  (input logic a, b, ci,
   output logic sum, co
  );
  assign {co,sum} = a + b + ci;
endmodule: rtl_adder
```

Figure 5-2: Synthesis result for Example 5-2: Add operator (adder with carry in/out)



Technology independent schematic (no target ASIC or FPGA selected)

The synthesis compiler used to generate the implementation shown in Figure 5-2 mapped the RTL adder functionality to a generic adder block. The next step in synthesis would be to target a specific ASIC or FPGA device. The generic adder would be mapped to a specific adder implementation during that step.

The concatenate and replicate operators are frequently used to create expressions that are used as operands to other operators. Examples of this will be seen in later sections of this chapter and in subsequent chapters.

There are some important rules to be aware of when using concatenations:

- Any number of expressions can be concatenated together, including just a single expression.
- The expressions within a concatenation must have a fixed size. Unsized literal values, such as the number 5 or '1, are not permitted.
- The result of a concatenation is always unsigned, regardless of the signedness of the expressions within the concatenation.

Don't confuse concatenations with assignment lists. SystemVerilog has an assignment list operator that is enclosed between '{' and '}' tokens. This operator is dis-

cussed in Chapters 3, section 3.7.3 (page 91) and 4, section 4.5.3 (page 125). Although the assignment list operator appears similar to a concatenate operator, the functionality is very different. The concatenate operator joins several values together to create a new, single value. The assignment list operator begins with an apostrophe ('), and is used to assign a collection of individual values to the individual elements of an array or the individual members of a structure.

5.3 Conditional (ternary) operator

A widely used operator in RTL modeling is the conditional operator, which is also referred to as a ternary operator. This operator is used to choose between two expressions. The tokens used to represent the conditional operator are listed in Table 5-2.

Table 5-2: Conditional (ternary) operator for RTL modeling

Operator	Example Usage	Description
? :	s? m : n	If s is true, select m; if s is false select n, otherwise perform a bit-by-bit comparison of m and n

The expression listed before the question mark (?) is referred to as the *control expression*. It can be a simple integral value (a vector of any size, including 1-bit) or the result of another operation that returns an integral value. For example:

```
logic      sel, mode, enableN;
logic [7:0] a, b, y1, y2;

assign y1 = sel ? a : b;
assign y2 = (mode & !enableN) ? a + b: a - b;
```

The control expression is evaluated as true or false based using the following rules:

- The expression is *true* if any bit is 1.
- The expression is *false* if all bits are 0.
- The expression is *unknown* if no bits are set and not all bits are 0, which can occur if there are some bits that are X or Z.

With 4-state values, it is possible for a control expression to be neither true nor false. In the following value, none of the bits are 1, but not all of the bits are 0.

```
4'b000z; // an expression that is neither true nor false
```

When the control expression is unknown, the conditional operator performs a bit-by-bit comparison of the two possible return values. If the corresponding bits are both 0, a 0 is returned for that bit position, and if the corresponding bits are both 1, a 1 is returned for that bit position. If the corresponding bits are different, or if either bit has an X or Z value, an X is returned for that bit position. The following example illustrates this simulation behavior.

Given the values:

```
sel = 1'bx;
a = 8'b01xz01xz;
b = 8'b11110000;
```

then the conditional operation:

```
sel ? a : b;
```

will return 8'bx1xx0xxx.

The conditional operator often behaves like a hardware multiplexor. Example 5-3 illustrates using the conditional operator to choose between two inputs to a register. Figure 5-3 shows the results from synthesizing this example. The conditional operator is mapped to four multiplexors, one for each bit of the 4-bit d1 and d2 inputs.

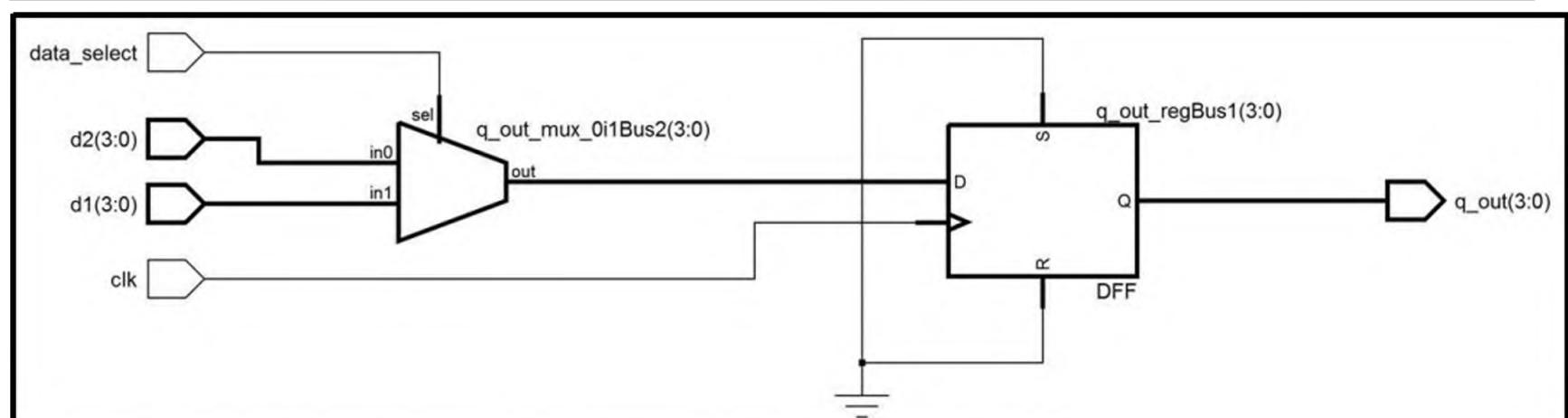
Example 5-3: Using the conditional operator: multiplexed 4-bit register D input

```
module muxed_register
#(parameter WIDTH = 4)                      // register size
(input logic                                     clk,           // 1-bit input
 input logic                                     data_select, // 1-bit input
 input logic [WIDTH-1:0] d1, d2,                 // scalable input size
 output logic [WIDTH-1:0] q_out                  // scalable output size
);

always_ff @ (posedge clk)
q_out <= data_select? d1 : d2;                // store d1 or d2

endmodule: muxed_register
```

Figure 5-3: Synthesis result for Example 5-3: Conditional operator (mux'ed register)



Technology independent schematic (no target ASIC or FPGA selected)

The circuit shown in Figure 5-3 is the intermediate generic synthesis result, before the synthesis compiler has mapped the circuit to a specific ASIC or FPGA target implementation. The synthesis compiler used to produce Figure 5-3 utilized generic flip-flops with unused set and reset inputs. The final implementation using an ASIC or FPGA library might be able to use flip-flops that do not have these inputs, if available in the target device. A different synthesis compiler might use different generic components to represent these intermediate results.

The conditional operator is not always implemented as a multiplexor. Synthesis compilers might map and optimize the conditional operator into other types of gate-level logic, based on the types of the operands and the context of the operation. In example 5-4, the conditional operator represents tri-state buffers, instead of multiplexed logic. Figure 5-4 shows the result of synthesizing this example.

Example 5-4: Using the conditional operator: 4-bit adder with tri-state outputs

```

module tri_state_adder
#(parameter N = 4)                                // N-bit adder size
(input logic enable, // output enable
 input logic [N-1:0] a, b, // scalable input size
 output tri logic [N-1:0] out // tri-state output, net type
);

assign out = enable? (a + b): 'z; // tri-state buffer

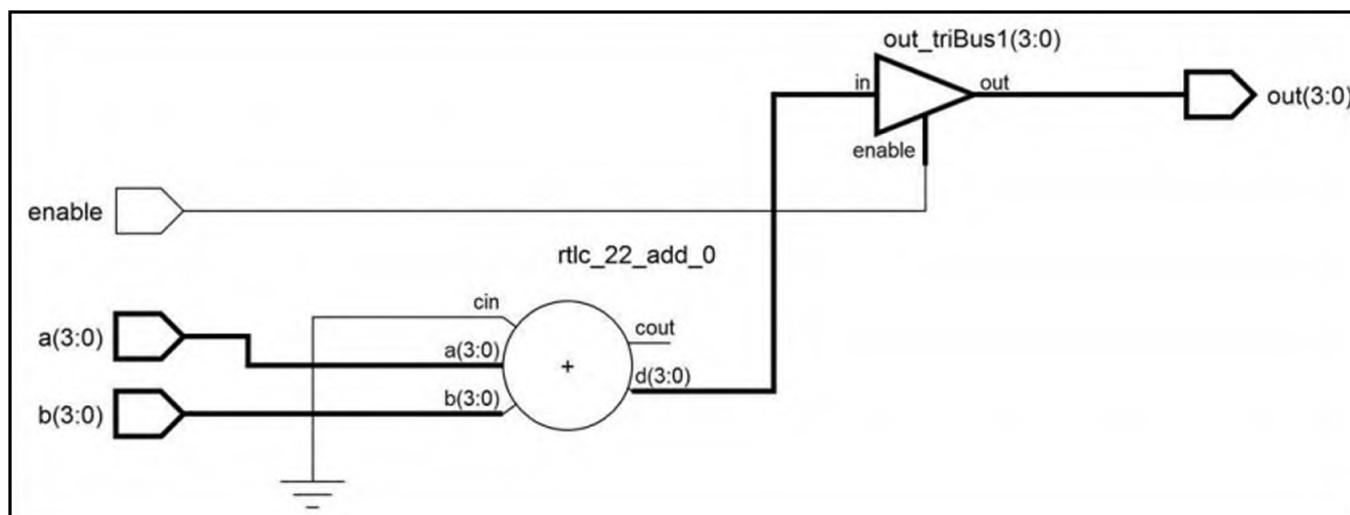
endmodule: tri_state_adder

```

In this example, the conditional operator (`? :`) selects whether the `out` port should be assigned the value of `(a + b)` or hi-impedance. If `en` is false, `out` is assigned '`z`'. The '`z`' token is a literal value that sets all bits of an expression to hi-impedance, and automatically scales to the vector-size of the expression. See section 3.2.2 (page 65) in chapter 3 for more details on vector fill literal values.

Observe in Example 5-4 that the `out` tri-state output port is declared as a **`tri logic`** type, instead of the usual **`logic`** type. The **`logic`** data type only defines that the port can have 4-state values. It does not define whether the port type is a net or variable type. An output port will default to a variable type, unless explicitly declared as a net type. (Conversely, an input port will default to a net type, unless explicitly declared as a variable). The **`tri`** keyword declares a net type. The **`tri`** type is the same as the **`wire`** type in every way, but the **`tri`** keyword can help document that the net or port is expected to have tri-state (hi-impedance) values.

Figure 5-4: Synthesis result for Example 5-4: Conditional operator (tri-state output)



Technology independent schematic (no target ASIC or FPGA selected)

5.4 Bitwise operators

Bitwise operators perform their operations one bit at a time, working from the right-most bit (the least-significant bit) towards the left-most bit (the most-significant bit). Table 5-3 lists the bitwise operators.

Table 5-3: Bitwise operators for RTL modeling

Operator	Example Usage	Description
<code>~</code>	<code>~m</code>	Invert each bit of <code>m</code> (one's complement)
<code>&</code>	<code>m & n</code>	AND each bit of <code>m</code> with <code>n</code>
<code> </code>	<code>m n</code>	OR each bit of <code>m</code> with <code>n</code>
<code>^</code>	<code>m ^ n</code>	Exclusive-OR each bit of <code>m</code> with <code>n</code>
<code>^~</code> <code>~~</code>	<code>m ^~ n</code>	Exclusive-NOR each bit of <code>m</code> with <code>n</code>

There is no bitwise NAND or NOR operator. A NAND or NOR operation requires inverting the result of an AND or OR operation, respectively, as in `~(m & n)`. The parentheses are required so that the AND will be performed first.

Bitwise operations require that both operands be the same vector size. Before performing the operation, the smaller operand will be left-extended to match the size of the larger operand, as described in section 5.1.3.

Bitwise inversion. The bitwise invert operator inverts each bit of its single operand, working from right to left. The result is a one's complement of the operand value. The bitwise inversion operator is X-pessimistic — the result of inverting an X or Z value is always an X. Table 5-4 shows the truth table for the bitwise inversion. The results in the table are for each bit of the operand.

Table 5-4: Bitwise inversion truth table

<code>~</code>	result
<code>0</code>	<code>1</code>
<code>1</code>	<code>0</code>
<code>x</code>	<code>x</code>
<code>z</code>	<code>x</code>

An example result from a bitwise inversion operation is:

```
logic [3:0] a, r1;
assign a = 4'b01zx;           // some bits are X or Z
assign r1 = ~a;              // results in 4'b10xx
```

Bitwise AND. The bitwise AND operator does a Boolean AND of each bit of the first operand with the corresponding bit in the second operand, working from right to left. The bitwise AND operator is X-optimistic: a 0 ANDed with any value will result in a 0. Table 5-5 shows the truth table for the bitwise AND. The results in the table are for each bit of the two operands.

Table 5-5: Bitwise AND truth table

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Some example results of bitwise AND operations are:

```
logic [3:0] a, b, c, r1, r2;

assign a = 4'b01zx;           // some bits are X or Z
assign b = 4'b0000;           // all bits are zero
assign c = 4'b1111;           // all bits are one

assign r1 = a & b;           // results in 4'b0000
assign r2 = a & c;           // results in 4'b01xx
```

Bitwise OR. The bitwise OR operator does a Boolean OR of each bit of the first operand with the corresponding bit in the second operand, working from right to left. The bitwise OR operator is X-optimistic — a 1 ORed with any value will result in a 1. Table 5-6 shows the truth table for the bitwise OR.

Table 5-6: Bitwise OR truth table

	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Some examples results of bitwise OR operations are:

```
logic [3:0] a, b, c, r1, r2;
```

```

assign a = 4'b01zx;           // some bits are X or Z
assign b = 4'b0000;           // all bits are zero
assign c = 4'b1111;           // all bits are one

assign r1 = a | b;          // results in 4'b01xx
assign r2 = a | c;          // results in 4'b1111

```

Bitwise XOR. The bitwise XOR operator does a Boolean exclusive-OR of each bit of the first operand with the corresponding bit in the second operand, working from right to left. The bitwise XOR operator is X-pessimistic — the result of exclusive-ORing an X or Z value is always an X. Table 5-7 shows the truth table for the bitwise XOR.

Table 5-7: Bitwise XOR truth table

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Some examples results of bitwise XOR operations are:

```

logic [3:0] a, b, c, r1, r2;

assign a = 4'b01zx;           // some bits are X or Z
assign b = 4'b0000;           // all bits are zero
assign c = 4'b1111;           // all bits are one

assign r1 = a ^ b;          // results in 4'b01xx
assign r2 = a ^ c;          // results in 4'b10xx

```

Bitwise XNOR. The bitwise XNOR operator does a Boolean exclusive-NOR of each bit of the first operand with the corresponding bit in the second operand, working from right to left. The bitwise XNOR operator is X-pessimistic — the result of exclusive-NORing an X or Z value is X.

Table 5-8 shows the truth table for the bitwise XNOR.

Table 5-8: Bitwise exclusive NOR truth table

$\wedge\sim$	0	1	x	z
$\sim\wedge$	1	0	x	x
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Some examples results of bitwise XNOR operations are:

```
logic [3:0] a, b, c, r1, r2;

assign a = 4'b01zx;           // some bits are X or Z
assign b = 4'b0000;           // all bits are zero
assign c = 4'b1111;           // all bits are one

assign r1 = a ~^ b;          // results in 4'b10xx
assign r2 = a ^~ c;          // results in 4'b01xx
```

Example 5-5 illustrates a small RTL model that utilizes bitwise operators.

Example 5-5: Using bitwise operators: multiplexed N-bit wide AND/XOR operation

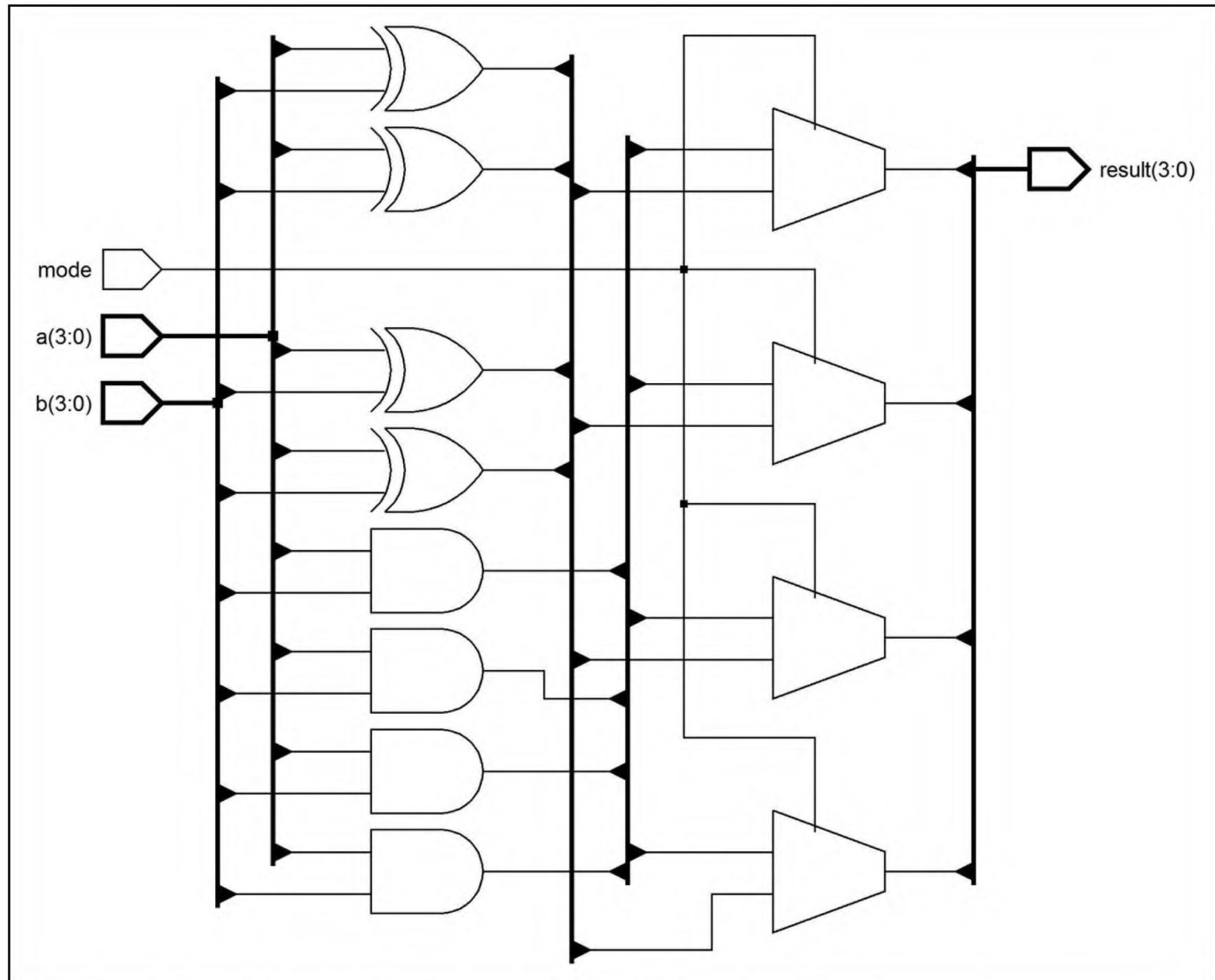
```
// User-defined type definitions
package definitions_pkg;
  typedef enum logic {AND_OP, XOR_OP} mode_t;
endpackage: definitions_pkg

// Multiplexed N-bit wide bitwise-AND or bitwise-XOR operation
module and_xor
  import definitions_pkg::*;
  #(parameter N = 4)                  // op size (default 8-bits)
  (input mode_t          mode,      // 1-bit enumerated input
   input logic [N-1:0] a, b,        // scalable input size
   output logic [N-1:0] result    // scalable output size
);

  always_comb
    case (mode)
      AND_OP: result = a & b;
      XOR_OP: result = a ^ b;
    endcase
endmodule: and_xor
```

Figure 5-5 shows how the RTL model in Example 5-5 might synthesize. As has been noted earlier in this chapter, the implementation created by synthesis can be influenced by a number of factors, including: the target device, any other operators or programming statements used in conjunction with the operator, the synthesis compiler utilized, as well as the synthesis options and constraints that were specified.

Figure 5-5: Synthesis result for Example 5-5: Bitwise AND and OR operations



Technology independent schematic (no target ASIC or FPGA selected)

5.5 Reduction operators

Reduction operators perform their operations on all the bits of a single operand and return a scalar (1-bit) result. Table 5-9 lists the reduction operators.

Table 5-9: Reduction operators for RTL modeling

Operator	Example Usage	Description
&	& m	AND all bits of m
~&	~& m	NAND all bits of m
	m	OR all bits of m
~	~ m	NOR all bits of m
^	^ m	Exclusive-OR all bits of m
~^	~^ m	Exclusive-NOR all bits of m

The reduction operators include a NAND and a NOR operator, which the bitwise operators do not have. The reduction AND, OR and XOR operators perform their operation one bit at a time, working from the right-most bit (the least-significant bit) towards the left-most bit (the most-significant bit). The operations use the same truth tables as their corresponding bitwise operators, as shown in section 5.4. The reduction NAND, NOR and XNOR operators first perform a reduction AND, OR or XOR operation, respectively, and then invert the 1-bit result.

The AND, NAND, OR and NOR operators are X-optimistic. For a reduction AND, if any bit in the operand is 0, the result will be 1'b0. For a reduction NAND, if any bit in the operand is 0, the result will be 1'b1. Similarly, for a reduction OR, if any bit in the operand is 1, the result will be 1'b1. For a reduction NOR, if any bit in the operand is 1, the result will be 1'b0. The reduction XOR and XNOR operators are X-pessimistic. If any bit of the operand is X or Z, the result will be 1'bx. Table 5-10 shows the result of each reduction operator for a few example values.

Table 5-10: Example results of reduction operations

Operand	&	~&		~	^	~^
4'b0000	1'b0	1'b1	1'b0	1'b1	1'b0	1'b1
4'b1111	1'b1	1'b0	1'b1	1'b0	1'b0	1'b1
4'b1000	1'b0	1'b1	1'b1	1'b0	1'b1	1'b0
4'b000z	1'b0	1'b1	1'bx	1'bx	1'bx	1'bx
4'b100x	1'b0	1'b1	1'b1	1'b0	1'bx	1'bx

Example 5-6 illustrates a small RTL model that utilizes reduction operators to check for correct parity of a data value. Figure 5-6 shows how this RTL model might synthesize.

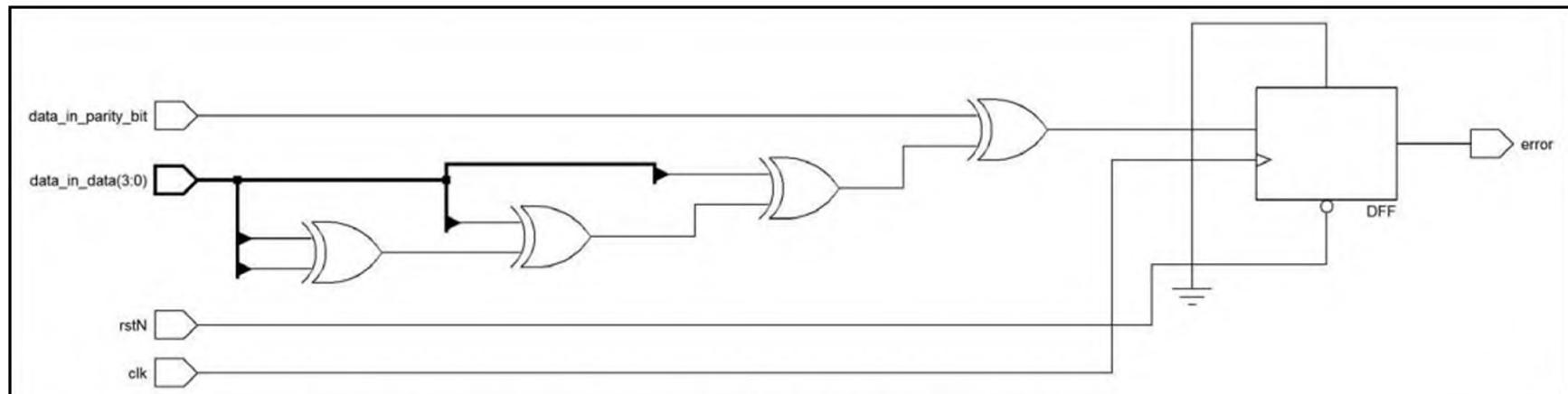
Example 5-6: Using reduction operators: parity checker using XOR

```
// User-defined type definitions
package definitions_pkg;
    typedef struct {
        logic [3:0] data;
        logic         parity_bit;
    } data_t;
endpackage: definitions_pkg

// Parity checker using even parity (the combined data value
// plus parity bit should have an even number of bits set to 1
module parity_checker
    import definitions_pkg::*;
    (input data_t data_in, // 5-bit structure input
     input clk,           // clock input
     input rstN,          // active-low asynchronous reset
     output logic error   // set if parity error detected
    );

    always_ff @(posedge clk or negedge rstN) // async reset
        if (!rstN) error <= 0;                // active-low reset
        else      error <= ^{data_in.parity_bit, data_in.data};
        // reduction-XOR returns 1 if an odd number of bits are
        // set in the combined data and parity_bit
endmodule: parity_checker
```

Figure 5-6: Synthesis result for Example 5-6: Reduction XOR (parity checker)



Technology independent schematic (no target ASIC or FPGA selected)

5.6 Logical operators

Logical operators evaluate their operands, and return a value indicating whether the result of the evaluation is true or false. For example, the operation `a && b` tests to see if both `a` and `b` are true. If both operands are true, the `&&` operator returns true. Otherwise, the operator returns false.

Logical operator return values. SystemVerilog does not have a built-in true or false Boolean value. Instead, the return of logical operators use the logic value `1'b1` (a one-bit wide logic 1) to represent true, and `1'b0` to represent false. Logical operators can also return a `1'bx` to indicate an ambiguous condition where simulation cannot determine if actual logic gates would evaluate as a true or false condition.

Evaluating an expression as true or false. To determine if an operand is true or false, SystemVerilog uses the following rules:

- An operand is *false* if all bits are 0
- An operand is *true* if any bit is 1
- An operand is *unknown* if any bit is X or Z , and no bits are 1

Table 5-11 lists the logical operators universally supported by RTL synthesis compilers.

Table 5-11: Logical operators for RTL modeling

Operator	Example Usage	Description
<code>&&</code>	<code>m && n</code>	Logical AND: Is <code>m</code> true AND is <code>n</code> true?
<code> </code>	<code>m n</code>	Logical OR: Is <code>m</code> true OR is <code>n</code> true?
<code>!</code>	<code>! m</code>	Logical negate: Is <code>m</code> not true?

The logical negate operator is often referred to as the *not* operator, which is short for “not true”.

Logical operators perform their operations by first doing a logical OR reduction of each operand, which yields a 1-bit result. That result is then evaluated to determine if it is true or false. In the case of the negate operator, the 1-bit result is first inverted, and then evaluated as true or false.

Tables 5-12 and 5-13 show the results of these logical operators for a few example values.

Table 5-12: Example results for logical AND and OR operations

Operand 1	Operand 2	&&	
4'b0000	4'b0000	1'b0	1'b0
4'b0000	4'b1000	1'b0	1'b1
4'b0000	4'b00zx	1'b0	1'bx
4'b0000	4'b01zx	1'b0	1'b1
4'b1000	4'b0000	1'b0	1'b1
4'b1000	4'b1000	1'b1	1'b1
4'b1000	4'b00zx	1'bx	1'b1
4'b1000	4'b01zx	1'b1	1'b1

Table 5-13: Example results of logical negate operations

Operand 1	!
4'b0000	1'b1
4'b1000	1'b0
4'b00zx	1'bx
4'b01zx	1'b0

5.6.1 Difference between negate and invert operations

Care should be taken to not confuse the logical negate operator (!) and the bitwise invert operator (~). The negate operator performs a true/false evaluation of its operand and returns a 1-bit value representing a true, false or unknown result. The invert operator performs a logical inversion of each bit of an operand (one's complement), and returns a value of the same bit width as the operand.

In some operations, the results of these operations happen to be the same, but, in other operations, they return very different values. The difference can result in faulty code when the operators are incorrectly used in conjunction with decision statements. Consider the following example:

```

logic enable; // 1-bit control signal
logic [1:0] select; // 2-bit control signal

assign enable = 1'b1;
assign select = 2'b01;

if (!enable) ... // evaluates as FALSE
if (~enable) ... // evaluates as FALSE

if (!select) ... // evaluates as FALSE
if (~select) ... // evaluates as TRUE -- BUG!

```

The reason for the difference in the last two lines of the preceding code snippet is that the two operators work differently. The logical negate operator (!) performs a true/false evaluation of the 2-bit select by OR-reducing the two bits together, and then inverting the 1-bit result. The bitwise invert operator (~) just inverts the value of each bit of the 2-bit select vector and returns a 2-bit result. The **if** statement then does a true/false test on the 2-bit vector, which evaluates as true because the inverted value still has a bit set to 1.

Best Practice Guideline 5-1

Use the bitwise invert operator to invert the bits of a value. Do not use the bitwise invert operator to negate logical true/false tests. Conversely, use the logical negate operator to negate the result of a true/false test. Do not use the logical negate operator to invert a value.

Best Practice Guideline 5-2

Only use the logical true/false operators to test scalar (1-bit) values. Do not perform true/false tests on vectors.

A logical operation will return true if any bit of the vector is set, which could lead to design errors when testing for specific bits. When evaluating vector values, use an equality or relational operator to test for acceptable values.

Example 5-7 illustrates a small RTL model that uses the negate, logical AND and logical OR operators. The design is a logical comparator that sets a flag if either of two data values fall within a configurable range of values.

Example 5-7: Using logical operators: set flag when values are within a range

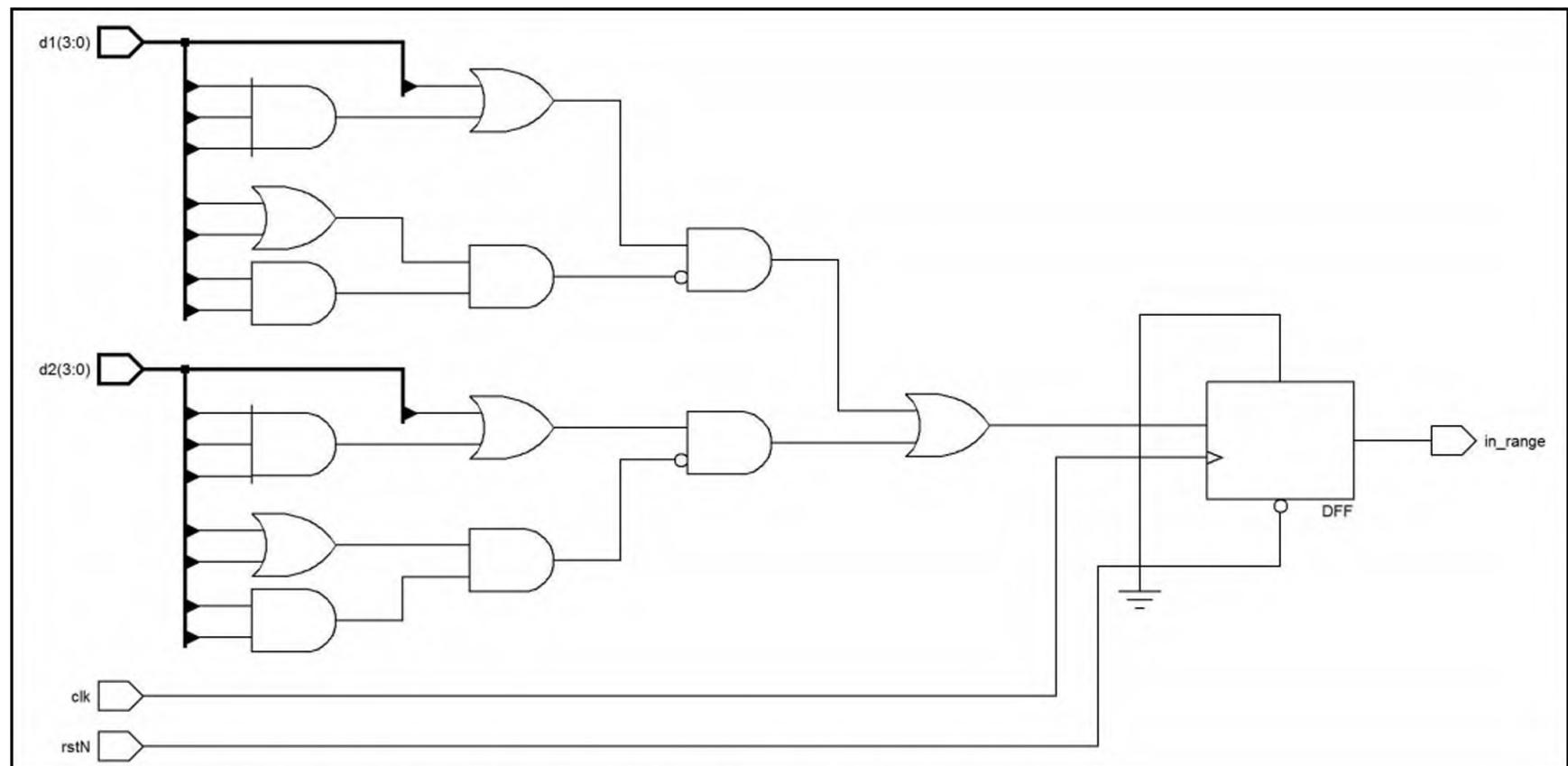
```

module status_flag
  #(parameter N      = 4,      // data bus size
    logic [N-1:0] MIN = 'h7, // minimum must-have value
    logic [N-1:0] MAX = 'hC // maximum must-have value
  )
  (input logic clk,           // clk input
   input logic rstN,          // active-low async reset
   input logic [N-1:0] d1, d2, // scalable input size
   output logic in_range     // set if either d1 or d2
  );
  begin
    always_ff @(posedge clk or negedge rstN) // async reset
      if (!rstN) in_range <= '0;           // active-low reset
      else in_range <= ((d1 >= MIN) && (d1 <= MAX))
        || ((d2 >= MIN) && (d2 <= MAX))
    );
  endmodule: status_flag

```

Figure 5-7 shows how the RTL model in Example 5-7 might synthesize.

Figure 5-7: Synthesis result for Example 5-7: Logical operators (in-range compare)



Technology independent schematic (no target ASIC or FPGA selected)

5.6.2 Short circuiting logical operations

SystemVerilog simulation will *short-circuit* the **&&** and **||** logical evaluation, which means simulation will abort the evaluation of the operation as soon as the result is known. In the case of **m || n**, for example, if the evaluation of **m** is true, a simulator will skip evaluating **n** because it has already been determined that the result of the logical OR will be true.

SystemVerilog simulators will do short-circuiting of logical operators, but the actual logic gate-level implementation generated by synthesis will never do short-circuiting. The actual logic gates will evaluate both operands in parallel. There is a rare corner case where this difference between simulation and synthesis can result in design bugs. The corner case occurs when an the second operand is a call to a function, and the function modifies the values of variables that are external to the function. In the actual gate-level implementation, the effects on the external variables will always occur. In simulation, however, if the function is not called because of short-circuiting, then the external variables modified by the function will not be updated.

Best Practice Guideline 5-3

A function should only modify its function return variable and internal temporary variables that never leave the function.

The short circuiting corner case described in the preceding paragraph can be avoided by not modifying variables that are external to the function. This corner case can lead to critical mismatches in how simulation behaves and the gate-level implementation from synthesis behaves.

5.6.3 Non-synthesizable logical operators

SystemVerilog-2009 added two additional logical operators that were not generally supported by RTL synthesis compilers at the time this book was written. These are the implication and equivalence operators. The tokens and descriptions of these two operators are listed in Table 5-14.

Table 5-14: Non-synthesizable logical operators

Operator	Example Usage	Description
<code>-></code>	<code>m -> n</code>	Logical implication: a shortcut for <code>(!m n)</code> (if m is not true, n needs to be true)
<code><-></code>	<code>m <-> n</code>	Logical equivalence: <code>(!m n) && (!n m)</code> (if m is not true then n needs to be true; if n is not true then m needs to be true)

5.7 Comparison operators (equality and relational)

Comparison operators evaluate their operands and return a value indicating whether the result of the evaluation is true or false. The logic value `1'b1` (a one-bit wide logic 1) represents true, and `1'b0` represents false. In simulation, these comparison operators can also return a `1'bx` to indicate an ambiguous condition where simulation cannot determine if actual logic gates would result in a 1 (true) or 0 (false).

Table 5-15 lists the comparison operators. All comparison operators are synthesizable.

Table 5-15: Comparison operators for RTL modeling

Operator	Example Usage	Description
<code>==</code>	<code>m == n</code>	Equality: Is m equal to n?
<code>!=</code>	<code>m != n</code>	Not Equality: Is m not equal to n?
<code><</code>	<code>m < n</code>	Less-than: Is m less than n?
<code><=</code>	<code>m <= n</code>	Less-than or equal: Is m less than or equal to n?
<code>></code>	<code>m > n</code>	Greater-than: Is m greater than n?
<code>>=</code>	<code>m >= n</code>	Greater-than or equal: Is m greater than or equal to n?

Pessimistic comparisons. Comparison operators are unique from most other SystemVerilog operators in that they are always pessimistic. If either operand has even a single bit that is X or Z, the operand is considered unknown, and therefore the result will be unknown. This pessimism at the RTL level is an abstraction from actual logic gate-level behavior, which would be more optimistic. Consider the following values with a greater-than operation:

```

logic [3:0] c, d;
logic      gt;

assign c = 4'b1001;      // the numeric value 9
assign d = 4'b000z;      // ambiguous, could be numeric 0 or 1

assign gt = (c > d);    // operation returns 1'bx (unknown)

```

The upper 3 bits of variable d are zero, but the least-significant bit has an ambiguous high-impedance value. In actual logic gates, this high-impedance bit might be sensed as either a 0 or a 1, which means the numeric value of d might be either 0 or 1.

If the greater-than operation were optimistic, which it is not, it could compare the bits of c and d. Since the most-significant bit of c is 1, and the most-significant bit of d is 0, an optimistic operation could determine that c must be greater than d, even though the least-significant bit of d is ambiguous. The hardware implementation of a comparator would look at the value of each bit, and would likely have this optimistic behavior.

The RTL comparison operators are more pessimistic than actual logic gates. Because of the ambiguity of the high-impedance bit in d, the (`c > d`) operation returns a 1'bx, indicating that the result of the comparison is unknown.

Signed and unsigned comparisons. The comparison operators can perform either signed or unsigned comparisons. The rule is: if both operands are signed expressions,

a signed comparison will be performed. If either operand is an unsigned expression, the other operand will be treated as an unsigned value.

This rule can lead to unexpected, and probably unintentional, results when signed and unsigned values are mixed in the same model. The result in the following snippet is probably a design bug:

```
logic      [7:0] u1;
logic signed [7:0] s1;
logic          gt;

assign u1 = 5;           // unsigned 5
assign s1 = -3;          // negative 3

assign gt = s1 > u1;    // returns true -- a GOTCHA
```

A “*gotcha*” is programming slang for code that is syntactically legal, but which yields unintentional or undesirable results. Having -3 evaluate as greater than 5 is a gotcha. The reason for this undesirable result is because a signed negative value is represented in two’s complement form, with the most-significant bit set to indicate that the value is negative. When an unsigned comparison treats this two’s complement bit pattern as a positive value, the most-significant bit makes the value a large positive value.

Best Practice Guideline 5-4

Avoid mixing signed and unsigned expressions with comparison operations. Both operands should be either signed or unsigned.

If mixed signed and unsigned comparisons are a requirement of the design, then it might be desirable to compare absolute values instead of negative values. SystemVerilog does not have an operator or built-in function that returns the absolute value of a negative value. Instead, the absolute value must be calculated by performing a two’s complement operation. The arithmetic unary subtract operator (-) can be used for this.

A synthesizable function to perform an absolute operation with parameterized bus widths is:

```
function [WIDTH-1:0] abs_f (logic signed [WIDTH-1:0] a);
    return (a >= 0)? a : -a; // 2's complement negative values
endfunction: abs_f
```

An example of using this function is:

```
parameter WIDTH = 8;
logic      [WIDTH-1:0] u1;
logic signed [WIDTH-1:0] s1;
logic          gt1, gt2;
```

```
assign u1 = 5;          // unsigned 5
assign s1 = -3;         // negative 3

assign gt1 = abs_f(s1) > abs_f(u1);    // returns false
assign gt2 = abs_f(u1) > abs_f(s1);    // returns true
```

Example 5-8 illustrates a small RTL model that uses the less-than, greater-than and equality comparison operators. Figure 5-8 shows how this model might synthesize.

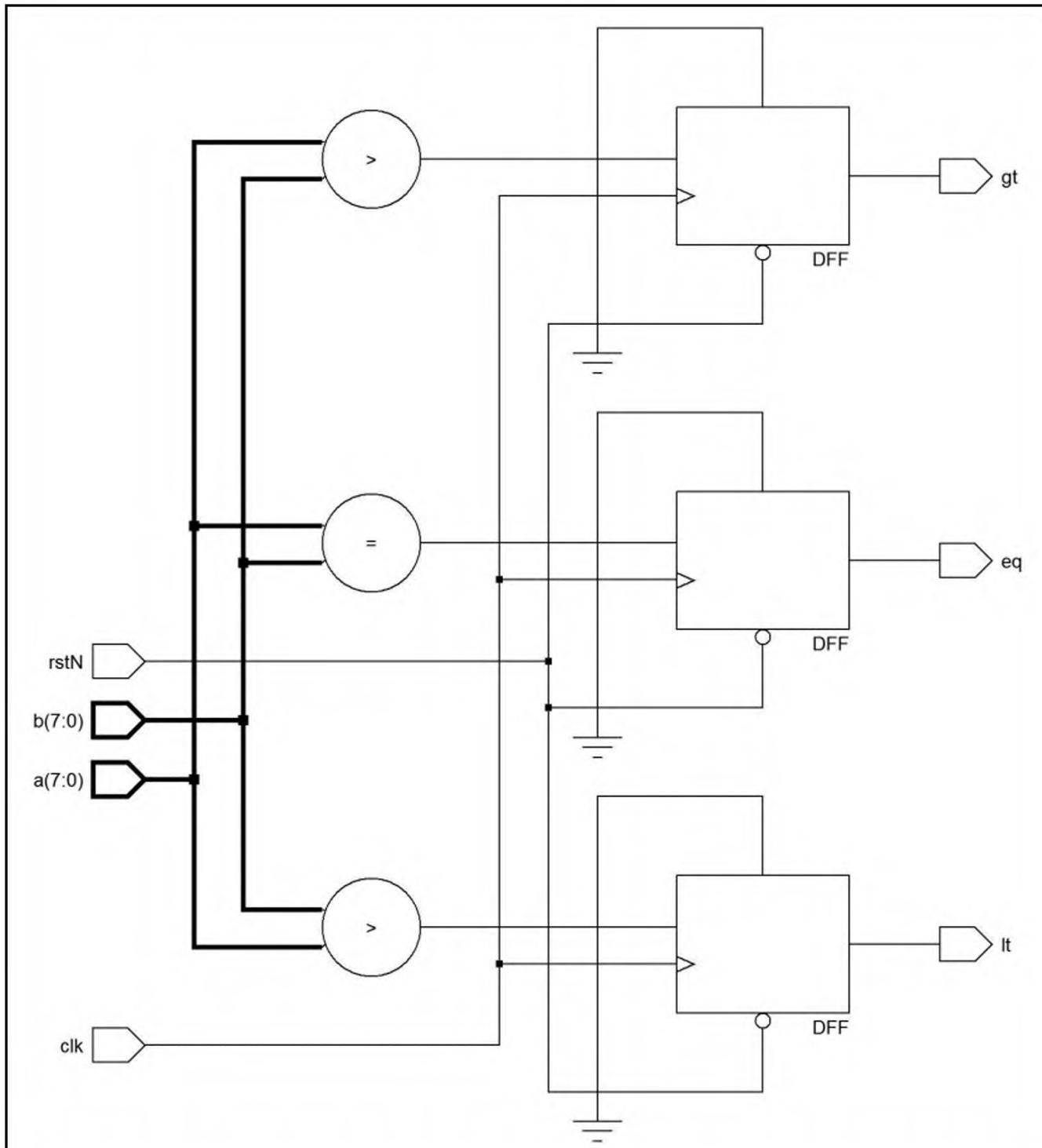
Example 5-8: Using comparison operators: a relationship comparator

```
//
// Set lt, eq and gt flags based on if a is less-than, equal-to
// or greater-than b, respectively
//

module comparator
#(parameter N = 8)           // data size (default 8-bits)
(input logic      clk,       // clock input
 input logic      rstN,      // active-low async reset
 input logic [N-1:0] a, b,    // scalable input size
 output logic     lt,        // set if a is less than b
 output logic     eq,        // set if a is equal to b
 output logic     gt)        // set if a is greater than b
);

always_ff @(posedge clk or negedge rstN) // async reset
  if (!rstN) {lt,eq,gt} <= '0; // reset flags
  else begin
    lt <= (a < b); // less-than operator
    eq <= (a == b); // equality operator
    gt <= (a > b); // greater-than operator
  end
endmodule: comparator
```

Figure 5-8: Synthesis result for Example 5-8: Relational operators (comparator)



Technology independent schematic (no target ASIC or FPGA selected)

The schematic shown in Figure 5-8 is based on generic components, before the synthesis compiler has mapped the functionality to a specific target ASIC or FPGA device. The synthesis compiler used to generate this generic schematic used a generic greater-than comparator twice, but the top instance has the *a* and *b* inputs reversed. The manner in which this generic functionality is mapped to actual components will depend on the types of components available in a specific target technology.

5.8 Case equality (identity) operators

In addition to the comparison operators discussed in section 5.7, SystemVerilog also has *case equality* operators, which are also referred to as *identity operators*.

Table 5-16 lists the case equality operators.

Table 5-16: Case equality (identity) operators for RTL modeling

Operator	Example Usage	Description
<code>==</code>	<code>m == n</code>	Case equality: Is m identical to n?
<code>!=</code>	<code>m != n</code>	Not case equality: Is m not identical to n?
<code>==?</code>	<code>m ==? n</code>	Wildcard case equality: Is m identical to n, after masking?
<code>!=?</code>	<code>m !=? n</code>	Wildcard not case equality: Is m not identical to n, after masking?

The `==` and `!=` *case equality* operators are similar in usage to the `==` and `!=` comparison equality operators, but with an important functional difference. The case equality operators perform their operation by comparing each bit of the two operands for all 4 possible logic values, 0, 1, Z and X, whereas the comparison equality operators only compare for values of 0 and 1 in each bit of the operands.

NOTE

Some RTL synthesis compilers do not support the `==` and `!=` case equality operators at all. Other RTL synthesizers support these operators, but restrict the usage to expressions that do not involve X or Z values.

Best Practice Guideline 5-5

Use the `==` and `!=` equality operators in RTL models. Do not use the `==` and `!=` case equality operators.

The equality operators are supported by all RTL synthesis compilers. The case equality operators are not universally supported. The case equality operators should only be used in testbench code that is not intended to be synthesized.

The `==?` and `!=?` *wildcard case equality operators* are synthesizable. These operators compare the bits of two values, with ability to mask out specific bits from the comparison. Bits are masked out by specifying an X, Z or ? for the masked bits in the second operand. The mask acts like a wildcard because the corresponding bit in the first operand can be any value since it is masked from the comparison. In Example 5-9, the comparison is only made on the upper 8 bits of a 16 bit word. The lower 8 bits are ignored, and could therefore be any value.

Example 5-9: Using case equality operators: a comparator for high address range

```

// Set high_addr flag if all bits of upper byte of address
// are set
//

module high_address_check
(input logic clk, // clock input
 input logic rstN, // active-low async reset
 input logic [31:0] address, // 32-bit input
 output logic high_addr // set high-byte all ones
);

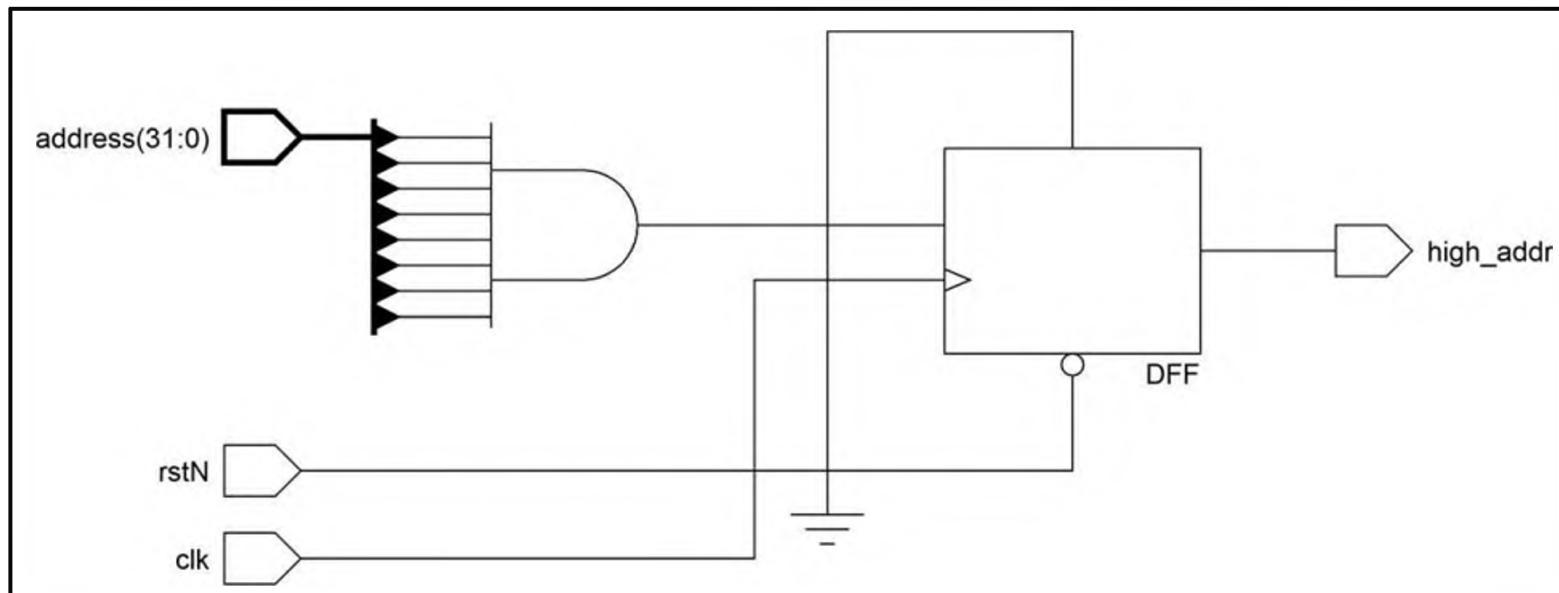
always_ff @(posedge clk or negedge rstN) // async reset
  if (!rstN) // active-low reset
    high_addr <= '0;
  else
    high_addr <= (address ==? 32'hFF??????); // mask low bits
endmodule: high_address_check

```

The mask bits in Example 5-9 could have been represented as `32'hFFxxxxxx`, `32'hFFzzzzzz`, `32'hFF??????`, or any combination of X, Z or ?, and using either lowercase or uppercase characters for the X and Z. While all these variations are functionally identical, the use of a question mark as a wildcard can make the code more understandable and self-documenting. The letters X and Z can be wildcards in some contexts, and literal values in other contexts. The dual usage of these letters can make code less intuitive when used as a wildcard instead of as a logic value.

The `==?` and `!=?` wildcard case equality operators are treated by synthesis compilers in the same manner as the `==` and `!=` equality operators, but with the masked bits left out of the comparator. Figure 5-9 shows how Example 5-9 might synthesize.

Figure 5-9: Synthesis result for Example 5-9: Case equality, `==?` (comparator)



Technology independent schematic (no target ASIC or FPGA selected)

5.9 Set membership (**inside**) operator

The **inside** set membership operator compares an expression to a set of other expressions that are enclosed in curly braces ({ }) and separated by commas. The operator returns a 1'b1 (representing true) if the first expression matches any of the expressions in the set. If none of the expressions in the list match, the operator returns 1'b0 (representing false). Table 5-17 shows the general syntax and usage of the **inside** operator.

Table 5-17: Set membership operator for RTL modeling

Operator	Example Usage	Description
inside {}	m inside {0, 1, n}	Set membership: Does m match 0, 1 or n?

An example of using the inside operator in synthesizable RTL code is:

```
always_ff @(posedge clk)
  if (address inside {0, 32, 64, 128, 256, 512, 1024})
    boundary <= '1;
  else
    boundary <= '0;
```

The inside operator returns a 1'b1 or 1'b0 value to represent true or false, respectively. In the preceding example the operator return value could be directly assigned to boundary, without the need of the if-else decision.

```
always_ff @(posedge clk)
  boundary <= address inside {0, 32, 64, 128, 256, 512, 1024};
```

This same functionality can be modeled using logical OR operators, but the inside operator is much more concise. Compare the previous code snippet to the following traditional Verilog style.

```
always @(posedge clk)
  boundary <= (address==0) || (address==32)
    || (address==64) || (address==128)
    || (address==256) || (address==512)
    || (address==1024);
```

The inside operator set of expressions can be a range of values between square brackets ([]), with a colon (:) between the two extremes of the range.

```
always_comb begin
  small_value = data inside {[0:255]};
end // true if data matches a value between 0 and 255
```

The inside operator also allows bits in the value list to be masked out of a comparison in the same way as the wildcard case equality operator (==?). Any bit in the set of values that is specified as X, Z or ? is not considered in the comparison. An ignored

bit is a wildcard — the corresponding bit position in the first operand could be any value, including 4-state values.

```
always_comb begin
    pattern_flag = data inside {8'b??1010??};
end // true if the middle bits of data match 1010
```

Example 5-10 illustrates a small RTL model that uses the **inside** operator.

Example 5-10: Using the set membership operator: a decoder for specific addresses

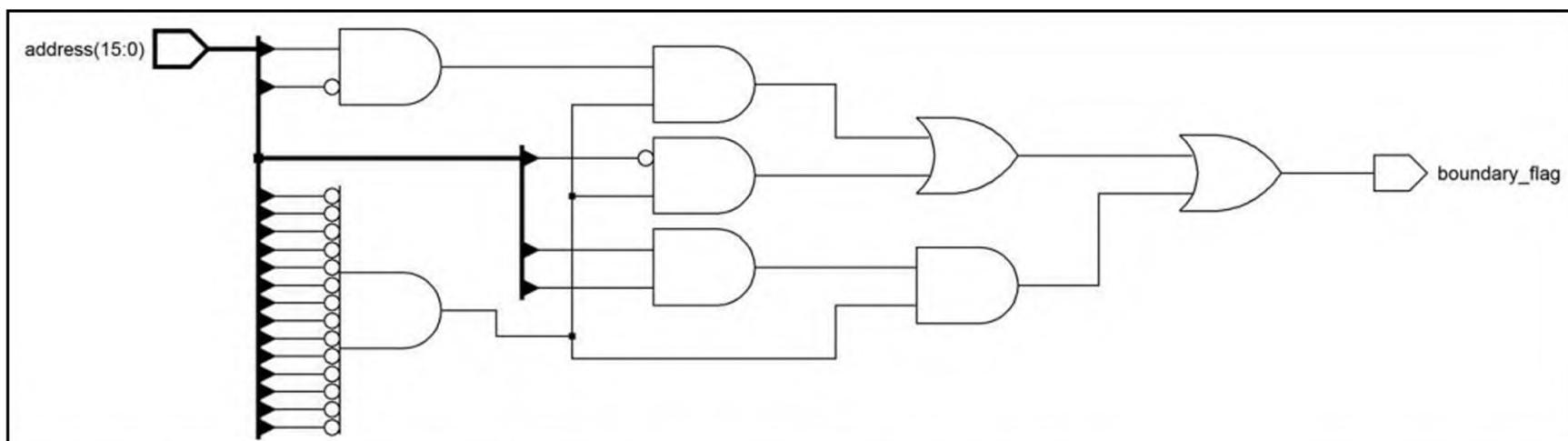
```
// Decoder that sets a flag whenever address is on a quadrant
// boundary of the address range
//

module boundary_detector
#(parameter N = 16)
(input logic [N-1:0] address,           // address bus
 output logic          boundary_flag // set when address is at
                                         // a quadrant boundary
);

always_comb begin
    boundary_flag = (address inside { (0),           // quad 1
                                         (((2**N)/4)*1), // quad 2
                                         (((2**N)/4)*2), // quad 3
                                         (((2**N)/4)*3)  // quad 4
} );
end
endmodule: boundary_detector
```

The **inside** operator is versatile, and can represent a variety of gate-level comparison circuits. Figure 5-10 shows how the model in Example 5-10 synthesized.

Figure 5-10: Synthesis result for Example 5-10: Inside operator (boundary detector)



Technology independent schematic (no target ASIC or FPGA selected)

Additional uses of the `inside` operator. The `inside` set membership operator allows the members of the list to be expressions that can change during simulation. Some other ways this operator can be used include:

1. The list of values can be expressions, such as other variables or nets.

```
always_comb begin
    data_matches = data inside {a, b, c};
end // true if data matches the current value of a, b or c
```

2. The set of values can be stored in an array.

```
always_comb begin
    prime_val = data inside {PRIMES};
end // true if data matches a value inside the PRIMES array
```

3. The operator can be used in continuous assignments.

```
assign prime_val = data inside {PRIMES};
```

NOTE

At the time this book was written, some RTL synthesis compilers did not fully support the `inside` operator. Make sure that all tools in the design flow support the ways the `inside` operator is being used in a project.

5.10 Shift operators

Shift operators shift the bits of a vector right or left a specified number of times. SystemVerilog has both bitwise and arithmetic shift operators, listed in Table 5-18.

Table 5-18: Shift operators for RTL modeling

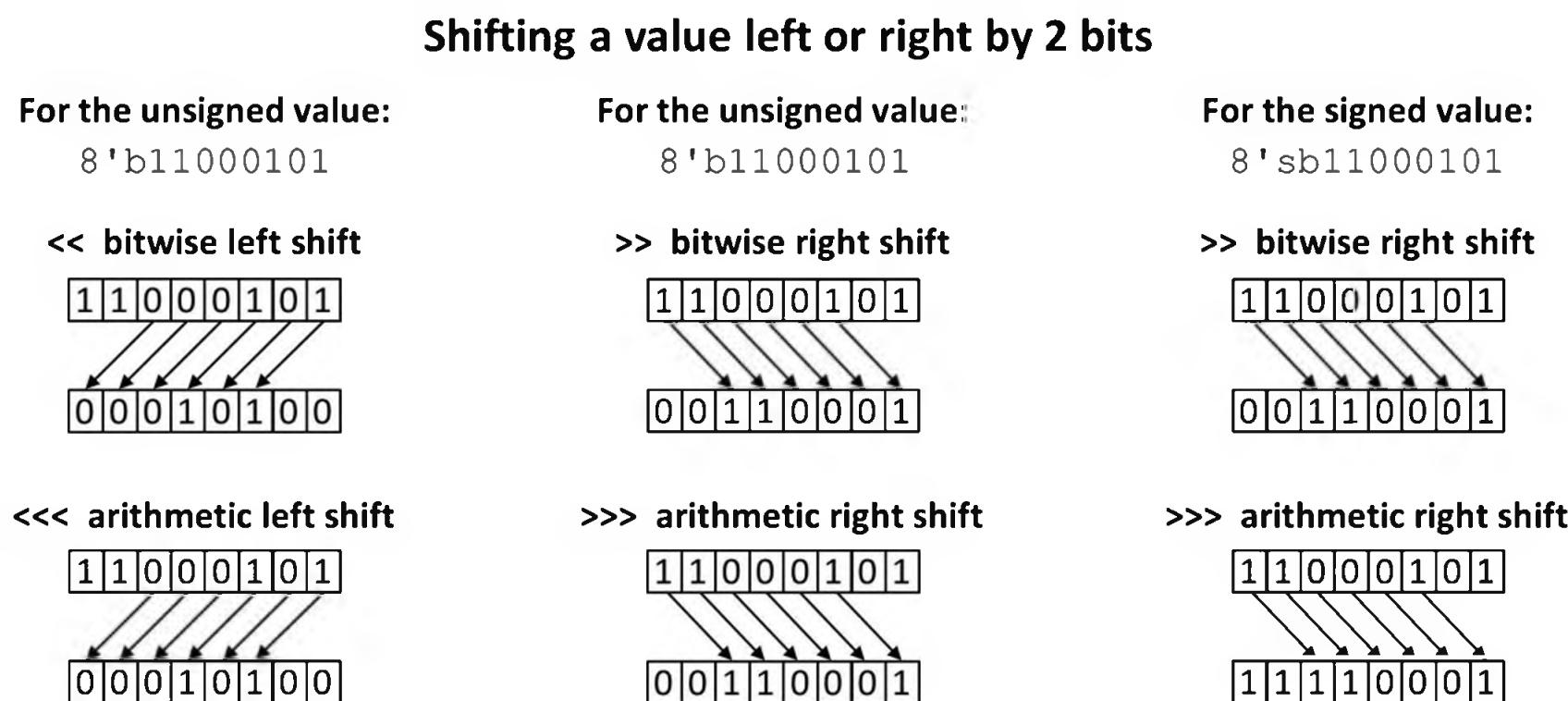
Operator	Example Usage	Description
<code>>></code>	<code>m >> n</code>	Bitwise right shift: shifts m right n times
<code><<</code>	<code>m << n</code>	Bitwise left shift: shifts m left n times
<code>>>></code>	<code>m >>> n</code>	Arithmetic right shift: shifts m right n times, preserving the value of the sign bit of a signed expression
<code><<<</code>	<code>m <<< n</code>	Arithmetic left shift: shifts m left n times (same result as bitwise left shift)

A *bitwise shift* simply moves the bits of a vector right or left the specified number of times. The bits that are shifted out of the vector are lost. The new bits that are shifted in are zero filled. For example, the operation `8'b11000101 << 2` will result in the value `8'b00010100`. A bitwise shift will perform the same operation, regardless of whether the value being shifted is signed or unsigned.

An *arithmetic left shift* performs the same operation as a bitwise right shift on both signed and unsigned expressions. An *arithmetic right shift* performs a different operation on unsigned and signed expressions. If the expression being shifted is unsigned, an arithmetic right shift behaves the same as a bitwise right shift, which is to fill the incoming bits with zero. If the expression is signed, an arithmetic right shift will maintain the signedness of the value by filling each incoming bit with the value of the sign bit.

Figure 5-11 shows how these shift operations move the bits of a vector by 2 bits.

Figure 5-11: Bitwise and arithmetic shift operations



5.10.1 Synthesizing shift operations

Shifting a fixed number of times. A shift operation for a fixed number of times simply rewrites the bits of a bus, with the incoming bits tied to ground. No logic gates are required to implement a fixed shift. Example 5-11 illustrates a simple divide-by-two combinational logic model, where the division is performed by shifting an 8-bit bus right by one bit.

Example 5-11: Using the shift operator: divide-by-two by shifting right one bit

```

// Divide-by-two operation by shifting an N-bit bus right by
// one bit. Fractional results are rounded down.
//

module divide_by_two
#(parameter N = 8)
(input logic [N-1:0] data_in, // N-bit input
 output logic [N-1:0] data_out // N-bit output
);

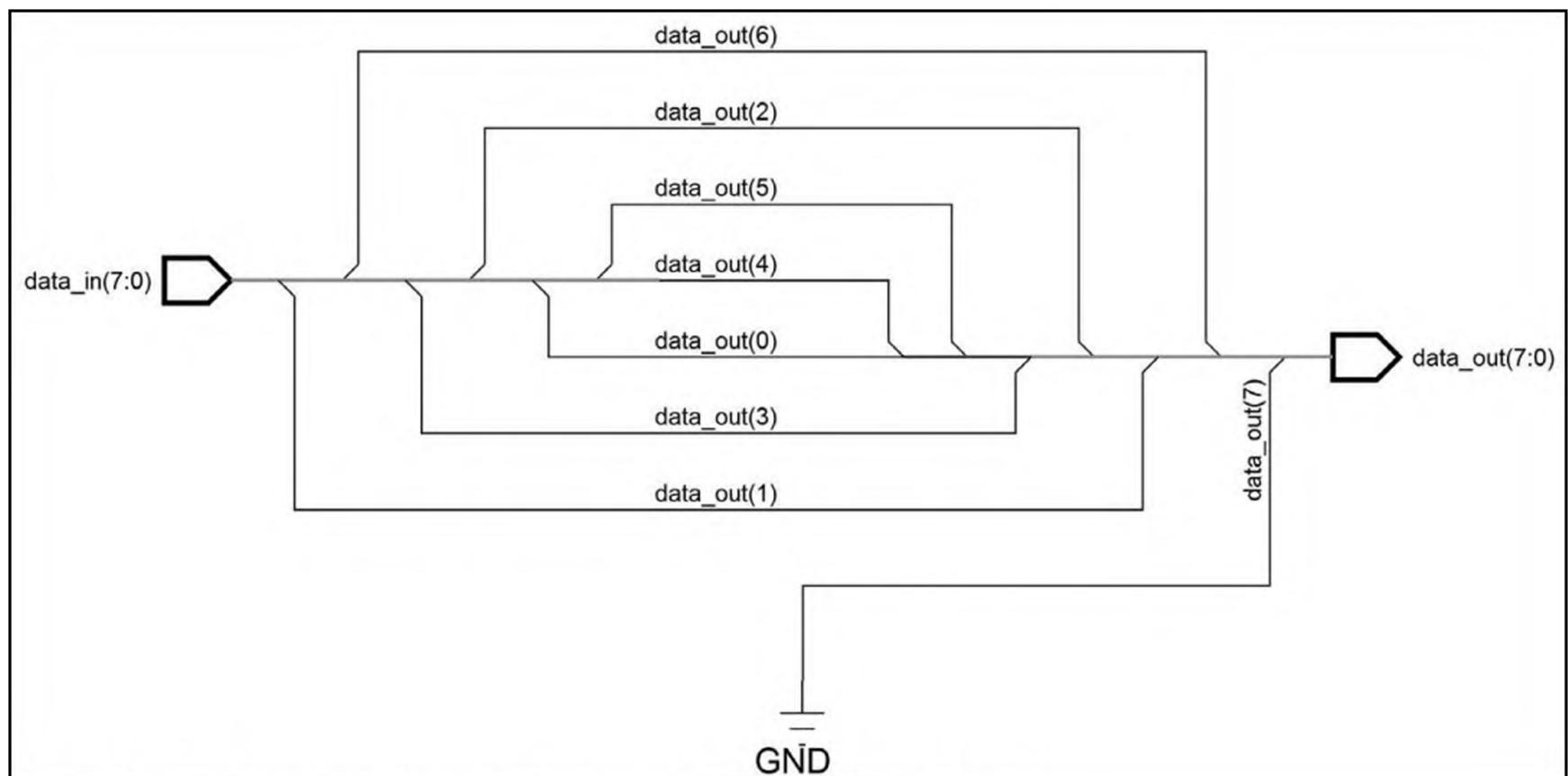
assign data_out = data_in >> 1; // shift right one bit

endmodule: divide_by_two

```

Figure 5-12 shows how this shift right for a fixed number of bits might synthesize. The synthesis compiler placed buffers on the inputs and outputs of the module, but did not utilize any additional gates to perform the operation.

Figure 5-12: Synthesis result for Example 5-11: Shift operator, right-shift by 1 bit



Technology independent schematic (no target ASIC or FPGA selected)

A shift for a fixed number of times can also be represented using a concatenate operation. The following two lines of code are functionally identical.

```

logic [7:0] in, out1, out2;

assign out1 = in >> 1;           // shift using shift op
assign out2 = {1'b0,in[7:1]}; // shift using concatenate op

```

Both styles of performing a shift operation will synthesize to the same rewired hardware. There is no advantage of one style over the other style.

Shifting a variable number of times. A shift operation for a variable number of times represents the functionality of a barrel shifter. The exact implementation, however, will depend on the gate-level functionality available in the specific target library. Some target devices might have a pre-built barrel shifter that has already been optimized for that device. Other devices might require synthesis to build-up the barrel shifter from lower-level gates.

One application of a barrel shifter is to multiply (by shifting left) or divide (by shifting right) by powers of 2. For example, shifting left by 1 bit multiplies a value by 2. Shifting left by 2 bits multiplies a value by 4.

Example 5-12 shows the code for a variable left-shift operation.

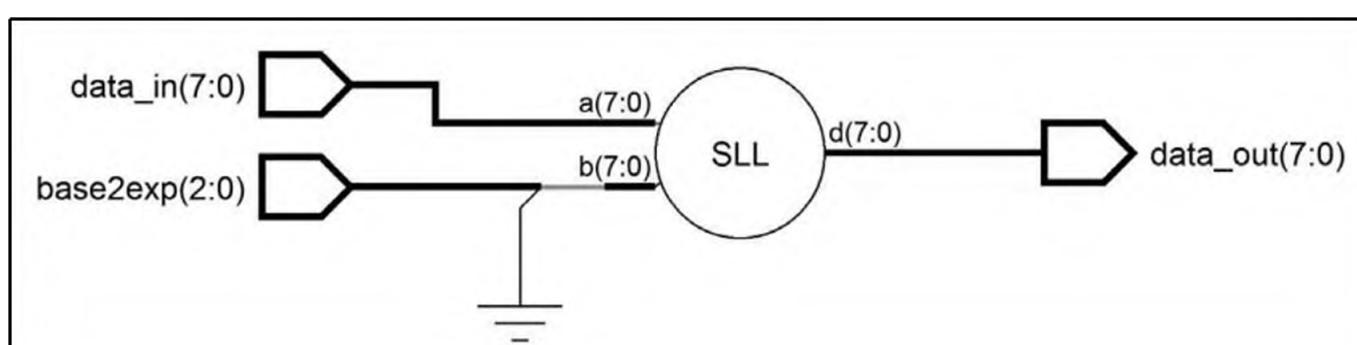
Example 5-12: Using the shift operator: multiply by a power of two by shifting left

```
//  
// Multiply by a power of two operation by shifting an N-bit  
// bus left by a variable number of times; no overflow.  
  
module multiply_by_power_of_two  
#(parameter N = 8)  
(input logic [N-1:0] data_in, // N-bit input  
 input logic [$clog2(N)-1:0] base2exp, // ceiling log2 of N  
 output logic [N-1:0] data_out // N-bit output  
);  
  
assign data_out = data_in << base2exp; // shift left  
  
endmodule: multiply_by_power_of_two
```

The **\$clog2** system function in this example is used to calculate the width of the **base2exp** input port. This function returns the ceiling (fractional values are rounded up to the next whole number) of the **log2** of a value. The function is a convenient way to calculate how many bits are required to represent a value.

Figure 5-13 illustrates how this model might synthesize. The schematic is the intermediate synthesis result, before the shift functionality has been mapped and optimized to a specific device. A generic “shift left logical” component represents the unmapped shift operation.

Figure 5-13: Synthesis result for Example 5-12: Shift operator, variable left shifts



Technology independent schematic (no target ASIC or FPGA selected)

The generic shift-left component in the synthesis results has the same number of bits for both of its inputs. The unused upper bits for the base2exp input are tied to ground. These unused bits might be removed when synthesis maps the generic shift-left component to a specific target implementation.

The shift operator can be used to multiply or divide by values other than a power of 2. The following example shifts a vector 7 times.

```
assign out = in << 7; // shift left by non-power of 2
```

Shifting by a value that is not a power of 2 can be implemented in hardware by cascading shift operations. For example, shifting left 7 times can be done by chaining a 4-bit left-shifter, a 2-bit left-shifter, and a 1-bit left-shifter.

Let synthesis do its job! Synthesis allows engineers to design at an abstract level, focusing on functionality without getting bogged down in implementation details, and without having to be overly concerned about the features of a specific ASIC or FPGA. The synthesis compiler translates the abstract functional model to an efficient implementation for a target ASIC or FPGA. While it is possible to model barrel shift behavior at a more detailed level, there is generally no advantage in doing so. Modern synthesis compilers recognize barrel-shift behavior in an abstract RTL model using the shift operator, and will produce an optimal implementation of this functionality in the target device. This implementation might vary for different target devices, depending on what standard cells, LUTs, or gate-arrays are available in that device.

5.10.2 Synthesizing rotate operations

SystemVerilog does not have a rotate operator. A rotation for a fixed number of times can be modeled using the concatenation operator. For example:

```
logic [7:0] in, out1, out2;

assign out1 = {in[0],in[7:1]}; // rotate in right one bit

assign out2 = {in[5:0],in[7:6]}; // rotate in left two bits
```

Rotating a vector a variable number of times can be accomplished by using a shift operation in conjunction with the concatenate operator.

```
logic [ 7:0] in, out1, out2;
logic [ 2:0] rotate_num;
logic [15:0] temp1, temp2; // size is twice the width of in

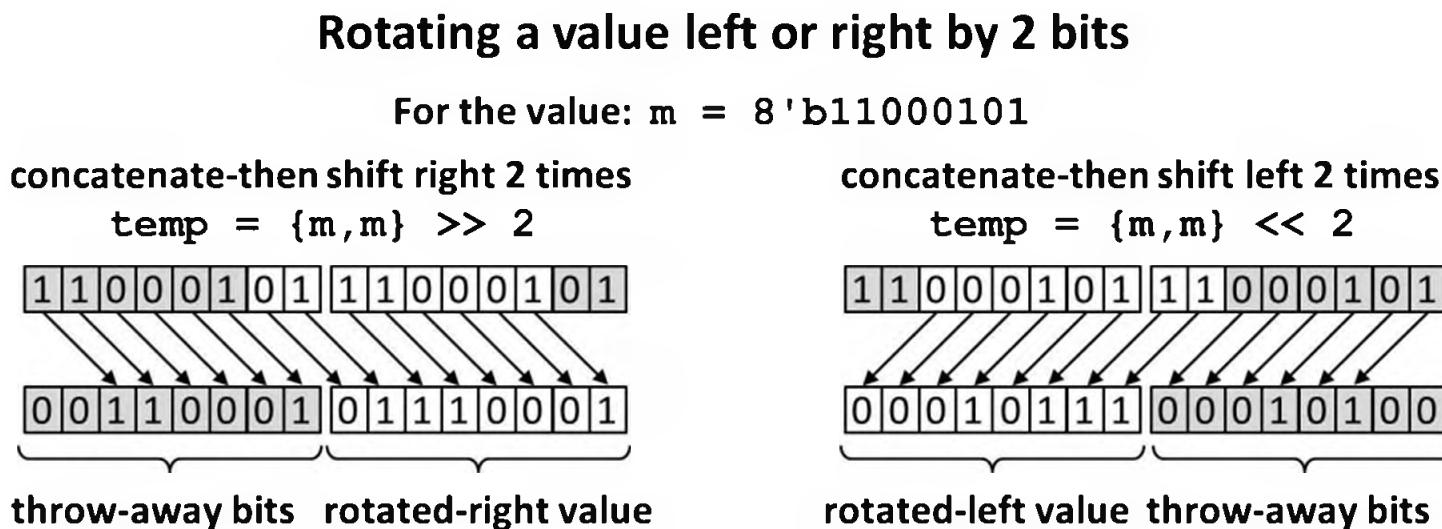
assign temp1 = {in,in} >> rotate_num; // shift right
assign out1 = temp1[7:0]; // select right half

assign temp2 = {in,in} << rotate_num; // shift left
assign out2 = temp2[15:8]; // select left half
```

The rotation works by first concatenating the expression to be rotated to itself, and then shifting the result the desired number of times. The shift operation causes the bits of the concatenated vector to be shifted over, with the effect of rotating the bits of one end of the original value shifting into the bit positions at the other end of the original value. After the shift operation, the half of the concatenated vector containing the rotation result is selected. For a shift-right operation, the right half contains the desired result. For a shift-left operation, the left half contains the desired result.

Figure 5-14 illustrates how the concatenate-then-shift operation moves the bits of a value. The illustration rotates a value two times, but the operation will work for rotating a value up to N times, where N is the number of bits in the original value. (Rotating more than N times causes zeros to shift into the original bit positions, and no longer behaves like a rotate operation).

Figure 5-14: Rotate a variable number of times using concatenate and shift operators



Synthesis of rotate operations. A rotation of a fixed number of times is simply rewiring the bits of a vector, and does not synthesize into any actual logic gates. A rotation of a variable number of bits synthesizes into cascaded multiplexors, similar to a barrel shifter used to shift a variable number of bits. The difference between the variable shift implementation and variable rotation implementation is how the multiplexors are wired together.

Example 5-13 shows a model of a variable left-rotate operation, and Figure 5-15 illustrates how this model might synthesize. The circuit shown in Figure 5-15 is the intermediate synthesis implementation before the design is mapped and optimized to a specific target ASIC or FPGA device. Observe that, in a barrel shifter, the number of multiplexors in each bank (a row or column, depending on the schematic orientation) is the number of bits in the vector to be shifted or rotated. The number of banks required is the number of bits in the shift or rotate control vector. This control vector should not be any larger than is necessary to represent one less than the number of bits in the expression to be rotated. The `$clog2` function can be used to calculate the size of the rotate control signal.

Example 5-13: Performing a rotate operation using concatenate and shift operators

```

//  

// Rotate an input vector left the number of times specified  

// by a rotation factor input.  

//  

module rotate_left_rfactor_times  

#(parameter N = 8)  

(input logic [N-1:0] data_in, // N-bit input  

 input logic [$clog2(N)-1:0] rfactor, // ceiling log2 of N  

 output logic [N-1:0] data_out // N-bit output  

);  

logic [N*2-1:0] temp;  

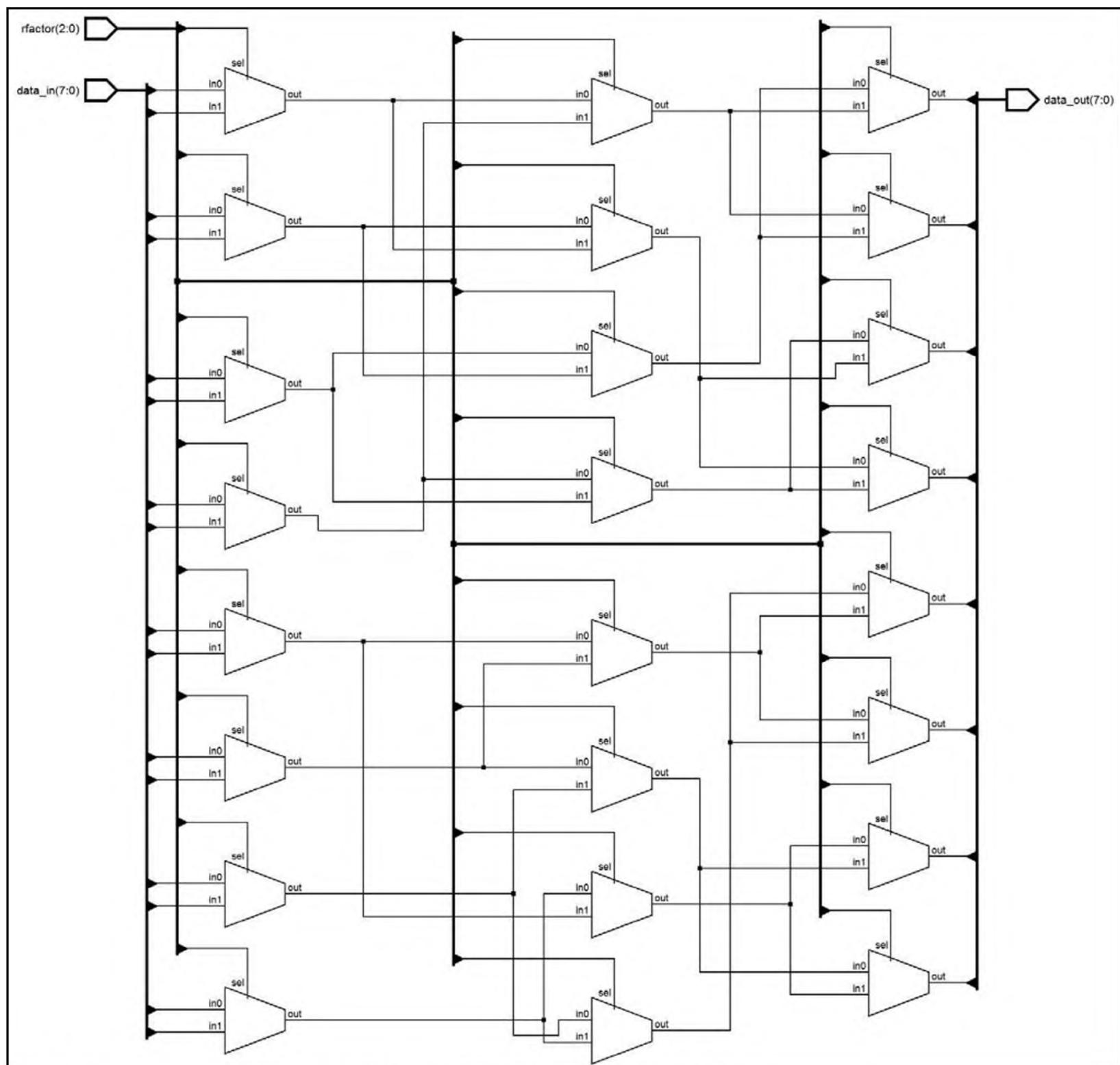
assign temp = {data_in,data_in} << rfactor; // rotate left  

assign data_out = temp[N*2-1:N]; // select left half of temp  

endmodule: rotate_left_rfactor_times

```

Figure 5-15: Synthesis result for Example 5-13: Concatenate and shift (rotate)

Technology independent schematic (no target ASIC or FPGA selected)

Rotate right operation shortcut. A rotate right operation can be performed without a temporary variable by taking advantage of SystemVerilog's assignment statement rules.

```
logic [ 7:0] in, out;
logic [ 2:0] rfactor;

assign out = {in,in} >> rfactor; // assignment truncates
// the upper bits
```

This shortcut works with a rotate-right because the result of the concatenation is a value that is twice the width of value to be rotated, with the right-most half of the concatenation containing the desired rotation result. Assignment statements operate from right to left. When the right-hand side of the assignment has more bits than the left-hand side, the upper bits of the right-hand side are truncated. This truncation yields the desired result for this rotation operation.

Some software tools, however, will generate a warning message whenever the left-hand side and right-hand side of an assignment statement are different vector sizes. This is a useful warning, and should not be treated casually. Most often, a mismatch in assignment sizes is an indication of an error in the declarations or operations in the model. The rotation operation using a concatenation and shift operation is an exception, where the assignment truncation is expected and produces desired result.

The SystemVerilog cast operator can be used to prevent truncation warnings when the truncation is intended. For example:

```
assign out = 8'({in,in} >> rfactor); // cast result to 8 bits
```

The cast operator is discussed in more detail in section 5.15 (page 198).

Rotate operations using loops (non-synthesizable). Although it is possible to code the functionality of a rotate operation for a variable number of times using a loop, the code will not work with most synthesis compilers.

Best Practice Guideline 5-6

Use operators to shift or rotate a vector a variable number of bits. Do not use loops to shift or rotate the bits of a vector a variable number of bits.

Synthesis compilers do not support loops that executes for a variable number of times. Synthesis requires that the number of times a loop will iterate be a static value that is available during compilation. Synthesizing static and data-dependent loop iterations is discussed in Chapter 6, section 6.3.1.1 (page 230).

5.11 Streaming operators (pack and unpack)

SystemVerilog's streaming operators, also referred to as pack and unpack operators, allow reorganized groups of bits within a vector. The SystemVerilog streaming operators can be used in a number of creative ways to pack and unpack data stored in vectors, arrays, and structures. Two synthesizable applications of these operators are shown in this section, reversing the order of bits within a vector, and reversing the order of bytes within a vector. The tokens used for the streaming operators are shown in Table 5-19.

Table 5-19: Streaming operators (pack and unpack) for RTL modeling

Operator	Example Usage	Description
{>> { }}	{>>m{ n } }	Right stream (extract) m-size blocks from n, working from the right-most block towards the left-most block
{<< { }}	{<<m{ n } }	Left stream (extract) m-size blocks from n, working from the left-most block towards the right-most block

The streaming operators pull-out or push-in groups of bits from or to a vector in a serial stream. The streaming operators can be used to either pack data into a vector or unpack data from a vector.

Packing occurs when the streaming operator is used on the right-hand side of an assignment. The operation will pull blocks as a serial stream from the right-hand expression, and pack the stream into a vector on the left-hand side. The bits pulled out can be in blocks of any number of bits. The default is 1 bit at a time, if a block size is not specified.

The following code uses this default of 1-bit blocks with the left-stream operator to extract bits from a vector beginning with the left-most (most-significant) bit and packing them into another variable beginning with the right-most (least-significant) bit. The SystemVerilog assignment operator always works from right to left. Thus, the first block extracted by the streaming operator (which is the left-most bit in the following code) is assigned to the right-most block of the variable on the left-hand side of the assignment. The variable on the left side of the assignment contains the packed result of the stream of blocks extracted from the right side of the assignment.

```

logic [7:0] a, b;

assign a = 8'b11000101;

always_comb begin
    b = { << {a}}; // sets b to 8'b10100011 (bit reverse of a)
end

```

A similar application of the left-stream operator is to reverse the bytes of a vector by extracting and streaming 8-bit block sizes of an expression on the right side of an assignment, and packing the stream into a vector on the left side of the assignment:

```
logic [31:0] in, out;  
assign out = {<<8{in}}; // repack bytes in opposite order
```

Unpacking occurs when a streaming operator is used on the left-hand side of an assignment. Blocks of bits are pulled out of the right-hand expression, and assigned to the expression blocks within the streaming operator.

The following code snippet unpacks a 32-bit vector, and streams the values into separate 8-bit elements of an array.

```
logic [7:0] a [0:3]; // array of 4 8-bit variables  
logic [31:0] in; // 32-bit vector  
  
assign in = 32'hAABBCCDD;  
  
always_comb begin  
    {>>8{a}} = in; // sets a[0]=AA, a[1]=BB, a[2]=CC, a[3]=DD  
end
```

NOTE

At the time this book was written, some synthesis compilers did not support the streaming operators, and some synthesis compilers only supported streaming when the default block size of 1-bit was used, as in the bit reversal code snippet example above.

Before using the streaming operators in RTL models, engineers should make sure the operators are supported by all software tools used in a design flow.

Synthesizing streaming operators. The streaming operators simply select individual or groups of bits from a bus. No logic gates are required to implement the functionality of the streaming operator. At the RTL level of modeling, however, the streaming operators can be a concise way to represent complex functionality, such as reversing the bit or byte order of data that has parameterized vector widths.

Example 5-14 illustrates using the streaming operator to do a bit reversal of a vector, where the width of the vector can be configured using parameter redefinition. Figure 5-12 shows how this left-stream operation might synthesize. The synthesis compiler used for this example placed buffers on the inputs and outputs of the module, but did not utilize any additional gates to perform the operation.

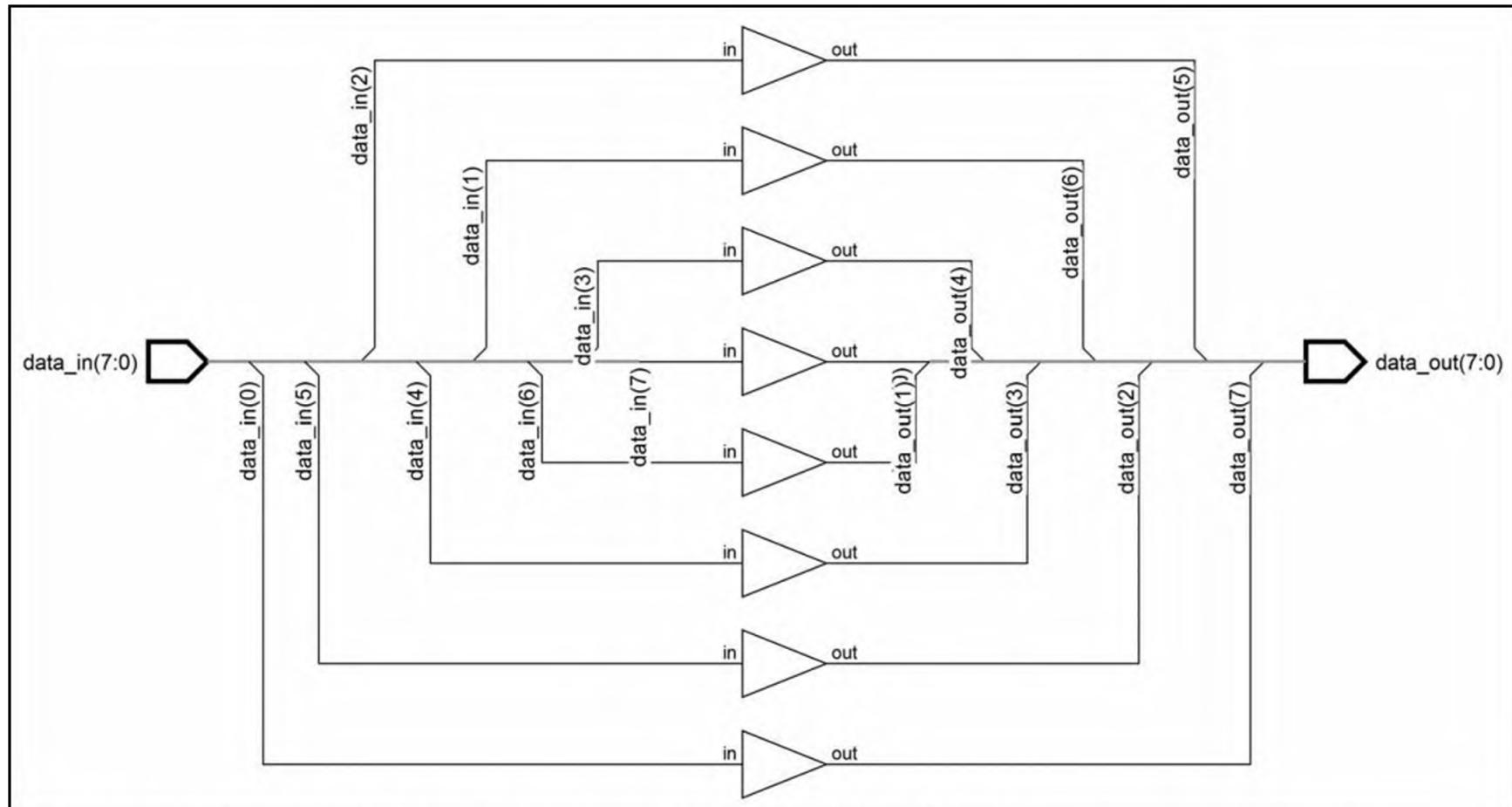
Example 5-14: Using the streaming operator: reverse bits of a parameterized vector

```
module reverse_bits
#(parameter N = 8)
(input logic [N-1:0] data_in,      // N-bit input
 output logic [N-1:0] data_out     // N-bit output
);

assign data_out = { << { data_in } }; // reverse of bit order

endmodule: reverse_bits
```

Figure 5-16: Synthesis result for Example 5-14: Streaming operator (bit reversal)



Technology independent schematic (no target ASIC or FPGA selected)

The synthesis compiler shows buffers to map each bit number of `data_in` to a different bit in `data_out`. These buffers will probably be removed when the technology independent netlist is mapped to a specific ASIC or FPGA target device.

5.12 Arithmetic operators

SystemVerilog has a number of arithmetic operators that calculate a result based on the value or one or more expressions. These operators are listed in Table 5-20.

Table 5-20: Arithmetic operators for RTL modeling

Operator	Example Usage	Description
+	$m + n$	Add: add the value of m to the value of n
-	$m - n$	Subtract: subtract the value of n from the value of m
-	$-m$	Unary minus of the value of m (two's complement of m)
*	$m * n$	Multiply: multiply the value of m by the value of n
/	m / n	Divide: divide the value of m by the value of n
%	$m \% n$	Modulus: remainder of m divided by n
**	$m ** n$	Power: value of m raised to the power of the value of n

All SystemVerilog arithmetic operators are synthesizable, but specific ASICs and FPGAs might have restrictions on what can be implemented at the gate-level in that device. Operations such as multiply, divide, modulus, and power are complex circuits in hardware, and can require a substantial amount of logic gates and propagation timing paths.

When writing RTL models for synthesis, it is important for the design engineer to remember that the final purpose of RTL code is not to be a software program that will run on a general purpose computer. The objective of RTL models is to be an abstract representation of digital logic gates. The simple code:

```
always_ff @(posedge clk)
    out <= a / b;
```

is asking the synthesis compiler to create a gate-level divider that reaches a completed result every clock cycle, with no intermediate pipelined stages. Whether this will be possible depends on a number of factors, such as the design's clock speed, the widths of the numerator and denominator vectors, and the capabilities of the target ASIC or FPGA device.

NOTE

The capabilities and limitations of each specific ASIC or FPGA device can vary widely. RTL models that use the multiply, divide, modulus and power operators should be written to match the capabilities of the target device.

Ideally, arithmetic operations in RTL models could be written without concern as whether the functionality will be implemented as an ASIC or FPGA. This ideal is not always possible when using the multiply, divide, modulus and power operators.

Best Practice Guideline 5-7

For better synthesis Quality of Results (QoR):

- (a) Use shift operators for multiplication and division by a power of 2, instead of the *, /, % and ** arithmetic operators.
 - (b) For multiplication and division by a non-power of 2, use a constant value for one operand of the operation, if possible.
 - (c) For multiplication and division when both operands are non-constant values, use smaller vector sizes, such as 8-bits.
-

Adhering to these guidelines will help to ensure that an RTL models can be synthesized to most target ASIC and FPGA devices.

Design engineers need to take extra steps when writing RTL models where the design specification requires operations that are outside of these suggested guidelines. It might be necessary model a pipelined data path to break an operation into multiple clock cycles. Some synthesis compilers have the ability to do register retiming, an automated process of moving combinational logic in a pipeline to different stages of the pipeline in order to achieve faster clock speeds. Another design technique is to use a Finite State Machine to break a complex arithmetic operation into multiple clock cycles.

Many target ASIC and FPGA devices have predefined gate-level arithmetic blocks or Intellectual Property (IP) models for complex arithmetic operations. These components can be used in place of a SystemVerilog arithmetic operator. The use of built-in gate-level arithmetic blocks or IP models can be very effective for achieving best Quality of Results (QoR) in an implementation. The trade-off is that the design models can become locked to a specific target ASIC or FPGA family. Rewriting, and reverifying, some of the RTL models might be necessary to change to a different target device.

High-level Synthesis (HLS) tools can also be used to map abstract complex operations into either RTL models or directly into logic gates. Where Register Transfer Level (RTL) modeling requires that the design engineer specify exactly what operations need to be done in each clock cycle, High-level Synthesis allows specifying that an operations has to be completed within a specific number of clock cycles. The synthesis compiler then determines how to best implement that requirement. High-level Synthesis is outside the scope of this book, which is on best coding practices for writing RTL models.

5.12.1 Integer and floating-point arithmetic

The arithmetic operators perform different types of operations based on the data types of the operands. The rules are:

- Perform *signed integer arithmetic* if both operands are signed integral values (an integral value is a literal integer value or vectors of 1 or more bits in size).
- Perform *unsigned integer arithmetic* if both operands are integral values, and at least one operand is an unsigned type.
- Perform *floating-point arithmetic* if either operand is a real value (a real value is a literal floating point value or the **real** or **shortreal** type).

The overloaded behavior of these arithmetic operators can simplify writing RTL models of hardware functionality. It is not necessary to use different operators to model unsigned, signed, or floating-point behavior. The same operator can model all three types of behavior. However, this overloading means that engineers must be careful to use the proper data types in order to represent the type of hardware intended.

The following three examples illustrate how operand data types affect the type of arithmetic operation that is performed.

- Example 5-15 illustrates a simple *unsigned adder* with no carry bit.
- Example 5-16 illustrates a *signed adder* with no carry bit.
- Example 5-17 illustrates a *floating-point adder*. (This example is not supported by synthesis compilers.)

Observe that only the port declarations were changed in these three examples. No changes were made to functional code for the RTL code for the adder.

Example 5-15: Using arithmetic operators with unsigned data types

```
module unsigned_adder
#(parameter N = 8)
(input logic [N-1:0] a, b, // N-bit unsigned inputs
 output logic [N-1:0] sum // N-bit unsigned output
);
assign sum = a + b; // adder with no carry
endmodule: unsigned_adder
```

Example 5-16: Using arithmetic operators with signed data types

```
module signed_adder
#(parameter N = 8)
(input logic signed [N-1:0] a, b, // N-bit signed inputs
 output logic signed [N-1:0] sum // N-bit signed output
);

assign sum = a + b; // adder with no carry

endmodule: signed_adder
```

Example 5-17: Using arithmetic operators with real data types

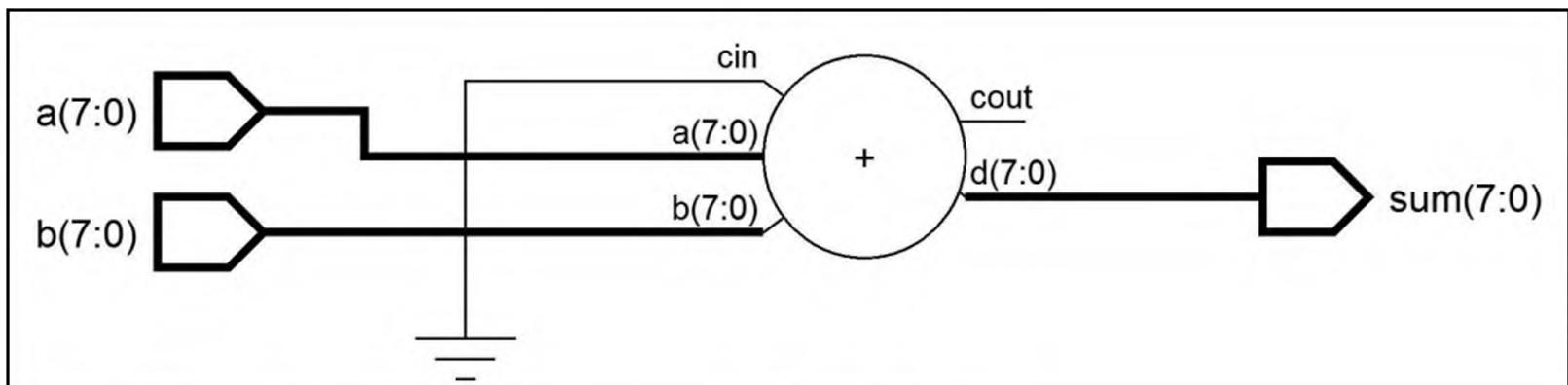
```
module floating_point_adder
(input real a, b, // double-precision floating-point inputs
 output real sum // double-precision floating-point output
);

assign sum = a + b; // floating-point adder

endmodule: floating_point_adder
```

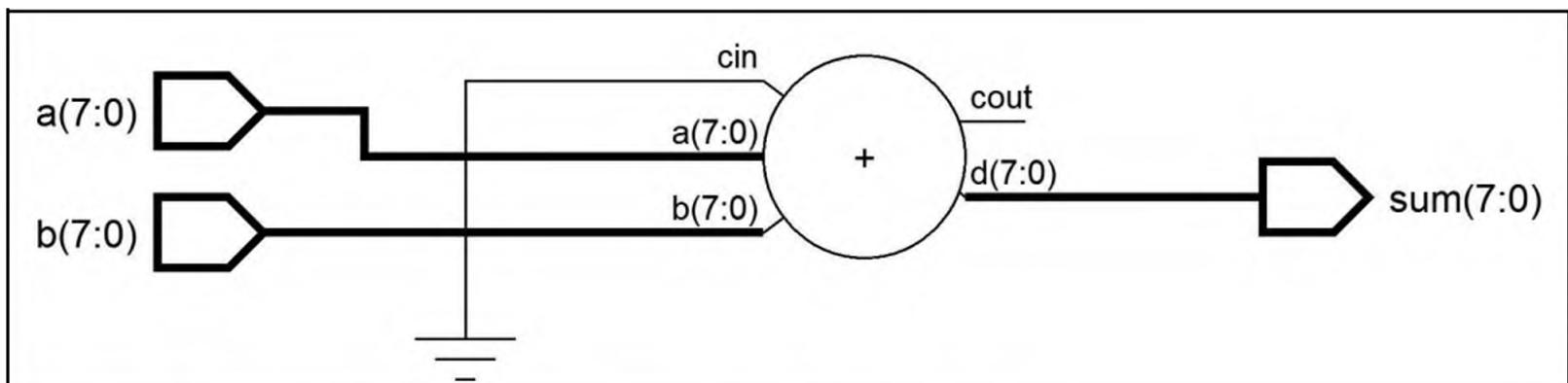
Figures 5-17 shows the Synthesis result for the unsigned adder in Example 5-15, and Figure 5-18 shows the Synthesis result for the signed adder in Example 5-16

Figure 5-17: Synthesis result for Example 5-15: Arithmetic operation, unsigned



Technology independent schematic (no target ASIC or FPGA selected)

Figure 5-18: Synthesis result for Example 5-16: Arithmetic operation, signed



Technology independent schematic (no target ASIC or FPGA selected)

Observe that the Synthesis result for the unsigned adder shown in Figure 5-17 is identical to the Synthesis result for the signed adder shown in Figure 5-18. The reason for this is discussed in section 5.12.2 (page 188) of this chapter.

The synthesis compiler used to generate the implementations shown in Figures 5-17 and 5-18 mapped the RTL adder functionality to a generic adder block that has unused carry-in and carry-out bits. The next step in synthesis would be to target a specific ASIC or FPGA device. The generic adder would be mapped to a specific adder implementation during that step. The device-specific adder might not have these carry-in and carry-out ports, depending on the components available in that specific device.

NOTE

Example 5-18 is not synthesizable, and is shown here to illustrate how data types can affect operations. RTL synthesis compilers typically do not support real (floating-point) expressions. High-level Synthesis (HLS) tools can be used for complex arithmetic design. Floating point and fixed point design is outside the scope of this book.

5.12.2 Unsigned and signed arithmetic might synthesize to the same gates

In simulation, an unsigned adder treats a negative input value as a large positive value. This is because negative values are represent in two's-complement form. The most significant bit (bit 7 for the 8-bit vector examples below) becomes the sign bit, which, when set, indicates that the value is negative. When a negative value with the sign bit set is treated as an unsigned value, the sign bit loses its meaning. That value with its most significant bit set become a large positive value.

For the unsigned adder modeled in Example 5-15, the following inputs values, shown in decimal and binary, produce unsigned values for the outputs:

```
a=1  (00000001), b=1  (00000001) : sum = 2  (00000010)
a=1  (00000001), b=255 (11111111) : sum = 0  (00000000)
a=1  (00000001), b=-3  (11111101) : sum = 254 (11111110)
a=-1 (11111111), b=-3  (11111101) : sum = 252 (11111100)
```

When the same input values are applied to the signed adder modeled in Example 5-16, the decimal results are different.

```
a=1  (00000001), b=1  (00000001) : sum = 2  (00000010)
a=1  (00000001), b=-1 (11111111) : sum = 0  (00000000)
a=1  (00000001), b=-3  (11111101) : sum = -2 (11111110)
a=-1 (11111111), b=-3  (11111101) : sum = -4 (11111100)
```

Observe that, in decimal, the results of the signed and unsigned adders are different. This is because the decimal radix interprets the most significant bit of `sum` as a sign bit. In binary, however, the output values for the unsigned adder and the signed adder are identical. The difference between signed and unsigned operations is not the binary result, it is how the most-significant bit of that result is interpreted. With unsigned types, the most-significant bit is simply part of the value. With signed types, the most-significant bit is a flag, indicating the value is negative.

This similarity in how unsigned and signed types synthesize is true for add, subtract, and multiply operations, but is not true for divide operations. The binary result for divide operations can be different for signed and unsigned operations because divide operations can have fractional results. For example, a signed divide operation of `1 / -1` will result in `-1`, whereas an unsigned divide operation will result in `0`. The reason is that `-1` as an unsigned value is `255`, so the unsigned operation is actually `1 / 255`, which is a fractional result that cannot be represented as an integer.

Best Practice Guideline 5-8

Use unsigned types for all RTL model operations. The use of signed data types is seldom needed to model accurate hardware behavior.

Declaring module ports and internal variables as the `logic` type will infer an unsigned net type for input and inout ports, and an unsigned variable for output ports.

5.13 Increment and decrement operators

SystemVerilog adds the `++ increment operator` and `-- and decrement operator` to the original Verilog language. The tokens for these operators are shown in Table 5-19.

Table 5-21: Increment and decrement operators for RTL modeling

Operator	Example Usage	Description
<code>++</code>	<code>++n</code> <code>n++</code>	pre-increment the value of <code>n</code> by 1, or post-increment <code>n</code> by 1
<code>--</code>	<code>--n</code> <code>n--</code>	pre-decrement the value of <code>n</code> by 1, or post-decrement <code>n</code> by 1

The operand for the `++` and `--` operators must be a vector variable with a size of 1 or more bits. The floating-point types of `real` or `shortreal` can also be used, but these types are not supported by most synthesis compilers.

In a pre-increment operation, the value of the operand is first incremented by 1, and a new value is returned from the operation. For example, in the statement:

```
n = 5;
y = ++n; // y=6, n=6
```

The current value of `n` is first incremented, and the result is assigned to `y`. Thus, after the statement is executed, `y` has the value of 6 and `n` has the value of 6.

In a post-increment operation, the current value of the operand is first returned, then the operand is incremented by 1. In the statement:

```
n = 5;
y = n++; // y=5, n=6
```

The current value of `n` is assigned to `y`, and then `n` is incremented. Thus, after the statement is executed, `y` has the value of 5 and `n` has the value of 6.

These same rules apply to the `--` decrement operator, except that the operand is decremented by 1.

```
n = 5;
y = --n; // y=4, n=4
y = n--; // y=4, n=3
```

5.13.1 Proper usage of increment and decrement operators

The increment and decrement operators are simply a shortcut for the statements:

```
n = n + 1; // same as n++
n = n - 1; // same as n--
```

NOTE

The increment and decrement operators use blocking assignment behavior.

It is important to note that these equivalent statements use blocking assignments. SystemVerilog has two types of procedural assignments: *blocking*, represented with a single equal token (`=`) and *nonblocking*, represented with a less-than-equal token (`<=`). The purpose and proper usage of these assignment types are discussed in Chapter 1, section 1.5.3.5 (page 27). In brief, blocking assignments are used when modeling the behavior of combinational logic, such as digital logic gates, multiplexors and decoders. Nonblocking assignments are used when modeling the behavior of sequential logic, such as flip-flops, registers, counters and pipelines.

Using the wrong type of assignment can lead to simulation race conditions, which is also discussed in Chapter 1, section 1.5.3.5 (page 27). A race condition occurs when a variable is both read from and written to at the same moment of simulation time. If blocking and nonblocking assignments are not used correctly, there can be a difference in the way in which simulation processes this simultaneous read and write, and the way actual logic gates propagate logic value changes. This mismatch in RTL and gate-level behavior can lead to a design that appears to have been fully verified in simulation, but that does not work properly in the actual ASIC or FPGA.

The following code snippet illustrates a proper usage of the increment and decrement operators in a combinational logic model:

```
logic [15:0] data_bus;
logic [ 3:0] count_ones;
always_comb begin
    count_ones = '0;
    for (int i=15; i>=0; i--)
        if (data_bus[i]) count_ones++;
end
```

This next code fragment shows an improper usage of an increment operator:

```
parameter MAX = 12;
logic [7:0] count, data, q;

always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN) count <= '0; // active-low reset
    else       count++; // BUG: increment count value

always_ff @(posedge clk)
    if (count < MAX) q = data + count; // read count value
    else             q = data + MAX;
```

The simulation race condition in the preceding example comes from the sequential logic block, that triggers on a positive edge of `clk`, incrementing the `count` variable at the same time the second sequential logic block is read the value of `count`. The reason the `++` increment operator should not be used in this example is that the sequential logic block assignment to `count` needs to be modeled with a nonblocking assignment, as in:

```
always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN) count <= '0; // active-low reset
    else       count <= count + 1; // increment count
```

Best Practice Guideline 5-9

Only use the increment and decrement operators with combinational logic procedures and to control loops iterations. Do not use increment and decrement to model sequential logic behavior.

The proper place for using blocking assignment behavior is when representing combinational logic. Using the increment and decrement operators to model sequential logic, such as counters, will cause simulation race conditions. Nonblocking assignments are required in order to avoid simulation race conditions in sequential logic procedures.

The most common usage of the increment and decrement operators is with `for` loop control variables, as in the following example.

```

logic [15:0] data_bus;
logic [ 3:0] highest_bit;

always_ff @(posedge clk) begin
    highest_bit <= '0;
    for (int i=0; i<=15; i++)
        if (data_bus[i]) highest_bit <= i;
end

```

In this code snippet, the registered variable being assigned on a positive edge of clock is `highest_bit`, and is assigned using a nonblocking assignment. The loop control variable `i` is a temporary variable that is not stored in the register, and is assigned using the blocking assignment behavior of the `++` operator.

Synthesis requires that loops execute in zero time — they cannot contain delays or clock cycles. In this zero-delay context, the loop iterations will represent combinational logic, even if the loop is within a sequential logic block, as in the preceding code snippet:

5.13.2 An example of correct usage of increment and decrement operators

Example 5-18 below is similar to the `count_ones` code snippet shown earlier in this section. This more complete example uses a parameter for the `data_bus` size so that the model can be scaled to different bus widths. An 8-bit bus size is used in order to keep the synthesized schematic size small for the page size of this book. Figure 5-19 shows the resulting synthesis schematic for this example.

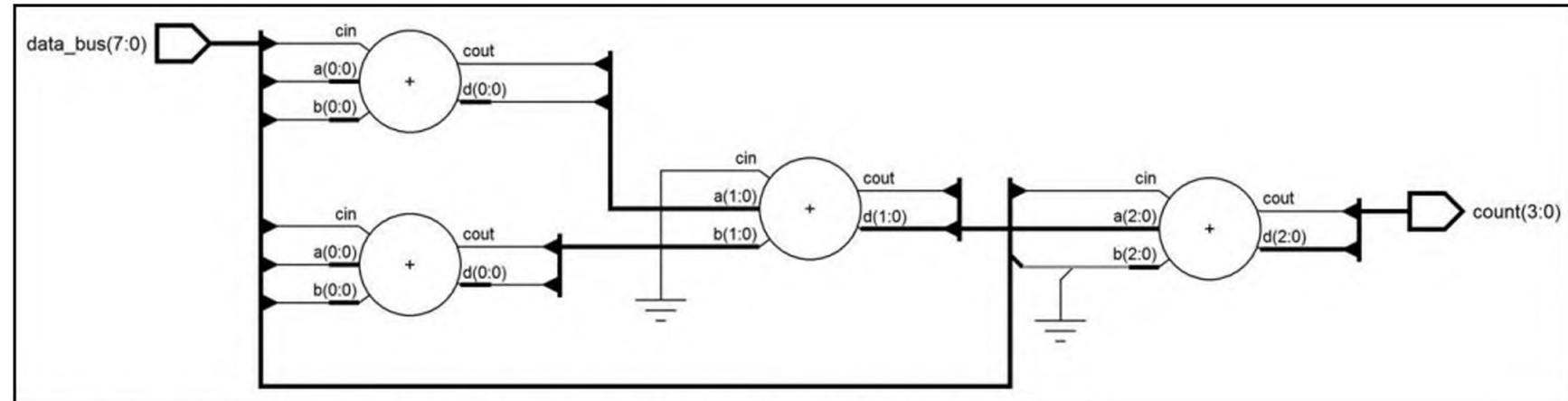
Example 5-18: Using increment and decrement operators

```

module count_ones
#(parameter N = 8)
(input logic [N-1:0] data_bus,
 output logic [$clog2(N):0] count // compute count width based
); // on the size of data_bus

always_comb begin
    count = '0;
    for (int i=N-1; i>=0; i--)
        if (data_bus[i]) count++;
end
endmodule: count_ones

```

Figure 5-19: Synthesis result for Example 5-18: Increment and decrement operators

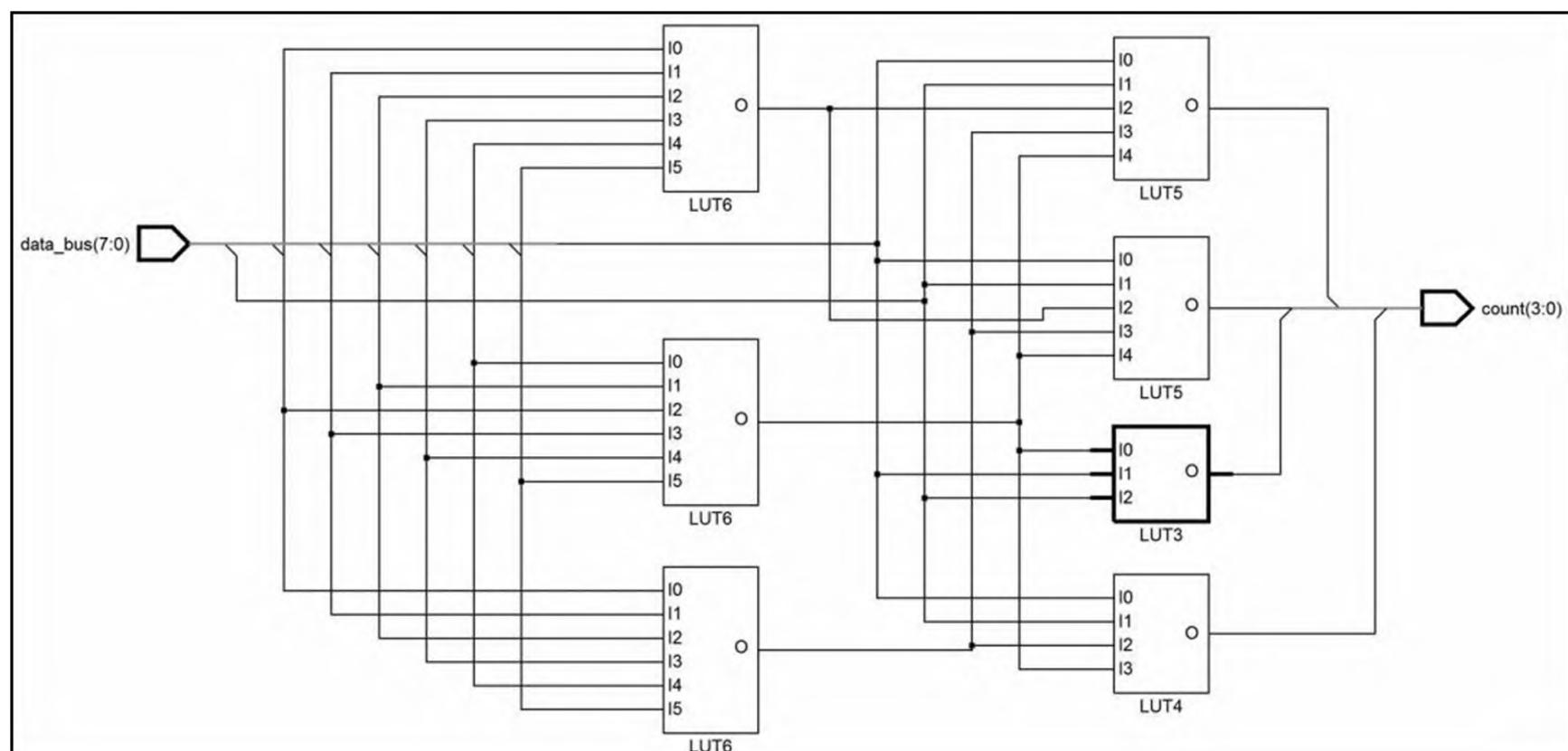
Technology independent schematic (no target ASIC or FPGA selected)

The synthesis compiler used to generate the implementation shown in Figure 5-19 mapped the RTL bit counter functionality to a series of generic adders. The adders represent the `++` operator that increments the count variable.

The decrement operator that is part of the `for` loop does not appear in the synthesized results. This is because the `for` loop in the RTL model is unrolled, to create adders for each pass of the loop. The generic adder has a carry-in input, and therefore can add up to 3 bits of the `data_bus`. For an 8-bit `data_bus`, 3 of these generic adders are instantiated from the `for` loop, and a 4th adder is used to sum the results of these 3 adders.

The next step in synthesis is to target a specific ASIC or FPGA device. The generic adders will be mapped to a specific implementation. The mapping process might perform further optimizations based on the adder types available in the target device.

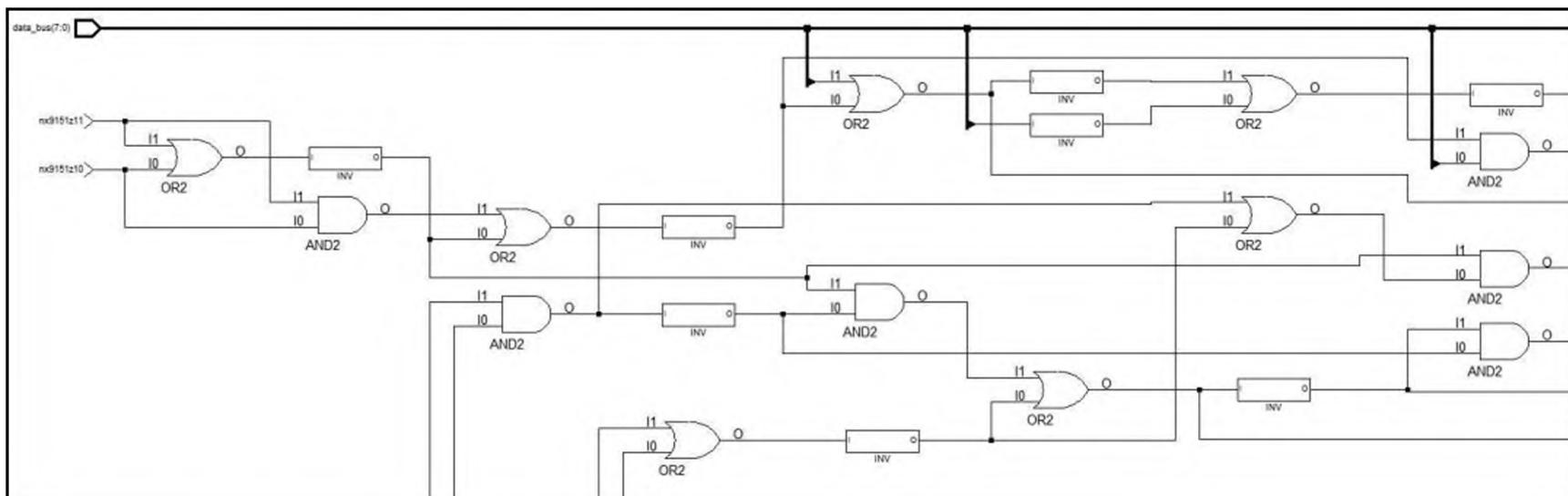
Figure 5-20 shows the results of synthesis targeting the generic increment adders to a Xilinx Virtex®-7 FPGA. The adders were replaced by functionality programmed into the device's LUTs (Look Up Tables). Each LUT contains a number of basic logic gates. The gates, and connections between them, can be programmed to implement specific functionality, such as the series of increment operation.

Figure 5-20: Synthesis result after mapping to a Xilinx Virtex®-7 FPGA

*Technology specific schematic generated by Mentor Precision Synthesis™ compiler

Figure 5-21 shows the results of synthesis targeting the generic increment adders to a Xilinx CoolRunner™-II FPGA. In this device, the increment functionality was mapped to discrete AND, OR and inverter gates. Again, only a portion of the schematic is shown, in order to focus on how the incrementers were implemented.

Figure 5-21: Synthesis result after mapping to a Xilinx CoolRunner™-II CPLD



*Technology specific schematic generated by Mentor Precision Synthesis™ compiler

5.13.3 Compound operations with increment and decrement operators

Multiple operations can be combined into a single statement. For example:

```
sum = a + b - c;
```

SystemVerilog has operator precedence rules to define the order in which multiple operations are performed. These rules are discussed in more detail in section 5.16.

NOTE

The increment/decrement operator has the same precedence as several other arithmetic operators. The order in which operations can be evaluated is ambiguous in a compound expression that uses increment/decrement in combination with other arithmetic operators.

When the increment or decrement operator is used in conjunction with other arithmetic operators that have the same evaluation precedence, the simulator can evaluate the operators in any order. For example:

```
n = 5;
y = n + ++n; // y could be assigned 11 or 12
```

In this code snippet, a simulator could either:

- Evaluate the `+` operator first, and then the `++` operator. In this case, simulation will use the current value of `n`, which is 5, plus the return of the pre-increment `++` operation, which is 6. The result of the compound operations is 11 ($5 + 6$).

- Evaluate the `++` operator first, and then the `+` operator. In this case, simulation will use the new value of `n`, which is 6, plus the return of the pre-increment `++` operation, which is 6. The result of the compound operations is 12 ($6 + 6$).

This ambiguity of evaluation order can lead to different results in different simulators, or, even more dangerous, a difference in the verified RTL simulation and the gate-level implementation from synthesis.

5.13.4 An anecdotal story on the increment and decrement operators

The original Verilog language was first introduced circa 1985. The author for most of the original Verilog was Phil Moorby. Phil leveraged many of the programming statements and operators in the C language as the foundation for Verilog, with many important differences in order to make Verilog syntax and semantic rules appropriate for modeling hardware behavior. (C is purely a software language, and does not inherently understand or behave correctly for representing hardware behavior such as sequential logic, propagation delays and concurrency.)

Although many of the operators in the original Verilog come from the C language, Phil did not include the C language `++` and `--` increment and decrement operators. This meant constructs such as a `for` loop had to be coded in a more awkward style, such as:

```
for (i = 0; i<=15; i = i + 1) ...
```

The SystemVerilog standards committee added the increment and decrement operators nearly 15 years after Phil Moorby created the original Verilog language. The author of this book had the opportunity to give a presentation on the features SystemVerilog added to the original Verilog language at the company where Phil Moorby was then working, and Phil was in the audience for this presentation. When the increment and decrement operators were discussed, the author looked over to Phil, and asked “Why didn’t you include these operators in the original Verilog language?” Phil scowled at being singled out, and then folded his arms across his chest and replied “Because I don’t like them!”

That is the simple reason that the original Verilog did not include the C `++` and `--` operators. Phil did not elaborate as to why he did not like these operators. Perhaps it was because of the hazards discussed in this section on potential race conditions and evaluation order of compound operations. It should be noted that Phil was also an active participant on the standards committee that defined SystemVerilog, and supported the addition of the increment and decrement operators in SystemVerilog.

5.14 Assignment operators

SystemVerilog added C-like assignment operators to the original Verilog language. These operators combine an operation along with a blocking assignment back to the first operand of the operator. For example, the assignment operation `a += b;` is a shortcut for the statement `a = a + b;.`

Table 5-22 lists the assignment operators generally supported by RTL logic synthesis compilers.

Table 5-22: Assignment operators for RTL modeling

Operator	Example Usage	Description
<code>+=</code>	<code>n += m</code>	add <code>m</code> to <code>n</code> and assign result to <code>n</code>
<code>-=</code>	<code>n -= m</code>	subtract <code>m</code> from <code>n</code> and assign result to <code>n</code>
<code>*=</code>	<code>n *= m</code>	multiply <code>n</code> by <code>m</code> and assign result to <code>n</code>
<code>/=</code>	<code>n /= m</code>	divide <code>n</code> by <code>m</code> and assign result to <code>n</code>
<code>%=</code>	<code>n %= m</code>	divide <code>n</code> by <code>m</code> and assign remainder to <code>n</code>
<code>&=</code>	<code>n &= m</code>	bitwise AND <code>m</code> with <code>n</code> and assign result to <code>n</code>
<code> =</code>	<code>n = m</code>	bitwise OR <code>m</code> with <code>n</code> and assign result to <code>n</code>
<code>^=</code>	<code>n ^= m</code>	bitwise Exclusive OR <code>m</code> with <code>n</code> and assign result to <code>n</code>
<code><<=</code>	<code>n <<= m</code>	bitwise left-shift <code>n</code> by the number of times indicated by <code>m</code> and assign result to <code>n</code>
<code>>>=</code>	<code>n >>= m</code>	bitwise right-shift <code>n</code> by the number of times indicated by <code>m</code> and assign result to <code>n</code>
<code><<<=</code>	<code>n <<<= m</code>	arithmetic left-shift <code>n</code> by the number of times indicated by <code>m</code> and assign result to <code>n</code>
<code>>>>=</code>	<code>n >>>= m</code>	arithmetic right-shift <code>n</code> by the number of times indicated by <code>m</code> and assign result to <code>n</code>

NOTE

The assignment operators use blocking assignment behavior.

SystemVerilog has two types of procedural assignments: *blocking*, represented with a single equal token (`=`) and *nonblocking*, represented with a less-than-equal token (`<=`). The purpose and proper usage of these assignment types are discussed in Chapter 1, section 1.5.3.5 (page 27). In brief, blocking assignments are used when modeling the behavior of combinational logic, such as digital logic gates, multiplexors and decoders. Nonblocking assignments are used when modeling the behavior of sequential logic, such as flip-flops, registers, counters and pipelines. Using a blocking assignment, including any of these assignment operators, to assign a value to the out-

put of a sequential logic block can result in simulation race conditions, which can lead to the RTL model behavior not matching the synthesized gate-level behavior.

Example 5-19 illustrates using assignment operations in a simple combinational logic block. Figure 5-22 shows the synthesis output for this example.

Example 5-19: Using assignment operators

```

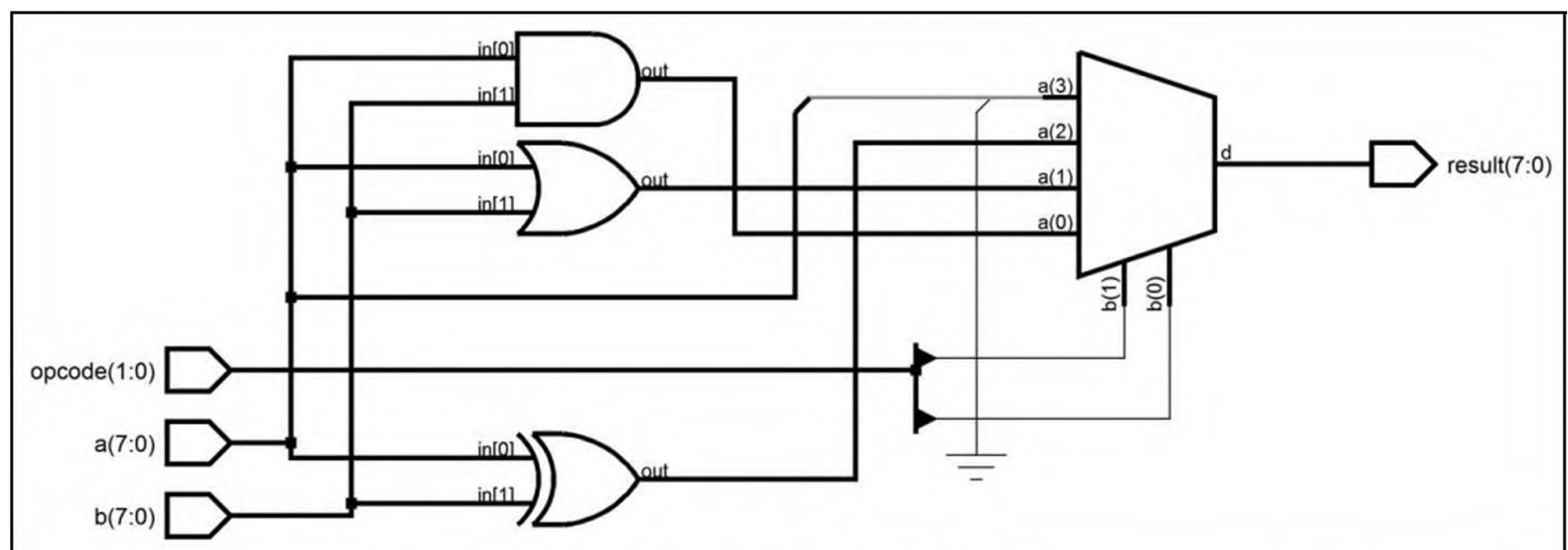
package bitwise_types_pkg;

typedef enum logic [1:0] {AND_OP, OR_OP, XOR_OP, RS1_OP} op_t;
endpackage: bitwise_types_pkg

module bitwise_unit
    import bitwise_types_pkg::*;
    #(parameter N = 8)
    (input logic [N-1:0] a, b,
     input op_t           opcode,
     output logic [N-1:0] result
    );
    begin
        always_comb begin
            result = a; // transfer a input to result output
            case (opcode) // modify result based on opcode
                AND_OP: result &= b;
                OR_OP : result |= b;
                XOR_OP: result ^= b;
                RS1_OP: result >>= 1;
            endcase
        end
    endmodule: bitwise_unit

```

Figure 5-22: Synthesis result for Example 5-19: Assignment operators



Technology independent schematic (no target ASIC or FPGA selected)

Assignment operators can be a convenient shortcut in verification code, but it is the author's opinion that these operators have little value in RTL modeling. A shortcut should reduce the number of lines of code, reduce the risk of coding errors, and help

make code more understandable. The author feels that these assignment operators do not meet any of these objectives when used in a synthesizable RTL context.

Observe that in the RTL model shown in Example 5-19, an intermediate assignment to `result` had to be made before the case statement, in order to use `result` as the first operand of the assignment operators in the case statement. This extra line of code is not needed when using regular assignments statements and operators. The following code snippet shows how the code in Example 5-19 can be modeled more concisely, and easier to read, when the assignment operators are not used.

```
always_comb begin
  case (opcode)
    AND_OP: result = a & b;
    OR_OP : result = a | b;
    XOR_OP: result = a ^ b;
    RS1_OP: result = a >> 1;
  endcase
end
```

5.15 Cast operators and type conversions

SystemVerilog provides a *cast operator* that allows explicitly changing the type, size or signedness of an expression. The three forms of the cast operator are listed in Table 5-23.

Table 5-23: Cast operators for RTL modeling

Operator	Example Usage	Description
<code><type>'()</code>	<code>int' (n)</code>	convert n to an int data type
<code><size>'()</code>	<code>16' (n)</code>	convert n to be 16 bits wide
<code><signedness>'()</code>	<code>signed' (n)</code>	convert n to a signed type

For those familiar with the C language, it should be noted that the syntax for type casting is different than C. SystemVerilog uses the format `<type>'(<expression>)`, whereas C uses the format `(<type>) <expression>`. The different syntax is necessary to maintain backward compatibility with how the original Verilog languages uses parentheses, and to provide the additional casting capabilities of size and sign casting that are not in C.

SystemVerilog is a loosely typed language, meaning an implicit conversion automatically happens when an expression of one type or size is assigned to an expression of a different type or size. Some simple examples of these loosely typed conversion are:

```
logic [15:0] u16; // 16-bit unsigned 4-state variable
logic [31:0] u32; // 32-bit unsigned 4-state variable
int          s32; // 32-bit signed 2-state variable
real         r64; // double-precision floating-point var.

initial begin
    u32 = u16; // u16 is left-extended to be 32 bits wide
    u16 = u32; // u32 is left-truncated to be 16 bits wide
    s32 = u32; // u32 is converted to a signed 2-state value
    u32 = s32; // s32 is converted to an unsigned 4-state val
    r64 = s32; // s32 is converted to a floating-point value
    s32 = r64; // r64 is rounded to a 32-bit integer value
end
```

An implicit type or size conversion can also occur as context-dependent operations are evaluated, as discussed in section 5.1.3 (page 144). In the statement:

```
assign u32 = u32 + u16; // 32-bit add operation
```

Arithmetic operators, such as `+`, require that both operands be the same type and size. All operands will be expanded to the largest vector size before the operation is performed. Therefore, in the operation `u32 + u16`, `u16` will first be converted to a 32-bit size by left-extending its value.

Another implicit conversion that can automatically occur is from a signed value to an unsigned value, or vice versa. In the statement:

```
assign s32 = s32 < u32; // 32-bit unsigned comparison
```

The signed `s32` value will be implicitly converted to an unsigned value. Context-dependent operations (see section 5.1.3, page 144) only perform a signed operation if both operands are signed. If one of the operands is unsigned, an unsigned operation is performed. In the less-than comparison of `s32 < u32`, `s32` is first converted to an unsigned value because `u32` is unsigned.

These conversion rules are defined in the IEEE 1800 SystemVerilog standard, so all software tools that use SystemVerilog, including simulators and synthesis compilers, perform the same conversions. The conversion rules that most often occur in RTL modeling are discussed in section 5.1.3 (page 144) of this chapter. Refer to the IEEE standard for a full description of all possible conversions that can occur when an expression of one type or size is assigned to an expression of another data type or size.

Observe that data can be lost in some of these loosely-typed conversions. In the assignment `u16 = u32;`, the left-most 16 bits are truncated. The value that was in those upper two bytes of `u32` are lost. In the assignment: `s32 = r64;`, the double-precision floating value is rounded off. Any precision of the decimal accuracy is lost, and any value greater than what a 32-bit integer can store is lost.

5.15.1 Type casting

SystemVerilog will perform implicit type casting when: a) an operation has a mix of operand types, or b) an expression of one type is assigned to an expression of another type. For the most part, this implicit casting will do the right thing, and will synthesize to the desired gate-level functionality. One purpose for the use of type casting in RTL modeling is to either make the implicit type conversion more obvious by doing an explicit type cast, or to do something different than the implicit conversion rules. A second purpose for using type casting in RTL models is when assigning values to enumerated variables, which do not have implicit conversion rules the way other SystemVerilog data types have.

Assignments to enumerated type variables. Enumerated variables are more strongly typed than most other SystemVerilog data types. An enumerated variable is defined to have a legal list of values. Assigning an enumerated variable a value outside of that legal list is an error, rather than being implicitly converted to the enumerated type. Enumerated variables and the assignment rules are discussed in more depth in Chapter 4, section 4.4 (page 114).

The following code snippet illustrates a decoder that assigns to a 3-bit enumerated variable called `opcode`. One branch of the decoder attempts to assign 3 bits from a data vector to the enumerated variable. Unfortunately, this assignment violates the rules for enumerated variable assignments and will not compile, even though it is the desired functionality.

```

typedef enum logic {ARITHMETIC, BRANCH} instruction_t;
typedef enum logic [2:0] {NOP, ADD, SUB, MULT,
                         DIV, AND, OR, XOR} opcode_t;

instruction_t instruction;
opcode_t      opcode;
logic [15:0]   data;

always_comb begin
    case (instruction)
        ARITHMETIC : opcode = data[2:0]; // illegal assignment
        BRANCH     : opcode = NOP;
    endcase

```

end

A type cast can be used to explicitly convert the 3-bit value from data to the enumerated type, making this assignment legal.

```
ARITHMETIC : opcode = opcode_t'(data[2:0]);
```

Mixed integer and floating-point operations. The following code snippet shows a compound operation with a mix of integer and floating-point types.

```
parameter PI = 3.14159;
logic [31:0] a, b, result;

assign result = a + (b * PI);
```

Because PI in this example is a fixed-point value, SystemVerilog's automatic type conversion will convert all operands in this compound expression to **real** values, and perform floating-point operations. Both the multiply operation (*****) and the add operation (**+**) will be simulated as double-precision floating operations. While floating-point operations will maintain the full accuracy of using PI to 5 decimal places, floating-point arithmetic units can be larger and slower than integer arithmetic units when implemented in hardware.

Type casting provides a means to specify that a data type conversion should occur at any point during the evaluation of an expression. The following code snippet casts the floating-point result of **b * PI** to an **integer** type. Now the two operands to the add operation will be 32-bit vector types, and will simulate and synthesize as an unsigned 32-bit integer adder.

```
assign result = a + integer'(b * PI);
```

Note that only built-in type keywords and user-defined types can be used with type casting. The following code snippet is illegal because it combines the built-in logic keyword with a vector size specification.

```
assign result = a + logic[31:0]'(b * PI); // illegal syntax
```

Specifying a vector size for size casting can be done by using a user-defined type, as follows:

```
typedef logic [31:0] DTYPE;
DTYPE a, b, result;
assign result = a + DTYPE'(b * PI);
```

SystemVerilog allows data types to be parameterized by using the **parameter type** keyword pair. A type parameter can be used with type casting.

```
parameter type DTYPE = logic [31:0];
DTYPE a, b, result;
assign result = a + DTYPE'(b * PI);
```

5.15.2 Size casting

The implicit conversion of an expression of one vector size to another vector size is widely used in RTL modeling. Perhaps one of the most common places an implicit size conversion occurs is with the literal values of 0 and 1. For example:

```
logic [7:0] count;

always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN) count <= 0;                      // active-low reset
    else      count <= count + 1; // increment count
```

The assignment statements of `count <= 0` and `count <= count + 1` are intuitive, and a common style of coding resets and increments. The assignment statements are functionally correct, but there are actually multiple size mismatches that require implicit size casting.

An implicit size conversion occurs in the assignment `count <= 0`. The number 0 is an unsized literal number, which defaults to a 32-bit value (see Chapter 3, section 3.2, page 62, for literal value rules), whereas `count` is an 8-bit variable. The assignment statement implicitly truncates the upper 24 bits of the literal 0, converting the 32-bit value to an 8-bit value. Synthesis compilers will automatically remove any left-most bits of an expression that are never used.

Two implicit size conversions occur in the assignment `count <= count + 1`. First, the number 1 is an unsized literal number, which defaults to a 32-bit value. The arithmetic `+` operator requires that both operands be the same vector size. Therefore, `count` is implicitly converted to a 32-bit vector by left-extending with zeros (which has no effect on its current value) before it is added to the literal value of 1. The second implicit size conversion occurs when the 32-bit result of the operation is assigned back to the 8-bit `count` variable. SystemVerilog makes assignments from the right-most bit towards the left-most bit. Thus, only the lower 8 bits of the operation result are assigned to `count`, and the upper 24 bits are implicitly truncated. These implicit size conversions occur automatically, and do not affect simulation and synthesis Quality of Results (QoR).

Functionally correct implicit size conversions. There is nothing functionally wrong with the statements `count <= 0` and `count <= count + 1`. This RTL model of the counter is taking advantage of SystemVerilog's implicit size conversions. The statement will simulate with the correct functionality, and synthesize to the correct gate-level implementation.

Another example of taking advantage of implicit size conversion was shown in Section 5.10.2 (page 177). The example used implicit size conversion to rotate a vector a variable number of times. The vector was concatenated to itself, and then shifted the result of the concatenation. For example:

```
logic [7:0] in, out;
logic [2:0] rfactor;
assign out = {in,in} >> rfactor; // variable rotate right
```

The variable `in` is 8-bits wide. The result of the concatenation `{in, in}` is a 16-bit value, which is being assigned to the 8-bit variable `out`. The upper 8 bits of the concatenation and shift result are implicitly truncated during the assignment, and so only the lower 8 bits are transferred to `out`. This is the desired effect of rotating right a variable number of times. The code takes advantage of the implicit size conversion defined in the SystemVerilog language.

Functionally incorrect implicit size casting. The following code snippet, a variable rotate-left operation, illustrates a design error resulting from an assignment mismatch and the implicit size truncation that occurs.

```
logic [7:0] in, out;
logic [2:0] rfactor;
assign out = {in,in} << rfactor; // variable rotate left
```

In this code, the desired bits of the rotation are in the upper 8 bits of the 16-bit operation result. The assignment, however, truncates those upper 8 bits. An example warning that a lint checker program might generate for this implicit truncation is^{*}:

```
Warning Rhs width '16' with shift (Expr: '({in,in} << rfactor)') is more than lhs width '8' (Expr: 'out'), this may cause overflow
```

*Partial output report generated by Synopsys Spyglass Lint® RTL style checker.

As a rotate-left, the implicit truncation is a design bug, and this lint checker warning helps the designer recognize that there is a problem in the code. A correct, synthesizable way to model this variable rotate-left operation is to use an intermediate 16-bit variable to store the concatenate and shift result, as shown earlier in this chapter in Example 5-13 (page 179).

Implicit size conversion warning messages. The IEEE 1800 SystemVerilog standard does not require assignment size mismatch warnings. Most SystemVerilog simulators and synthesis compilers do not generate these warnings, assuming — and trusting — that the design engineer deliberately intended to have a mismatch in the assignment sizes.

On the other hand, lint checkers (tools that verify that code adheres to RTL modeling guidelines) will generate warnings when the expressions sizes on the left-hand and right-hand side of an assignment do not match. These truncation size mismatch warnings can be useful if there is an error in the code and the designer's intent is to have the same vector size on both sides of an assignment.

The size mismatch in the left-rotate example above is a design mistake. The mismatch warning message generated by a lint checker (but probably not by simulators or synthesis compilers) is a desirable warning, that can find and prevent design bugs.

The size mismatch in the rotate and counter examples shown earlier in this section, however, are false warnings. The implicit size conversion is functionally correct in both simulation and synthesis. A lint warning is neither warranted nor wanted. Engi-

neering time can be lost analyzing a false warning to determine that the truncation is OK in this circumstance. These false warnings need to be ignored, which adds a risk of then accidentally ignoring other size mismatch warnings that might have indicated a design error. In a larger design, there can be hundreds of false warnings regarding implicit size and type conversions. These false warnings can hide a warning message for an incorrect or undesired size or type mismatch. This is a serious problem!

Using size casting to prevent false size mismatch warnings. The cast operator can help to make code more self-documenting and intuitive, as well as eliminating false warning messages. The variable rotate-right operation can be coded to explicitly show that the result of the concatenate/shift operation is to be 8 bits wide.

```
assign out = 8'({in,in} >> rfactor); //variable rotate right
```

Size casting follows the same rules as assignment statements. 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. (The example above is an unsigned expression, because the result of a concatenation is always unsigned.)

The size specified with the cast operator can be a run-time constant, which allows for parameterized modules to scale appropriately when parameter values are redefined. For example:

```
parameter N = 8;
logic [N-1:0] in; // N-bit vector
logic [$clog2(N):0] rfactor; // calculate max rotate size
logic [N-1:0] out; // N-bit vector

assign out = N'({in,in} >> rfactor); //variable rotate right
```

The **\$bits** system function can also be used for the size of a size cast. For example:

```
assign out = $bits(out)'({in,in} >> rfactor);
```

Example 5-20 shows the full code for a variable rotate-right operations that uses size casting to eliminate false lint warnings. Figure 5-23 shows the results of synthesizing this example.

Example 5-20: Using size casting

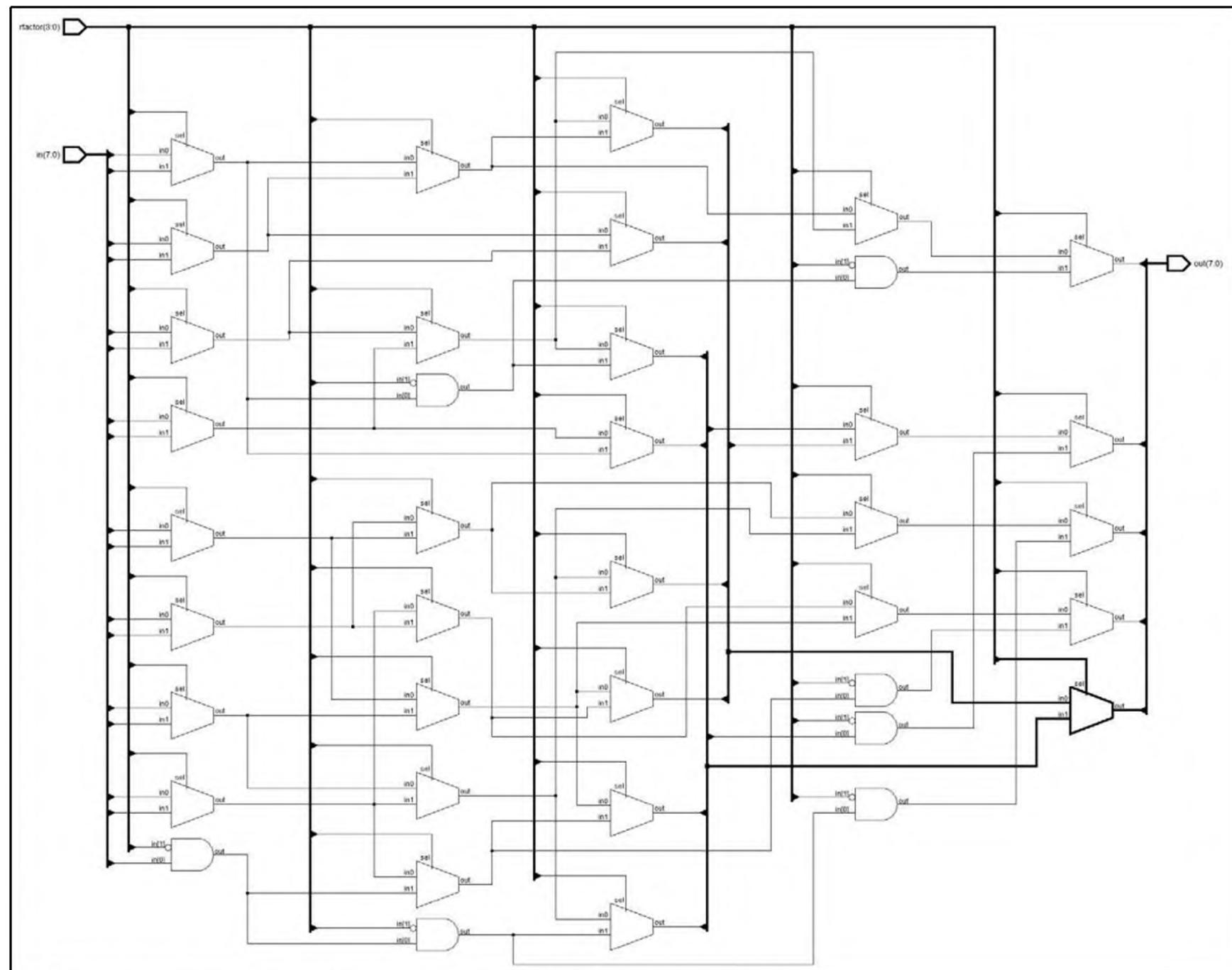
```

module rotate_right_n
#(parameter N = 8)
(input logic [N-1:0] in, // N-bit input
 input logic [$clog2(N):0] rfactor, // calculate max size
 output logic [N-1:0] out // N-bit output
);

assign out = N'({in,in} >> rfactor); // variable rotate right

endmodule: rotate_right_n

```

Figure 5-23: Synthesis result for Example 5-20: Size casting

Technology independent schematic (no target ASIC or FPGA selected)

Example 5-20 will simulate and synthesize correctly, with or without the size casting. The purposes of the size casting are to: (1) make the code more self-documenting that only N bits of the concatenate result are being used, and (2) perhaps more importantly, to eliminate problematic false warnings from RTL lint checkers.

5.15.3 Signedness casting

SystemVerilog will do implicit signedness conversions when the operands in context-dependent operations (see section 5.1.3, page 144) have a mix of signed and unsigned expressions. This implicit conversion is not obvious, and sometimes is not the conversion desired. The primary purpose for signedness casting in RTL modeling is to explicitly ensure that all operands in an operation are either signed or unsigned.

Mixed operand signedness. The implicit signedness conversion rule for when the operands in a context-dependent operation have mixed signedness, is that the signed expression will be converted to an unsigned value.

The following code snippet shows a less-than relational operation with a signed operand and an unsigned operand.

```

logic      [7:0] u1; // 8-bit unsigned variable
logic signed [7:0] s1; // 8-bit signed variables

initial begin
    s1 = -5;
    u1 = 1;
    if (s1 < u1)
        $display("%0d is less than %0d", s1, u1);
    else
        $display("%0d is equal or greater than %0d", s1, u1);
end

```

When simulated, this code snippet will display the message:

-5 is equal or greater than 1

It might seem that -5 is less than 1, and yet this code evaluates -5 as being greater than 1. This happens because SystemVerilog's implicit type conversion will change the `s1` value to unsigned, so as to match the unsigned type of `u1`. The result of this conversion is that the value of -5 becomes 251. (The twos-complement of an 8-bit -5 is 1111011 in binary, which, when treated as an unsigned value, is 251 in decimal.)

Both operands of a context-determined operation must be signed in order for the operation to be signed. Signedness casting provides a means to specify that a data type conversion should occur at any point during the evaluation of an expression. The following code snippet uses casting to explicitly convert `u1` to a signed expression. The operation will now correctly evaluate that -5 is less than 1.

```

if (s1 < signed'(u1))
    $display("%0d is less than %0d", s1, u1);
else
    $display("%0d is equal or greater than %0d", s1, u1);

```

Conversely, this example can be explicitly coded as an unsigned comparator by casting `s1` to an unsigned expression.

```

if (unsigned'(s1) < u1)
    $display("%0d is less than %0d", s1, u1);
else
    $display("%0d is equal or greater than %0d", s1, u1);
end

```

Explicitly casting s1 to unsigned does the same conversion that would implicitly occur. However, using an explicit signedness cast make the code clear that the intent is to have an unsigned operation, which is not obvious when an implicit automatic conversion occurs.

The SystemVerilog signedness cast operator performs the same conversion as the Verilog **\$signed** and **\$unsigned** system functions that were in Verilog-2001. Both signedness casts and the **\$signed** and **\$unsigned** system functions are synthesizable, but the signedness cast operator is more consistent with the syntax used by type and size casting.

Example 5-21 illustrates a small RTL model that uses the less-than, greater-than and equality comparison operators. Figure 5-24 shows how this model might synthesize.

Example 5-21: Using sign casting for a mixed signed and unsigned comparator

```

// Set lt, eq and gt flags based on if s is less-than, equal-to
// or greater-than u, respectively
//
module signed_comparator
    #(parameter N = 8)                                // data size
    (input logic                               clk, // clock input
     input logic                               rstN, // active-low async reset
     input logic signed [N-1:0] s,      // scalable input size
     input logic [N-1:0] u,      // scalable input size
     output logic lt,        // set if s less than u
     output logic eq,        // set if s equal to u
     output logic gt        // set if s greater than u
    );

```

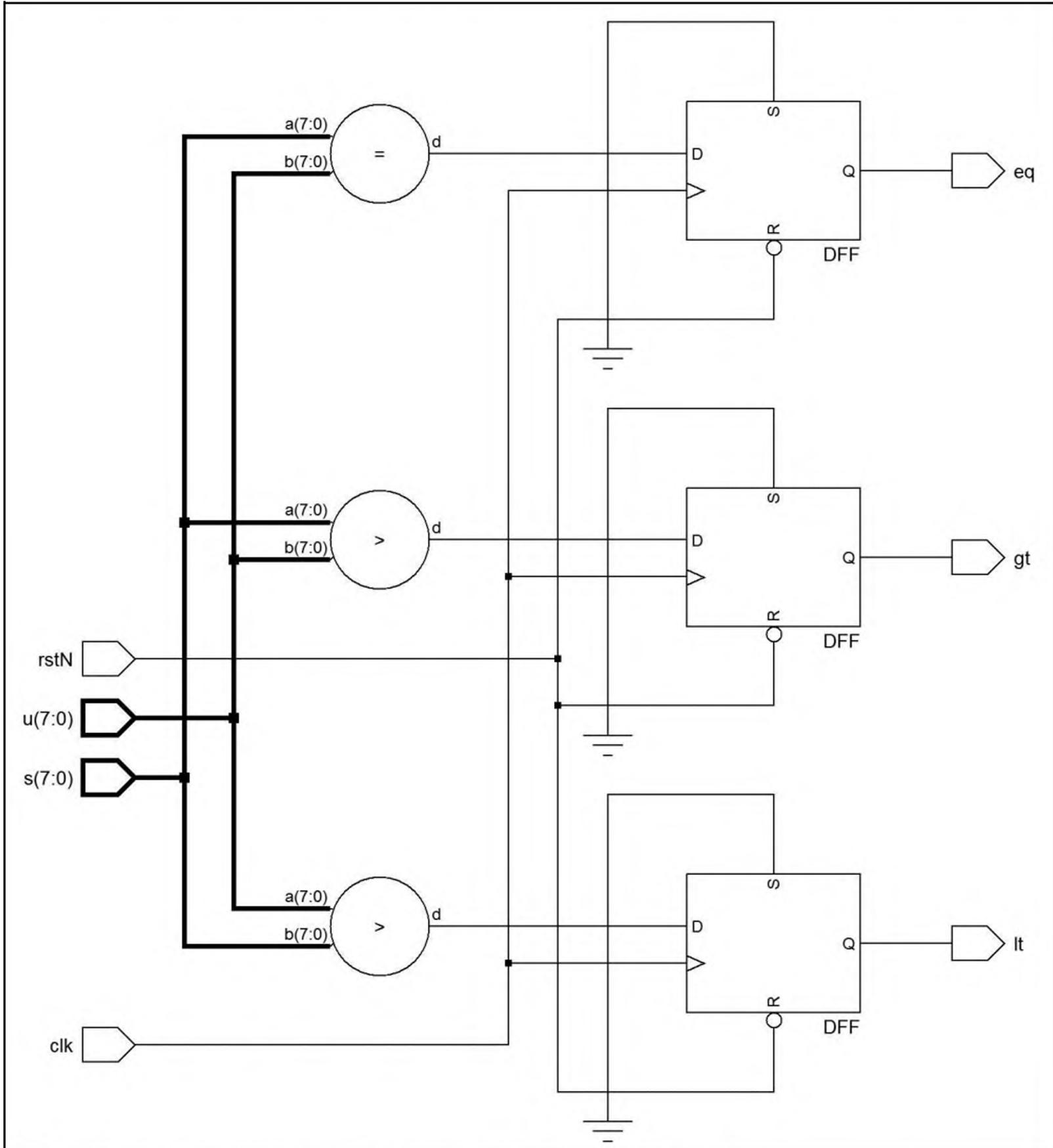


```

    always_ff @(posedge clk or negedge rstN) // async reset
        if (!rstN) {lt,eq,gt} <= '0; // reset flags
        else begin
            lt <= (s < signed'(u)); // less-than operator
            eq <= (s == signed'(u)); // equality operator
            gt <= (s > signed'(u)); // greater-than operator
        end
    endmodule: signed_comparator

```

Figure 5-24: Synthesis result for Example 5-21: Sign casting



Technology independent schematic (no target ASIC or FPGA selected)

The schematic shown Figure 5-24 is based on generic components, before the synthesis compiler has mapped the functionality to a specific target ASIC or FPGA device. Example 5-8 (page 167) earlier in this chapter showed an unsigned version of this same comparator. Comparing Figure 5-8 (page 168) and Figure 5-24 shows that synthesis mapped the unsigned and signed versions to the same generic components. The gate-level implementations are simply comparing the bits that are set in two vectors. It does not actually matter if those vectors are considered signed or unsigned, so long as both vectors are the same signedness. For a signed comparator with operands of mixed signedness, the cast operator ensures that both operands are treated as signed values during the comparison.

5.16 Operator precedence

It is common for an RTL statement to have multiple operators, such as:

```
if (a < b && b < c) ...
```

SystemVerilog defines a precedence order for when multiple operators are used in a statement. When operators differ in precedence, the operators with higher precedence are evaluated first. In the example above, the less-than operator (`<`) has a higher precedence than the logical AND operator (`&&`) operator. Therefore, it is the results of `a < b` and `b < c` that are logically anded.

Table 5-24 lists the operator precedence for SystemVerilog.

Table 5-24: Operator precedence

Operator	Precedence
() [] :: .	highest
+ - ! ~ & ~& ~ ^ ~^ ^~ ++ -- (unary ops)	
**	
* / %	
+ - (binary operators)	
<< >> <<< >>>	
< <= > >= inside dist	
== != === !== ==? !=?	
& (binary operator)	
^ ~^ ^~ (binary operators)	
(binary operator)	
&&	
? : (conditional operator)	
-> <->	
= += -= *= /= %= &= ^= = <=>= <<=>>= := :/ <= (assignment operators)	
{ } { { } }	lowest



Operators on the same row have the same precedence. With three exceptions, multiple operators that have the same precedence are evaluated from left to right (referred to as operator associativity).

In the following example, `a` is first added to `b`, and then `c` is subtracted from the result of `a + b`.

```
assign sum = a + b - c;
```

The three exceptions to a left-to-right associativity are the conditional (`? :`), implication (`->`), and equivalence (`<->`) operators. These operators are evaluated from right to left.

The evaluation order of operations can be explicitly controlled using parentheses. In the following statement, the divide operator has a higher precedence than the add operator, so the normal evaluation order would be to evaluate the `b ** 2` power operation first, and then add that result to `a`.

```
assign out = a + b**2;
```

This implicit evaluation order based on operator precedence and associativity can be changed by using parentheses. In this next snippet, `(a + b)` will be evaluated first, and that result will be raised to the power of 2.

```
assign out = (a + b)**2;
```

5.17 Summary

SystemVerilog offers an extensive set of programming operators — more than most other programming languages. This chapter has examined the rich set of SystemVerilog operators, with a focus on operators that are synthesizable and can be used in RTL modeling. The simulation behavior has been discussed, with guidelines and cautions for using these operators to model hardware designs. Many small RTL code examples have illustrated how these operators can be used in a proper manner, and, at a generic level, the gate-level implementation that might result from synthesizing these examples.

This chapter has also elaborated on SystemVerilog's loosely typed value conversion rules for when an operation involves different data types. SystemVerilog ensures that the operands of operations are converted to a common type and vector size before performing operations. These implicit conversions occur automatically. The SystemVerilog RTL guidelines and implicit conversions discussed in this chapter will generally ensure that the RTL code will synthesize into a proper gate-level implementation. This is because SystemVerilog is a Hardware Description Language, and not a software programming language. However, the implicit conversions are not always obvious, and occasionally an engineer might want to do something different than the implicit conversions. This chapter has shown how the cast operator can be used to both document conversions and cause specific type, size or signedness conversions to explicitly happen.

Chapter 6

RTL Programming Statements

Abstract — Programming statements, such as if-else decisions and for-loops are used to model hardware behavior and at abstract level, without the complexity and details of logic gates, propagation delays, setup times, and connectivity. This chapter discusses the SystemVerilog programming statements that are appropriate for RTL modeling. Important best coding practices for simulation and synthesis are emphasized. The topics presented in this chapter include:

- General purpose **always** procedural block and sensitivity lists
- Specialized **always_ff**, **always_comb** and **always_latch** procedural blocks
- Procedural **begin...end** statement groups
- Decision statements
- Looping statements
- Jump statements
- No-op statement
- Tasks and functions

6.1 SystemVerilog procedural blocks

A *procedural block* is a container for programming statements. The primary purpose of a procedural block is to control when programming statements should execute, such as whenever a rising edge of a clock occurs or whenever a signal or bus changes value. SystemVerilog has two primary types of procedural blocks: *initial procedures* and *always procedures*.

Initial procedures are a verification construct, and are not supported by synthesis compilers. An exception is that synthesis compilers support initial procedures for loading memory blocks using the **\$readmemb** or **\$readmemh** system tasks or assigning to specific memory addresses. FPGA synthesizers might also allow using initial procedures to model device power-up states. Initial procedures are not discussed or used in this book, since they are not used for modeling RTL functionality.

Always procedures are infinite loops. They execute their programming statements, and, upon completion, automatically start over again. The general concept is that when power is on, hardware is always doing something. This continuous behavior is modeled using always procedures.

SystemVerilog has four types of always procedures: a general purpose procedure using the keyword **always**, and specialized always procedures that use the keywords **always_ff**, **always_comb** and **always_latch**.

General purpose always procedures. The **always** procedural block can be used to model many types of functionality, including synthesizable RTL models, abstract behavioral models such as RAMs that will not be synthesized, and verification code such as clock oscillators or continuous response checkers. While the flexibility of the general purpose **always** procedure makes it useful in a wide variety of modeling and verification projects, that same flexibility means that software tools do not know when the intended usage of **always** is for synthesizable RTL models. Synthesis places a number of coding restrictions on general purpose **always** procedures in order to accurately translate the RTL model into ASIC or FPGA devices.

Specialized RTL always procedures. The **always_ff**, **always_comb** and **always_latch** specialized always procedural blocks behave the same as the general purpose **always** procedural block, but impose important coding restrictions required by synthesis. These additional restrictions help to ensure that the behavior of RTL simulations will match the gate-level behavior of the actual ASIC or FPGA. As the names of these specialized procedures suggest, **always_ff** imposes certain synthesis restrictions required for modeling sequential logic devices such as flip-flops. The **always_comb** procedure imposes certain synthesis restrictions for modeling combinational logic such as decoders, and **always_latch** imposes certain synthesis restrictions for modeling latched behavior. Chapter 7 discusses synthesizable modeling styles for combinational and latched logic, Chapter 8 discusses modeling sequential logic, and Chapter 9 discusses modeling latched logic. These chapters contain several examples of how to properly use the specialized always procedures.

6.1.1 Sensitivity lists

An always procedure tells simulation that the functionality being modeled should “always” be evaluated (an infinite loop), but both simulation and synthesis need to know more information to accurately model hardware behavior. These tools also need to know *when* to execute the statements in the procedural block. For RTL modeling, the *when* is either on a clock edge, which represents sequential logic, or on any of the signals used by that procedure change value, which represents combinational or latched logic.

To control when programming statements are executed in synthesizable RTL models, always procedures begin with a *sensitivity list*, which is a list of signals for which a value change will trigger the execution of the procedure. The general purpose **always** and the RTL-specific **always_ff** procedures require that the sensitivity list

be explicitly specified by the RTL design engineer. The RTL-specific **always_comb** and **always_latch** procedures infer an implicit sensitivity list.

An explicitly specified sensitivity list is introduced with the @ token, verbally spoken as “at”. In synthesizable RTL modeling, the sensitivity list contains a list of one or more net or variable names. The names can be separated with either a comma (,) or the keyword **or**.

The following two explicit sensitivity lists are functionally identical:

```
always @(a, b, c) ...
```

```
always @(a or b or, c) ...
```

In the context of a sensitivity list, the **or** keyword is simply a separator. It is not an OR operation. The use of a comma versus the keyword **or** is a matter of user-preference. There is no functional advantage of one style over the other.

A sensitivity list can also specify a specific edge of a scalar (1-bit) signal that will trigger the always procedure. The edge is specified with the keywords **posedge** and **negedge**. Edge sensitivity is important for clock-based functionality.

```
always @(posedge clk or negedge rstN) ...
```

The **posedge** keyword is short for “positive edge”, and **negedge** for “negative edge”. A positive edge is any transition that might be sensed by silicon transistors as a positive going transition. Thus, **posedge** will trigger on a 0-to-1, 0-to-Z, 0-to-X, Z-to-1, X-to-1, Z-to-X and X-to-Z transition. Conversely, **negedge** will trigger on a 1-to-0, 1-to-Z, 1-to-X, Z-to-0, X-to-0, Z-to-X and X-to-Z transition.

Sequential logic sensitivity. Sequential logic components, such as flip-flops, trigger on a clock edge, most often the rising edge of that clock. (Some ASIC and FPGA devices have components that trigger on a falling edge of clock, and, rarely, ones that trigger on both edges of a clock.) To indicate that an always procedure represents a clock-triggered sequential logic behavior, the **always** or **always_ff** keyword is followed by @(**posedge** <clock_name>) or @(**negedge** <clock_name>). For example:

```
always_ff @(posedge clk)
    q <= d;                                // sequential logic flip-flop
```

Some sequential components have asynchronous inputs, such as set or reset controls. These asynchronous signals also affect when simulation or synthesis should evaluate the always procedure, and are, therefore, included in the sensitivity list.

```
always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN) q <= '0;                  // active-low reset
    else           q <= d;
```

Chapter 8 discusses modeling sequential logic in more detail, including synchronous and asynchronous resets, enable controls, and guidelines on the proper usage of the general purpose **always** and the specialized **always_ff** procedural blocks.

Combinational logic sensitivity. The outputs of combinational logic, such as an adder or decoder, reflect a combination of the current input values to that block of logic. Therefore, the programming statements in combinational logic need to be re-evaluated whenever (that's the sensitivity list *when*) any input to that logic changes value. To model this behavior, the **always** keyword is followed by an explicit sensitivity list that includes all signals that are read by that block of logic, in the form of `@(<signal_name>, <signal_name>, ...)`. For example:

```
always @(a, b)
    sum = a + b;
```

One of the features of the **always_comb** specialized always procedure is that it automatically infers a proper combinational logic sensitivity list. The adder code above is modeled using **always_comb** as:

```
always_comb
    sum = a + b;
```

Chapter 7 discusses modeling combinational logic in more detail, along with proper usage guidelines for **always** and **always_comb** procedural blocks.

Latched logic sensitivity. Latches are a form of combinational logic blocks that can store their current state. Modeling latched behavior follows the same sensitivity list rules as modeling combinational logic behavior. The general purpose **always** keyword is followed by a sensitivity list that includes all signals that are read by that block of logic, in the form of `@(<signal_name>, <signal_name>, ...)`, as in:

```
always @(enable, data)
    if (enable) out <= data;
```

The **always_latch** specialized always procedure automatically infers a proper combinational logic sensitivity list.

```
always_latch
    if (enable) out <= data;
```

Chapter 9 discusses modeling latched logic in more detail, including best-practice coding guidelines for using the **always** and **always_latch** procedural blocks.

Non-synthesizable sensitivity lists. Syntactically, a sensitivity list can contain operations, such as `@(a + b)`, or an **iff** guard condition. The **posedge** and **negedge** qualifiers can also be used with vectors that are greater than 1-bit wide, but only the least-significant bit (the right-most bit) of the vector is used. Changes on other bits in the vector will not trigger the sensitivity list. Operations, **iff**, and edges of vectors are not generally supported by RTL synthesis compilers.

6.1.2 Begin-end statement groups

All forms of procedural blocks can contain either a single statement or a single group of statements. A statement group is contained between the keywords **begin** and **end**, and can contain any number of statements, including none. The following code snippets show an always procedure with a single statement and an always procedure with a begin-end group.

```
always_ff @(posedge clk or negedge rstN)
  if (!rstN)          // if-else is the single statement
    q <= '0;
  else
    q <= d;

always_comb
  begin              // begin-end is the single group
    sum = a + b;
    dif = a - b;
  end
```

A statement can be nested within another statement, as in:

```
always @(posedge clk)
  if (enable)           // single outer statement
    for (int i; i<=15; i++) // nested statement
      out[i] = a[i] ^ b[(N-1)-i]; // another nested stmt
```

In the preceding code snippet, the outer statement is the single statement in the always procedure, and therefore does not require a **begin...end** group.

A begin-end group can be named, using the syntax:

begin: <name>

A named statement group can contain local variable declarations. Local variables can be used within the statement group, but cannot be referenced outside of the group in synthesizable RTL models. (A later version of SystemVerilog added the ability to declare local variables in unnamed begin-end groups, but this was not supported by most synthesis compilers at the time this book was written.)

Optionally, the matching **end** of the group can also be named. Naming the end of a statement group can help visually match up nested statement groups. SystemVerilog requires that the names used for the **begin** and the **end** must match exactly.

The use of local variables help ensure proper synthesis results in certain contexts. A temporary intermediate variable calculated in a sequential always procedure and used by another procedure might appear to work in simulation, but can synthesize into gate-level functionality that might not match the RTL simulation behavior. Declaring a local variable within a procedure will prevent this coding error — a local variable cannot be accessed from outside of the procedure.

The following example declares a temporary variable local to an **always_ff** sequential logic procedure. The temporary variable is used to calculate an intermediate result, which is then used to calculate a final result. (The calculations in this example were purposely kept simple in order to focus on the declaration of local variables, rather than some complex algorithm that might require intermediate calculations.)

```
always_ff @(posedge clk) begin : two_steps
    logic [7:0] tmp; // local temporary variable
    tmp = a + b;
    out <= c - tmp;
end:two_steps
```

Observe that white space is permitted before and after the colon, as shown with the named **begin** above. White space is not required, however, as shown with the named **end** above. The use of white space can help to make complex code easier to read.

6.1.3 Using variables and nets in procedural blocks

The left-hand side of procedural assignments can only be variable types, including user-defined types based on variables. Variables retain their previous value until updated by an operator or assignment statement. This characteristic of variables can affect both simulation and synthesis.

In the following code snippet, `sum` must be declared as a variable type because it is on the left-hand side of a procedural assignment. See Chapter 3, section 3.4 (page 67) for a discussion on synthesizable variable types that can be used in RTL modeling.

```
wire [15:0] a, b; // net types
logic [15:0] sum; // variable types

always_comb
    sum = a + b; // sum must be a variable type
```

It is only the left-hand side of a procedural assignment that must be a variable. The right-hand side of assignments can use variables, nets, parameters or literal values.

6.2 Decision statements

Decision statements allow the execution flow of a procedural block to branch to specific statements, based on the current values of signals in a design. SystemVerilog has two primary decision statements: **if...else** statements and case statements, which use the keywords **case**, **case...inside**, **casex** and **casez**.

6.2.1 if-else statements

An *if-else* statement evaluates an expression and executes one of two possible branches, a true branch or a false branch.

```
always_comb
  if (!select) y = a;      // true branch
  else          y = b;      // false branch
```

The if-else expression can be a net or variable of any vector size, or the return of an operation. A vector expression evaluates as true if one or more bits of the expression are set to a 1. The expression evaluates as false if all bits of the expression are 0. For example:

```
logic [7:0] a, b, y;
always_comb
  if (a & b) y = a & b;      // true branch
  else          y = a ^ b;      // false branch
```

The result of the bitwise AND of a with b is an 8-bit vector (because both a and b are 8-bit vectors). If the AND operation results in any bit being set, then the true branch will be executed. If the result of the logical AND is zero, then the false branch will be executed.

Best Practice Guideline 6-1

Only use 1-bit values or the return of a true/false operation for an if-else condition expression. Do not use vectors as an if-else expression.

Operators that return a true/false result are listed in Chapter 5, sections 5.6, 5.7, 5.8 and 5.9.

Do not perform true/false tests on vectors. Evaluating vectors as true or false could lead to design errors. In the preceding example, did the engineer writing the code intend to test `(a & b)`, which is an 8-bit vector value, or `(a && b)`, which is a 1-bit result of a true/false logical operation? Which branch the if-else decision executes can be different for some values of a and b. This ambiguity and possible coding bug will be avoided by following the guideline to only use scalar (1-bit) values, or the return of operations that have a true/false result.

With 4-state values, it is possible that an expression is neither true or false, as in the value `8'b0000000z`. An expression that is neither true nor false is considered to be unknown. The false branch will be executed when the expression of an if-else decision evaluates as unknown. This can cause a mismatch in how RTL models simulate, and in how post-synthesis gate-level models actually behave. This circumstance is discussed in Appendix C on X-optimism and X-pessimism in SystemVerilog models.

Each branch of an if-else decision can be a single statement or a group of statements enclosed between `begin` and `end`, as shown in the following code snippet.

```

always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN) begin // reset branch
        lt <= '0;
        eq <= '0;
        gt <= '0;
    end
    else begin // clocked branch
        lt <= (a < b);
        eq <= (a == b);
        gt <= (a > b);
    end

```

(The full code and synthesis results for this code is in Example 5-8, page 167).

If statements without an else branch. The **else** (false) branch of an if-else decision is optional. If there is no **else** branch, and the expression evaluates as false (or unknown), then no statement is executed. In the following code snippet, if `enable` is 0, then `out` is not changed. Since `out` is a variable (see section 6.1.3, page 216), it retains its previous value, modeling the storage behavior of a latch.

```

always_latch
    if (enable) out <= data;

```

Chained if-else-if decisions. A series of decisions can be formed by a chain of if-else decisions, as in the following code snippet.

```

always_comb begin
    if (opcode == 2'b00) y = a + b;
    else if (opcode == 2'b01) y = a - b;
    else if (opcode == 2'b10) y = a * b;
    else y = a / b;
end

```

Observe that SystemVerilog does not have an **elsif** keyword like some programming languages. A chain of decisions is formed by each **else** branch containing a nested if-else decision. This nesting is more obvious when the code snippet above is coded with different indentation, as shown below.

```

always_comb begin
    if (opcode == 2'b00) y = a + b;
    else
        if (opcode == 2'b01) y = a - b;
        else
            if (opcode == 2'b10) y = a * b;
            else y = a / b;
end

```

A series of if-else-if decisions is evaluated in the order in which the statements are listed. This gives priority to the decisions listed first. The following example illustrates a flip-flop that can be set or reset. If both the set and reset are active at the same

time, the reset has priority because it is evaluated first in the series of decisions. The set and reset controls in this example are active-low signals.

```
always_ff @(posedge clk or negedge rstN or negedge setN)
  if      (!rstN) q <= '0; // reset register
  else if (!setN) q <= '1; // set register
  else      q <= d; // clock the register
```

(This set-reset flip-flop example has a potential simulation glitch, which is discussed in Chapter 8, section 8.1.5.4, page 290.)

Synthesizing if-else decisions. The way synthesis compilers implement an if-else decision depends on the context of the decision statement and the types of components available in the target ASIC or FPGA. Some general rules are:

- An **if-else** statement in combinational logic behaves as a multiplexor, and will often be realized as a mux in the gate-level implementation.
- An **if** without an **else** in combinational logic will behave as a latch if no other statement assigns to the same variable. This is because the variable being assigned retains its previous value. Synthesis will generally implement this storage effect as a latch.
- An **if-else-if** series of statements in combinational logic simulates with priority-encoded behavior, where each **if** statement takes precedence over any subsequent **if** statements in the series. Synthesis compilers will remove the priority encoding if all of the decision expressions are mutually exclusive (two or more expressions can never be true at the same time).
- An **if-else** statement that is evaluated on a clock edge behaves as flip-flop, and will be synthesized as some type of register in the gate-level implementation.

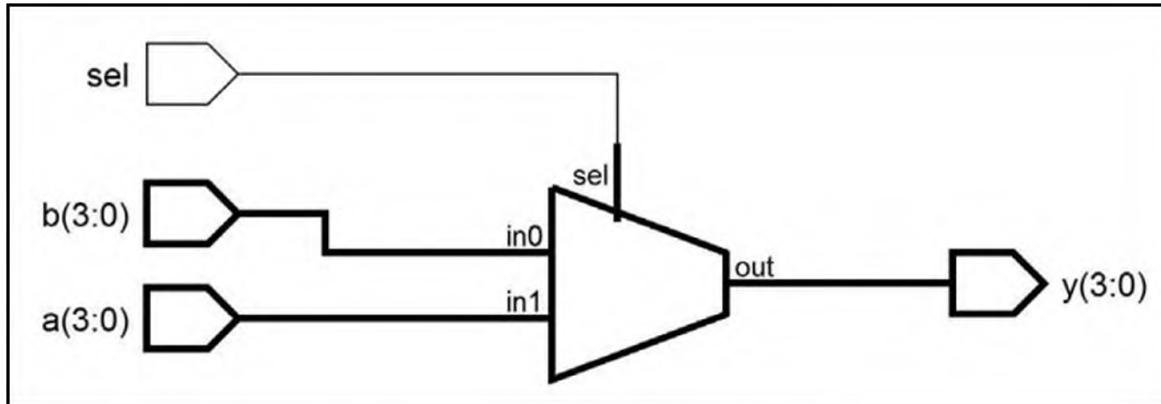
Using if-else as a multiplexor. Example 6-1 and its accompanying synthesis result in Figure 6-1 show if-else being used in the context of a multiplexor.

Example 6-1: Using **if-else** to model multiplexor functionality

```
module mux2to1
#(parameter N = 4)           // data word size
(input logic      sel,      // 1-bit input
 input logic [N-1:0] a, b,   // scalable input size
 output logic [N-1:0] y);   // scalable output size

  always_comb begin
    if (sel) y = a;
    else   y = b;
  end
endmodule: mux2to1
```

Figure 6-1: Synthesis result for Example 6-1: if-else as a MUX



Technology independent schematic (no target ASIC or FPGA selected)

Using if-else as a latch. Example 6-2 shows an **if** statement representing a latch.

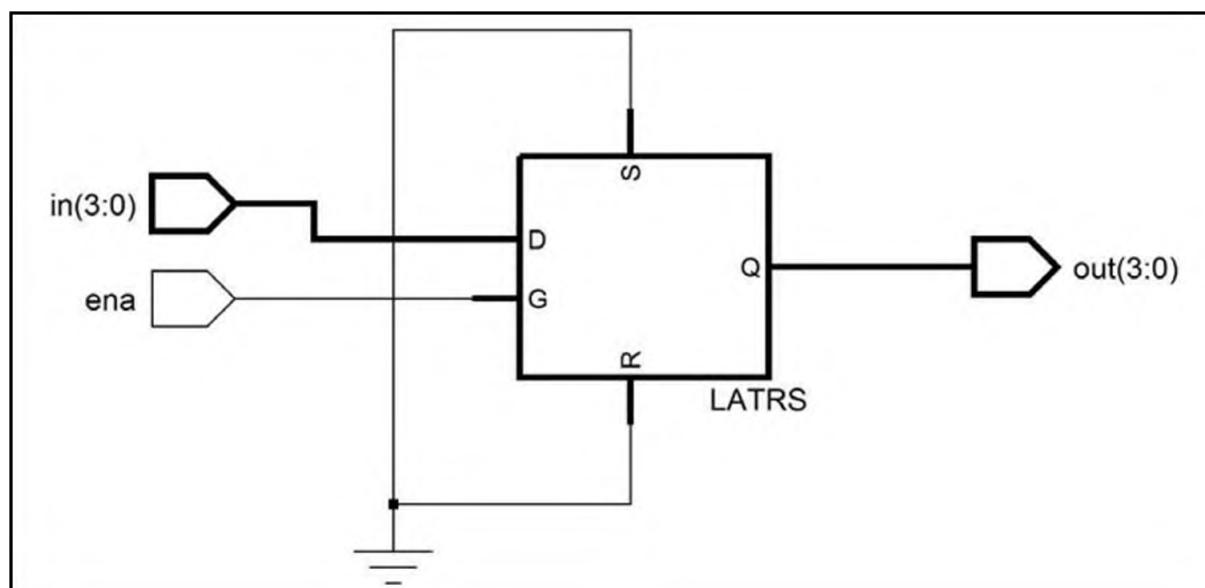
Example 6-2: Using **if** without **else** to model latch functionality

```

module latch
# (parameter N = 4)           // data word size
(input logic ena,          // 1-bit input
 input logic [N-1:0] in,   // scalable input size
 output logic [N-1:0] out // scalable output size
);

always_latch begin
    if (ena) out <= in;
end
endmodule: latch
    
```

Figure 6-2: Synthesis result for Example 6-2: if-else as a latch



Technology independent schematic (no target ASIC or FPGA selected)

The synthesis compiler used to generate Figure 6-2 translated the RTL functionality to a generic latch device that has unused set and reset inputs. The specific type of latch used in the final implementation will depend on the latch types available in the target ASIC or FPGA.

Using if-else as a priority encoder. Example 6-3 illustrates an if-else-if in the context of a 4-to-2 priority encoder. (Example 6-6, page 227, shows a variation of this same design.)

Example 6-3: Using an if-else-if series to model a priority encoder

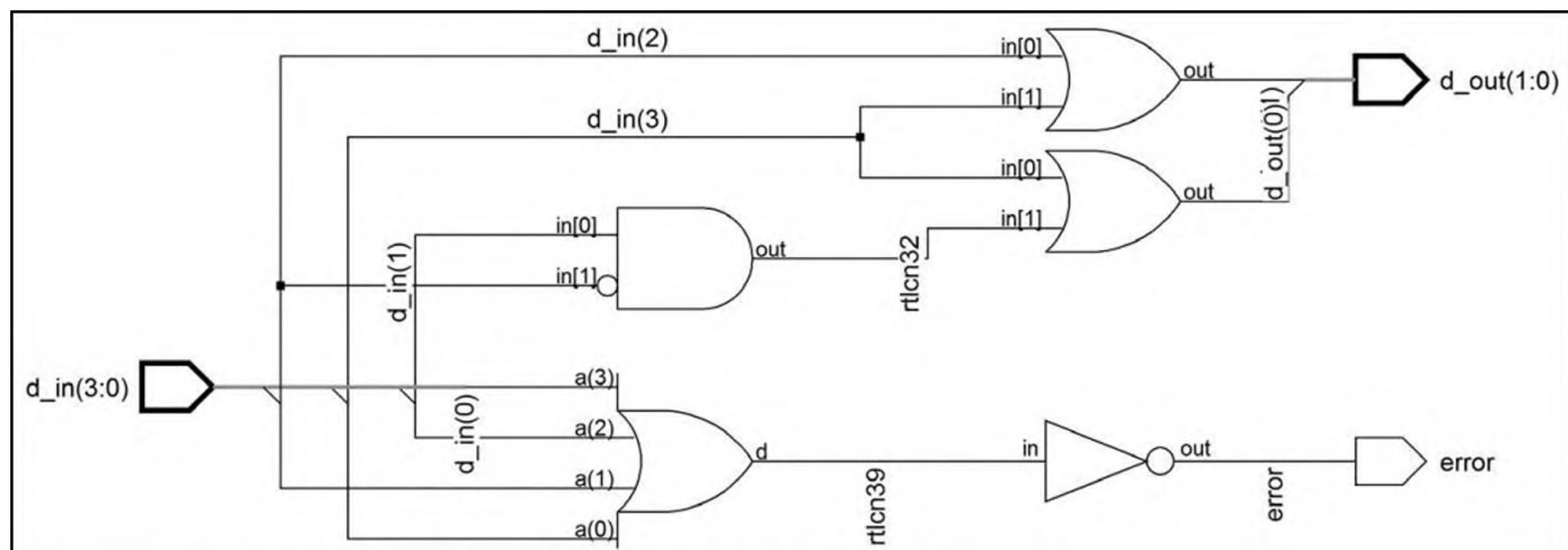
```

module priority_4to2_encoder (
    input logic [3:0] d_in,
    output logic [1:0] d_out,
    output logic error
);

    always_comb begin
        error = '0;
        if (d_in[3]) d_out = 2'h3; // bit 3 is set
        else if (d_in[2]) d_out = 2'h2; // bit 2 is set
        else if (d_in[1]) d_out = 2'h1; // bit 1 is set
        else if (d_in[0]) d_out = 2'h0; // bit 0 is set
        else begin // no bits set
            d_out = 2'b0;
            error = '1;
        end
    end
endmodule: priority_4to2_encoder

```

Figure 6-3: Synthesis result for Example 6-3: if-else as a priority encoder



Technology independent schematic (no target ASIC or FPGA selected)

The priority encoding in Figure 6-3 is implemented as series of logic gates where the output of one stage becomes the input to the next stage in the series, rather than encoding all of the bits of `d_in` in parallel. This serial data path is a result of the priority in which the bits of `d_in` are evaluated in the if-else-if series.

Using if-else as a flip-flop. Example 6-4 shows an if-else-if in the context of a sequential logic flip-flop with reset and chip-enable (also called load-enable or data-enable) inputs. Because the reset input is evaluated first, it has priority over the enable input. Figure 6-4 shows the result of synthesizing this if-else-if decision series.

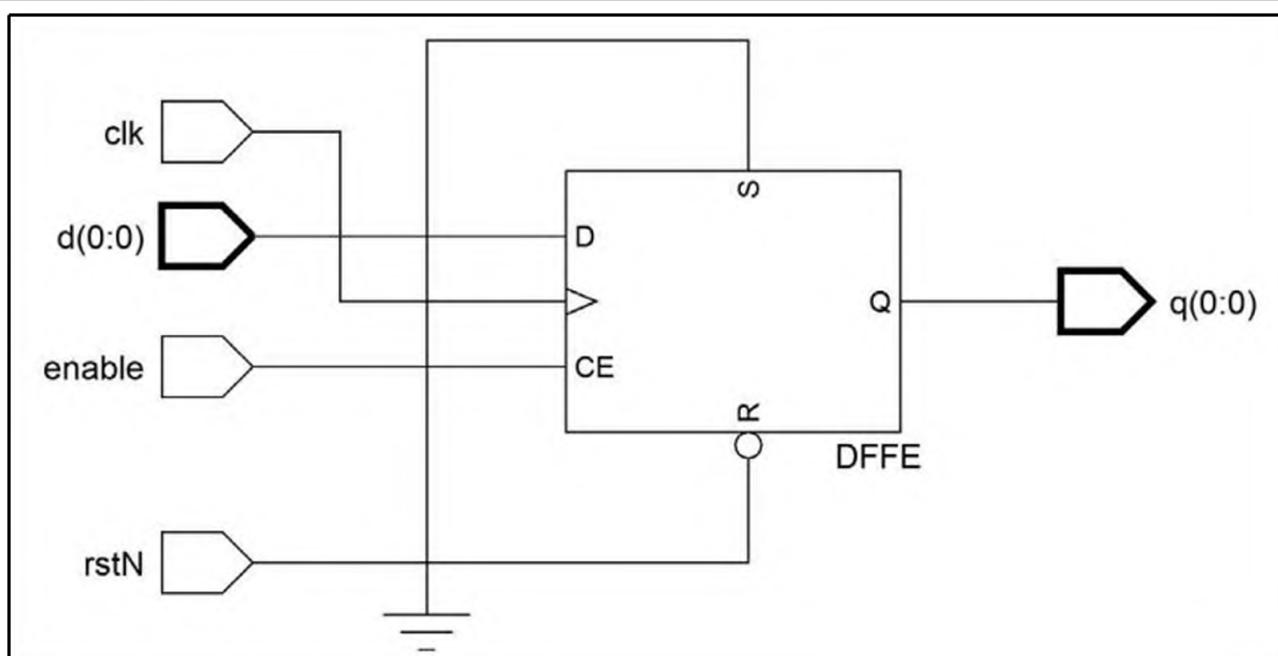
Example 6-4: Using if-else-if series to model a flip-flop with reset and chip-enable

```

module enable_ff
#(parameter N = 1) // bus size
(input logic clk, // posedge triggered clk
 input logic rstN, // active low async reset
 input logic enable, // active high chip enable
 input logic [N-1:0] d, // scalable input size
 output logic [N-1:0] q // scalable output size
);
  always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN) q <= '0; // active-low reset
    else if (enable) q <= d; // store if enabled
endmodule: enable_ff

```

Figure 6-4: Synthesis result for Example 6-4: if-else as a chip-enable flip-flop



Technology independent schematic (no target ASIC or FPGA selected)

Figure 6-4 shows how synthesis has mapped the chip-enable flip-flop with active-low reset to a generic component. The next step in the process is for the synthesis compiler to map this generic component to a specific type of flip-flop available in a target ASIC or FPGA device. If that target device does not have a chip-enable flip-flop, then synthesis will add multiplexor functionality outside of the flip-flop to mimic the chip-enable behavior. The multiplexor will pass the new value of data to the D input if the flip-flop is enable, and will feed the flip-flop Q output back to the D input if the flip-flop is not enabled. In a similar manner, if the target device does not have flip-flops with asynchronous active-low resets, the synthesis compiler will add functionality outside of the flip-flop to mimic this behavior. Modeling and synthesizing flip-flops with various types of resets is discussed in Chapter 8, section 8.1.5 (page 286).

6.2.2 Case statements

A **case** statement provides a concise way to represent a series of decision choices. For example:

```
always_comb begin
    case (opcode)
        2'b00 : result = a & b;
        2'b01 : result = a | b;
        2'b10 : result = a ^ b;
        2'b11 : result = a >> 1;
        default: result = 'X;
    endcase
end
```

A SystemVerilog **case** statement is similar to a C **switch** statement, but there are important differences. SystemVerilog does not use a **break** statement the way C uses **break** to exit from each branch of a **switch** statement. A case statement exits automatically after a branch is executed, without needing to execute a **break**. It is illegal to use a SystemVerilog **break** statement to exit a case statement.

SystemVerilog has 4 variations of case statements, with the keywords **case**, **case...inside**, **casex** and **casez**. The general syntax and usage of these various case statements is the same. The differences are described later in this section.

The **case**, **casex** or **casez** keyword is followed by a *case expression* enclosed in parentheses. The case expression can be a net, variable, user-defined type, parameter constant, literal value, or the result of an operation.

The case expression is compared to one or more *case items*, which can also be a net, variable, user-defined type, parameter constant or literal value. The case item is followed by a colon, and then a single statement, or a begin-end group of statements, to be executed if the case expression matches the case item.

The default case item. An optional *default case item* can be specified by using the **default** keyword. The default will be executed if the case expression did not match any of the case items. In the example above, the case items cover all the possible 2-state values of a 2-bit opcode. If *opcode* is a 4-state type, however, there are additional X and Z values that are not decoded by the case items. If *opcode* should have any bits that are X or Z, the **default** branch will be executed, which, in the preceding example, will propagate an X value onto the *result* variable. The default case item does not need to be the last case item. Syntactically, the default can be the first case item, or anywhere in the middle of the case items. A best-practice coding style for code readability is to make the default case item the last case item.

Comma-separated lists of case item values. A case item can be a comma-separated list of values, as in the following snippet:

```
always_comb begin
    case (opcode)
        2'b00, 2'b01: result = a & b;
        2'b10, 2'b11: result = a | b;
    endcase
end
```

The first branch of the case statement will be executed if `opcode` has a value of either `2'b00` or `2'b01`, and the second branch if the value is `2'b10` or `2'b11`.

6.2.2.1 Case versus case...inside

When just the `case` keyword, the case expression is compared to the case items using the behavior of the *== case equality operator* (see Chapter 5, section 5.8, page 168). The `==` operator compares each bit of the expressions for an exact match of 4-state values. In the following code snippet, the third branch will be executed if `select` has a value of `1'bz`, and the fourth branch if `select` is `1'bx`. (This example is not synthesizable; Synthesis does not allow comparing for X and Z values.)

```
always_comb begin
    case (select)
        1'b0: y = a;
        1'b1: y = b;
        1'bz: y = 'z;
        1'bx: y = 'x;
    endcase
end
```

With the `case...inside` case statement, the case expression is compared to the case items using the behavior of the *==? wildcard case equality operator* (see Chapter 5, section 5.8, page 168). The `==?` operator allows bits to be masked from the comparison. Any bit in a case item that is set to `x` or `z` or `?` is masked, and that bit position is ignored when the case expression is compared to the case item.

In the following example, the first branch will be executed if the most significant bit of `selector` is set. All the remaining bits of `selector` are ignored. The second branch will be taken if the upper two bits of `selector` have the value 01, and the remaining bits are ignored, and so forth.

```
always_comb begin
    case (selector) inside
        4'b1???: out = a; // MSB is set
        4'b01???: out = b;
        4'b001?: out = c;
        4'b0001: out = d;
    default: out = '0; // no bits are set
    endcase
end
```

6.2.2.2 The obsolete casex and casez statements

The original Verilog language, before SystemVerilog extended the language in 2005, used the **casex** and **casez** keywords to mask bits from a comparison. SystemVerilog replaced **casex** and **casez** with the **case...inside** keyword pair. The **casex** case statement masks out any bits set to **x** or **z** or **?**. The **casez** case statement only masks out bits set to **z** or **?**.

Best Practice Guideline 6-2

Use the **case...inside** decision statement to ignore specific bits in case items. Do not use the obsolete **casex** and **casez** decision statements.

The reason SystemVerilog replaced **casex** and **casez** is because they have a serious flaw in their simulation rules that can synthesize into logic gates that behave very differently than the RTL simulation. In brief, **casex** and **casez** not only allow bits to be masked in the case items, but also allow masking bits in the case expression. This double masking can lead to a branch being executed that was not intended, and that might not be the same branch that the gate-level implementation created by synthesis would take. The hazards of **casex** and **casez** are not discussed in this book because there is no need to ever use these constructs — the **case...inside** statement makes these older construct obsolete.

6.2.2.3 Case item priority and automatic synthesis optimization

Case items are evaluated in the order in which they are listed. Therefore, each case item has priority over all subsequent case items. Simulation will always observe this priority when a case statement is evaluated.

This inferred priority encoding is often not desirable in an ASIC or FPGA implementation. Priority-encoded logic requires more logic gates and longer propagation paths than a parallel evaluation. Synthesis compilers will analyze the case item values before translating case statements into logic gates. If there is no possibility that two case items can be true at the same time, synthesis compilers will automatically optimize the gate-level implementation to evaluate the case items in parallel, instead of as priority-encoded functionality.

If, however, it is possible for two or more case items to be true at the same time, synthesis will implement priority-encoded logic that is inherent in how case statements simulate. By implementing the priority encoding, synthesis ensures that the gate-level behavior of the ASIC or FPGA matches the RTL simulation behavior.

Example 6-5 illustrates a 4-to-1 multiplexor. In this example, the four case expressions have unique, non-overlapping values. Synthesis will recognize that there is no possibility of two case expressions being true at the same time, and automatically remove the priority encoded evaluation of the case items. It is not needed. Figure 6-5 shows how synthesis implements the case statement in gates.

Example 6-5: Using a **case** statement to model a 4-to-1 MUX

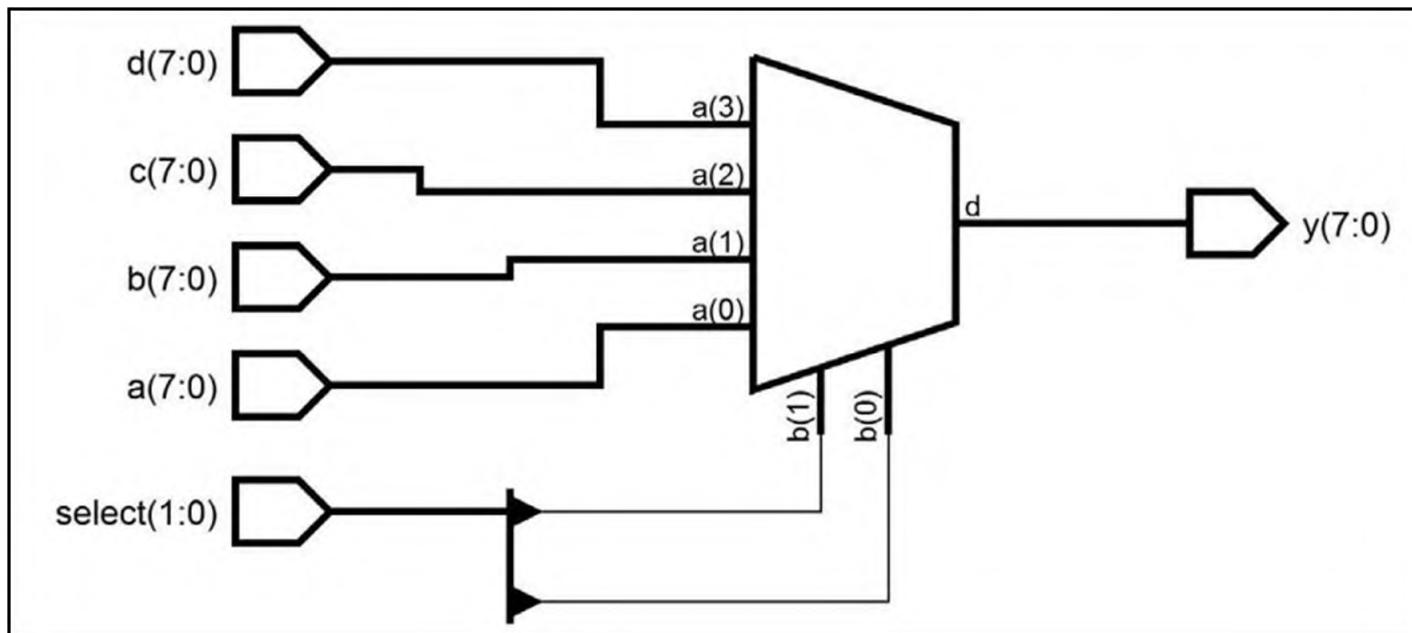
```

module mux4to1
# (parameter N=8)
(
    input logic [N-1:0] a, b, c, d,
    input logic [ 1:0] select,
    output logic [N-1:0] y
);

always_comb begin
    case (select)
        2'b00: y = a;
        2'b01: y = b;
        2'b10: y = c;
        2'b11: y = d;
    endcase
end
endmodule: mux4to1

```

Figure 6-5: Synthesis result for Example 6-5: case statement as a 4-to-1 MUX



Technology independent schematic (no target ASIC or FPGA selected)

The case items in Example 6-5 are mutually exclusive, meaning it is not possible for two of these case items to be true at the same time. Therefore, the synthesis compiler removed the priority encoded behavior of the case statement, and implemented a more gate-efficient parallel evaluation of the case items, in the form of a multiplexor.

The removal of priority logic by synthesis compilers occurs automatically, as long as synthesis can determine that all case items are mutually exclusive (there will never be two or more case items that evaluate as true at the same time). Synthesis compilers will leave in the priority evaluation of case items if it cannot determine that the case items are mutually exclusive.

Example 6-6 is similar to the 4-to-2 priority encoder shown in Example 6-3, but this time uses **case...inside** to allow for checking only specific bits in the 4-bit **d_in** value. Because other bits are ignored, there is a possibility of more than one

case item being true at the same time. Simulation will execute the first matching branch, and synthesis compilers will match that behavior by leaving in the priority encoding that is inherent in case statements.

Example 6-6: Using an case-inside to model a priority encoder

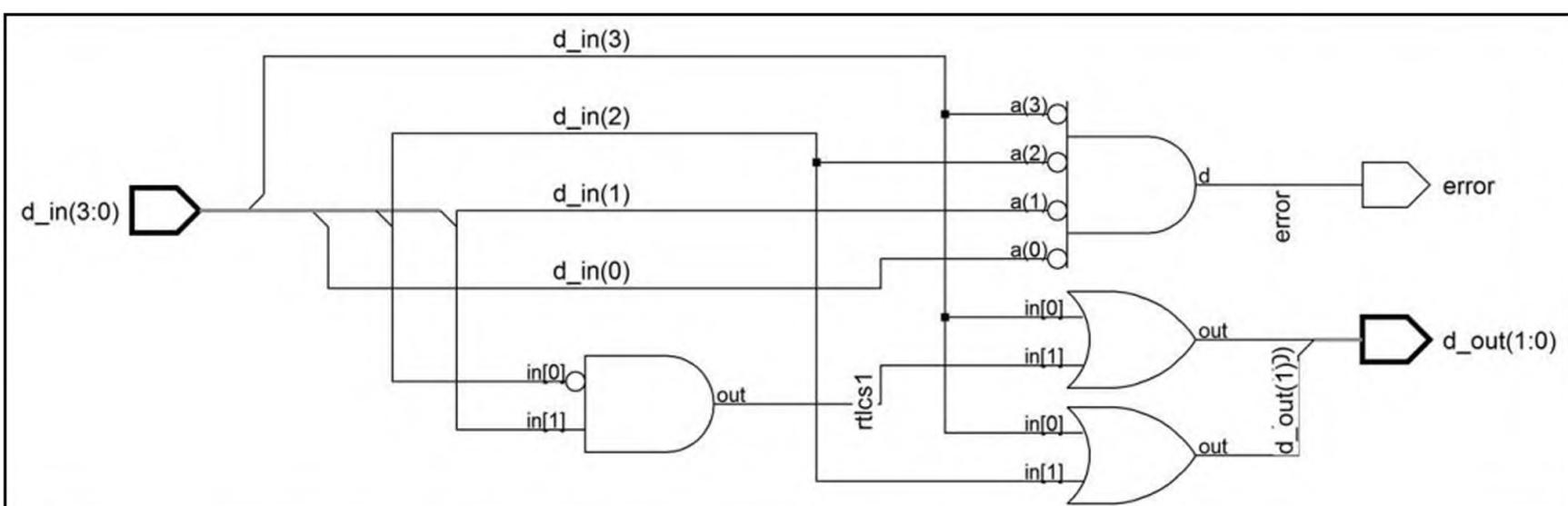
```

module priority_4to2_encoder (
    input logic [3:0] d_in,
    output logic [1:0] d_out,
    output logic error
);

    always_comb begin
        error = '0;
        case (d_in) inside
            4'b1???: d_out = 2'h3; // bit 3 is set
            4'b01???: d_out = 2'h2; // bit 2 is set
            4'b001?: d_out = 2'h1; // bit 1 is set
            4'b0001: d_out = 2'h0; // bit 0 is set
            4'b0000: begin // no bits set
                d_out = 2'b0;
                error = '1;
            end
        endcase
    end
endmodule: priority_4to2_encoder

```

Figure 6-6: Synthesis result for Example 6-6: case...inside as a priority encoder



Technology independent schematic (no target ASIC or FPGA selected)

The effect of the priority logic can be seen in the series of gates through which different bits of `d_in` propagate. The circuitry is very similar to what the synthesis compiler generated for this same design when a series of if-else-if decisions were used, as shown earlier in Example 6-3 and Figure 6-3 (page 221).

6.2.3 Unique and priority decision modifiers

SystemVerilog provides three modifiers to case and if-else decision statements: **unique**, **unique0** and **priority**. These decision modifiers are only mentioned in this chapter. The modifiers are discussed in more detail where they are applied in RTL models in Chapter 7, section 7.4.2 (page 266) and in Chapter 9, section 9.3.5 (page 340).

In brief, the **unique**, **unique0** and **priority** modifiers do two things:

- They affect how synthesis compilers render case statements in the gate-level implementation.
- They turn on warning messages in simulation that help to verify that the effects on synthesis will work as intended.

An example usage of these decision modifiers is:

```
always_comb begin // 3-state FSM output decoder
  unique case (state)
    2'b00: {ok,busy,done} = 3'b100;
    2'b01: {ok,busy,done} = 3'b010;
    2'b10: {ok,busy,done} = 3'b101;
  endcase
end
```

For synthesis, the **unique** modifier in this example informs synthesis compilers that the **case** statement can be considered complete, even though only three of the four possible values of the 2-bit state variable are decoded. It also informs synthesis compilers that a parallel evaluation of the case items is OK.

For simulation, the **unique** keyword enables two run-time checks in simulation. A violation report will be generated if the case statement is evaluated and the value of state does not match any of the case items. This check helps verify that it is safe for synthesis to treat the case statement as complete. A violation report will also be generated if the value of state matches more than one case item at the same time. This check helps verify that it is safe to evaluate the case items in parallel, rather than in the order the case items are listed.

6.3 Looping statements

Looping statements allow the execution of a programming statement or begin-end group of statements to be executed multiple times. The looping statements in SystemVerilog are: **for**, **repeat**, **while**, **do...while**, **foreach** and **forever**. Of these, only the **for** and **repeat** loops are supported by all synthesis compilers. The other types of loops might be supported with restrictions by some synthesis compilers, but the restrictions limit the usefulness of those loops. This book focuses on the **for** and **repeat** loop, which are supported by all synthesis compilers.

6.3.1 For loops

The general syntax of a **for** loop is:

```
for (initial_assignment ; end_expression ; step_assignment )
    statement_or_statement_group
```

- The *initial_assignment* is only executed once, when the loop starts.
- The *end_expression* is evaluated before the first pass of the loop. If the expression is true, the *statement_or_statement_group* is executed. If the expression is false, the loop exits.
- The *step_assignment* is executed at the end of each pass of the loop. The *end_expression* is evaluated again. If it is true, the loop is repeated, otherwise the loop exits.

The following code snippet illustrates a simple example of using a **for** loop. This example exclusive-ORs each bit of *a_bus* with the reverse bit position in *b_bus*. For a 4-bit bus, *a_bus[0]* is exclusive-ORed with *b_bus[3]*, *a_bus[1]* with *b_bus[2]*, and so forth.

```
parameter N = 4;
logic [N-1:0] a, b, y;

always_comb begin
    for (int i=0; i<=N-1; i++)
        y[i] = a[i] ^ b[(N-1)-i]; // XOR a and reverse order of b
    end
```

Synthesis compilers implement loops by first “unrolling” the loop, meaning the statement or begin-end statement group in the loop is replicated the number of times that the loop iterates. In the code snippet above, the assignment statement is replicated four times, because *i* will iterate from 0 to 3. The code that synthesis sees after it unrolls the loop is:

```
always_comb begin
    y[0] = a[0] ^ b[3-0];
    y[1] = a[1] ^ b[3-1];
    y[2] = a[2] ^ b[3-2];
    y[3] = a[3] ^ b[3-3];
end
```

The number of iterations a loop will execute must be a fixed number times in order for synthesis to unroll the loop. Loops with a fixed number of iterations are referred to as static loops, and are discussed in more detail in section 6.3.1.1 (page 230).

The advantage of loops becomes apparent when there are larger number of iterations. If *a* and *b* had been 64-bit busses in the **for** loop snippet above, it would have required 64 lines of code to manually exclusive-or the two 64-bit busses. With a **for** loop, only two lines of code are needed regardless of the vector size of the busses.

Example 6-7 shows a complete parameterized model of the code snippet above. Figure 6-7 shows the results of synthesizing this model.

Example 6-7: Using a **for** loop to operate on bits of vectors

```

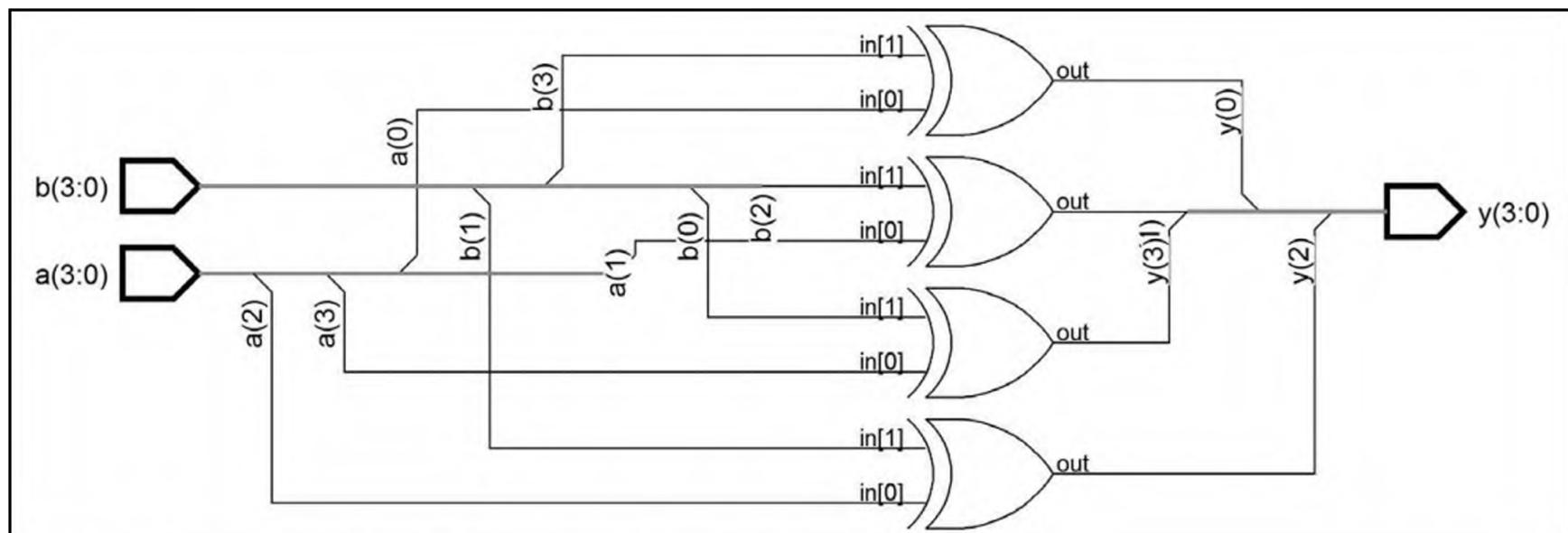
module bus_xor
#(parameter N = 4)           // bus size
(input logic [N-1:0] a, b,    // scalable input size
 output logic [N-1:0] y       // scalable output size
);

always_comb begin
  for (int i=0; i<N; i++) begin
    y[i] = a[i] ^ b[(N-1)-i]; // XOR a and reverse order of b
  end
end

endmodule: bus_xor

```

Figure 6-7: Synthesis result for Example 6-7: for-loop to operate on vector bits



Technology independent schematic (no target ASIC or FPGA selected)

It can be seen in Figure 6-7 how the four iterations of the **for** loop were unrolled to become four instances of the exclusive-or operation.

6.3.1.1 Static loops versus data-dependent loops

A *static loop*, also called a *data-independent loop*, is a loop where the number of iterations can be determined without having to know the values of any nets or variables. The loop **for** (**int** i=0; i<=3; i++) is a static loop. It can be determined that the loop will iterate 4 times (i=0 to i<=3). There is no dependency on an other signals to determine how many times the loop will iterate.

A *data-dependent loop* is a non-static loop that requires evaluating the value of a net or variable to determine how many times the loop will execute. The loop **for** (**int** i=0; i<=count; i++) is data-dependent because it cannot be determined how many times the loop will iterate without knowing the value of count.

6.3.1.2 Zero-delay and timed loops

A zero-delay loop does not contain any form of timing. A zero-delay loop represents combinational logic. In simulation, a zero-delay loop executes instantaneously. In the gate-level implementation generated by synthesis, a zero-delay loop executes within a single clock cycle. The **for** loop shown in the preceding Example 6-7 (page 230) is a zero-delay static loop.

A timed loop consumes time to execute each pass of the loop. Timed loops do not represent the behavior of combinational logic because the execution of the loop might take more than a clock cycle to complete.

Best Practice Guideline 6-3

Code **for** loops as static, zero-delay loops with a fixed number of iterations.

In order to unroll a loop, synthesis compilers need to be able to statically determine how many times the loop will iterate. It is possible — and all too easy — to code a **for** loop that will simulate, but that is not synthesizable. An example of this is:

```
always_comb begin
    // find lowest bit that is set in a 32-bit vector
    low_bit = '0;
    end_count = 32;
    for (int i=0; i<end_count; i++) begin
        if (data[i]) begin
            low_bit = i;
            end_count = i; // cause loop to terminate early
        end
    end
end
```

The intent of the code snippet is to iterate through the `data` vector to find the lowest numbered bit that is set. The loop starts with the least-significant bit, bit 0, of `data` and iterates upward until a bit in `data` is set to 1. The loop terminates as soon as the first set bit is found by modifying the value of `end_count`, which is the loop end condition. Although `end_count` is initialized to 32 before the loop starts, its value can change as the loop is executing.

The problem synthesis compilers have with this code snippet is that it is impossible to statically determine how many times the loop will iterate because the end condition of the loop can change, based on the simulation value of `data`. In order to unroll a loop, synthesis requires that the loop executes a fixed number of times.

Synthesizable way to exit a loop without data dependence. Example 6-8 shows a coding style for the preceding snippet that is synthesizable. Instead of depending on the value of `data` to determine the end of the loop, Example 6-8 uses a static loop that executes a fixed number of times. Rather than terminating the loop early when the

lowest set bit is found, the loop simply does nothing for the remaining iterations, after finding the lowest bit that is set. Figure 6-8 shows the results from synthesizing this example. The bus size of data is parameterized in this example, and set to only 4-bits wide in order to reduce the size of the schematic to fit the page size of this book.

Example 6-8: Using a **for** loop to find the lowest bit that is set in a vector

```

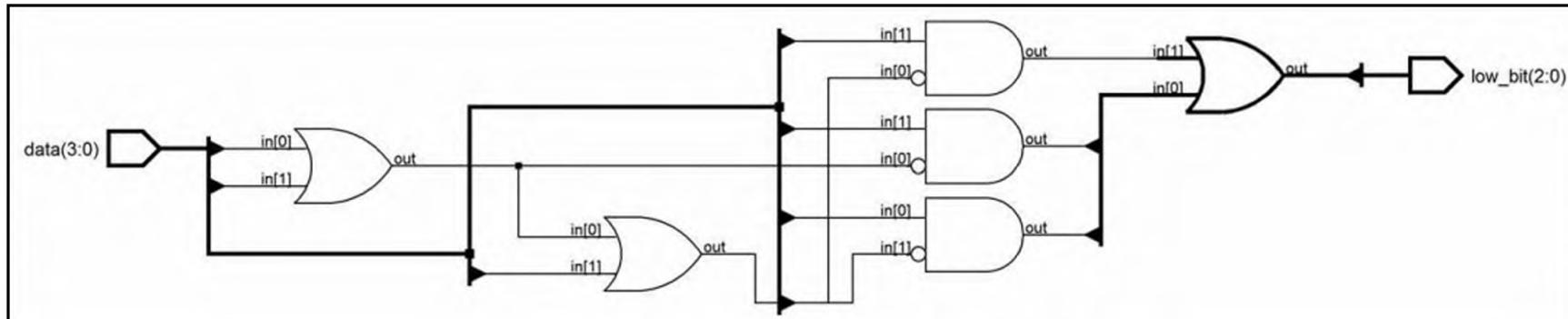
module find_lowest_bit
#(parameter N = 4) // bus size
(input logic [N-1:0] data,
 output logic [$clog2(N):0] low_bit
);

logic done; // local flag

always_comb begin
 // find lowest bit that is set in a vector
 low_bit = '0;
 done = '0;
 for (int i=0; i<=N-1; i++) begin
   if (!done) begin
     if (data[i]) begin
       low_bit = i;
       done = '1;
     end
   end
 end
 end
endmodule: find_lowest_bit

```

Figure 6-8: Synthesis result for Example 6-8: for-loop to find lowest bit set



Technology independent schematic (no target ASIC or FPGA selected)

Best Practice Guideline 6-4

Code all loops with a fixed iteration size. This coding style ensures the loop can be unrolled, and will be supported by all synthesis compilers.

6.3.1.3 For-loop iterator variable lifetime and visibility

The variable used to control a for loop is referred to as the *loop iterator* variable. Typically, the loop iterator is declared as part of the initial assignment, as in:

```
for (int i=0; ...
```

When declared as part of the initial assignment, the iterator variable is local to the **for** loop, and cannot be referenced outside of the loop. A local iterator variable is *automatic*, meaning the variable is created at the simulation time in which the loop starts, and disappears when the loop exits.

The iterator variable can also be declared outside of the **for** loop, such as at the module level or in a named begin-end group (see section 6.1.2, page 215). An externally declared iterator variable exists after the loop exits, and can be used elsewhere in the same scope in which the variable was declared. The value of the external variable when the loop exits will be the last value assigned by the step assignment before the end condition evaluated as false.

6.3.2 Repeat loops

A *repeat loop* executes a loop a set number of times. The general syntax of a **repeat** loop is:

```
repeat (iteration_number)
    statement_or_statement_group
```

The following example uses a **repeat** loop to raise a data signal to the power of 3 (data cubed).

```
always_comb begin
    result = data;
    repeat (2) begin
        result = result * data;
    end
end
```

SystemVerilog has an exponential power operator (see section 5.12, page 184, in Chapter 5), but some synthesis compilers do not support this operator. The code snippet above shows how a **repeat** loop can be used to perform an exponential operation algorithmically, by repeatedly multiplying a value with itself.

As with **for** loops, a **repeat** loop is synthesizable if the bounds of the loop are static, meaning the number of times the loop will iterate is fixed, and not dependent on the value of something that can change during simulation.

Example 6-9 shows a more complete example of the exponential operation snippet above. In this example the width of the data input and the exponent or the power operation are parameterized in order to make the example more versatile. Parameters are run-time constants that become fixed at compilation time. Therefore, a **repeat** loop that uses a parameter for the iteration number is a static loop that is synthesizable. The

output of the model, `q`, is sequential logic, and so `q` is assigned using a nonblocking assignment. The iterations within the loop are combinational logic, the final result of which is registered in `q`. Blocking assignments are used for the temporary variable, so that its new value is always available for the next iteration of the loop or to store in `q`. Chapter 8, section 8.1.4.3 (page 283) discuss when it is appropriate to use of blocking assignments in a sequential logic block.

Example 6-9: Using a repeat loop to raise a value to the power of an exponent

```

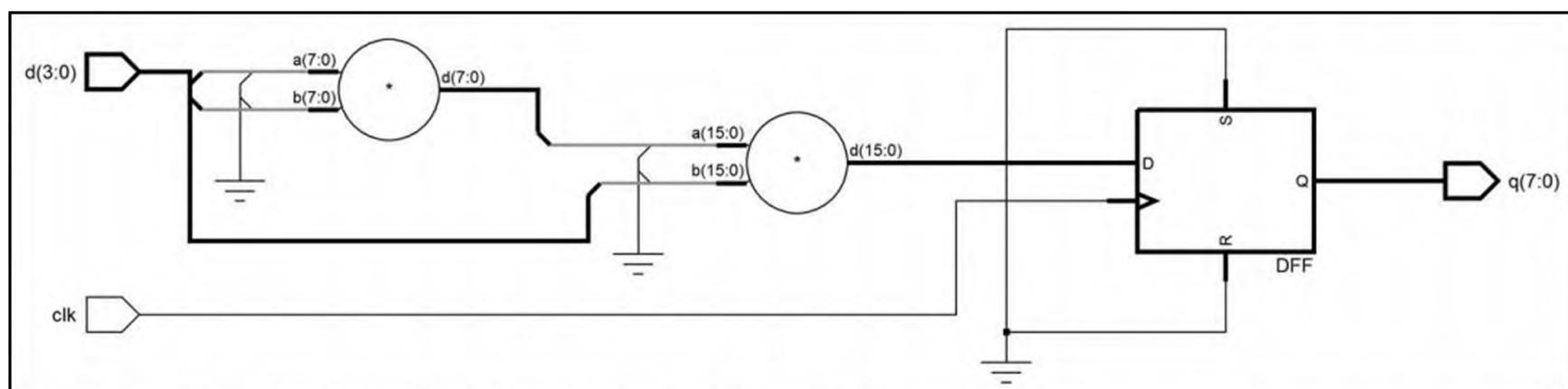
module exponential
#(parameter E = 3,      // power exponent
  parameter N = 4,      // input bus size
  parameter M = N*2    // output bus size
)
(input  logic          clk,
 input  logic [N-1:0] d,
 output logic [M-1:0] q
);

always_ff @(posedge clk) begin: power_loop
  logic [M-1:0] q_temp; // temp variable for inside the loop
  if (E == 0)
    q <= 1; // do to power of 0 is a decimal 1
  else begin
    q_temp = d;
    repeat (E-1) begin
      q_temp = q_temp * d;
    end
    q <= q_temp;
  end
end: power_loop
endmodule: exponential

```

Figure 6-9 shows the result of synthesizing Example 6-9. With `E` having a value of 3, the repeat loop executes 2 times, resulting synthesis creating 2 instances of a multiplier. Each bit of the output vector `q` is registered by a generic flip-flop. Only the first of the output register flip-flops are shown in this figure.

Figure 6-9: Synthesis result for Example 6-9: repeat loop to raise to an exponent



Technology independent schematic (no target ASIC or FPGA selected)

Synthesis timing considerations. A static, zero-delay **for** loop or **repeat** loop will synthesize to combinational logic. If the output of this combinational logic will be registered in flip-flops, then the total propagation delay of the combinational logic inferred by the loop must be less than one clock cycle.

NOTE

The capabilities and limitations of each specific ASIC or FPGA device can vary widely. RTL models that use the multiply, divide, modulus and power operators should be written to match the capabilities of the target device.

Observe that, in Figure 6-9, the multipliers inferred by the **repeat** loop in Example 6-9 are cascaded. The total propagation delay of the chain of multipliers needs to fit within one clock cycle in order for a valid and stable result to be registered in the output flip-flops. Some synthesis compilers can do *register retiming*, to insert or move registers to create a pipeline within the combinational logic. Register retiming is a feature of synthesis compilers, and is outside the scope of this book. Refer to the documentation of a specific synthesis compiler for more information on this topic.

If register retiming is not available, then a loop that does not meet the clock period of the design will need to be re-coded as a pipeline or state machine in order to manually break the loop into multiple clock cycles.

6.3.3 While and do-while loops

Best Practice Guideline 6-5

Use **for** loops and **repeat** loops for RTL modeling. Do not use **while** and **do-while** loops.

Although these loops are supported by many synthesis compilers, they have restrictions that limit their usefulness in RTL models, and can make code difficult to maintain and reuse. Instead, use **for** loops or **repeat** loops with a static number of times the loop will iterate. The **while** and **do-while** loops are shown in this section for completeness, but are not recommended.

A *while loop* executes a programming statement or begin-end group of statements until an *end_expression* becomes false. The end expression is tested at the top of the loop. If the end expression is false when the loop is first entered, the statement or statement group is not executed at all. If the end expression is true, the statement or statement group is executed, and then the loop returns back to the top and tests the end expression again.

A *do-while loop* also executes a programming statement or begin-end group of statements until an *end_expression* becomes false. With a **do-while** loop, the end

expression is tested at the bottom of the loop. Thus, the statements in the loop will always be executed a first time when the loop is first entered. If the end expression is false when the loop reaches the bottom, the loop exits. If the end expression is true, the loop returns back to the top and executes the statement or statement group again.

The following code shows a non-synthesizable example of using a **while** loop:

```
always_comb begin: count_ones
    logic [15:0] temp; // local temporary variable
    num_ones = 0;
    temp = data;
    while (temp) begin // loop as long as a bit in temp is set
        if (temp[0]) num_ones++;
        temp >>= 1; // shift bits of temp right by 1
    end
end: count_ones
```

This example counts how many bits of the 16-bit data signal are set to 1. The value of data is copied into a temporary variable called temp. If bit 0 of temp is set, the num_ones counter is incremented. The temp variable is then shifted right 1 time, which shifts out bit 0 and shifts a 0 into bit 15. The loop continues as long as temp evaluates as true, meaning at least one bit of temp is still set. When temp evaluates as false, the loop exits. A value in temp that has X or Z in some bits and no bits set to 1 would also cause the **while** loop to exit.

This example is non-synthesizable because the number of times the loop will execute is data-dependent, rather than static, as discussed earlier in this chapter, in section 6.3.1.1 (page 230). Synthesis cannot statically determine how many times the loop will execute, and therefore cannot roll out the loop.

6.3.4 *Foreach loops and looping through arrays*

A *foreach loop* iterates through all the dimensions of an unpacked array. An unpacked array is a collection of nets or variables, where the collection can be manipulated as a whole by using the array name, or individual elements of the array can be manipulated using an index into the array. The elements of an array can be any data type and vector size, but all elements of the array must be the same type and size. Arrays can have any number of dimensions. Some examples of array declarations are:

```
// a 1-dimensional unpacked array of 4096 8-bit variables
logic [7:0] mem [0:4095];

// a 2-dimensional unpacked array of 32-bit variables
logic [31:0] look_up_table [8][256];
```

The number of elements in each dimension of an array can be specified by using a **[starting_address:ending_address]** style, as with the mem array above, or by using a **[dimension_size]** style, as with look_up_table array. Section 3.7 (page 89) of Chapter 3 discusses declaring and working with unpacked arrays in more detail.

The **foreach** loop is used to iterate through array elements. The **foreach** loop will automatically declare its loop control variables, automatically determine the starting and ending indices of the array, and automatically determine the direction of the indexing (increment or decrement the loop control variables).

The following example iterates through a 2-dimensional array that represents a look-up table with some data. For each element in the array, a function is called to do some sort of manipulation on that value (the function is not shown).

```
bit [7:0] LUT [0:7][0:255]; // look-up table (2-state)

always @(posedge clk)
  if (update) begin
    foreach (LUT [i,j] ) begin
      update_function(LUT[i][j]);
    end
  end
```

Note that the *i* and *j* variables are not declared — the foreach-loop automatically declares these variables internally. Nor is it necessary to know the bounds each dimension of the array. The foreach-loop automatically iterates from the lowest index value to the highest index value for each dimension.

NOTE

At the time this book was written, some synthesis compilers did not support the **foreach** loop. Engineers should make sure all tools used in a project support this loop type before using it in RTL models.

An alternate coding style to iterate through all dimensions of an array is to use for-loops. The preceding example could be rewritten using static **for** loops that all synthesis compilers support.

```
always @(posedge clk)
  if (update) begin
    for (int i=0; i<=7; i++) begin
      for (int j=0; j<=255; j++) begin
        update_function(LUT[i][j]);
      end
    end
  end
```

Observe that, in this nested for-loop example, the size of each array dimension and its starting and ending index values must be hard-coded to match the array declaration. SystemVerilog also provides array query system functions, which can be used to make the for-loop more generic and adaptable to arrays of different sizes or parameterized sizes. The preceding example can be written as:

```

always @(posedge clk)
  if (update) begin
    for (int i=$left(LUT,1);
          i<=$right(LUT,1);
          i=i-$increment(LUT,1)) begin
      for (int j=$left(LUT,2);
            j<=$right(LUT,2);
            j=j-$increment(LUT,2)) begin
        update_function(LUT[i][j]);
      end
    end
  end

```

NOTE

At the time this book was written, some synthesis compilers did not support the array query system functions. Engineers should make sure all tools used in a project support these functions before using them in RTL models.

Following is a brief description of the array query system functions. Refer to the IEEE 1800 SystemVerilog Language Reference Manual for more information on these array query functions.

\$right(array_name, dimension) — Returns the right-most index 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.

\$left(array_name, dimension) — Returns the left-most index number of the specified dimension. Dimensions are numbered the same as with \$right.

\$increment(array_name, dimension) — Returns 1 if \$left is greater than or equal to \$right, and -1 if \$left is less than \$right.

\$low(array_name, dimension) — Returns the lowest index number of the specified dimension, which may be either the left or the right index.

\$high(array_name, dimension) — Returns the highest index number of the specified dimension, which may be either the left or the right index.

\$size(array_name, dimension) — Returns the total number of elements in the specified dimension (same as \$high - \$low + 1).

\$dimensions(array_name) — Returns the number of dimensions in the array, including both packed and unpacked dimensions.

6.4 Jump statements

Jump statements allow procedural code to skip over one or more programming statements. The SystemVerilog jump statements are **continue**, **break** and **disable**.

6.4.1 The continue and break jump statements

The **continue** and **break** jump statements are used within loops to control the execution of statements within the loop. These jump statements can only be used in for-loops, while-loops and foreach loops. They cannot be used outside of a loop.

The **continue** statement jumps to the end of a loop and evaluates the end expression of the loop to determine if the loop should continue for another iteration. The following code snippet uses a for-loop to iterate through the addresses of a small look-up-table modeled as a 1-dimensional array of 16-bit words. Locations in the table with a value of 0 are skipped by using the **continue** statement. For non-zero locations, a function is called to do some sort of manipulation on that value (the function is not shown).

```
bit [15:0] LUT [0:255]; // look-up table (2-state storage)

always_ff @(posedge clk)
  if (update) begin
    for (int i = 0; i <= 255; i++) begin
      if (LUT[i] == 0) continue; // skip empty elements
      update_function(LUT[i], new_data);
    end
  end
```

The **break** statement terminates the execution of a loop immediately. The loop exits, and any loop control statements, such as a **for** loop step assignment are not executed.

Example 6-10 illustrates the use of both **continue** and **break** to find the first bit that is set within a range of bits. Figure 6-10 shows the results of synthesizing this example.

Example 6-10: Controlling for loop execution using continue and break

```
module find_bit_in_range
```

```

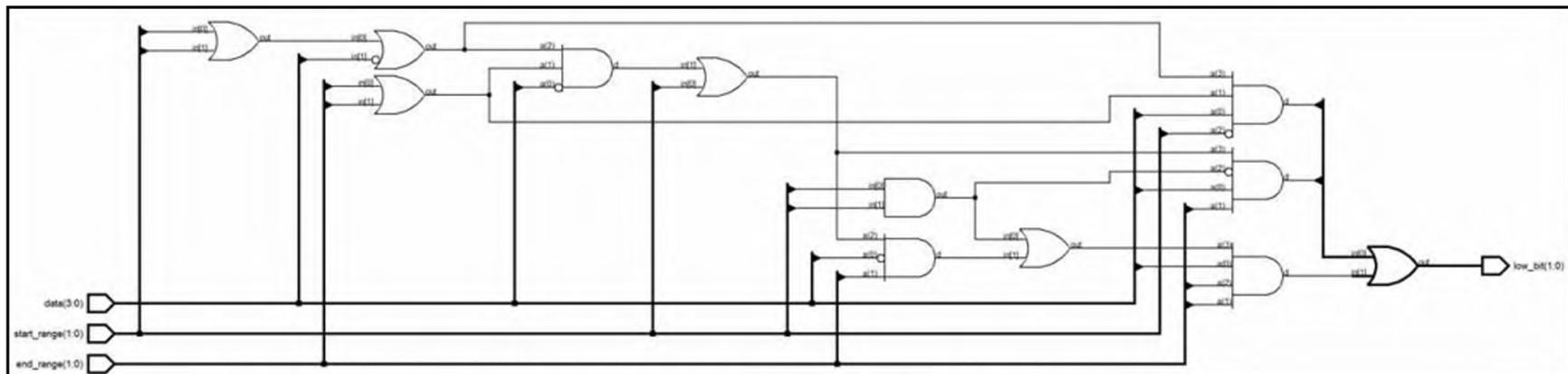
#(parameter N = 4) // bus size
(input logic [N-1:0]          data,
 input logic [$clog2(N)-1:0] start_range, end_range,
 output logic [$clog2(N)-1:0] low_bit
);

always_comb begin
    low_bit = '0;
    for (int i=0; i<N; i++) begin
        if (i < start_range) continue; // skip rest of loop
        if (i > end_range)   break;   // exit loop
        if (data[i]) begin
            low_bit = i;
            break;                  // exit loop
        end
    end // end of the loop
    // ... // process data based on lowest bit set
end

endmodule: find_bit_in_range

```

Figure 6-10: Synthesis result for Example 6-10



Technology independent schematic (no target ASIC or FPGA selected)

6.4.2 The disable jump statement

The SystemVerilog **disable** statement is analogous to a go-to statement in other programming languages. The **disable** jumps to the end of a named group of statements or to the end of a task. The general usage of a **disable** jump statement is:

```

begin : block_name
    repeat (64) begin
        // do something useful
        if (parity_error) disable block_name ; // exit loop early
    end
end: block_name

```

In this code snippet, the begin-end statement group was given the name `search_loop`. The `disable` statement instructs simulation to immediately jump to the end of this named begin-end group.

The original Verilog language did not have `continue` and `break` jump statements. Instead the general purpose go-to behavior of the `disable` statement was used to jump to the end of a loop, but continue execution of the next pass of the loop. The `disable` statement also had to be used to prematurely break out of a loop, by jumping past the end of the loop. To jump over statements within a loop but continue executing the loop, the named begin-end group must be contained within the loop. To break out of a loop, the named begin-end group must enclose the entire loop.

The following example shows the same functionality as Example 6-10, except using `disable` jump statements instead of `continue` and `break` statements.

```
always_comb begin
    low_bit = '0;
    begin: loop_block
        for (int i=0; i<N; i++) begin: loop
            if (i < start_range) disable loop; //skip rest of loop
            if (i > end_range)   disable loop_block; // exit loop
            if ( data[i] ) begin
                low_bit = i;
                disable loop_block; // exit loop
            end
        end: loop
    end: loop_block
    // ... // process data based on lowest bit set
end
```

Best Practice Guideline 6-6

Use the `continue` and `break` jump statements to control loop iterations. Do not use the `disable` jump statement.

The `disable` jump statement can be used to give the same functionality as `break` and `continue` jump statements, as shown above. However, the `disable` jump statement makes the code more difficult to read and to maintain. Using `continue` and `break` is a simpler and more intuitive coding style.

The `disable` jump statement is a general purpose go-to that can be used in ways that can be useful in verification testbenches. These other ways of using `disable` are not generally supported by synthesis compilers.

6.5 No-op statement

SystemVerilog programming statements are completed with a semicolon (;). A semicolon by itself is considered a complete programming statement, as well. Since there is no functionality to be executed, the solitary semicolon performs a *null operation*, often referred to as a *no-op statement*.

The following code snippet represents a register (using flip-flops) that stores the data variable. A multiplexed input, represented by the **case** statement, determines the value to be stored in the data register.

```
always_ff @(posedge clk)
  case (mode)
    2'b00: data <= data_in; // load data
    2'b01: data <= data << 1; // shift left
    2'b10: data <= data >> 1; // shift right
  endcase
```

The case statement in this code snippet does not decode the value of 2'b11 for mode. Although functionally correct in this example, the incomplete case statement is not self-documenting, and could lead to questions during a code review, or should another engineer need to maintain or reuse the code. Was not decoding a mode value of 2'b11 intentional, or was it an oversight (a bug) in the model? There is nothing in this example to indicate one way or the other. Adding a comment about the unused 2'b11 value would be helpful, but it is common to come across code that is not well commented.

Using a no-op statement can help make the RTL model more self-documenting. The following code snippet is functionally identical to the preceding example, but, even if there were no comments, it is obvious that it is intentional that a mode value of 2'b11 is not intended to change the data register.

```
always_ff @(posedge clk)
  case (mode)
    2'b00: data <= new_data ; // load new data into register
    2'b01: data <= data << 1; // shift data left
    2'b10: data <= data >> 1; // shift data right
    2'b11: ; // do nothing
  endcase
```

A no-op statement in sequential logic is ignored by synthesis compilers. There is no functionality to be implemented, so the register will retain its stored value. A no-op statement in combinational logic, however, cannot be ignored by synthesis compilers. When no assignment to a variable is made, it will retain its previous value. Synthesis might add a latch so that the logic can hold previous values. Latch inference in combinational logic is discussed in Chapter 9, section 9.2 (page 327).

Best Practice Guideline 6-7

Do not use the no-op statement for RTL modeling.

Although the no-op is supported by synthesis compilers, it serves no purpose in RTL functionality, and can lead to unintended latches in combinational logic. The no-op statement was discussed for completeness, but is not recommended in RTL code.

6.6 Functions and tasks in RTL modeling

SystemVerilog has *functions* and *tasks* that make it possible to partition complex functionality into smaller, reusable blocks of code. Functions can be very useful for RTL modeling, and are examined in this section. Tasks, though synthesizable with limitations, have little value in RTL models. Using void functions, which are discussed later in this section, is a better RTL coding style than using tasks. Therefore, tasks are only discussed briefly in this book.

Functions and tasks can be defined within the module or interface (see chapter 10) in which they are used. The definition can appear before or after the statements that call the function or task. Functions and tasks can also be defined in a package, and then imported into the module or interface. The package **import** statement must appear before the function or task is called. Packages and package importing are discussed in Chapter 4, section 4.2 (page 102).

6.6.1 Functions

When called, a function executes its programming statements and returns value. A call to a function can be used anywhere an expression such as a net or variable can be used. An example function definition and call to the function are shown here. More practical synthesizable examples are shown later in this section.

```
function automatic logic [N-1:0] factorial_f([N-1:0] in);
    logic [N-1:0] f;
    if (in <= 1) f = 1;
    else          f = in * factorial_f(in-1);
    return f;
endfunction: factorial_f

always_ff @(posedge clk)
    out <= factorial_f(a) + factorial_f(b);
```

SystemVerilog syntax requires that functions execute in zero simulation time. A synthesizable function cannot contain clock cycles or propagation delays.

Static and automatic functions. Functions (and tasks) can be declared as **static** or **automatic**. If neither is specified, the default is static for functions defined in a module, interface or package.

A *static function* retains the state of any internal variables or storage from one call to the next. The function name and function inputs are implicit internal variables, and will retain their values when the function exits. The effect of this static storage is that a new call to a function can remember values from a previous call. This memory can be useful in verification code, but the behavior does not always accurately model the gate-level behavior that synthesis compilers implement from functions, which can lead to a mismatch between the RTL model simulations and the actual functionality of an ASIC or FPGA.

An *automatic function* allocates new storage each time the function is called. Recursive function calls, such as the `factorial_f` function example shown above, require automatic storage. (Re-entrant task calls, where two different procedures call the same task at the same time, also require automatic storage.)

Best Practice Guideline 6-8

Declare functions used in RTL models as **automatic**.

The default of static storage is not appropriate for RTL modeling of hardware behavior. Furthermore, synthesis compilers require that functions declared in a package or interface must be declared as automatic.

There is an historical reason that functions default to static storage. In the early years of Verilog simulation, when computer memory was limited and processor power was much slower, static storage helped improve simulation run-time performance. There is no real performance advantage of static storage versus automatic storage with modern simulators and compute servers. The SystemVerilog standard has kept the original language default of static functions in order to remain backward compatible with legacy verification code that might have been written to utilize the static storage of a function.

Function returns. The return data type of a function is defined immediately before the name of the function. In the `factorial_f` example above, the function returns an N-bit wide vector with a **logic** (4-state) type. If no return type is specified, functions return a 1-bit **logic** (4-state) type by default.

SystemVerilog provides two ways to specify the return value from a function. One way is to use the **return** keyword, as shown in the preceding `factorial_f` example above. The **return** keyword is followed by the value to be returned by the function. Optionally, this return value can be enclosed in parentheses.

A second way to specify the return value is to assign a value to the name of the function. The function name is an implicit variable of the same data type as the return

type. This implicit variable can be used for temporary storage while the function is calculating the return value. The last value assigned to the function name becomes the function return value. The `factorial_f` function shown at the beginning of this section could be re-coded to use the function name as an implicit internal variable to calculate and return a value.

```
function automatic logic [N-1:0] factorial_f([N-1:0] in);
    if (in <= 1) factorial_f = 1;
    else          factorial_f = in * factorial_f(in-1);
endfunction: factorial_f
```

Void functions. Optionally, a function return type can declared as `void`. Void functions do not return a value, and cannot not be used as an expression like other functions. A void function is called as a statement, instead of as an expression.

```
typedef struct {
    logic [31:0] data;
    logic [ 3:0] check;
    logic         valid;
} packet_t;

function void set_packet_f ([31:0] in, output packet_t out);
    out.data = in;
    for (int i=0; i<=7; i++)
        out.check[i] = ^in[(8*i)+:8];
    out.valid = 1;
endfunction
```

Best Practice Guideline 6-9

Use void functions in place of tasks for RTL modeling. Only use tasks in verification code.

The only difference between a void function and a task is that a function must execute in zero time. Most synthesis compilers do not support any form of clock delay in tasks. Using a void function in place of a task makes this synthesis restriction a syntax requirement, and can prevent writing RTL models that simulate, but will not synthesize.

Function arguments. The arguments in the definition of a function are referred to as the *formal arguments*. The arguments in a call to a function are referred to as the *actual arguments*.

The formal arguments can be `input`, `output` or `inout`, and are declared with the same syntax as module ports. The default direction, if not defined, is `input`. The `in` formal argument in the `fill_packet` example above is a 32-bit 4-state `input`, and `out` is an `output` formal argument of the user-defined `packet_t` type.

A formal argument can also be declared as **ref** (short for *reference*) instead of a direction. A **ref** argument is a form of a pointer to the actual argument of the call to the function. A function must be declared as **automatic** to use **ref** arguments.

Best Practice Guideline 6-10

Only use **input** and **output** formal arguments in functions used in RTL models. Do not use **inout** or **ref** formal arguments.

All RTL synthesis compilers support **input** and **output** function arguments. The **inout** and **ref** arguments are not supported by some RTL synthesis compilers.

Calling functions. There are two coding styles for passing actual arguments to the formal arguments when a function is called: *pass-by-order* and *pass-by-name*. With pass-by-order, the first actual argument is passed to the first formal argument, the second actual argument to the second formal argument, and so forth. Pass-by-name uses the same syntax as connecting modules by name. The name of the formal argument is preceded by a period (.), followed by the actual argument enclosed in parentheses.

Given the function definition:

```
function automatic int inc_f(int count, step);
    return (count + step);
endfunction
```

The two styles of passing actual arguments are:

```
always_ff @(posedge master_clk)
    m_data <= inc_f(data_bus, 1); // pass-by-order

always_ff @(posedge slave_clk) // pass-by-name
    s_data <= inc_f(.count(data_bus), .step(8));
```

Function input default values. Formal input arguments can be assigned a default value, as in:

```
function automatic int inc_f(int count, step=1);
    return (count + step);
endfunction
```

Arguments with a default value do not need to be passed an actual value. If no actual value is passed in, the default value is used. For example:

```
always_ff @(posedge master_clk)
    m_data <= inc_f( .count(data_bus) );
```

If an actual value is passed in, the actual value is used, as in:

```
always_ff @(posedge slave_clk)
    s_data <= inc_f( .count(data_bus), .step(8) );
```

NOTE

Default input values were not supported by some synthesis compilers at the time this book was written. Engineers should make sure all tools in the design flow used in a project support default input values before using them in RTL models.

Using return to exit a function early. The **return** statement can also be used to exit from a function before all statements in the function have been executed. The following example can exit the function at 3 different points. If the `max` input is 0, the function exits prior to executing the `for` loop. If the `for` loop iterator reaches the value of `max`, the function exits before reaching the end of the loop. If the `for` loop completes, the function exits when the `endfunction` is reached.

```
parameter N = 32;

function automatic void sum_to_endpoint_f
(output [N-1:0] result,
input  [$clog2(N)-1:0] endpoint,
input  [N-1:0] data_array [64] // look-up-table array
);
    result = data_array[0];
    if (endpoint == 0) return; // exit the function early
    for (int i=1; i<=63; i++) begin
        result = result + data_array[i];
        if (i == endpoint) return; // exit the function early
    end
endfunction // exit at completion of function
```

Parameterized functions. Parameterized modules are a powerful and widely used capability in SystemVerilog. Parameters can be redefined for each instance of the module, making the module easily configurable and reusable. Module-level parameters can be used in function definitions, as shown in the preceding `sum_to_endpoint_f` function example. Using module-level parameters means that all calls to the function will have the same vector size. The function cannot be configured so that each place the function is called uses a different vector size.

Functions cannot be parameterized in the same way that modules can be. SystemVerilog does not allow function definitions to have internal parameters that can be redefined each place the function is called. This limits the ability to write reusable, configurable functions. There is a work-around for this limitation, however, which is to declare static functions in a parameterized virtual class. A static function within a class definition can be called directly using a scope resolution operator (::), without creating an object.

Each place the function is called, the class parameters can be redefined, as shown in the following example:

```

virtual class Functions #(parameter SIZE=8);
  static function [SIZE-1:0] adder_f (input [SIZE-1:0] a, b);
    return a + b; // defaults to 8-bit adder
  endfunction
endclass

always_comb begin
  y16 = Functions #(SIZE(16))::adder_f(a16, b16);
    // reconfigure to 16-bit adder
  y32 = Functions #(SIZE(32))::adder_f(a32, b32);
    // reconfigure to 32-bit adder
end

```

Parameterized functions make it is possible to create and maintain only one version of the function, instead of having to define several versions with different data types, vector widths, or other characteristics.

Observe that, in a class definition, the **static** keyword comes before the **function** keyword, whereas in a module, the **static** or **automatic** keyword comes after the **function** keyword. There is an important semantic difference. In a class, **static function** declares the lifetime of the function within the class, and restricts what the function can access within the class. in a module, **function static** or **function automatic** refers to the lifetime of the arguments and variables within the function.

NOTE

At the time this book was written, not all synthesis compilers supported static functions in parameterized virtual classes. Engineers should make sure all tools used in a project support static functions in parameterized virtual classes before using them in RTL models.

6.6.2 Tasks

A task is a subroutine that encapsulates one or programming statements, so that the encapsulated statements can be called from different places or reused in other projects. Unlike functions, tasks do not have a return value. An example task is:

```

task automatic ReverseBits (input [N-1:0] in,
                           output [N-1:0] out);
  for (int i=0; i<N; i++)
    out [(N-1)-i] = in[i];
endtask

```

A task is called as a programming statement, and uses output formal arguments to pass values out of the task.

```
always_ff @(posedge clk) begin
    ReverseBits(a, a_reversed);
    ReverseBits(b, b_reversed);
end
```

Syntactically, a task is very similar to a function, except that a task does not have a return type. An important difference between tasks and functions is that tasks can contain clock cycles and propagation delays. Most synthesis compilers, however, require that the programming statements within a task run in zero simulation time. This synthesis restriction makes tasks nearly identical to void functions. Since void functions syntactically enforce zero-time execution, a best coding practice is to use void functions instead of tasks when subroutines are needed in RTL models. Void functions are discussed in section 6.6.1. The ReverseBits task can be rewritten as a void function as follows:

```
function automatic void ReverseBits (input [N-1:0] in,
                                         output [N-1:0] out);
    for (int i=0; i<N; i++)
        out[(N-1)-i] = in[i];
endfunction

always_ff @(posedge clk) begin
    ReverseBits(a, a_reversed);
    ReverseBits(b, b_reversed);
end
```

6.7 Summary

SystemVerilog has a robust set of programming statements, in the form of decisions, loops, jumps and a go-to. This chapter has focused on the programming statements that are supported by most synthesis compilers, and the coding restrictions required to ensure RTL models will simulate correctly and synthesize correctly.

RTL simulators and synthesis compilers need to know when to execute programming statements. An always procedure with a sensitivity list (explicit or inferred) is used to control when statements are executed. SystemVerilog has four types of always procedures, the generic **always**, and the type-specific, **always_ff**, **always_comb** and **always_latch** procedures. This chapter has introduced these constructs and used them in a number of code examples. The next chapters will examine the proper usage of these always procedures in much greater detail as the topics of modeling combinational logic, sequential logic and latched logic components are examined.

Programming statements can also be contained in functions and tasks, which are called from always procedures. The rules and best coding practices for functions and tasks were covered in this chapter.

Chapter 7

Modeling Combinational Logic

Abstract — This chapter builds on the programming statements and operators discussed in the previous chapters, and adds more details on best-practice coding styles for RTL models of combinational logic. An emphasis is placed on writing RTL models that ensure simulation behavior matches post-synthesis gate-level behavior.

Digital gate-level circuitry can be divided into two broad categories: *combinational logic*, discussed in this chapter, and *sequential logic*, discussed in the next chapter. Latches are a cross between combinational and sequential logic, and are treated as a separate topic in Chapter 9.

Combinational logic describes gate-level circuitry where the outputs of a block of logic directly reflect a combination of the input values to that block. The output of a two-input AND gate, for example, is the logical-and of the two inputs. If an input value changes, the output value will reflect that change. RTL models of combinational logic need to reflect this gate-level behavior, meaning that the output of a block of logic must always reflect a combination of the current input values to that block of logic.

SystemVerilog has three ways to represent combinational logic at a synthesizable RTL level: continuous assignments, always procedures, and functions. Each of these coding styles is explored in this chapter, and best-practice coding styles are recommended.

The topics presented in this chapter include:

- Continuous assignment statements
- Always procedures, when modeled following strict coding guidelines
- The `always_comb` procedure and simulation rules
- The obsolete `always @*` procedure
- Using functions to model combinational logic

7.1 Continuous assignments (Boolean expressions)

A continuous assignment drives an expression or the result of an operation onto a net or a variable. An explicit continuous assignment is a statement that begins with the **assign** keyword. A simple example of a continuous assignment is:

```
assign sum = a + b;
```

The left-hand side of the assignment, `sum` in the example above, is updated whenever any change of value occurs on the right-hand side, which is whenever `a` or `b` changes in the example above. This continuous updating of the left-hand side whenever the right-hand side changes is what models the behavior of combinational logic.

The continuous assignment syntax allows for a propagation delay to be specified between when a change on the right-hand side occurs and when the left-hand side is updated. Synthesis compilers, however, expect zero-delay RTL models, and will ignore delays in continuous assignments. This can lead to a mismatch between a design that was verified with delays, and the synthesized implementation that ignored the delays. Only zero-delay examples are shown in this book.

Left-side types. The left-hand side of a continuous assignment can be a scalar (1-bit) or vector net or a variable type, or a user-defined type. The left-hand side cannot be an unpacked structure or unpacked array.

There is an important difference between using a net and a variable on the left-hand side of a continuous assignment:

- Net types, such as **wire** or **tri**, can be driven by multiple sources, including multiple continuous assignments, multiple connections to output or inout ports of module or primitive instances, or any combination of drivers.
- Variable types, such as **var** or **int**, can only be assigned a value from a single source, which can be: a single input port, a single continuous assignment, or any number of procedural assignments (multiple procedural assignments are considered to be a single source; synthesis requires the multiple procedural assignments be in the same procedure).

Note that the **logic** keyword infers a data type, but is not, in itself, a net or variable type. When **logic** is used by itself, a variable is inferred, with its single-source assignment restriction). A variable is also inferred when the keyword pair **output logic** is used to declare a module port. when the keyword pair **input logic** or **inout logic** is used to declare a module port, a **wire** net type is inferred, with its multiple driver capability.

Chapter 3, sections 3.4.1 (page 68) and 3.6.1 (page 84) discuss the rules and proper usage of the logic type in more detail.

Best Practice Guideline 7-1

Use variables on the left-hand side of continuous assignments to prevent unintentional multiple drivers. Only use `wire` or `tri` nets on the left-hand side when it is intended for a signal to have multiple drivers.

Only use a net type (such as `wire` or `tri`) when multiple drivers are intended, such as for a shared bus, a tri-state bus or an `inout` bidirectional module port. See section 3.6.1 (page 84) for more information declaring module port data types.

For RTL modeling, there is an important advantage to the semantic rule that variables can only have a single source. Most signals in ASIC and FPGA devices are expected to be single-source logic, with the exception of tri-state busses and bidirectional ports. The single-source restriction of variables can help prevent inadvertent coding errors, where multiple continuous assignments or connections are made to the same signal. With variable types, a multiple-source coding mistake will be reported as a compilation or elaboration error in both simulation and synthesis.

Vector width mismatches. The left-hand side of a continuous assignment can be a different vector size than the signal or expression result on the right-hand-side. When this occurs, SystemVerilog automatically adjusts the vector width of the right-hand side to match the size of the left-hand side. If the right-hand side is a larger vector than the left-hand side, the most-significant bits of the right-hand side will be truncated to the size of the left-hand side. If the right-hand side is a smaller vector size, the right-hand side value will be left-extended to the size of the left-hand side. The left extension will extend with zeros if the expression or operation result is unsigned. Sign extension will be used if the right-hand expression or operation result is signed. Chapter 3, section 3.4.3 (page 74) discusses assignment rules in more detail.

Best Practice Guideline 7-2

Ensure that both sides of continuous assignments and procedural assignments are the same vector width. Avoid mis-matched vector sizes on the left-hand and right-hand side expressions.

There are rare circumstances where it is intentional to have vectors of different sizes on the right-hand and left-hand side of an assignment. An example of this is a variable rotate operation, as shown in Chapter 5, section 5.10.2 (page 177). When a size mismatch is intentional, size casting documents within the RTL code that the mismatch is intentional. Size casting can also eliminate size mismatch warning messages from lint checker tools when the mismatch is intentional. This helps ensure that any unintentional mismatch warnings will not be overlooked. See Chapter 5, section 5.15.2 (page 202) for examples of using size casting.

7.1.1 Explicit and inferred continuous assignments

There are two forms of continuous assignments: *explicit continuous assignment statements* and *implicit net declaration continuous assignments*. An *explicit continuous assignment* is declared with the **assign** keyword, as has been shown in the preceding code snippets and examples. This form of continuous assignment can assign to both net and variable types. An *implicit net declaration continuous assignment* combines the declaration of a net type with a continuous assignment. The continuous nature of this form is inferred, even though the **assign** keyword is not used.

An example inferred net declaration assignment is:

```
wire [7:0] sum = a + b;
```

Note that an inferred net declaration assignment is not the same as a variable initialization, such as:

```
int i = 5;
```

A variable initialization is only executed one time, whereas an inferred net declaration assignment is a process that updates the left-hand net whenever there is a change of value on the right-hand expression. An inferred net declaration assignment is synthesizable. Variable initialization in synthesizable RTL models is discussed in Chapter 3, section 3.4.4 (page 75).

7.1.2 Multiple continuous assignments

A module can contain any number of continuous assignments. Each continuous assignment is a separate process that runs in parallel with other continuous assignments. All continuous assignments begin evaluating the right-hand side at simulation time zero, and run to the end of simulation.

Multiple procedural assignments in a module can be used to represent a dataflow behavior, where functionality is modeled with Boolean equations that use SystemVerilog operators to produce outputs, rather than using procedural programming statements. In RTL models, dataflow assignments represent the combinational logic through which data flows between registers.

The following example uses continuous assignments to model the flow of data through an adder, multiplier and subtractor. The result of this dataflow is stored in a register at each positive edge of a clock.

Example 7-1: Add, multiply, subtract dataflow processing with registered output

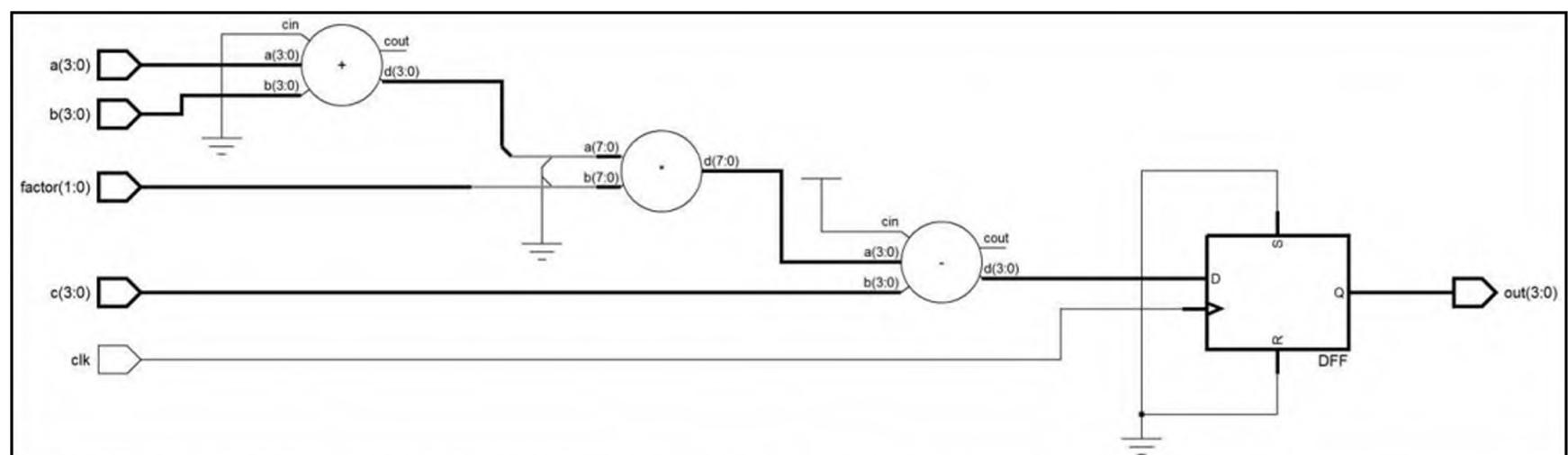
```

module dataflow
  #(parameter N = 4)                      // bus size
  (input logic clk,                         // scalar input
   input logic [N-1:0] a, b, c,             // scalable input size
   input logic [ 1:0] factor,              // fixed input size
   output logic [N-1:0] out                // scalable output size
  );
  logic [N-1:0] sum, diff, prod;
  assign sum = a + b;
  assign diff = prod - c;
  assign prod = sum * factor;
  always_ff @(posedge clk)
    out <= diff;
endmodule: dataflow

```

Because multiple continuous assignments in a module run in parallel, the order of the assignments in the RTL source code makes no difference. This can be seen by comparing the order of the continuous assignment statements in Example 7-1 and the dataflow order in synthesis results shown in Figure 7-1. The RTL code lists the assignment statements in the order of add, subtract, multiply, but the dataflow of the operations is add, multiply, subtract.

Figure 7-1: Synthesis result for Example 7-1: Continuous assignment as comb. logic



Technology independent schematic (no target ASIC or FPGA selected)

7.1.3 Using both continuous assignments and always procedures

A module can contain a mix of continuous assignments and always procedures.

The following simple example illustrates a simple static RAM with a bidirectional data bus. The data bus is driven as an output when reading from the RAM. When not being read, the data bus is assigned high-impedance, so that other devices can drive the bus. A continuous assignment is used to model the output functionality, and an

always procedure is used to model the input functionality in order to trigger on rising edges of the clock.

```

module SRAM (inout wire [7:0] data,
              input logic [7:0] addr,
              input logic      rw,    // 0 = read, 1 = write
              input logic      clk
            );

  logic [7:0] mem [0:255]; // array for RAM storage

  // drive data if rw = 0, tri-date data if rw = 1
  assign data = (!rw)? mem[addr] : 'Z;

  // synchronous write into RAM if rw = 1
  always @(posedge clk)
    if (rw) mem[addr] <= data;

endmodule: SRAM

```

The data bus is a bidirectional inout port, and must be a net type, such as **wire** or **tri**, in order to have multiple drivers. The data bus can be driven by the RAM when it is an output from the RAM, and by some other module when data bus is an input writing into the RAM. Only continuous assignment can assign to net data types.

Each continuous assignment and each always procedure is a separate process that runs in parallel, beginning at simulation time zero and running throughout simulation. The order of continuous assignments and always procedures within a module does not matter because the processes are running in parallel.

7.2 The **always** and **always_comb** procedures

The primary RTL modeling construct for combinational logic is the always procedure, using either the general purpose **always** keyword or the RTL-specific **always_comb** keyword. These always procedures can take advantage of the robust set of operators programming statements that are discussed in Chapters 5 and 6, whereas continuous assignments are limited to using only SystemVerilog operators. Examples of a simple combinational logic adder modeled as an **always** procedure and an **always_comb** procedure are:

```

always @(a, b) begin
  sum = a + b;
end

always_comb begin
  sum = a + b;
end

```

7.2.1 Synthesizing combinational logic always procedures

Both **always** and **always_comb** procedures are supported by synthesis compilers.

When using the general purpose **always** procedure, synthesis compilers impose several coding restrictions that the RTL design engineer must be aware of and adhere to. These restriction include:

- The procedure sensitivity list should include every signal for which the value can affect the output(s) of the combinational logic. Sensitivity lists are discussed in more detail in section 7.2.2.
- The procedure sensitivity list must be sensitive to all possible value changes of each signal. It cannot contain **posedge** or **negedge** keywords that limit the sensitivity to specific changes.
- The procedure should execute in zero simulation time, and should not contain any form of propagation delay using **#**, **@** or **wait** time controls.
- A variable assigned a value in a combinational logic procedure should not be assigned a value by any other procedure or continuous assignment. (Multiple assignments within the same procedure are permitted.)

Best Practice Guideline 7-3

Model all RTL combinational logic with zero delays.

Synthesis will not allow **@** or **wait** time control delays, and will ignore **#** delays. Ignoring **#** delays can lead to mismatches in the RTL models that were verified in simulation that used delays, and the gate-level implementation from synthesis that ignored the delays.

7.2.2 Modeling with the general purpose always procedure

Best Practice Guideline 7-4

Use the RTL-specific **always_comb** procedure to model combinational logic. Do not use the generic **always** procedure in RTL models.

The RTL-specific **always_comb** automatically enforces the coding restrictions listed above. The sensitivity is inferred, no **@** or **wait** time controls are permitted, and a variable assigned in an **always_comb** procedure cannot be assigned by any other procedure or continuous assignment.

Though not recommended for RTL modeling, properly using the general purpose **always** procedure for modeling combinational logic is discussed in this book because it is common to see this general purpose procedure in legacy Verilog models.

Combinational logic sensitivity lists. The general purpose **always** procedure requires a sensitivity list to tell simulators when to process the programming statements in the procedure. A sensitivity list is specified using the form `@(list_of_signals)`, as shown in the following example:

```
always @(a, b, mode) begin
    if (!mode) result = a + b; // add when mode = 0
    else        result = a - b;
end
```

Each signal in the sensitivity list can be separated by a comma, as in the example above, or by the keyword **or**, as in: `@(a or b or mode)`. There is no advantage or disadvantage to using commas versus the **or** keyword. Some engineers prefer the comma-separated list because the **or** keyword could be mistaken as a logical-OR operation, rather than just a separator between signals in the list.

Complete sensitivity lists. With combinational logic, the outputs of the combinational block are a direct reflection of the current values of the inputs to that block. In order to model this behavior, the **always** procedure needs to execute its programming statements whenever any signal changes value that affects the outputs of the procedure. An input to the combinational always procedure is any signal of which the value is read by the statements in the procedure. In adder example above, the inputs to the procedure — the signals that are read within the procedure — are: `a`, `b` and `mode`.

Procedure inputs versus module inputs. The inputs to a combinational logic procedure might not correspond to the input ports of the module containing the procedure. A module might contain several procedural blocks and continuous assignments, and, therefore, have input ports for each of these blocks. A module might also contain internal signals that pass values between procedural blocks or continuous assignments. These internal signals will not be included in the module port list.

Incomplete sensitivity lists — a modeling gotcha. A *gotcha* is a programming term for code that is syntactically legal, but which does not perform as expected. The general purpose **always** procedure allows making this type of coding mistake. If one or more inputs to the combinational logic procedure are inadvertently omitted from the sensitivity list, the RTL model will compile, and might even appear to simulate correctly. Thorough verification would show, however, that there are periods of time when the output(s) of the combinational logic block are not reflecting a combination of the current input values. Consider the following code snippet:

```
always @(a, b) begin
    if (!mode) result = a + b; // add when mode = 0
    else        result = a - b;
end
```

If `mode` changes value, the `result` output will not be updated to the new operation result until either `a` or `b` changes value. The value of `result` is incorrect during the time between when `mode` changed and `a` or `b` changed.

This coding mistake is an obvious one in small combinational logic blocks that only read the values of a few signals, but it is not uncommon for larger, more complex blocks of logic to read 10, 20 or even several dozen signals. It is easy to inadvertently omit a signal in the sensitivity list when so many signals are involved. It is also common to modify an always block during the development of a design, adding another signal to the logic, but forgetting to add it to the sensitivity list.

A serious hazard with this coding gotcha is that many synthesis compilers will still implement this incorrect RTL model as gate-level combinational logic, possibly with a warning message that is easy to overlook. Though the implementation from synthesis might be what the designer intended, it is not the design functionality that was verified during RTL simulation. Therefore, the design functionality was not fully verified, which could result in a bug in the actual ASIC or FPGA.

The obsolete always @* procedure. The IEEE 1364-2001 standard, often referred to as Verilog-2001, attempted to address the gotcha of incomplete sensitivity lists with the addition of a special token that would automatically infer a complete sensitivity list, @*. For example:

```
always @* begin
    if (!mode) result = a + b; // add when mode = 0
    else        result = a - b;
end
```

Optionally, the asterisk can be enclosed in parentheses, as in @(*). The @* token offers a better coding style than explicitly listing signals in a combinational logic sensitivity list. There are two problems with this token, however. First, synthesis compilers impose a number of restrictions on modeling combinational logic. Using @* infers a sensitivity list, but does not enforce other synthesis rules for modeling combinational logic. These rules are discussed in section 7.2.1 (page 257). The second problem with @* is a corner case where a complete sensitivity list is not inferred. If a combinational logic procedure calls a function, but does not pass in as function arguments all signals used within the function, an incomplete sensitivity list will be inferred.

Best Practice Guideline 7-5

Use the SystemVerilog **always_comb** RTL-specific procedure to automatically infer correct combinational logic sensitivity lists. Do not use the obsolete @* inferred sensitivity list.

An **always_comb** procedure will infer an accurate sensitivity list without the hazards of explicit lists or the corner-case problem of @*. The **always_comb** procedure also enforces the coding restrictions that synthesis compilers require for accurately modeling combinational logic behavior.

The original Verilog language that was introduced in the 1980s only had the general purpose **always** procedure. Though very useful, the general purpose nature of this procedure has important limitations when used for RTL modeling. As a general purpose procedure, **always** can be used to model combinational logic, sequential logic, latched logic and various verification processes. When a synthesis compiler encounters an **always** procedure, the compiler has no way to know what type of functionality a design engineer intended to model. Instead, a synthesis compiler must analyze the contents of the procedure and try to infer a designer's intent. It is all too possible for synthesis to infer a different type of functionality than what an engineer intended.

Another limitation of the general purpose **always** procedure is that it does not enforce RTL coding rules required by synthesis compilers for representing combinational logic behavior, as summarized in section 7.2.1 (page 257). Models using general purpose **always** procedures might appear to simulate correctly, but might not synthesize to the intended functionality, resulting in lost engineering time by having to rewrite the RTL models and reverify the functionality in simulation before the model can be synthesized.

7.2.3 Modeling with the RTL-specific `always_comb` procedure

SystemVerilog introduced the RTL specific always procedures, such as **always_comb**, to address the limitations of the general purpose **always** procedure. The following example models the same Arithmetic Logic Unit functionality shown previously, but using **always_comb** instead of **always**.

```
always_comb begin
    if (!mode) result = a + b; // add when mode = 0
    else      result = a - b;
end
```

The **always_comb** procedure has many benefits when writing RTL models:

- A complete sensitivity list is automatically inferred. This list is fully complete, and avoids the corner case where `@*` would infer an incomplete sensitivity list.
- Using `#`, `@` or `wait` to delay execution of a statement in an **always_comb** procedure is not permitted, enforcing the synthesis guideline for using zero-delay procedures. Using these time controls in **always_comb** is an error that will be caught during the compilation and elaboration of the RTL models.
- Any variable assigned a value in an **always_comb** procedure cannot be assigned from another procedure or continuous assignment, which is a restriction required by synthesis compilers. A coding mistake that violates this synthesis rule will be caught during compilation and elaboration of the RTL models.

The semantic rules of **always_comb** match the coding restrictions that synthesis compilers require for RTL models of combinational logic. These rules help to ensure that engineering time is not lost verifying a design that cannot be synthesized.

Automatic evaluation at the start of simulation. The `always_comb` procedure also has a semantic rule that is specific to simulation. The behavior of combinational logic is that the value of the outputs represent a combination of the input values to that block of logic. With a general purpose `always` procedure, a value change must occur to a signal in the sensitivity list in order to trigger an execution of the assignment statements within the procedure. If none of the signals in the sensitivity list change value at the start of simulation, the outputs of the combinational logic procedure are not updated to match the values of the inputs to the procedure at the start of simulation. The combinational logic `always` procedure will continue to have incorrect output values until a signal in the sensitivity list changes value. This problem is an RTL simulation glitch. The gate-level implementation will not have this problem.

The RTL-specific `always_comb` procedure resolves this simulation glitch. An `always_comb` procedure will automatically trigger once at the start of simulation, to ensure that all variables assigned in the procedure accurately reflect the values of the inputs to the procedure at simulation time zero.

7.2.4 Using blocking (combinational logic) assignments

Best Practice Guideline 7-6

Only use blocking assignments (`=`) when modeling combinational logic behavior.

SystemVerilog has two forms of assignment operators: a *blocking assignment* (`=`) and a *nonblocking assignment* (`<=`). These assignment types affect the order in which simulation updates the value of the left-hand side of an assignment statement, relative to any other simulation activity at that moment of simulation time. The blocking assignment (`=`) immediately updates the variable on the left-hand side, allowing the new value to be available for use by any subsequent statement in a begin-end sequence of statements. The immediate update effectively models the behavior of value propagation in combinational logic dataflow.

The following code snippet illustrates a combinational logic dataflow through multiple assignments in a combinational logic procedural block.

```
always_comb begin
    sum      = a + b;
    prod    = sum * factor;
    result  = prod - c;
end
```

In this procedure, the variable `sum` is immediately updated to the result of the operation `a + b`. This new value of `sum` then flows to the next statement, where the new value is used to calculate a new value for `prod`. This new value for `prod` then flows to the next line of code and is used to calculate the value of `result`.

The blocking behavior of the assignment statement is critical for this dataflow to simulate correctly in a zero-delay RTL model. The blocking assignment in each line of code blocks the evaluation of the next line, until the current line has updated its left-hand side variable with a new value. The blocking of the evaluation of each subsequent line of code is what ensures that each line is using the new value of variables assigned by the previous lines.

Had nonblocking assignments been inappropriately used in the code snippet above, each assignment would have used the previous values of its right-side variables, before those variables were updated to new values. This is not combinational logic behavior! Synthesis compilers, however, might still create combinational logic when nonblocking assignments are used, resulting in the behavior that was verified in RTL simulation not matching the actual gate-level behavior after synthesis.

Simulation event scheduling, and the execution of blocking and nonblocking assignments, are discussed in more detail in Chapter 1, section 1.5.3.5 (page 27).

7.2.5 Avoiding unintentional latches in combinational logic procedures

A common problem in RTL modeling is the inference of latch behavior in code that is intended to represent combinational logic behavior. SystemVerilog language rules require that the left-hand side of procedural assignments must be some type of variable. Net data types are not permitted on the left-hand side of procedural assignments. This requirement to use variables can lead to inadvertent latches, where pure combinational logic was intended. Latch behavior occurs when a non-clocked always procedure, meaning a combinational logic procedure, is triggered, and no assignment is made to the variables used by the procedure. The two most common ways that this can occur are:

1. A decision statement assigns to different variables in each branch, as in the following code snippet.

```
always_comb begin
    if (!mode) add_result      = a + b;
    else      subtract_result = a - b;
end
```

2. A decision statement does not execute a branch for every possible value of the decision expression. The following code snippet illustrates this problem.

```
always_comb begin
    case (opcode)
        2'b00: result = a + b;
        2'b01: result = a - b;
        2'b10: result = a * b;
    endcase
end
```

In simulation, this simple example appears to correctly model a combinational logic adder, subtractor and multiplier. If, however, the `opcode` input should have a

value of 2'b11, this example does not make any assignment to the `result` variable. Because `result` is a variable, it retains its previous value. The retention of a value behaves as a latch, even though the intent is that the `always_comb` procedural behave as combinational logic.

A latch will be inferred even when an `always_comb` procedure is used. Synthesis compilers and lint checkers, will however, report a warning or non-fatal error that a latch was inferred in an `always_comb` procedure. This warning is one of the several advantages of `always_comb` over a general `always` procedures. An `always_comb` procedure documents the design engineers intent, allowing software tools to report when the code within the procedure does not match that intent. Chapter 9 discusses the proper coding style for representing latches in RTL models, and how to avoid unintentional latches when combinational logic is intended.

7.3 Using functions to represent combinational logic

Functions, when coded correctly, behave and synthesize as combinational logic. Chapter 6, section 6.6 (page 243), discusses defining functions to encapsulate code that can be used from multiple places.

Best Practice Guideline 7-7

Always declare functions used in RTL models as `automatic`.

In order to represent combinational logic behavior, a new function return value must be calculated each time the function is called. If a static function is called, and no return value is assigned, the static function will implicitly return the value of its previous call. This is the behavior of latched logic, not combinational logic. This coding error can be avoided by declaring all functions used in RTL models as automatic functions, as discussed in Chapter 6, section 6.6.1 (page 243).

Example 7-2 defines a function that calculates a multiplication operation using the Russian Peasant Multiplication algorithm (a series of add and shift operations). The function is defined in a package, making the multiplier available to any module.

SystemVerilog infers a variable that is the same name and data type as the function. The code in Example 7-2 takes advantage of this. The name of the function, `multiply_f`, is used as a temporary variable to hold the intermediate calculations within the for-loop. The final value stored in the function name becomes the function return value when the function exits.

Figure 7-2 shows the results from synthesizing this function, along with the module that calls the function from a continuous assignment statement.

Example 7-2: Function that defines an algorithmic multiply operation

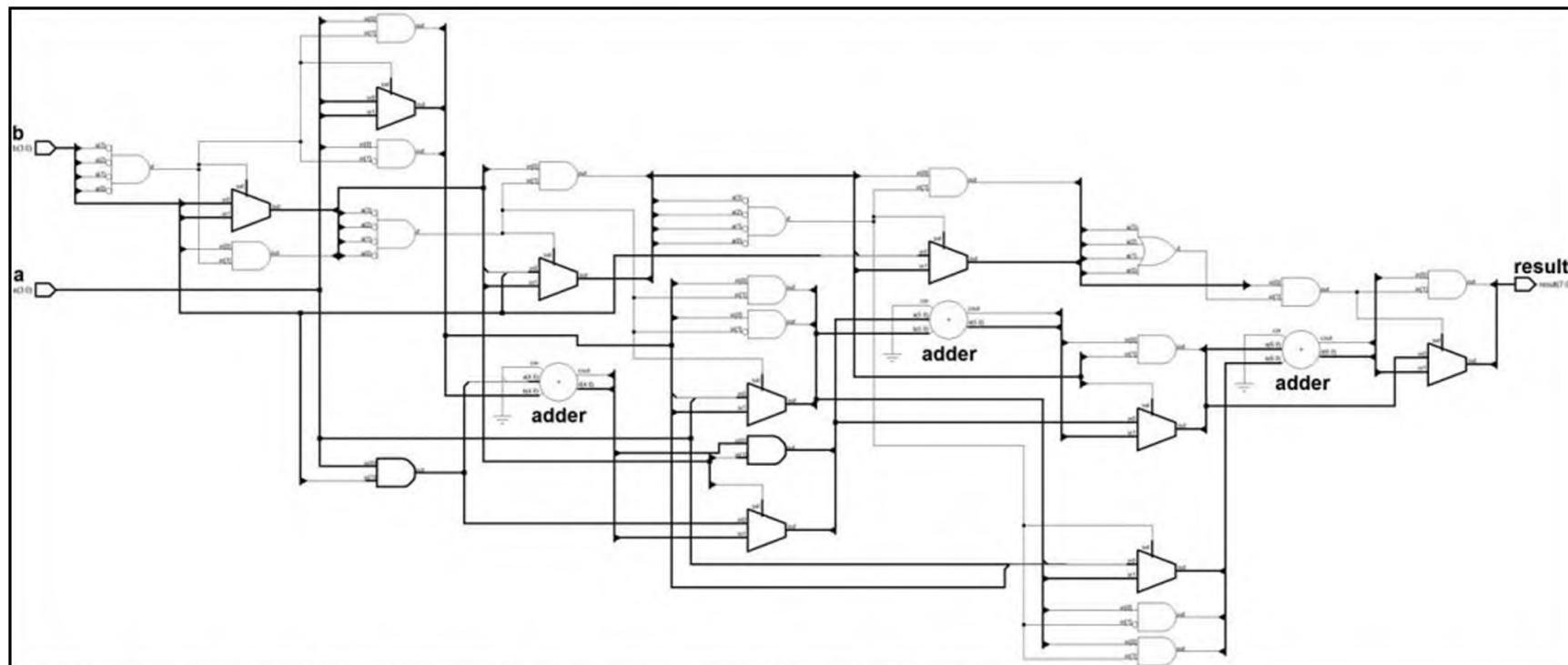
```

package definitions_pkg;
  // Russian Peasant Multiplication Algorithm
  function automatic [7:0] multiply_f([7:0] a, b);
    multiply_f = 0;
    for (int i=0; i<=3; i++) begin
      if (b == 0) continue; // all done, finish looping
      else begin
        if (b & 1) multiply_f += a; // function name is a var
        a <<= 1; // multiply by 2 by shifting left 1 time
        b >>= 1; // divide by 2 by shifting right 1 time
      end
    end
  endfunction
endpackage: definitions_pkg

module algorithmic_multiplier
  import definitions_pkg::*;
  (input logic [3:0] a, b,
   output logic [7:0] result
  );
  assign result = multiply_f(a, b);
endmodule: algorithmic_multiplier

```

Figure 7-2: Synthesis result for Example 7-2: Function as combinational logic



Technology independent schematic (no target ASIC or FPGA selected)

Best Practice Guideline 7-8

When possible, use SystemVerilog operators for complex operations such as multiplication, rather than using loops and other programming statements.

Example 7-2 of an algorithmic multiplier also illustrates why it is preferable to use SystemVerilog operators for complex operations such as multiply and divide. If the multiply operator (`*`) had been used in Example 7-2, synthesis compilers could map the operator to the most efficient multiplier implementation for a specific target ASIC or FPGA.

Design engineers need to exercise caution when using either arithmetic operators or algorithms to represent complex operations. RTL models are not software programs that run on general purpose computers with vast memory resources. RTL models are an abstraction of gate-level implementations. The functionality represented needs to physically fit in a target ASIC or FPGA, and temporally within a finite number of clock cycles. Chapter 5, section 5.12 (page 184) presents guidelines on using arithmetic operators, such as multiple and divide, in RTL models.

7.4 Combinational logic decision priority

SystemVerilog semantics for if-else-if decision series and case statements is that the series of choices are evaluated sequentially. Only the first matching branch is executed. This behavior makes it possible to represent priority encoded logic, where one choice takes precedence over another. The following code snippet illustrates a 4-to-2 priority encoder modeled with an if-else-if decision chain, where high-order bits take precedence over lower-order bits.

```
logic [3:0] d_in;
logic [1:0] d_out;

always_comb begin
    if      (d_in[3]) d_out = 2'h3; // bit 3 is set
    else if (d_in[2]) d_out = 2'h2; // bit 2 is set
    else if (d_in[1]) d_out = 2'h1; // bit 1 is set
    else if (d_in[0]) d_out = 2'h0; // bit 0 is set
    else             d_out = 2'hX; // no bits set
end
```

This same priority encoder can also be modeled by using a case statement. (This example uses a coding style referred to as a *reverse case statement*, a coding technique discussed in more detail in Chapter 8, section 8.2.5, page 313.)

```
always_comb begin
    case (d_in) inside
        4'b1???: d_out = 2'h3; // bit 3 is set
        4'b01???: d_out = 2'h2; // bit 2 is set
        4'b001?: d_out = 2'h1; // bit 1 is set
        4'b0001: d_out = 2'h0; // bit 0 is set
        4'b0000: d_out = 2'hX; // no bits set
    endcase
end
```

The if-else-if example and the case statement example are functionally identical, and will synthesize to equivalent gate-level circuitry. The results from synthesizing these examples are shown in Chapter 6, Figures 6-3 (page 221) and 6-6 (page 227), respectively.

7.4.1 Removing unnecessary priority encoding from case decisions

The priority encoder examples above depend on the prioritized evaluation flow of if-else-if decisions and case statements. Most decisions series, however, do not depend on this simulation semantic of evaluating decision choices in the order in which they are listed. A one-hot next state decoder for a Finite State Machine (FSM) illustrates this. Each one-hot value is unique from all other values. Therefore the case items are mutually exclusive — no two case items can be true at the same time. With mutually exclusive case items, the order in which the case items are evaluated does not matter, and the prioritized nature of the case statement is irrelevant.

The following example shows a simple one-hot state machine decoder. The one-hot encoding is in the literal values of the enumerated type labels.

```
typedef enum logic [2:0] {READY = 3'b001,
                           SET    = 3'b010,
                           GO     = 3'b100} states_t;

always_comb begin
  case (current_state)
    READY   : next_state = SET;
    SET     : next_state = GO;
    GO      : next_state = READY;
    default: next_state = READY;
  endcase
end
```

Chapter 8, section 8.2 (page 299) discusses modeling Finite State Machines in more detail, and shows the full context of combinational logic state decoders.

Synthesis compilers optimize case statement priority. When translating an RTL case statement into a gate-level implementation, synthesis compilers will preserve the priority encoded evaluation when it is required, such as with the BCD examples shown earlier. When the case items are mutually exclusive, however, synthesis compilers will automatically remove the priority encoding, and create parallel logic to evaluate the case items. The parallel circuitry will be faster, and require fewer gates, than a priority-encoded circuit.

7.4.2 The unique and unique0 decision modifiers

There are rare situations where the implicit priority evaluation of a case statement is not required, but the synthesis compiler cannot statically determine that the case items are mutually exclusive for all conditions. When this occurs, synthesis compilers are

pessimistic — they will leave in the priority-encoded logic in the gate-level implementation, just in case it is needed. This situation typically occurs when either:

- The case item expressions use wildcard bits that could be any value. The **case inside** decision allows wildcard bits. Since these bits can be any value, it might be possible for the case expression to match multiple case items.
- The case item expressions use variables. Synthesis is a static compilation process, and, therefore, cannot determine if the values of variables will never overlap.

Example 7-3 is a reverse case statement one-hot decoder, where the case items are bits of a variable. (This style is discussed in Chapter 8, section 8.2.5, page 313).

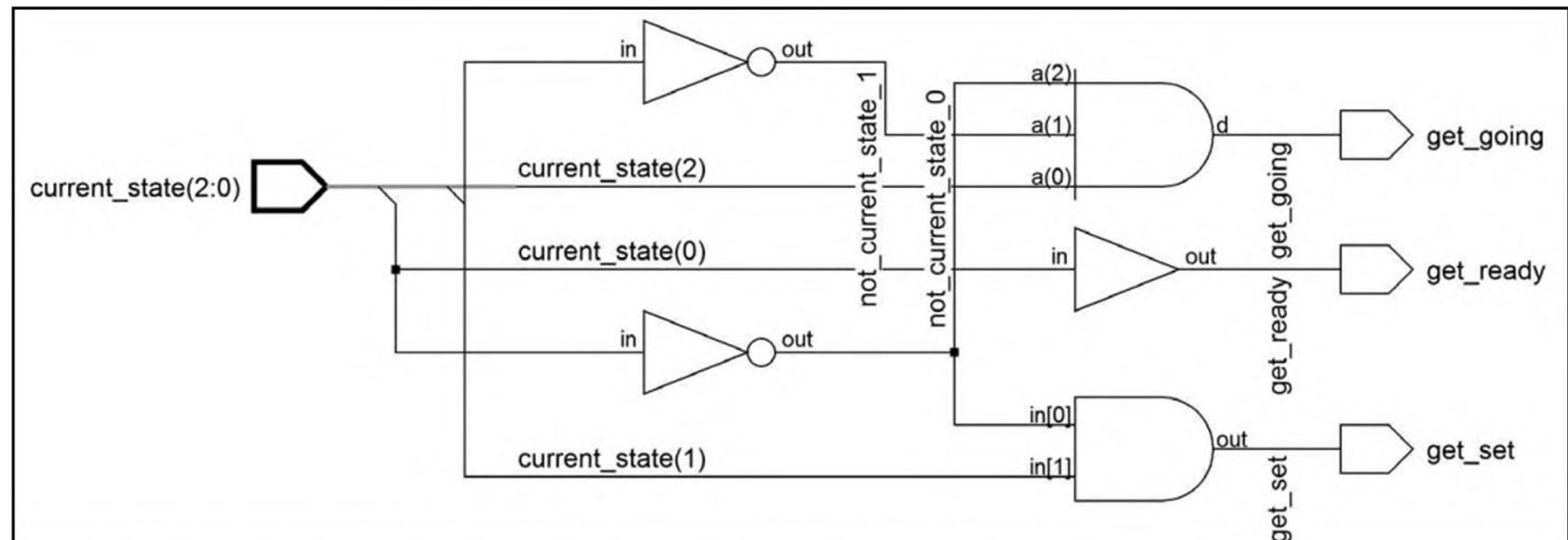
Example 7-3: State decoder with inferred priority encoded logic (*partial code*)

```
typedef enum logic [2:0] {READY= 3'b001,
                           SET    = 3'b010,
                           GO     = 3'b100} states_t;

always_comb begin
  {get_ready, get_set, get_going} = 3'b000;
  case (1'b1)
    current_state[0]: get_ready = '1;
    current_state[1]: get_set   = '1;
    current_state[2]: get_going = '1;
  endcase
end
```

The designer might know that `current_state` uses one-hot encoding, and therefore the case items are mutually exclusive. Synthesis compilers, however, cannot statically determine that the value of the `current_state` variable will only have a single bit set in all circumstances. Therefore, synthesis will implement this one-hot decoder with priority encoded logic. The case statement will not be automatically optimized for parallel evaluation. Figure 7-3 shows the results of synthesizing this reverse case statement.

Figure 7-3: Synthesis result for Example 7-3: Reverse case statement with priority



Technology independent schematic (no target ASIC or FPGA selected)

Observe the series of buffers and logic gates in order to decode even this very simple one-hot set of values. This is because the synthesis compiler is not able to recognize that the `current_state` variable will only have one-hot values, and, therefore, the case items are mutually exclusive.

The unique decision modifier. When synthesis cannot automatically detect that the case item values are mutually exclusive, the design engineer needs to inform the synthesis compiler that the case items are indeed unique from each other. This can be done by adding a `unique` decision modifier before the `case` keyword, as in the following example.

Example 7-4: State decoder with unique parallel encoded logic (*partial code*)

```

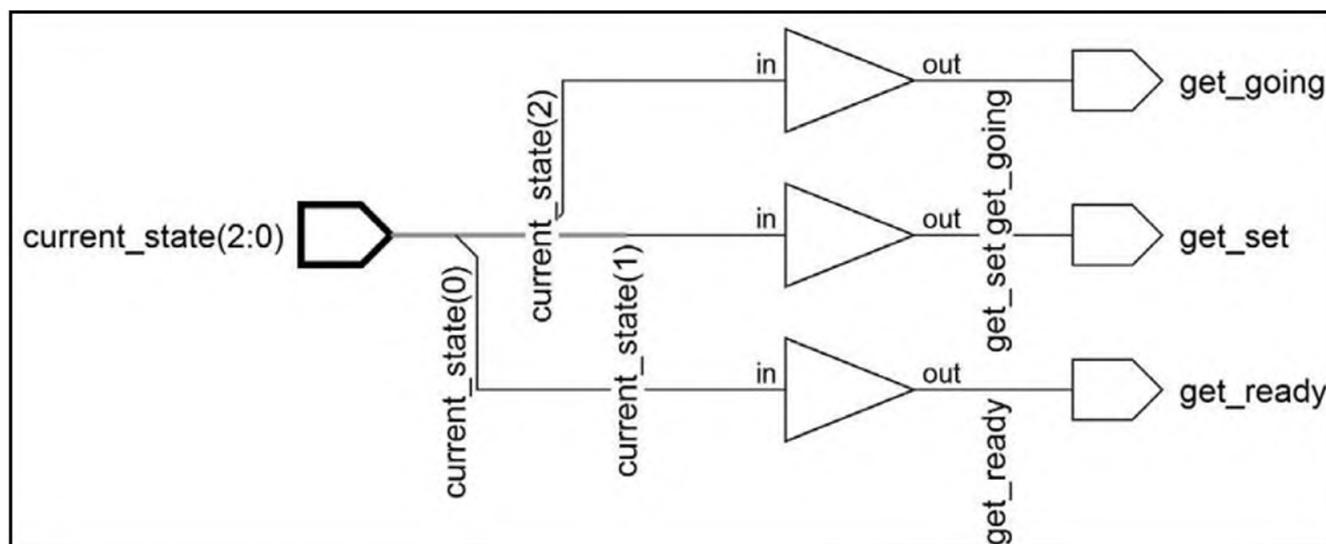
typedef enum logic [2:0] {READY= 3'b001,
                           SET    = 3'b010,
                           GO     = 3'b100} states_t;

always_comb begin
  {get_ready, get_set, get_going} = 3'b000;
  unique case (1'b1)
    current_state[0]: get_ready = '1;
    current_state[1]: get_set   = '1;
    current_state[2]: get_going = '1;
  endcase
end

```

Figure 7-4 shows the results of synthesizing this example.

Figure 7-4: Synthesis result for Example 7-4: Reverse case statement, using unique



Technology independent schematic (no target ASIC or FPGA selected)

Using `unique` instructs the synthesis compiler that the case items can be evaluated in parallel. This significantly reduced the number of gates and propagation paths for this one-hot decoder, compared to the priority implementation shown in Figure 7-3.

For synthesis, the `unique` decision modifier indicates that every case item expression will have a mutually exclusive, “unique” values, and therefore the gate-level implementation can evaluate the case items in parallel. The `unique` modifier further informs synthesis that any case expression values that were not used in the case state-

ment can be ignored. This can trigger synthesis optimizations that reduce gate counts and propagation paths, but these optimizations might not be desirable in some designs. The synthesis effects and best practice guidelines for using **unique** are discussed in Chapter 9, section 9.3.5 (page 340).

For simulation, **unique** enables run-time error checking. A violation message will be reported if:

- There are never multiple case item expressions true at the same time
- There is a branch for every case expression values that occurs.

Best Practice Guideline 7-9

Only use the **unique** decision modifier if it is certain that the synthesis logic reduction optimization effects are desirable.

Most case statements do not need, and should not use, the **unique** decision modifier. The **unique** modifier can result in synthesis gate-level optimizations that might not be desirable in many designs.

The reverse case statement coding style shown in Examples 7-3 and 7-4 is one of the few exceptions where synthesis compilers require a decision modifier in order to achieve optimal Quality of Results (QoR).

The unique0 decision modifier. SystemVerilog-2009 added a **unique0** decision modifier. Like **unique**, the **unique0** decision modifier informs synthesis compilers that every case item expression will have a mutually exclusive, “unique” values, and therefore the gate-level implementation can evaluate the case items in parallel. Unlike **unique**, however, the **unique0** modifier does not inform synthesis to ignore any case expression values that were not used in the case statement. The synthesis effects and best practice guidelines of **unique** and **unique0** are discussed in Chapter 9, section 9.3.5 (page 340).

For simulation, the **unique0** decision modifier only enables run-time error checking that there are never multiple case item expressions true at the same time. There will not be a run-time violation message if the case statement is evaluated and no case items match.

Best Practice Guideline 7-10

Use the **unique** decision modifier in RTL models. Do not use the **unique0** decision modifier. The **unique0** modifier might be recommended in the future, but, at the time this book was written, some simulators and most synthesis compilers did not support **unique0**.

7.4.3 The obsolete parallel_case synthesis pragma

SystemVerilog added the **unique** and **unique0** decision modifiers to the original Verilog language. In traditional Verilog, the only way for design engineers to tell synthesis compilers that all case items could be treated as mutually exclusive was through a **parallel_case** synthesis pragma. Synthesis pragmas are special comments that begin with the word **synthesis**. Simulators ignore the comments, but synthesis compilers act on these special pragmas.

```
case (<case_expression>) // synthesis parallel_case
```

NOTE

At the time this book was written, one commercial synthesis compiler did not recognize // synthesis as a synthesis pragma. That compiler required that pragmas start with // pragma or // synopsys.

WARNING — There is a dangerous risk with using a comment to give instructions to the synthesis compiler. Pragmas such as **parallel_case** can have a significant effect on the gate-level implementation of a case statement. *These effects are not verified in simulation!* To a simulator, the synthesis pragma is nothing more than a comment. Verification of a design at the RTL level is not verifying the same functionality as the gate-level implementation.

The **unique** and **unique0** decision modifiers replace the **parallel_case** synthesis pragma. These decision modifiers are an active part of the language, rather than a comment.

- A **unique0** case has the same effect in synthesis as the **parallel_case** pragma, plus **unique0** enables run-time simulation checking that, each time the case statement is evaluated, the case expression matches at most just one case item (it is not an error if the case expression does not match any case item).
- A **unique** case has the same effect in synthesis as a duo of synthesis pragmas, **parallel_case** and **full_case** (discussed in Chapter 9, section 9.3.7, page 350). The **unique** modifier enables run-time simulation checking that, each time the case statement is evaluated, the case expression matches exactly one case item.

Best Practice Guideline 7-11

Do not use the obsolete **parallel_case** synthesis pragma!

Synthesis compilers are very good at automatically detecting when a case statement can be implemented as a parallel decoder without affecting design functionality.

In the rare situations where the synthesis compiler needs to be told to use a parallel implementation, use the **unique** decision modifier. The **unique** decision modifier

informs synthesis compilers that the case items can be treated as mutually exclusive in the same way as the `parallel_case` pragma, but the decision modifier adds simulation run-time checking to help detect potential problems with parallel decoding of the case items during RTL simulation.

(The `unique0` decision modifier more accurately describes the `parallel_case` synthesis pragma, but this book does not recommend the use of `unique0` because it was not supported by most synthesis compilers at the time this book was written.)

7.5 Summary

This chapter has examined best coding practices for representing combinational logic in RTL models. Proper simulation behavior and synthesis Quality of Results (QoR) have been considered. The simple definition of combinational logic is that the output values are always representing a combination of the input values. If any input changes value, the output is updated to reflect this change. This chapter presented three SystemVerilog modeling constructs that can model combinational logic when used properly: continuous assignments, always procedures, and functions.

The `assign` continuous assignment statement is a simple way to model combinational logic. A continuous assignment is automatically re-evaluated any time the value of a net or variable on the right-hand side of the assignment changes. RTL continuous assignments can use any synthesizable SystemVerilog operator, but cannot use programming statements. Continuous assignments can have function calls on the right-hand side of the assignment, and the function can use programming statements.

Always procedures are an infinite loop containing a single programming statement or a begin-end group of programming statements. To model combinational logic behavior, an always procedures must start with a sensitivity list that triggers on a change of every net or variable that can affect the outputs of the combinational logic. The `always` SystemVerilog procedure is a general purpose procedure that can be used to model combinational logic, sequential logic, latched logic, verification loops, and other behaviors. A general purpose `always` procedure requires that the design engineer explicitly define an accurate sensitivity list, and follow strict coding guidelines in order for an always procedure to synthesize as combinational logic. Engineers can easily make mistakes with the always general purpose procedure that seem to simulate OK, but do not synthesize as intended. SystemVerilog added the `always_comb` procedure to the original Verilog language to address these issues. An `always_comb` procedure automatically infers a correct sensitivity list, and syntactically or semantically requires adherence to several synthesis restrictions.

Functions allow grouping programming statements together so that the group can be used from multiple places without having to duplicate code. A function whose return value is based on the inputs to the function can represent combinational logic.

This chapter also discussed the implied priority when case statements simulate. When this priority evaluation is not needed, such as with mutually exclusive case item values, synthesis compilers will remove the unnecessary priority encoded logic from the gate-level implementation of case statements. Synthesis compilers will almost always do this automatically, but there are rare situations where the design engineer needs to guide the synthesis compiler. The **unique** and **unique0** decision modifiers are the best practice modeling style for this rare situation. The obsolete **parallel_case** synthesis pragma should never be used.

* * *

Chapter 8

Modeling Sequential Logic

Abstract — Digital gate-level circuitry can be divided into two broad categories: *combinational logic*, discussed in the previous chapter, and *sequential logic*. Flip-flops — clock-triggered sequential logic — are discussed in this chapter. Latches — level-sensitive sequential logic — are discussed in the next chapter.

Sequential logic describes gate-level circuitry where the output reflects a value that has been stored by an internal state of the gates. Only certain input changes, such as a clock-edge, cause the storage to change. For D-type flip-flops, a specific edge of the clock input will change the storage of the flip-flop, but changes to the D input value do not directly change the storage. Instead, the specific clock edge causes the internal storage of the flip-flop to be updated to the value of the D input at the time of the clock edge.

RTL models of sequential logic need to reflect this gate-level behavior, meaning that the output of a block of logic must store a value over one or more clock cycles, and only update the stored value for specific input changes, but not all input changes. At the RTL level, an **always** or **always_ff** procedure is used to model this sequential behavior. This chapter examines:

- Synthesis requirements for RTL sequential logic
- The **always_ff** sequential logic RTL procedure
- Sequential logic clock-to-Q propagation and setup/hold times
- Using nonblocking assignments to model clock-to-Q propagation effect
- Synchronous and asynchronous resets
- Multiple clocks and clock domain crossing (CDC)
- Using unit delays in sequential logic RTL models
- Modeling Finite State Machines (FSMs)
- Modeling Mealy and Moore FSM architectures
- State decoders, and using **unique case** for 1-hot decoders
- Modeling memory devices such as RAMs

8.1 RTL models of flip-flops and registers

Flip-flops and *registers* are used to store information for some period of time. The terms flip-flop and register are often used synonymously, even though there can be differences in how they are loaded and reset. Flip-flops are a storage element that change the state of storage on a clock edge. A wide variety of hardware applications can be built from flip-flops, such as counters, data registers, control registers, shift registers, and state registers. Registers can be built from any type of data storage device, including flip-flops, latches and RAMs. Most hardware registers are built from flip-flops.

RTL models of clocked sequential logic flip-flops and registers are modeled with an **always** or **always_ff** procedure with a sensitivity list that uses a clock edge to trigger evaluation of the procedure. An example of an RTL flip-flop is:

```
always_ff @ (posedge clk)
    q <= d;
```

In general, RTL models are written to trigger flip-flops on the positive edge of a clock input. All ASIC and FPGA devices support flip-flops that trigger on a rising edge of the clock (the positive edge). Some ASIC or FPGA devices also support flip-flops that trigger on a falling edge of a clock. Flip-flops, and registers made from flip-flops, can either be non-resettable or resettable. The reset can be synchronous or asynchronous to the clock trigger. Some flip-flops also have an asynchronous set input.

At the gate-level of design, there are several types of flip-flops, such as: SR, D, JK and T flip-flops. RTL models can abstract from this implementation detail, and be written as generic flip-flops. In RTL modeling, the focus is on design functionality, not design implementation. It is the role of a synthesis compiler to map the abstract RTL functional description to a specific gate-level implementation. Most ASIC and FPGA devices use D-type flip-flops, so this book assumes that this will be the type of flip-flop synthesis compilers will infer from an RTL flip-flop.

8.1.1 Synthesis requirements for RTL sequential logic

Synthesis compilers will attempt to infer a flip-flop when the sensitivity list of an always procedure contains the keyword **posedge** or **negedge**. However, synthesis compilers also require additional code restrictions be met in order to infer a flip-flop.

- The procedure sensitivity list must specify which edge of the clock triggers updating the state of the flip-flop (**posedge** or **negedge**).
- The sensitivity list must specify the leading edge (**posedge** or **negedge**) of any asynchronous set or reset signals (synchronous sets or resets are not listed in the sensitivity list).
- Other than the clock, asynchronous set or asynchronous reset, the sensitivity list cannot contain any other signals, such as the D input or an enable input.

- The procedure should execute in zero simulation time. Synthesis compilers ignore # delays, and do not permit @ or **wait** time controls. An exception to this rule is the use of intra-assignment unit delays (see section 8.1.7.1, page 297).
- A variable assigned a value in a sequential logic procedure cannot be assigned a value by any other procedure or continuous assignment (multiple assignments within the same procedure are permitted).
- A variable assigned a value in a sequential logic procedure cannot have a mix of blocking and nonblocking assignments. For example, the reset branch cannot be modeled with a blocking assignment and the clocked branch modeled with a non-blocking assignment.

8.1.2 Always procedures and always_ff procedures

The general-purpose **always** procedure can be used to model any type of logic, including combinational logic, clocked sequential logic (flip-flops) and level-sensitive sequential logic (latches). In order for the general-purpose always procedure to model flip-flop behavior, the **always** keyword must be immediately followed by a sensitivity list that specifies either the **posedge** or **negedge** of a clock signal, as in:

```
always @ (posedge clk)
    q <= d;
```

Although this example is functionally accurate, the general purpose **always** procedure does not require nor enforce any of the synthesis requirements listed previously, in section 8.1.1 (page 274). The next example is syntactically legal, but will not synthesize:

```
always @ (posedge clk or enable) // not synthesizable
    if (enable) q <= d;
```

This example will compile and run in simulation with no warnings or error messages, but synthesis compilers will report an error when trying to compile the example. It does not meet the requirement that no other signals other than the clock and the leading edge of an asynchronous set or reset can be included in the sensitivity list. Careful verification of the RTL simulation would reveal that the state of the flip-flop updates its internal storage each time enable changes value, even when no clock trigger occurred. A gate-level flip-flop would not do this.

The **always_ff** procedure also requires a sensitivity list that specifies a **posedge** or **negedge** of a clock, but **always_ff** also enforces many of the synthesis requirements listed in 8.1.1 (page 274). The SystemVerilog standard requires that all software tools report an error if:

- The body of the procedure contains a #, **wait**, or @ time control delay that blocks the procedure execution until a future simulation time. An intra-assignment delay is allowed, because it does not block the procedure (see section 8.1.7.1, page 297).
- There is a call to a task in the procedure (because tasks can contain delays).

- Any other procedure, continuous assignment or input port assigns to the same variables as the `always_ff` procedure.

The IEEE standard also suggests, but does not require, that software tools check for other synthesis restrictions, such as an incorrect sensitivity list. Design engineering tools such as synthesis compilers and lint checkers (that check coding style) perform these optional checks, but most simulators do not perform additional checking on `always_ff` procedures. These errors and optional additional checking help to ensure that RTL models with sequential logic will both simulate correctly and synthesize correctly.

The `always_ff` procedure must be followed by a sensitivity list that meets synthesis requirements. The sensitivity list cannot be inferred from the body of the procedure in the way `always_comb` can infer a sensitivity list. The reason is simple. The clock signal is not named within the body of the `always_ff` procedure. The clock name, and which edge of the clock triggers the procedure, must be explicitly specified by the design engineer in the sensitivity list.

Best Practice Guideline 8-1

Use the SystemVerilog `always_ff` RTL-specific procedure to model RTL sequential logic. Do not use the general purpose `always` procedure.

The `always_ff` RTL-specific procedure enforces the coding style required by synthesis compilers in order to correctly model sequential logic behavior.

8.1.3 Sequential logic clock-to-Q propagation and setup/hold times

At the implementation level in ASICs and FPGAs, clocked sequential logic has characteristics that are unique from combinational logic. One of these characteristics is the propagation delay from when the clock input triggers to when the flip-flop output changes. This is often referred to as the *clock-to-Q delay*. A second characteristic is the setup and hold time. The *setup time* is the period of time in which the data input must be stable before a clock trigger. The *hold time* is the period of time in which data must remain stable after the clock trigger. If data should change within the setup and hold period, the value stored as the new flip-flop state will be uncertain. Under these conditions, it is also possible for a flip-flop's state to oscillate between values for a period of time before settling to a stable value. This unstable period is referred to as *metastability*.

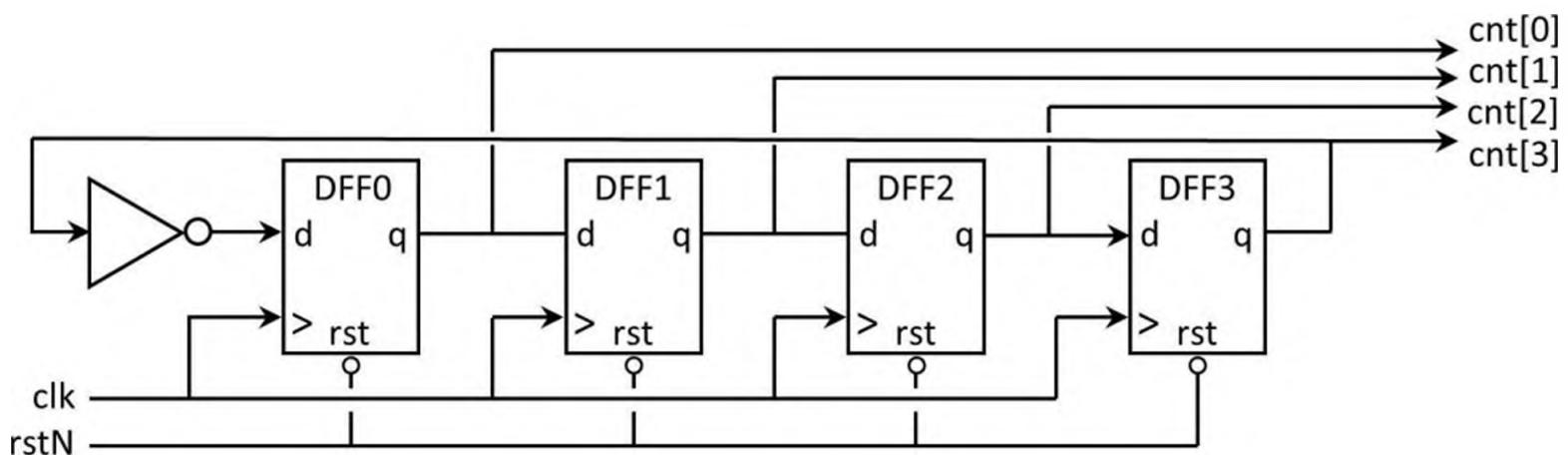
Abstract RTL models should be zero-delay models — a requirement for best synthesis Quality of Results (QoR) — which means the RTL models do not have propagation delays. The output of a flip-flop changes at the same moment of simulation time in which the clock trigger occurs, without the gate-level clock-to-Q propagation delay. As zero-delay models, abstract RTL flip-flops also do not have setup and hold times, and cannot go metastable. Nevertheless, the behavior clock-to-Q propagation

must be represented in abstract RTL models, and the RTL models need to reflect proper design techniques to avoid metastable conditions once implemented in an ASIC or FPGA.

Sequential logic clock-to-Q propagation delays. At the implementation level in ASICs and FPGAs, clocked sequential devices have a clock-to-Q propagation delay. The state, or internal storage, of a flip-flop is updated on an edge of the clock. The transition to a new state does not happen instantly. There is a small amount of time involved for the internal state to change value. During that transition time, the previous state of the flip-flop is available on the flip-flop output. When multiple flip-flops are chained together in series, this clock-to-Q propagation delay through each flip-flop creates a cascading effect through the series of flip-flops. Shift registers and counters make use of this cascade effect.

The circuit in Figure 8-1 represents a 4-bit Johnson counter, which is a shift register, with the output of the last flip-flop inverted and fed back to the input of the first flip flop.

Figure 8-1: 4-bit Johnson counter diagram



An example output from this 4-bit Johnson counter after reset is:

```

cnt[0:3] = 0000
cnt[0:3] = 1000
cnt[0:3] = 1100
cnt[0:3] = 1110
cnt[0:3] = 1111
cnt[0:3] = 0111
cnt[0:3] = 0011
cnt[0:3] = 0001
cnt[0:3] = 0000

```

The cascade effect from one flip-flop to the next is readily apparent in this output. The 0 output from the last flip-flop, DFF4, is inverted, and becomes a 1 on the D input of the first flip-flop, DFF1. On the first clock cycle, this 1 is stored into DFF1, while the old state of DFF1, a 0, is cascaded into DFF2. On the second clock cycle, the 1 on the output of DFF1 is cascaded into DFF2. On the third clock cycle, the 1 in DFF2 cascades to DFF3, and on the fourth clock cycle the 1 in DFF3 cascades into DFF4. After that fourth clock, the DFF4 output becomes 1, and the D input to DFF1 becomes 0. On

the next clock cycle, that 0 loads into DFF1, and that 0 cascades through the four flip-flops on each subsequent clock cycle.

The Johnson counter design depends on the clock-to-Q propagation delay of each flip-flop, which allows the previous state of each flip-flop in the series to be a stable D input for each subsequent stage in the series of flip-flops. It is critical that RTL models maintain this clock-to-Q propagation delay behavior, even though the RTL code is modeled with zero-delays. This all-important characteristic of flip-flop behavior is represented by the nonblocking assignment token (`<=`).

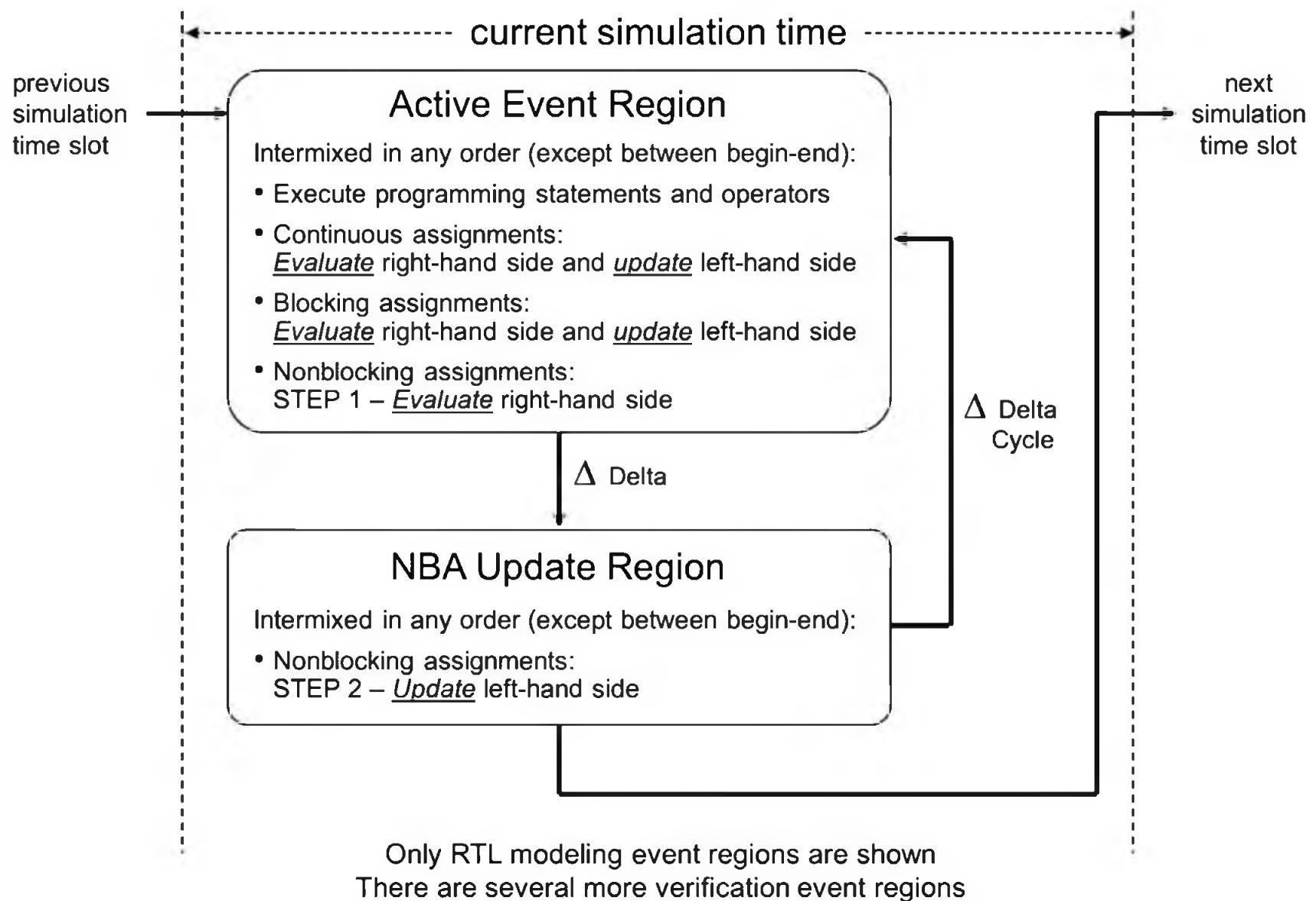
8.1.4 Using nonblocking (sequential logic) assignments

SystemVerilog has two types of assignment operators, *blocking assignments* and *nonblocking assignments*. The blocking assignment is represented with an equal sign (`=`), and is used to model combinational logic behavior, such as Boolean operations and multiplexors. The nonblocking assignment is represented with a less-than-equal sign (`<=`), and is used to model sequential logic, such as flip flops and latches.

Chapter 1, section 1.5.3 (page 23), discusses how SystemVerilog simulators schedule changes, called *events*. In brief, blocking assignments are scheduled in the Active event region, and are executed as a single event; the right-hand side of the assignment is evaluated, and the variable on the left-hand side is immediately updated.

Nonblocking assignments are scheduled as a two-step process. The right-hand side of the assignment is evaluated in the Active event region, but the change to the left-hand side is postponed, and not changed until the NBA Update region (NBA stands for nonblocking assignment). This two-step process mimics the clock-to-Q behavior of sequential logic devices. Even though the flip-flop is modeled with zero propagation delay, it behaves as if there is a delay between when the clock trigger occurs and the output of the flip-flop (the left-hand side of assignments) changes value. During the delta between the Active and NBA Update regions, the variable representing the flip-flop output still has its previous state. This allows the previous flip-flop output to cascade into the input of another flip-flop.

Figure 8-2 illustrates a simplified flow of SystemVerilog's event regions. Additional details on SystemVerilog event scheduling are discussed in Chapter 1, section 1.5.3 (page 23).

Figure 8-2: Simplified SystemVerilog event scheduling flow

Best Practice Guideline 8-2

Only use nonblocking assignments (`<=`) to assign the output variables of sequential logic blocks.

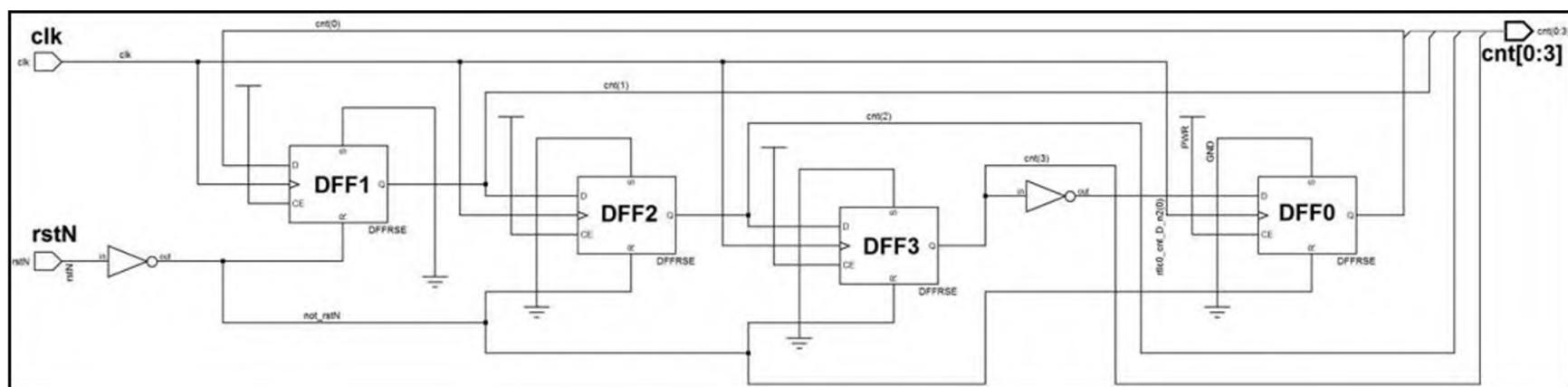
Correctly using blocking assignments and nonblocking assignments is critical for accurate simulation of zero-delay RTL models. Example 8-1 is an RTL model for the Johnson counter shown in Figure 8-1.

Example 8-1: RTL model of a 4-bit Johnson counter

```
module jcounter
(output logic [0:3] cnt,
 input logic clk, rstN
);
  always_ff @(posedge clk)
    if (!rstN) cnt <= '0; // synchronous active-low reset
    else begin           // shift the count
      cnt[0] <= ~cnt[3];
      cnt[1] <= cnt[0];
      cnt[2] <= cnt[1];
      cnt[3] <= cnt[2];
    end
endmodule: jcounter
```

Nonblocking assignments provide the abstract behavior of actual flip-flop clock-to-Q propagation delays. The right-hand side of each nonblocking assignment statement — the D input of the flip-flop — is evaluated in the simulator’s Active event region for the simulation time in which the clock triggers the procedure, before any of the left-hand sides of the assignments — the flip-flop outputs — change value. The left-hand side of each nonblocking assignment is updated after all Active events have been processed, and simulation enters the NBA Update region for that moment of time. The delta between the clock trigger in the Active region and the change in the NBA Update region represents a clock-to-Q propagation delay. Figure 8-3 shows the results of synthesizing Example 8-1. Compare this with Figure 8-1 (page 277).

Figure 8-3: Synthesis result for Example 8-1: Nonblocking assignments, J-Counter



Technology independent schematic (no target ASIC or FPGA selected)

8.1.4.1 Execution order of events within an event region

Assignments within begin-end blocks. Statements between begin-end are scheduled into an event region in the order they are listed in the begin-end. In the RTL code for the Johnson counter shown above, there is a definitive order in which the four assignment statements will be processed. That order is not representative of the gate-level implementation, however. The four flip-flops that make up this counter are clocked in parallel, not sequentially. The physical placement of the flip-flops in the ASIC or FPGA could be in any order, which does not matter because the flip-flops are clocked in parallel. It is the interconnection between the flip-flops and the clock-to-Q propagation delay (or RTL simulation delta) that creates the cascade effect of the counter, not the physical placement or RTL statement order.

When modeling sequential logic flip-flops, the order of statements within a begin-end group should be modeled so that there is no dependency on the order. The use of nonblocking assignments provides this capability. Observe that assignment statements in the following version of the Johnson counter RTL model are not in the same order as the cascade effect through the counter.

```

always_ff @(posedge clk)
  if (!rstN) cnt <= '0; // synchronous active-low reset
  else begin           // shift the count
    cnt[1] <= cnt[0];
    cnt[3] <= cnt[2];
    cnt[0] <= ~cnt[3];
    cnt[2] <= cnt[1];
  end

```

The two-step execution process of nonblocking assignments schedules the evaluation of the right-hand side for all four assignments statements in the order they are listed in the begin-end. The order does not matter, however, because all scheduled Active events are processed before any NBA Update events are processed. Therefore, every right-hand side evaluation is using the previous value (the current state) of each variable, before any new values have propagated to those variables after the delta transition to the NBA Update region.

Assignments in concurrent processes. Statements from concurrent processes, such as multiple always procedures or multiple modules, are scheduled in the Active and NBA Update event regions in an arbitrary order. The RTL designer has no control over this order, but does have control over the event region in which events are scheduled. This control is through the use of blocking or nonblocking assignments. The following example splits the Johnson counter model into four separate always procedures, which could be defined in a single module, or in separate modules connected together in a netlist.

```

always_ff @(posedge clk)
  if (!rstN) cnt <= '0; // reset
  else       cnt[0] <= ~cnt[3]; // store

always_ff @(posedge clk)
  if (!rstN) cnt <= '0; // reset
  else       cnt[2] <= cnt[1]; // store

always_ff @(posedge clk)
  if (!rstN) cnt <= '0; // reset
  else       cnt[1] <= cnt[0]; // store

always_ff @(posedge clk)
  if (!rstN) cnt <= '0; // reset
  else       cnt[3] <= cnt[2]; // store

```

A simulator can schedule the assignments from the four concurrent procedures in any order within the Active and NBA Update regions. The order does not matter, because of the two-step execution process of nonblocking assignments.

8.1.4.2 Improper use of blocking assignments in sequential flip-flop behavior

Had blocking assignments been used in any of the three preceding Johnson counter examples, each flip-flop output would immediately update to a new value. In a begin-end procedural block, this new value would become the input (the right-hand side of the assignment) of the next flip-flop assignment. Instead of a rippling effect, the input of the first flip-flop would also immediately become the input value of the subsequent flip-flop.

Example 8-2 incorrectly uses blocking assignments to model — or attempt to model — sequential logic behavior.

Example 8-2: 4-bit Johnson counter incorrectly modeled with blocking assignments

```
module jcounter_bad
(output logic [0:3] cnt,
 input logic clk, rstN
);
    always_ff @(posedge clk)
        if (!rstN) cnt <= '0; // synchronous active-low reset
        else begin           // shift the count
            cnt[0] = ~cnt[3];
            cnt[1] = cnt[0];
            cnt[2] = cnt[1];
            cnt[3] = cnt[2];
        end
endmodule: jcounter_bad
```

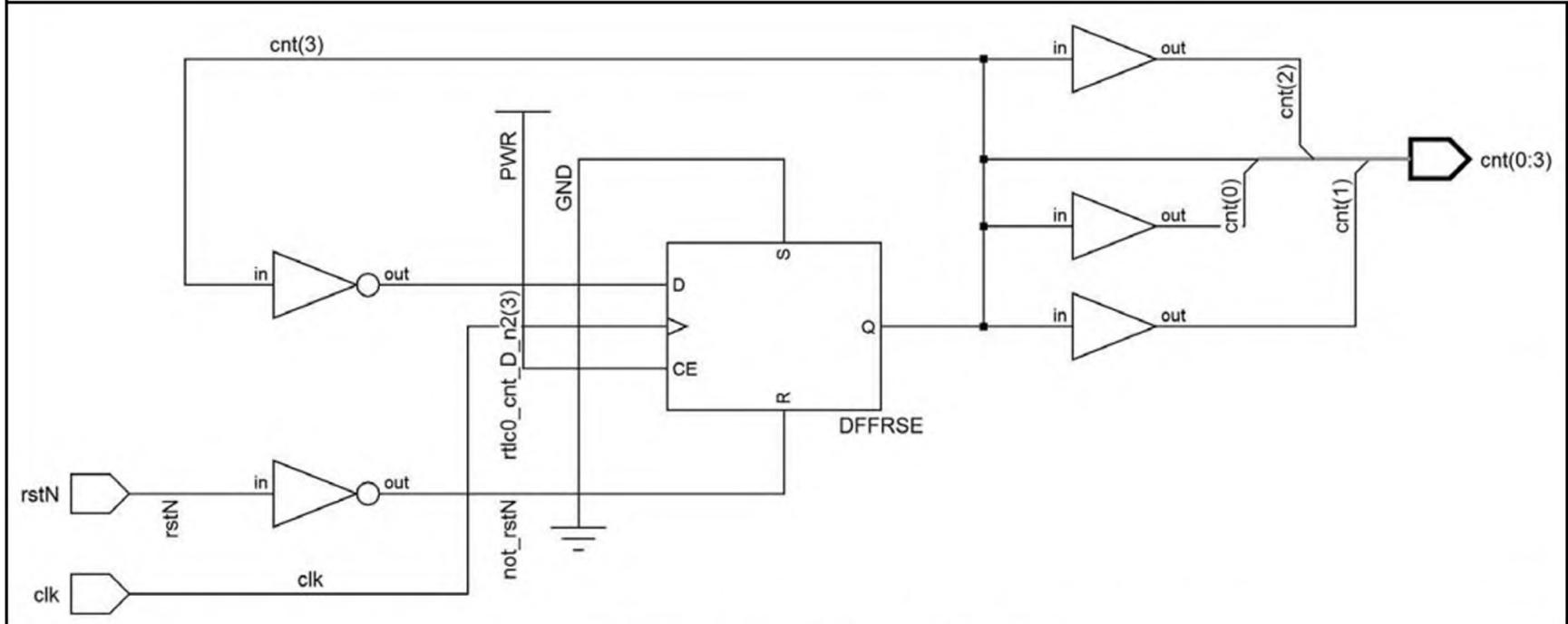
The simulation results from this example, after the counter is reset, are:

```
cnt[0:3] = 0000
cnt[0:3] = 1111
cnt[0:3] = 0000
cnt[0:3] = 1111
cnt[0:3] = 0000
```

The blocking assignments behave as combinational logic, and do not have the clock-to-Q propagation delta required to model the cascade effect of flip-flops connected in series. Synthesis will implement this example by collapsing the four assignments into a single flip-flop, with the single output going to all four bits of the cnt signal.

Figure 8-4 shows the results of synthesizing Example 8-2.

Figure 8-4: Synthesis result for Example 8-2: Blocking assignments, bad J-Counter



Technology independent schematic (no target ASIC or FPGA selected)

Synthesis compilers might not give any warning or error messages because of the incorrect usage of blocking assignments to represent sequential logic flip-flops. The synthesis compiler simply creates an implementation that matches the way the RTL model simulates. Design engineers need to understand how blocking and nonblocking assignments behave, and use them correctly. Running lint check programs on RTL models, which check code for proper modeling style, will report warnings when blocking assignments are used in procedures that are triggered on a clock edge.

8.1.4.3 Using blocking assignments for temporary variables

Best Practice Guideline 8-3

Use separate combinational logic processes to calculate intermediate values required in sequential logic procedures. Do not embed intermediate calculations inside of sequential logic procedures.

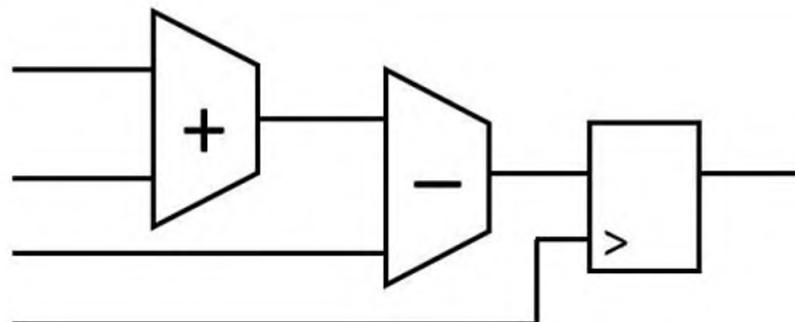
The next example violates this guideline in order to illustrate the difference in how blocking and nonblocking assignments behave.

Using blocking assignments to represent sequential logic will almost always result in simulation race conditions and risk synthesis generating hardware implementations that do not have the intended design functionality. It would seem reasonable for the SystemVerilog language to make nonblocking assignments in an **always_ff** or **always_latch** procedure a syntax requirement. If blocking assignments were illegal in these procedures, it would prevent engineers from making coding errors that might look functionally correct in simulation, but do not synthesize as expected or desired.

There is a reason blocking assignments are allowed, however. While not a recommended best-practice coding style, it is possible to mix nonblocking and blocking assignments in the same procedure. When used correctly this style will simulate correctly and synthesize into a correct implementation.

There can be times when a sequential procedure needs to calculate an intermediate value before assigning the result of that calculation to the variable representing the flip-flop. In the following diagram, the result of the adder become an input to the subtractor. Only the output of the subtractor is registered in flip-flops.

Figure 8-5: Blocking assignment to intermediate temporary variable



A mix of blocking and non blocking assignments is required in order to code this behavior as a single always procedure.

```

always_ff @(posedge clk) begin: two_steps
    logic [7:0] tmp; // local temporary variable
    tmp = a + b; // calculate tmp immediately
    out <= c - tmp; // store final result in a register
end: two_steps
    
```

The assignment to tmp is a blocking assignment, so tmp is immediately updated with a new value, and, therefore, the next line of code uses the new value. The effect of this intermediate blocking assignment is that, even though the procedure is triggered on a clock, the adder is combinational logic that is an input to the subtractor on the next line. The output of the subtractor is stored in a register.

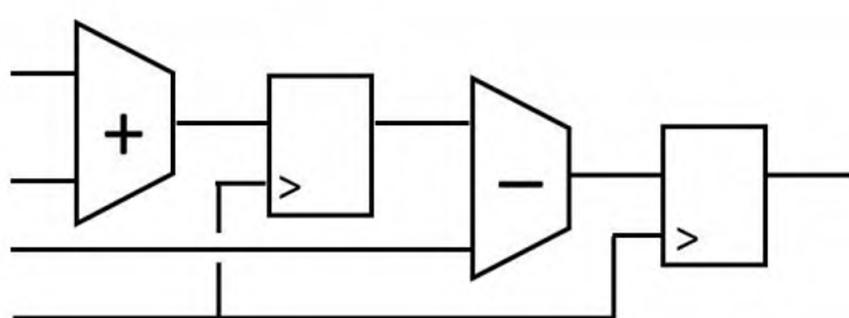
On the other hand, if the intermediate tmp variable were to be assigned by using a nonblocking assignment, as follows,

```

always_ff @(posedge clk) begin: two_stages
    logic [7:0] tmp; // local temporary variable
    tmp <= a + b; // store tmp result in a register
    out <= c - tmp; // store final result in a register
end: two_stages
    
```

then tmp would not be updated until the NBA Update region. The subtraction operation in next line of code would always use the previous state of tmp. This simulates and synthesizes as a pipeline, as in Figure 8-6.

Figure 8-6: Nonblocking assignment to intermediate temporary variable



Another example of using temporary variables and blocking assignments in a sequential logic block is for the calculation of values within a loop. The following example uses a repeat-loop to calculate an exponential result, where the exponent, E, is a parameter that can be configured to be a different value for different instantiations of the module containing this code.

```
always_ff @ (posedge clk) begin: power_loop
    logic [7:0] q_temp; // temp variable for inside the loop
    if (E == 0)
        q <= 1; // nonblocking sequential assign
    else begin
        q_temp = d; // blocking combinational assign
        repeat (E-1) begin
            q_temp = q_temp * d; // blocking combinational assign
        end
        q <= q_temp; // nonblocking sequential assign
    end
end: power_loop
```

Chapter 6, section 6.3.1.3, example 6-9 (page 234) shows the complete code for this example, along with how the example synthesizes.

NOTE

Temporary variables assigned with a blocking assignment and used within a sequential logic always procedure cannot be used outside of the procedure.

A simulation race condition could occur if the value of a temporary variable that is assigned by a blocking assignment in a sequential logic block is read from outside of the sequential block. A race condition occurs whenever a variable is assigned by a blocking assignment on a clock edge, and a concurrent sequential always procedure reads the value on the same clock edge. The change to the variable value and the reading of the variable are both active events, which could be scheduled in any order by a simulator. Therefore, the procedure that reads the value of the variable might see the value before it is changed or after it is changed on that clock edge.

Best Practice Guideline 8-4

Declare temporary variables that are used in a sequential logic block as local variables within the block.

A local variable declared within a procedural block cannot be read from out side of the procedure. This will prevent potential simulation race conditions.

The preceding examples follow this guideline. The temporary variable is declared as a local variable within the procedure (see Chapter 6, section 6.1.2, page 215).

8.1.5 Synchronous and asynchronous resets

At the implementation level, actual flip-flops can be non-resettable (no reset input), or can have a reset input control. The reset control can be synchronous or asynchronous to the clock, and can be an active-high or active-low control. Some flip-flop devices also have a set (sometimes called a preset) input. There are advantages and disadvantages to each of these types of flip-flops. These engineering trade-offs are outside the scope of this book, which focuses on the RTL modeling styles that reflect these implementation characteristics.

NOTE

Specific target ASIC or FPGA devices might only support one type of reset.

ASICs and FPGAs can differ in which type of reset the device uses, which might impact RTL modeling style. FPGA devices, in particular, often only have flip-flops with one type of reset (perhaps synchronous, active-high). In contrast, many ASIC devices, and some FPGA devices, have both synchronous and asynchronous flip-flops available. Likewise, some devices only have flip-flops with a reset input, whereas other devices also have flip-flops with both set and reset inputs.

Best Practice Guideline 8-5

Write RTL models using a preferred type of reset, and let synthesis compilers map the reset functionality to the type of reset supported by the target ASIC or FPGA. Only write RTL models to use the same type of reset that a specific target ASIC or FPGA uses if it is necessary in order to achieve the most optimal speed and area for that specific device.

Many RTL design engineers, including the author of this book, model with a preferred style or reset without concern for what the target device supports. Synthesis compilers can map any type of reset in an RTL model to any type of reset available in a target ASIC and FPGA device. For example, if the RTL model uses active-low resets and the target device only has flip-flops with active-high resets, then a synthesis compiler will add extra gate-level logic invert the reset used in the RTL model. If the RTL model uses synchronous resets, and the target device only has flip-flops with asynchronous resets, then a synthesis compiler will add extra gate-level logic external to the asynchronous flip-flop to reset it synchronous to the clock. Modern ASICs and FPGAs have ample speed and capacity. A fully functional design can be obtained without having to worry about whether the synthesis process had to add some extra logic to map the RTL style reset to the target device's type of flip-flop.

Most of the examples in this book are modeled with active-low asynchronous resets, though some of the smaller code snippets use active-low synchronous resets.

8.1.5.1 Non-resettable RTL flip-flop models

A flip-flop with no reset input can only be controlled through the data input and clock. An RTL model transfers the data input to the flip-flop output every time the procedure triggers. There is no if-else condition that potentially assigns a value other than the data input. An example RTL model of a non-resettable flip-flop is:

```
always_ff @ (posedge clk)
    q <= d;
```

When synthesis implements this RTL functionality in a specific ASIC or FPGA target, a flip-flop type available in that device library will be selected. This might be a flip-flop with no reset or set inputs, a flip-flop with only a reset input that is tied off to which ever value makes it inactive, or a flip-flop with both set and reset inputs that are tied off to be inactive.

8.1.5.2 Synchronous reset RTL flip-flop models

A flip-flop with synchronous reset has a reset control input, but that input is only sampled when the clock input is triggered. The RTL model contains an **if** condition that assigns a value when reset is active, or branches to the **else** statement when reset is inactive. The reset signal is not part of the always procedure's sensitivity list. The sensitivity list only contains the clock edge that triggers the flip-flop. Therefore, the programming statements in the always procedure are only evaluated when a clock trigger occurs.

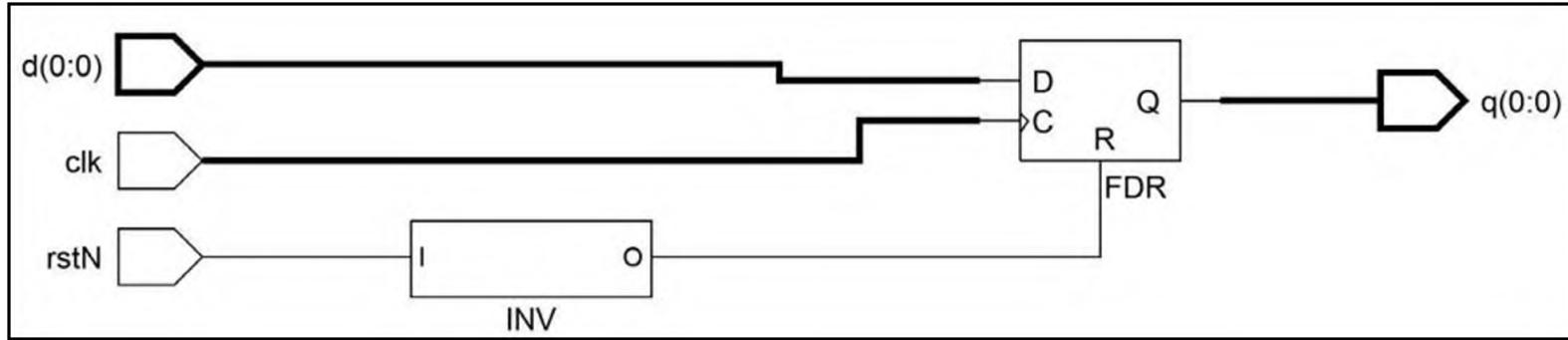
The following example illustrates a flip-flop with an active-low synchronous reset:

```
always_ff @ (posedge clk)
    if (!rstN) q <= '0; // synchronous active-low reset
    else        q <= d;
```

When synthesis maps this RTL functionality to a specific ASIC or FPGA target, a flip-flop with synchronous reset will be selected if one is available. If the target device has active-high flip-flops, synthesis will add an inverter to the `rstN` signal. If synchronous reset flip-flops are not available in the target device, the synchronous reset signal will be ANDed with the data input to the flip-flop. Any time the reset is active, a 0 will be clocked into the flip-flop, which gives the functionality of a reset that is synchronous to the clock. If the target device flip-flop also has asynchronous reset or set inputs, they will be tied off to be inactive.

Figure 8-8 shows the results of synthesis targeting the generic flip-flop to a device that has synchronous reset flip-flops, a Xilinx Virtex®-6 FPGA. The flip-flop reset is active-high, so the `rstN` input is inverted.

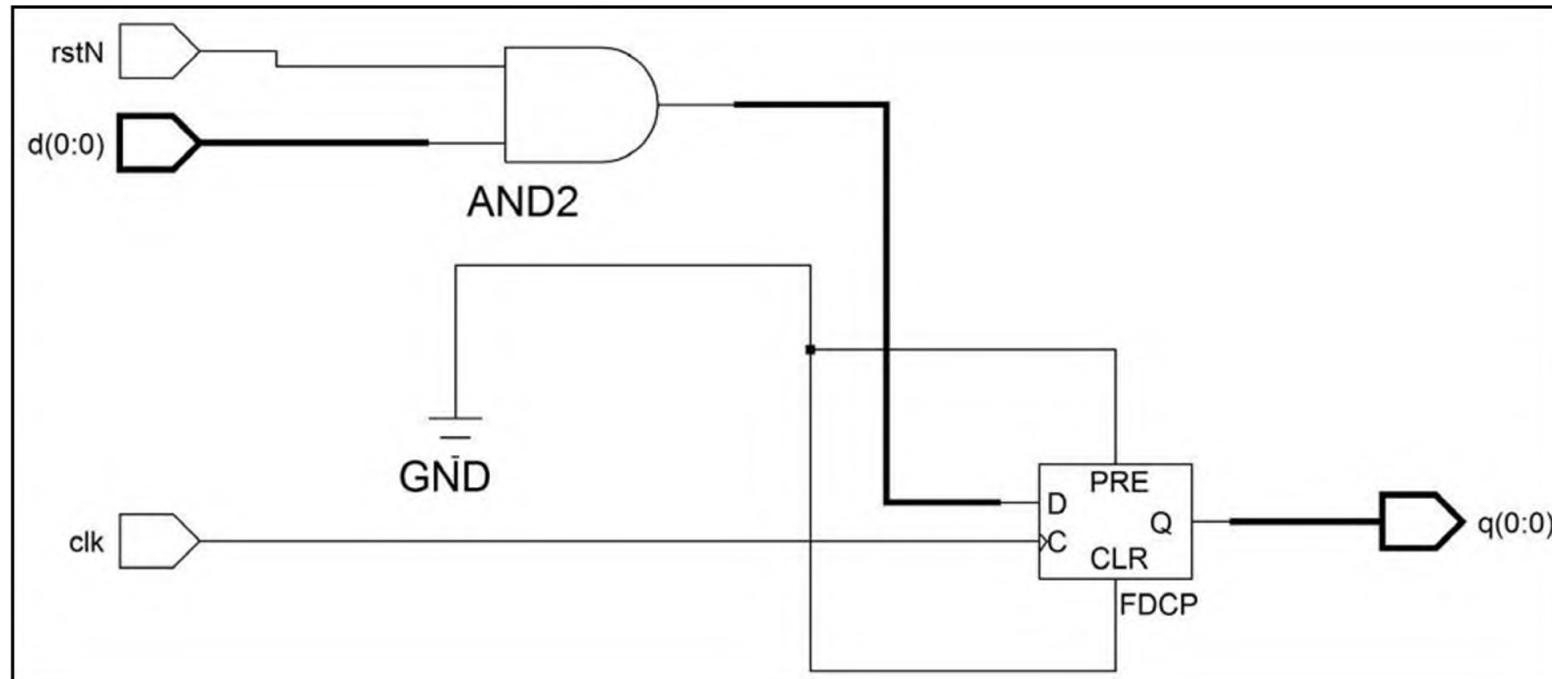
Figure 8-7: Synthesis result: Async reset DFF mapped to Xilinx Virtex®-6 FPGA



*Technology specific schematic generated by Mentor Precision Synthesis™ compiler

Figure 8-8 shows the results of targeting the same generic flip-flop to a device that does not have synchronous reset flip-flops, a Xilinx CoolRunner™-II CPLD. The flip-flop's asynchronous active-high CLR and PRE inputs are not used.

Figure 8-8: Synthesis result: Async reset mapped to Xilinx CoolRunner™-II CPLD



*Technology specific schematic generated by Mentor Precision Synthesis™ compiler

8.1.5.3 Asynchronous reset RTL flip-flop models

A flip-flop with an asynchronous reset has a reset control input that will change the flip-flop state the moment reset goes active, regardless of the clock. The RTL model sensitivity list contains both the clock edge that triggers the flip-flop and the leading edge of the reset signal. Because the leading edge of reset is in the sensitivity list, the always procedure will trigger whenever reset goes active, without having to wait for the clock input to change. The RTL model contains an **if** condition that assigns a value when reset is active, or branches to the **else** statement when reset is inactive.

The following example models a flip-flop with an active-low asynchronous reset:

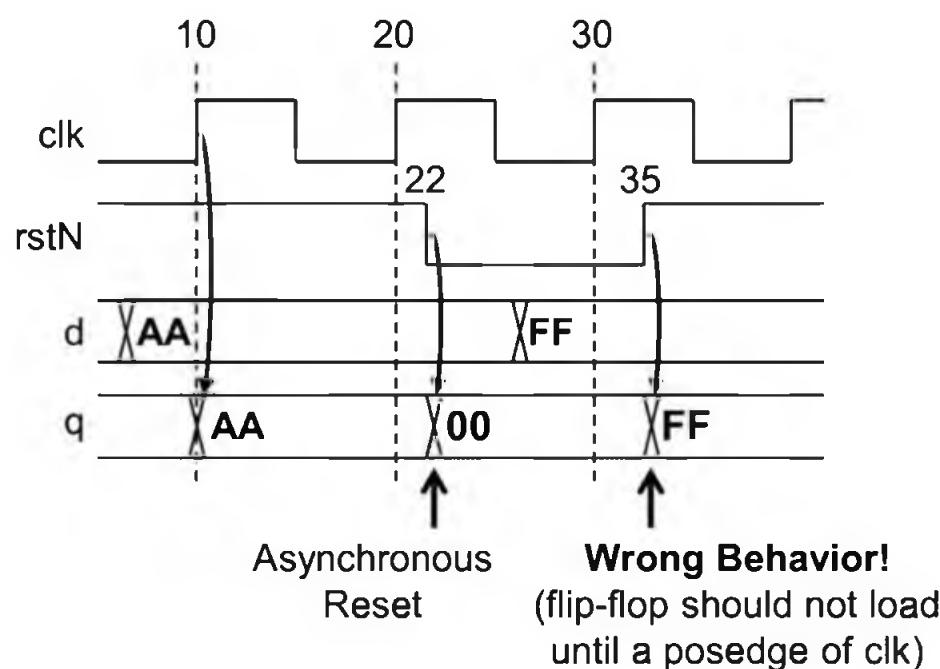
```
always_ff @(posedge clk or negedge rstN)
  if (!rstN) q <= '0; // asynchronous active-low reset
  else       q <= d;
```

Ignoring the trailing edge of an asynchronous reset. A synthesis requirement for always procedures is that, if **posedge** or **negedge** is used for one signal in a sensitivity list, then an edge must be specified for all signals in this list. The following code snippet is not synthesizable because **rstN** is not qualified with **posedge** or **negedge**:

```
always @(posedge clk or rstN) // incorrect sensitivity
  if (!rstN) q <= '0; // asynchronous active-low reset
  else      q <= d;
```

Correct simulation behavior also requires that only the leading edge of an asynchronous reset be listed in the always procedure sensitivity list. The sensitivity list controls when the programming statements in the always procedure are executed. For an asynchronous reset, the sensitivity list needs to trigger when the reset goes active in order to immediately reset the flip-flop. If, however, the always procedure also triggered when the reset goes back to its inactive state, the **if** test for reset being active would evaluate as false, and the **else** branch would be executed. It would appear as though a clock event had occurred when there was no clock edge. Figure 8-9 shows a waveform illustrating the incorrect simulation behavior for the code snippet above.

Figure 8-9: Waveform showing result of incorrectly modeled asynchronous reset



Both simulation and synthesis require that only the leading edge of an asynchronous reset be included in the sensitive list — simulation requires this for proper asynchronous reset behavior, and synthesis requires it syntactically. The SystemVerilog syntax does not enforce this restriction. It is an RTL coding style that design engineers must follow. Lint checkers can check that this coding style is being followed.

Best Practice Guideline 8-6

Be consistent in the use of active-high or active-low resets. Use a consistent naming convention for active-high and active-low control signals.

Although either active-high or active-low resets can be used, all RTL models in a project should be consistent. A mix of reset polarity can lead to code that is difficult to understand, maintain and reuse.

This book uses the convention of adding a capital “N” to the end of the names of active-low signals. Another common convention is to append an “_n” to active-low signal names.

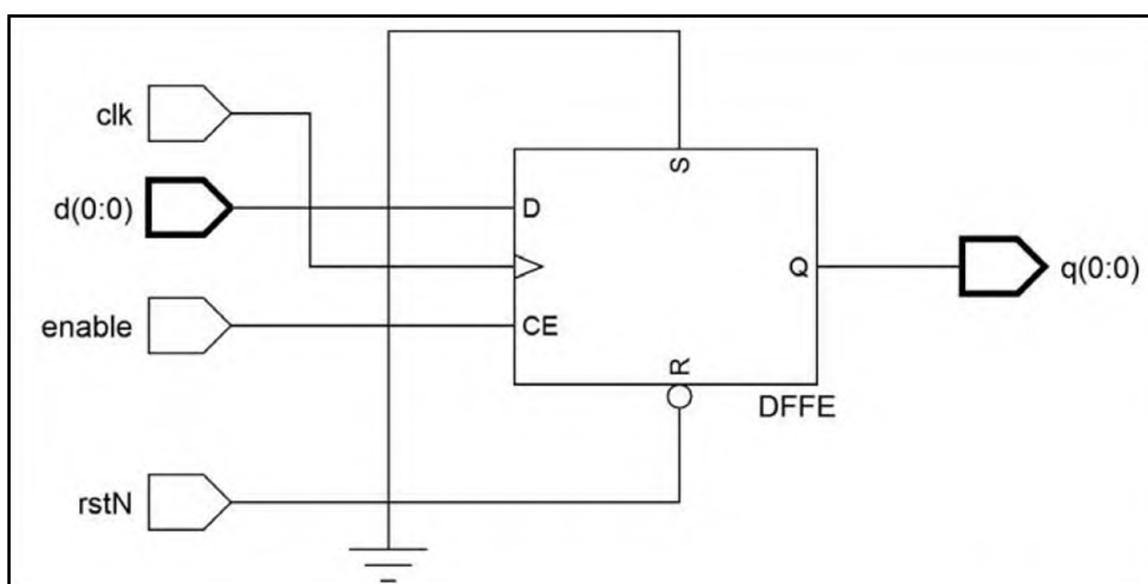
8.1.5.4 Chip-enable flip-flops

A chip-enabled flip-flop, also referred to as a load-enable or data-enable flip-flop, has an additional input that enables the flip-flop. When a clock trigger occurs, a new value is only stored in the flip-flop if the enable input is active. If the enable is inactive, the flip-flop retains its previous state.

The RTL code for a chip-enable flip-flop is similar to a regular flip-flop, but with an additional **if** condition. The enable signal is not added to the sensitivity list. Its value is only sampled synchronously on a clock edge. The following example shows the code for a chip-enable flip-flop, and Figure 8-10 shows the synthesis results.

```
always_ff @(posedge clk or negedge rstN)
  if      (!rstN) q <= '0; // asynchronous active-low reset
  else if (enable) q <= d; // store d input if enabled
```

Figure 8-10: Synthesis result for a chip-enable flip-flop

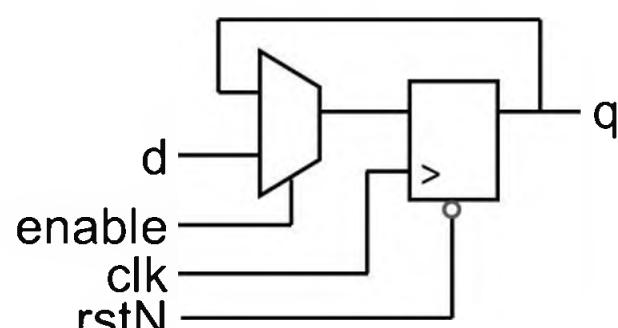


Technology independent schematic (no target ASIC or FPGA selected)

Some ASIC and FPGA devices have pre-defined and optimized chip-enable flip-flops. Synthesis compilers can translate the RTL model of a chip-enable flip-flop to these pre-defined components, if available.

If the target ASIC or FPGA does not have a chip-enable flip-flop, then synthesis can implement the chip-enable behavior by adding functionality outside of a flip-flop. This might be done by adding a multiplexor on the data input that selects either the new data value or the output of the flip-flop, as illustrated in Figure 8-11.

Figure 8-11: External logic to create the functionality of a chip-enable flip-flop



Observe that the clock to the flip-flop is always present. A chip-enable flip-flop gates the data input to the flop-flop, not the clock input.

8.1.5.5 Asynchronous set-reset flip-flops

Some sequential devices have both reset and set control inputs. An if-else-if decision series is used to model this behavior, as in the following example.

```
always_ff @ (posedge clk or negedge rstN or negedge setN)
  if      (!rstN) q <= '0; // reset (active low)
  else if (!setN) q <= '1; // set (active low)
  else          q <= d;   // clock
```

An if-else-if statement gives priority to the first input tested. In the example above, the reset takes priority of the set input, should both be active at the same time.

NOTE

The priority given to the set or the reset input in an RTL model should match a specific target ASIC or FPGA device. Some devices give priority to the reset input, whereas other devices give priority to the set input.

Best Practice Guideline 8-7

Model RTL flip-flops with just a reset input or a set input in order to achieve best synthesis Quality of Results (QoR). Only model set/reset flip-flops if needed for the functionality of the design.

If a set/reset flip-flop behavior is required, write the RTL model priority for set versus reset to match the priority of the specific target device in which the design will be implemented.

Since not all target devices have the same set/reset priority, it is difficult to write set/reset flip-flop RTL models that will synthesize optimally for all target devices. If the target device does not have a set/reset flip-flop with the same priority as the RTL model, synthesis compilers can add additional logic outside of the flip-flop to make it match the RTL model. This additional logic can affect device timing, however, and cause race conditions with other parts of a design when coming out of a reset state.

Set/reset flip-flops also have more exacting setup and hold times for these inputs than a flip-flop that only has a set or reset input, but not both. Even when the priority of the set and reset in the RTL model matches the priority of the target device, designers need to be careful that the design can meet these setup and hold requirements.

Simulation glitch with set/reset flip-flops. The example shown above of a set-reset flip-flop RTL model is functionally correct, and will synthesize correctly. There is, however, a potential simulation glitch with this code. The glitch occurs if `setN` and `rstN` control inputs are both active at the same time, and then `rstN` becomes inactive. In this example, `rstN` takes priority, and the flip-flop properly resets. When the

`rstN` input becomes inactive, the `setN` input should take over, and the flip-flop should switch to its set state. The glitch in simulation is that the RTL model is only sensitive to the leading edge of `rstN` — a requirement of synthesis compilers — and is not sensitive to the trailing edge of `rstN`. When `rstN` becomes inactive, the sensitivity list will not trigger, and therefore miss setting the flip-flop, even though `setN` is still active. This glitch will only last until the next positive edge of clock, which will trigger the always procedure, and cause the procedure to be reevaluated.

The solution to prevent this simulation glitch is to add the trailing edge of reset to the sensitivity list. However, just adding the trailing edge of reset would lead to the same problem described earlier, where the trailing edge of reset could act as a clock (see section 8.1.5.3, page 288). Therefore, the inactive level of the reset input needs to be ANDed with the active level of the set input, and the sensitivity list will trigger when that result becomes true.

The revised sensitivity list to prevent a simulation glitch is:

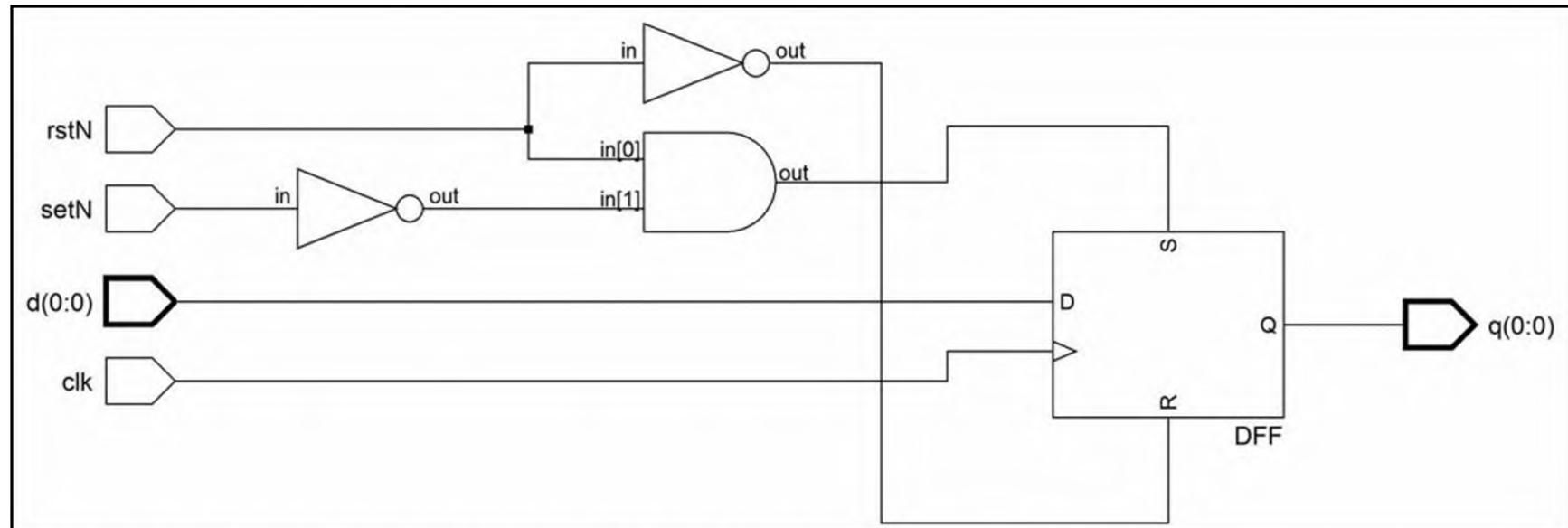
```
always_ff @(
    posedge clk
    or negedge rstN
    or negedge setN
    or posedge (rstN & ~setN) // not synthesizable
)
if (!rstN) q <= '0; // reset (active low)
else if (!setN) q <= '1; // set (active low)
else q <= d; // clock
```

Triggering on the result of an expression is legal in the SystemVerilog language, but is not permitted by synthesis compilers. The additional trigger to avoid the simulation glitch needs to be hidden from synthesis. This can be done using synthesis `translate_off` / `translate_on` pragmas. Synthesis pragmas are special comments that begin with the word `synthesis`. Simulators ignore these comments, but synthesis compilers act on them. The following snippet adds the `translate_off` / `translate_on` pragmas.

```
always_ff @(
    posedge clk
    or negedge rstN
    or negedge setN
    // synthesis translate_off
    or posedge (rstN & ~setN) // not synthesizable
    // synthesis translate_on
)
if (!rstN) q <= '0; // reset (active low)
else if (!setN) q <= '1; // set (active low)
else q <= d; // clock
```

Figure 8-12 shows the results from synthesizing this set-reset flip-flop code.

Figure 8-12: Synthesis result for an asynchronous set-reset flip-flop



Technology independent schematic (no target ASIC or FPGA selected)

Observe that the synthesis compiler added additional logic before the set input of the generic flip-flop in order to enforce that `rstN` has priority of `setN`. If the target ASIC or FPGA device has set/reset flip-flops where reset has priority over set, then this additional logic will be removed when the generic flip-flop is mapped to that target ASIC or FPGA. Otherwise, the additional logic will be left in so that the ASIC or FPGA implementation matches the RTL model.

The generic flip-flop that the synthesis compiler used before targeting a specific ASIC or FPGA has active-high set and reset inputs. Therefore, the synthesis compiler added inverters to the active-low signals in the RTL model. These inverters will be removed if the target device has active-low control inputs.

An alternative to using the `translate_off` and `translate_on` synthesis pragmas is to use conditional compilation. Most synthesis compilers have a predefined `SYNTHESIS` macro that can be used to conditionally include or exclude code that the synthesis tool compiles. To exclude the non-synthesizable line in the previous example, the code would be:

```
'ifndef SYNTHESIS // compile if not a synthesis compiler
  or posedge (rstN & ~setN) // not synthesizable
`endif // end of synthesis exclusion
```

NOTE

At the time this book was written, one commercial synthesis compiler did not recognize `// synthesis` as a synthesis pragma. That compiler required that pragmas start with `// pragma` or `// synopsys`.

8.1.5.6 Asynchronous reset metastability synchronizers

At the implementation level of ASICs and FPGAs, asynchronous reset flip-flops have a reset recovery time before the next clock trigger can occur. A flip-flop can go metastable if the trailing edge of the reset occurs too close to the next clock trigger. A reset synchronizer should be used when an asynchronous reset is an input to a module, and the design engineer cannot control the de-assertion of reset relative to the module's clock.

As zero-delay models, RTL flip-flops do not have a problem with the trailing edge of reset occurring too close to the clock edge. RTL models, however, should include reset synchronizers wherever they might be required in the final ASIC or FPGA implementation. An example reset synchronizer is:

```
always_ff @(posedge clk or negedge rstN)
  if (!rstN) begin                      // asynchronous active-low reset
    rstN_tmp <= '0;
    rstN_synced <= '0;
  end
  else begin
    rstN_tmp <= '1;                      // begin end of reset
    rstN_synced <= rstN_tmp;           // stabilize reset
  end
```

A full discussion of proper design and usage of reset synchronizers is an engineering topic, and is outside the scope of this book. This book focuses on properly modeling synchronizer functionality, once the engineering choices have been made. Appendix D lists some additional resources on the topic of resets, metastability, and synchronization.

8.1.5.7 Flip-flop power-up values (FPGA-specific)

Some FPGA devices can be programmed so that flip-flops will power-up to a reset or set value when power is turned on. Specifying power-up values can eliminate the need to reset some types of digital circuits. This allows flip-flops with no reset input to use, which can simplify the internal routing and congestion of the FPGA.

FPGA synthesis compilers support modeling power-up flip-flops by specifying an initial value for the output variable of the flip-flop. For example:

```
logic [3:0] q = '1; // power-up with all bits set

always_ff @(posedge clk)
  q <= d;
```

FPGA synthesis compilers might also support using an **initial** procedure to specify the initial (power-up) value of variables.

NOTE

Using initial variable values to model flip-flop power-up is specific to FPGA devices. ASIC devices do not support this capability.

Synthesis compilers might require special options to enable the use of initial values in variable declarations. Refer to the documentation of the specific compiler.

8.1.6 Multiple clocks and clock domain crossing (CDC)

It is common for a design to use more than one clock, with some parts of a design running at a faster clock speed than other parts of the design. *Clock Domain Crossing* (CDC) occurs when data or control signals that are an output of registers that trigger on one clock are stored in registers that trigger on a different clock.

Best Practice Guideline 8-8

Multiple clock designs should be partitioned into multiple modules, so that each module only uses a single clock.

Synthesis compilers, timing analyzers and Clock Domain Crossing (CDC) analysis tools are more effective when all sequential logic in a module uses the same clock.

When data moves from one clock domain to another, care must be taken to avoid metastability problems. A metastable condition can occur when the data input to a flip-flop changes too close to the clock trigger. The flip-flop's setup time is the amount of time before the clock edge in which the inputs must be stable. The hold time is the amount of time after the clock edge in which the inputs must remain stable.

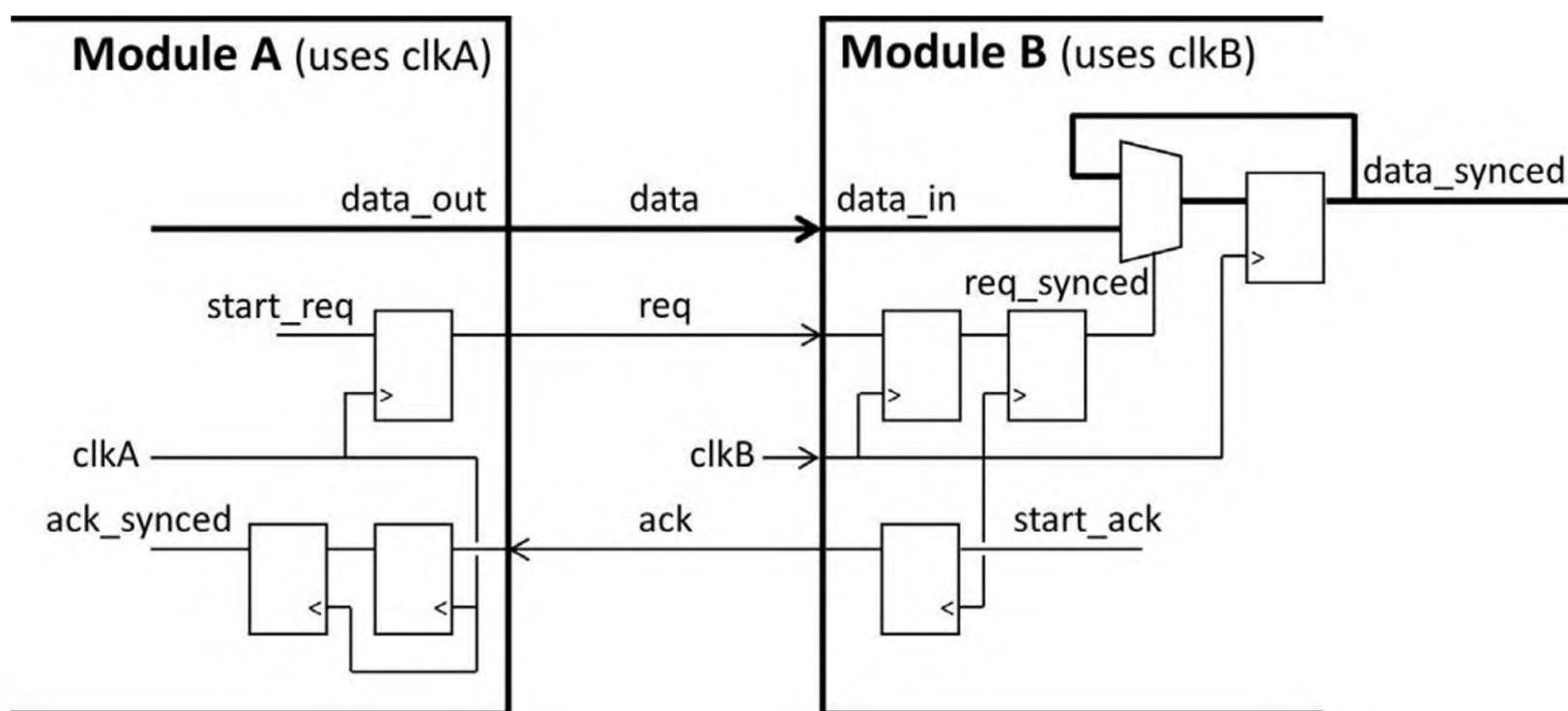
Setup and hold violations are most likely to occur on signals that cross clock domains, as the data transfers from the output of one module to an input of another module that uses a different clock. In multiple clock designs, these modules could be, and often are, running at different frequencies. There is a risk that inputs to a module that originated in a different clock domain could change too close to a clock edge in the current module's clock domain, resulting in a metastable condition. To avoid the risk of metastability, synchronizer circuits need to be added to any input ports where that input originated in a different clock domain.

One common way to pass data vectors from one clock domain to another domain is to use request and acknowledge handshake control signals. The module sending out the data issues a request to the receiving module, requesting that the receiving module read the data bus. The request signal originates in the clock domain of the sending module, and could potentially arrive close to a clock edge of the receiving module's clock domain. To avoid the risk of metastability, the receiving module passes the

incoming request through a clock synchronizer, before registering the incoming data. After the receiving module has registered the data, it sends out an acknowledgement to the sending module, which the sending module synchronizes to its clock domain. The sending module holds the data stable until the acknowledge handshake is received and synchronized.

Single-bit CDC synchronizers are most often implemented with a two-stage shift register. Figure 8-13 shows a typical synchronizer circuit.

Figure 8-13: Two flip-flop clock synchronizer for 1-bit control signals



An RTL flip-flop is modeled with zero-delays, and has no setup and hold times. Signals that cross clock domains will appear to always work in RTL models, even when there is no CDC synchronization. Nonetheless, the RTL models should include the synchronizer circuits that will be needed in the gate-level implementation of the RTL model. An example of a 1-bit control line synchronizer is:

```
always_ff @ (posedge clk or negedge rstN)
  if (!rstN) begin
    req_tmp <= '0;
    req_synced <= '0;
  end
  else begin
    req_tmp <= req; // register req input
    req_synced <= req_tmp; // stabilize req
  end
```

ASIC and FPGA devices might have optimized clock synchronizers in their target libraries. Synthesis compilers will recognize the RTL clock synchronizer behavior and map this behavior to an appropriate target component, if available.

The proper design of clock domain crossing synchronizers is an engineering topic, and is outside the scope of this book. Appendix D lists some additional resources on the topic of clock-domain crossing, metastability, and synchronization.

8.1.7 Additional RTL sequential logic modeling considerations

8.1.7.1 Adding unit delays to sequential logic RTL models

RTL models are zero-delay models, meaning they do not have any propagation delay as a value propagates through a block of functionality or is registered into a flip-flop. RTL models are an abstraction from the physical characteristics of the gates that make up an ASIC or FPGA. Abstract RTL models do not represent many of the details of actual gate-level implementation, including the propagation delays inherent in physical logic gates and flip-flops.

SystemVerilog's nonblocking assignment (`<=`) has a zero-delay delta that represents the clock-to-Q propagation delay of a physical flip-flop. Unlike a real propagation delay, however, the nonblocking delta cannot be seen in a waveform display. In a waveform, it appears that the state of a flip-flop changes instantaneously, when the clock triggers the flip-flop. Any combinational logic that the flip-flop output propagates through is also zero-delay, and the output of that combinational logic appears to immediately arrive at the input to the next register in the logic flow. The invisible delta delay of nonblocking assignments can make it more difficult to debug a complex design. It can be hard to see in a waveform display the cause and effect of each clock in a pipelined data path, where the output of one register passes through combinational logic and instantaneously becomes the input of the next pipeline stage.

SystemVerilog has a special form of a delay, called an intra-assignment delay, that can be used with nonblocking assignments. An intra-assignment delay is specified on the right-hand side of an assignment, whereas a blocking delay is on the left-hand side of the assignment. An example of each type of delay is:

```
#1ns q1 <= d1; // 1 nanosecond blocking delay  
q2 <= #1ns d2; // 1 nanosecond intra-assignment delay
```

An intra-assignment delay does not delay, or block, the evaluation of statements in an always procedure. The right-hand side of a nonblocking assignment is evaluated immediately, and the always procedure still executes in zero-time — a requirement of synthesis compilers. Instead of blocking the always procedure, an intra-assignment delay adds a measurable delta between the evaluation of the right-hand side of the assignment and the update to the left-hand side of the assignment. The clock-to-Q propagation effect of the nonblocking assignment can now be seen in a waveform display, which can aid in debugging a complex RTL model.

The `always_ff` procedure permits intra-assignment delays. An example of a synthesizable RTL register with intra-assignment propagation delays is:

```
timeunit 1ns; timeprecision 1ns;  
  
always_ff @(posedge clk or negedge rstN)  
  if (!rstN) q <= #1 '0; // 1ns intra-assignment delay  
  else       q <= #1 d;  // 1ns intra-assignment delay
```

In addition to having the appearance of a clock-to-Q propagation delay of an actual flip-flop in a waveform display, the unit intra-assignment delay can also help ensure that hold-times are being met when an RTL sequential logic variable is connected to the input of a gate-level model.

NOTE

Synthesis compilers ignore intra-assignment delays. This can lead to a mismatch in the RTL simulation behavior and the actual gate-level implementation, especially if the intra-assignment delay, or a series of intra-assignment delays, are longer than a clock cycle.

Best Practice Guideline 8-9

If intra-assignment delays are used at all, only use a unit delay.

A unit delay is a delay of one unit of time (`#1`), using the module's `timeunit` definition (or a ``timescale` compiler directive, if there is no local `timeunit` definition in the module, or the simulation time units if there is no ``timescale` in effect).

8.1.7.2 Inferred combinational logic in a sequential logic procedure

Using `always_ff` and specifying a clock edge in a sensitivity list does not mean that all the logic in procedure will be sequential logic. Synthesis compilers will infer flip-flops for each variable that is assigned with a nonblocking assignment. Blocking assignments might also infer flip-flops, depending on the order and context of the assignment statement relative to other assignments and operations in the procedure.

There are situations, however, where combinational logic dataflow behavior will be inferred from within a sequential procedure:

- Operators on the right-hand side of sequential assignments will synthesize as combinational logic, the output of which becomes the D input to the flip-flop.
- If the right-hand side of sequential assignments calls a function, the function will synthesize as combinational logic, the output of which becomes the D input to the flip-flop.
- Decision statements surrounding the assignment statements might synthesize as combinational multiplexor logic that selects which expression is used as the D input to the flip-flop.
- Blocking assignments to temporary variables in a begin-end statement group, as discussed earlier in this chapter, in section 8.1.4.3 (page 283), will simulate as combinational logic, where the temporary variable becomes an input to flip-flops inferred in subsequent nonblocking assignments in the begin-end group.

8.2 Modeling Finite State Machines (FSMs)

This book does not discuss Finite State Machines (FSM) design theory. It is assumed the reader is already familiar with state machine design. The focus of this book is on best-practice coding styles for RTL models of FSMs.

Finite State Machines spread a series of operations over multiple clock cycles, often with decision branches as to which operations are to be executed. A common usage of state machines is to set various control signals for different conditions as data is being processed.

Best Practice Guideline 8-10

Model FSMs in a separate module. (Support logic for the FSM, such as a counter that is only used by the FSM, can be included in the same module.)

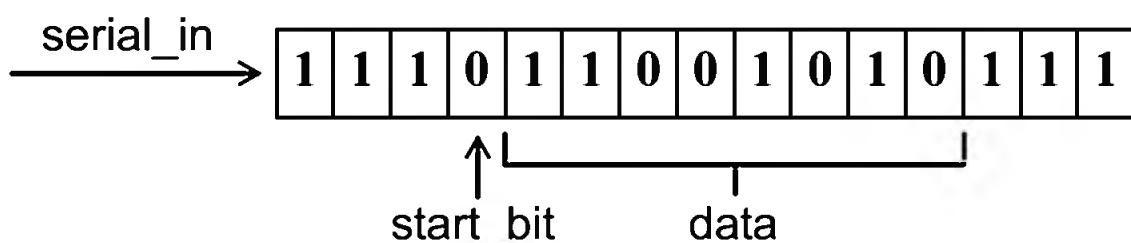
Keeping the FSM code separate from other design logic helps make the FSM easier to maintain and to reuse in multiple projects. Synthesis compilers and design automation tools that aid in FSM design also work better when the FSM code is in a separate module, and not mingled with other functionality.

The examples in this section represent the flow of a simplified 8-bit Serial-to-Parallel Interface (SPI). In order to focus on the state machine logic, this simplified SPI design does not have any control registers, and is not configurable the way a more complex SPI might be.

The simple SPI has a 1-bit `serial_in` input that is loaded into an 8-bit register over 8 clock cycles. On the first clock cycle the `serial_in` value is loaded into the most-significant bit (MSB) of the register. On the next cycle, the MSB of the register is shifted down one bit, and the next `serial_in` value is loaded into the register's MSB. This shift-and-load operation occurs 8 times over 8 clock cycles to load the serial input stream into the 8-bit parallel register.

The `serial_in` input stream includes a start-bit to indicate when to start loading the 8-bit register. The `serial_in` input is held at 1 when no transfer is occurring. The first time `serial_in` is 0 indicates that an 8-bit stream of data will follow. Thus, the `serial_in` pattern is 9 bits; a start bit followed by 8 bits of data. Figure 8-14 illustrates the pattern for an 8-bit data value.

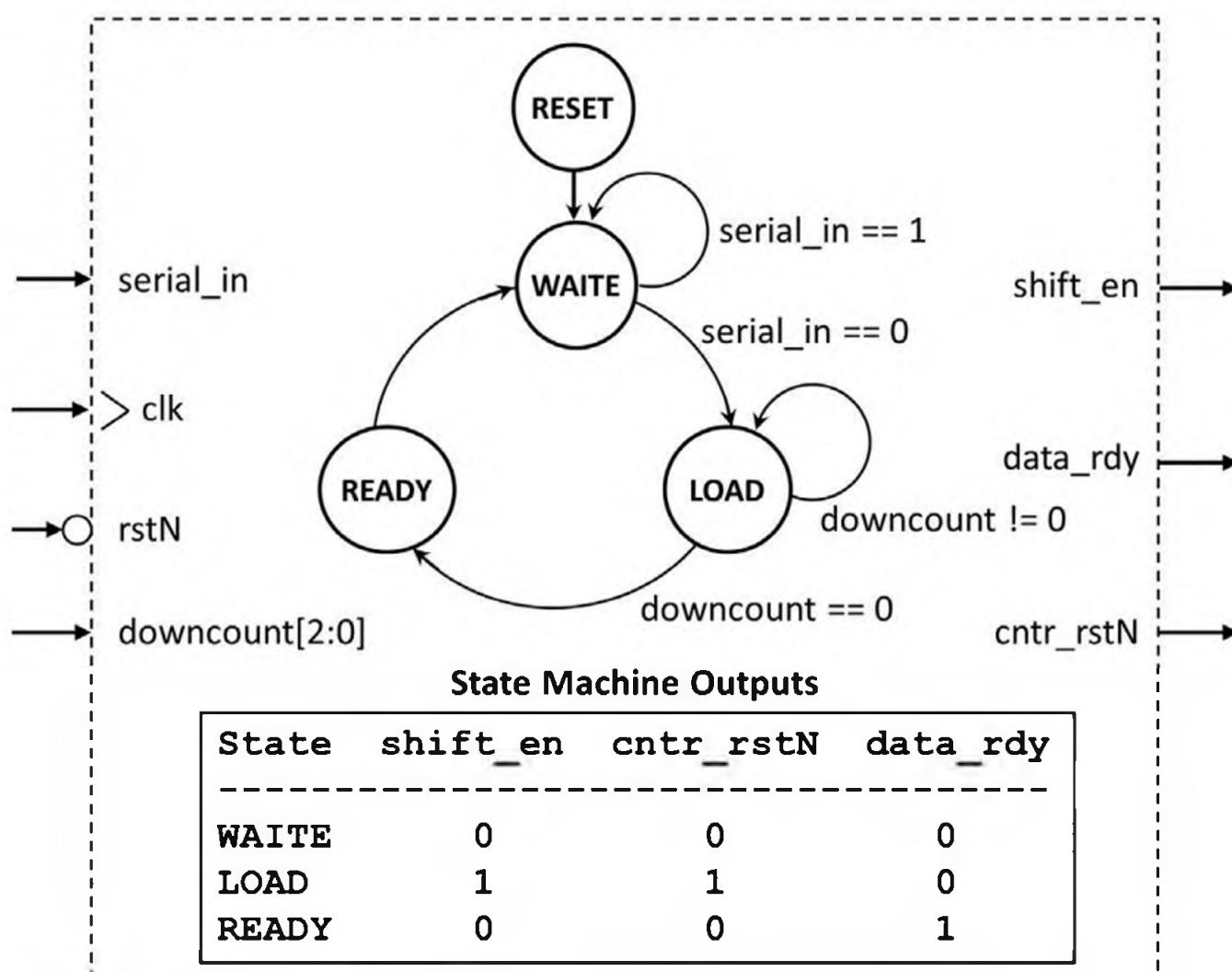
Figure 8-14: An 8-bit serial value of hex CA, plus a start bit



The simple SPI uses three states to load an 8-bit serial input stream: WAITE, LOAD and READY. (The state name WAITE is purposely spelled with an “E” at the end to differentiate it from the SystemVerilog `wait` keyword. This is not necessary syntactically, because SystemVerilog is a case sensitive language. However, some engineering tools can be invoked in a case-insensitive mode, and would not see a difference between `wait` and `WAIT`. Synthesis compilers can also generate design netlists case-insensitive languages, such as VHDL or EDIF.)

The state flow for this simple SPI state machine is shown in Figure 8-15:

Figure 8-15: State flow for an 8-bit serial-to-parallel Finite State Machine



The FSM resets to a RESET state. On the next clock cycle after reset is deasserted, the state machine transitions to a WAITE state, and then remains in the WAITE state until the `serial_in` input goes to zero, which represents the start bit. The state machine then transitions to a LOAD state, and remains in that state for 8 clock cycles. A 3-bit decrement counter is used to control how long the FSM stays in the LOAD state. When the counter reaches a count of 0, the state machine transitions to a READY state. On the next clock cycle, the FSM then transitions back to the WAITE state, where it remains until the next start bit is detected.

The FSM in this simple SPI sets three control signals as the FSM outputs:

- `cntr_rstN` is used to hold the 3-bit decrement counter in a reset state (a full count of 7).
- `shift_en` is used to enable an 8-bit shift register. When `shift_en` is 1, the values in the register are shifted down 1 bit, and a new value is loaded into the most-significant bit of the register.
- `data_rdy` is set to 1 when the 8-bit parallel register has been loaded.

The table shown in Figure 8-15 shows the values for the control signals in each state of the FSM.

The full code and resulting synthesis schematic for this simple SPI state machine is shown later in this chapter, in section 8.2.4 (page 309), after various aspects of Finite State Machine modeling have been examined.

8.2.1 Mealy and Moore FSM architectures

There are two primary architectures used for most ASIC and FPGA state machine designs: Mealy and Moore (named after George H. Mealy and Edward F. Moore, respectively). The primary difference in these architectures is when outputs from the state machine can change relative to changes in the state of the FSM. With a Moore architecture, outputs values are based solely on the current state of the FSM. Thus, the output can only change values when the state changes. In a Mealy architecture, the output values are based on a combination of the current state and other inputs to the state machine. Thus, the outputs can change asynchronous to when state changes.

At the abstract RTL modeling level, Mealy and Moore architectures are represented by the decision statements that set the FSM outputs. If only the state variable is used to determine the output values, then the behavior represents a Moore architecture. An example of this output decoding is:

```
always_comb begin
    case (state)
        RESET: begin
            cntr_rstN = '0; shift_en = '0; data_rdy = '0;
        end
        WAITE: begin ... end
        LOAD : begin ... end
        READY: begin
            cntr_rstN = '0; shift_en = '0; data_rdy = '1;
        end
    endcase
end
```

If the state variable and other signals are used to determine the outputs, then the state machine behaves as a Mealy architecture.

The following example represents a Mealy architecture because it decodes the value of state and, when in the READY state, also decodes a data_valid signal to set the output controls.

```
always_comb begin
    case (state)
        RESET: begin
            cntr_rstN = '0; shift_en = '0; data_rdy = '0;
        end
        WAITE: begin ... end
        LOAD : begin ... end
        READY: begin
            cntr_rstN = '0; shift_en = '0;
            if (data_valid) data_rdy = '1;
            else                 data_rdy = '0;
        end
    endcase
end
```

8.2.2 State encoding

The states in a Finite State Machine are represented by encoded values. There are many different codes that can be used, such as: binary count, one-hot, one-hot-0, Gray code (named after Frank Gray), and Johnson count (named after Dr. Robert Royce Johnson). The advantages and appropriate times to use each encoding style is a design engineering topic that is beyond the scope of this book. Once that choice has been made, however, it can be reflected in the RTL model of the state machine. Enumerated type labels can be defined to with specific values to represent the encoding values. Several examples follow:

```
// Gray code encoding
typedef enum logic [1:0] {RESET = 2'b00,
                           WAITE = 2'b01,
                           LOAD  = 2'b10,
                           READY = 2'b11
} states_gray_code_t;

// Johnson count encoding
typedef enum logic [1:0] {RESET = 2'b00,
                           WAITE = 2'b10,
                           LOAD  = 2'b11,
                           READY = 2'b01
} states_johnson_count_t;
```

All of these encoding examples use a **typedef** to create a user-defined type, such as `states_onehot_t`. The state variables can be declared as a user-defined type, which then limits the variables to only use the values represented by the encoded labels. If the user-defined type is defined in a package, the definition can be imported by both the state machine module and the verification testbench.

Best Practice Guideline 8-11

Define a **logic** (4-state) base type and vector size for enumerated variables.

The SystemVerilog enumerated type has a default base data type of **int**. This is a 32-bit 2-state data type, which can hide design bugs in simulation that would show up as an X value with a 4-state **logic** type. Although synthesis compilers will optimize out any unused bits of the 32-bit default vector size, it is a better coding style to be explicit for the vector size needed for the encoded values.

Enumerated types versus parameter constants. The legacy Verilog language did not have enumerated types. Instead, state values were encoded by using parameters (either the **parameter** or **localparam** keyword). For example:

```
// one-hot encoding
localparam [3:0] RESET = 4'b0001,
                  WAITE = 4'b0010,
                  LOAD  = 4'b0100,
                  READY = 4'b1000;
```

Best Practice Guideline 8-12

Use enumerated variables for FSM state variables. Do not use parameters and loosely typed variables for state variables.

Enumerated variables have strongly typed assignment rules that can prevent common coding mistakes.

While the appearance and functionality of parameters versus enumerated type labels are similar, the constructs have very different language rules. When parameters are used, the state variables will be a simple variable type, such as:

```
logic [3:0] state, next;
```

SystemVerilog variables are loosely typed, meaning a value of a different type or size can be assigned to the variable, and an implicit cast conversion will occur (see Chapter 5, section 5.15, page 198). This implicit conversion can lead to a number of programming gotchas when modeling state machines. The gotchas will compile and simulate, but can have functional bugs. These functional bugs can be subtle. At best, they impact the design schedule because the bugs need to be detected, debugged, corrected, and the design reverified. It is possible, however, that a subtle bug can go undetected, and affect the gate-level implementation of the design.

When enumerated types are defined, the state variables can be declared as that user-defined type. For example:

```
states_t state, next;
```

Enumerated type variables are more strongly typed. The definition of an enumerated type cannot have size mismatches or duplicate values. Assignments to enumerated type variables must be one of the defined labels for that enumerated type, or another enumerated variable from the same enumerated definition. Chapter 4, section 4.4.3 (page 118) discusses enumerated type assignment rules in more detail.

Synthesizing state encoding. Synthesis compilers will recognize the encoding that is defined in the RTL model, and, by default, use that encoding in the gate-level implementation. One-hot encoding might require additional information in order to achieve best synthesis Quality of Results (QoR). This is discussed in section 8.2.5 (page 313). Most synthesis compilers are configurable, and can be directed to use the RTL encoding in the gate-level implementation, or to choose an alternate encoding. This flexibility allows experimenting with different encoding schemes during synthesis to find the implementation that is best for the target ASIC or FPGA, and best meets design area, speed and power goals.

Best Practice Guideline 8-13

Make the engineering decision on which encoding scheme to use for a Finite State Machine at the RTL modeling stage of design, rather than during the synthesis process.

Most verification is done at the RTL level. Choosing the encoding scheme early in the design process:

- Ensures that the design is verified with that encoding scheme used in the gate-level implementation.
- Allows the use of a Logic Equivalence Checker — a design tool that statically compares the boolean functionality of two versions of a design — to compare the state decoding logic of the RTL model and the gate-level implementation.

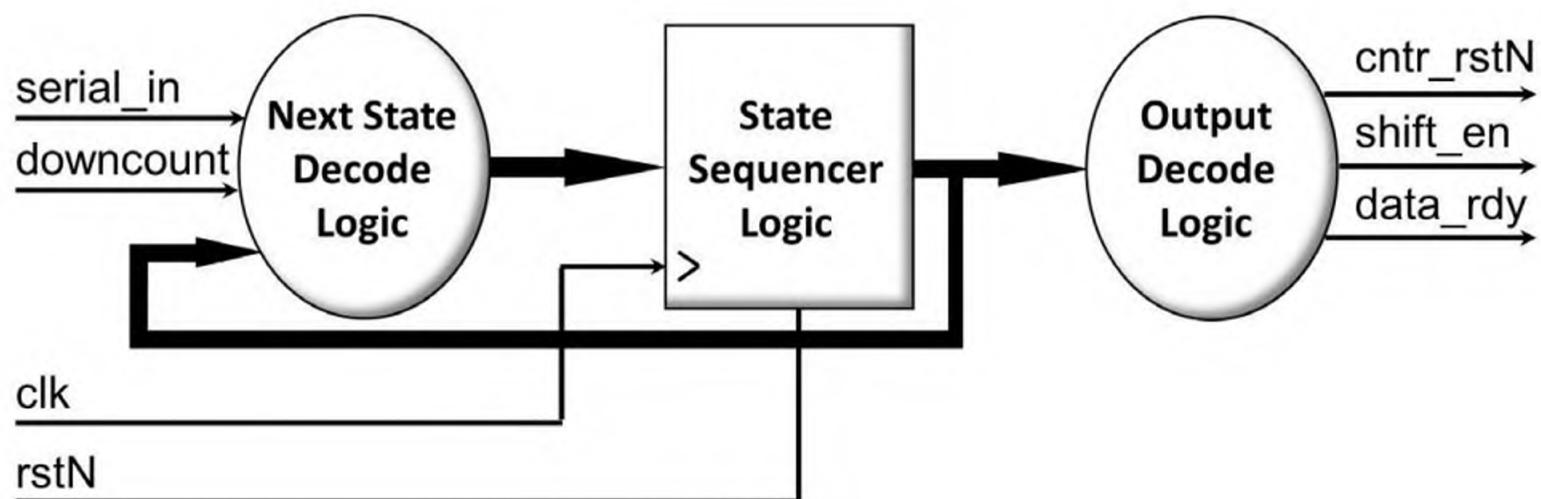
8.2.3 One, two and three-procedure FSM coding styles

State machines require a mix of sequential logic and combinational logic. Most Finite State Machines have a mix of three main blocks of functionality:

- A *state sequencer* — a sequential logic block (flip flops) that transitions from a current state to a next state on a clock edge.
- A *next state decoder* — a combinational logic block that decodes various signals to determine what the next state of the FSM should be.
- An *output decoder* — a combinational logic block that decodes the current state (Moore architecture), and possible other signals (Mealy architecture), and sets the values of the outputs of the state machine.

Figure 8-16 shows the main functional blocks that make up a typical Finite State Machine.

Figure 8-16: Primary functional blocks in a Finite State Machine



8.2.3.1 Three process state machine models

Best Practice Guideline 8-14

For most Finite State Machines, use a three-process coding style, where a separate process models each of the three main blocks of the state machine.

A three-process state machine model is simple to model and maintain, and will usually produce good synthesis Quality of Results (QoR). While it is possible to model a Finite State Machine with just one or two processes, doing so can make the RTL code harder to write, harder to debug, and harder to reuse. There can be exceptions to this guideline, where a two-process state machine model can be advantageous. Section 8.2.3.2 (page 307) discusses this situation.

Following is an example three separate process Finite State Machine, modeled as incomplete pseudocode.

```
// Current State Logic -- sequential logic
always_ff @(posedge clk or negedge rstN)
  if (!rstN)
    state <= RESET;
  else
    state <= next_state;

// Next State logic -- combinational logic
always_comb begin
  unique case (state)
    RESET: next_state = ...
    WAITE: next_state = ...
    LOAD: next_state = ...
    READY: next_state = ...
  endcase
end

// FSM outputs -- Moore architecture
always_comb begin
  unique case (state)
    RESET: fsm_outputs = ...
    WAITE: fsm_outputs = ...
    LOAD: fsm_outputs = ...
    READY: fsm_outputs = ...
  endcase
end
```

The combinational logic blocks of a state machine can also be modeled by using **assign** continuous assignment statements. Continuous assignments can be appropriate if the combinational decode logic is relatively simple.

There are several advantages for using separate processes to model each of the three main functional blocks of a state machine:

- The functionality of each block is visually obvious. This makes the code less likely to have coding errors, and easier to debug if there are errors.
- The combinational functionality is evaluated as the inputs change, which is the same way the gate-level implementation will behave. (If the next state were calculated with the state sequencer logic, inputs that affect the next state would not be evaluated until the same clock edge that transitions to the next state.)

- The separate processes are more reusable. The code for each processes can be copied into a different state machine design, and each block more easily modified for use in the new design.
- The state sequencer code, in particular, will be almost exactly the same in every state machine model.
- The next state and output combinational blocks can be sensitive to different inputs. Modeling a Moore FSM architecture almost always requires that separate procedural blocks be used.

For most designs, there is no disadvantage to not using three separate processes to model the three main functional blocks of a state machine. An exception might arise, however, if both the next state decoding and the output decoding have complex combinational logic that requires similar algorithms. In that situation, it might be more advantageous to combine these two combinational logic procedures, as discussed next, in section, 8.2.3.2.

Another rare exception to the advantages of a three-process state machine can occur if an state machine output is to be registered on the same clock cycle in which a state is entered, rather than after the new state has been stored in the state flip-flops. In this situation, some of the next state decoding might be moved into the state sequencer procedure.

NOTE

FSM outputs coming directly from combinational logic can have glitches.

An important principle of hardware design needs to be remembered and applied when designing Finite State Machines — non-registered signals can have glitches between clock cycles. State machine outputs can be stored in clocked registers to make the outputs more stable.

8.2.3.2 Two process state machine models

Since the next state decoder block and the output value decoder block are both combinational logic, it is possible to combine these two blocks of code into the same **always_comb** procedure. (The state sequencer sequential logic is the second procedure.) An advantage to the two-process state machine coding style is when both the next state combinational logic and the FSM output combinational logic share a common complex algorithm. In the three-process coding style, this algorithm would need to be duplicated in the separate procedures. This redundancy can be eliminated by combining the next state and output decoding into a single combinational logic procedure.

There are at least three disadvantages to combining the next state and output functionality into the same procedure, however. This first disadvantage is that the sensitivity list must include all the signals that can affect either the next state or the output

functionality. If, therefore, an input to the next state decoder changes, the output decoder must also be evaluated. Combining the next state and output decoders into a single process will most likely require using Mealy architecture. A second disadvantage is code maintenance and debugging. The single combinational logic procedure will contain intermingled lines of code for calculating the next state and the output values. Changes to one algorithm can inadvertently affect the other functional block. A third disadvantage is that the combined functionality might be more difficult to reuse in other projects because more project-specific signals and algorithms are lumped into the single procedure.

Highly encoded state values. An alternative two-process modeling style is to eliminate the output decoding block altogether. Instead of using conventional state encoding, such as binary count, Gray code or 1-hot, the output values for each state can be used as the state encoding. This style is sometimes referred to as a *highly-encoded state machine*. The simple SPI state machine can be coded using this style.

```
// highly encoded states -- lower 3 bits equal output values
typedef enum logic [3:0] {RESET = 4'b0_000,
                           WAITE = 4'b1_000,
                           LOAD  = 4'b1_110,
                           READY = 4'b1_001
                           } states_t;

states_t state, next;

assign {cntr_rstN, shift_en, data_rdy} = state[2:0];
```

The continuous assignment statement in this example does not represent any output decoding logic. It is simply a buffer to change the names of the state bits to the output signal names.

Highly encoded state machines are a Moore architecture, because the output values are dependent solely on the current state, and no other inputs.

8.2.3.3 One process state machine models

It is possible to model all of the functionality of a state machine in a single sequential logic procedure. Incomplete pseudocode for this modeling style is:

```
/// State Sequencer with next state and output decoding ///
always_ff @(posedge clk or negedge rstN)
  if (!rstN)
    state <= RESET;
  else begin
    case (state)
      RESET: begin
        state <= ...
        fsm_outputs <= ...
      end
```

```

WAITE: begin
    if (serial_in == '0) begin
        state <= ...
        fsm_outputs <= ...
    end
    else begin
        state <= ...
        fsm_outputs <= ...
    end
end

LOAD: begin
    if (downcount != '0) begin
        state <= ...
        fsm_outputs <= ...
    end
    else begin
        state <= ...
        fsm_outputs <= ...
    end
end

READY: begin
    state <= ...
    fsm_outputs <= ...
end

endcase
end

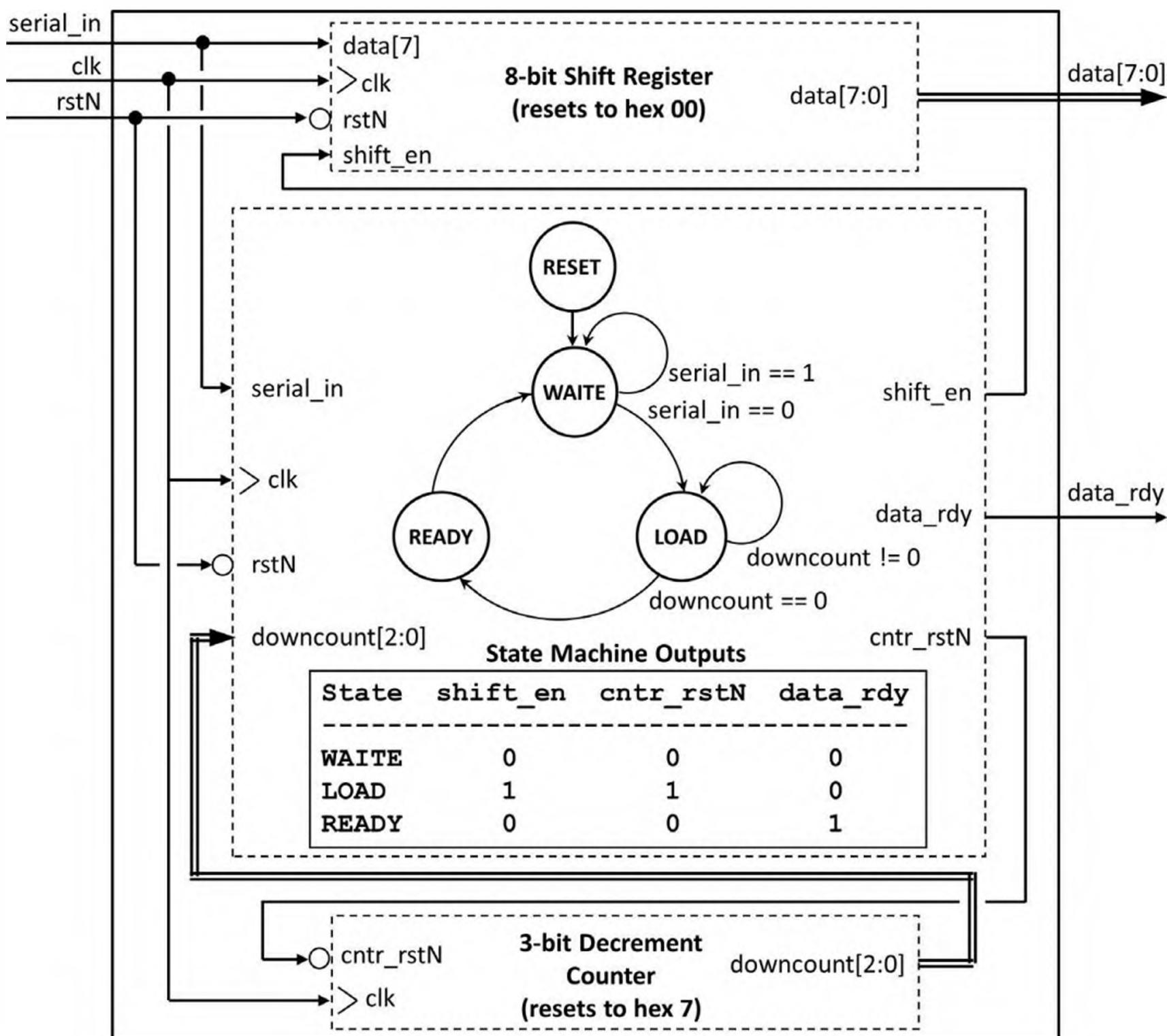
```

There are engineers, often those who come from a VHDL modeling background, who are adamant that a single process coding style is a preferred coding style. In SystemVerilog, however, a one process state machine has many disadvantages. Chief among these disadvantages is that the RTL simulation does not accurately model the gate-level implementation. At the gate level, combinational logic outputs update whenever an input value changes. In the one-process RTL simulation, however, the combinational logic mixed in the sequential logic block is only evaluated on a clock edge. Another disadvantage is that, because the single process has all the FSM functionality jumbled together, this modeling style can be difficult to debug if a functional bug is found during RTL simulation. For the same reason, the one process coding style is also difficult to reuse in other projects, without having to rewrite a lot of the code within the one procedure.

8.2.4 A complete FSM example

A more complete functional block diagram for the simple SPI state machine is shown in Figure 8-17. The full code for the simple SPI is shown in Example 8-3, and the generic gate-level implementation from synthesis is shown in Figure 8-18. The countdown decrement counter and the register to store the serial_in data stream are integral to the functionality of the state machine, and are included in the state machine module.

Figure 8-17: Functional block diagram for a serial-to-parallel finite state machine



Example 8-3: RTL model of an 8-bit serial-to-parallel finite state machine

```

module simple_spi
(output logic [7:0] data,
output logic data_rdy,
input logic serial_in, clk, rstN);

logic [2:0] downcount;
logic cntr_rstN, shift_en;

// One-hot-0 state machine encoding
typedef enum logic [2:0] {
    RESET = 3'b000,
    WAITE = 3'b001,
    LOAD = 3'b010,
    READY = 3'b100
} states_t;

states_t state, next_state; // internal state variables

```

```
//////////  
// 4-state State Machine with async active-low reset  
//////////  
  
// Current State Logic -- sequential logic  
always_ff @(posedge clk or negedge rstN)  
  if (!rstN)  
    state <= RESET;  
  else  
    state <= next_state;  
  
// Next State logic -- combinational logic  
always_comb begin  
  unique case (state)  
    RESET:  
      next_state = WAITE;      // move out of reset  
    WAITE:  
      if (serial_in == '0)  
        next_state = LOAD;     // start bit sensed  
      else  
        next_state = WAITE;   // no start bit  
    LOAD:  
      if (downcount == '0)  
        next_state = READY;   // 8 bits are loaded  
      else  
        next_state = LOAD;    // keep loading  
    READY:  
      next_state = WAITE;   // return to wait state  
  endcase  
  end  
  
// FSM outputs -- Moore architecture  
always_comb begin  
  unique case (state)  
    RESET: {cntr_rstN, shift_en, data_rdy} = 3'b000;  
    WAITE: {cntr_rstN, shift_en, data_rdy} = 3'b000;  
    LOAD: {cntr_rstN, shift_en, data_rdy} = 3'b110;  
    READY: {cntr_rstN, shift_en, data_rdy} = 3'b001;  
  endcase  
  end  
  
//////////  
//8-bit shift register with enable, async active-low reset  
//////////  
always_ff @(posedge clk or negedge rstN)  
  if (!rstN)  
    data <= '0;  
  else if (shift_en)  
    data <= {serial_in, data[7:1]};
```

```
//////////  

// 3-bit Decrement Counter with async active-low reset  

//////////  

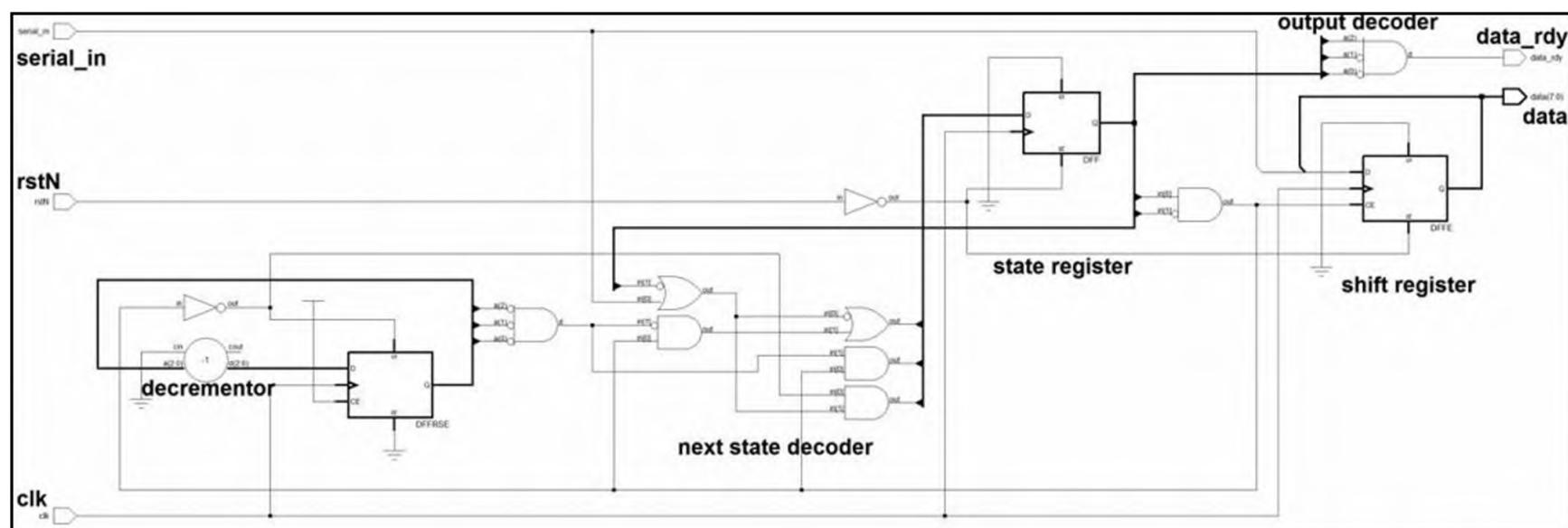
always_ff @(posedge clk) // synchronous active-low reset
  if (!cntr_rstN)
    downcount <= '1;                      // reset to full count
  else
    downcount <= downcount - 1;      // decrement counter
```

endmodule: simple_spi

NOTE

Proper usage of blocking and nonblocking assignments is critical for obtaining proper, race-free simulation behavior. Use blocking assignments for all combinational logic assignments. Use nonblocking assignments for all sequential logic assignments. Correct usage of blocking and nonblocking assignments is discussed in Chapter 1, section 1.5.3 (page 23), Chapter 7, section 7.2.4 (page 261) and in section 8.1.4 (page 278) of this chapter.

Figure 8-18: Synthesis result for Example 8-3: Simple-SPI using a state machine



Technology independent schematic (no target ASIC or FPGA selected)

This schematic does not display well in the page size of this book, but the five major portions of the Simple-SPI model are visible: The decrement counter, the data shift register, and the three processes that make up the state machine, with combinational next state logic before the state register, the state register itself, and combinational logic to decode the state machine outputs.

8.2.5 Reverse case statement one-hot decoder

The SystemVerilog case statement compares a *case expression* to a series of *case items*. The case expression is in parentheses, immediately after the **case** keyword. The normal usage of case statements is to specify a variable for the case expression, along with fixed values for the case items. In the following typical usage example, the state variable is the case expression, and the fixed values of the enumerated state labels, RESET, WAITE, etc. are the case items.

```
// one-hot values
typedef enum logic [3:0] {RESET = 4'b0001,
                           WAITE = 4'b0010,
                           LOAD  = 4'b0100,
                           READY = 4'b1000
} states_t;

states_t state, next;

always_comb begin
  unique case (state)
    RESET: next =           WAITE;
    WAITE: next = (serial_in == '0) ? WAITE : LOAD;
    LOAD:  next = (downcount == '0) ? LOAD : READY;
    READY: next =           WAITE;
  endcase
end
```

NOTE

Synthesis compilers might infer a multi-bit comparator when a multi-bit case expression is compared to a multi-bit case item. This is not an efficient gate-level implementation of one-hot state machine encoding.

Since only one bit is set (“hot”) in one-hot encoding, only 1-bit comparators are needed to determine which bit is set. The example above, however, is modeled as a 4-bit comparator, and might synthesize as just that — 4-bit gate-level comparators. Synthesis compilers might require the addition of a pragma or configuration setting to instruct the compiler to optimize the implementation as one-bit comparators.

Best Practice Guideline 8-15

Use reverse case statements to model one-hot state machines that evaluate 1-bit values. Do not use multi-bit vectors for one-hot case expressions and case items.

A best-practice coding style is to model one-hot state machines in such a way so that all synthesis compilers will recognize the one-hot encoding without the use of proprietary synthesis pragmas or configuration options.

SystemVerilog allows the case expression value and the case item value to be reversed — the fixed value to be matched can be the case expression, and variables used as the case items. This reversed case statement is a convenient way to decode a one-hot state variable, where only one bit is set in the state variable. For example:

```
typedef enum logic [3:0] {
    RESET = 4'b0001, // one-hot encoding
    WAITE = 4'b0010,
    LOAD  = 4'b0100,
    READY = 4'b1000
} states_t;

states_t state, next; // enumerated state variables

always_comb begin
    unique case (1'b1)
        state[0]: next = WAITE;
        state[1]: next = (serial_in == '0) ? WAITE : LOAD;
        state[2]: next = (downcount == '0) ? LOAD : READY;
        state[3]: next = WAITE;
    endcase

```

end

By reversing the case statement, all that is required at the gate-level is a 1-bit comparator to compare the case expression and each case item, as opposed to when the full state vector is compared to all bits of each case item. Synthesis compilers might yield more optimized synthesis results using the reversed case style for one-hot state machines, than the standard style of case statements.

One drawback of the reverse case statement shown above is that the code is not self-documenting. A case item such as `state[0]:` does not make it obvious that this is the `RESET` state. Reverse case statements can be made more readable by using constants to define names for the index numbers of the one-hot bits.

One way of labeling the state bits is shown in the following example. This example is simple and easy to read, but has limitations, which are discussed after the example.

```
localparam RESET = 0, // index of RESET one-hot bit
            WAITE = 1, // index of WAITE one-hot bit
            LOAD  = 2, // index of LOAD one-hot bit
            READY = 3; // index of READY one-hot bit

logic [3:0] state, next; // simple 4-bit variables
```

```

always_comb begin
    next = '0;           // clear all bits in next
    unique case (1'b1) // set the bit representing next state
        state[RESET]:               next[WAITE] = '1;
        state[WAITE]: if (serial_in == '0) next[WAITE] = '1;
                           else          next[LOAD ] = '1;
        state[LOAD ]: if (downcount == '0) next[LOAD ] = '1;
                           else          next[READY] = '1;
        state[READY]:               next[WAITE] = '1;
    endcase
end

```

In the preceding example, `state` and `next` are loosely-typed 4-state variables, instead of enumerated variables. This means the strongly-typed protections of enumerated types have been lost (see Chapter 4, section 4.4.3, page 118). A better coding style combines using enumerated types for the state variables and using constant names for the one-hot bits. This style also allows the definition of the one-hot values for each state to be derived from the definition for the state bits.

```

localparam RESET_BIT = 0, // index of RESET one-hot bit
             WAITE_BIT = 1, // index of WAITE one-hot bit
             LOAD_BIT  = 2, // index of LOAD one-hot bit
             READY_BIT = 3; // index of READY one-hot bit

typedef enum logic [3:0] {
    RESET = 1 << RESET_BIT, // set the RESET bit
    WAITE = 1 << WAITE_BIT, // set the WAITE bit
    LOAD  = 1 << LOAD_BIT, // set the LOAD bit
    READY = 1 << READY_BIT // set the READY bit
} states_t;

states_t state, next; // enumerated state variables

always_comb begin
    unique case (1'b1)
        state[RESET_BIT]: next =           WAITE;
        state[WAITE_BIT]: next = (serial_in=='0) ? WAITE : LOAD;
        state[LOAD_BIT ]: next = (downcount=='0) ? LOAD : READY;
        state[READY_BIT]: next =           WAITE;
    endcase
end

```

The value for each state label is calculated by shifting a value of 1 (decimal) to the bit position that is “hot” for that state. In this example, the enumerated variable is 4 bits wide. A value of 0001 shifted 0 times (the value of `RESET_BIT`) is 0001 (binary). A 0001 shifted 1 time (the value of `WAITE_BIT`) is 0010 (binary), and shifted 2 times (the value of `LOAD_BIT`) is 0100 (binary), and shifted 3 times (the value of `READY_BIT`) is 1000 (binary).

Basing the enumerated state labels off of the state bit constants means:

- There is no possibility of a coding error that defines different one-hot bit positions in the local parameter and the enumerated type definitions.
- Should the design specification change the one-hot definitions, only the local parameters specifying the bit positions have to change. The enumerated type defining the state names will automatically reflect the change.

Using unique case to Optimize reverse case statements. All of the reverse case statements above used the keyword pair **unique case**. This decision modifier is necessary in these examples to help ensure optimal synthesis results and better design verification in simulation. SystemVerilog language rules require that case items are evaluated in the order in which they are listed. This rule means that each case item takes priority over all subsequent case items. This priority encoded behavior requires more logic gates and longer propagation paths than a simple parallel decoder.

Synthesis compilers will analyze case statements to see if all case item values are unique, meaning no two case items have the same value. If synthesis can determine that it is impossible for two case items to be true at the same time, synthesis compilers will automatically remove the priority encoding, and evaluate the case items in parallel.

With a reverse case statement, however, the case items are not literal values that a synthesis compiler can evaluate for having unique values. Instead, the case items are bits of the state variable that are set and changed during simulation. Because the case items are variable, synthesis compilers cannot determine that all case items have unique values. Therefore, synthesis compilers will implement this reverse case statement one-hot decoder with priority encoded logic, and will not automatically optimize the case statement decoding for parallel evaluation.

The **unique** decision modifier tells synthesis compilers to treat the case items as unique values, even when the compiler cannot determine this on its own. Synthesis will optimize the gate-level implementation to have parallel decoding, instead of priority encoded logic.

The **unique** decision modifier also has an important effect on simulation by enabling two dynamic checks during simulation. A run-time warning is issued if the case statement is entered and two or more case items are true at the same time. Thus, simulation will catch any design bugs that result in the state variable having two bits set at the time. A run-time warning is also issued if the case statement is entered and no case items are true. This can help detect design bugs, such as the state variable being reset to 0, instead of a one-hot value.

SystemVerilog also has a **unique0** decision modifier. Like **unique**, this modifier informs synthesis compilers to assume all case items are mutually exclusive, and to use parallel decoding instead of priority encoded logic. However, the **unique0** modifier only enables run-time checking that multiple case items are never true at the same time. The modifier does not enable checking for no case items being true. The

unique modifier is more appropriate for reverse case statements because it enables run-time checking for only one case item matching as well as for no case items matching.

Chapter 7, section 7.4.2 (page 266) discusses the **unique** and **unique0** decision modifiers in more detail.

8.2.6 Avoiding latches in state machine decoders

Decoding state values to determine the next state or output control values can potentially infer unintentional latches in the gate-level implementation created by synthesis. Latches might be inferred if the state variable could possibly have more values than those that are decoded by the combinational logic block. If, for example, the state variable used one-hot encoding, any values that are not one-hot are not used. Synthesis compilers might add latches to the gate-level implementation to handle those unused state values.

Unintentional latches can occur in Finite State Machine combinational logic procedures because it is common not to use every possible value of a state variable. For example, a 4-state FSM encoded using 1-hot values requires a 4-bit vector, but only 4 of the 16 possible values of that vector are used. Chapter 9, section 9.2 (page 327) discusses what can cause latches to be inferred, and several coding styles that can be used to avoid these latches.

8.3 Modeling memory devices such as RAMs

A register, which is most often made from flip-flops, stores a single value. A collection of several registers can be used to store multiple values. Designs often need large blocks of storage as well, such as a program memory or data memory. Using flip-flop based registers is not a practical way to implement these large blocks of memory storage at the gate level. Instead, memory components, such as RAM (Random Access Memory) devices, are used for this type of storage.

ASICs and FPGAs have pre-defined and pre-optimized memory components for larger blocks of storage. Since these are predefined in the ASIC or FPGA library, they are not modeled at the RTL level and are not synthesized.

Best Practice Guideline 8-16

Behavioral RAM models should be defined in a separate module.

RTL models of a design that access memory devices need to instantiate behavioral models of RAMs in order to fully verify the RTL functionality of the rest of the design. These behavioral RAM models are not synthesized. Instead, after the RTL design has been synthesized to a gate-level implementation, the instance of the behav-

ioral RAM module can be replaced with an instance of the optimized memory device from the target ASIC or FPGA library.

Modeling memory device storage. At an abstract, non-RTL level of modeling, the storage of a memory device is represented by a one-dimensional array of variables. For example:

```
logic [7:0] mem [0:255];
```

This one-dimensional array is sometimes referred to as a *memory array*. The word size of this memory example is 8 bits wide and the number of storage locations is 256, starting with address 0 and ending with address 255. The array of variables can be either a 4-state **logic** type or a 2-state **bit** type. The advantage of the **logic** type is that it begins simulation with a value of X in every location of the array. Therefore, an X value will be returned if an array location is read before it has been written. Reading from a nonexistent array location will also return an X value, which can indicate that an address value in the design is out-of-bounds.

There is also a disadvantage to using the 4-state **logic** type for memory models. In simulation, two bits of virtual memory are required for each bit of modeled memory in order to encode a 4-state value for each bit. A 128 gigabyte RAM model will require 256 gigabytes of virtual memory to simulate with 4-state values. Simulating very large blocks of memory using 4-state types can require excessive usage of operating system virtual memory, which can impact the run-time performance of the simulation.

The 2-state **bit** type can also be used to represent the storage of a memory array. For example:

```
bit [7:0] mem [0:255];
```

Only one bit of virtual memory is needed to represent each bit of a 2-state memory model. Very large memories will be much more efficient in simulation when using a 2-state type for the memory array. The trade-off is that the uninitialized value of a 2-state type is 0. Reading from an uninitialized array location can appear to retrieve a valid value, which could hide bugs in a design. Reading from a nonexistent array location will also return 0.

8.3.1 Modeling asynchronous and synchronous memory devices

RAMs and other types of block storage are either asynchronous or synchronous.

Asynchronous memory models. An asynchronous RAM can be written to, or read from, at anytime, without the need of a clock. An example of a behavioral asynchronous RAM model is:

```

module RAM
  (inout logic [7:0] data, // bidirectional port
   input logic [7:0] addr,
   input logic nrd, // active-low read control
   nwr, // active-low write control
   ncs // active-low chip select
  );
  logic [7:0] mem [0:255];
  assign data = (!nrd && !ncs) ? mem[addr] : 'Z;
  always @ (nwr, ncs, addr, data)
    if (!nwr && !ncs) mem[addr] <= data;
endmodule: RAM

```

The storage in the RAM model is represented by the one-dimensional array of 8-bit variables. The rest of the functionality in the model is the logic to read a value from a specific address of the array, or to write a value into a specific address of the array. This RAM example has three control inputs in addition to the primary data and address inputs. All three control inputs are active-low. The `ncs` (not chip select) must be active (0) in order to write to, or read from, the RAM. The `nwr` (not write) control is active when writing to the RAM, and `nrd` (not read) is active when reading from the RAM.

The `data` port is bidirectional, and is used as an input when writing into the RAM and an output when reading from the RAM. The RAM drives data as an output when both `nrd` and `ncs` are active (0). The RAM tri-states the data bus, allowing an external driver to put values on the bus, when either the RAM is not selected (`ncs` is high), or when it is not being read (`nrd` is high).

SystemVerilog syntax requires that bidirectional ports be declared as a net data type such as `wire` or `tri`. The example above declares `data` as a `logic` type, which can have 4-state values, but does not declare a data type for `data`. When no data type is specified for a module port, SystemVerilog infers a `wire` data type for input and inout ports, and a `var` variable data type for output ports. This implicit data type inference is correct for this RAM model.

Synchronous memory models. Synchronous RAMs store values, and read back values, on a clock edge. Synchronous RAMs behave similarly to flip-flops, and are modeled in a similar way.

```

module SRAM
  (inout logic [7:0] data, // bidirectional port
   input logic [7:0] addr,
   input logic      clk,
                  nrd, // active-low read control
                  nwr, // active-low write control
                  ncs  // active-low chip select
  );
  logic [7:0] mem [0:255];
  assign data = (!nrd && !ncs) ? mem[addr] : 'Z;
  always @(posedge clk)
    if (!nwr && !ncs) mem[addr] <= data;
endmodule: SRAM

```

At the abstract behavioral level of modeling, the only difference between asynchronous and synchronous memories is the sensitivity list of the always procedure.

Observe that the two RAM models in the preceding examples use the general purpose **always** procedure instead of the RTL-specific **always_ff** or **always_latch** procedures. The RTL-specific procedures enforce coding rules for synthesis, one of which is that the variables assigned in the procedure cannot be assigned from any other source. Abstract memory models are not intended to be synthesized, and do not need to adhere to these synthesis rules. Indeed, enforcing synthesis rules would limit the usefulness of these abstract behavioral models. It is common to load memory models from outside of the always procedure, such as for a testbench to load a program into a RAM model. The general purpose **always** procedure permits this external loading of the memory array, whereas an **always_ff** or **always_latch** procedure would prohibit it.

8.3.2 Loading memory models using \$readmemb and \$readmemh

To simulate and verify RTL models that access memories, it can be useful to load a memory array with useful data that is stored in a file. SystemVerilog provides **\$readmemb** and **\$readmemh** system tasks to read values from a file and load the values into a memory array. These verification tasks load an array in a single operation, in zero simulation time and clock cycles.

The **\$readmemb** and **\$readmemh** system tasks read a *pattern file*. Pattern files are simple ASCII text files containing a list of logic values, referred to as a pattern. **\$readmemb** requires that each pattern represent a binary value, comprising the characters 0, 1, Z or X (the Z and X are not case sensitive). **\$readmemh** requires that each

pattern represent a hexadecimal value, comprising the characters 0 through F, Z or X (A through F, Z and X are not case sensitive). Each pattern is separated by a white space, which can be a space, tab or new-line. Pattern files can contain either style of SystemVerilog comment, which are ignored by the readmem commands. Pattern files can also contain an address for where in the memory array the next pattern should be loaded. An address is preceded by the @ token, and is always specified in hex, regardless of whether the value patterns are in binary or hex.

An example of a binary pattern file is:

```
/* Program file to load into RAM */
0100_1100 // 1st pattern
1100_1100 // 2nd pattern
1010_1010 // ...
@F0 // load next pattern at address F0 (hex)
1111_0000
0110_1101
1011_0011
```

Both of the readmem commands have four arguments:

```
$readmemb("file", array_name, start_address, end_address);
$readmemh("file", array_name, start_address, end_address);
```

- "file" is the name of the pattern file, specified in quotation marks. The file name string can be a simple file name, or can include a relative or full directory path. By default, SystemVerilog searches for this file in the same operating system directory from which simulation was invoked. Simulators might provide ways to change this default search location.
- array_name is the name of the memory array in which the patterns are to be loaded. Readmem tasks are typically called from within verification code, and not from within the memory module containing the array. Therefore, the array name is typically specified with a full module instance hierarchy path.
- start_address specifies the address of the array into which the first pattern should be loaded. Each subsequent pattern is loaded into each subsequent array address, unless a new address is specified in the pattern file. The start address argument is optional. If it is not specified, the first pattern in the file is loaded into the lowest address number of the array.
- end_address specifies where to stop loading the array. This argument is also optional. If not specified, the task continues loading the array until either the last address of the array or the end of the pattern file is reached.

Following is an example of using a readmem task to load one of the RAM examples shown earlier in this section.

```
initial begin
    $readmem("boot_program.txt", top.chip.ram1.mem);
end
```

Although initial procedures are generally not synthesizable, synthesis compilers recognize this specialize usage of an initial procedure to load a memory array.

8.4 Summary

This chapter has explored best practice coding styles for modeling synthesizable flip-flops, registers and Finite State Machines. Abstract, non-synthesizable behavioral models of memory devices such as RAMs were also discussed.

Two important coding practices should be followed when writing RTL models of sequential logic:

- Use the RTL-specific **always_ff** procedure
- Use nonblocking assignments.

Adhering to these practices must be done by the engineer writing the RTL code. The SystemVerilog language does not mandate these important coding styles.

Sequential devices can have many different ways of being reset. This chapter has explored the proper coding styles and best-practice considerations for modeling and synthesizing synchronous, asynchronous, active-high, and active-low resets. Avoiding potential simulation glitches with set/reset flip-flops has also been examined.

Finite State Machines have three major parts: a state sequencer, next state decoding, and output value decoding. While there are a variety of possible ways to code state machines, the most advantageous style is to use 3 separate always procedures to represent the three major parts of the state machine.

* * *

Chapter 9

Modeling Latches and Avoiding Unintentional Latches

Abstract — This chapter presents best coding practice recommendations for modeling latches in synthesizable RTL designs. The use of latches in ASIC and FPGA designs is an oft-debated engineering topic. This book is neutral on this debate. The purpose of this book is to show how to properly model latch behavior, should the engineering decision be made to utilize latches in a project.

A related topic is avoiding unintentional latches. This chapter discusses the coding styles that might infer latches where none are wanted, and several coding styles to avoid inferring latches. The pros and cons of these coding styles are presented, and best-practice coding styles recommended.

The topics presented in this chapter include:

- Proper RTL coding styles for representing latch behavior
- RTL code that infers latches when none are intended
- Modeling full (complete) decision statements
- The SystemVerilog `unique`, `unique0` and `priority` decision modifiers
- The obsolete X value assignment, and its disadvantages
- The obsolete `full_case` and `parallel_case` synthesis pragmas

9.1 Modeling Latches

There are several types of latches in digital circuitry, with the most common types being *SR latches* (also called set/reset latches) and *transparent latches* (also called D-type latches). The reasons for using — or not using — latches in a design is a general digital engineering topic, and beyond the scope of this book. Suffice it to say that the Static Timing Analysis (STA) and Design for Test (DFT) tools used in the back-end steps of designing ASICs or FPGAs work well with synchronous flip-flops, but using these tools with latch-based designs is more difficult. Many ASIC and FPGA designers avoid the use of latches to simplify the STA and DFT steps of a design flow.

Most ASIC and FPGA technologies support the use of transparent D-type latches. This section discuss best-practice guidelines for modeling this type of latch, should the choice be made to have latches in the design.

From an RTL modeling perspective, a latch is a cross of combinational logic and sequential logic. Latches do not have a clock, and do not change on a positive or negative edge transition. With latches, the output value is based on the values of the inputs, which is the behavior of combinational logic. However, latches also have storage characteristics. The output value is a reflection of both the input values and the state of the internal storage, which is the behavior of sequential logic.

Transparent latch behavior can be modeled by using either the general purpose **always** procedure or an RTL-specific **always_latch** procedure. The sensitivity list for transparent latches is identical to the sensitivity list for combinational logic. It must contain all signals that are read within the procedure.

The same synthesis restrictions and best practice guidelines for combinational logic procedures also apply to latch procedures: The body of the procedure should not contain any form of propagation delay (no #, @ or **wait** time delays), and no other procedural block or continuous assignment can make assignments to the same variables used on the left-hand side of the latch procedure.

Best Practice Guideline 9-1

Use nonblocking assignments (`<=`) to model latch behavior.

Because latches propagate the storage of an internal variable, nonblocking assignments should be used to assign values to the output variables. Nonblocking assignments mimic the gate-level propagation of sequential devices that have internally stored values. Refer to Chapter 1, section 1.5.3.4 (page 27) for a more detailed discussion regarding nonblocking assignments.

9.1.1 Modeling latches with the general purpose always procedure

Best Practice Guideline 9-2

Use the RTL-specific **always_latch** procedure to model latch based logic.
Do not use the generic **always** procedure in RTL models.

Though not recommended for RTL modeling, properly using the general purpose **always** procedure for modeling latch-based logic is discussed briefly, because it is common to see this general purpose procedure in legacy Verilog models.

When using the general purpose **always** procedure, the sensitivity list must be explicitly specified, or inferred using an `@*` sensitivity list, in the same way as with a

combinational logic **always** procedure. The following code snippet illustrates a simple transparent latch modeled with the general purpose **always** procedure.

```
logic [7:0] in, out; // 8-bit variables
logic ena;          // scalar (1-bit) variable

always @(in, ena) begin // combinational logic sensitivity
    if (ena) out <= in;
end
```

Observe that the **if** statement does not have an **else** branch. When the **always** procedure sensitivity list triggers, the output variable, **out**, is updated to a new value if **ena** is true (1). If **ena** is false (0), however, no change is made to **out**. As a variable, **out** retains the value of its previously assigned value. Synthesis compilers will infer a transparent latch from this code to maintain the stored state of a variable.

A disadvantage of using always to model latches. There are other ways in which a general purpose **always** procedure can represent latches. A latch will be inferred whenever a non-clocked always procedure triggers, and there is a possibility that a variable that is an output of the procedure is not updated. The following simple example has an **if** statement with an **else** branch, but each branch updates a different variable.

```
always @* begin // inferred combinational logic sensitivity
    if (sel) y1 = in;
    else     y2 = in;
end
```

The variable that is not updated in each branch will store its previous value. Synthesis will infer latches for both the **y1** and **y2** variables in this example.

This example shows a problem with using the general purpose **always** procedure to model latches. There is no way for software tools, or other engineers, to know if a latch was intended. Since the transparent latch sensitivity list is identical to a combinational logic sensitivity list, it might be that the RTL designer's intent was to model combinational logic, and just inadvertently left out code to assign to all variables each time the **always** procedure is entered.

9.1.2 Modeling latches with the **always_latch** procedure

SystemVerilog adds an RTL-specific **always_latch** procedure to the original Verilog language. Using **always_latch** documents that it is intended to have latched behavior in the procedure. Software tools, such as lint checkers and synthesis compilers, can issue warnings or errors if the procedure does not represent latched functionality.

The **always_latch** is an always procedure with additional modeling rules to help ensure that RTL code adheres to synthesis requirements. These rules are:

- A complete combinational logic sensitivity list is automatically inferred. This automatic sensitivity list includes all signals that are read within the procedure.
- Using #, @ or **wait** to delay execution of a statement in an **always_latch** procedure is not permitted, enforcing the synthesis guideline for using zero-delay procedures. (Nonblocking intra-assignment unit delays are permitted, since this type of delay does not delay the execution of a statement. See Chapter 8, section 8.1.7.1, page 297 for details on this special type of synthesizable delay.)
- Any variable assigned a value in an **always_latch** procedure cannot be assigned from another procedure or continuous assignment, which is a restriction required by synthesis compilers.

These **always_latch** rules are identical to those of the **always_comb** procedure introduced in chapter 7, section 7.2.3 (page 260). The rules match the coding restrictions that synthesis compilers require for RTL models of latched logic, and help to ensure that engineering time is not lost verifying a design that cannot be synthesized.

Example 9-1 illustrates the use of **always_latch** procedures to latch the inputs of each stage of a two-stage pipeline. Each latch is gated by a different clock. The latches are in transparent mode when the clock is high, and are in a latched mode when the clock is low. The pipelined latches allow the multiplier in the first stage of the pipeline to take up to one-half clock cycle longer to produce the intermediate result used as an input in the second stage. Instead of the first multiplier output being registered into flip-flops by a positive edge of clock, the multiplier output can continue to be calculated during the entire positive edge portion of the clock cycle. The effect is that each multiplier is stealing time from the next clock cycle to complete the multiplication operation.

Example 9-1: Using intentional latches for a cycle-stealing pipeline

```

module latch_pipeline
#(parameter N = 4)                                // bus size
(input logic          clk1, clk2,    // clock inputs
 input logic [N-1:0] a, b, c,      // scalable input size
 output logic [N-1:0] out        // scalable output size
);
logic [N-1:0] tmp;

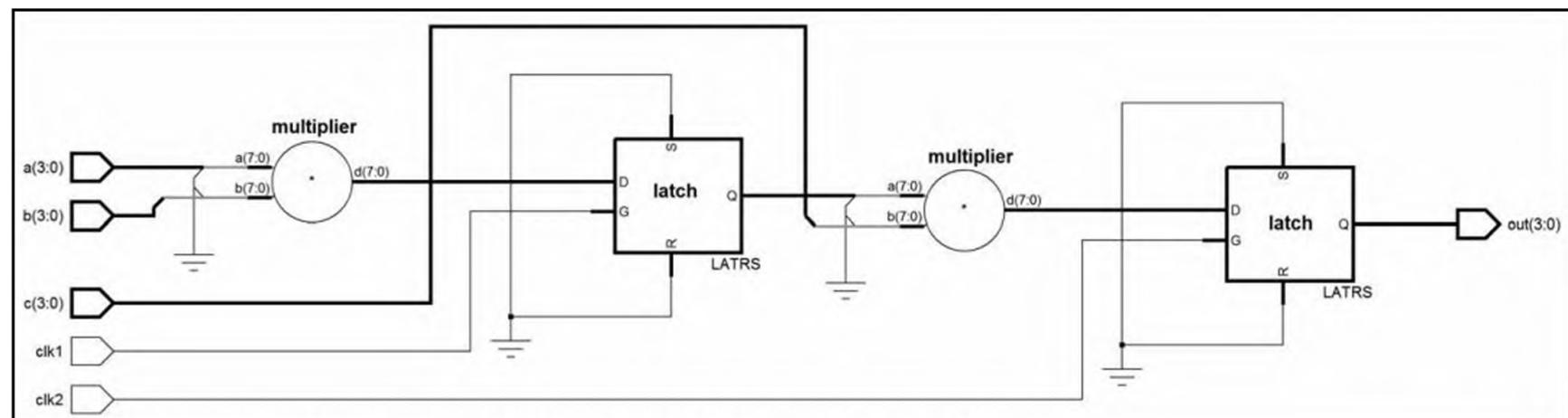
always_latch begin
  if (clk1)           // transparent when clk1 is high
    tmp <= a * b;
  end                  // latched when clk1 is low

always_latch begin
  if (clk2)           // transparent when clk2 is high
    out <= tmp * c;
  end                  // latched when clk2 is low
endmodule: latch_pipeline

```

Figure 9-1 shows the results from synthesizing Example 9-1. The latched output of each multiplier is evident.

Figure 9-1: Synthesis result for Example 9-1: Pipeline with intentional latches



Technology independent schematic (no target ASIC or FPGA selected)

The latch symbols in this schematic are generic latch symbols that could represent any type of latch. When the synthesis compiler maps this schematic to a specific target ASIC or FPGA, an appropriate device-specific latch will be selected from the latch types available in that device.

9.2 Unintentional latch inference

A common problem in RTL modeling is synthesis compilers inferring latches where no latch was intended.

NOTE

Synthesis will infer a latch whenever a non-clocked always procedure is entered, and there is a possibility that one or more of the variables used on the left-hand side of assignment statements will not be updated.

A variable that is not updated when a non-clocked always procedure is triggered retains the previous value of the variable. The non-clocked storage of a variable value requires a latch in order for the gate-level implementation to also have storage. To avoid unintentional latches, combinational logic cannot use the storage of a variable.

When coding at the RTL level, it is possible to model combinational logic that inadvertently retains the storage of a variable, thereby inferring latched behavior that was not intended. This circumstance can occur in decision statements in one of two ways:

1. **Incomplete decision statements** — A *decision statement* is incomplete if it does not have a decision branch for every possible value of the decision expression. Incomplete decision statements can occur with either if-else and case statements. The following example statement illustrates an incomplete case decision statement:

```
always_comb begin // 3-to-1 multiplexor
  case (select)
    2'b00: y = a;
    2'b01: y = b;
    2'b10: y = c;
  endcase
end
```

The decision branches in the 3-to-1 MUX are complete as far as RTL behavior is concerned — the MUX correctly selects one of its three inputs — but the decision is not complete from a synthesis perspective because the select value of 2'b11 is not used.

2. **Incomplete decision branches** — A *decision branch* is incomplete if it does not make assignments to all of the variables that are outputs of the procedure.

```
always_comb begin // add or subtract
  case (mode)
    1'b0: add_result      = a + b;
    1'b1: subtract_result = a - b;
  endcase
end
```

If the first branch of this case statement is executed, only the `add_result` variable is updated. The `subtract_result` variable retains its previous value. Conversely, if the second branch is executed, only the `subtract_result` variable is updated, and the `add_result` variable retains its previous value.

The two ways in which inadvertent latches are inferred can occur anywhere in RTL models where if-else or case decision statements are used. Design engineers need to be continually vigilant when writing RTL models to fully specify decision statements.

Inadvertent latches in state machine models. State machines where the number of states is not a power of 2 do not use all the bits of the state variable, and therefore have the potential of the case statement evaluating, and no branch being executed. The output variables for the procedure are not updated, and retain their previous value. A 5-state FSM will have at least 3 unused values.

```
always_comb begin
  case (current_state)      // 5 states, binary count encoding
    3'b000: control_bus = 4'b0000;
    3'b001: control_bus = 4'b1010;
    3'b011: control_bus = 4'b1110;
    3'b100: control_bus = 4'b0110;
    3'b101: control_bus = 4'b0101;
  endcase
end
```

State machines that use one-hot encoding might also infer latches. When the always procedure triggers, if the state variable has a non one-hot value (no bits set or multiple bits set), then no branch will be executed. For example:

```
always_comb begin
    case (1'b1)          // 5 states, one-hot encoding
        current_state[0]: control_bus = 4'b0110;
        current_state[1]: control_bus = 4'b1010;
        current_state[2]: control_bus = 4'b1110;
        current_state[3]: control_bus = 4'b0110;
        current_state[4]: control_bus = 4'b0101;
    endcase
end
```

In both of these previous examples, synthesis compilers will add latches to match the simulation behavior of value retention. *Synthesis compilers are doing the right thing.* Inserting gate-level latches ensures that the gate-level ASIC or FPGA behavior is the same as the RTL behavior that was verified in simulation.

9.3 Avoiding latches in intentionally incomplete decisions

There are times when the design engineer knows (or at least assumes) something about the design that the synthesis tool cannot see by examining the decision statement. In the 3-to-1 MUX example, the designer may know (or assume) that the design will never generate a select value of 2'b11 and, therefore, the stored value of the `y` variable will never be needed. In the one-hot state decoder example, the designer may know (or assume) that the design will never produce a non one-hot value and, therefore, the `control_bus` will never need to retain a previous value.

A synthesis compiler can only see that not all possible values of the case expression were decoded. Synthesis sees a potential of the decision statement being evaluated, and no branch taken. This would result in the variables assigned in the decision statement not being updated.

When an incomplete decision statement is appropriate for the design functionality, the design engineer needs to let synthesis compilers know that the unspecified decision expression values can be ignored. There are several ways to tell synthesis that all values used by the decision statement have been specified, and, therefore, latches are not needed. Five common coding styles are:

1. Use a `default` case item within the case statement that assigns known output values (discussed in section 9.3.3, page 335).
2. Use a pre-case assignment before the case statement that assigns known output values (see section 9.3.4, page 338).
3. Use the `unique` and `priority` decision modifiers (section 9.3.5, page 340).
4. Use the obsolete — and dangerous — `full_case` synthesis pragma (section 9.3.6, page 345).
5. Use an X assignment value to indicate “don’t care” conditions (section 9.3.6, page 345).

Each coding style has advantages and disadvantages. These styles and their pros and cons are discussed in the following subsections.

Best synthesis Quality of Results (QoR) will most often be achieved using either coding style 1 (a default case item within the case statement that assigns known values), or coding style 2 (a pre-case assignment before the case statement that assigns known values).

9.3.1 Latch avoidance coding style trade-offs

Before examining the five latch avoidance coding styles in more detail, it is important to understand some of the engineering trade-offs these styles require.

Trade-off 1: Fully implemented decision versus logic-reduced decision. A fully implemented decision has gate-level decoding logic for all possible decision conditions, including conditions that are not normally used in the design. The 3-to-1 MUX example shown in at the beginning of section 9.2 (page 327) only uses the select values of $2^{'b}00$, $2^{'b}01$ and $2^{'b}10$. The value of $2^{'b}11$ is not used by the design, but a fully implemented decision statement would also decode this value, and produce an output value determined by the design engineer.

A logic-reduced decision will remove any gates that would have been used to decode decision conditions that are not used. Synthesis compilers will utilize techniques adapted from various well-known logic reduction algorithms, such Karnaugh mapping, Quine-McCluskey logic minimization, and Espresso logic minimization. An obvious advantage of logic reduction is that a design will require fewer gates in a target ASIC or FPGA. The reduced logic can potentially shorten the propagation paths, possibly making the combinational logic able to meet flip-flop setup times in critical timing paths.

There is an important trade-off to logic reduction that removes the gates to decode conditions that should not occur. However, unpredictable or undesired behavior can occur in the ASIC or FPGA, should a hardware glitch occur that causes an unexpected value on the decision condition for which the gates have been removed. This could result in an ASIC or FPGA that does not work under all conditions.

A design will be more robust when decision statements are fully specified, even for logic values that are not used by the design functionality. The design will be better able to handle unexpected conditions that might not have been simulated, such as power-on glitches or glitches resulting from interference.

Design technology in the 20th century often required aggressive logic reduction techniques. ASICs and FPGAs were more limited for the total number of gates, and the propagation delays through combinational logic gates were more substantial. These limitations are much less of a concern in modern ASICs and FPGAs. The additional logic required to fully implement a decision statement, and the minimal additional propagation delay for those additional gates, is seldom an issue.

Coding guidelines that use logic reduction coding styles for decision statements are based on old-school techniques that were a best-practice coding style many years ago, but are seldom necessary today.

Best Practice Guideline 9-3

Fully specify the output values of decision statements to avoid unintended latches. Do not use logic reduction optimizations to avoid latches, unless needed for a specific circumstance.

Latch avoidance coding styles 1 and 2 both fully implement decision statements, which is a better style for most designs targeting modern ASICs and FPGAs. These two styles are discussed in sections 9.3.3 (page 335) and 9.3.4 (page 338), respectively. The choice between these styles is mostly a personal preference.

The other three latch-avoidance coding styles use logic minimization techniques, which have potential risks that might lead to ASIC or FPGA implementations that are not as robust. The logic reduction coding styles should only be used when gate counts need to be reduced in order to fit the design in a specific target device, or to meet flip-flop setup times in a critical timing path.

Trade-off 2: Run-time error trapping versus X-value propagation. Coding styles 3, 4 and 5 listed in section 9.3 (page 329) help avoid unintentional latches by using logic reduction for unused decision conditions. The three coding styles yield nearly identical synthesis Quality of Results (QoR), but simulate very differently.

Coding style 3 uses either the **unique** or **priority** decision modifier. The effects of these modifiers are discussed in section 9.3.5 (page 340). In brief, these modifiers inform synthesis to apply logic reduction algorithms, and—importantly—enable run-time simulation checking. A violation message will be reported if a decision statement is entered, and the decision condition does not match any branch. This additional run-time checking can help ensure that the design will work correctly with gate-level logic reduction for unused decision conditions.

Coding style 4 assigns an X value to the procedure outputs whenever the decision condition does not match any branch. The propagation of X values should cause verification tests to fail, indicating that a problem has occurred somewhere in the design. Using X value assignments was considered a best-practice for avoiding latches in the 1990s and early 2000s, but this technique is now out-of-date.

Best Practice Guideline 9-4

Use the **unique** or **priority** decision modifiers if gate-level logic reduction is needed for avoiding unintended latches. Do not use the antiquated Verilog-2001 coding style of X value assignments.

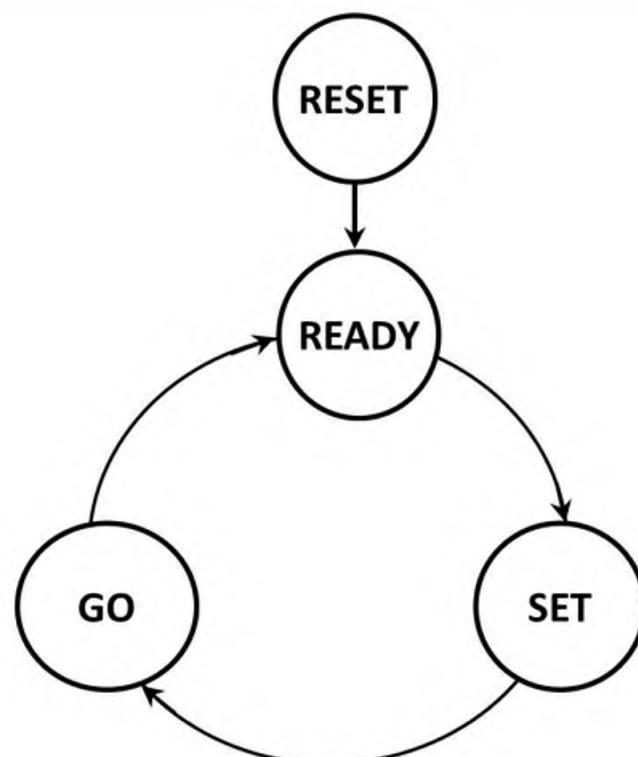
Using logic reduction for unused decision conditions is no longer a best-practice. The speed and capacity of modern ASICs and FPGAs can accommodate fully implementing all possible decision expression values, including values that the design does not use. There may be rare exceptions where gate count or critical timing paths are a concern, and applying logic reduction techniques are needed. The old coding style of using X value assignments for logic reduction was adequate for the design sizes in the 20th century, but this coding style has many disadvantages when used with modern design sizes and software tools. Section 9.3.6 (page 345) discusses these disadvantages.

There are old-school Verilog design engineers who still prefer to aggressively use logic reduction techniques with incomplete decision statements, and who prefer using X value assignments to achieve this logic reduction, rather than the modern **unique** or **priority** decision modifiers. Just because using X value assignments was the best-practice for the designs in the 1990s, however, does not mean it is still the best way.

9.3.2 A small example to illustrate avoiding unintentional latches

A simple Finite State Machine (FSM) is used in this chapter to illustrate where unintentional latches can occur, and various coding styles to prevent these latches. For simplicity, the state machine flow is a simple round-robin, with no branching forward or backward. Figure 9-2 shows the state flow of this simple state machine.

Figure 9-2: Round-robin Finite State Machine state flow



A combinational logic block is used to decode the current state of the state machine in order to determine the next state. The state values use a partial 3-bit Johnson Count encoding, where a value of 1 is shifted into the state register for each subsequent state.

The code for the next state decoder is:

```
typedef enum logic [2:0] {RESET = 3'b000, // Johnson Count
                           READY = 3'b001,
                           SET   = 3'b011,
                           GO    = 3'b111} states_t;

states_t current_state, next_state;

always_comb begin
  case (current_state)
    RESET  : next_state = READY;
    READY   : next_state = SET;
    SET     : next_state = GO;
    GO      : next_state = READY;
  endcase
end
```

This next state decoder functions correctly in simulation, but synthesis will infer latches. Synthesis sees `current_state` as a 3-bit vector, which can have 8 possible values, but the `case` statement only decodes 4 of those 8 values. Synthesis infers latches for the `next_state` bits because of the possibility that the `always` procedure could be entered, and `current_state` having a value other than the 4 values that are decoded. If this should occur, the `next_state` variable would not be updated.

Synthesis warnings with `always_comb`. An advantage of using `always_comb` for modeling combinational logic is that lint checkers and synthesis compilers can know that the designer's intent is to represent combinational logic. Since latches are inferred by the next state decoder in this example, software tools can generate an error or warning message. For example, the Mentor Graphics Precision Synthesis compiler reports:

```
Warning: Latch inferred for net next_state[2:0] inside
always_comb block
```

NOTE

Some synthesis compilers report latch inference in an `always_comb` procedure as a warning. *This is should be an error!* The design engineer has indicated an intent to have combinational logic. If a latch is inferred, there is a mistake in the code. Synthesis compilation should abort with an error, rather than issue a warning that might be overlooked or treated as non-critical.

Had this next state decoder been modeled using the traditional Verilog general purpose `always` procedure — which can be used to model either combinational logic and latch logic — synthesis compilers might assume the designer intended to model latch behavior, and not generate a warning message to indicate that latched logic was inferred. (Some synthesis compilers might have an option to enable latch inference warnings from the general purpose `always` procedure.)

Simple FSM code and synthesis results. Example 9-2 shows the full context for this simple state machine. Figure 9-3 shows the results of synthesizing this example. Observe the latches in the schematic that were inferred for the `next_state` bits.

Example 9-2: Simple round-robin state machine that will infer latches

```

module simple_fsm
(input logic clk, rstN,
 output logic get_ready, get_set, get_going
);

typedef enum logic [2:0] {RESET = 3'b000, // Johnson Count
                           READY = 3'b001,
                           SET   = 3'b011,
                           GO    = 3'b111} states_t;

states_t current_state, next_state;

// state sequencer
always_ff @(posedge clk or negedge rstN) // async reset
  if (!rstN) current_state <= RESET; // active-low reset
  else      current_state <= next_state;

// next state decoder
always_comb begin
  case (current_state)
    RESET : next_state = READY;
    READY  : next_state = SET;
    SET    : next_state = GO;
    GO     : next_state = READY;
  endcase

```

```

end

// output decoder (using pre-case assignment)
always_comb begin
  {get_ready, get_set, get_going} = 3'b000; // clear bits
  case (current_state)
    READY  : get_ready = '1;
    SET    : get_set   = '1;
    GO     : get_going = '1;
  endcase

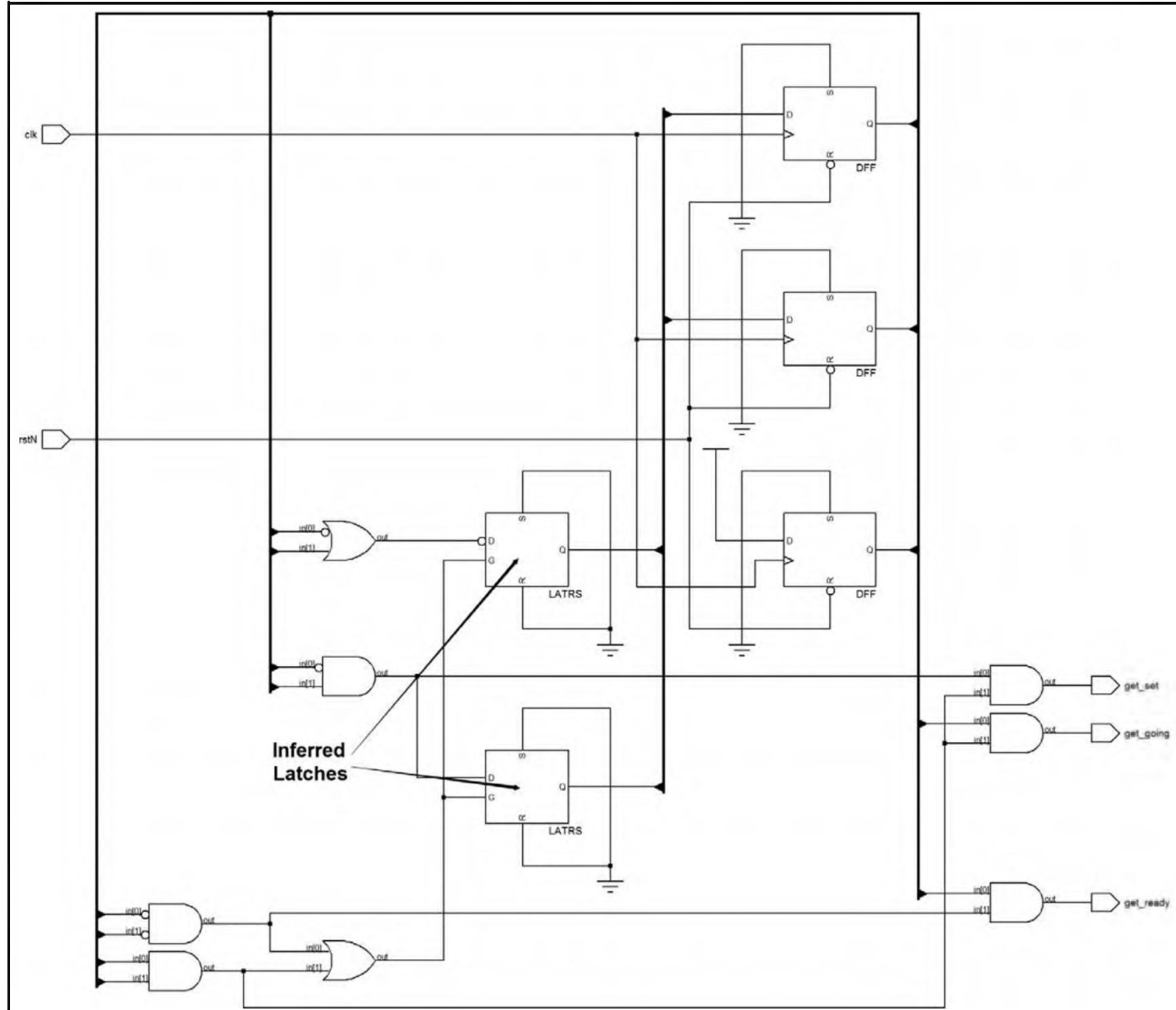
```

```

end
endmodule: simple_fsm

```

Figure 9-3: Synthesis result for Example 9-2: FSM with unintended latches



Technology independent schematic (no target ASIC or FPGA selected)

The simple FSM in Example 9-2 is functionally correct — the RTL works as intended, and the synthesized implementation matches the RTL functionality. The state encoding does not use all possible values of the `current_state` vector, and therefore the next state decoder should not need to decode these unused values.

The following sections present several ways to let synthesis compilers know that some `current_state` values are not used, and therefore no latches should be inferred.

9.3.3 Latch avoidance style 1 — Default case item with known values

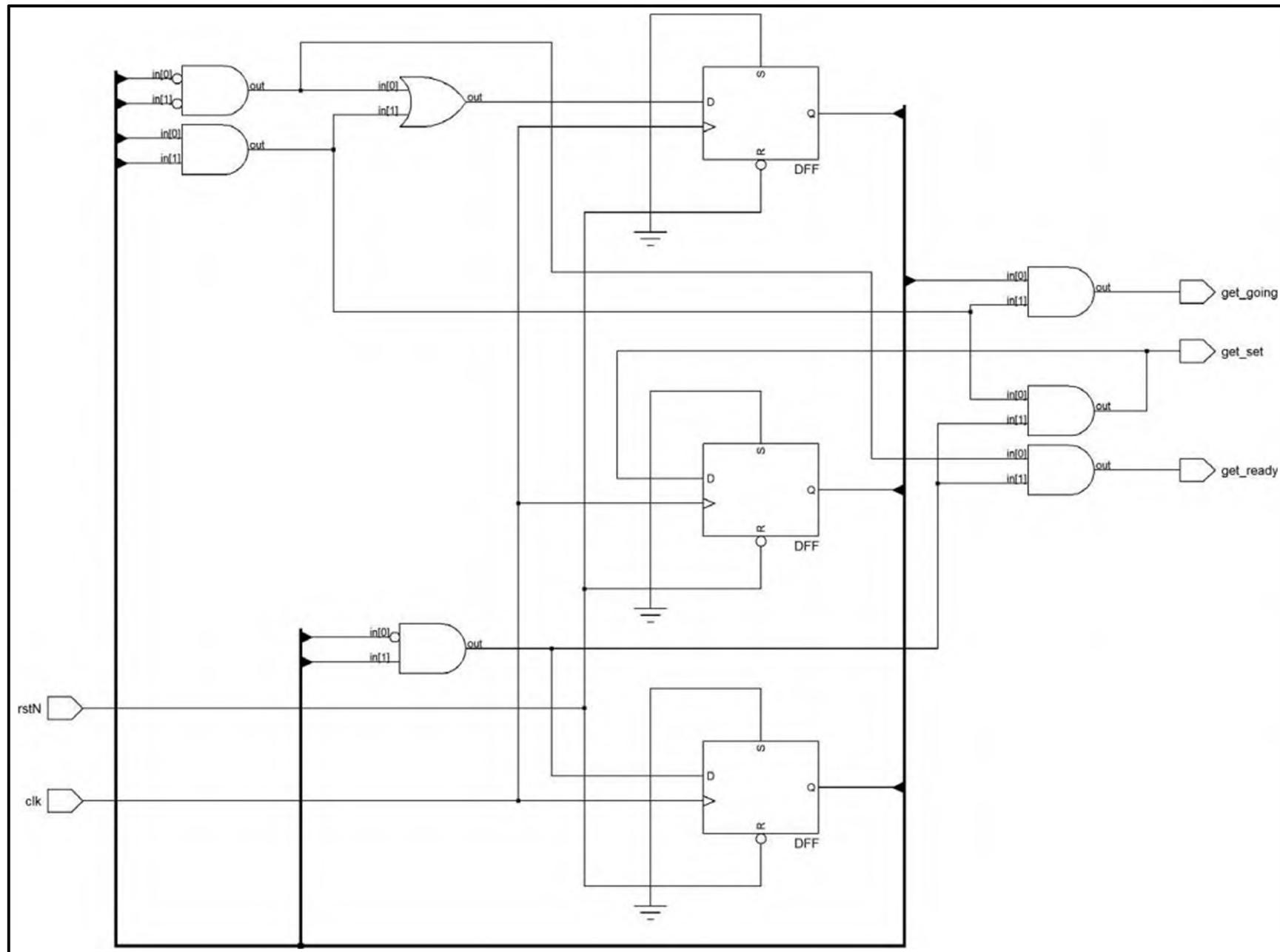
With case statements, a *case expression*, which is in parentheses, is compared to one or more *case items*. One coding style to prevent unintentional latches from a case statement is to ensure that there is a case item for every possible value of the case expression. Using a default case item, specified using the `default` keyword, ensures that every possible value of the case expression is decoded.

For the next state decoder in Example 9-2, one of the `next_state` values can be used as a default value, should a value of `current_state` occur that is not expected.

```
always_comb begin
    case (current_state)
        RESET  : next_state = READY;
        READY   : next_state = SET;
        SET     : next_state = GO;
        GO      : next_state = READY;
        default: next_state = RESET; // reset if error
    endcase
end
```

Figure 9-4 shows the results of synthesizing the same simple state machine, but with the default case item shown in the code snippet above. Observe that no latches are inferred for `next_state`.

Figure 9-4: Synthesis result when using a default case item to prevent latches



Technology independent schematic (no target ASIC or FPGA selected)

Pros and cons of using a default case item with case statements. Using a case default branch ensures that all values of the case expression are decoded — even values that should never occur. Should an unexpected case expression value occur, a defined action will be taken. The gate-level implementation will be robust, and can handle circumstances that were not anticipated in the RTL model. In the Johnson Count state encoding example used in this section, `current_state` should never

have values of 010, 100, 101 or 110. But unexpected values can sometimes occur in actual silicon. A chip might power up with a state value that is not used, or some EMF interference could cause a glitch that results in a momentary state value that is not used. The coding style of using a default branch that assigns a known value means these unexpected conditions are decoded and handled in a defined way.

One disadvantage of using a default branch that assigns a known value is that this style only addresses one cause of a latch being inferred — an incomplete case statement. A latch could still be inferred if every branch of the case statement, including the default branch, does not assign to the same variables.

A second disadvantage of using a default branch that assigns a known value is that extra gate-level circuitry is required to decode the case expression values that are not used by the design. A 16-state one-hot state machine requires a 16-bit vector, which has 65,536 possible values (2^{16}), of which only 16 are needed by the design. Decoding the remaining 65,520 values can require many more logic gates, which, if everything is working as expected, will never be used. These additional gates could have a negative effect on the area, speed, and power consumption of the ASIC or FPGA.

These extra gates are usually not a problem. Most modern ASICs and FPGAs have ample capacity and speed to handle the extra gates. Furthermore, many synthesis compilers have special FSM optimization algorithms that perform a reachability analysis, and minimize the effects of the extra logic needed to decode all possible values of a case expression.

NOTE

A default branch does not guarantee that latches will not be inferred. Even with a fully-specified case statement, a latch might be inferred if a non-clocked procedural block is entered, and there is a possibility that one or more variables will not be updated. If the combinational logic procedure does not have any pre-case assignments (see section 9.3.4), then every branch of a case statement must make assignments to the same variables, including the default branch.

An alternate style is to assign an X value for the default branch, instead of a known value. This style has very different behavior for both simulation and synthesis, and is discussed separately, in section 9.3.6 (page 345).

9.3.4 Latch avoidance style 2—Pre-case assignment, known values

Latch behavior is inferred whenever a non-clocked always procedure is entered, and it is possible that one or more variables that are outputs of the procedure will not be updated. A latch will not be inferred from an incomplete decision statement, so long as the variables assigned within the decision statement are updated before the always procedure completes its loop.

The following code snippet will not infer a latch from the incomplete case statement because a pre-case assignment to the `get_ready`, `get_set` and `get_going` variables is made when the always procedure is triggered. This unconditional assignment before the decision statement ensures these variables are updated every time the always procedure is entered.

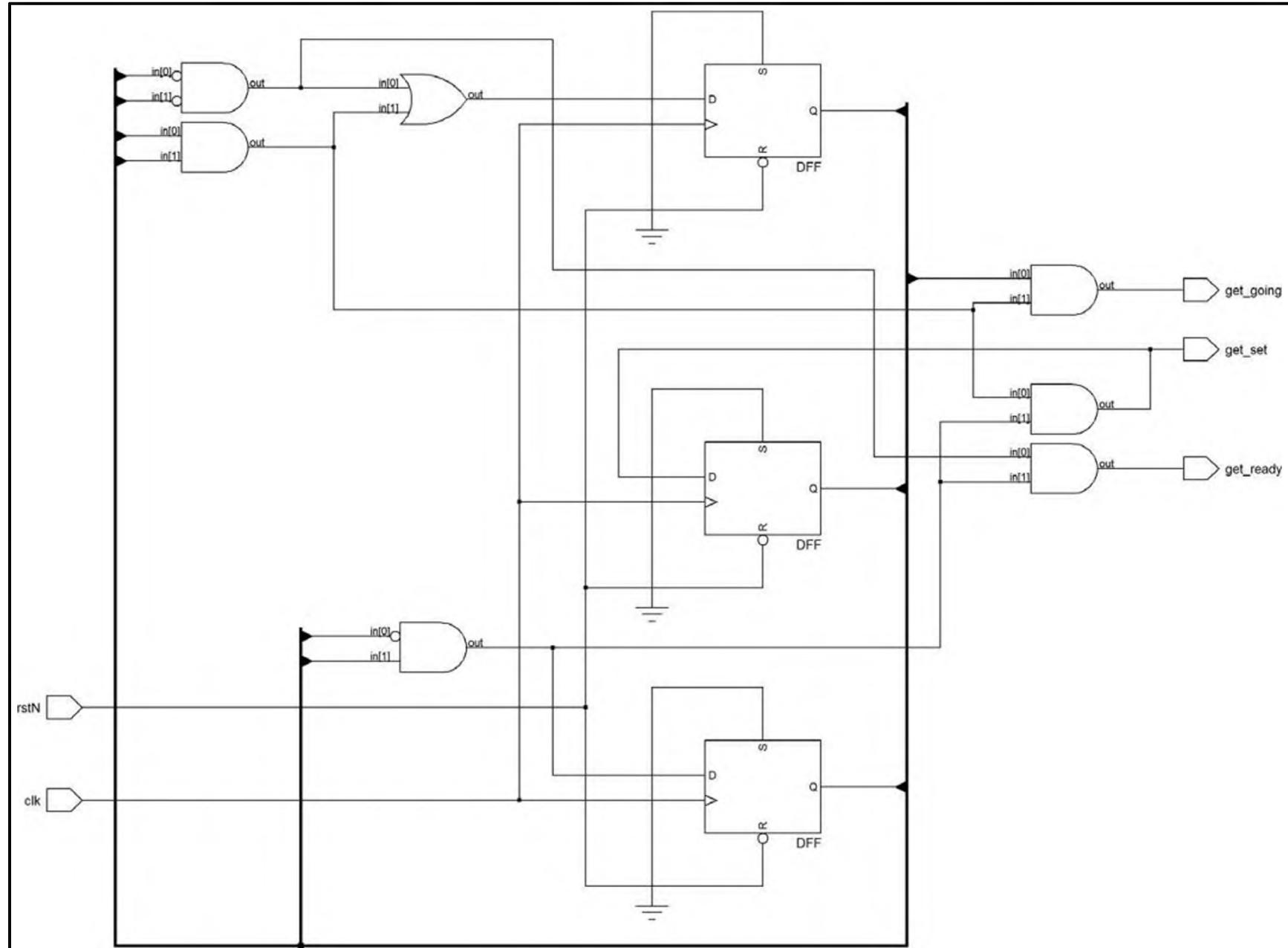
```
always_comb begin
    {get_ready, get_set, get_going} = '0; // clear all bits
    case (current_state)
        READY: get_ready = '1;
        SET   : get_set   = '1;
        GO    : get_going = '1;
    endcase
end
```

The next-state decoder can be coded in a similar way:

```
always_comb begin
    next_state = RESET; // default to reset if invalid state
    case (current_state)
        RESET  : next_state = READY;
        READY  : next_state = SET;
        SET    : next_state = GO;
        GO     : next_state = READY;
    endcase
end
```

Figure 9-5 shows the results of synthesizing the same simple state machine shown in Example 9-2 (page 334), but with the a pre-case assignment before the incomplete case statement. Observe that no latches are inferred for `next_state`.

Figure 9-5: Synthesis result using a pre-case assignment to prevent latches



Technology independent schematic (no target ASIC or FPGA selected)

The default assignment of known values within a case statement (style 1) and the pre-case assignment (style 2) produce similar synthesis Quality of Results (QoR). This can be seen by comparing Figure 9-5 with Figure 9-4 (page 336).

Pros and cons of pre-case assignments before decision statements. One advantage of having a pre-case assignment is that designers do not need to be concerned about ensuring that decision statements are complete in the rest of the procedure. Another advantage is that the decision statements can focus on which combinational outputs are significant for each branch of the decision. In the code snippet above, it is obvious that the READY state affects the get_ready output, SET affects the get_set output, and so forth.

A disadvantage of this coding style is it can be more difficult to see what is assigned to all the variables used by the procedure by looking at a specific case item branch. It is necessary to look at both the pre-case assignment and the case item assignments to see all the values assigned. In a large, complex decoder, these assignments could be separated by many lines of code.

An alternate coding style is to assign an X value for the pre-case assignment, instead of a known value. This style has important trade-offs on design quality and robustness, which are discussed in Section 9.3.6 (page 345).

9.3.5 Latch avoidance style 3 — unique and priority decision modifiers

Section 9.3.3 (page 335) used a default case item to avoid inferring unintentional latches in the next state decoder. Section 9.3.4 (page 338) accomplished latch avoidance by having a pre-case assignment prior to the decision statement. Both coding styles behave as a fully specified decision in RTL simulations. Synthesis will implement decoder logic for all case item values, including values that are not used by the design. This is a safe coding style. Should a glitch or some other circumstance cause an unexpected value on the case expression signal, the additional logic gates will decode the value and perform as specified — and verified — in the RTL default case item assignment or the pre-case assignment.

The disadvantage of these styles is that, if the unspecified values never occur, the ASIC or FPGA contains logic gates and propagation paths that are never used. These additional gates and propagation paths can make the IC larger, slower and less power efficient. The functionality to decode values that never occur can be costly for designs that push clock speeds or device gate count to their limits, or need to reduce power consumption as much as possible.

SystemVerilog has three decision modifiers, **unique**, **unique0** and **priority**, that can enable certain gate-level reduction optimizations during the synthesis process. These decision modifiers are specified immediately before the **case**, **case-inside**, **casez** or **casex** keyword, as in:

```
always_comb begin
  unique case (current_state)
    RESET   : next_state = READY;
    READY   : next_state = SET;
    SET     : next_state = GO;
    GO     : next_state = READY;
  endcase
end
```

These decision modifiers can also be specified with an if-else-if decision series:

```
always_comb begin
  unique if (opcode == 2'b00) y = a + b;
  else if (opcode == 2'b01) y = a - b;
  else if (opcode == 2'b10) y = a * b;
end
```

These decision modifiers affect both simulation and how synthesis compilers translate RTL code into a gate-level implementation.

9.3.5.1 The `unique` decision modifier

Simulation. The `unique` decision modifier enables run-time checking for two conditions:

1. A case item or if-else condition has been specified for all values that actually occur during simulation. This means a decision branch is executed for every case expression value that occurs during simulation.
2. There are never multiple decision branches true at the same time, meaning each case item decodes a value that is unique from every other case item value.

Synthesis. The `unique` modifier instructs synthesis to perform two types of optimizations:

1. Treat the decision statement as fully specified (sometimes referred to as *full case*), and perform appropriate logic reduction optimizations.
2. Treat the decision statement conditions as mutually exclusive, meaning there will never be multiple conditions true at the same time (sometimes referred to as *parallel case*), and perform optimization to evaluate the decision conditions in parallel, instead of with priority encoded logic.

Using `unique` is appropriate for functionality such as a one-hot state decoder, where all one-hot values need to be decoded, and any value that is not one-hot should never occur. Nor should multiple case items decode the same one-hot value.

9.3.5.2 The `unique0` decision modifier

Simulation. The `unique0` decision modifier enables run-time simulation checking for one condition: There are never multiple decision branches true at the same time, meaning each case item decodes a value that is unique from every other case item.

Synthesis. •The `unique0` modifier instructs synthesis to treat the decision statement conditions as mutually exclusive (*parallel case*), and perform optimizations to evaluate the decision conditions in parallel, instead of with priority encoded logic.

NOTE

At the time this book was written, some simulators and most synthesis compilers did not support the `unique0` decision modifier.

Best Practice Guideline 9-5

Use the `unique` decision modifier in RTL models when logic reduction is desirable to prevent inferred latches. Do not use the `unique0` modifier.

The **unique0** decision modifier does not help prevent unintentional latches, since it does not instruct synthesis to treat a decision as being fully specified.

An appropriate usage of the **unique0** decision modifier is when a decision statement will not infer latches, but a synthesis cannot recognize that the case items can be evaluated in parallel. The following code snippet illustrates this circumstance.

```

typedef enum logic [2:0] {A = 3'b100,
                           B = 3'b010,
                           C = 3'b001} modes_t;
modes_t selector_switch;

always_comb begin
    control = 4'h1;      // assume switch is in A position
    unique0 case (selector_switch) inside
        4'b?1?: control = 4'h2; // switch is in B position
        4'b??1: control = 4'h3; // switch is in C position
    endcase
end

```

In this example, no latch will be inferred because of the pre-case assignment to the `control` output. However, synthesis compilers will not recognize that the case item values can be evaluated in parallel, because the wildcard bits could potentially allow two or more case items to be true at the same time. Adding the **unique0** modifier tells synthesis that `selector_switch` will always have just one bit set, and therefore can be evaluated in parallel.

The **unique** decision modifier would not be appropriate in this example. The **unique** modifier would inform synthesis that the case statement was complete. Synthesis might interpret this to mean that the only values `control` can have are those assigned within the case statement, which are the values of 2 and 3. The pre-case assignment would be ignored, and no gate-level logic would be implemented to generate a `control` value of 1.

In simulation, a **unique** decision statement will generate a run-time simulation violation report any time the case statement is evaluated and no branch is taken. A violation message would occur whenever the `selector_switch` has a value of A. *This is an important violation message!* It is saying that the case statement is not decoding all values that occur, and therefore should not be synthesized as if it were a complete case statement.

The previous code snippet could have been coded as a reverse case statement where the use of the **unique** decision modifier would be appropriate, as shown Chapter 8, section 8.2.5 (page 313). The `selector_switch` decoding might also have been coded without using wildcard bits in the case items, so that synthesis compilers could recognize that the case item values are unique, and could be automatically optimized as parallel decoding logic, without having to use a decision modifier.

9.3.5.3 The priority decision modifier

Simulation. The **priority** decision modifier enables run-time checking for one condition:

A case item or if-else condition has been specified for all values that actually occur during simulation. This means a decision branch is executed for every case expression value that occurs during simulation.

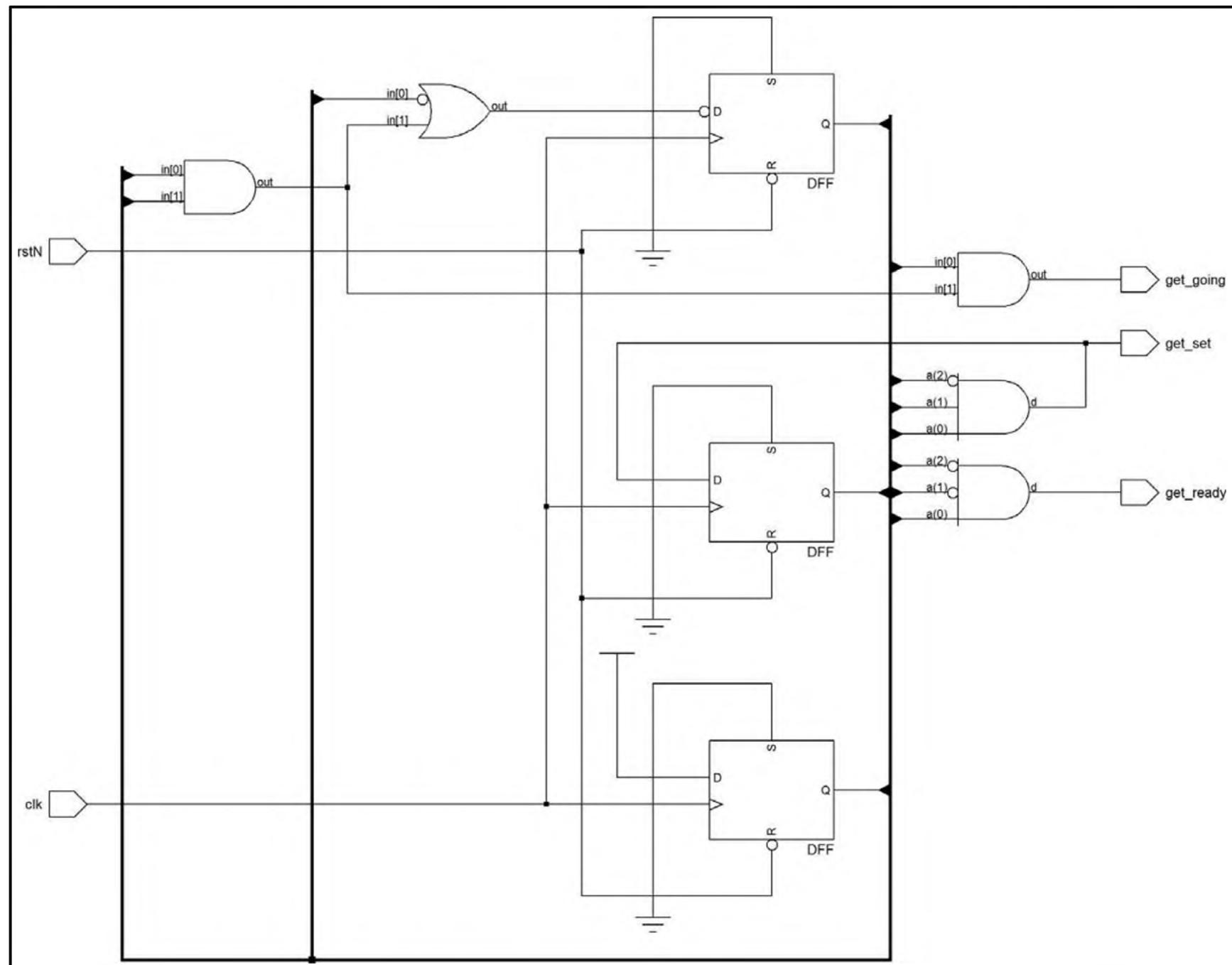
Synthesis. The **priority** modifier instructs synthesis to treat the decision statement as fully specified (*full case*), and perform appropriate logic reduction optimizations.

Using **priority** is appropriate for functionality such as a priority interrupt decoder, when there is a possibility that more than one branch of the decision could be true at the same time, and the highest-priority interrupt should be serviced first. Priority encoded behavior is discussed in more detail in Chapter 7, section 7.4 (page 265).

9.3.5.4 Decision modifier example

Figure 9-6 shows the results of synthesizing the simple state machine shown in Example 9-2 (page 334), but with a **unique case** statement.

Figure 9-6: Synthesis result when using a unique case statement to prevent latches



Technology independent schematic (no target ASIC or FPGA selected)

The next state decoding logic in this small state machine example is too simple to benefit from the logic reduction optimizations that are enabled by the **unique** decision statement modifier. Even so, some gate-level logic reduction can be seen by comparing Figure 9-6 above, with the fully specified decision statement examples shown in Figures 9-4 (page 336) and 9-5 (page 339).

Pros and cons of decision modifiers. The primary advantage of the **unique** and **priority** decision modifiers is the optimization of the gate-level implementation of the RTL functionality. The logic reduction optimizations triggered by the **unique** and **priority** modifiers can result in smaller, faster designs that consume less power. A secondary advantage of the **unique** and **priority** modifiers is that they can prevent unintentional inferred latches from an incomplete case statement, so long as the variables that are assigned values in the combinational always procedure are updated every time the procedure executes.

A disadvantage of the **unique** and **priority** modifiers is that the gate-level logic reduction optimizations can lead to a gate-level implementation that is not robust, and can have unpredictable or undesired behavior, should a hardware glitch occur that causes an unexpected value on the case expression. This could result in an ASIC or FPGA that does not work under all conditions. The run-time simulation checking that is part of the **unique** and **priority** modifiers can help reduce the risks of logic reduction optimization by verifying that the expression values not decoded never occur. This verification, however, is only as effective as the test stimulus.

Best Practice Guideline 9-6

In general, fully specify all case statements using either a default case item that assigns known values, or a pre-case assignment with known values. An exception to this guideline is a one-hot state decoder using a reverse case statement.

Only use the **unique** or **priority** decision modifiers if it is certain that the gate-level logic reduction optimization is desirable. An exception to this guideline is a reverse case statement, where using a **unique** or **priority** case statement is the preferred way to avoid unintentional latches.

When decision statements are fully specified, a design will be more predictable for unexpected conditions behave, such as power-on glitches or glitches resulting from interference. A fully specified case statement with known values assigned for all possible conditions can be important when designing fault-tolerant functionality. Fault tolerance affects many aspects of a design, not just decision statements. This is a general engineering topic that is outside the scope of this book.

In the 1980s and 1990s, it was much more important to take every advantage of gate-level minimization techniques in order to fit designs into the capacity and speed of the ASICs and FPGAs available. This is not as critical with today's ASIC and

FPGA technologies. Most designs will fit and run at the desired speed without concern for gate-level logic reduction, with its associated risks.

NOTE

A **unique case** or **priority case** does not guarantee that latches will not be inferred. A latch will be inferred any time there is a possibility that a non-clocked procedural block can be entered and a variable is not updated.

If the procedure does not have any pre-case assignments (see section 9.3.4, page 338), then every branch of a case statement must make assignments to the same variables in order to avoid unintentional latches (see section 9.2 (page 327) for ways in which a latch might be inferred).

9.3.6 Latch avoidance style 4 — X assignments for unused decision values

Best Practice Guideline 9-7

Use the **unique** or **priority** decision modifiers instead of an X value assignment for unused decision values.

X value assignments were a best-practice coding style from traditional Verilog days, before SystemVerilog. The SystemVerilog **unique** or **priority** decision modifiers instruct synthesis to perform the same gate-level logic reduction optimizations as an X value assignment, but also add built-in checks to at least partially verify the gate-level optimizations.

Traditional Verilog did not have the **unique** and **priority** decision modifiers discussed in section 9.3.5 (page 340). Instead, design engineers using Verilog-2001 would assign an X value to the procedure output variables to inform synthesis that any value not explicitly decoded in the case statement could be ignored.

The X value assignment could be either a default case item or a pre-case assignment, as shown in the following two code snippets:

```
always_comb begin
    case (current_state)
        RESET  : next_state = READY;
        READY   : next_state = SET;
        SET     : next_state = GO;
        GO      : next_state = READY;
        default: next_state = states_t('x); //don't care branch
    endcase
end
```

```

always_comb begin
    next_state = states_t'('x); //case stmt should clear X's
    case (current_state)
        RESET   : next_state = READY;
        READY   : next_state = SET;
        SET     : next_state = GO;
        GO     : next_state = READY;
    endcase
end

```

Observe that, when assigning to an enumerated variable, only labels in the enumerated definition can be directly assigned. The cast operator used in the examples above overrides this restriction and forces the enumerated variable to an X value. An alternate way to do this is to add another label to the enumerated type definition that has an X value, and assign that label in the default branch. For example:

```

typedef enum logic [2:0] {RESET = 3'b000, // Johnson Count
                           READY = 3'b001,
                           SET   = 3'b011,
                           GO    = 3'b111,
                           XXX   = 3'bXXXX} states_t;

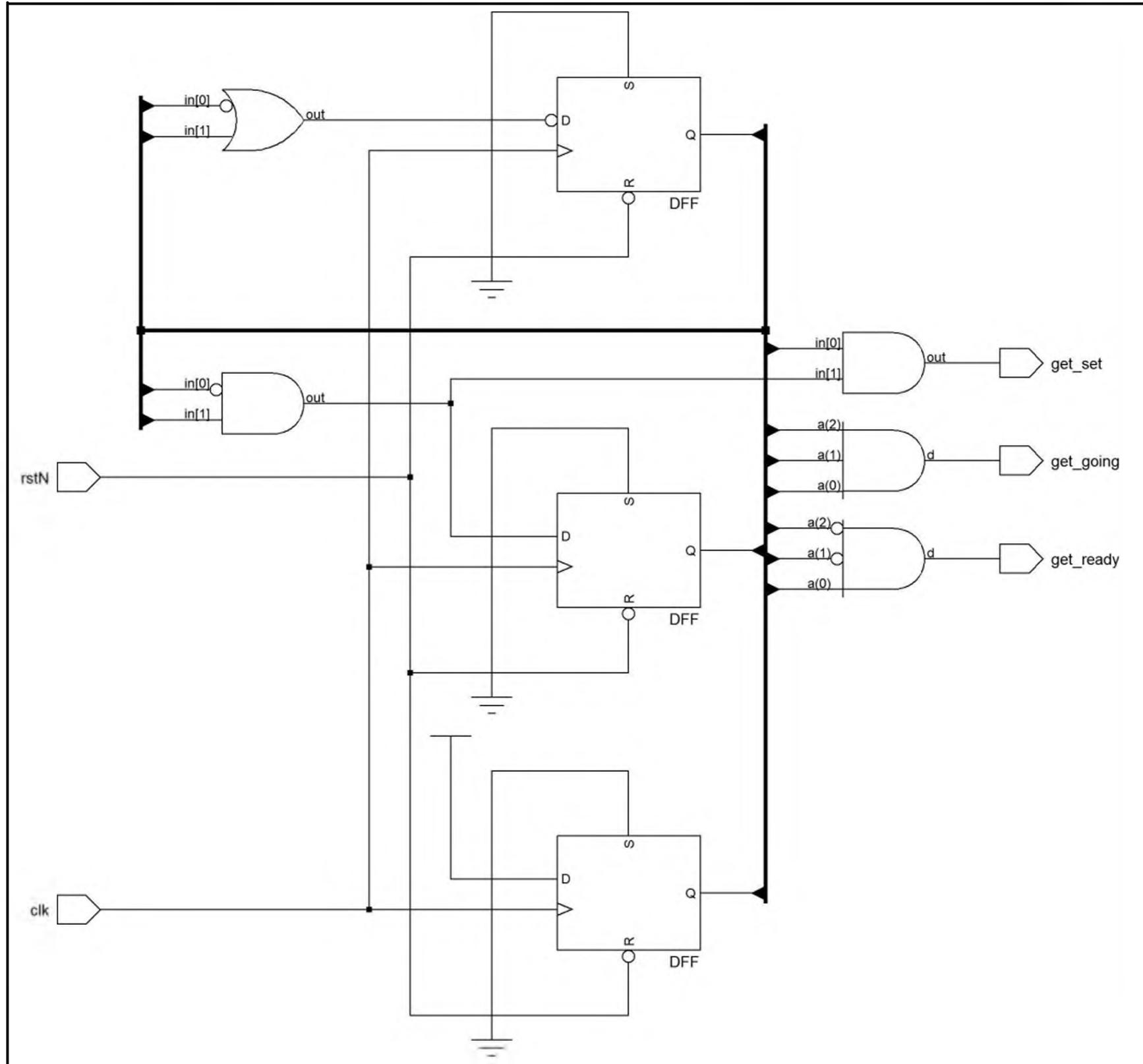
always_comb begin
    case (current_state)
        RESET   : next_state = READY;
        READY   : next_state = SET;
        SET     : next_state = GO;
        GO     : next_state = READY;
        default: next_state = XXX; // don't care branch
    endcase
end

```

Assigning a value of X as a default affects simulation and synthesis differently:

- **Simulators** will propagate an X onto the output variable(s) if an unexpected decision value that is not decoded should occur. In the example above, a current_state value of 3'b010, 3'b100, 3'b101 or 3'b110 will result in a next_state value of 3'bXXX. This X value can be caught in verification, and traced back to the unexpected value of current_state.
- **Synthesis compilers** treat a default assignment of an X value as a special flag to indicate that decision values that were not explicitly decoded in the decision statement are not of interest, and can be ignored. Synthesis will apply logic reduction optimizations to minimize the gate-level implementation, so that the logic gates only decode logic for the values explicitly listed in the decision statement.

Figure 9-7 shows the results of synthesizing the state machine shown in Example 9-2 (page 334), but with a **default** case item that assigns an X value.

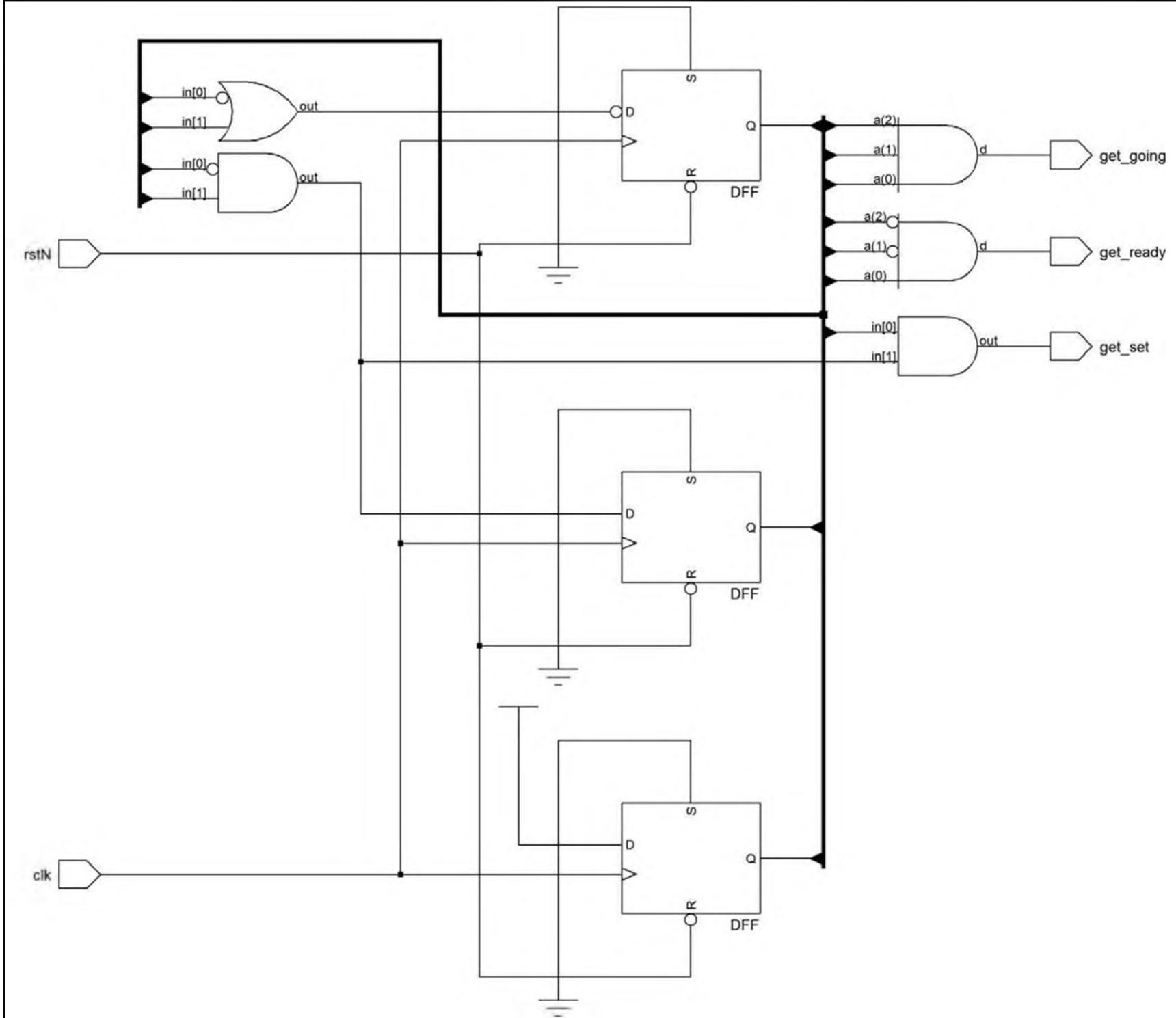
Figure 9-7: Synthesis result using a default case X assignment to prevent latches

Technology independent schematic (no target ASIC or FPGA selected)

The logic reduction optimizations by using an X value assignment, as seen in Figure 9-7, above, is nearly identical to the logic reduction achieved using the **unique** decision statement modifier shown in Figure 9-7 (page 347). There is a small variance in how the synthesis compiler rendered the two schematics, but the number of gates used to decode the next state are the same. Much larger state decoders might see minor differences in how unique versus an X value assignment optimize, but the difference should be negligible.

Figure 9-8 shows the results of using a pre-case assignment of an X value before the case statement.

Figure 9-8: Synthesis results when using a pre-case X assignment



Technology independent schematic (no target ASIC or FPGA selected)

The gate-level optimization for this simple next state decoder is the same with either the default X value assignment or the pre-case X value default assignment. There can be other advantages to using one style versus the other, however. The pros and cons of default case items versus pre-case assignments are discussed in sections 9.3.3 (page 335) and 9.3.4 (page 338).

Pros and cons of default assignment to an X value. One advantage of an X value assignment is the synthesis optimization of the gate-level implementation. In contrast to the default assignment or pre-case assignment of a known value, the optimizations triggered by an X value assignment can result in smaller, faster designs that consume less power.

A second possible advantage is that the X value can aid in design debugging. If, in simulation, an unexpected decision value occurs, an X value will propagate into the design. Debugging the cause of the X value can determine the cause of the unexpected case expression value, potentially finding a bug elsewhere in the design.

There are a number of significant disadvantages of the X value assignment coding style.

- Due to the gate-level optimizations performed by synthesis, there will be no definitive behavior in the ASIC or FPGA, should a hardware glitch occur and a value that is not decoded happens. This unpredictable implementation behavior could result in an ASIC or FPGA that does not work under all conditions. This is the same disadvantage that exists with using **unique** and **priority** decision modifiers, and is discussed in section 9.3.1 (page 330).
- X value assignment is problematic for RTL versus gate-level logic equivalence checking. A Logic Equivalence Checker (LEC) is an engineering tool that analyzes the functionality of two models to determine if the models are logically the same. The X values used in an RTL model will not be equivalent to the gate level circuitry. To work around this, engineers must designate portions of the design as a black box that is not fully analyzed by LEC.
- X value assignments can be problematic if the design flow uses a test methodology that is not X-tolerant, such as Built-in Self Test (BIST).
- X value assignments place a burden on the verification engineers to write verification code that will detect when an X value occurs in simulation. Detecting X values requires careful probing of internal design signals. When verification detects an X value, tedious, time-consuming debugging effort might be needed to trace the X value back through design logic and clock cycles to find the cause of the X value. Debugging the cause of an X value through many layers of logic and clock cycles is not an efficient verification technique for the large, complex designs of today.
- Depending on X values to indicate there is a problem with a decision statement can be risky for verification. RTL models can hide X values due to X-optimism, making the X values difficult to detect. It is also possible that an X value will go undetected due to X-optimism. Appendix C discusses the perils of X-optimism and how it can hide design problems in simulations.

Propagating and detecting X values as a means to verify that no unexpected case item values occurred is an obsolete coding style that was used in traditional Verilog. Some old-time Verilog design engineers have argued that they like X value propagation because they make waveforms “bleed red” (most simulator waveform tools display X values in red). The use of waveform displays to verify modern designs is not practical, however. Today’s designs have far too much functionality and run far too many clock cycles for using waveforms to trace an X value back through large amounts of code and clock cycles.

SystemVerilog replaces this antiquated traditional Verilog verification style of X value propagation with the **unique** and **priority** decision modifiers. These modifiers perform run-time checking during simulation, and report violation messages at a specific clock cycle and line of code when an unexpected condition occurs in a case statement. Rather than having to spend time debugging the source of an X value, these violation reports indicate that a problem occurred “right here, right now”.

9.3.7 Latch avoidance style 5 — the `full_case` synthesis pragma

Best Practice Guideline 9-8

Use the `unique` or `priority` decision modifiers if logic reduction optimizations are required. Do not use the `full_case` (or `parallel_case`) synthesis pragma. Ever.

If logic reduction from an incomplete decision statement is appropriate for the design, use the SystemVerilog `unique` or `priority` decision modifiers instead of the `full_case` synthesis pragma.

The `full_case` pragma is often seen in older Verilog RTL models or in examples found on the Internet. Therefore, the construct is described in this book, even though it should never be used.

Synthesis compilers provide a way for design engineers to hide synthesis-specific commands within comments in the RTL models. These tool-specific commands are referred to as *synthesis pragmas* or *synthesis directives*. Simulators ignore these comments, but synthesis compilers look for comments that begin with the word `synthesis`.

To avoid latches from an incomplete case statement (but not an if-else statement), synthesis compilers look for the pragma comment:

```
case (<case_expression>) // synthesis full_case
```

Either style of SystemVerilog comment can be used, and the comments are not case sensitive. The `full_case` pragma must be specified immediately after the case expression, before the list of case items.

NOTE

At the time this book was written, one commercial synthesis compiler did not recognize `// synthesis` as a synthesis pragma. That compiler required that pragmas start with `// pragma` or `// synopsys`.

The `full_case` pragma instructs synthesis compilers to assume that a case statement is fully specified, as described in section 9.2 (page 327). This means that synthesis can ignore any values for the case expression that do not match a case item. Synthesis will perform the same gate-level logic reduction optimizations described for the `unique` and `priority` decision modifiers (see 9.3.5, page 340).

An example of using the `full_case` pragma is:

```
always_comb begin
    case (select)      // synthesis full_case
        2'b00: y = a;
        2'b01: y = b;
        2'b10: y = c;
    endcase
end
```

NOTE

The gate-level logic reduction that can occur from the `full_case` pragma is a very different behavior than the simulation behavior of the RTL model. This means the gate-level implementation was not verified in RTL simulation, which can lead to a gate-level implementation that does not work as intended.

Modern ASIC and FPGA designs have the speed and capacity to fully implement all possible decision values, even those that are not expected to be used by the design. Gate-level logic reduction for unused decision values is usually not needed. If, and only if, these gate-level synthesis optimizations are needed in the design, the `unique` or `priority` decision modifier should be used instead of the `full_case` pragma comment. The SystemVerilog `unique` or `priority` decision modifiers have the same synthesis optimization effects, but also report violation messages if a decision expression value occurs that is not decoded in the decision statement. These violations help to verify that the gate-level optimizations are safe to use.

A `full_case` pragma does not guarantee that latches will not be inferred. There are other things that can infer latches, in addition to a case statement that is not “full” (complete). Latches will be inferred if there is a possibility that any of the variables assigned in a combinational logic procedure will not be updated every time the `always` procedure is entered.

9.3.8 Additional notes about synthesis pragmas

Synthesis compilers also look for a `parallel_case` pragma, which enables specific gate-level optimizations of case statements. This pragma is discussed in Chapter 7, sections 7.4.3 (page 270). Just like the `full_case` pragma, using the `parallel_case` pragma can result in synthesis compilers creating a gate-level implementation that behaves very differently than the RTL model that was verified in simulation.

The SystemVerilog `priority`, `unique` and `unique0` decision modifiers replace the obsolete `full_case` and `parallel_case` synthesis pragmas. These modifiers inform synthesis to do the same optimizations as the pragmas, and — importantly — also enable run-time simulation checking to help verify that design conditions do not

occur that would not work with the gate-level optimizations. These run-time checks are described in section 9.3.5 (page 340).

- **priority** enables the same synthesis gate-level optimizations as the pragma
`// synthesis full_case`
- **unique** enables the same synthesis gate-level optimizations as the pragma
`// synthesis full_case parallel_case`
- **unique0** enables the same synthesis gate-level optimizations as the pragma
`// synthesis parallel_case.`

NOTE

At the time this book was written, one commercial synthesis compiler did not recognize `// synthesis` as a synthesis pragma. That compiler required that pragmas start with `// pragma` or `// synopsys`.

The pragma pair `translate_off` tells synthesis compilers to ignore all code that follows, until a `translate_on` pragma is encountered. This allows debug code to be embedded into RTL code. Simulation, which ignores comments, will compile and execute the debug statements, but synthesis compilers will not try to implement the code in the target ASIC or FPGA. An alternative to using the `translate_off` and `translate_on` synthesis pragmas is to use conditional compilation. Most synthesis compilers have a predefined `SYNTHESIS` macro that can be used to conditionally include or exclude code that the synthesis tool compiles.

```
`ifdef SYNTHESIS // compile if using a synthesis compiler
...
`endif           // end of synthesis inclusion

`ifndef SYNTHESIS // compile if not a synthesis compiler
...
`endif           // end of synthesis exclusion
```

Synthesis compilers will also recognize `// pragma` to indicate a synthesis compiler directive.

9.4 Summary

This chapter has presented the best-practice coding style for modeling intentional latches as RTL models. Latch behavior in RTL simulations will occur anytime a non-clocked (no posedge or negedge sensitivity) always procedure triggers, and one or more variables are not assigned a value. In simulation, the variable will keep its previous value. This state retention requires some form of latch at the gate-level circuitry. Synthesis will automatically recognize when there is a potential for state retention in the RTL code, and infer latches in the ASIC or FPGA implementation.

A related topic to modeling latches is avoiding unintentional latches in RTL models. The most common cause of unintentional latch behavior is an incomplete if-else decision or an incomplete case statement. This chapter has shown several coding styles to aid in preventing unintended latches. The preferred coding styles that were discussed include: (1) Using a default assignment with a known value; (2) Using a pre-case assignment before the decision statement; (3) Using the **unique** or **priority** decision modifiers.

The old-fashioned coding style of using X value default assignments was discussed, along with several disadvantages of this older coding style. The obsolete **full_case** synthesis pragma was also discussed, along with the reasons it should never be used.

The **priority**, **unique** and **unique0** decision modifiers enable synthesis optimizations, and replace the old synthesis **full_case** and **parallel_case** pragmas. These decision modifiers add RTL verification checks to help ensure that the gate-level optimizations will work as intended.

* * *

Chapter 10

Modeling Communication Buses — Interface Ports

Abstract — Designs often use standard bus protocols such as PCI Express, USB, or AMBA AXI. Bus protocols bundle together several signals, including data signals, address signals, and various control signals. Bus protocols require functionality on each end of the bus to set and clear control lines in a specified order, and to transfer data and address values.

SystemVerilog interfaces are a type of module port, but are more versatile than a simple input, output or inout port. In its simplest form, an interface port bundles related signals together as a single, compound port. For example, all the individual signals that make up an AMBA AXI bus can be grouped together as an interface port. An interface can do more than just encapsulate bus signals. SystemVerilog interfaces provide a means for designers to centralize the functionality of a bus, as opposed to having the functionality scattered throughout several modules in a design. This simplifies the design engineer's work at the RTL level, and lets synthesis do the work of distributing the gate-level bus hardware appropriately throughout the design.

Interfaces are synthesizable when specific modeling guidelines and restrictions are followed. Interfaces can also be used at a non-synthesizable transaction level of modeling, and as part of a verification testbench. Advanced verification methodologies such as UVM, OVM and VMM, utilize interfaces.

This chapter examines interfaces as a synthesizable RTL modeling construct. The concepts covered in this chapter are:

- Interface declarations
- Connecting interfaces to module ports
- Differences between interfaces and modules
- Interface ports and port directions (modports)
- Tasks and functions in interfaces (interface methods)
- Procedural blocks in interfaces
- Parameterized interfaces

10.1 Interface port concepts

For synthesizable RTL modeling, the primary purpose of an interface is to encapsulate the declarations and some protocol functionality of a multi-signal bus into a single definition. This interface definition can then be used in any number of modules, without having to repeat the declarations of the bus signals and protocol functionality.

An interface is defined between the keywords **interface** and **endinterface**. A synthesizable RTL interface definition can contain:

- Variable and net declarations with data types and vector widths.
- Module port definitions that give the direction of signals. Different definitions can be specified for different modules that will use the interface.
- Functions to model zero-delay, zero clock cycle bus functionality.

Interfaces can also contain non-synthesizable transaction-level functionality and verification code, including initial procedures, always procedures, tasks and assertions. These non-synthesizable aspects of interfaces are not discussed in this book.

The examples in this chapter use a simplified version of an AMBA AHB bus, referred to as “simple AHB”, to communicate between a master and slave module. This simplified version only uses 8 of the 19 signals that comprise a full AMBA AHB bus. The simple AHB signals are:

Table 10-1: Simplified AMBA AHB signals

hclk	1-bit	Bus transfer clock, generated externally
hresetN	1-bit	Active-low bus reset, generated externally
haddr	32-bits	Transfer address
hwdata	32-bits	Data value sent from the master to the slave (some examples add a 1-bit odd-parity bit)
hrdata	32-bits	Data value sent from the slave back to the master (some examples add a 1-bit odd-parity bit)
hsize	3-bits	Control signal indicating the size of the transfer
hwrite	1-bit	Transfer direction control from the master to the slave (1 for write, 0 for read)
hready	1-bit	Response from the slave indicating the transfer is finished

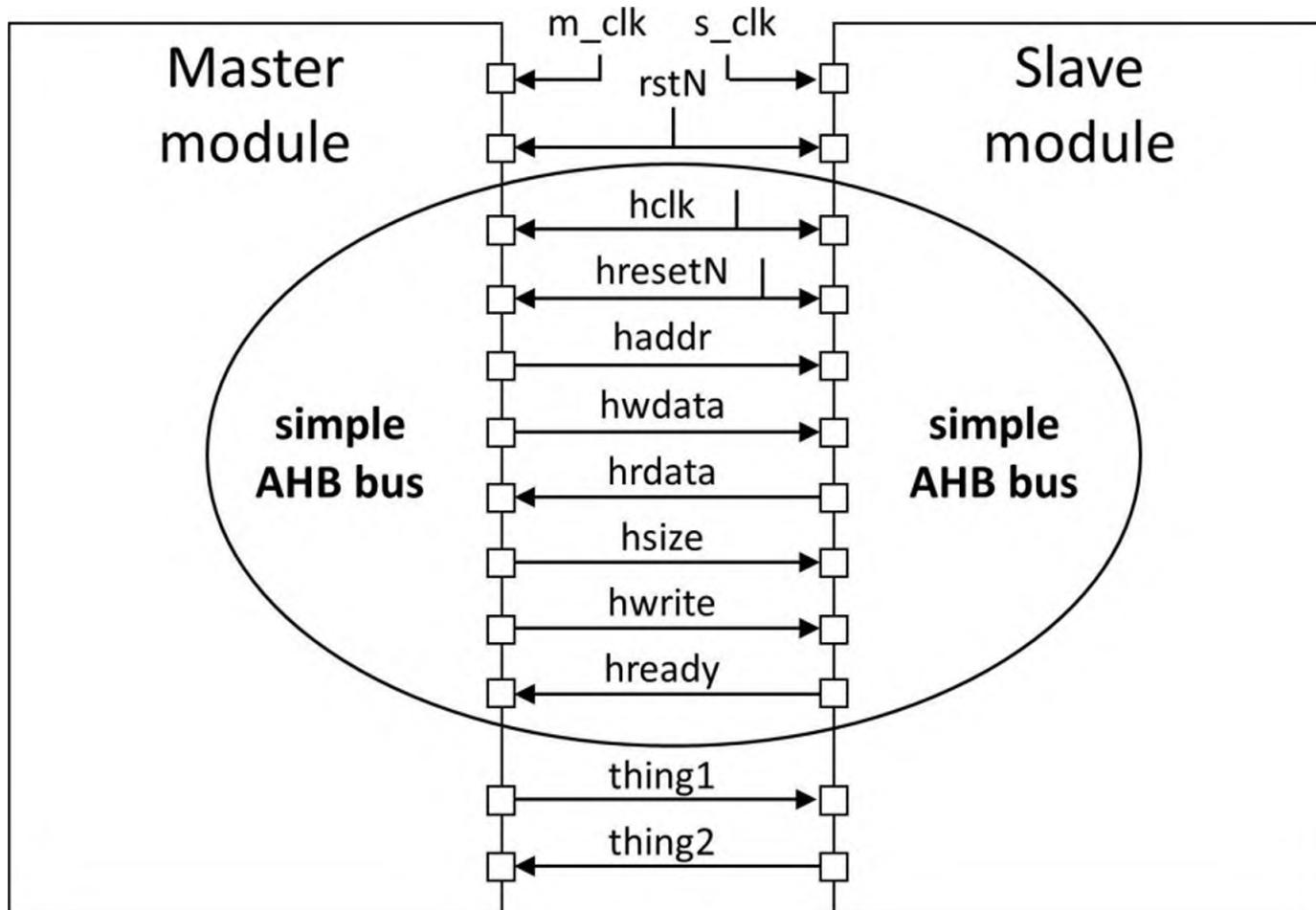
This simple AHB bus communicates between a single master and single slave module, and therefore does not require the bus arbiter and decoder blocks that a full AMBA AHB bus would need.

10.1.1 Traditional Verilog bus connections

Without interfaces, the signals that make up a communication bus must be declared as separate ports in each module that utilizes the bus. These port declarations must be repeated in each module that uses the bus, and duplicated yet again as interconnecting nets in the netlist module that connects the bus to the other modules.

Figure 10-1 shows a block diagram for connecting the master and slave modules together, using the 8 signals for the simplified version of an AMBA AHB bus. Four additional signals that are not related to the simple AHB bus are also shown. Design blocks that use some form of bus protocol to communicate typically have other inputs and outputs beyond those that make up the bus.

Figure 10-1: Block diagram connecting a Master and Slave using separate ports



Example 10-1 shows the code for connecting the master and slaves modules illustrated in Figure 10-1. Observe the repetition of declarations for the 8 signals that comprise a the simple AHB bus. The same signals must be declared in the master module, the slave module, the chip-level module that connects master and slave, and in the connections to the module instances for master and slave. Example 10-1 is modeled using traditional Verilog-2001 style and data types.

Example 10-1: Master and slave module connections using separate ports

```
/////////////////////////////
// Master Module Port List -- Verilog-2001 style
/////////////////////////////
module master (
    // simplified AHB bus signals
    input wire          hclk,
    input wire          hresetN,
    output reg [31:0]   haddr,
    output reg [31:0]   hwdata,
    output reg          hwrite,
    output reg [2:0]    hsize,
    input wire [31:0]   hrdata,
    input wire          hready,
    // other signals
    input wire          m_clk,    // master clock
    input wire          rstN,    // reset, active low
    input wire [7:0]    thing1,  // misc signal; not part of bus
    output reg [7:0]   thing2  // misc signal; not part of bus
);
    ... // master module functionality not shown
endmodule: master
```

Master module needs individual ports for the simplified AHB signals.

```
/////////////////////////////
// Slave Module Port List -- Verilog-2001 style
/////////////////////////////
module slave (
    // simplified AHB bus signals
    input wire          hclk,
    input wire          hresetN,
    input wire [31:0]   haddr,
    input wire [31:0]   hwdata,
    input wire          hwrite,
    input wire [2:0]    hsize,
    output reg [31:0]  hrdata,
    output reg          hready,
    // other signals
    input wire          s_clk,   // slave clock
    input wire          rstN,   // reset, active low
    output reg [7:0]   thing1, // misc. signal; not part of bus
    input wire [7:0]   thing2 // misc. signal; not part of bus
);
    ... // slave module functionality not shown
endmodule: slave
```

**Slave module needs duplicate ports for the simplified AHB signals.
Vector sizes must match for correct functionality**

```

///////////////////////////////
// Top-level Netlist Module -- Verilog-2001 style
///////////////////////////////

module chip_top;
  // Simplified AHB bus signals
  wire          hclk;
  wire          hresetN;
  wire [31:0]   haddr;
  wire [31:0]   hwdata;
  wire          hwrite;
  wire [ 2:0]   hsize;
  wire [31:0]   hrdata;
  wire          hready;

  // Other signals
  wire          m_clk;           // master clock
  wire          s_clk;           // slave clock
  wire          chip_rstN;       // reset, active low
  wire [7:0]    thing1;         // misc signal; not part of bus
  wire [7:0]    thing2;         // misc signal; not part of bus

  master m (// simplified AHB bus connections
    .hclk(hclk),
    .hresetN(hresetN),
    .haddr(haddr),
    .hwdata(hwdata),
    .hsize(hsize),
    .hwrite(hwrite),
    .hrdata(hrdata),
    .hready(hready),
    // Other connections
    .m_clk(m_clk),
    .rstN(chip_rstN),
    .thing1(thing1),
    .thing2(thing2)
  );

  slave s (// simplified AHB bus connections
    .hclk(hclk),
    .hresetN(hresetN),
    .haddr(haddr),
    .hwdata(hwdata),
    .hsize(hsize),
    .hwrite(hwrite),
    .hrdata(hrdata),
    .hready(hready),
    // Other connections
    .s_clk(s_clk),
    .rstN(chip_rstN),
    .thing1(thing1),
    .thing2(thing2)
  );

```

Higher-level module must duplicate the simplified AHB signals again in order to connect master and slave.

Connection to the master ports must duplicate the simplified AHB signals again (SystemVerilog's dot-name or dot-star shortcuts could reduce this redundancy if all names match exactly).

Connection to the slave ports duplicates the simplified AHB signals again (SystemVerilog's dot-name or dot-star shortcuts could reduce this redundancy if all names match exactly).

```

... // remaining chip-level code not shown

endmodule: chip_top

```

Disadvantages of separate module ports. Using separate module ports for the bus signals provides a simple and intuitive way of describing the interconnections between the blocks of a design. The individual ports accurately model the signals that make up the physical implementation of the bus. In large, complex designs, however, using individual module ports have several shortcomings. Some of these disadvantages are:

- Declarations must be duplicated in multiple modules.
- Communication protocols, such a handshake sequence, 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.

The signals that make the simplified AHB 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 master and slave modules together. Even with the simplified AHB bus example listed above — which only uses 8 of the 19 AHB bus signals, and only has a single slave module — the duplication of names is obvious. Each AHB signal is named a total of 7 times!

This duplication not only requires typing in lot of lines of code, but has a high potential for coding mistakes. A mis-typed name or incorrect vector size in one place can result in a functional bug in the design that is not detected until late in the design process when modules are connected together for full chip verification.

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), each and every module that shares the bus must be changed. The netlists used to connect the modules using the bus must also be changed. This widespread effect of a change is counter to good coding style. One goal of good 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 of using discrete input and output ports is that a change to the ports in one module will usually require changes in other files.

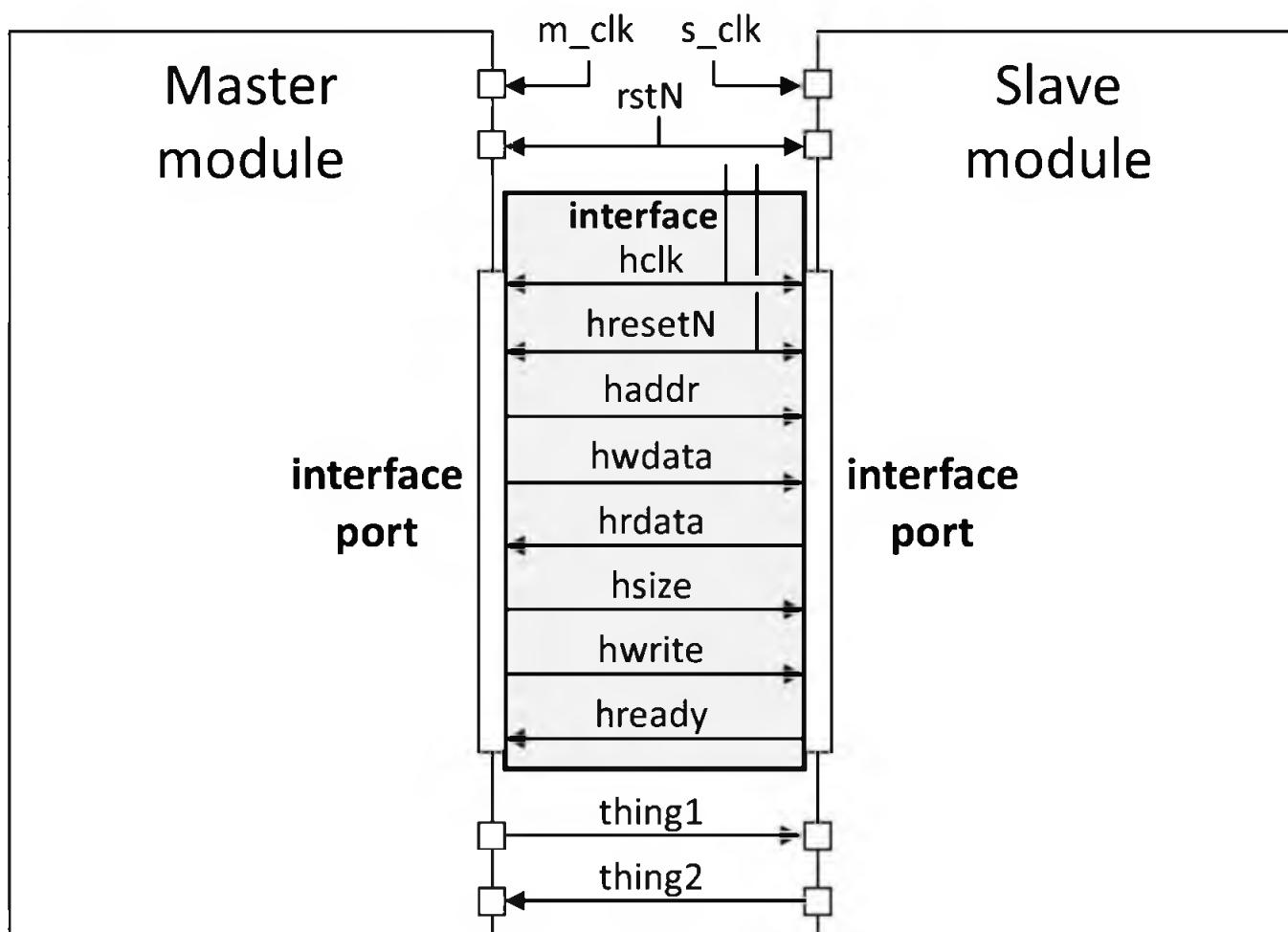
Another disadvantage of using discrete input and output module ports is that communication protocols must be duplicated in each module that utilizes the interconnecting signals between the modules. If, for example, three modules read and write from a shared memory device, then the read and write control logic must be duplicated in each of these modules.

10.1.2 SystemVerilog interface definitions

SystemVerilog adds a powerful new port type to Verilog, called an interface port. 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 encapsulated between the keywords **interface** and **endinterface**. Each module that uses these signals then has a single port of the interface type, instead of many ports for the discrete signals.

Figure 10-2 shows how an interface combines several individual ports into a single port that connects to an interface.

Figure 10-2: Block diagram connecting a Master and Slave using interface ports



The following three-part example shows how using interfaces can reduce the amount of code required to model the simple AHB communication bus shown in section 10.1.1 (page 357) above.

Example 10-2 shows a definition of an interface component that encapsulates the signals that make up the simple AHB as an interface.

Example 10-3 (page 363) shows the definitions of the master and slave modules. The 8 separate ports on the master module for the simple AHB bus have been replaced by single interface port. Instead of declaring this interface port as **input**, **output** or **inout**, the interface port is declared as `simple_ahb`, which is the name of the interface defined in Example 10-2. The interface ports eliminate the redundant simple AHB signal declarations within the master and slave modules when traditional individual input and output ports are used, as was the case in Example 10-1.

Example 10-4 (page 364) shows the higher-level netlist that connects the master and slave modules. Gone are the 24 lines of code to declare 8 separate bus signals and

then connect those 8 signals to the master and the slave module ports that were listed in Example 10-1. Instead, the `simple_ahb` interface is instantiated in the same way as a module, and the instance name is connected to the interface ports of the master and slave module instances.

Example 10-2: An interface definition for the 8-signal simple AMBA AHB bus

```
///////////////////////////////
// Simple AMBA AHB Interface -- SystemVerilog-2012 style
/////////////////////////////
interface simple_ahb (
    input logic hclk,      // bus transfer clk
    input logic hresetN   // bus reset, active low
);
    logic [31:0] haddr;    // transfer start address
    logic [31:0] hwdata;   // data sent to slave
    logic [31:0] hrdata;   // return data from slave
    logic [2:0] hsize;     // transfer size
    logic         hwrite;   // 1 for write, 0 for read
    logic         hready;   // 1 for transfer finished

    // master module port directions
    modport master_ports (
        output haddr, hwdata, hsize, hwrite, // to AHB slave
        input  hrdata, hready,                // from AHB slave
        input  hclk, hresetN                 // from chip level
    );

    // slave module port directions
    modport slave_ports (
        output hrdata, hready,              // to AHB master
        input  haddr, hwdata, hsize, hwrite, // from AHB master
        input  hclk, hresetN                 // from chip level
    );
endinterface: simple_ahb
```

Ports on interfaces. An interface can have input, output, and inout ports just like a module. The simple AHB interface shown in Example 10-2 has two input ports, `hclk` and `hresetN`. These signals are generated outside of the interface and passed into the interface through the two input ports. The declaration of ports on an interface is the same as for ports on modules.

An interface can also have interface ports, the same way a module can have interface ports. This allows one interface to be connected to another interface. For example, the main bus of a design might have one or more sub-busses. Both the main bus and its sub-busses can be modeled as interfaces, and sub-bus interfaces can be used as ports of the main bus interface.

Interface modports, a first look. The interface definition above includes two `modport` definitions, with the names `master_ports` and `slave_ports`. The keyword `modport` is an abbreviation for “module’s ports”, and defines whether a module sees the signals in the interface as inputs to the module or outputs from the module. One advantage of interfaces is that the data types and vector sizes of the signals used in the bus protocol are defined one time. The modport definitions simply add a direction, from a module’s perspective, to the signals defined in the interface. Modport definitions are covered in more detail in section 10.3 (page 367).

The following example of a master and slave module illustrates using the `simple_ahb` interface as a port to each module. Observe how the single interface port replaces the 8 discrete input and output ports shown in the master module, and 8 more ports in the slave module, as shown in Example 10-1 (page 358).

Example 10-3: Master and slave modules with interface ports

```
/////////////////////////////
// Master Module Port List -- SystemVerilog-2012 style
/////////////////////////////
module master
(simple_ahb.master_ports ahb, ←———— Interface port with modport
 // other ports
  input logic      m_clk,    // master clock
  input logic      rstN,     // reset, active low
  input logic [7:0] thing1, // misc signal; not part of bus
  output logic [7:0] thing2 // misc signal; not part of bus
);
  ... // master module functionality not shown
endmodule: master

/////////////////////////////
// Slave Module Port List
/////////////////////////////
module slave
(simple_ahb.slave_ports ahb, ←———— Interface port with modport
 // other ports
  input logic      s_clk,    // slave clock
  input logic      rstN,     // reset, active low
  output logic [7:0] thing1, // misc signal; not part of bus
  input logic [7:0] thing2 // misc signal; not part of bus
);
  ... // slave module functionality not shown
endmodule: slave
```

Connecting interface ports, a first look. With traditional module ports, a top-level module must declare individual nets for the bus signals, and then make separate connections for each individual signal to the ports of each module instance. Interfaces

greatly simplify these connections. When a module with an interface port is instantiated, an instance of an interface is connected to the interface port.

The following code instantiates the `simple_ahb` interface and gives it an instance name of `ahb1`. This instance name is then used in the port connections of the master and slave module instances.

Example 10-4: Netlist connecting the master and slave interface ports

```
///////////////////////////////
// Top-level Netlist Module -- SystemVerilog-2012 style
/////////////////////////////
module chip_top;
    logic          m_clk;           // master clock
    logic          s_clk;           // slave clock
    logic          hclk;            // AHB bus clock
    logic          hresetN;         // AHB bus reset, active low
    logic          chip_rstN;       // reset, active low
    logic [7:0]     thing1;          // misc signal; not part of bus
    logic [7:0]     thing2;          // misc signal; not part of bus

    simple_ahb ahb1(.hclk(hclk),           ← Instantiate the interface,
                    .hresetN(hresetN)           just like a module)
                    );

    // instantiate master and connect the interface instance
    // to the interface port

    master m (.ahb(ahb1),                  ← Connect the interface instance
              .rstN(chip_rstN),           name to master's interface port
              .m_clk,                   // dot-name connection shortcut
              .thing1,                  // for the other ports
              .thing2
              );

    slave s (.ahb(ahb1),                  ← Connect the interface instance
              .rstN(chip_rstN),           name to slave's interface port
              .*                         // wildcard connection shortcut
              );
              // for the other ports

    ... // remaining chip-level code not shown
endmodule: chip_top
```

In the examples above, all the signals that make up the simple AHB bus protocol have been encapsulated into the `simple_ahb` interface. The master, slave, and top-level modules do not duplicate the declarations of these bus signals. Instead, the master and slave modules simply use the interface as the connection between the modules. The interface eliminates the redundant declarations of separate module ports.

NOTE

A module interface port cannot be left unconnected.

A module **input**, **output** or **inout** port can be left unconnected on a module instance. This is not the case for interface ports. An interface port must be connected to an interface instance. An elaboration error will occur if an interface port is left unconnected.

Section 10.2 (page 366) provides additional details on declaring module interface ports and connecting to interface ports.

10.1.3 Referencing signals within an interface

An interface port is a compound port that has signals inside the port. Within a module that has an interface port, the signals inside the interface are accessed by using the port name, using the following syntax:

```
<port_name>.<internal_interface_signal_name>
```

The `simple_ahb` interface above contains a signal called `hclk`, and `master` has an interface port named of `ahb`. The master module can access `hclk` by using `ahb.hclk`.

```
always_ff @ (posedge ahb.hclk)
  ...

```

Best Practice Guideline 10-1

Use short names for interface port names in RTL models. The port name will need to be referenced frequently in the RTL code.

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.

10.1.4 Differences between modules and interfaces

There are three fundamental differences between an interface and 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 10.3 (page 367).

10.1.5 Source code declaration order

The name of an interface is referenced in two contexts: as a port of a module, and as an instance of the interface. Interfaces can be instantiated and 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 Using interfaces as module ports

A port of a module can be declared as an interface type, instead of the traditional **input**, **output** or **inout** port directions. A module can have any number of interface ports, and the interface ports can be specified in any order with other ports. The examples in this book list the interface port first, simply to emphasize the interface port.

There are two styles of interface port declarations — *generic* and *type-specific*. Both styles are synthesizable.

10.2.1 Generic interface ports

A *generic interface port* defines the port type by 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 type of 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 `bridge` is defined with two generic interface ports:

```
interface ahb_bus;
  ... // signal declarations for an AMBA AHB bus
endinterface

interface usb_bus;
  ... // signal declarations for a USB bus
endinterface

module bridge (interface bus_in,    // generic interface port
               interface bus_out); // generic interface port
  ...
endmodule
```

Each generic interface port could have either an `ahb_bus` interface instance or a `usb_bus` interface instance connected to it (or any other type of interface).

10.2.2 Type-specific interface ports

A module port can be explicitly declared as a specific type of interface. A *type-specific interface port* is declared by using the name of an interface as the port type. The syntax is:

```
module <module_name> (<interface_name> <port_name>);
```

For example:

```
module CACHE (ahb_bus ahb, // interface-specific port
               input    rstN);
  ...
endmodule
```

A type-specific interface port can only be connected to an instance of an interface of the same type. In the example above, a higher-level netlist could instantiate the CACHE module and connect an instance of an `ahb_bus` interface, but could not connect an instance of an `usb_bus` interface. A simulator or synthesis tool will issue an elaboration error if the wrong type of interface instance is connected to the type-specific interface port. Type-specific 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 makes the port type more obvious to anyone else who needs to review or maintain the module. With type-specific interface ports, it is easier to see exactly how the port is intended to be used.

Best Practice Guideline 10-2

Use type-specific interface ports for RTL models. Do not use generic interface ports in design modules.

The functionality of a module will almost always need to reference signals within the interface. With a type-specific interface port, the signal names within the interface are known at the time the module is written, and can be referenced without concern. With generic interface ports, there is no guarantee that every interface instance connected to the module's interface port will have the same signal names within the interface.

10.3 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 unique view of the signals within the interface. For example, on an AHB bus, the `hwdata` signal is an output from the master module, whereas the same `hwdata` is an input to a slave on the same bus.

SystemVerilog interfaces provide a means to define different views of the interface signals, so that each module sees an interface port with the correct port directions. The definition is made within the interface, using a **modport** keyword. *Modport* is an abbreviation for *module port*, and 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. The modport definitions do not duplicate vector size and type information that have already been defined in the interface signal declarations. A modport only defines whether the connecting module sees a signal as an **input**, **output**, **inout** port.

Two modport declaration examples are shown in the following interface:

```
interface simple_ahb (
    input logic hclk,      // bus transfer clk
    input logic hresetN // bus reset, active low
);
    logic [31:0] haddr;    // transfer start address
    logic [31:0] hwdata;   // data sent to slave
    logic [31:0] hrdata;   // return data from slave
    logic [2:0] hsize;     // transfer size
    logic          hwrite;  // 1 for write, 0 for read
    logic          hready;  // 1 for transfer finished

    // master module port directions
modport master_ports (
    output haddr, hwdata, hsize, hwrite, // to AHB slave
    input  hrdata, hready,                // from AHB slave
    input  hclk, hresetN                // from chip level
);

    // slave module port directions
modport slave_ports (
    output hrdata, hready,                // to AHB master
    input  haddr, hwdata, hsize, hwrite, // from AHB master
    input  hclk, hresetN                // from chip level
);
endinterface: simple_ahb
```

10.3.1 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 port declaration in the module definition
- As part of the interface connection to a module instance

Both of these styles are synthesizable, but there are advantages to specifying the modport as part of the module port definition, which are discussed in the following paragraphs.

Selecting the modport in the module's interface port declaration. The specific modport to be used from an interface can be specified directly as part of the interface port declaration within the module. The modport to be connected to the interface is specified as:

```
<interface_name>.<modport_name>
```

For example:

```
module master
(simple_ahb.master_ports ahb, // interface port & modport
 // other ports
input logic      m_clk,    // master clock
input logic      rstN,     // reset, active low
input logic [7:0] thing1,   // misc signal; not part of bus
output logic [7:0] thing2  // misc signal; not part of bus
);
... // master module functionality not shown
endmodule: master
```

Only type-specific interface ports can specify a modport as part of the port declaration. A generic interface port cannot specify a modport.

At the higher-level module that instantiates and connects this `master` module, an instance of the interface is connected to the module port, without specifying the name of a modport. For example:

```
module chip_top;
... // local net declarations

simple_ahb ahb1(.hclk(hclk), // interface instance
                 .hresetN(hresetN)
               );

master m (.ahb(ahb1),      // connect interface port
          .rstN(chip_rstN),
          .m_clk,           // dot-name connection shortcut
          .thing1,          // for the other ports
          .thing2
        );
...
endmodule: chip_top
```

Selecting the modport in the module instance. An alternate coding style is to leave the modport selection out of the module definition, and instead postpone selecting the modport until the module is instantiated. The following example declares the first

port of the slave module as a `simple_ahb` interface port, but does not specify which modport definition to use.

```
module slave
  (simple_ahb           ahb, // interface port without modport
   // other ports
   input logic          s_clk, // slave clock
   input logic          rstN, // reset, active low
   output logic [7:0] thing1, // misc signal; not part of bus
   input logic [7:0] thing2 // misc signal; not part of bus
 );
  ... // slave module functionality not shown
endmodule: slave
```

The specific modport of the interface can then be specified when the module is instantiated, and an instance of an interface is connected to a module instance. The connection is specified as:

<interface_instance_name>.<modport_name>

For example:

```
slave s (.ahb(ahb1.slave_ports), // select slave modport
         .rstN(chip_rstN),
         .*                      // wildcard connection shortcut
       );
```

When the modport to be used is specified in the module instance, the module definition can use either a type-specific interface port or a generic interface port type, as discussed in section 10.2 (page 366).

NOTE

A modport can be selected in either the module port definition or the module instance, but not both.

Best Practice Guideline 10-3

Select the modport to be used by a module as part of the module's interface port declaration. Do not select the modport at the netlist level.

Specifying the modport as part of the port declaration also allows the module to be synthesized independently from other modules. It also helps make the module more self-documenting. Engineers who read or maintain the module can immediately see which modport is to be used with that module.

Connecting to interfaces without specifying a modport. Module interface ports can be connected to an interface instance without specifying a specific modport definition. When no modport is specified, all nets in the interface are assumed to have a bidirectional `inout` direction. In simulation, variables in the interface are assumed to be of type `ref`. (A `ref` port permits both sides of the port to read and modify the variable. `ref` ports are not synthesizable and are not discussed in this book.) Synthesis compilers treat all signals in an interface port where no modport was specified as being `inout` bidirectional ports.

10.3.2 Using modports to define different sets of connections

In a more complex interface used by several different modules, it might 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 that uses the interface.

A module can only 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 some signals that are only used by modules connected to the interface through the `master_ports` modport, and not by modules connected through the `slave_ports` modport.

The full AMBA AHB bus has 19 signals, several of which are only used by the bus master, and not by the bus slave(s).

The following example illustrates a custom version of the `simple_ahb` interface that adds 3 additional AMBA AHB signals, `hprot`, `hburst` and `htrans`, that are only used by the master module. The modport declarations for `master_ports` and `slave_ports` ensure that the master and slave modules see the correct set of signals for the respective modules.

Example 10-5: Interface with modports for custom views of interface signals

```
interface simple_ahb (
    input logic hclk,      // bus transfer clk
    input logic hresetN    // bus reset, active low
);
    logic [31:0] haddr;    // transfer start address
    logic [31:0] hwdata;   // data sent to slave
    logic [31:0] hrdata;   // return data from slave
    logic [2:0] hsize;     // transfer size
    logic        hwrite;    // 1 for write, 0 for read
    logic        hready;    // 1 for transfer finished
    // additional AHB signals only used by bus master
    logic [3:0] hprot;     // transfer protection mode
    logic [2:0] hbust;     // transfer burst mode
    logic [1:0] htrans;    // transfer type
```

```

// master module port directions
modport master_ports (
    output haddr, hwdata, hsize, hwrite, // to AHB slave
    input hrdata, hready,           // from AHB slave
    input hclk, hresetN,          // from chip level
    // additional AHB signals only used by bus master
    output hprot, hburst, htrans
);

// slave module port directions
modport slave_ports (
    output hrdata, hready,           // to AHB master
    input haddr, hwdata, hsize, hwrite, // from AHB master
    input hclk, hresetN            // from chip level
);
endinterface: simple_ahb

```

A module that uses the `simple_ahb.master_ports` modport can use the `hprot`, `hburst` and `htrans` signals. A module that uses the `simple_ahb.slave_ports` modport cannot access these 3 signals. Since these signals are not listed in the `slave_ports` modport, it is as if those signals do not even exist.

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.

10.4 Interface methods (tasks and functions)

SystemVerilog interfaces can do more than just group related signals together. Interfaces can also encapsulate functionality for the communication between modules. By adding the communication functionality to the interface, each module that uses the interface can simply reference the functionality, without having to duplicate that functionality in each module. Encapsulated functionality in an interface can also be verified independent of the modules that use the interface.

Functionality encapsulated in an interface can be defined by using tasks and functions. Tasks and functions in an interface are referred to as *interface methods*. The interface methods (tasks and functions) can be imported into the modules that need them by using an `import` statement within the `modport` definition for the module. Importing functions in a modport is similar to importing functions from a package, as described in Chapter 4, section 4.2.2 (page 104).

The following example adds two functions to the simple AHB interface — one to generate a parity bit value (using odd parity), and another function to check that data matches the calculated parity. The `hwdata` and `hrdata` vectors have been declared 1-bit wider than in previous examples, with the extra bit used for as a parity bit.

Example 10-6: Interface with internal methods (functions) for parity logic

```

interface simple_ahb (
    input logic hclk,      // bus transfer clk
    input logic hresetN // bus reset, active low
);
    logic [31:0] haddr;    // transfer start address
    logic [32:0] hwdata;   // data to slave, with parity bit
    logic [32:0] hrdata;   // data from slave, with parity bit
    logic [ 2:0] hsize;    // transfer size
    logic          hwrite; // 1 for write, 0 for read
    logic          hready; // 1 for transfer finished

    function automatic logic parity_gen(logic [31:0] data);
        return(^data); // calculate parity of data (odd parity)
    endfunction

    function automatic logic parity_chk(logic [31:0] data,
                                         logic           parity);
        return (parity === ^data); // 1=OK, 0=parity error
    endfunction

    // master module port directions
    modport master_ports (
        output haddr, hwdata, hsize, hwrite, // to AHB slave
        input  hrdata, hready,                // from AHB slave
        input  hclk, hresetN,                // from chip level
        import parity_gen, parity_check     // function import
    );

    // slave module port directions
    modport slave_ports (
        output hrdata, hready,                // to AHB master
        input  haddr, hwdata, hsize, hwrite, // from AHB master
        input  hclk, hresetN,                // from chip level
        import parity_check                 // function import
    );
endinterface: simple_ahb

```

In this example, the `master_ports` modport definition imports both the `parity_gen` and `parity_chk` methods. Modports define port directions and imports from the perspective of the master module. Thus, a module that uses the `master_ports` modport is importing these functions, similar to the way a module can import functions from a package.

The `slave_ports` modport only imports the `parity_chk` method. A module that uses the `slave_ports` modport cannot access the `parity_gen` method. Since that method was not included in the modport imports, it is as if that method does not exist from the view of the interface provided by `slave_ports`.

Importing methods using a method prototype. Optionally, a modport **import** declaration can specify a full prototype of the task or function arguments. Instead of importing just the method name, the **import** keyword is followed by the declaration line(s) of the actual method definition. The basic syntax of this style of import declarations is:

```
modport (import task <task_name> (<formal_args>) );
modport (import function <function_name> <return_type>
(<formal_args>) );
```

For example:

```
// slave module port directions
modport slave_ports_alt (
    output hrdata, hready,           // to AHB master
    input haddr, hwdata, hsize, hwrite, // from AHB master
    input hclk, hresetN,             // from chip level
    import function logic parity_chk(logic [31:0] data,
                                      logic          parity)
);
```

The function prototype does not include the **automatic** keyword, even if the actual functions was declared as automatic (which is required for synthesis).

There is little advantage to using a full prototype to import a method. Some engineers feel that 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 when the actual task or function is defined in a package, and imported into the interface. The prototype makes the method type and arguments visible in the interface definition, so that engineers do not need to go to the file containing the package to see the method type and arguments.

10.4.1 Calling methods defined in an interface

Imported methods are part of the interface, and are called by using the interface port name, in the same way signals in an interface are referenced. The syntax is:

```
<interface_port_name>. <internal_interface_method_name>
```

The master module shown previously in this chapter has an interface port named `ahb`. Thus, the master module can call the `parity_gen` method within the interface by referencing it as `ahb.parity_gen`. For example:

```
always_ff @(posedge ahb.hclk)
    ahb.hwdata[32] <= ahb.parity_gen(ahb.hwdata[31:0]);
```

10.4.2 Synthesizing interface methods

Conceptually, synthesis compilers replace an imported method by creating a local copy of that method within the module, and then synthesizing the local copy. The post-synthesis version of the module will contain the logic of the imported method, and will no longer look to the interface for that functionality.

Best Practice Guideline 10-4

For synthesizable RTL interfaces, only use functions and void functions in the interface. Do not use tasks or always procedures.

Synthesis compilers place the same RTL coding restrictions on the contents of an interface that are placed in modules. One of these restrictions is that tasks must execute in zero time. Using a void function instead of a task enforces this synthesis restriction. Chapter 7, section 7.3 (page 263) discusses the advantages of using void functions in synthesizable RTL models.

NOTE

Imported functions or tasks must be declared as **automatic**, and cannot contain static declarations in order to be synthesized. This is the same synthesis rule as when a module imports functions or tasks from a package.

An automatic function or task allocates new storage each time it is called. When a module calls an imported method, a new copy of all internal storage is allocated. This allows synthesis to treat the method as if it were a local copy within the module.

10.4.3 Abstract, non-synthesizable interface methods

SystemVerilog interfaces are capable of representing bus protocols at a much higher level of abstraction than what RTL synthesis compilers support. For example, an interface task, which can take multiple clock cycles to execute, could represent a full master-slave handshaking protocol. The protocol could start with the master issuing a transfer request, arbitrating which slave receives the request, waiting for a grant from the slave, transferring the data, and receiving an acknowledgement that the data was received.

These interface capabilities can be useful for abstract transaction level modeling, but are not supported by current RTL synthesis compilers. Current SystemVerilog synthesis tools require limiting the functionality coded in an interface to be zero-delay and zero clock cycle models. These synthesis restrictions can be met by limiting the functional code defined in an interface to functions. SystemVerilog syntax rules

require that functions must execute in zero simulation time, which adheres to the synthesis requirement of zero-delay interface functionality.

An interface can also contain verification routines and assertions. This verification code can be hidden from synthesis by enclosing it in the pragma pair:

```
//synthesis translate_off and //synthesis translate_on.
```

10.5 Interface procedural code

In addition to task and functions methods, interfaces can also contain initial and always procedural blocks and continuous assignments. Procedural code can be used to model functionality within an interface that affects the information communicated across the bus the interface represents.

Example 10-7 adds a clock generator for the simple AHB bus `hclk`, and a reset synchronizer for the bus `hresetN`. In the previous examples of this interface, these signals were generated external to the interface, and passed in as input ports of the simple AHB interface. This example replaces those inputs with the chip (or system) level clock and reset, and uses these chip-level signals to generate a local bus clock and bus reset. This local functionality then becomes part of the encapsulated bus communication between the master and slave modules.

Example 10-7: Interface with internal procedural code to generate bus functionality

```
interface simple_ahb (
    input logic chip_clk, // external clock from the chip
    input logic chip_rstN // bus reset, active low
);
    logic hclk; // local bus transfer clk
    logic hresetN; // local bus reset, active low
    logic [31:0] haddr; // transfer start address
    logic [31:0] hwdata; // data sent to slave
    logic [31:0] hrdata; // return data from slave
    logic [2:0] hsize; // transfer size
    logic hwrite; // 1 for write, 0 for read
    logic hready; // 1 for transfer finished

    // generate AHB clock (divide-by-two of chip_clk)
    always_ff @(posedge chip_clk or negedge chip_rstN)
        if (!chip_rstN) hclk <= '0;
        else           hclk <= ~hclk;
```

```

// sync trailing edge of hresetN to hclk
logic rstN_tmp; // temp variable used inside the interface
always_ff @(posedge hclk or negedge chip_rstN)
    if (!chip_rstN) begin // asynchronous active-low reset
        rstN_tmp <= '0;
        hresetN <= '0;
    end
    else begin
        rstN_tmp <= '1;           // begin end of reset
        hresetN <= rstN_tmp; // stabilize reset
    end

// master module port directions
modport master_ports (
    output haddr, hwdata, hsize, hwrite, // to AHB slave
    input hrdata, hready,                // from AHB slave
    input hclk, hresetN                // from chip level
);

// slave module port directions
modport slave_ports (
    output hrdata, hready,                // to AHB master
    input haddr, hwdata, hsize, hwrite, // from AHB master
    input hclk, hresetN                // from chip level
);
endinterface: simple_ahb

```

Synthesizing interface procedures. How synthesis compilers handle procedural code in an interface is not well-defined, unlike it is for interface methods (tasks and functions). Methods are synthesized by conceptually copying the method code into the module with an interface port, and synthesizing the local copy. This can be done because the method is called from within a module, and executes as if the method were part of that module. Procedural code, however, is executed from within the interface, and affects all modules that use the interface. Procedural code in an interface is akin to global functionality, which is not well supported by synthesis compilers, if at all.

Best Practice Guideline 10-5

Use functions to model functionality inside an interface. Do not use initial procedures, always procedures or continuous assignments in synthesizable RTL interfaces.

Procedural code within an interface is not well supported by synthesis compilers. If supported at all, procedural code might be handled in very different ways by different synthesis compilers.

10.6 Parameterized interfaces

Interfaces can use parameter redefinition in the same way as modules. This allows interface models to be configurable, so that each instance of the interface can have a different configuration. Parameters can be used in interfaces to make vector sizes and other declarations within the interface reconfigurable by using SystemVerilog's parameter redefinition constructs. The parameter values of an interface can be redefined when the interface is instantiated, in the same way as module redefinition. Chapter 3, section 3.8 (page 93) discusses the various styles of parameter redefinition.

The following variation of the simple AHB example adds parameters to make the data vector widths configurable when the interface is instantiated. Any module interface ports to which the interface instance is connected will use the vector sizes of that interface instance.

Example 10-8: Parameterized interface with configurable bus data word size

```

interface simple_ahb
#(parameter DWIDTH=32) // Data bus width, 32-bit default
(
    input logic hclk, // bus transfer clk
    input logic hresetN // bus reset, active low
);
    logic [31:0] haddr; // transfer start address
    logic [DWIDTH-1:0] hwdata; // data sent to slave
    logic [DWIDTH-1:0] hrdata; // return data from slave
    logic [ 2:0] hsize; // transfer size
    logic hwrite; // 1 for write, 0 for read
    logic hready; // 1 for transfer finished

    // master module port directions
    modport master_ports (
        output haddr, hwdata, hsize, hwrite, // to AHB slave
        input hrdata, hready, // from AHB slave
        input hclk, hresetN // from chip level
    );

    // slave module port directions
    modport slave_ports (
        output hrdata, hready, // to AHB master
        input haddr, hwdata, hsize, hwrite, // from AHB master
        input hclk, hresetN // from chip level
    );
endinterface: simple_ahb

```

The following code snippet redefines the data word size of the interface in Example 10-8 to a 64-bit word size.

```
simple_ahb #( .DWIDTH(64) ) ahb1(.hclk,  
                                .hresetN  
                            );
```

10.7 Synthesizing interfaces

Interfaces are a powerful modeling construct that SystemVerilog added to the original Verilog HDL. An interface port is an abstraction from traditional Verilog modeling, where a group of related signals had to be declared one signal at a time. Those separate declarations then had to be duplicated in every module that used the related signals, as well as at the block level that connected modules together.

In its most basic form, SystemVerilog interfaces encapsulate related signals together as a reusable modeling component. The interface can then be used as a single port on modules, replacing multiple individual ports for a group of related signals. The modeling abstraction provided by interfaces can be a powerful tool for RTL design engineers. Designers can define a group of related signals one time, as an interface, and then use those signals any number of times without having to duplicate the definitions.

Synthesis compilers handle using interfaces to encapsulate related signals very well. Design engineers can work at a higher level of abstraction — with all the advantages of abstraction — and synthesis compilers translate the abstract encapsulation of signals into the individual module ports, without engineers needing to get bogged down with the individual port declarations, and ensuring that redundant declarations in multiple modules match perfectly.

Synthesis compilers support both styles of specifying which modport is to be used with a module as part of the port declaration, or when the module is installed (see section 10.3.1, page 368). However, the modport must be specified with the port declaration if a module is synthesized independently from other modules.

Synthesis compilers will expand the interface port of a module into the individual ports represented in the modport definition when a module is synthesized independent of other modules, or when multiple modules are synthesized with the synthesis compiler configured to preserve the RTL module hierarchy. Most synthesis compilers will use a Verilog-1995 port declaration style, where the port list contains the port names and order, and the port sizes and data types are declared inside the module, instead of in the port list. A module can have any number of interface ports, and the interface ports can be specified in any order with other ports. The examples in this book list the interface port first, simply to emphasize the interface port.

The following code snippets show the possible pre-synthesis and post-synthesis module definitions of a master module that uses the `simple_ahb` interface shown in Example 10-3 (page 363).

Pre-synthesis module port list, with an interface port:

```
module master
(simple_ahb.master_ports ahb, // interface port & modport
// other ports
input logic m_clk, // master clock
input logic rstN, // reset, active low
input logic [7:0] thing1, // misc signal; not part of bus
output logic [7:0] thing2 // misc signal; not part of bus
);
```

The `master_ports` definition for this example is:

```
// master module port directions
modport master_ports (
    output haddr, hwdata, hsize, hwrite, // to AHB slave
    input hrdata, hready, // from AHB slave
    input hclk, hresetN // from chip level
);
```

Post-synthesis model, using Verilog-1995 coding style. The following post-synthesis example illustrates a typical result for how an interface port will synthesize. This example was not generated by any specific synthesis compiler.

```
module master (haddr, hwdata, hsize, hwrite,
               hrdata, hready, hclk, hresetN,
               m_clk, rstN, thing1, thing2);
    output [31:0] haddr;
    output [31:0] hwdata;
    output [2:0] hsize;
    output hwrite;
    input [31:0] hrdata;
    input hready;
    input hclk;
    input hresetN;
    input m_clk;
    input rstN;
    input [7:0] thing1;
    output [7:0] thing2;

    wire [31:0] haddr;
    wire [31:0] hwdata;
    wire [2:0] hsize;
    wire hwrite;
    wire [31:0] hrdata;
    wire hready;
    wire hclk;
    wire hresetN;
    wire m_clk;
    wire rstN;
    wire [7:0] thing1;
    wire [7:0] thing2;
```

```
... // master module functionality not shown  
endmodule: master
```

If a modport definition was specified, synthesis will use the directions specified in the modport. If no modport is specified when the model is synthesized, then all signals within the interface become bidirectional **inout** ports in the synthesized module.

Configurable interfaces are synthesizable in the same way as configurable modules. Interfaces can use parameters to configure bus widths and data types. Chapter 3, section 3.8 (page 93) contains examples of synthesizable, parameterized modules. Interfaces can be made configurable in the same way.

Interfaces can also encapsulate functionality related to those signals through the use of methods (tasks and functions) and procedural code. Functions in interfaces are synthesizable, as discussed in section 10.4.2 (page 375). This can be useful, and RTL design engineers should take advantage of this synthesis capability. Encapsulating functions with the signals on which they operate is a best-practice coding style for writing robust, reusable code.

Best Practice Guideline 10-6

Limit functionality in an interface to what can be modeled using functions.

The RTL synthesis compilers available at the time this book was written are somewhat limited in the support of using interfaces to encapsulate functionality using tasks and procedural code.

It is possible, for example, to encapsulate the full functionality of a FIFO within an interface, which would allow modules that use the encapsulated signals to run at different clock speeds without any loss of data. Complete error-correction functionality, and other complex operations related to a group of signals, can also be bundled with those signals. This more advanced level of encapsulation is not supported, or has only limited support, by most synthesis compilers. These restrictions limit the usefulness of procedural code in an interface.

Interfaces can also bundle verification code, such as assertions and self-checking routines for the encapsulated signals and functionality. Verification related code in an interface can be ignored by synthesis compilers using synthesis `translate_off` and `translate_on` pragmas or '**ifdef**' conditional compilation.

10.8 Summary

This chapter has presented interfaces and interface ports, powerful RTL modeling constructs which SystemVerilog added to the original Verilog language. 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 an interface definition, where those bus details only need to be defined once, instead of in many different modules.

The interface modport definition provides a simple yet powerful way to customize the interface for each module that is connected to the interface. Each modport definition defines the port directions for a particular view of the interface. One module can see a specific signal in the interface as an output, while another module sees that same signal as an input. Modport definitions also allow some signals or methods in an interface to be hidden from certain modules.

Interfaces do more than provide a way to bundle signals together. Interfaces can also encapsulate functionality that operates on the related signals by using methods (tasks and functions). The ability to incorporate methods in an interface further reduces redundant code that is used in multiple modules. Methods are defined in one place, in the interface, and can be imported into any number of modules as part of each module's modport definition. Functions in interfaces are synthesizable.

A synthesizable interface must adhere to the same RTL modeling rules as a synthesizable module. Interfaces are capable of modeling at non-RTL levels as well, and are a powerful construct for transaction-level modeling and verification testbenches. Advanced verification methodologies such as UVM, OVM and VMM rely on interfaces to communicate between an object-oriented testbench and the design modules being verified.

* * *

Appendix A

Best Practice Coding Guidelines

This book emphasizes writing RTL models that simulate and synthesize correctly, and yield best Quality of Results (QoR) in the gate-level implementation created by synthesis compilers. Each chapter several short “*Best Practice Guideline*” coding recommendations. For convenience, this appendix provides a summary of these recommendations.

Readers are encouraged to refer to the full description on each best practice coding recommendation for the full details of the recommendation and to understand why it is important.

Chapter 1: SystemVerilog Simulation and Synthesis

- 1-1 Use packages for shared declarations instead of the `$unit` declaration space. (p. 22)
- 1-2 Use the SystemVerilog `timeunit` keyword to specify simulation time units and precision, instead of the old `'timescale` compiler directive. (p. 25)

Chapter 2: RTL Modeling Fundamentals

- 2-1 The code portions of a model should only contain one-line comments that begin with `//`. Do not use block comments in the code body that encapsulate the comment between `/*` and `*/`. (p. 41)
- 2-2 Specify a `'begin_keywords` directive before every module, interface and package. Specify a matching `'end_keywords` directive at the end of every module, interface and package. (p. 46)
- 2-3 Use named port connections for all module instances. Do not use port order connections. (p. 56)

Chapter 3: Net and Variable types

- 3-1 Only use binary and hexadecimal literal integers in RTL models. These number bases have an intuitive meaning in digital logic. (p. 64)
- 3-2 Use a lint check program (also known as a modeling rule checker) in conjunction with simulation, and before synthesizing the RTL model. (p. 65)

- 3-3 Use the 4-state **logic** data type to infer variables in RTL models. Do not use 2-state types in RTL models. An exception to this guideline is to use the **int** type to declare for-loop iterator variables. (p. 68)
- 3-4 Use a simple vector declaration when a design mostly selects either the entire vector or individual bits of the vector. Use a vector with subfields when a design frequently selects parts of a vector, and those parts fall on known boundaries, such as byte or word boundaries. (p. 74)
- 3-5 Only use variable initialization in RTL models that will be implemented as an FPGA, and only to model power-up values of flip-flops. (p. 76)
- 3-6 Only use in-line variable initialization in RTL models. Do not use initial procedures to initialize variables. (p. 76)
- 3-7 Use a **logic** data type to connect design components together whenever the design intent is to have single driver functionality. Use **wire** or **tri** net types only when the design intent is to permit multiple drivers. (p. 78)
- 3-8 If the default net type is changed, always use `default_netttype as a pair of directives, with the first directive setting the default to the desired net type, and the second directive setting the default back to **wire**. (p. 82)
- 3-9 Use the ANSI-C style declarations for module port lists. Declare both input ports and output ports as a **logic** type. (p. 88)
- 3-10 Use in-line named parameter redefinition for all parameter overrides. Do not use in-line parameter-order redefinition or defparam statements. (p. 99)

Chapter 4: User-defined Types and Packages

- 4-1 Only use **localparam** or **const** definitions for package constants. Do not use **parameter** definitions in packages. (p. 104)
- 4-2 Avoid using \$unit like the Bubonic plague! Instead, use packages for shared definitions. (p. 114)
- 4-3 Use the explicit-style enumerated type declarations in RTL models, where the base type and label values are specified, rather than inferred. (p. 115)
- 4-4 Only use packed unions in RTL models. (p. 132)

Chapter 5: RTL Expression Operators

- 5-1 Use the bitwise invert operator to invert the bits of a value. Do not use the bitwise invert operator to negate logical true/false tests. Conversely, use the logical negate operator to negate the result of a true/false test. Do not use the logical negate operator to invert a value. (p. 162)
- 5-2 Only use the logical true/false operators to test scalar (1-bit) values. Do not perform true/false tests on vectors. (p. 162)

- 5-3 A function should only modify its function return variable and internal temporary variables that never leave the function. (p. 164)
- 5-4 Avoid mixing signed and unsigned expressions with comparison operations. Both operands should be either signed or unsigned. (p. 166)
- 5-5 Use the `==` and `!=` equality operators in RTL models. Do not use the `==>` and `!==>` case equality operators. (p. 169)
- 5-6 Use operators to shift or rotate a vector a variable number of bits. Do not use loops to shift or rotate the bits of a vector a variable number of bits. (p. 180)
- 5-7 For better synthesis Quality of Results (QoR):
 - (a) Use shift operators for multiplication and division by a power of 2, instead of the `*`, `/`, `%` and `**` arithmetic operators.
 - (b) For multiplication and division by a non-power of 2, use a constant value for one operand of the operation, if possible.
 - (c) For multiplication and division when both operands are non-constant values, use smaller vector sizes, such as 8-bits. (p. 185)
- 5-8 Use unsigned types for all RTL model operations. The use of signed data types is seldom needed to model accurate hardware behavior. (p. 189)
- 5-9 Only use the increment and decrement operators with combinational logic procedures and to control loops iterations. Do not use increment and decrement to model sequential logic behavior. (p. 191)

Chapter 6: RTL Programming Statements

- 6-1 Only use 1-bit values or the return of a true/false operation for an if-else condition expression. Do not use vectors as an if-else expression. (p. 217)
- 6-2 Use the `case...inside` decision statement to ignore specific bits in case items. Do not use the obsolete `casex` and `casez` decision statements. (p. 225)
- 6-3 Code `for` loops as static, zero-delay loops with a fixed number of iterations. (p. 231)
- 6-4 Code all loops with a fixed iteration size. This coding style ensures the loop can be unrolled, and will be supported by all synthesis compilers. (p. 232)
- 6-5 Use `for` loops and `repeat` loops for RTL modeling. Do not use `while` and `do-while` loops. (p. 235)
- 6-6 Use the `continue` and `break` jump statements to control loop iterations. Do not use the `disable` jump statement. (p. 241)
- 6-7 Do not use the no-op statement for RTL modeling. (p. 243)

- 6-8 Declare functions used in RTL models as **automatic**. (p. 244)
- 6-9 Use void functions in place of tasks for RTL modeling. Only use tasks in verification code. (p. 245)
- 6-10 Only use **input** and **output** formal arguments in functions used in RTL models. Do not use **inout** or **ref** formal arguments. (p. 246)

Chapter 7: Modeling Combinational Logic

- 7-1 Use variables on the left-hand side of continuous assignments to prevent unintentional multiple drivers. Only use **wire** or **tri** nets on the left-hand side when it is intended for a signal to have multiple drivers. (p. 253)
- 7-2 Ensure that both sides of continuous assignments and procedural assignments are the same vector width. Avoid mis-matched vector sizes on the left-hand and right-hand side expressions. (p. 253)
- 7-3 Model all RTL combinational logic with zero delays. (p. 257)
- 7-4 Use the RTL-specific **always_comb** procedure to model combinational logic. Do not use the generic **always** procedure in RTL models. (p. 257)
- 7-5 Use the SystemVerilog **always_comb** RTL-specific procedure to automatically infer correct combinational logic sensitivity lists. Do not use the obsolete @* inferred sensitivity list. (p. 259)
- 7-6 Only use blocking assignments (=) when modeling combinational logic behavior. (p. 261)
- 7-7 Always declare functions used in RTL models as **automatic**. (p. 263)
- 7-8 When possible, use SystemVerilog operators for complex operations such as multiplication, rather than using loops and other programming statements. (p. 264)
- 7-9 Only use the **unique** decision modifier if it is certain that the synthesis logic reduction optimization effects are desirable. (p. 269)
- 7-10 Use the **unique** decision modifier in RTL models. Do not use the **unique0** decision modifier. The **unique0** modifier might be recommended in the future, but, at the time this book was written, some simulators and most synthesis compilers did not support **unique0**. (p. 269)
- 7-11 Do not use the obsolete **parallel_case** synthesis pragma! (p. 270)

Chapter 8: Modeling Sequential Logic

- 8-1 Use the SystemVerilog **always_ff** RTL-specific procedure to model RTL sequential logic. Do not use the general purpose **always** procedure. (p. 276)
- 8-2 Only use nonblocking assignments (<=) to assign the output variables of sequential logic blocks. (p. 279)

- 8-3 Use separate combinational logic processes to calculate intermediate values required in sequential logic procedures. Do not embed intermediate calculations inside of sequential logic procedures. (p. 283)
- 8-4 Declare temporary variables that are used in a sequential logic block as local variables within the block. (p. 285)
- 8-5 Write RTL models using a preferred type of reset, and let synthesis compilers map the reset functionality to the type of reset supported by the target ASIC or FPGA. Only write RTL models to use the same type of reset that a specific target ASIC or FPGA uses if it is necessary in order to achieve the most optimal speed and area for that specific device. (p. 286)
- 8-6 Be consistent in the use of active-high or active-low resets. Use a consistent naming convention for active-high and active-low control signals. (p. 289)
- 8-7 Model RTL flip-flops with just a reset input or a set input in order to achieve best synthesis Quality of Results (QoR). Only model set/reset flip-flops if needed for the functionality of the design. (p. 291)
- 8-8 Multiple clock designs should be partitioned into multiple modules, so that each module only uses a single clock. (p. 295)
- 8-9 If intra-assignment delays are used at all, only use a unit delay. (p. 298)
- 8-10 Model FSMs in a separate module. (Support logic for the FSM, such as a counter that is only used by the FSM, can be included in the same module.) (p. 299)
- 8-11 Define a `logic` (4-state) base type and vector size for enumerated variables. (p. 303)
- 8-12 Use enumerated variables for FSM state variables. Do not use parameters and loosely typed variables for state variables. (p. 303)
- 8-13 Make the engineering decision on which encoding scheme to use for a Finite State Machine at the RTL modeling stage of design, rather than during the synthesis process. (p. 304)
- 8-14 For most Finite State Machines, use a three-process coding style, where a separate process models each of the three main blocks of the state machine. (p. 305)
- 8-15 Use reverse case statements to model one-hot state machines that evaluate 1-bit values. Do not use multi-bit vectors for one-hot case expressions and case items. (p. 313)
- 8-16 Behavioral RAM models should be defined in a separate module. (p. 317)

Chapter 9: Modeling Latches and Avoiding Unintentional Latches

- 9-1 Use nonblocking assignments (`<=`) to model latch behavior. (p. 324)
- 9-2 Use the RTL-specific `always_latch` procedure to model latch based logic. Do not use the generic `always` procedure in RTL models. (p. 324)

- 9-3 Fully specify the output values of decision statements to avoid unintended latches. Do not use logic reduction optimizations to avoid latches, unless needed for a specific circumstance. (p. 331)
- 9-4 Use the **unique** or **priority** decision modifiers if gate-level logic reduction is needed for avoiding unintended latches. Do not use the antiquated Verilog-2001 coding style of X value assignments. (p. 331)
- 9-5 Use the **unique** decision modifier in RTL models when logic reduction is desirable to prevent inferred latches. Do not use the **unique0** modifier. (p. 341)
- 9-6 In general, fully specify all case statements using either a default case item that assigns known values, or a pre-case assignment with known values. An exception to this guideline is a one-hot state decoder using a reverse case statement. (p. 344)
- 9-7 Use the **unique** or **priority** decision modifiers instead of an X value assignment for unused decision values. (p. 345)
- 9-8 Use the **unique** or **priority** decision modifiers if logic reduction optimizations are required. Do not use the `full_case` (or `parallel_case`) synthesis pragma. Ever. (p. 350)

Chapter 10: Modeling Communication Buses — Interface Ports

- 10-1 Use short names for interface port names in RTL models. The port name will need to be referenced frequently in the RTL code. (p. 365)
- 10-2 Use type-specific interface ports for RTL models. Do not use generic interface ports in design modules. (p. 367)
- 10-3 Select the modport to be used by a module as part of the module's interface port declaration. Do not select the modport at the netlist level. (p. 370)
- 10-4 For synthesizable RTL interfaces, only use functions and void functions in the interface. Do not use tasks or always procedures. (p. 375)
- 10-5 Use functions to model functionality inside an interface. Do not use initial procedures, always procedures or continuous assignments in synthesizable RTL interfaces. (p. 377)
- 10-6 Limit functionality in an interface to what can be modeled using functions. (p. 381)

Appendix B

SystemVerilog Reserved Keywords

Abstract — Each version of the Verilog and SystemVerilog standard has added additional reserved keywords to the previous generation of the standard. This appendix lists:

- Table B.1 — the full SystemVerilog-2012 reserved keyword list
- Table B.2 — the original Verilog-1995 reserved keyword list
- Table B.3 — additional keywords reserved in the Verilog-2001 standard
- Table B.4 — additional keywords reserved in the Verilog-2005 standard
- Table B.5 — additional keywords reserved in the SystemVerilog-2005 standard
- Table B.6 — additional keywords reserved in the SystemVerilog-2009 standard
- Table B.7 — additional keywords reserved in the SystemVerilog-2012 standard

Section 2.2.4 in Chapter 2 discusses using the **'begin_keywords** and **'end_keywords** compiler directive pair to control which keywords should be reserved when SystemVerilog source code is compiled.

B.1 All SystemVerilog-2012 reserved keywords

Table B-1 lists the reserved keywords for the SystemVerilog-2012 standard. The compiler directive **'begin_keywords "1800-2012"** instructs compilers to reserve the keywords listed in this table.

Note: Some keywords in this table have been hyphenated in order to fit the format of this book. No actual keywords contain hyphens.

accept_on	endchecker	inside	pullup	sync_accept-
alias	endclass	instance	pulsestyle-	_on
always	endclocking	int	_on-detect	sync_reject-
always_comb	endconfig	integer	pulsestyle-	_on
always_ff	endfunction	interconnect	_on-event	table
always_latch	endgenerate	interface	pure	tagged
and	endgroup	intersect	rand	task
assert	endinterface	join	randc	this
assign	endmodule	join_any	randcase	throughout
assume	endpackage	join_none	randsequence	time
automatic	endprimitive	large	r_cmos	timeprecision
before	endprogram	let	real	timeunit
begin	endproperty	liblist	realtime	tran
bind	endspecify	library	ref	tranif0
bins	endsequence	local	reg	tranif1
binsof	endtable	localparam	reject_on	tri
bit	endtask	logic	release	tri0
break	enum	longint	repeat	tri1
buf	event	macromodule	restrict	triand
bufif0	eventually	matches	return	trior
bufif1	expect	medium	r_cmos	trireg
byte	export	modport	r_cmos	type
case	extends	module	rtran	typedef
casex	extern	nand	rtranif0	union
casez	final	negedge	rtranif1	unique
cell	first_match	nettype	s_always	unique0
chandle	for	new	s_eventually	unsigned
checker	force	nexttime	s_nexttime	until
class	foreach	nmos	s_until	until_with
clocking	forever	nor	s_until_with	untyped
cmos	fork	noshowcan-	scalared	use
config	forkjoin	celled	sequence	wire
const	function	not	shortint	var
constraint	generate	notif0	shortreal	vectored
context	genvar	notif1	showcan-	virtual
continue	global	null	celled	void
cover	highz0	or	signed	wait
covergroup	highz1	output	small	wait_order
coverpoint	if	package	soft	wand
cross	iff	packed	solve	weak
deassign	ifnone	parameter	specify	weak0
default	ignore_bins	pmos	specparam	weak1
defparam	illegal_bins	posedge	static	while
design	implements	primitive	string	wildcard
disable	implies	priority	strong	wire
dist	import	program	strong0	with
do	incdir	property	strong1	within
edge	include	protected	struct	wor
else	initial	pul10	super	xnor
end	inout	pul11	supply0	xor
endcase	input	pulldown	supply1	

Table B-1: Complete list of SystemVerilog-2012/2017 reserved keywords
(IEEE 1800-2012)

B.2 Verilog-1995 reserved keywords

Table B-2 lists the reserved keywords used in the original Verilog-1995 standard. The compiler directive '`'begin_keywords "1364-1995"`' instructs compilers to reserve only the keywords listed in this table.

<code>always</code>	<code>ifnone</code>	<code>rpmos</code>
<code>and</code>	<code>initial</code>	<code>rtran</code>
<code>assign</code>	<code>inout</code>	<code>rtranif0</code>
<code>begin</code>	<code>input</code>	<code>rtranif1</code>
<code>buf</code>	<code>integer</code>	<code>scalared</code>
<code>bufif0</code>	<code>join</code>	<code>small</code>
<code>bufif1</code>	<code>large</code>	<code>specify</code>
<code>case</code>	<code>macromodule</code>	<code>specparam</code>
<code>casex</code>	<code>medium</code>	<code>strong0</code>
<code>casez</code>	<code>module</code>	<code>strong1</code>
<code>cmos</code>	<code>nand</code>	<code>supply0</code>
<code>deassign</code>	<code>negedge</code>	<code>supply1</code>
<code>default</code>	<code>nmos</code>	<code>table</code>
<code>defparam</code>	<code>nor</code>	<code>task</code>
<code>disable</code>	<code>not</code>	<code>time</code>
<code>edge</code>	<code>notif0</code>	<code>tran</code>
<code>else</code>	<code>notif1</code>	<code>tranif0</code>
<code>end</code>	<code>or</code>	<code>tranif1</code>
<code>endcase</code>	<code>output</code>	<code>tri</code>
<code>endmodule</code>	<code>parameter</code>	<code>tri0</code>
<code>endfunction</code>	<code>pmos</code>	<code>tri1</code>
<code>endprimitive</code>	<code>posedge</code>	<code>triand</code>
<code>endspecify</code>	<code>primitive</code>	<code>trior</code>
<code>endtable</code>	<code>pull0</code>	<code>trireg</code>
<code>endtask</code>	<code>pull1</code>	<code>vectored</code>
<code>event</code>	<code>pullup</code>	<code>wait</code>
<code>for</code>	<code>pulldown</code>	<code>wand</code>
<code>force</code>	<code>rcmos</code>	<code>weak0</code>
<code>forever</code>	<code>real</code>	<code>weak1</code>
<code>fork</code>	<code>realtime</code>	<code>while</code>
<code>function</code>	<code>reg</code>	<code>wire</code>
<code>highz0</code>	<code>release</code>	<code>wor</code>
<code>highz1</code>	<code>repeat</code>	<code>xnor</code>
<code>if</code>	<code>rnmos</code>	<code>xor</code>

Table B-2: Complete list of Verilog-1995 reserved keywords
(IEEE 1364-1995)

B.3 Verilog-2001 reserved keywords

Table B-3 lists only the reserved keywords that were added with the Verilog-2001 language. The compiler directive '**begin_keywords** "1364-2001"' instructs compilers to reserve the keywords listed in this table, plus the keywords reserved in previous versions.

automatic	genvar	noshowcancelled
cell	inmdir	pulsestyle_onevent
config	include	pulsestyle_onedetect
design	instance	showcancelled
endconfig	liblist	signed
endgenerate	library	unsigned
generate	localparam	use

Table B-3: Additional reserved keywords added with Verilog-2001
(IEEE 1364-2001)

B.4 Verilog-2005 reserved keywords

The Verilog-2005 reserved only one additional keyword, which is listed Table B-4. The compiler directive '**begin_keywords** "1364-2005"' instructs compilers to reserve the keyword listed in this table, plus all keywords reserved in all previous versions, as listed in Tables B-2 and B-3.

uwire

Table B-4: Additional reserved keywords added with Verilog-2005
(IEEE 1364-2005)

B.5 SystemVerilog-2005 reserved keywords

SystemVerilog added substantial new capabilities to Verilog-2005, and reserved many more keywords. Table B-5 lists only the reserved keywords that were added with the SystemVerilog-2005 standard. The compiler directive `'begin_keywords "1800-2005"` instructs compilers to reserve the keywords listed in this table, plus all keywords reserved in previous versions, as listed in Tables B-2, B-3 and B-4.

<code>alias</code>	<code>endproperty</code>	<code>protected</code>
<code>always_comb</code>	<code>endsequence</code>	<code>pure</code>
<code>always_ff</code>	<code>enum</code>	<code>rand</code>
<code>always_latch</code>	<code>expect</code>	<code>randc</code>
<code>assert</code>	<code>export</code>	<code>randcase</code>
<code>assume</code>	<code>extends</code>	<code>randsequence</code>
<code>before</code>	<code>extern</code>	<code>ref</code>
<code>bind</code>	<code>final</code>	<code>return</code>
<code>bins</code>	<code>first_match</code>	<code>sequence</code>
<code>binsof</code>	<code>foreach</code>	<code>shortint</code>
<code>bit</code>	<code>forkjoin</code>	<code>shortreal</code>
<code>break</code>	<code>iff</code>	<code>solve</code>
<code>byte</code>	<code>ignore_bins</code>	<code>static</code>
<code>chandle</code>	<code>illegal_bins</code>	<code>string</code>
<code>class</code>	<code>import</code>	<code>struct</code>
<code>clocking</code>	<code>inside</code>	<code>super</code>
<code>const</code>	<code>int</code>	<code>tagged</code>
<code>constraint</code>	<code>interface</code>	<code>this</code>
<code>context</code>	<code>intersect</code>	<code>throughout</code>
<code>continue</code>	<code>join_any</code>	<code>timeprecision</code>
<code>cover</code>	<code>join_none</code>	<code>timeunit</code>
<code>covergroup</code>	<code>local</code>	<code>type</code>
<code>coverpoint</code>	<code>logic</code>	<code>typedef</code>
<code>cross</code>	<code>longint</code>	<code>union</code>
<code>dist</code>	<code>matches</code>	<code>unique</code>
<code>do</code>	<code>modport</code>	<code>var</code>
<code>endclass</code>	<code>new</code>	<code>virtual</code>
<code>endclocking</code>	<code>null</code>	<code>void</code>
<code>endgroup</code>	<code>package</code>	<code>wait_order</code>
<code>endinterface</code>	<code>packed</code>	<code>wildcard</code>
<code>endpackage</code>	<code>priority</code>	<code>with</code>
<code>endprimitive</code>	<code>program</code>	<code>within</code>
<code>endprogram</code>	<code>property</code>	

Table B-5: Additional reserved keywords added with SystemVerilog-2005
(IEEE 1800-2005)

B.6 SystemVerilog-2009 reserved keywords

Table B-6 lists the reserved keywords that were added with the SystemVerilog-2009 standard. The compiler directive '**begin_keywords** "1800-2009"' instructs compilers to reserve the keywords listed in this table, plus all keywords reserved in all previous versions.

accept_on	reject_on	sync_accept_on
checker	restrict	sync_reject_on
endchecker	s_always	unique0
eventually	s_eventually	until
global	s_nexttime	until_with
implies	s_until	untyped
let	s_until_with	weak
nexttime	strong	

Table B-6: Additional reserved keywords added with SystemVerilog-2009
(IEEE 1800-2009)

B.7 SystemVerilog-2012 reserved keywords

The SystemVerilog-2012 standard reserved four more keywords. Table B-7 lists only the reserved keywords that were added with SystemVerilog-2012. The compiler directive '**begin_keywords** "1800-2012"' instructs compilers to reserve the keywords listed in this table, plus all keywords reserved in previous versions.

implements	nettype
interconnect	soft

Table B-7: Additional reserved keywords added with SystemVerilog-2012
(IEEE 1800-2012)

B.8 SystemVerilog-2017 reserved keywords

The SystemVerilog-2017 standard does not add any additional reserved keywords to the SystemVerilog-2012 standard. Table B-7 lists only the reserved keywords that were added with SystemVerilog-2012. The compiler directive '**begin_keywords** "1800-2017"' instructs a compiler to use the same reserved keyword list as '**begin_keywords** "1800-2012"'.

Appendix C

X Optimism and X Pessimism in RTL Models

This appendix is a reprint of a conference paper published and presented at the Design and Verification Conference (DVCon), San Jose, California, USA, 2013. The original paper has been reformatted to fit the page size and heading/subheading conventions used in this book. The original paper is available at sutherland-hdl.com.

I'm Still In Love With My X!

(but, do I want my X to be an
optimist, a pessimist, or eliminated?)

Stuart Sutherland
SystemVerilog Trainer and Consultant
Sutherland HDL, Inc.
Portland, Oregon
stuart@sutherland-hdl.com

Abstract—This paper explores the advantages and hazards of X-optimism and X-pessimism, and of 2-state versus 4-state simulation. A number of papers have been written over the years on the problems of optimistic versus pessimistic X propagation in simulation. Some papers argue that Verilog/SystemVerilog is overly optimistic, while other papers argue that SystemVerilog can be overly pessimistic. Which viewpoint is correct? Just a few years ago, some simulator companies were promising that 2-state simulations would deliver substantially faster, more efficient simulation run-times, compared to 4-state simulation. Now it seems the tables have turned, and Verilog/SystemVerilog simulators are providing modes that pessimistically propagate X values, with the promise that 4-state simulation will more accurately and efficiently detect obscure design bugs. Which promise is true? This paper answers these questions.

Keywords—*Verilog, SystemVerilog, RTL simulation, 2-state, 4-state, X propagation, X optimism, X pessimism, register initialization, randomization, UVM*

C.1 Introducing My X

SystemVerilog uses a four-value logic system to represent digital logic behavior: 0, 1, Z (high-impedance) and X (unknown, uninitialized, or don't care). Values 0, 1 and Z are abstractions of the values that exist in actual silicon (abstract, because these values do not reflect voltage, current, slope, or other characteristics of physical silicon). The fourth value, X, is not an abstraction of actual silicon values. Simulators can use X to indicate a degree of uncertainty in how physical hardware would behave under specific circumstances, i.e., when simulation cannot predict whether an actual silicon value would be a 0, 1 or Z. For synthesis, an X value provides design engineers a way to specify "don't care" conditions, where the engineer is not concerned about whether actual hardware will have a 0 or a 1 value for a specific condition.

X values are useful, but can also be a challenge for design verification. Of particular concern is how X values propagate through digital logic in RTL and gate-level simulation models. A number of conference papers have been written on this topic. The title of this paper is inspired by two earlier papers on X propagation, "*The Dangers of Living with an X*" by Turpin [1] and "*Being Assertive with Your X*" by Mills [2], presented in 2003 and 2004, respectively. Both the SystemVerilog standard and SystemVerilog simulators have added many new features since those papers were written. This paper reiterates concepts and good advice from earlier papers, and adds coding guidelines that reflect the latest in the SystemVerilog language and software tool features.

Terminology. For the purposes of this paper, *X-optimism* is defined as any time simulation converts an X value on an expression or logic gate input into a 0 or a 1 on the result. *X-pessimism* is defined as any time simulation passes an X on an input to an expression or logic gate through to the result. As will be shown in this paper, sometimes X-optimism is desirable, and sometimes it is not. Conversely, in different circumstances, X-pessimism can be the right thing or the wrong thing.

Note: In this paper, the term "*value sets*" is used to refer to 2-state values (0 and 1) and 4-state values (0, 1, Z, X). The term "*data types*" is used as a general term for all net types, variable types, and user-defined types. The terms *value sets* and *data types* are not used in the same way in the official IEEE SystemVerilog standard [3], which is written primarily for companies that implement software tools such as simulators and synthesis compilers. The SystemVerilog standard uses terms such as "*types*", "*objects*" and "*kinds*", which have specific meaning for those that implement tools, but which this author feels are neither common place nor intuitive for engineers that use the SystemVerilog language.

C.2 How did my one (or zero) become my X?

Logic X is a simulator's way of saying that it cannot predict whether the value in actual silicon would be 0 or 1. There are several conditions where simulation will generate a logic X:

- Uninitialized 4-state variables
- Uninitialized registers and latches
- Low power logic shutdown or power-up
- Unconnected module input ports
- Multi-driver Conflicts (Bus Contention)
- Operations with an unknown result
- Out-of-range bit-selects and array indices
- Logic gates with unknown output values
- Setup or hold timing violations
- User-assigned X values in hardware models
- Testbench X injection

C.2.1 Uninitialized 4-state variables

The SystemVerilog keywords that will declare or infer a 4-state variable are: **var**, **reg**, **integer**, **time**, and, depending on context, **logic**.

The **var** keyword explicitly declares a variable. It can be used by itself, or in conjunction with other keywords. In most contexts, the **var** keyword is optional, and is seldom used.

```
var integer i1; // same as "integer i1"
var i2;          // same as "var reg i2"
```

Example 1: The var variable type

The **logic** keyword is *not* a variable type or a net type. Nor is the **bit** keyword. **logic** and **bit** define the digital value set that a net or variable models; **logic** indicates a 4-state value set (0, 1, Z, X) and **bit** indicates a 2-state value set (0, 1). The **reg**, **integer**, **time** and **var** variable types infer a 4-state **logic** value set.

The **logic** keyword can be used in conjunction with the **var**, **reg**, **integer** or **time** keyword or a net type keyword (such as **wire**) to explicitly indicate the value set of the variable or net. For example:

```
var logic [31:0] v; // 4-state 32-bit variable
wire logic [31:0] w; // 4-state 32-bit net
```

Example 2: 4-state variable and net declarations

The **logic** (or **bit**) keyword can be used without the **var** or a net type keyword. In this case, either a variable or net is inferred, based on context. If **logic** or **bit** is used in conjunction with an **output** port, an **assign** keyword, or as a local declaration, then a variable is inferred. If **logic** is used in conjunction with an **input** or **inout** port declaration, then a net of the default net type is inferred (typically **wire**). An **input** port can also be declared with a 4-state variable type, using either the keyword triplet **input var logic** or the keyword pair **input var**.

```
module m1 (
    input logic [7:0] i; // 4-state wire inferred
    output logic [7:0] o; // 4-state var inferred
);
    logic [7:0] t; // 4-state var inferred
    ...
endmodule
```

Example 3: Default port data types

The SystemVerilog standard [3] defines that 4-state variables begin simulation with an uninitialized value of X. This rule is one of the biggest causes of X values at the start of simulation.

C.2.2 Uninitialized registers and latches

“Register” and “latch” refer to models that store logic values over time. This storage behavior can be represented as either abstract RTL procedural code or as low-level User-defined primitives (UDPs). Most often, the storage of registers and latches is modeled with 4-state variables, such as the **reg** data type.

Note: The **reg** keyword does not, in and of itself, indicate a hardware register. The **reg** data type is simply a general purpose 4-state variable with a user-defined vector size. A **reg** variable can be used to model pure combinational logic, as well as hardware registers and latches.

In SystemVerilog, 4-state variables begin simulation with an uninitialized value of X. This means that register and latch outputs will have a logic X at the start of simulation. Register outputs will remain an X until the register is either reset or a known input value is clocked into the register. Latch outputs will remain an X until the latch is both enabled and the latch input is a known value. This is true for both abstract RTL simulations and gate-level simulations.

There are ways to handle uninitialized registers and latches. Section C.5 (page 426) discusses using 2-state simulation, section C.7 (page 430) discusses using proprietary simulation options, and section C.10.2 (page 435) discusses using the SystemVerilog UVM standard.

C.2.3 Low power logic shutdown or power-up

Simulation of low-power models can result in registers and latches that had been initialized changing back to logic X during simulation. The effect is similar to uninitialized registers and latches at the beginning of simulation, except that the X storage occurs sometime during simulation, instead of at the beginning of simulation. Once a register has gone back to storing an X, the outputs will remain at X until the register is either reset or a known input value is clocked into the register. A latch that has stored an X will remain an X until the latch is both enabled and the latch input is a known value.

This X lock-up when a design block is powered back up from a low-power mode is especially problematic when registers are only set by loading a value, instead of being reset or preset. This behavior is a 4-state simulation anomaly. Actual silicon registers or latches would power up from a low power mode with a 0 or a 1.

C.2.4 Unconnected module input ports

Unconnected module inputs generally represent a floating input, and result in a simulation value of Z on that input (assuming the input data type is **wire**, which is the default in SystemVerilog). When an input floats at high-impedance, it will often result in a logic X elsewhere within the model.

C.2.5 Multi-driver Conflicts (Bus Contention)

SystemVerilog net types allow multiple outputs to drive the same net. Each net type (**wire**, **tri**, **tri0**, **tri1**, **wor**, **wand** and **trireg**) has a built-in resolution function to resolve the combined value of the multiple drivers. If the final value that would occur in silicon cannot be predicted, the simulation value will be an X. (The SystemVerilog-2012 standard also allows engineers to specify user-defined net types and resolution functions, which might also resolve to a logic X under specific conditions).

C.2.6 Operations with an unknown result

All SystemVerilog RTL operators are defined to work with 4-state values for the operands. Some operators have optimistic rules and others have pessimistic rules. Sections C.3 (page 405) and C.4 (page 417) discuss when X values can result from optimistic operations and pessimistic operations, respectively.

C.2.7 Out-of-range bit-selects and array indices

A *bit-select* is used to read or write individual bits out of a vector. A *part-select* reads or writes a group of contiguous bits from a vector. An *array index* is used to access specific members or slices of an array.

When reading bits from a vector, if the index is outside the range of bits in the vector, a logic X is returned for each bit position that is out-of-range. When reading members of an array, if the index is outside the range of addresses in the array, a logic X is returned for the entire word being read. Of course, even in-range bit-selects, part-selects and array selects can result in an X value being returned, if the vector or array being selected contains X values.

Section C.4.8 (page 425), on X-pessimism, discusses reading vector bits and array members with unknown indices. Section C.3.6 (page 414), on X-optimism, discusses writing to vector bits and array members with unknown or out-of-range indices.

C.2.8 Logic gates with unknown output values

SystemVerilog built-in primitives and User-defined primitives (UDPs) are used to model design functionality at a detailed level of abstraction. These primitives operate on 4-state values for the gate inputs. An input with a logic X or Z value can result in a logic X output value.

C.2.9 Setup or hold timing violations

SystemVerilog provides timing violation checks, such as \$setup, \$hold, and a few more. Typically, these constructs are used by model library developers for models of flip-flops, RAMs, and other devices that have specific timing requirements. These timing checks can be modeled to be either optimistic or pessimistic, should a timing violation occur. An optimistic timing check will generate a run-time violation report when a violation occurs, but leave the values of the model a known value. A pessimistic timing check will generate the run-time violation report and set one or more of the model outputs to X.

C.2.10 User-assigned X values in hardware models

A common source of X values in RTL simulation is user code that intentionally assigns logic X to a variable. There are two reasons for doing this: to trap error situations in a design such as a state condition that should never occur, and to indicate a “don’t care” situation for synthesis. A common example of a user-assigned X is a **case** statement, such as this 3-to-1 multiplexor:

```
always_comb begin
    case (select)
        2'b01: y = a;
        2'b10: y = b;
        2'b11: y = c;
        default: y = 'x; // don't care about any
                  // other values of select
    endcase
end
```

Example 4: User-assigned X values

In this example, a `select` value of `2'b00` is not used by the design, and should never occur. The default assignment of a logic X serves as a simulation flag, should `select` ever have a value of `2'b00`. The same default assignment of X serves as a don't care flag for synthesis. Synthesis tools see this X-assignment as an indication that logic minimization can be performed for any values of the case expression (`select`, in this example) that were not explicitly decoded.

C.2.11 Testbench X injection

A testbench will often send logic X values into the design being tested. One way this can occur is when a testbench uses 4-state variables to calculate and store stimulus values. These stimulus variables will begin simulation with a logic X, and will retain that X until the testbench assigns a known value to the variable. Often, a test might not make the first assignment to a stimulus variable until many hundreds of clock cycles into a simulation.

Some verification engineers will write a test to deliberately drive certain design inputs to an X value when the design should not be reading those specific inputs. This deliberate X injection can bring out errors in a design, should the design read that input at an incorrect time. For example, a design specification might be that the `data` input is only stored when `load_enable` is high. To verify this functionality was correctly implemented, the testbench could deliberately set the `data` input to an X while `data_enable` is low. If that X value propagates into the design, it can indicate the design has a bug.

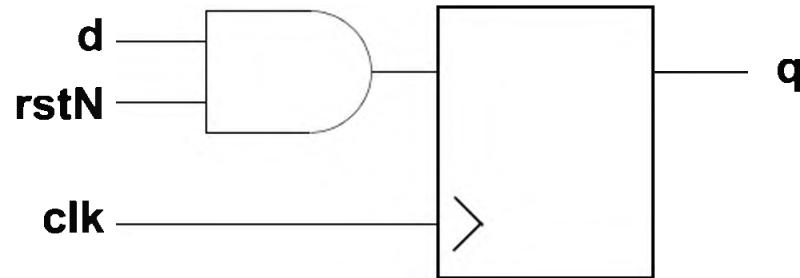
C.3 An optimistic X — is that good or bad?

Optimism: *an inclination to put the most favorable construction upon actions and events or to anticipate the best possible outcome.* [4]

In simulation, X-optimism is when there is some uncertainty on an input to an expression or gate (the silicon value might be either 0 or 1), but simulation comes up with a known result instead of an X. SystemVerilog is, in general, an optimistic language. There are many conditions where an ambiguous condition exists in a model, but SystemVerilog propagates a 0 or 1 instead of a logic X. A simple example of X-optimism is an AND gate. In SystemVerilog, an X ANDed with 0 will result in 0, not X.

An optimistic X can be a good thing! X-optimism can more accurately represents silicon behavior when an ambiguous condition occurs in silicon. Consider the following example, shown in Figure C-1.

Figure C-1: Flip-flop with synchronous reset



This circuit shows a flip-flop with synchronous, active-low reset. In actual silicon, the `d` input might be ambiguous at power-up, powering up as either a 0 or 1. If the `rstN` input of the AND gate is 0, however, the output of the AND gate will be 0, despite the ambiguous power-up value of `d`. This correctly resets the flip-flop at the next positive edge of `clk`.

In simulation, the ambiguous power-up value of `d` is represented as an X. If this X were to pessimistically propagate to the AND gate output, even when `rstN` is 0, the design would not correctly reset, which could cause all sorts of problems. Fortunately, SystemVerilog AND operators and AND gates are X-optimistic. If any input is 0, the result is 0. Because of X-optimism, simulation accurately models silicon behavior, and the simulated model functions correctly.

An optimistic X can also be a bad thing! X-optimism can, and will, hide design problems, especially at the abstract RTL level of verification. At best, these design bugs are not caught until late in the design cycle during gate-level simulations or when other low-level analysis tools are used. At worst, design ambiguities that were hidden by X-optimistic simulation might not be discovered until the design has been implemented in actual silicon.

Several X-optimistic SystemVerilog constructs are discussed in more detail in this section.

C.3.1 If...else statements

SystemVerilog has an optimistic behavior when the *control condition* of an `if...else` statement is unknown. The rule is simple: should the *control condition* evaluate to unknown, the `else` branch is executed.

```

always_comb begin
    if (sel) y = a; // if sel is 1
    else      y = b; // if sel is 0, X or Z
end
    
```

Example 5: if...else statement X-optimism

This optimistic behavior can hide a problem with `sel`, the *control condition*. In actual silicon, the ambiguous value of `sel` will be 0 or 1, and `y` will be set to a known result. How accurately does SystemVerilog's X-optimistic behavior match the behavior in actual silicon? The answer depends in part on how the `if...else` is implemented in silicon.

The behavior of this simple **if...else** statement might be implemented a number of ways in silicon. Figure C-2 and Figure C-3 illustrate two possibilities, using a Multiplexor or NAND gates, respectively.

Figure C-2: 2-to-1 selection — MUX gate implementation

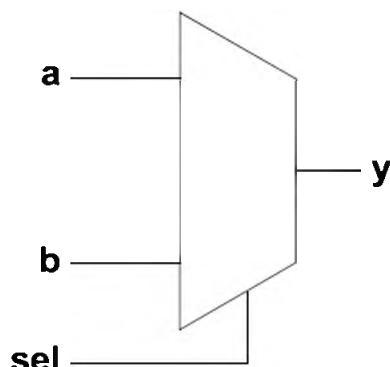


Figure C-3: 2-to-1 selection — NAND gate implementation

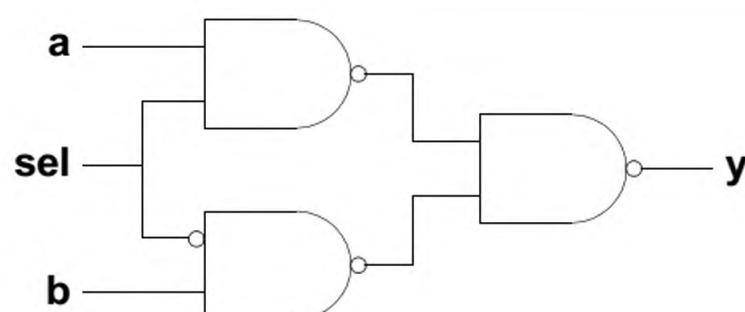


Table C-1 shows the simulation results for an X-optimistic **if...else** when the control expression (**sel**) is unknown, compared to the simulation behavior of MUX and NAND implementations and actual silicon behavior.

inputs			output (y)			actual silicon behavior	
sel	a	b	simulation behavior				
			if...else RTL	MUX gate	NAND gates		
X	0	0	0	0	0	0	
X	0	1	1	X	X	0 or 1	
X	1	0	0	X	X	0 or 1	
X	1	1	1	1	X	1	

Table C-1: if...else versus gate-level X propagation

Some important things to note from this table are:

- For all rows, the **if...else** statement propagates a known value instead of the X value of **sel**. This X-optimistic behavior could hide error conditions in the design.
- For rows 2 and 3, the X-optimistic **if...else** behavior only matches one of the possible values that could occur in actual silicon. *The other possible value is not propagated and therefore the design is not verified with that other possible value.*

- The MUX implementation of an **if...else** is the most accurate, and propagates an X when there is a potential of actual silicon having either a 0 or a 1.
- The NAND-gate implementation is overly pessimistic for when a and b are both 1 (row 4), and propagates an X value, even though the actual silicon would have a known value of 1.

Following is a more detailed example that illustrates how optimistic **if...else** X propagation can hide a design problem. The example is a program counter that: can be reset, can be loaded with a new count, or can increment the current count. The program counter is instantiated within a larger design, `cpu`, that does not need the ability to load the program counter, and leaves the `loadN` and `new_count` inputs unconnected.

```
module program_counter (
    input logic      clock, resetN, loadN,
    input logic [15:0] new_count,
    output logic [15:0] count
);
    always_ff @ (posedge clock or negedge resetN)
        if (!resetN)      count <= 0;
        else if (!loadN)  count <= new_count;
        else              count <= count + 1;
endmodule: program_counter

module cpu (...);
    ...
    program_counter pc (.clock(m_clk),
                        .resetN(m_rstN),
                        .loadN(/* not used */),
                        .new_count(/* not used */),
                        .count(next_addr) );
    ...
endmodule: cpu
```

Example 6: Program counter with unused inputs, X-optimistic rules

In actual silicon, each bit of these unconnected inputs will have ambiguous values — they will be sensed as either 0 or 1, depending on factors such as transistor technology and interconnect capacitance. If actual silicon senses `loadN` as 1, the counter will increment on each clock, which is the desired functionality. If silicon senses `loadN` as 0, the counter will load an ambiguous `new_count` value on each clock, and the program counter will not work correctly.

X-optimism hides this design bug! The `loadN` and `new_count` inputs will float at high-impedance (assuming the default net type of `wire`). Instead of seeing `loadN` as being either 0 or 1, the way silicon would, RTL simulation always takes the **else** branch, which increments the counter. This X-optimistic behavior happens to be the desired behavior for this design, but **it is a dangerous simulation hazard!** In RTL simulation, the design appears to work correctly, and a serious design bug could go undetected.

Later sections of this paper show several ways to detect problems with **if** conditions, so that design bugs of this nature do not remain hidden by an optimistic X.

C.3.2 Case statements without a default-X assignment

The control value of a **case** statement is referred to as the *case expression*. The values to which the control value is compared are referred to as *case items*.

```
always_comb begin
    case (sel)          // sel is the case expression
        1'b1: y = a;   // 1'b1 is a case item
        1'b0: y = b;   // 1'b0 is a case item value
    endcase
end
```

Example 7: case statement X-optimism

Functionally, **case** and **if...else** represent similar logic. However, SystemVerilog's X-optimistic behavior for a **case** statement without a **default** branch is very different than an **if...else** decision when the select control is unknown, as is shown in Table C-2.

inputs			previous value of y	output (y)			
sel	a	b		case RTL	if...else RTL	MUX gate	silicon
X	0	0	0	0	0	0	0
X	0	1	0	0	1	X	0 or 1
X	1	0	0	0	0	X	0 or 1
X	1	1	0	0	1	1	1
X	0	0	1	1	0	0	0
X	0	1	1	1	1	X	0 or 1
X	1	0	1	1	0	X	0 or 1
X	1	1	1	1	1	1	1

Table C-2: case versus if...else versus MUX X propagation

Observe in this table that a **case** statement without a default branch retains its previous value whenever the case expression is unknown.

A **case** statement with a **default** assignment of a known value is also optimistic, but in a different way. Consider the following example:

```
always_comb begin
    case (sel)
        1'b1:      y = a;
        default:  y = b;
    endcase
end
```

Example 8: case statement with default assignment of a known value X-optimism

If any bit in the *case expression* is an X or Z, the value of the **default** *case item* is assigned to y, instead of keeping the previous value. Table C-3 show this difference.

inputs			previous value of y	output (y)		
sel	a	b		case without default	case with default	silicon
X	0	0	0	0	0	0
X	0	1	0	0	1	0 or 1
X	1	0	0	0	0	0 or 1
X	1	1	0	0	1	1
X	0	0	1	1	0	0
X	0	1	1	1	1	0 or 1
X	1	0	1	1	0	0 or 1
X	1	1	1	1	1	1

Table C-3: case with default versus case without default

As can be seen in this table, **case** statements with or without a default assignment are X-optimistic, and will hide problems in the *case expression*. With either coding style, the X-optimism does not accurately reflect the ambiguity that exists on the results in actual silicon, should a selection control be ambiguous.

Section C.3.2 (page 409) on pessimistic modeling styles will discuss what happens when a **case** statement **default** branch assigns an X as the decoded result. Sections C.7 (page 430) and C.9 (page 432) present other ways to reduce or eliminate this X-optimism problem with **case** statements.

C.3.3 Casex, casez and case...inside statements

SystemVerilog's **casex**, **casez** and **case...inside** statements allow specific bits to be masked out — i.e., ignored — from being compared for each case branch. Collectively, these three constructs are sometimes referred to as *wildcard case statements*.

With **casez**, any bit in the *case expression* or *case item* that is set to Z will be ignored. With **casex**, any bit in the *case expression* or *case item* that is either X or Z will be ignored. (In literal numbers, a ? can be used in place of the letter Z.)

```
always_comb begin
    casex (sel)      // sel is 3 bits wide
        3'b1???: y = a; // matches 100, 101, 110, 111
        3'b00?: y = b; // matches 000, 001
        3'b01?: y = c; // matches 010, 011
```

```

    default: $error("sel had unexpected value");
endcase
end

```

Example 9: casex statement X-optimism

By using “don’t care” values, the 3 *case items* above decode all 8 possible 2-state values of sel. Less obvious is that these *case items* also decode all possible unknown values of sel, because the don’t care bits in *case items* ignore all values in those bit positions, including X and Z values. Furthermore, any X or Z bits in the *case expression* are also considered to be don’t care bits, and are ignored in any comparisons. The values of sel will decode as:

- 3'b1?? matches sel values of:

```

100, 101, 110, 111,
10X, 11X, 1X0, 1X1, 1XX,
10Z, 11Z, 1Z0, 1Z1, 1ZZ,
1XZ, 1ZX,
X00, X01, X10, X11,
XXX, XZZ, XZX, XXZ,
ZZZ, ZZX, ZXZ, ZXX

```

- 3'b00? matches sel values of:

```

000, 001, 00X, 00Z,
0X0, 0X1, 0XX, 0XZ,
0Z0, 0Z1, 0ZZ, 0ZX

```

- 3'b01? matches sel values of:

```
010, 011, 01X, 01Z
```

- **default** does not match any values, because all possible 4-state values have already been decoded by the previous case items.

Using **casez** instead of **casel** changes the X-optimism. With **casez**, only Z values (also represented with a ?) in the *case expression* or *case items* are treated as don’t care values.

```

always_comb begin
  casez (sel)      // sel is 3 bits wide
    3'b1???: y = a; // matches 100, 101, 110, 111
    3'b00?: y = b; // matches 000, 001
    3'b01?: y = c; // matches 010, 011
    default: $error("sel had unexpected value");
  endcase
end

```

Example 10: casez statement X-optimism

With **casez**, the values each *case item* represents are:

- 3'b1?? matches sel values of:

```

100, 101, 110, 111,
10X, 11X, 1X0, 1X1, 1XX,

```

```
10Z, 11Z, 1Z0, 1Z1, 1ZZ,
1XZ, 1ZX,
ZZZ, ZZX, ZXZ, ZXX
```

- `3'b00?` matches `sel` values of:
000, 001, 00X, 00Z,
0Z0, 0Z1, 0ZZ, 0ZX
- `3'b01?` matches `sel` values of:
010, 011, 01X, 01Z
- `default` matches `sel` values of:
X00, X01, X10, X11,
XXX, XZZ, XZX, XXZ,
0X0, 0X1, 0XX, 0XZ,

Using `casez`, some, but not all, of the possibilities of `sel` having a bit with an X or Z value fall through to the `default`. Since `y` is not assigned a value in the default branch, the value of `y` would not be changed, and would retain its previous value.

The `case...inside` statement is also X-optimistic, but less so than `casex` or `casez`. With `case...inside`, only the bits in *case items* can have mask (don't care) bits. Any X or Z bits in the *case expression* are treated as literal values.

```
always_comb begin
    case (sel) inside
        3'b1???: y = a;
        3'b00?: y = b;
        3'b01?: y = c;
        default: $error("sel had unexpected value");
    endcase
end
```

Example 11: case...inside statement X-optimism

Using `case...inside`, the values each *case item* represents are:

- `3'b1???` matches `sel` values of:
100, 101, 110, 111,
10X, 11X, 1X0, 1X1, 1XX,
10Z, 11Z, 1Z0, 1Z1, 1ZZ,
1XZ, 1ZX
- `3'b00?` matches `sel` values of:
000, 001, 00X, 00Z
- `3'b01?` matches `sel` values of:
010, 011, 01X, 01Z
- `default` matches `sel` values of:
0X0, 0X1, 0XX, 0XZ,
0Z0, 0Z1, 0ZZ, 0ZX,
X00, X01, X10, X11,

XXX, XZZ, XZX, XXZ,
ZZZ, ZZX, ZXZ, ZXX

All forms of wildcard case statements are X-optimistic, but in different ways. The **case...inside** does the best job of modeling actual silicon optimism, but can still differ from true silicon behavior, and can hide problems with a case expression. Sections C.7 (page 430) and C.9 (page 432) discuss ways to reduce or eliminate this X-optimism problem with wildcard case statements.

C.3.4 Bitwise, unary reduction, and logical operators

Many, but not all, of SystemVerilog's RTL programming operators are X-optimistic. An X or Z bit in an operand might not propagate to an unknown result. For example, 0 ANDed with any value, including an X or Z, will result in 0, and 1 ORed with any value will result in 1. This optimistic behavior can accurately represent the silicon behavior of an actual AND or OR gate, but it can also hide the fact that there was a problem on the inputs to the RTL operation.

The optimistic operators are:

- Bitwise: AND (&), OR (|)
- Unary: AND (&), NAND (~&), OR (|), NOR (~|)
- Logical: AND (&&), OR (||), Implication (->), and Equivalence (<->)

The logical AND and OR operators evaluate each operand to determine if the operand is true or false. These operators have two levels of X-optimism that can hide X or Z values:

- An operand is considered true if any bit is a 1, and false if all bits are 0. For example, the value 4'b010x will evaluate as true, hiding the X in another bit.
- Logical operators “short circuit”, meaning that if the result of the operation can be determined after evaluating the first operand, the second operand is not evaluated.

The following example illustrates a few ways in which X-optimistic RTL operators could hide a problem in a design.

```
logic [3:0] a = 4'b0010;
logic [3:0] b = 4'b000x;
logic [1:0] opcode;
always_comb begin
    case (opcode) inside
        2'b00: y = a & b;
        2'b01: y = a | b;
        2'b10: y = &b;
        2'b10: y = a || b;
    endcase
end
```

Example 12: Bitwise, unary and logical operator X-optimism

For the values of `a` and `b` shown above:

- `a & b` results in `4'b0000` — the `X` in `b` is hidden, but the operation result accurately represents silicon behavior.
- `a | b` results in `4'b001x` — the `X` in `b` is propagated, accurately indicating there will be ambiguity in silicon behavior.
- `&b` results in `1'b0` — the `X` in `b` is hidden, but the operation result accurately represents silicon behavior.
- `a || b` results in `1'b1` — the `X` in `b` is hidden, but the operation result accurately represents silicon behavior.

Note that the X-optimism of these operators accurately models silicon behavior. As noted at the beginning of section C.3 (page 405), there are times that this optimism is desirable and necessary, in order for RTL simulation to work correctly, but the optimism can also obscure design problems.

Not all SystemVerilog operators are optimistic. Sections C.4.5 (page 422) and C.4.6 (page 423) list several operators that are X-pessimistic.

C.3.5 And, nand, or, nor, logic primitives

SystemVerilog's `and`, `nand`, `or` and `nor` gate-level primitives are used for low-level, timing-detailed modeling. These constructs are often used in ASIC, FPGA and custom model libraries. These primitives follow the same truth tables as their RTL operator counterparts, and have the same X-optimistic behavior.

C.3.6 User-defined primitives

SystemVerilog provides ASIC, FPGA and custom library developers a means to create custom user-defined primitives (UDPs). UDPs are defined using a 4-state truth table, allowing library developers to define specific behavior for `X` and `Z` input values. It is common for developers to “reduce pessimism” by defining known output values for when an input is `X` or `Z`, or when there is a transition to or from an `X` or `Z`. As with other X-optimistic constructs, UDPs with reduced pessimism might accurately model actual silicon behavior, but can hide inputs that have an `X` or `Z` value.

C.3.7 Array index with X or Z bits for write operations

SystemVerilog is X-optimistic when making an assignment to an array with an ambiguous array index. If the index has any bits that are `X` or `Z`, the write operation is ignored, and no location in the array is modified.

```
logic [7:0] RAM [0:255];
logic [7:0] data = 8'b01010101
logic [7:0] addr = 8'b0000000x;
```

```
always_latch
  if (write && enable) RAM[addr] = data;
```

Example 13: Array index ambiguity X-optimism

In this example, only the least-significant bit of `addr` is unknown. A pessimistic approach would have been to write an unknown value into the `RAM` locations that might have been affected by this unknown address bit (addresses 0 and 1 in this example). SystemVerilog's X-optimistic rule, however, acts as if no write operation had occurred. This completely hides the fact that the address has a problem, and does not accurately model silicon behavior.

C.3.8 Net data types

Net types are used to connect design blocks together. In the original Verilog language, net data types were also required to be used internally within a module for all input ports. In SystemVerilog, module input ports can be either a net or a variable, but the default is still a net type.

Net types have driver resolution functions, which control how simulation resolves multiple drivers on the same net. Multi-driver resolution is important for specific design situations, such as shared data and address busses that can be controlled by more than one device output. When single-source logic is intended, however, the resolution function of a net can optimistically hide design problems by propagating a resolved value instead of an X.

The most commonly used net type in SystemVerilog is the `wire` type. The multi-driver resolution for `wire` is that driven values of a stronger strength take precedence over driven values of a weaker strength (logic 0 and logic 1 each have 8 strength levels). If, for example, two sources drive the same `wire` net, and one value is a weak-0 and the other a strong-1, the `wire` resolves to the strong-1 value. If two values of equal strength but opposing logic values are driven, the wire to resolves to a logic X.

Consider the following module port declarations:

```
module program_counter (
  input           clock, resetN, loadN,
  input logic [15:0] new_count,
  output logic [15:0] count
);
  ...
endmodule: program_counter
```

Example 14: Program counter with default wire net types

In Example 14, `clock`, `resetN` and `loadN` are input ports, but no data type has been defined. These signals will all default to `wire` nets. The signal `new_count` is declared as `input logic`, and will also default to `wire` (`logic` only defines that `new_count` can have 4-state values, but does not define the data type of `new_count`). Conversely, `count` is declared as `output logic`. Module output ports default to a variable of type `reg`, unless explicitly declared a different data type.

(**Note:** The default data type rules changed between the SystemVerilog-2005 and SystemVerilog-2009 standards for when `logic` is used as part of port declaration.)

Design bugs can easily occur when a mistake is made and a `wire` net, that was intended to only have one driver, is unintentionally driven by two sources. Since `wire` types support and resolve multiple drivers, simulation will only propagate an X if the two values are of the same strength and opposing values. Any other combination will resolve to a known value, and hide the fact that there were unintentional multiple drivers.

C.3.9 Posedge and negedge edge sensitivity

In SystemVerilog, the `posedge` keyword represents any transition that might be sensed as a positive going change in silicon. Value changes of 0->1, 0->Z, Z->1, 0->X, and X->1 are all positive edge transitions. A `negedge` transition includes the value changes of 1->0, 1->Z, Z->0, 1->X, and X->0.

The following example illustrates a simple RTL register with an asynchronous active-low reset.

```
always_ff @(posedge clk or negedge rstN)
  if (!rstN) q <= 0;
  else       q <= d;
```

Example 15: Edge sensitivity X-optimism

Table C-4 shows SystemVerilog's X-optimistic RTL behavior and actual silicon behavior if `clk` transitions from 0 to X (indicating that in silicon, the new value of `clk` might be either 0 or 1, but simulation is not certain which one). This table assumes `rstN` is high (inactive), and only shows the effects of a transition on the clock input. For this table, all signals are 1-bit wide.

inputs		old q	output (q)	
clk	d	q	RTL	silicon
0->X	0	0	0	0
0->X	0	1	0	0 or 1
0->X	1	0	1	0 or 1
0->X	1	1	1	1

Table C-4: Ambiguous clock edge X-optimism

As shown in this table, SystemVerilog's X-optimism rules for transitions will behave as if a clock edge occurred every time there is an ambiguous possibility of a positive edge on the clock. The ambiguous clock is hidden, instead of propagating the ambiguity onto the `q` output in the form of an X value.

The behavior of an ambiguous asynchronous reset is more subtle. Actual silicon would either reset or hold its currently stored value. SystemVerilog RTL semantics

behave quite differently. If the asynchronous `rstN` transitions from 1->Z, Z->0, 1->X, or X->0, the following results will occur:

inputs		old q	output (q)	
rstN	d	q	RTL	silicon
1->X	0	0	0	0
1->X	0	1	0	0 or 1
1->X	1	0	1	0
1->X	1	1	1	0 or 1

Table C-5: Ambiguous reset edge X optimism

Table C-5 shows that the ambiguous transition from 1 to X on the reset acts as if a positive edge of the clock occurred. This X-optimism not only hides that there was a problem with the reset, it does not at all behave like actual silicon!

C.4 A pessimistic X — is that any better?

Pessimism: *an inclination to emphasize adverse aspects, conditions, and possibilities or to expect the worst possible outcome.* [4]

In simulation, X-pessimism occurs when simulation yields an X where there is no uncertainty in actual silicon behavior. A common misconception is that SystemVerilog RTL code is always X-optimistic, and gate-level code is always X-pessimistic. This is not true. Some RTL operators and programming statements are optimistic, but others are pessimistic. Likewise, some gate-level primitives and UDPs are optimistic and some are pessimistic.

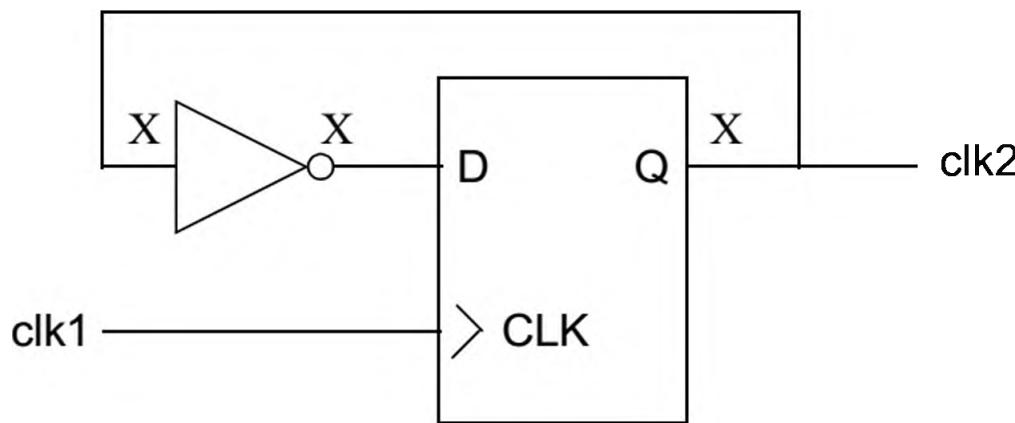
While X-optimism often accurately represents actual silicon behavior, the optimism can hide X values by propagating a known result. X-pessimism, on the other hand, guarantees that all ambiguities (one or more bits that are X or Z) will propagate to downstream code, helping to ensure that the problem will be detected, so that it can be debugged and corrected. X-pessimism will not hide design bugs, but there are at least three difficulties that can arise from X-pessimism.

One difficulty of X-pessimism is the point where verification first observes the X might be far downstream from the original source of the problem. An engineer might have to tediously trace an X value back through many lines of code, and over many clock cycles, to find where and when the X originated.

Another difficulty is that X-pessimism can propagate X results, where actual silicon would work without a problem. This section will show several examples where an X value should not have been propagated, but X-pessimism does so anyway. A great deal of engineering time can be lost debugging the cause of a pessimistic X, only to find out that there is no actual design problem.

A third difficulty with X-pessimism is the potential of simulation locking up in an unknown condition, where actual silicon, though perhaps ambiguous about having 0 or 1 values, will function correctly and not lock up. Figure C-4 illustrates a common X lock-up situation, a clock divider (divide-by-two, in this example).

Figure C-4: Clock divider with pessimistic X lock-up



In actual silicon, the internal storage of this flip-flop might power up as either a 0 or a 1. Whichever value it is, `clk2` will change value every second positive edge of `clk1`, and give the desired behavior of a divide-by-two. In simulation, however, the ambiguity of starting as either a 0 or 1 is represented as an X. The pessimistic inverter will propagate this X to the `D` input. Each positive edge of `clk1` will propagate this X onto `Q`, which once again feeds back to the input of the inverter. The result is that `clk2` is stuck at an X.

The failure of `clk2` to toggle between 0 and 1 will likely lock up downstream registers that are controlled by `clk2`. The X-pessimistic simulation will be locked up in an X state, where actual silicon would not have a problem. This X-pessimism exists at both the RTL level and at the gate level. The invert operator and the `not` inverter primitive are both X-pessimistic. An RTL assignment statement, such as `Q <= D`, and the typical gate-level flip-flop will both propagate an X when `D` is an X.

Several overly pessimistic SystemVerilog constructs that can cause simulation problems are discussed in this section.

C.4.1 If...else statements with X assignments

Section C.3.1 (page 406) showed how SystemVerilog `if...else` statements are, by default, X-optimistic, and can propagate known values, even though a decision condition has ambiguity (one or more bits at X or Z). It was also shown that this X-optimistic behavior did not always accurately represent silicon behavior.

It is possible to code decision statements to be more X-pessimistic. Consider the following example:

```

always_comb begin
    if (sel)          y = a;
    else
        // synthesis translate_off
        if (!sel)
            // synthesis translate_on
            y = b;
        // synthesis translate_off
        else          y = 'x;
        // synthesis translate_on
end

```

Example 16: if...else statement with X-pessimism and synthesis pragmas

Assuming that `sel` is only 1-bit wide, the `if (sel)` will evaluate as true if, and only if, `sel` is 1. The first `else` branch is taken if `sel` is 0, X or Z (X-optimistic). This first `else` branch then tests for `if (!sel)`, which will evaluate as true if, and only if, `sel` is 0. If `sel` is X or Z, the last `else` branch will be taken. This last branch assigns `y` to X, thus propagating the ambiguity of `sel`. This `if...else` statement is now X-pessimistic, propagating X values rather than known values when there is a problem with the select condition.

Note that the additional code to make the `if...else` decision be X-pessimistic might not yield optimal synthesis results. Therefore, the additional checking must be hidden from synthesis, using either conditional compilation (`'ifdef` commands) or synthesis “pragmas”. A pragma is a tool-specific command hidden within a comment or attribute. The synthesis pragma is ignored by simulation, but tells synthesis compilers to skip over any code that should not be synthesized.

C.4.2 Conditional operator

Coding an X-pessimistic `if...else` statement might not be the best choice for all circumstances. The X-pessimism will not hide a problem with the select condition the way an X-optimistic statement would, as described in section C.3.1 (page 406). However, the pessimistic model will also propagate X values at times where there is no ambiguity in hardware. This can occur when the select condition is unknown, but the values assigned in both branches are the same. The value propagated in hardware would be that value, with no ambiguity.

Turpin, in his paper “*The Dangers of Living with an X*” [1], recommends using the conditional operator (`? :`) instead of `if...else` statements in combinational logic. The conditional operator is a mix of X-optimistic and X-pessimistic behavior.

The syntax is:

condition ? expression1 : expression2

- If the *condition* evaluates as true (any bit is a 1), the operator returns the value of *expression1*.

- If the *condition* evaluates as false (all bits are 0), the operator returns the value of *expression2*.
- If the *condition* evaluates to unknown, the operator does a bit-by-bit comparison of the values of *expression1* and *expression2*. For each bit position, if that bit is 0 in both expressions, then a 0 is returned for that bit. If both bits are 1, a 1 is returned. If the corresponding bits in each expression are different, or Z, or X, then an X is returned for that bit.

The following example and table compare the X-optimistic behavior of **if...else**, with a pessimistic **if...else**, the mixed-optimism conditional operator, and actual silicon. The table is based on all signals being 1-bit wide.

```

always_comb begin      // X-optimistic if...else
    if (sel) y1 = a;
    else      y1 = b;
end

always_comb begin      // X-pessimistic if...else
    if (sel)      y2 = a;
    else if (!sel) y2 = b;
    else          y2 = 'x;
end

always_comb begin      // mixed pessimism ?:
    y3 = sel? a: b;
end

```

inputs			output (y1, y2, y3)			
sel	a	b	optimistic if...else	pessimistic if...else	?:	silicon
X	0	0	0	X	0	0
X	0	1	1	X	X	0 or 1
X	1	0	0	X	X	0 or 1
X	1	1	1	X	1	1

Table C-6: Conditional operator X propagation compared to optimistic if...else and pessimistic if...else

As can be seen in this table, the conditional operator represents a mix of X-optimism and X-pessimism, and more accurately represents the ambiguities of actual silicon behavior, given an uncertain selection condition. For this reason, Turpin [1] recommends using the conditional operator instead **if...else** in combinational logic.

This author does not agree with Turpin's coding guideline for two reasons. First, complex decode logic often involves multiple levels of decisions. Coding with **if...else** and **case** statements can help make complex logic more readable, easier to debug, and easier to reuse. Coding the same logic with nested levels of conditional

operators obfuscates code and adds a risk of coding errors. Furthermore, synthesis compilers might not permit or properly translate nested conditional operators.

A second reason the conditional operator should not always be used in place of **if...else** is when the condition is based on a signal or expression that is more than one bit wide. The condition is evaluated as a true/false expression. In a multi-bit value, if any bit is 1, the condition is considered to be true, even if some other bits are X or Z. The conditional operator will optimistically return the value of expression 1, rather than propagate an X.

Sections C.7 (page 430) and C.9 (page 432) show ways to keep the benefits of **if...else** and **case** statements, and also have the benefit of the conditional operator's balance of X-optimism and X-pessimism.

C.4.3 Case statements with X assignments

A **case** statement can also be coded to be X-pessimistic:

```
always_comb begin
    case (sel)
        2'b00: y = a;
        2'b01: y = b;
        2'b10: y = c;
        2'b11: y = d;
        default: y = 'x;
    endcase
end
```

Example 17: case statement with X-pessimism

If *sel* (the *case expression*) should have any bit at X or Z, none of the explicit *case item* values will match. Without a **default case item**, no branch of the **case** statement would be executed, and *y* would retain its old value (X-optimistic, but not accurate silicon behavior). By adding a **default case item** that assigns *y* to X, this **case** statement becomes X-pessimistic. If *sel* should have any bit at X or Z, *y* will be assigned X, propagating the ambiguity of the *case expression*.

This coding style is supported by synthesis, and so synthesis pragmas are needed, as was the case with an X-pessimistic **if...else**. Engineers should be aware, however, that this coding style can result in synthesis performing logic minimizations that might or might not be desirable in a design. It should also be noted that this coding style can reduce the risk of unintentional latches during synthesis, but it does not guarantee latches will not be inferred.

This pessimistic coding technique does not work as well with **casex**, **casez** and **case...inside** wildcard case statements. As already shown in section C.3.3 (page 410), any don't care bits specified in *case items*, and possibly in the *case expression*, will mask out X or Z values. This masking will always make wildcard case statements at least partially X-optimistic, which can hide design problems, and not accurately represent silicon behavior.

C.4.4 Edge-sensitive X pessimism

Edge transitions can also be coded in an X-pessimistic style. As described in section C.3.9 (page 416), value changes to and from X or Z are treated as transitions, which results in X-optimistic behavior that does not accurately represent possible ambiguities in silicon behavior. The following example shows how this optimism can be removed:

```
always_ff @(posedge clk or negedge rstN)
  // synthesis translate off
  if ($isunknown(rstN) )
    q = 'x;
  // synthesis translate on
  if (!rstN) q <= 0;
  else
    // synthesis translate off
    if (rstN & clk)
      // synthesis translate on
      q <= d;
    // synthesis translate off
  else
    q = 'x;
  // synthesis translate on
```

Example 18: Edge sensitivity X-optimism

Note that the extra checking to eliminate the X-optimism is not synthesizable, and needs to be hidden from synthesis compilers. This coding style does prevent the problems of X-optimism for edge sensitivity, but the coding style is awkward and non-intuitive. Section C.9 (page 432) shows another approach to this problem that is preferred by the author.

C.4.5 Bitwise, unary reduction, and logical operators

While many SystemVerilog operators are X-optimistic (see section C.3.4, page 413), several operators are X-pessimistic. An X or Z bit in an operand will always propagate to an unknown result, even when there would be no ambiguity in the actual silicon result. The pessimistic operators are:

- Bitwise: INVERT (~), XOR (^), and XNOR (~^)
- Unary: XOR (^), and XNOR (~^)
- Logical: NOT (!)

Example 19 illustrates a 5-bit linear-feedback shift register that uses the logical exclusive-OR operator for the feedback taps. The initial value of the LFSR is seeded using a synchronous (multiplexed) active-low reset. In this example, the most-significant bit of the seed value is shown as Z (perhaps due to an interconnect error or some other design bug).

```

logic [4:0] lfsr;
logic [4:0] seed = 5'bz1010; // problem with seed!

always @ (posedge clk)
  if (!rstN
      lfsr <= seed; // seed has a bug with MSB
    begin
      lfsr    <= {lfsr[0], lfsr[4:1]}; // rotate
      lfsr[2] <= lfsr[3] ^ lfsr[0];     // xor tap
      lfsr[3] <= lfsr[4] ^ lfsr[0];     // xor tap
    end

```

Example 19: Bitwise operator with X-pessimism

Simulation cannot predict which value would be seen in silicon for the MSB of seed, but does it really matter? In actual silicon, this floating input would be seen as either a 0 or 1, and the LFSR would work without a problem, though perhaps with a different seed value than intended. A fully X-optimistic model would propagate known values through the LFSR, and hide the ambiguity that exists in silicon. The logical XOR, however, is pessimistic, and the problem with the seed value will show up as X values on the outputs of the LFSR. This X-pessimism does not accurately represent silicon behavior, and can result in X values propagating to downstream logic that can be difficult and time consuming to debug.

Example 20 shows a place where an X-pessimistic operator is desirable. The example is a verification code snippet that takes advantage of — and relies on — the X-pessimism of the exclusive-or operator:

```

logic [3:0] d = 4'b001x;

if (^d === 1'bx) // check for any unknown bit
  $display("d has one or more X or Z bits");

```

Example 20: Unary-reduction operator with X-pessimism

In this example, if any bit of d has a value of X or Z, the unary exclusive-OR operator will return an X, allowing the verification code to detect a problem with d.

C.4.6 Equality, relational, and arithmetic operators

SystemVerilog's equality, relational and arithmetic operators are X-pessimistic. An ambiguity (any bit with X or Z) in an operand will propagate as a result of X. SystemVerilog's X-pessimism for equality, relational, and arithmetic operators sometimes propagates an X where no hardware ambiguity exists. A simple example of this pessimism is a greater-than or less-than comparator.

```

logic [3:0] a = 4'b1100;
logic [3:0] b = 4'b001x;
logic       gt;

always_comb begin
  gt = (a > b); // compare a to b

```

```
end
```

Example 21: Logical operators with X-pessimism

The return from the expression `(a > b)` for the values shown in this example will be `1'bx`. In this simple code snippet, it is obvious that the value of `a` is greater than the value of `b`, regardless of the actual value of the least-significant bit of `b`. Actual silicon would not have an ambiguous result.

Arithmetic operations are also X-pessimistic, and will propagate an X if there is any ambiguity of the input values.

```
logic [3:0] a = 4'b0000;
logic [3:0] b = 4'b001z;
logic [3:0] sum;

always_comb begin
    sum = a + b;
end
```

Example 22: Arithmetic operator with X-pessimism

With arithmetic operators, all bits of the operation result are X, which can be overly pessimistic. In this example, `sum` will have a value of `4'bxxxx`. In silicon, only the least-significant bit is affected by the ambiguous bit in `b`. The silicon result would be either `4'b0010` or `4'b0011`. A more accurate representation of the silicon ambiguity would be: `4'b001x`.

Arithmetic operations are X-pessimistic, even when the result in silicon would not have any ambiguity at all.

```
logic [3:0] b = 4'b001x;
logic [4:0] product;

always_comb begin
    product = b * 0; // multiply b with 0
end
```

Example 23: Overly pessimistic arithmetic operation

In this example, `product` will have an overly pessimistic value of `4'bxxxx`, but in silicon (and in normal arithmetic) zero times anything, even an ambiguous value, would result in 0.

C.4.7 User-defined primitives

ASIC, FPGA and custom library developers can create custom primitives (UDPs) to represent library-specific components. UDPs are defined using a 4-state truth table, allowing library developers to define specific behavior for X and Z input values. In addition to specifying an output value for each combination of 4-state logic values, the truth tables can also define an output value for transitions between logic values (e.g. what happens on a posedge of clock).

Since each input can have 4 values and 12 transitions to and from those values, these truth tables can be quite large. By default, UDPs are pessimistic — any undefined input value combination that is not explicitly defined in the table will default to a result of X. Library developers often take advantage of this default to reduce the number of lines that need to be defined in the truth table. It is not uncommon for a UDP to only define output values for all possible 2-state combinations and transitions. Any X or Z values on an input, or transitions to and from X or Z, will default to propagating an X on the UDP output.

An inadvertent omission from the UDP truth table will also propagate an X value. This pessimism might be great for finding bugs in the library, but is often a source of frustration for engineers using a library from a 3rd party vendor.

C.4.8 Bit-select, part-select, array index on right-hand side of assignments

SystemVerilog defines that if the index value of a bit-select, part-select or array index is unknown (any bit is X or Z), the return from the operation will be X. If this X occurs on the right-hand side of an assignment statement, it will propagate to the left-hand side, even if there would be no ambiguity in actual silicon behavior. Consider the following:

```
logic [7:0] data = 8'b10001000;
logic [2:0] i = 3'b0x0;
logic       out;

always_comb begin
    out = data[i]; // variable bit select of data
end
```

Example 24: Ambiguous bit select with X-pessimism

The ambiguity of the value of i means that out will be X. This pessimistic rule means that problems with an index will propagate to the result of the operation. Since the values of data and i could change during simulation, this pessimism will be sure to propagate an X whenever an ambiguous value of i might occur.

This X-pessimistic rule does not accurately represent silicon behavior, however. There are times when an ambiguity in the index can still result in a known value. With the values shown in Example 24, the ambiguous value of i would either select bit 0 or 2. In either case, out would receive the deterministic value of 0 in actual silicon.

C.4.9 Shift operations

SystemVerilog has several shift operators, all of which are X-pessimistic if the shift factor is ambiguous (any bit is X or Z).

```
logic [7:0] data = 8'b10001000;
logic [2:0] i = 3'b0x0;
logic [7:0] out ;
```

```
always_comb begin
    out = data << i; // shift of data
end
```

Example 25: Ambiguous shift operation with X-pessimism

The result of this shift operation is `8'bxxxxxxxxx`. As with other pessimistic operations, this will be sure to propagate an X result whenever the exact number of times to shift is uncertain. Setting all bits of the result to X, however, can be overly pessimistic, and not represent actual silicon behavior, where only some bits of the result might be ambiguous, instead of all bits. Given the values in Example 25, data is either shifted 0 times or 2 times. The two possible results are `8'b10001000` and `8'b00100000`. If only the ambiguous bits of these two results were set to X, the X-optimistic value of `out` would be `8'bx0X0X000` instead of an overly pessimistic `8'bxxxxxxxxx`.

C.4.10 X-pessimism summary

Sections C.3 (page 405) and C.4 (page 417) have shown that, while there are times X-optimism and X-pessimism can be desirable in specific situations, neither is ideal for every situation. Subsequent sections in this paper will explore solving this problem by:

- Eliminating X values using 2-state simulation or 2-state data types.
- Breaking SystemVerilog rules in order to find a compromise between X-optimism and X-pessimism.
- Trapping X values rather than propagating Xs.

C.5 Eliminating my X by using 2-state simulation

There have been arguments made that it is better to just eliminate logic X rather than to deal with the hazards and difficulties of X-optimism and X-pessimism (see [1], [5], [6]). Some SystemVerilog simulators offer a 2-state simulation mode, typically enabled using an invocation option such as `-2state` or `+2state`.

Using 2-state simulation can offer several advantages:

- Eliminates uninitialized register and X propagation problems (the clock divider X lock-up problem shown in section C.4 (page 417) would not occur in a 2-state simulation).
- Eliminates certain potential mismatches between RTL simulation and how synthesis interprets that code, because synthesis only considers 2-state values in most RTL modeling constructs.
- RTL and gate-level simulation behaves more like actual silicon, since silicon always has a 0 or 1, and never an X.

- Reduces the simulation virtual memory footprint; Encoding 4-state values for each bit, along with strength values for net types, requires much more memory than just storing simple 2-state values.
- Improves simulation run-time performance, since 4-state encoding, decoding, and operations do not need to be performed.

On the other hand, there are several hazards to consider when only 2-state values are simulated.

First, a functional bug in the RTL or gate-level code might go undetected. Logic X is a simulator's way of indicating that it cannot accurately predict what actual silicon would do under certain conditions. When X values occur in simulation, it is an indication that there might be a design problem. Without X values, verification and detection of possible design ambiguities can be more difficult.

A second hazard of 2-state simulation values is that, since there is no X value, simulators must choose either a 0 or a 1 when situations occur where the simulator cannot accurately predict actual silicon behavior. The value that is chosen only represents one of the conditions that might occur in silicon. This means the design is verified for that one value, and leaves any other possible values untested. ***That is dangerous!*** Some simulators handle this hazard by simulating both values in parallel and merging the results of the parallel threads. This concept is discussed in more detail in section C.7 (page 430).

A third hazard is that all design registers, clock dividers, and input ports begin simulation with a value of 0 or 1 instead of X. Silicon would also power up with values of 0 or 1, but are they the same values that were simulated? Cummings and Bening [6] suggest that the most effective 2-state verification is performed by running hundreds of simulations with each register bit beginning with a random 2-state value. Cummings and Bening [6] also note that, at the time the paper was written, a preferred way for handling seeding and repeatability of randomized 2-state register initialization was patented by Hewlett-Packard, and might not be available for public use.

A fourth hazard is that verification cannot check for design problems using a logic X or Z. The following two verification snippets will not work with 2-state simulations:

```
assert (ena == 0 && data === 'Z)
else $error("Data bus failed to tri-state");

assert (^data != 'X)
else $error("Detected contention on data bus");
```

Example 26: Verification hazard with 2-state simulation

A fifth hazard of 2-state simulation to consider is the use of X assignments within RTL code. The following example illustrates a common modeling style used in combinational logic case statements:

```

case ( {sel1,sel2} )
 2'b01: result = a + b;
 2'b10: result = a - b;
 2'b11: result = a * b;
 default: result = 'X;
endcase

```

Example 27: Assigning 4-state values in 2-state simulation

Synthesis compilers treat assignment of a logic X value as a don't care assignment, meaning the design engineer does not care if silicon sees a logic 0 or a logic 1 for each bit of the assignment. In a 2-state simulation, the simulator must convert each bit of the X assignment value to either a 0 or a 1. The specific value would be determined by the simulator, since 2-state simulation is a feature of the simulator and not the language. There is a high probability that the values used in simulation and the values that occur in actual silicon will not be the same. In theory, this should not matter, since by assigning a logic X, the engineer has indicated that the actual value is a "don't care". The hazard is that, without X propagation, this theory is left unproven in 2-state simulation.

C.6 Eliminating some of my X with 2-state data types

The original Verilog language only provided 4-state data types. The only way to achieve the benefits of 2-state simulation was to use proprietary options provided by simulators, as discussed in the previous section. These proprietary 2-state algorithms do not work the same way with each simulator. 2-state simulation modes also make it difficult to mix 2-state simulation in one part of a design and 4-state simulation in other parts of the design.

SystemVerilog improves on the original Verilog language by providing a standard way to handle 2-state simulations. Several SystemVerilog variable types only store 2-state values: **bit**, **byte**, **shortint**, **int**, and **longint**. SystemVerilog-2012 adds the ability to have user-defined 2-state net types, as well.

Using these 2-state data types has two important advantages of simulator-specific 2-state simulation modes:

- All simulators follow the same semantic rules for what value to use in ambiguous conditions (such as power-up).
- It is easy to mix 2-state and 4-state within a design, which allows engineers to select the appropriate type for each design or verification block.

The uninitialized value of 2-state variables is 0. This can help prevent blocks of design logic from getting stuck in a logic X state at the start of simulation, as discussed in section C.2.2 (page 402) earlier in this paper. The clock-divider circuit that was described at the beginning of section C.4 (page 417) will work fine if the flip-flop storage is modeled as a 2-state type.

Having all variables begin with a logic 0 does not accurately mimic silicon behavior, however, where each bit of each register can power-up to either 0 or 1. When all variables start with a value of 0, only one extreme and unlikely hardware condition is verified. Bening [5] suggests that simulation should begin with random values for all bits in all registers, and that hundreds of simulations with different seed values should be run, in order to ensure that silicon will function correctly at power-up under many different conditions.

The ability to declare nets and variables that use either 2-state or 4-state value sets makes it possible to freely mix 2-state and 4-state within a simulation. Engineers can choose the benefits of 2-state performance in appropriate places within a design or testbench, and choose the benefits of 4-state simulation where greater accuracy is required.

SystemVerilog defines a standard rule for mapping 4-state values to 2-state values. The rule is simple. When a 4-state value is assigned to a 2-state net or variable, any bits that are X or Z are converted to 0. This simplistic rule eliminates X values, but does not accurately mimic silicon behavior where each ambiguous bit might be either a 0 or a 1, rather than always 0.

Your X just might be your best friend!

Section C.5 (page 426) of this paper discussed several hazards with using 2-state simulation modes. All of those hazards also apply to using 2-state data types. X is the simulator's way of saying there is some sort of ambiguity in a design. As much as all engineers hate to see an X show up during simulation, an X indicates there is a potential design problem that needs to be investigated. The following example illustrates how 2-state types can hide a serious design error.

```
module program_counter (          // 2-state types
    input  bit      clock, resetN, loadN,
    input  bit [15:0] new_count,
    output bit [15:0] count
);
    always_ff @ (posedge clock or negedge resetN)
        if (!resetN)      count <= 0;
        else if (!loadN)  count <= new_count;
        else              count <= count + 1;
endmodule: program_counter

module cpu (                      // 4-state types
    wire      m_clk, m_rstN,
    wire [15:0] next_addr
);
    ...
    program_counter pc (.clock(m_clk),
                        .resetN(m_rstN),
                        .loadN(/* not used */),
```

```

    .new_count /* not used */),
    .count(next_addr) );
...
endmodule: cpu

```

**Example 28: Program counter with unused inputs,
2-state data types**

The program counter in this example is loadable, using an active-low `loadN` control. The CPU model has an instance of the program counter, but does not use the loadable `new_count` input or `loadN` control. Since they are not used, these inputs are left unconnected, which is probably an inadvertent design bug! With 2-state data types, however, the unconnected inputs will have a constant value of 0, which means the statement

```
if (!loadN) count <= new_count;
```

will always evaluate as true, and the program counter will be locked in the load state, rather than incrementing on each clock edge.

In this small example, this bug would be easy to find. Imagine, though, a similar bug in a huge ASIC or FPGA design. Simple mistakes that are hidden by not having a logic X show up in simulation can become very difficult to find. Worse, the symptom of having a logic 0, instead of a logic X, might make a design bug appear to be working at the RTL level, and not show up until gate-level simulations are run. (And what if your team doesn't do gate-level simulations?)

After having a 2-state data type hide a design error or cause bizarre simulation results in a large, complex design, you too might feel, as the author does, that "*I'm still in love with my X!*"

C.7 Breaking the rules—simulator-specific X-propagation options

The previous sections in this paper have shown that SystemVerilog can sometimes be overly optimistic, and at other times overly pessimistic in how logic X is propagated, and that 2-state simulations and data types can hide design problems by completely eliminating Xs. Can a balance between these two extremes be found by breaking the IEEE 1800 SystemVerilog standard X propagation rules and simulating with a different algorithm?

Some simulators provide proprietary invocation options to begin simulation with random variable values, instead of with X values. Using simulator-specific options can accomplish Bening's recommended approach of randomly initializing all registers using a different seed [5]. Since these options are not part of the SystemVerilog language, however, the capability is not available on every simulator and does not work the same way on simulators that have the feature.

Some SystemVerilog simulators offer a way to reduce X-optimism in RTL simulation by using a more pessimistic, non-standard algorithm. For example, the Synopsys

VCS “**-xprop**” [11] simulation option causes VCS to use simulator-specific X propagation rules for **if...else** and **case** decision statements and **posedge** or **negedge** edge sensitivity. This non-standard approach tries to find a balance between X-optimism and X-pessimism. See Evans, Yam and Forward [12] and Greene, Salz and Booth [13] for more information on—and experience with—using proprietary X-propagation rules to change SystemVerilog’s X-optimism and X-pessimism behavior.

One concern with proprietary X propagation rules is that their purpose is to ensure that design bugs will propagate downstream from the cause of the problem, so that the bug will be detected instead of hidden. This then requires tracing the cause of an X back through many lines of code, branching statements, and clock cycles to find the original cause of the problem. Though most simulators provide powerful debug tools for tracing back X values, the process can still be tedious and time consuming.

Another concern is the risk of false failures, by making simulation more X-pessimistic. Finding a balance between X-optimism and being overly pessimistic can be good, but, like the **? :** conditional operator, will not always perfectly match silicon behavior (see section C.4.2, page 419). There might still be times when this balance of X-optimism and X-pessimism can result in false failures. At best, these false failures can consume significant project man-hours to determine that there is no actual design problem. Worse — and very possible — these false failures could potentially cause problems with simulation locking up in X states, as described in section C.2.2 (page 402).

C.8 Changing the rules — A SystemVerilog enhancement wish list

There have been proposals to modify SystemVerilog’s X-optimism and X-pessimism rules in some future version of SystemVerilog. If readers of this paper feel these enhancements would be important for their projects, they should pressure their EDA vendors to push for these enhancements in the next SystemVerilog standard.

One of the X-optimism issues presented in this paper is that wildcard “don’t care” bits in **casex**, **casez** and **case...inside** statements mask out all 4 possible 4-state values, causing unknown bits in *case expressions* to be treated as don’t care values.

Turpin [1] proposed adding the ability to specify 2-state wildcard “don’t care” values using an asterisk (instead of X, Z or ?), as follows:

```
always_comb begin
    case (sel) inside
        3'b1**: y = a; // matches 100, 101, 110, 111
        3'b00*: y = b; // matches 000, 001
        3'b01*: y = c; // matches 010, 011
        default: y = 'x;
    endcase
end
```

Example 29: Proposed case...inside with 2-state don’t cares

In normal SystemVerilog X-optimistic semantics, if either of the lower 2 bits of `sel` were X or Z, those bits could potentially be masked out by the 4-state don't cares in the *case items*, causing `y` to be assigned a known value instead of propagating an X. The proposed 2-state don't care bits (represented by an asterisk) would not mask out X or Z values, and result in the `default` branch propagating an X whenever there is a problem with the *case expression*.

Cummings [15] proposed adding new procedural blocks that are X-pessimistic instead of X-optimistic. The proposed keywords are `initialx`, `alwaysx`, `always_combx`, `always_latchx` and `always_ffx`. Cummings proposes that any time a decision control expression or loop control expression evaluates to X or Z, simulation should do three things:

- Assign X values to all variables assigned within the scope of the decision statement or loop.
- Ignore all system tasks and functions within the scope of the decision statement or loop.
- Optionally report a warning or error message that the tested expression evaluated to an X or Z.

An example usage is:

```
always_ffx @ (posedge clk or negedge rstN)
  if (!rstN) q <= 0;
  else       q <= d;
```

Example 30: Proposed procedural block with X-pessimism

Under SystemVerilog's normal X-optimistic rules, if `rstN` evaluated as X or Z, then `q` would be set to the value of `d`, hiding the ambiguous reset problem. Using the proposed X-pessimistic rules for `always_ffx`, if `rstN` evaluated as X or Z, then `q` would be set to X, propagating the ambiguous reset problem.

The author of this paper does not fully concur with the semantics Cummings has proposed. The author likes the concept of special RTL procedures with more accurate X-propagation behavior, but feels the proposed semantics are overly pessimistic, and could result in causing false X values or X-lockup problems — the same issues noted earlier in this paper regarding excessive X pessimism. The author would prefer to see semantics that are similar to the T-merge algorithm used by the proprietary VCS - xprop simulation option.

C.9 Detecting and stopping my X at the door

Let's face it, when an X shows up, trouble is sure to follow! Rather than having X problems propagate through countless lines of code, decision branches, and clock cycles, it would be much better to detect an X the moment it occurs. ***Detecting when an X first appears solves the problems of both X-optimism and X-pessimism!***

X-optimism results in X values propagating as 0 or 1 values to downstream logic, potentially hiding design problems. X-pessimism results in all X values propagating to downstream logic, potentially causing simulation problems such as X-lockup, that would not exist in actual silicon. In either case, design problems might not be detected until far down stream in both logic and clock cycles from the original cause of the bug. Engineers must then spend a great deal of valuable engineering time debugging the cause of the problem.

SystemVerilog immediate assertions can be used to detect X values at the point the value occurs, rather than detecting the X value after it has (maybe) propagated downstream to other logic in the design. The way to do this is to use assertions to monitor all input ports of a module, as well as selection control values on conditional operations.

An additional advantage of using assertions to monitor for X values is that assertions can be disabled when X values are expected, such as before and during reset or during a low power shut down mode. Disabling and re-enabling of assertions can be done at any time during simulation, and can be on a global scale, on specific design blocks, or on specific assertions.

The syntax for an immediate assertion is:

```
assert ( expression ) [ pass_statement ]
[ else fail_statement ] ;
```

An immediate **assert** statement is similar to an **if** statement, except that both the *pass_statement* and the **else** clause are optional.

The pass or fail statements can be any procedural statements, such as printing messages or incrementing counters. Typically, the pass statement is not used, and the fail statement is used to indicate that an X value has been detected, as shown in the following code example for a simple combinational **if...else** statement:

```
always_comb begin
    assert (!$isunknown(sel))
    else $error("%m, sel = X");

    if (sel) y = a;
    else      y = b;
end
```

Example 31: if...else with X-trap assertion

This is the same **if...else** example that has presented in previous sections, but with an added assertion to validate the value of *sel* each time it is evaluated.

Without the assertion, this simple **if...else** decision has several potential X hazards, as was discussed in sections C.3.1 (page 406) and C.4.1 (page 418). Adding an immediate assertion to verify **if** conditions is simple to do, and avoids all of these hazards. A problem with the **if** condition is detected when and where the problem occurs, rather than hoping that propagating an X will make it visible sometime, some-

where. Assert statements are ignored by synthesis, so no code has to be hidden from synthesis compilers.

The author recommends that **if** statements that are conditioned on a module input port have an immediate assertion to validate the **if** condition. A text-substitution macro could be defined to simplify using this assertion in many places.

```

`define assert_condition (cond) \
  assert (^cond === 1'bx) \
  else $error("%m, ifcond = X")

always_comb begin
  `assert_condition(sel)
  if (sel) y = a;
  else      y = b;
end

always_comb begin
  `assert_condition({a,b,c,d})
  `assert_condition(sel)
  case (sel)
    2'b00 : out = a;
    2'b01 : out = b;
    2'b01 : out = c;
    2'b01 : out = d;
  endcase
end

```

Example 32: Using an X-trap assertion macro

SystemVerilog assertions are ignored by synthesis compilers, and therefore can be placed directly in RTL code without having to hide them from synthesis using conditional compilation or pragmas. It is also possible to place the assertions in a separate file and bind them to the design module using SystemVerilog's binding mechanism.

C.10 Minimizing problems with my X

This section presents a few coding guidelines that help to appropriately use and benefit from SystemVerilog's X-optimism and X-pessimism, and minimize the potential hazards associated with hiding or propagating an X.

C.10.1 2-state versus 4-state guidelines

Your X can be your best friend. X values indicate that there is some sort of ambiguity in the design. Eliminating X values using 2-state data types does not eliminate the design ambiguity. Sutherland HDL recommends using 4-state data types in all places, with two exceptions:

- The iterator variable in **for**-loops is declared as an **int** 2-state variable.

- Verification stimulus variables that will (or might) have randomly generated values are declared as 2-state types.

This coding guideline uses 2-state types only for variables that will never be built in silicon, and therefore do not need to reflect an ambiguous condition that might exist in silicon.

There is one other place where 2-state types might be appropriate, which is the storage of large memory arrays. Using 2-state types for large RAM models can substantially reduce the virtual memory needed to simulate the memory. This savings comes at a risk, however. Should the design fail to correctly write or read from a memory location, there will be no X values to indicate there was a problem. To help minimize that risk, it is simple to model RAM storage, so that it can be configured to simulate as either 2-state storage (using the `bit` type) or 4-state storage (using the `logic` type).

C.10.2 Register initialization guidelines

Section C.2.2 (page 402) discussed the problems associated with design variables, especially those used to model hardware registers, beginning simulation with X values. Section C.5 (page 426) discussed using proprietary simulation options to initialize register variables with random values. If that feature is available, it should be used!

Another way to randomly initialize registers is using the UVM Register Abstraction Layer (RAL). UVM is a standard, and is well supported in major SystemVerilog simulators. A UVM testbench and RAL are not trivial to set up, but can provide a consistent way to randomly initialize registers. The advantage of using UVM to initialize registers is that it will work with all major simulators.

C.10.3 X-assignment guidelines

Using X assignments to make `if...else` and `case` statements more pessimistic should not be used. They add overhead to simulation, and can simulate differently than the logic that is generated from synthesis. The pessimistic X propagation can lead to false failures that can take time to debug and determine that there not be a problem in actual silicon. In lieu of using pessimistic coding styles to propagate X values, problems should be trapped at the select condition, as shown in Section C.9 (page 432), and discussed in the following guideline.

C.10.4 Trapping X guidelines

All RTL models intended for synthesis should have SystemVerilog assertions detect X values on `if...else` and `case` select conditions. Other critical signals can also have X-detect assertions on them. Design engineers should be responsible for adding these assertions. Section C.9 (page 432) showed how easy it is to add X-detecting assertions.

C.11 Conclusions

This paper has discussed the benefits and hazards of X values in simulation. Sometimes SystemVerilog is optimistic about how X values affect design functionality, and sometimes SystemVerilog is pessimistic.

X-optimism has been defined in this paper as any time simulation converts an X value on the input to an operation or logic gate into a 0 or 1 on the output. Some key points that have been discussed include:

- X-optimism can accurately represent real silicon behavior when an ambiguous condition occurs. For example, if one input to an AND gate is uncertain, but the other input is 0, the output of the gate will be 0. SystemVerilog X-optimistic AND operator and AND primitive behave the same way.
- X-optimism is essential for some simulation conditions, such as the synchronous reset circuit shown in Section C.3 (page 405).
- SystemVerilog can be overly optimistic, meaning an X propagates as a 0 or 1 in simulation when actual silicon is still ambiguous. Over optimism can lead to only one of the possible silicon values being verified.
- In all circumstances, X-optimism has the risk of hiding design bugs. A ambiguous condition that causes an X deep in the design might not propagate as an X to a point in the design that is being observed by verification. The value that does propagate might appear to be a good value.

X-pessimism is defined in this paper as any time simulation passes an X on an input through to the output. X-pessimism can be desirable or undesirable.

- X-pessimism will not hide design bugs the way X-optimism might. An ambiguous condition deep within a design will propagate as an X value to points that verification is observing.
- X-pessimism can lead to false failures, where actual silicon will function correctly, such as if one input to an AND gate is an X, but the other input is 0. A false X might need to be traced back through many levels of logic and clock cycles before determining that there is not an actual problem.
- X-pessimism can lead to simulation locking up with X values, where actual simulation will function correctly, even if the logic values in silicon are ambiguous. The clock divider shown in section C.4 (page 417) is an example of this.

It might be tempting to use 2-state data types or 2-state simulation modes to eliminate the hazards of an X. Although there are some advantages to 2-state simulation, those advantages do not outweigh the benefits of 4-state simulation. 2-state simulation will hide all design ambiguities, and often not simulate with the same values that actual silicon would have. 2-state data types should only be used for generating random stimulus values. Design code should use 4-state types.

The best way to handle X problems is to detect the X as close to its original source as possible. This paper has shown how SystemVerilog assertions can be used to easily detect and isolate design bugs that result in an X. With early detection, it is not necessary to rely on X propagation to detect design problems.

All engineers should be in love with their X! X values indicate that there might be some ambiguity in an actual silicon implementation of intended functionality.

C.11.1 About the author

Stuart Sutherland is a well-known Verilog and SystemVerilog expert, with more than 24 years of experience using these languages for design and verification. His company, Sutherland HDL, specializes in training engineers to become true wizards using SystemVerilog. Stuart is active in the IEEE SystemVerilog standards process, and has been a technical editor for every version of the IEEE Verilog and SystemVerilog Language Reference Manuals since the IEEE standards work began in 1993. Prior to founding Sutherland HDL, Mr. Sutherland worked as an engineer on high-speed graphics systems used in military flight simulators. In 1988, he became a corporate applications engineer for Gateway Design Automation, the founding company of Verilog, and has been deeply involved in the use of Verilog and SystemVerilog ever since. Mr. Sutherland has authored several books and conference papers on Verilog and SystemVerilog. He holds a Bachelors Degree in Computer Science with an emphasis in Electronic Engineering Technology and a Masters Degree in Education with an emphasis on eLearning. You can contact Mr. Sutherland at stuart@sutherland-hdl.com.

C.12 Acknowledgments

The author appreciates the contributions Don Mills and Shalom Bresticker have made to this paper. Don provided several of the examples and coding recommendations in this paper, and provided valuable suggestions regarding the paper content. Shalom made an in-depth technical review of the paper draft and provided detailed comments on how to improve the content of the paper.

The author also expresses gratitude to his one-and-only wife of more than 30 years (she will never be an “X”), who, despite the title of the paper, painstakingly reviewed the paper for grammar, punctuation, and sentence structure.

C.13 References

- [1] Turpin, “The dangers of living with an X,” Synopsys Users Group Conference (SNUG) Boston, 2003.
- [2] Mills, “Being assertive with your X (SystemVerilog assertions for dummies),” Synopsys Users Group Conference (SNUG) San Jose, 2004.
- [3] “P1800-2012/D6 Draft Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language (re-ballot draft)”, IEEE, Piscataway, New Jersey. Copyright 2012. ISBN: (not yet assigned).
- [4] Merriam-Webster online dictionary, <http://www.merriam-webster.com/>, accessed 11/20/2012.
- [5] Bening, “A two-state methodology for RTL logic simulation,” Design Automation Conference (DAC) 1999.
- [6] Cummings and Bening, “SystemVerilog 2-state simulation performance and verification advantages,” Synopsys Users Group Conference (SNUG) Boston, 2004.
- [7] Piper and Vimjam, “X-propagation woes: masking bugs at RTL and unnecessary debug at the netlist,” Design and Verification Conference (DVcon) 2012.
- [8] Weber and Pecor, “All My X values Come From Texas...Not!,” Synopsys Users Group Conference (SNUG) Boston, 2004.
- [9] Turpin, “Solving Verilog X-issues by sequentially comparing a design with itself,” Synopsys Users Group Conference (SNUG) Boston, 2005.
- [10] Chou, Chang and Kuo, “Handling don’t-care conditions in high-level synthesis and application for reducing initialized registers,” Design Automation Conference (DAC) 2009.
- [11] Greene, “Getting X Propagation Under Control”, a tutorial presented by Synopsys, Synopsys Users Group Conference (SNUG) San Jose, 2012.
- [12] Evans, Yam and Forward, “X-Propagation: An Alternative to Gate Level Simulation”, Synopsys Users Group Conference (SNUG) San Jose, 2012.
- [13] Greene, Salz and Booth, “X-Optimism Elimination during RTL Verification”, Synopsys Users Group Conference (SNUG) San Jose, 2012.
- [14] Browy and K. Chang, “SimXACT delivers precise gate-level simulation accuracy when unknowns exist,” White paper, http://www.avery-design.com/files/docs/SimXACT_WP.pdf, accessed 11/12/2012.
- [15] Cummings, “SystemVerilog 2012 new proposals for design engineers,” presentation at SystemVerilog Standard Working Group meeting, 2010, <http://www.eda.org/sv-ieee1800/Meetings/2010/February/Presentations/Cliff%20Cummings%20Presentation.pdf>, accessed 11/12/2012.
- [16] Mills, “Yet another latch and gotchas paper” Synopsys Users Group Conference (SNUG) San Jose, 2012.

Appendix D

Additional Resources

This appendix lists some additional resources that are closely related to the topics presented in this book, and might be of particular interest to RTL design engineers.

Books. Two books that are important companions to this book are:

- IEEE Std 1800-2012, *SystemVerilog Language Reference Manual (LRM)*. Officially titled, “*IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*”. Copyright 2013 by The Institute of Electrical and Electronics Engineers, Inc. ISBN 978-0-7381-8110-3 (PDF) and 978-0-7381-8111-0 (print).

This is the official standard for the syntax and semantics of the SystemVerilog language. The book is written primarily for companies that implement Electronic Design Automation (EDA) tools, such as simulators and synthesis compilers. Note that standard does not distinguish between what aspects of the language are for design and what aspects are for verification. Available to download at: <https://standards.ieee.org/getieee/1800/download/1800-2012.pdf>.

- “*SystemVerilog for Verification, Third Edition*”, by Chris Spear and Greg Tumbush. Copyright 2012, Springer, New York, New York. ISBN 978-1-4614-0715-7.

Presents the numerous verification constructs in SystemVerilog, which are not covered in this book. For more information, refer to the publisher’s web site: <http://www.springer.com/engineering/circuits+%26+systems/book/978-1-4614-0714-0>.

Some additional books that might be of interest include:

- “*Constraining Designs for Synthesis and Timing Analysis*”, by Sridhar Gangadharan and Sanjay Churiwala. Copyright 2013, Springer, New York, New York. ISBN 978-1-4614-3268-5.

A guide to specifying timing constraints for synthesis, static timing analysis and placement and routing using the industry format of Synopsys Design Constraints (SDC). Chapters on synthesis constraints, multi-clock boundaries and clock-domain crossing are particularly germane to the RTL synthesis coding styles presented in this book. For more information, refer to the publisher’s web site: <http://www.springer.com/engineering/circuits+%26+systems/book/978-1-4614-3268-5>.

- “*SystemVerilog Assertions Handbook, 4th edition*”, by Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, Lisa Piper. Copyright 2015, VhdlCohen, Palos Verdes Peninsula, California. ISBN: 978-1518681448.

This book presents Assertion-Based Verification techniques using the SystemVerilog Assertions portion of the SystemVerilog standard. For more information, refer to the publisher’s web site: <https://www.createspace.com/5810350>.

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 manual covers the synthesizable subset of the obsolete Verilog-2001 language, before the introduction of SystemVerilog. ***The manual is grossly out-of-date***, and most Verilog-2001 synthesis compilers did not adhere to the standard. Its usefulness as an additional reference is limited to just comparing how much RTL modeling capability SystemVerilog added to the early Verilog language. The difference between traditional Verilog and SystemVerilog is substantial!

Available at <https://standards.ieee.org/findstds/standard/1364.1-2002.html>

Conference papers. Following is a list of a few conference papers that explore some of the topics discussed in this book, and which might be of particular interest to RTL design engineers.

NOTE

Examples shown in conference papers might not use up-to-date best practice coding styles. Some of the papers cited in this appendix were written before SystemVerilog extended the original Verilog language. The capabilities of both the language and synthesis compilers have evolved significantly since some of these older papers were written. The engineering principles discussed in these papers are still relevant and useful, but some examples and recommended coding styles might be obsolete. It is left to the reader to rewrite these examples using the best-practice SystemVerilog coding styles presented in this book.

1. “*Who Put Assertions In My RTL Code? And Why? How RTL Design Engineers Can Benefit from the Use of SystemVerilog Assertions*”, by Stuart Sutherland. Presented at the 2013 Silicon Valley Synopsys Users Group Conference (SNUG). Available for download at sutherland-hdl.com.
2. “*Synchronization and Metastability*”, by Steve Golson. Presented at the 2014 Silicon Valley Synopsys Users Group Conference (SNUG). Available at trilobyte.com.

3. “*Yet Another Latch and Gotchas Paper*”, by Don Mills. Presented at the 2012 Silicon Valley Synopsys Users Group Conference (SNUG). Available at lcdm-eng.com.
4. “*RTL Coding Styles That Yield Simulation and Synthesis Mismatches*”, by Don Mills and Clifford Cummings. Presented at the 2001 Europe Synopsys Users Group (SNUG). Available at lcdm-eng.com or sunburst-design.com.
5. “*Asynchronous & Synchronous Reset Design Techniques — Part Deux*”, by Clifford Cummings, Don Mills and Steve Golson. Presented at the 2003 Boston Synopsys Users Group Conference (SNUG). Available at lcdm-eng.com, sunburst-design.com, or trilobyte.com.
6. “*‘full_case parallel_case’, the Evil Twins of Verilog Synthesis*”, by Clifford Cummings. Presented at the 1999 Boston Synopsys Users Group Conference (SNUG). Available at sunburst-design.com.
7. “*Language Wars in the 21st Century: Verilog versus VHDL – Revisited*”, by Steve Golson and Leah Clark. Presented at the 2016 Silicon Valley Synopsys Users Group Conference (SNUG). Available at trilobyte.com.

In addition to these papers, Stuart Sutherland, the author of this book, has authored or co-authored many papers on SystemVerilog topics (and on the original Verilog language). These papers and presentation slides are available at sutherland-hdl.com.

Index

Symbols

- `^` bitwise XOR operator *153*
- `^` reduction XOR operator *158*
- `^=` bitwise XOR assignment operator *196*
- `^~` bitwise XNOR operator *153*
- `^~` reduction XNOR operator *158*
- `_` in literal value *65*
- `--` decrement operator *189*
- `-:` variable part select *72*
- `-=` subtract assignment operator *196*
- `->` logical implication operator *164*
- `-f` invocation option *110*
- `;` no-op statement *241–242*
- `::` scope resolution operator *104, 107, 247*
- `::*` wildcard package import *104*
- `!` logical negate operator *160, 161*
- `? :` conditional operator *150–152, 417*
- `? as a Z value` *65*
- `.*` port connection *58–59, 81*
- `.sv` file name extension *48*
- `.v` file name extension *48*
- `'` compiler directives *52*
- `'begin_keywords` *46–49, 52, 391–396*
- `'default_nettpe` *79, 81–82*
- `'default_nettpe none` *82*
- `'define` *22, 52, 111, 113, 124*
- `'else` *52*
- `'elsif` *52, 103*
- `'end_keywords` *46–49, 52, 391–396*
- `'endif` *52, 103, 111, 293, 352*
- `'ifdef` *52, 103, 111, 113, 352, 381, 417*
- `'ifndef` *52, 111, 293, 352*
- `'include` *52, 102, 110*
- `'timescale` *22, 24, 52*
- `' apostrophe` *63*
- `'()` cast operator *198*
- `'{ }` list of values *91, 125, 149*
- `'0` vector fill literal value *65*
- `'1` vector fill literal value *65*
- `'x` vector fill literal value *65*
- `'z` vector fill literal value *65*
- `(* attribute start` *42*
- `[:]` part select *71*
- `[:]` unpacked array declaration *90*
- `[:]` vector declaration *71*
- `[]` bit select *71*
- `{ }` concatenate operator *92, 146–150*
- `{ }` enumerated list *114*
- `{ }` structure definition *124*
- `{ }` union definition *131*
- `{ { } }` replicate operator *146–150*
- `@` sensitivity list *212*
- `@ time control` *212, 257, 275, 324, 325*
- `@* inferred sensitivity list` *259, 324*
- `* wildcard package import` *104*
- `*) attribute end` *42*
- `/* block comment end` *41*
- `*= multiply assignment operator` *196*
- `/* block comment start` *41*
- `// one-line comments` *40*
- `/= divide assignment operator` *196*
- `\ escape character` *49*
- `&` bitwise AND operator *144, 153*
- `& reduction AND operator` *158*
- `&&` logical AND operator *144, 160*
- `&= bitwise AND assignment operator` *196*
- `# delay` *26, 257, 275, 297, 324, 325*
- `#() parameter list` *53, 94–96*
- `#() parameter redefinition` *55, 97–98*
- `#1 unit delay` *298*
- `%= modulus assignment operator` *196*
- `+:` variable part select *72*
- `++` increment operator *74*
- `+= add assignment operator` *196*
- `+define+` *52*
- `<->` logical equivalence operator *164*
- `<<<=` arithmetic left shift assignment op *196*
- `<<=` bitwise left shift assignment op *196*
- `<=` nonblocking assignment *28, 278–281*
- `= blocking assignment` *28, 196, 261–262*
- `>>= bitwise right shift assignment op` *196*
- `>>>= arithmetic right shift assignment` *196*
- `|` bitwise OR operator *153*
- `| reduction OR operator` *158*
- `= bitwise OR assignment operator` *196*
- `|| logical OR operator` *160*
- `~` bitwise invert operator *153, 161*
- `~^` bitwise XNOR operator *153*
- `~^` reduction XNOR operator *158*
- `& reduction NAND operator` *158*
- `~| reduction NOR operator` *158*
- `$ system task/function` *51*
- `$cast` *120*
- `$dimensions` *238*

\$display 51
\$error 51
\$fatal 51
\$high 238
\$increment 238
\$increment system function 238
\$info 51
\$left 238
\$readmemb 320
\$readmemh 320
\$right 238
\$signed 207
\$size 238
\$unit name space 22, 50, 52, 112–114
\$unsigned 207
\$warning 51

Numerics

0 as a value 61
 1 as a value 61
 1-dimensional array 90
 1364 Verilog standard 1, 438
 1364-1995 3, 46
 1364-2001 3, 46
 1364-2005 5, 46
 1800 SystemVerilog standard xxvii, xxxi, 40
 1800-2005 5, 46
 1800-2009 6, 46
 1800-2012 6, 46
 1800-2017 6, 46
 2-state
 data 66–67
 operations 142
 simulation 424
 variables 68, 426
 uninitialized value 74
 4-state
 data 66–67
 nets 77
 operations 142
 values 398
 variables 68
 uninitialized value 74

A

abstraction levels 6
 algorithmic 11
 behavioral 11
 bus-functional 11
 gate-level 7–9
 RTL 10–11
 switch-level 9
 transaction-level 11
 Accellera 4

acknowledgements xxxi
 Active event region 27, 278–279
 actual arguments, tasks and functions 245
 algorithmic models 11
 always procedures
 always (general purpose) 10, 19, 39, 74,
 212, 213, 214, 256–260, 324–325
 always_comb 10, 19, 39, 74, 212, 214, 256–
 257, 260–261, 333
 always_ff 10, 19, 39, 74, 212, 213
 always_latch 11, 19, 39, 74, 212, 214, 325–
 326
 types of procedures 10, 19, 39, 211
 AND
 bitwise operator 154
 logical operator 160
 reduction operator 158
 anonymous
 enumerated types 116
 structures 126, 127
 unions 131
 ANSI-style port list 85
 apostrophe 63
 arguments, tasks and functions 245
 arrays
 '{ } list of values 91, 149
 assignments 91–93
 copying 91
 default values 92
 elements 89
 in structures or unions 136
 memory 318
 of structures or unions 136
 packed 71–74
 passing through ports 93
 traversing with foreach loops 236
 unpacked 89, 89–93
 X-optimism 412
 X-pessimism 423
 arrival time 35
 ASIC 2, 53, 78
 design flow 12–15
 assign, continuous 10, 74, 83, 252–256
 assignments
 blocking 28–30, 196, 261–262, 282, 283,
 312
 continuous 10, 74, 83, 252–256
 nonblocking 28–30, 278–281, 297–298, 312
 operators 74
 to arrays 91–93
 to enumerated types 118–120
 to nets 83–84
 to structures 125
 to unions 132
 to variables 74–76

asynchronous reset flip-flop 288–289, 404
asynchronous set/reset flip-flop 291–292
attributes 42
automatic 375
automatic functions 111, 243–244

B

backquote 52
backslash 49
backtick 52
Backus–Naur Form 40
backward compatibility 46–49
base type 114
begin-end statement group 19, 40, 214–216, 280
behavioral models 11
bidirectional 53, 84
big endian 71
binary count state encoding 302
binary literal values 63
BIST, see Built-in Self Test
bit data type 68, 69, 70, 94, 426
bit select 71–72, 92, 423
bitwise operators 153–157
block comments 41
block name space 50
blocking assignment 28–30, 196, 261–262, 312
 for temporary variables 283
 incorrect usage 282
BNF, see Backus–Naur Form
Boolean equations 254
break statement 239
Bresticker, Shalom *xxxii*
Built-in Self Test 349
bus-functional models 11
byte data type 68, 69, 426

C

Cadence Design Systems 3
case sensitivity 49
case statements 222–227, 265
 case expression 223
 case items 223, 225
 case vs. case...inside 224
 case...inside 223, 224, 408–411
 casex 223, 224, 408–411
 casez 223, 224, 408–411
 default case item 223, 335–337, 407, 419
 don't care 224
 full_case synthesis pragma 350–351
 pre-case assignment 338–339
 reverse case 265, 313–317
 wildcard 224

X default assignment 345–349, 419
X-optimism 407
X-pessimism 419
casting
 enumerated types 120, 200, 346
 operator 198–208
 signedness 206–208
 size 202–205
 type 200–201
CDC, see Clock Domain Crossing
chaining packages 109–110
chandle data type 70
checker name space 50
chip-enable flip-flop 221, 290
Clark, Leah *xxxii*, 439
class
 handle data type 70
 name space 50
 parameterized 247
 virtual 247
CLB 15
Clock Domain Crossing 295–296
clock frequency 35
clock period 35
clock tree synthesis 14
clock-to-Q delay 297
CMOS 78
code examples *xxix*
code snippets *xxix*
coding style, ASICs vs. FPGA 16
combinational logic 213, 251–272, 298
 definition 251
comments 40–43
companion book on verification *xxvii*, *xxxii*, 6, 20, 437
compilation
 and elaboration 21
 conditional 52, 103, 111, 113, 293, 352, 381, 417
 order 110–111
compiler directives 52
component name space 50
concatenate operator 92, 146–150
conditional compilation 52, 103, 111, 113, 293, 352, 381, 417
conditional operator ?: 150–152, 417
connection size mismatch 83–84
constant functions 95
constants
 const 99, 103
 localparam 94
 parameter 93–99
context-determined operands 144
continue statement 239
continuous assignments 10, 74, 83, 252–256

explicit declaration 254
implicit declaration 254
convention, naming 50–51, 102
copying arrays 91
copying structures 127
Cummings, Clifford xxxi, 439
curly braces { }
enumerated list 114
structures 124
unions 131
cycle-stealing 326

D

D-type latches 323
data types
bit 68, 69, 426
byte 68, 69, 426
chandle 70
class handle 70
event 70
int 68, 69, 426
logic 68
longint 68, 69, 426
nets 66–67, 76–84
pull0 77
pull1 77
real 69
realtime 70
reg 68–69
shortint 68, 69, 426
shortreal 69
string 70
supply0 77
supply1 77
time 70
tri 77, 152, 252
triand 77
trior 77
trireg 77
user-defined 101–102
uwire 77
var 67, 70, 252
variables 66–76
virtual interface 70
wand 77
wire 77–80, 252
wor 77
data-dependent loops 230
data-enable flip-flop, see chip-en
dataflow 254, 262, 298
Davidmann, Simon 6
DC-Ultra xxx
decimal literal values 63
decision modifiers 227, 266–277

340–345, 351
decision statements **216–228**
 fully implemented **330**
 incomplete branches **328**
 incomplete decisions **327**
 logic reduction **330**
default
 array values **92**
 case item **223, 407, 419**
 for latch avoidance **335–337**
 structure values **126**
definitions name space **49**
defparam **98**
delays
 # **26, 257, 275, 297, 324, 325**
 clock-to-Q **297**
 intra-assignment **297, 326**
 primitives **9**
 propagation **25, 32, 35, 235, 243, 252, 324**
 time precision **24**
 time units **24**
 unit **275, 297, 326**
 zero-delay **54, 192, 230, 243, 252, 257, 260,**
 262, 275, 324, 326, 356, 375
delta **297**
delta cycle **27, 278**
Department of Defense **3**
design flow
 ASIC **12–15**
 FPGA **15–16**
Design for Test **323**
Design Rule Checker **14**
DFT, see Design for Test
digital simulation **17–30**
disable statements **240–241**
do-while loop statements **235–236**
DoD, see Department of Defense
dot-name port connections **57–58, 81**
dot-star port connections **58–59, 81**

E

ECO, see Engineering Change Order
EDA, see Electronic Design Automation
EDIF **32**
elaboration **21**
Electronic Design Automation **2, 4**
element, of an array **89**
end-of-file **43**
endinterface **356, 361**
endmodule **39, 53**
endpackage **103**
Engineering Change Order **36**
enum **114**
enumerated types **114–124**

- anonymous **116**
- assignment rules **118–120**
- casting **200**
- explicit **115**
- implicit **114**
- import from package **117**
- in FSM models **302**
- label scope **116**
- label sequences **116**
- methods **121–123**
- typed **116**
- variable base type **114**
- vs. parameter constants **303**
- escaped names **49**
- Espresso logic minimization **330**
- evaluate **417**
- event data type **70**
- event regions **27**
 - Active events **27**
 - NBA update events **27**
- event scheduling **26–30**
- events
 - Active **278–279**
 - NBA update **278–279**
 - regions **27**
 - scheduling order **280**
 - simulation **26, 278**
- examples
 - obtaining copies of **xxx**
 - simulators used **xxx**
 - synthesis compilers used **xxx, 33**
- exclusive-NOR **155**
- exclusive-OR **155**
- explicit
 - continuous assignment **254**
 - enumerated types **115**
 - package item import **105**
 - package item reference **107**
 - signedness casting **206–208**
 - size casting **202–205**
 - type casting **200–201**
- export, packages **103, 109**
- expression
 - definition of **142**
- expressions
 - real **145**
 - true or false **150, 160**
- extern modules **88**
- F**
- false or true expression rules **150, 160**
- Field Programmable Gate Array, see FPGA
- file name extension **48**
- Finite State Machines **266, 299–317**
- Mealy **301**
- Moore **301**
- next state decoder **305**
- one process style **308–309**
- output decoder **305**
- state encoding **302–305**
 - binary count **302**
 - Gray code **302**
 - highly encoded **308**
 - Johnson count **302**
 - one-hot **302, 328**
 - one-hot-0 **302**
- state sequencer **305**
- three process style **306–307**
- two process style **307–308**
- unintentional latches **328**
- first, enumerated type method **121**
- fixed-point, see real expressions
- Flake, Peter **6**
- flip-flops
 - asynchronous reset **288–289, 404**
 - asynchronous set/reset **291–292**
 - chip-enable **221, 290**
 - clock-to-Q delay **297**
 - data-enable, see chip-enable
 - hold time **295**
 - intra-assignment delay **297**
 - load-enable, see chip-enable
 - metastability **294, 295**
 - non-resettable **287**
 - power-up initial value **294**
 - reset recovery time **294**
 - reset synchronizer **294**
 - resets **286–295**
 - setup time **295**
 - synchronous reset **287–288**
 - synthesis rules **274**
 - vs. registers **274**
- floating-point, see real expressions
- for loop statements **228–233**
- foreach loop statements **236–238**
- fork-join statement group **19**
- formal arguments, tasks and functions **245**
- formal verification **14, 36**
- four-state, see 4-state
- FPGA **53, 78**
 - design flow **15–16**
- frequency, clock **35**
- FSM, see Finite State Machines
- full_case **43, 329, 350–351**
- fully implemented decision **330**
- function calls
 - pass-by-name **246**
 - pass-by-order **246**

functions

- actual arguments **245**
- as combinational logic **263**
- assignment statement **74**
- automatic **243–244**
- constant **95**
- default input values **246**
- formal arguments **245**
- in modules **243–248**
- in packages **103, 111**
- parameterized **247–248**
- return value **244**
- static **243–244, 247**
- void **245**

G

- Gate Array, see FPGA
- gate-level models **7–9**
- gate-level simulation **404**
- Gateway Design Automation **2**
- Gateway Design Automation. **1**
- GDSII **14**
- Genus RTL Compiler **xxx**
- glitch, set/reset flip-flop **291**
- global declarations **22**
- Golson, Steve **xxxi, 438, 439**
- Graphic Data System, see GDSII
- grave accent **52**
- Gray code state encoding **302**
- Gray, Frank **302**

H

- Hardware Design Language **1**
- HDL, see Hardware Design Language
- hexadecimal literal values **63**
- hierarchy **54–55**
- High-level Synthesis **185**
- highly-encoded state machine **308**
- HLS, see High-level Synthesis
- hold time **35, 295**

I

- identifiers (names) **49–51**

- IEEE **3**

- IEEE 1364 Verilog **1, 438**

- IEEE 1800 SystemVerilog **xxvii, xxxi, 40**

- if-else statements **216–222, 265, 404, 416**

- illegal characters **49**

- implicit

- continuous assignments **254**
- enumerated type **114**
- net type **80–82**
- signedness casting **206–208**
- size casting **202–205**

- type casting **200–201**

- import

- enumerated types **117**
- explicit package item **105**
- functions and tasks **243**
- import statement **104, 105**
- interface methods **372**
- multiple packages **108–109**
- package chaining **109**
- packages **103, 104–110**
- statement placement **106**
- wildcard **104**

- in-line initialization **75**

- incomplete decision branches **328**

- incomplete decision statements **327**

- increment operator **74**

- inferred nets, see implicit net type

- inferred port connections **57–59, 81**

- inherited port declarations **87**

- initial procedures **19, 39, 74, 76, 211, 322**

- inout

- function arguments **245**

- ports **53, 84, 252, 400**

- input

- argument default values **246**

- function arguments **245**

- port default values **87**

- ports **53, 84, 252**

- instance name **55**

- instance, of module **54–55**

- int data type **68, 69, 70, 426**

- integer data type **68, 70**

- integer value **62**

- Intellectual Property models (IP) **185**

- interconnect ports **87**

- interfaces **356–381**

- accessing signals **365**

- automatic functions **375**

- connecting **363–365**

- declarations **366**

- defining **361–366**

- import methods **372**

- interface keyword **356, 361**

- interface port **361**

- methods **372–376**

- modports **363, 367–372**

- module

- generic interface port **366**

- interface type-specific port **367**

- name space **50**

- parameterized **378**

- procedural code **376**

- referencing signal within **365**

- source code order **366**

synthesizing 375, 377, 379–381
virtual 70

intra-assignment delay 275, 297, 326

invert operator 153

invert vs. negate operator 161

IP, see Intellectual Property models

iterator variable 232

J

Johnson count state encoding 302

Johnson counter 279

Johnson counter, bad example 282

Johnson, Robert Royce 302

jump statements 238–241

K

Karnaugh mapping 330

keyword backward compatibility 46, 391

keywords 44–49, 391

L

labels

scope 116

sequences 116

Language Reference Manual *xxxii*

language rules 40

Larson, Kelly *xxxii*

last, enumerated type method 121

latch avoidance

coding style trade-offs 330–332

default assignment 335–337

full_case synthesis pragma 350–351

pre-case assignment 338–339

priority case 340–345

unique case 340–345

X value assignment 345–349

latches 220

D-type 323

intentional 323–326

sensitivity list 214

set/reset 323

transparent 323

unintentional 262, 263, 317, 327–351

least-significant bit 71

LEC, see Logic Equivalence Checker

left-extend 144

legal identifiers (names) 49

limitations

ASICs and FPGAs 53

simulators 23

synthesis compilers 32

lint checker *xxx*, 35–36, 63, 65, 253, 289, 325

literal values 62–66

literal values, real 66

little endian 71

load-enable flip-flop, see chip-enable flip-flop

local variable 215, 285

localparam 94, 103, 114, 124

logic data type 68, 70, 94, 252

Logic Equivalence Checker (LEC) 14, 36, 116, 305, 349

logic reduction 330

logical equivalence operator 164

logical implication operator 164

logical operators 160–161

longint data type 68, 69, 426

loop iterator variable 232

loops 228–238

 data-dependent 230

 static 230

 timed 231

 zero-delay 230

loosely typed 118, 304

LRM, SystemVerilog *xxxii*

LSB, see Least Significant Bit

LUT 15, 193

M

masking 224

Mealy state machines 301

Mealy, George H. 301

members, of structures 125

memory array 90, 318

memory devices, see RAM

Mentor Graphics *xxx*

metastability 294, 295

methods

 enumerated types 121–123

 first 121

 last 121

 name 121

 next 121

 num 121

 prev 121

 interfaces 372–376

Mills, Don *xxxii*, 438

mismatch

 connections 83–84

 continuous assignment size 253

 literal integers 64

modports 363, 367–372

module

 definition of 53

 hierarchy 54–55

 input port default values 87

 instance 54–55

 keyword 39

 name 53

- name space **50**
- output port initialization **87**
- port
 - default direction, type, size **86**
 - port declaration inheritance **87**
 - port list **84**
- modules **39–40, 52–54**
- Moorby, Phil **xxv, xxxi**
- Moore state machines **301**
- Moore, Edward F. **301**
- most-significant bit **71, 299**
- MSB, see most significant bit
- multidimensional array **90**
- multiple
 - clocks **295**
 - drivers **76, 78, 252**
 - packages **108–109**
 - sources **252**
- multiplexor **151, 219**
- mutually exclusive **266**

N

name space **49–50**
\$unit **50**
block **50**
class **50**
component **50**
definitions **49**
package **49**
name, enumerated type method **121**
named port connections **56–57**
named statement group **215**
names, see identifiers
naming convention **50–51, 102**
NAND
 reduction operator **158**
NBA Update event region **27, 278–279**
negate operator **160**
negate vs. invert operator **161**
negedge **213, 414**
nested modules **88**
net assignment rules **83–84**
netlist **32, 55**
nets **66–67, 76–84**
Network to Network Interface (NNI) **131**
new line **43**
next state decoder **305**
next, enumerated type method **121**
NNI, see Network to Network Interface
no-op statement **241**
non-resettable flip-flop **287**
nonblocking assignment **28–30, 278–281,**
297–298, 312
NOR reduction operator **158**

null operation, see no-op statement
num, enumerated type method **121**

O

octal literal values **63**
one process style state machine **308–309**
one-dimensional array **90**
one-hot
 decoder **266**
 encoding **328**
 state encoding **302**
one-hot-0 **302**
one-line comments **40**
one's complement **153, 161**
Open Verilog International **3**
operands **141**
operations
 2-state **142**
 4-state **142**
 context-determined operands **144**
 self-determined operands **144**
 signed **145**
 unsigned **145**
operator associativity **209**
operators
 \wedge bitwise XOR **153**
 \wedge reduction XOR **158**
 $\wedge=$ bitwise XOR assignment **196**
 $\wedge\sim$ bitwise XNOR **153**
 $\wedge\sim$ reduction XNOR **158**
 \neg decrement **189**
 $-=$ subtract assignment **196**
 \rightarrow logical implication **164**
 $::$ scope resolution **104, 107**
 $!$ logical negate **160, 161**
 $'()$ cast **198**
 $\{ \}$ **92**
 $*=$ multiply assignment **196**
 $/=$ divide assignment **196**
 $\&$ bitwise AND **144, 153**
 $\&$ reduction AND **158**
 $\&\&$ logical AND **144, 160**
 $\&=$ bitwise AND assignment **196**
 $\%=$ modulus assignment **196**
 ++ increment **74**
 += add assignment **196**
 <-> logical equivalence **164**
 $\ll<<$ arithmetic left shift assignment **196**
 $\ll<$ bitwise left shift assignment **196**
 $\gg=$ bitwise right shift assignment **196**
 $\gg>=$ arithmetic right shift assignment **196**
 \mid bitwise OR **153**
 \mid reduction OR **158**
 $\|=$ bitwise OR assignment **196**

- || logical OR **160**
- ~ bitwise invert **153, 161**
- ~^ bitwise XNOR **153**
- ~^ reduction XNOR **158**
- ~& reduction NAND **158**
- ~| reduction NOR **158**
- bitwise **153–157**
 - AND **154**
 - invert **153**
 - OR **154**
 - XNOR **155**
 - XOR **155**
- casting **198–208**
- concatenate { } **146–150**
- conditional ?: **150–152**
- expression rules **141–145**
- invert **153**
- logical **160–161**
 - AND **160**
 - equivalence **164**
 - implication **164**
 - OR **160**
- negate **160**
- negate vs. invert **161**
- precedence **209**
- reduction **158–159**
 - AND **158**
 - NAND **158**
 - NOR **158**
 - OR **158**
 - XNOR **158**
 - XOR **158**
- replicate {{ }} **146–150**
- short circuiting **163**
- X-optimistic **411–412**
- X-pessimistic **420–422**
- OR
 - bitwise operator **154**
 - logical operator **160**
 - reduction operator **158**
- output
 - function arguments **245**
 - port initialization **87**
 - ports **53, 84**
- output decoder **305**
- OVI, see Open Verilog International
- OVM **355**
- P**
- packages **102–112**
 - chaining **109–110**
 - compilation order **110–111**
 - defining **103–104**
- enumerated types **117**
- explicit item import **105**
- explicit item reference **107**
- export **109**
- import **243**
 - import multiple packages **108–109**
 - import placement **106**
 - import statement **109**
 - name space **49, 50**
 - package items **104**
 - referencing items **104–110**
 - tasks and functions **111**
 - wildcard import **104**
- packed
 - arrays **71–74**
 - structure **127–130**
 - union **132–133**
- parallel_case **270, 351**
- parameter **93–99, 103, 124**
 - declaration **94**
 - list **53, 94–96**
 - override **97**
 - redefinition **55, 97–98**
 - type **97**
- parameterized
 - class **247**
 - functions **247–248**
 - interfaces **378**
 - modules **93–94**
- part select **71–72, 92, 423**
- passing arrays through ports **93**
- passing structures through ports **129**
- passing unions through ports **134**
- pattern file **320**
- period, clock **35**
- Phil Moorby *xxv, xxxi*
- place and route **14**
- port
 - connections
 - by order **55–56**
 - dot-name shortcut **57–58**
 - dot-star shortcut **58–59**
 - inferred **57–59**
 - port name **56–57**
 - declaration **53, 84–89**
 - inheritance **87**
 - default direction, type, size **86**
 - default values **87**
 - expressions **88**
 - interconnect **87**
 - port order **55–56**
 - ref **87**
- posedge **213, 414**
- power-up values **75, 294**

pragmas 42–43, 350–352
 full_case 43, 329, 350–351
 parallel_case 270, 351
 translate_off 43, 292, 352, 376, 381
 translate_on 43, 292, 352, 376, 381
 pre-case assignment 338–339
 precedence of operators 209
 Precision RTL Synthesis xxx, 33
 prev, enumerated type method 121
 primitive name space 50
 primitives 7
 priority decision modifier 47, 227, 343–345, 351
 priority encoded logic 265–269
 priority encoder 220, 226, 265
 procedural assignments 83, 216
 procedural blocks 10, 19, 39, 211–216
 program name space 50
 programming statements
 ; no-op 241–242
 break 239
 case 222–227, 407, 419
 case vs. case...inside 224
 case...inside 223, 224
 casex 223, 224
 casez 223, 224
 continue 239
 data-dependent loops 230
 decision modifiers 227, 266–271, 316–317, 351
 decisions 216–228
 disable 240–241
 do-while loop 235–236
 for loop 228–233
 foreach loop 236–238
 if-else 216–222, 404, 416
 jump 238–241
 loops 228–238
 no-op 241
 priority decision modifier 47, 227, 343–345, 351
 repeat loop 233–234
 static loops 230
 timed loops 231
 unique decision modifier 227, 266–271, 316–317, 340–341, 351
 unique0 decision modifier 227, 266–271, 341–342
 while loop 235–236
 zero-delay loops 230
 propagation delay 25, 32, 35, 235, 243, 252, 257, 324
 ASIC cell 13
 primitive 9
 prototypes

interface task/function 374
 pull0 data type 77
 pull1 data type 77
 purpose of this book xxvii
Q
 Quality of Results 185, 269, 276, 291, 330, 339
 Quartus Prime xxx
 Questa simulator xxx
 question mark 65
 Quine-McCluskey logic minimization 330

R
 race condition 20, 285
 RAM 78, 317–322
 loading with \$readmemb/\$readmemh 320
 real
 data type 69
 expressions 145
 literal values 62, 66
 realtime data type 70
 recovery time 294
 reduction operators 158–159
 ref arguments 245
 ref ports 87
 referencing package items 104–110
 reg data type 68–69, 94, 124
 register retiming 185, 235
 Register Transfer Level 10–11
 register vs. flip-flop 274
 repeat loop statement 233–234
 replicate operator {{}} 146–150
 reserved keywords 44–49
 reset
 recovery time 294
 synchronizer 294
 resets
 flip-flops 286–295
 return
 exit a function 246
 from a function 244
 reverse apostrophe 52
 reverse case statements 265, 313–317
 RTL, see Register Transfer Level
 run-time constants 94
 Russian Peasant Multiplication 263

S
 scalar 10, 70
 scan chain insertion 14
 scope resolution operator :: 104, 107, 247
 self-determined operands 144
 semantics, definition 40

- sensitivity list *39, 212–214*
 @* *259*
 combinational logic *257, 258*
 sequential logic *275–276*
 X-optimism *414*
 X-pessimism *420*
sequential logic *213, 273–322*
 definition *273*
 synthesis rules *274*
Serial-to-Parallel Interface (SPI) *299, 309*
set all bits *65*
set/reset latches *323*
setup and hold violations *295*
setup time *35, 295*
short-circuiting *163*
shortint data type *68, 69, 426*
shortreal data type *69*
sign extension *64, 144, 253*
signed *71, 94, 128, 253*
 literal values *63*
 operations *145*
 structures *128*
simulation event regions *27*
simulation events *26, 278*
simulation race condition *20, 285*
simulation time *26–30*
simulator limitations *23*
simulators
 Questa *xxx*
 VCS *xxx*
single driver *78*
single source *74, 252, 253*
size
 context *145*
 expression *144*
 literal value mismatch *64*
 vector mismatch *253*
size mismatch *83–84*
sized literal values *63*
snippets, code *xxix*
SoC, see System on Chip
source code order *21*
sources of information *xxxii*
spaces *43*
Spear, Chris *xxvii, xxxi, 6*
SPI, see Serial-to-Parallel Interface
SpyGlass *xxx*
square brackets [] *71, 90*
SR latches, see set/reset latches
STA, see Static Timing Analysis
standard cells *13*
state encoding *302–305*
 binary count *302*
 Gray code *302*
 Johnson count *302*
one-hot *302*
one-hot-0 *302*
state machines, see Finite State Machines
state sequencer *305*
static functions *243–244, 247*
static loops *230*
static timing analysis *14, 35, 323*
strength level *76*
string data type *70*
strongly typed *118, 304*
struct *124*
structures
 '{ } list of values *125*
 anonymous *126, 127*
 assignment *125*
 copying *127*
 declaration and usage *124–130*
 default values *126*
 expressions *125*
 in arrays *136*
 members *125*
 packed *127–130*
 passing through ports *129*
 signed *128*
 synthesis considerations *130*
 typed *126, 127*
 unpacked *127–130*
 with arrays *136*
Stuart Sutherland *vii, 397, 435*
subfields *73*
supply0 data type *77*
supply1 data type *77*
Sutherland HDL, Inc. *vii*
sutherland-hdl.com *397, 438, 439*
Sutherland, LeeAnn *xxxii*
Sutherland, Stuart *vii, 6, 397, 435*
SVA, see SystemVerilog Assertions
switch C programming statement *223*
switch-level models *9*
synchronous reset flip-flop *287–288*
Synopsys *xxx, 2*
Synplify-Pro *xxx*
syntax, definition *40*
synthesis
 attribute *42*
compilers
 Cadence Genus RTL Compiler *xxx*
 Intel Quartus Prime *xxx*
 Mentor Precision RTL *xxx, 33*
 Synopsys DC-Ultra *xxx*
 Synopsys Synplify-Pro *xxx*
 Xilinx Vivado *xxx*
constraints *32, 34–35*
flow *31–35*

- interfaces 371, 375
- limitations 32
- pragmas 350–352
 - full_case 43, 329, 350–351
 - parallel_case 270, 351
 - translate_off 43, 352, 376, 381
 - translate_on 43, 352, 376, 381
- SystemVerilog subset 12
- Verilog subset 12
- synthesis compilers 32
- synthesis directives, see synthesis pragmas
- synthesizable net types 77
- synthesizable variable types 67
- system functions 51
 - \$dimensions 238
 - \$high 238
 - \$increment 238
 - \$left 238
 - \$right 238
 - \$size 238
- System on Chip (SoC) 12
- system tasks 51
- SystemVerilog *xxxi*, 5, 40, 48
 - Accellera 3.0 standard 4
 - Accellera 3.1 standard 4
 - history *xxv*, *xxvii*, 1–6
 - IEEE 1800-2005 standard 46
 - IEEE 1800-2009 standard 46
 - IEEE 1800-2012 standard 46
 - IEEE 1800-2017 standard 46
- SystemVerilog Assertions (SVA) 36, 431–432, 438
- SystemVerilog for Verification book *xxvii*, *xxxi*, 6, 20, 437

- T**
- tabs 43
- tagged unions 132
- taping out 14
- tasks
 - assignment statement 74
 - in modules 248
 - in packages 103, 111
- technology library 32, 35
- temporary variables 283
- ternary operator, see conditional operator
- testbench 18
- this book
 - chapter overview *xxviii*
 - code examples *xxix*
 - code snippets *xxix*
 - example convention *xxix*
 - intended audience *xxviii*
 - obtaining examples *xxx*
- purpose *xxvii*
- three process style state machine 306–307
- tick (apostrophe) 63
- time
 - data type 70
 - precision 24–25
 - units 24–25, 103
- timed loops 231
- timeprecision keyword 24
- timeunit keyword 24
- timing analysis 14, 35
- top-level module 20
- transaction-level models 11
- translate_off 43, 292, 352, 376, 381
- translate_on 43, 292, 352, 376, 381
- transparent latches 323
- tri data type 77, 152, 252
- tri-state buffers 152
- triand data type 77
- trior data type 77
- trireg data type 77
- true or false expression rules 150, 160
- Tumbush, Greg *xxvii*, 6
- two process style state machine 307–308
- two-state, see 2-state
- type conversion, see casting operators
- type parameters 97
- typed
 - enumerated types 116
 - structures 126, 127
 - unions 131
- typedef 101, 103, 126

- U**
- UDP 7
- underscore 65
- UNI, see User Network Interface
- uninitialized value 74
- unintentional latches 262, 263, 317, 327–351
- unions
 - anonymous 131
 - assigning 132
 - declaration and usage 131–135
 - in arrays 136
 - packed 132–133
 - passing through ports 134
 - reading 132
 - tagged 132
 - typed 131
 - unpacked 132
 - with arrays 136
- unique decision modifier 227, 266–271, 316–317, 340–341, 351
- unique0 decision modifier 227, 266–271,

341–342, 351

unit delay 275, 297, 326

unpacked

arrays 89, 89–93

structure 127–130

union 132

unsigned 71, 94, 128, 149, 253

literal values 63

operations 145

unsized literal values 63

User Defined Primitives 7

User Network Interface (UNI) 131

user-defined types 94, 101–102

UVM 355

uwire data type 77, 79

V

var data type 67, 70, 252

when to use 70

variable part select 72

variables 66–76

assignment rules 74–76

initialization 254

local 215, 283, 285

loop iterator 232

non-synthesizable types 69

synthesizable types 67

temporary 283

with subfields 73

VCS xxx

vector 10, 71

vector extension 144

vector size 144

verification methodologies 355

Verilog 48, 438

2001 standard 3, 32, 46, 207

2005 standard 5, 46

95 standard 3, 46

history xxv, 1–6

Verilog-XL simulator 2

Verilog++ 4

VHDL 3, 32, 439

virtual class 247

virtual interface data type 70

Vivado xxx

VMM 355

void functions 245

W

wait time control 115, 257, 260, 275, 324, 325

wand data type 77

while loop statements 235–236

white space 43

wildcard

case equality operator ==? 224

case statements 224

package import 104

wire data type 77–80, 252

wor data type 77

X

X value 61, 69

X value assignment 345–349

X-extend 144

X-optimism 142–143, 397, 403–415

arrays, writing 412

case statement 407–408

case...inside statement 408–411

casex statement 408–411

casez statement 408–411

if-else 404–407

net types 413

operators 411–412

primitives 412

sensitivity lists 414

X-pessimism 142–143, 397, 415–424

arrays, reading 423

bit select 423

case statement 419

conditional operator 417

if-else 416

operators 420–422

part select 423

primitives 422

sensitivity lists 420

shift operators 423

X-propagation 428

Xilinx CoolRunner CPLD 194, 288

Xilinx Virtex FPGA 193, 287

XNOR

bitwise operator 155

reduction operator 158

XOR

bitwise operator 155

reduction operator 158

Z

Z value 61, 65, 69

Z-extend 144

zero-delay loop 192, 230

zero-delay model 54, 252, 257, 260, 262, 275, 324, 326, 356, 375

zero-extend 144

"Many published textbooks on the design side of SystemVerilog assume that the reader is familiar with Verilog, and simply explain the new extensions. It is time to leave behind the stepping-stones and to teach a single consistent and concise language in a single book, and maybe not even refer to the old ways at all!"

"If you are a designer of 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 learn the design aspects of SystemVerilog."

— Phil Moorby, creator of Verilog (excerpt from the foreword)

This book is both a **tutorial** and a **reference** for engineers who use the SystemVerilog Hardware Description Language (HDL) to design ASICs and FPGAs. The book shows how to write SystemVerilog models at the Register Transfer Level (RTL) that simulate and synthesize correctly, using proper coding styles and best practices.

SystemVerilog is the latest generation of the original Verilog language, and adds many important capabilities to efficiently and more accurately model increasingly complex designs. This book reflects the SystemVerilog-2012/2017 standard.

This book is for engineers who already know, or who are learning, digital design engineering, and who want to master using SystemVerilog in digital design. The book does not present digital design theory; it shows how to apply that theory to write RTL models that simulate and synthesize correctly.

Stuart Sutherland is an industry expert in SystemVerilog. He has been using Verilog and SystemVerilog in design and verification for more than 25 years, and is the founder of Sutherland HDL, Inc., provider of SystemVerilog training and consulting services. Mr. Sutherland has published several papers and books on Verilog and SystemVerilog.

published by:

SUTHERLAND
training engineers to be **HDL**
SystemVerilog and UVM Wizards
sutherland-hdl.com

ISBN 9781546776345



9 781546 776345